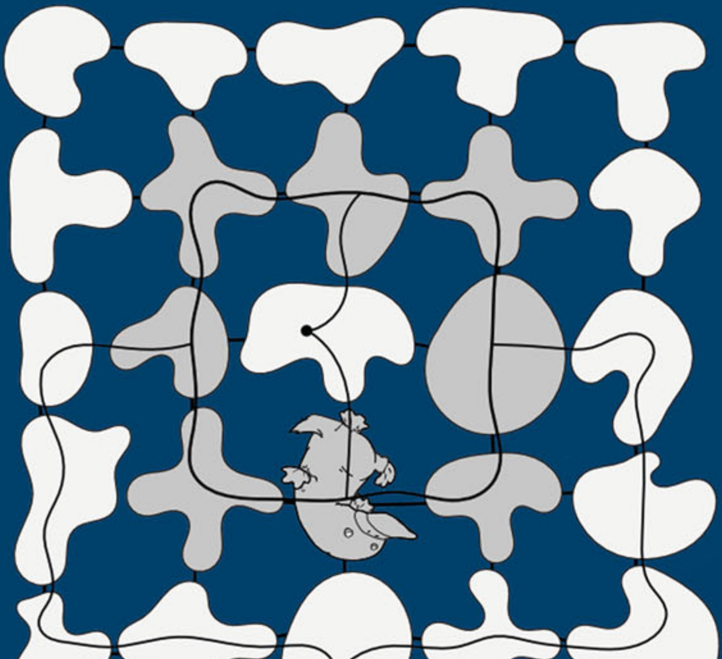
Marek Cygan · Fedor V. Fomin
Łukasz Kowalik · Daniel Lokshtanov
Dániel Marx · Marcin Pilipczuk
Michał Pilipczuk · Saket Saurabh

Parameterized Algorithms



Springer

Parameterized Algorithms

Marek Cygan · Fedor V. Fomin
Łukasz Kowalik · Daniel Lokshtanov
Dániel Marx · Marcin Pilipczuk
Michał Pilipczuk · Saket Saurabh

Parameterized Algorithms

Springer

Marek Cygan
Institute of Informatics
University of Warsaw
Warsaw
Poland

Fedor V. Fomin
Department of Informatics
University of Bergen
Bergen
Norway

Łukasz Kowalik
Institute of Informatics
University of Warsaw
Warsaw
Poland

Daniel Lokshtanov
Department of Informatics
University of Bergen
Bergen
Norway

Dániel Marx
Institute for Computer Science and Control
Hungarian Academy of Sciences
Budapest
Hungary

Marcin Pilipczuk
Institute of Informatics
University of Warsaw
Warsaw
Poland

Michał Pilipczuk
Institute of Informatics
University of Warsaw
Warsaw
Poland

Saket Saurabh
C.I.T. Campus
The Institute of Mathematical Sciences
Chennai
India

ISBN 978-3-319-21274-6 ISBN 978-3-319-21275-3 (eBook)
DOI 10.1007/978-3-319-21275-3

Library of Congress Control Number: 2015946081

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media
(www.springer.com)

Preface

The goal of this textbook is twofold. First, the book serves as an introduction to the field of parameterized algorithms and complexity accessible to graduate students and advanced undergraduate students. Second, it contains a clean and coherent account of some of the most recent tools and techniques in the area.

Parameterized algorithmics analyzes running time in finer detail than classical complexity theory: instead of expressing the running time as a function of the input size only, dependence on one or more parameters of the input instance is taken into account. While there were examples of nontrivial parameterized algorithms in the literature, such as Lenstra's algorithm for integer linear programming [319] or the disjoint paths algorithm of Robertson and Seymour [402], it was only in the late 1980s that Downey and Fellows [149], building on joint work with Langston [180, 182, 183], proposed the systematic exploration of parameterized algorithms. Downey and Fellows laid the foundations of a fruitful and deep theory, suitable for reasoning about the complexity of parameterized algorithms. Their early work demonstrated that fixed-parameter tractability is a ubiquitous phenomenon, naturally arising in various contexts and applications. The parameterized view on algorithms has led to a theory that is both mathematically beautiful and practically applicable. During the 30 years of its existence, the area has transformed into a mainstream topic of theoretical computer science. A great number of new results have been achieved, a wide array of techniques have been created, and several open problems have been solved. At the time of writing, Google Scholar gives more than 4000 papers containing the term "fixed-parameter tractable". While a full overview of the field in a single volume is no longer possible, our goal is to present a selection of topics at the core of the field, providing a key for understanding the developments in the area.

Why This Book?

The idea of writing this book arose after we decided to organize a summer school on parameterized algorithms and complexity in Będlewo in August 2014. While planning the school, we realized that there is no textbook that contains the material that we wanted to cover. The classical book of Downey and Fellows [153] summarizes the state of the field as of 1999. This book was the starting point of a new wave of research in the area, which is obviously not covered by this classical text. The area has been developing at such a fast rate that even the two books that appeared in 2006, by Flum and Grohe [189] and Niedermeier [376], do not contain some of the new tools and techniques that we feel need to be taught in a modern introductory course. Examples include the lower bound techniques developed for kernelization in 2008, methods introduced for faster dynamic programming on tree decompositions (starting with *Cut & Count* in 2011), and the use of algebraic tools for problems such as Longest Path. The book of Flum and Grohe [189] focuses to a large extent on complexity aspects of parameterized algorithmics from the viewpoint of logic, while the material we wanted to cover in the school is primarily algorithmic, viewing complexity as a tool for proving that certain kinds of algorithms do not exist. The book of Niedermeier [376] gives a gentle introduction to the field and some of the basic algorithmic techniques. In 2013, Downey and Fellows [154] published the second edition of their classical text, capturing the development of the field from its nascent stages to the most recent results. However, the book does not treat in detail many of the algorithmic results we wanted to teach, such as how one can apply important separators for Edge Multiway Cut and Directed Feedback Vertex Set, linear programming for Almost 2-SAT, *Cut & Count* and its deterministic counterparts to obtain faster algorithms on tree decompositions, algorithms based on representative families of matroids, kernels for Feedback Vertex Set, and some of the reductions related to the use of the Strong Exponential Time Hypothesis.

Our initial idea was to prepare a collection of lecture notes for the school, but we realized soon that a coherent textbook covering all basic topics in equal depth would better serve our purposes, as well as the purposes of those colleagues who would teach a semester course in the future. We have organized the material into chapters according to techniques. Each chapter discusses a certain algorithmic paradigm or lower bound methodology. This means that the same algorithmic problem may be revisited in more than one chapter, demonstrating how different techniques can be applied to it. Thanks to the rapid growth of the field, it is now nearly impossible to cover every relevant result in a single textbook. Therefore, we had to carefully select what to present at the school and include in the book. Our goal was to include a self-contained and teachable exposition of what we believe are the basic techniques of the field, at the expense of giving a complete survey of the area. A consequence of this is that we do not always present the strongest result for a particular problem. Nevertheless, we would like to point out that for many problems the book actually contains the state of the art and brings the reader to the frontiers of research. We made an effort to present full proofs for most of the results, where this was feasible within the textbook format. We used the opportunity of writing this textbook to revisit some of the results in the literature and, using the benefit of hindsight, to present them in a modern and didactic way.

At the end of each chapter we provide sections with exercises, hints to exercises and bibliographical notes. Many of the exercises complement the main narrative and cover important results which have to be omitted due to space constraints. We use (✎) and (☠) to identify easy and challenging exercises. Following the common practice for textbooks, we try to minimize the occurrence of bibliographical and historical references in the main text by moving them to bibliographic notes. These notes can also guide the reader on to further reading.

Organization of the Book

The book is organized into three parts. The first seven chapters give the basic toolbox of parameterized algorithms, which, in our opinion, every course on the subject should cover. The second part, consisting of Chapters 8-12, covers more advanced algorithmic techniques that are featured prominently in current research, such as important separators and algebraic methods. The third part introduces the reader to the theory of lower bounds: the intractability theory of parameterized complexity, lower bounds based on the Exponential Time Hypothesis, and lower bounds on kernels. We adopt a very pragmatic viewpoint in these chapters: our goal is to help the algorithm designer by providing evidence that certain algorithms are unlikely to exist, without entering into complexity theory in deeper detail. Every chapter is accompanied by exercises, with hints for most of them. Bibliographic notes point to the original publications, as well as to related work.

- Chapter 1 motivates parameterized algorithms and the notion of fixed-parameter tractability with some simple examples. Formal definitions of the main concepts are introduced.
- Kernelization is the first algorithmic paradigm for fixed-parameter tractability that we discuss. Chapter 2 gives an introduction to this technique.
- Branching and bounded-depth search trees are the topic of Chapter 3. We discuss both basic examples and more advanced applications based on linear programming relaxations, showing the fixed-parameter tractability of, e.g., Odd Cycle Transversal and Almost 2-SAT.
- Iterative compression is a very useful technique for deletion problems. Chapter 4 introduces the technique through three examples, including Feedback Vertex Set and Odd Cycle Transversal.

- Chapter 5 discusses techniques for parameterized algorithms that use randomization. The classic color coding technique for Longest Path will serve as an illustrative example.
- Chapter 6 presents a collection of techniques that belong to the basic toolbox of parameterized algorithms: dynamic programming over subsets, integer linear programming (ILP), and the use of well-quasi-ordering results from graph minors theory.
- Chapter 7 introduces treewidth, which is a graph measure that has important applications for parameterized algorithms. We discuss how to use dynamic programming and Courcelle's theorem to solve problems on graphs of bounded treewidth and how these algorithms are used more generally, for example, in the context of bidimensionality for planar graphs.
- Chapter 8 presents results that are based on a combinatorial bound on the number of so-called "important separators". We use this bound to show the fixed-parameter tractability of problems such as Edge Multicut and Directed Feedback Vertex Set. We also discuss randomized sampling of important cuts.
- The kernels presented in Chapter 9 form a representative sample of more advanced kernelization techniques. They demonstrate how the use of min-max results from graph theory, the probabilistic method, and the properties of planar graphs can be exploited in kernelization.
- Two different types of algebraic techniques are discussed in Chapter 10: algorithms based on the inclusion–exclusion principle and on polynomial identity testing. We use these techniques to present the fastest known parameterized algorithms for Steiner Tree and Longest Path.
- In Chapter 11, we return to dynamic programming algorithms on graphs of bounded treewidth. This chapter presents three methods (subset convolution, *Cut & Count,* and a rank-based approach) for speeding up dynamic programming on tree decompositions.
- The notion of matroids is a fundamental concept in combinatorics and optimization. Recently, matroids have also been used for kernelization and parameterized algorithms. Chapter 12 gives a gentle introduction to some of these developments.
- Chapter 13 presents tools that allow us to give evidence that certain problems are not fixed-parameter tractable. The chapter introduces parameterized reductions and the W-hierarchy, and gives a sample of hardness results for various concrete problems.
- Chapter 14 uses the (Strong) Exponential Time Hypothesis to give running time lower bounds that are more refined than the bounds in Chapter 13. In many cases, these stronger complexity assumptions allow us to obtain lower bounds essentially matching the best known algorithms.
- Chapter 15 gives the tools for showing lower bounds for kernelization algorithms. We use methods of composition and polynomial-parameter transformations to show that certain problem, such as Longest Path, do not admit polynomial kernels.

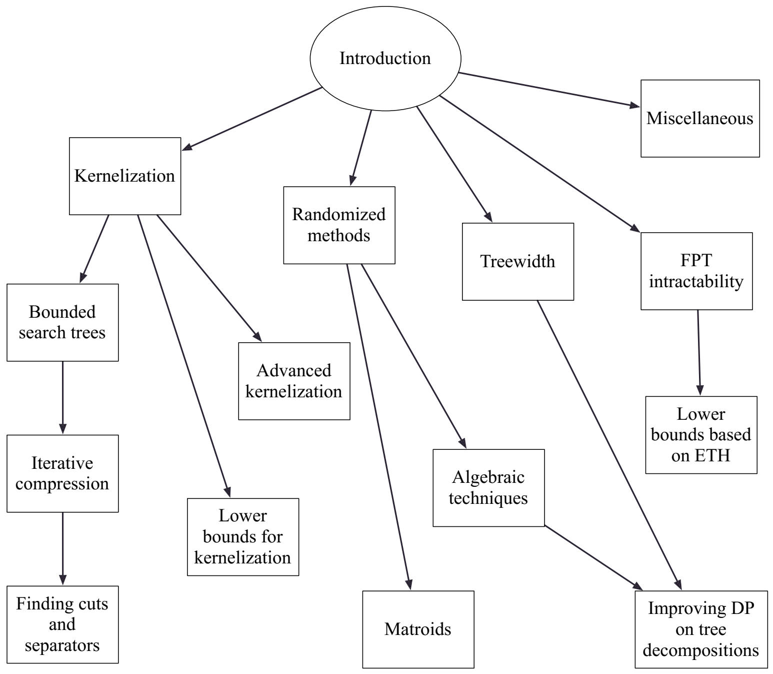


Fig. 0.1: Dependencies between the chapters

As in any textbook we will assume that the reader is familiar with the content of one chapter before moving to the next. On the other hand, for most chapters it is not necessary for the reader to have read *all* preceeding chapters. Thus the book does not have to be read linearly from beginning to end. Figure 0.1 depicts the dependencies between the different chapters. For example, the chapters on Iterative Compression and Bounded Search Trees are considered necessary prerequisites to understand the chapter on finding cuts and separators.

Using the Book for Teaching

A course on parameterized algorithms should cover most of the material in Part I, except perhaps the more advanced sections marked with an asterisk. In Part II, the instructor may choose which chapters and which sections to teach based on his or her preferences. Our suggestion for a coherent set of topics from Part II is the following:

- All of Chapter 8, as it is relatively easily teachable. The sections of this chapter are based on each other and hence should be taught in this order, except that perhaps Section 8.4 and Sections 8.5–8.6 are interchangeable.
- Chapter 9 contains four independent sections. One could select Section 9.1 (Feedback Vertex Set) and Section 9.3 (Connected Vertex Cover on planar graphs) in a first course.
- From Chapter 10, we suggest presenting Section 10.1 (inclusion–exclusion principle), and Section 10.4.1 (Longest Path in time $2^k \cdot n^{\mathcal{O}(1)}$).
- From Chapter 11, we recommend teaching Sections 11.2.1 and *11.2.2, as they are most illustrative for the recent developments on algorithms on tree decompositions.
- From Chapter 12 we recommend teaching Section 12.3. If the students are unfamiliar with matroids, Section 12.1 provides a brief introduction to the topic.

Part III gives a self-contained exposition of the lower bound machinery. In this part, the sections not marked with an asterisk give a set of topics that can form the complexity part of a course on parameterized algorithms. In some cases, we have presented multiple reductions showcasing the same kind of lower bounds; the instructor can choose from these examples according to the needs of the course. Section 14.4.1 contains some more involved proofs, but one can give a coherent overview of this section even while omitting most of the proofs.

Bergen, Budapest, Chennai, Warsaw
June 2015
Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk and Saket Saurabh

Acknowledgements

Many of our colleagues helped with valuable comments, corrections and suggestions. We are grateful for feedback to Yixin Cao, Chandra Chekuri, Kyril Elagin, Mike Fellows, Tobias Friedrich, Serge Gaspers, Petr Golovach, Fabrizio Grandoni, Jiong Guo, Gregory Gutin, Danny Hermelin, Pim van ’t Hof, Philip Klein, Sudeshna Kolay, Alexander Kulikov, Michael Lampis, Neeldhara Misra, Jesper Nederlof, Sang-il Oum, Fahad Panolan, Venkatesh Raman, M.S. Ramanujan, Lajos Rónyai, Dimitrios M. Thilikos, and Magnus Wahlström. We would like to especially thank Bart M.P. Jansen for his enormous number of very insightful comments, as well as his help in organizing the FPT School in Będlewo.

We are greatly indebted to Felix Reidl, the creator of the squirrel, the platypus, the hamster, and the woodpecker, for the magnificent illustrations embellishing this book.

Our work has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreements No. 267959 (Rigorous Theory of Preprocessing), 306992 (Parameterized Approximation), and 280152 (PARAMTIGHT: Parameterized complexity and the search for tight complexity results), from the Bergen Research Foundation and the University of Bergen through project “BeHard”, as well as from Polish National Science Centre grant UMO-2012/05/D/ST6/03214.

An important part of this work was done while subsets of the authors attended Dagstuhl Seminar 14071 “Graph Modification Problems” and ICERM Research Cluster “Towards Efficient Algorithms Exploiting Graph Structure”.

We also acknowledge the great help we have received from the staff of the Conference Center of the Institute of Mathematics of the Polish Academy of Science when organizing our school in Będlewo.

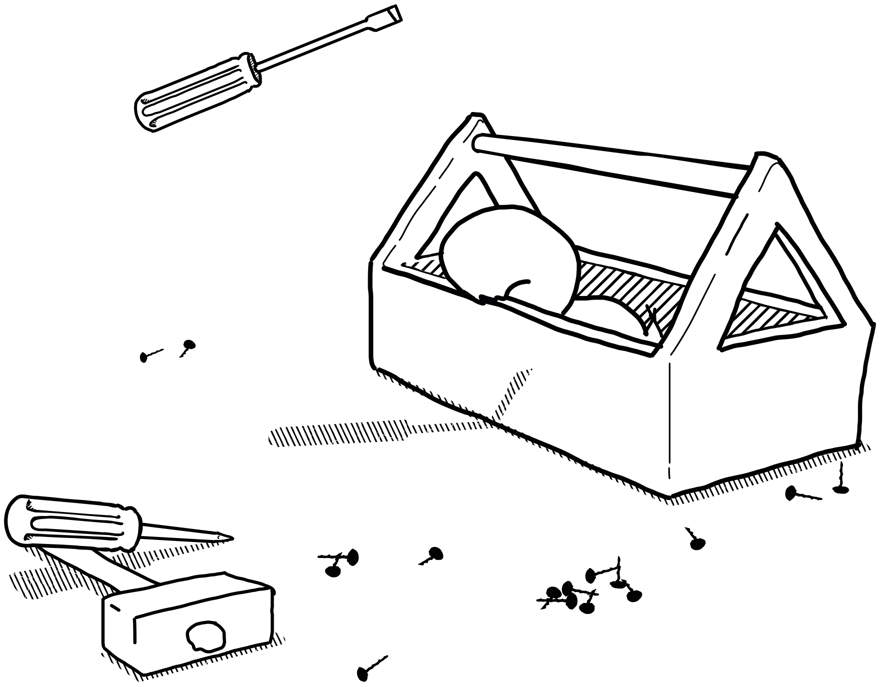
Saket would like to thank his mentor Venkatesh Raman for introducing him to this area and bearing with him for the last fourteen years. To him, he says “Thank you, sir!”

Daniel, Fedor, and Saket express their deepest gratitude to Mike Fellows, for teaching them parameterized complexity and being generous in sharing new ideas.

Last but not least, we would like to thank our families and other people who supported us during the writing of this book, including among others: Agata, Olga, Ania, Szymon, Monika, Zosia, Piotruś, Sushmita, Rebecca, Tanya, Nikolas, Shurik, Gosha, Zsuzsa, Veronika, and Zsolt.

Contents

Part I
Basic toolbox



Chapter 1
Introduction

A squirrel, a platypus and a hamster walk into a bar...

Imagine that you are an exceptionally tech-savvy security guard of a bar in an undisclosed small town on the west coast of Norway. Every Friday, half of the inhabitants of the town go out, and the bar you work at is well known for its nightly brawls. This of course results in an excessive amount of work for you; having to throw out intoxicated guests is tedious and rather unpleasant labor. Thus you decide to take preemptive measures. As the town is small, you know everyone in it, and you also know who will be likely to fight with whom if they are admitted to the bar. So you wish to plan ahead, and only admit people if they will not be fighting with anyone else at the bar. At the same time, the management wants to maximize profit and is not too happy if you on any given night reject more than k people at the door. Thus, you are left with the following optimization problem. You have a list of all of the n people who will come to the bar, and for each pair of people a prediction of whether or not they will fight if they both are admitted. You need to figure out whether it is possible to admit everyone except for at most k troublemakers, such that no fight breaks out among the admitted guests. Let us call this problem the Bar Fight Prevention problem. Figure 1.1 shows an instance of the problem and a solution for $k = 3$. One can easily check that this instance has no solution with $k = 2$.

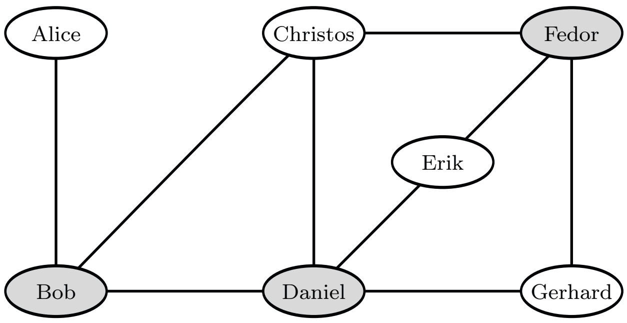


Fig. 1.1: An instance of the BAR FIGHT PREVENTION problem with a solution for $k = 3$. An edge between two guests means that they will fight if both are admitted

Efficient algorithms for BAR FIGHT PREVENTION

Unfortunately, BAR FIGHT PREVENTION is a classic NP-complete problem (the reader might have heard of it under the name VERTEX COVER), and so the best way to solve the problem is by trying all possibilities, right? If there are $n = 1000$ people planning to come to the bar, then you can quickly code up the brute-force solution that tries each of the $2^{1000} \approx 1.07 \cdot 10^{301}$ possibilities. Sadly, this program won't terminate before the guests arrive, probably not even before the universe implodes on itself. Luckily, the number k of guests that should be rejected is not that large, $k \leq 10$. So now the program only needs to try $\binom{1000}{10} \approx 2.63 \cdot 10^{23}$ possibilities. This is much better, but still quite infeasible to do in one day, even with access to supercomputers.

So should you give up at this point, and resign yourself to throwing guests out after the fights break out? Well, at least you can easily identify some peaceful souls to accept, and some troublemakers you need to refuse at the door for sure. Anyone who does not have a potential conflict with anyone else can be safely moved to the list of people to accept. On the other hand, if some guy will fight with at least $k + 1$ other guests you have to reject him — as otherwise you will have to reject all of his $k+1$ opponents, thereby upsetting the management. If you identify such a troublemaker (in the example of Fig. 1.1, Daniel is such a troublemaker), you immediately strike him from the guest list, and decrease the number k of people you can reject by one.[1]

If there is no one left to strike out in this manner, then we know that each guest will fight with at most k other guests. Thus, rejecting any single guest will resolve at most k potential conflicts. And so, if there are more than k^2

[1] The astute reader may observe that in Fig. 1.1, after eliminating Daniel and setting $k = 2$, Fedor still has three opponents, making it possible to eliminate him and set $k = 1$. Then Bob, who is in conflict with Alice and Christos, can be eliminated, resolving all conflicts.

potential conflicts, you know that there is no way to ensure a peaceful night at the bar by rejecting only k guests at the door. As each guest who has not yet been moved to the accept or reject list participates in at least one and at most k potential conflicts, and there are at most k^2 potential conflicts, there are at most $2k^2$ guests whose fate is yet undecided. Trying all possibilities for these will need approximately $\binom{2k^2}{k} \leq \binom{200}{10} \approx 2.24 \cdot 10^{16}$ checks, which is feasible to do in less than a day on a modern supercomputer, but quite hopeless on a laptop.

If it is safe to admit anyone who does not participate in any potential conflict, what about those who participate in exactly one? If Alice has a conflict with Bob, but with no one else, then it is always a good idea to admit Alice. Indeed, you cannot accept both Alice and Bob, and admitting Alice cannot be any worse than admitting Bob: if Bob is in the bar, then Alice has to be rejected for sure and potentially some other guests as well. Therefore, it is safe to accept Alice, reject Bob, and decrease k by one in this case. This way, you can always decide the fate of any guest with only one potential conflict. At this point, each guest you have not yet moved to the accept or reject list participates in at least two and at most k potential conflicts. It is easy to see that with this assumption, having at most k^2 unresolved conflicts implies that there are only at most k^2 guests whose fate is yet undecided, instead of the previous upper bound of $2k^2$. Trying all possibilities for which of those to refuse at the door requires $\binom{k^2}{k} \leq \binom{100}{10} \approx 1.73 \cdot 10^{13}$ checks. With a clever implementation, this takes less than half a day on a laptop, so if you start the program in the morning you'll know who to refuse at the door by the time the bar opens. Therefore, instead of using brute force to go through an enormous search space, we used simple observations to reduce the search space to a manageable size. This algorithmic technique, using reduction rules to decrease the size of the instance, is called *kernelization,* and will be the subject of Chapter 2 (with some more advanced examples appearing in Chapter 9).

It turns out that a simple observation yields an even faster algorithm for Bar Fight Prevention. The crucial point is that every conflict has to be resolved, and that the only way to resolve a conflict is to refuse at least one of the two participants. Thus, as long as there is at least one unresolved conflict, say between Alice and Bob, we proceed as follows. Try moving Alice to the reject list and run the algorithm recursively to check whether the remaining conflicts can be resolved by rejecting at most $k-1$ guests. If this succeeds you already have a solution. If it fails, then move Alice back onto the undecided list, move Bob to the reject list and run the algorithm recursively to check whether the remaining conflicts can be resolved by rejecting at most $k-1$ additional guests (see Fig. 1.2). If this recursive call also fails to find a solution, then you can be sure that there is no way to avoid a fight by rejecting at most k guests.

What is the running time of this algorithm? All it does is to check whether all conflicts have been resolved, and if not, it makes two recursive calls. In

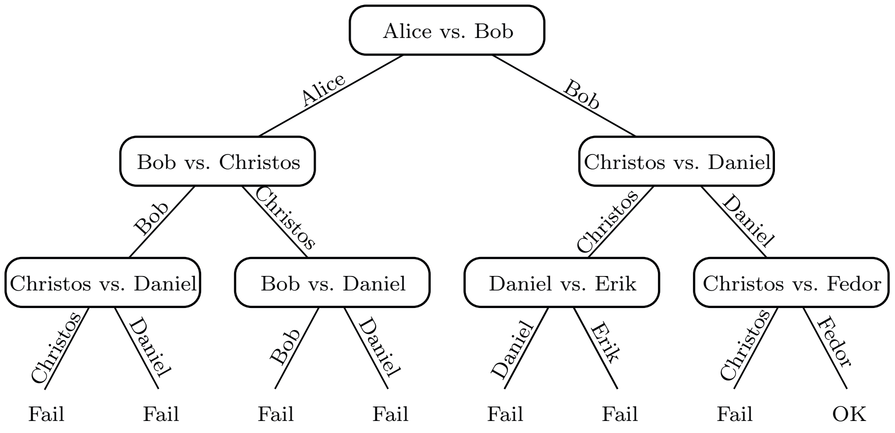


Fig. 1.2: The search tree for Bar Fight Prevention with $k = 3$. In the leaves marked with "Fail", the parameter k is decreased to zero, but there are still unresolved conflicts. The rightmost branch of the search tree finds a solution: after rejecting Bob, Daniel, and Fedor, no more conflicts remain

both of the recursive calls the value of k decreases by 1, and when k reaches 0 all the algorithm has to do is to check whether there are any unresolved conflicts left. Hence there is a total of 2^k recursive calls, and it is easy to implement each recursive call to run in linear time $\mathcal{O}(n + m)$, where m is the total number of possible conflicts. Let us recall that we already achieved the situation where every undecided guest has at most k conflicts with other guests, so $m \leq nk/2$. Hence the total number of operations is approximately $2^k \cdot n \cdot k \leq 2^{10} \cdot 10{,}000 = 10{,}240{,}000$, which takes a fraction of a second on today's laptops. Or cell phones, for that matter. You can now make the Bar Fight Prevention app, and celebrate with a root beer. This simple algorithm is an example of another algorithmic paradigm: the technique of *bounded search trees*. In Chapter 3, we will see several applications of this technique to various problems.

The algorithm above runs in time $\mathcal{O}(2^k \cdot k \cdot n)$, while the naive algorithm that tries every possible subset of k people to reject runs in time $\mathcal{O}(n^k)$. Observe that if k is considered to be a constant (say $k = 10$), then both algorithms run in polynomial time. However, as we have seen, there is a quite dramatic difference between the running times of the two algorithms. The reason is that even though the naive algorithm is a polynomial-time algorithm for every fixed value of k, the exponent of the polynomial depends on k. On the other hand, the final algorithm we designed runs in linear time for every fixed value of k! This difference is what parameterized algorithms and complexity is all about. In the $\mathcal{O}(2^k \cdot k \cdot n)$-time algorithm, the combinatorial explosion is restricted to the parameter k: the running time is exponential in k, but depends only polynomially (actually, linearly) on n. Our goal is to find algorithms of this form.

Algorithms with running time $f(k) \cdot n^c$, for a constant c independent of both n and k, are called *fixed-parameter algorithms*, or FPT algorithms. Typically the goal in parameterized algorithmics is to design FPT algorithms, trying to make both the $f(k)$ factor and the constant c in the bound on the running time as small as possible. FPT algorithms can be put in contrast with less efficient XP algorithms (for *slice-wise polynomial*), where the running time is of the form $f(k) \cdot n^{g(k)}$, for some functions f, g. There is a tremendous difference in the running times $f(k) \cdot n^{g(k)}$ and $f(k) \cdot n^c$.

In parameterized algorithmics, k is simply a *relevant secondary measurement* that encapsulates some aspect of the input instance, be it the size of the solution sought after, or a number describing how "structured" the input instance is.

A negative example: vertex coloring

Not every choice for what k measures leads to FPT algorithms. Let us have a look at an example where it does not. Suppose the management of the hypothetical bar you work at doesn't want to refuse anyone at the door, but still doesn't want any fights. To achieve this, they buy $k-1$ more bars across the street, and come up with the following brilliant plan. Every night they will compile a list of the guests coming, and a list of potential conflicts. Then you are to split the guest list into k groups, such that no two guests with a potential conflict between them end up in the same group. Then each of the groups can be sent to one bar, keeping everyone happy. For example, in Fig. 1.1, we may put Alice and Christos in the first bar, Bob, Erik, and Gerhard in the second bar, and Daniel and Fedor in the third bar.

We model this problem as a graph problem, representing each person as a vertex, and each conflict as an edge between two vertices. A partition of the guest list into k groups can be represented by a function that assigns to each vertex an integer between 1 and k. The objective is to find such a function that, for every edge, assigns different numbers to its two endpoints. A function that satisfies these constraints is called a *proper k-coloring* of the graph. Not every graph has a proper k-coloring. For example, if there are $k+1$ vertices with an edge between every pair of them, then each of these vertices needs to be assigned a unique integer. Hence such a graph does not have a proper k-coloring. This gives rise to a computational problem, called Vertex Coloring. Here we are given as input a graph G and an integer k, and we need to decide whether G has a proper k-coloring.

It is well known that Vertex Coloring is NP-complete, so we do not hope for a polynomial-time algorithm that works in all cases. However, it is fair to assume that the management does not want to own more than $k = 5$ bars on the same street, so we will gladly settle for a $\mathcal{O}(2^k \cdot n^c)$-time algorithm for some constant c, mimicking the success we had with our first problem. Unfortunately, deciding whether a graph G has a proper 5-coloring is NP-complete, so any $f(k) \cdot n^c$-time algorithm for Vertex Coloring for any function f and constant c would imply that P = NP ; indeed, suppose such an algorithm existed. Then, given a graph G, we can decide whether G has a proper 5-coloring in time $f(5) \cdot n^c = \mathcal{O}(n^c)$. But then we have a polynomial-time algorithm for an NP-hard problem, implying P = NP . Observe that even an XP algorithm with running time $f(k) \cdot n^{g(k)}$ for any functions f and g would imply that P = NP by an identical argument.

A hard parameterized problem: finding cliques

The example of Vertex Coloring illustrates that parameterized algorithms are not all-powerful: there are parameterized problems that do not seem to admit FPT algorithms. But very importantly, in this specific example, we could explain very precisely why we are not able to design efficient algorithms, even when the number of bars is small. From the perspective of an algorithm designer such insight is very useful; she can now stop wasting time trying to design efficient algorithms based only on the fact that the number of bars is small, and start searching for other ways to attack the problem instances. If we are trying to make a polynomial-time algorithm for a problem and failing, it is quite likely that this is because the problem is NP-hard. Is the theory of NP-hardness the right tool also for giving negative evidence for fixed-parameter tractability? In particular, if we are trying to make an $f(k) \cdot n^c$-time algorithm and fail to do so, is it because the problem is NP-hard for some fixed constant value of k, say $k = 100$? Let us look at another example problem.

Now that you have a program that helps you decide who to refuse at the door and who to admit, you are faced with a different problem. The people in the town you live in have friends who might get upset if their friend is refused at the door. You are quite skilled at martial arts, and you can handle at most $k - 1$ angry guys coming at you, but probably not k. What you are most worried about are groups of at least k people where everyone in the group is friends with everyone else. These groups tend to have an “all for one and one for all” mentality — if one of them gets mad at you, they all do. Small as the town is, you know exactly who is friends with whom, and you want to figure out whether there is a group of at least k people where everyone is friends with everyone else. You model this as a graph problem where every person is a vertex and two vertices are connected by an edge if the corresponding persons are friends. What you are looking for is a *clique* on k vertices, that is, a set of k vertices with an edge between every pair of them. This problem is known as the Clique problem. For example, if we interpret now the edges of Fig. 1.1 as showing friendships between people, then Bob, Christos, and Daniel form a clique of size 3.

There is a simple $\mathcal{O}(n^k)$-time algorithm to check whether a clique on at least k vertices exists; for each of the $\binom{n}{k} = \mathcal{O}(\frac{n^k}{k^2})$ subsets of vertices of size k, we check in time $\mathcal{O}(k^2)$ whether every pair of vertices in the subset is adjacent. Unfortunately, this XP algorithm is quite hopeless to run for $n = 1000$ and $k = 10$. Can we design an FPT algorithm for this problem? So far, no one has managed to find one. Could it be that this is because finding a k-clique is NP-hard for some fixed value of k? Suppose the problem was NP-hard for $k = 100$. We just gave an algorithm for finding a clique of size 100 in time $\mathcal{O}(n^{100})$, which is polynomial time. We would then have a polynomial-time algorithm for an NP-hard problem, implying that P = NP. So we cannot expect to be able to use NP-hardness in this way in order to rule out an FPT algorithm for Clique. More generally, it seems very difficult to use NP-hardness in order to explain why a problem that does have an XP algorithm does not admit an FPT algorithm.

Since NP-hardness is insufficient to differentiate between problems with $f(k) \cdot n^{g(k)}$-time algorithms and problems with $f(k) \cdot n^c$-time algorithms, we resort to stronger complexity theoretical assumptions. The theory of W[1]-hardness (see Chapter 13) allows us to prove (under certain complexity assumptions) that even though a problem is polynomial-time solvable for every fixed k, the parameter k has to appear in the exponent of n in the running time, that is, the problem is not FPT. This theory has been quite successful for identifying which parameterized problems are FPT and which are unlikely to be. Besides this qualitative classification of FPT versus W[1]-hard, more recent developments give us also (an often surprisingly tight) quantitative understanding of the time needed to solve a parameterized problem. Under reasonable assumptions about the hardness of CNF-SAT (see Chapter 14), it is possible to show that there is no $f(k) \cdot n^c$, or even a $f(k) \cdot n^{o(k)}$-time algorithm for finding a clique on k vertices. Thus, up to constant factors in the exponent, the naive $\mathcal{O}(n^k)$-time algorithm is optimal! Over the past few years, it has become a rule, rather than an exception, that whenever we are unable to significantly improve the running time of a parameterized algorithm, we are able to show that the existing algorithms are asymptotically optimal, under reasonable assumptions. For example, under the same assumptions that we used to rule out an $f(k) \cdot n^{o(k)}$-time algorithm for solving Clique, we can also rule out a $2^{o(k)} \cdot n^{\mathcal{O}(1)}$-time algorithm for the Bar Fight Prevention problem from the beginning of this chapter.

Any algorithmic theory is incomplete without an accompanying complexity theory that establishes intractability of certain problems. There

Problem	Good news	Bad news
Bar Fight Prevention	$\mathcal{O}(2^k \cdot k \cdot n)$-time algorithm	NP-hard (probably not in P)
Clique with Δ	$\mathcal{O}(2^{\Delta} \cdot \Delta^2 \cdot n)$-time algorithm	NP-hard (probably not in P)
Clique with k	$n^{\mathcal{O}(k)}$-time algorithm	W[1]-hard (probably not FPT)
Vertex Coloring		NP-hard for $k = 3$ (probably not XP)

Fig. 1.3: Overview of the problems in this chapter

is such a complexity theory providing lower bounds on the running time required to solve parameterized problems.

Finding cliques — with a different parameter

OK, so there probably is no algorithm for solving Clique with running time $f(k) \cdot n^{o(k)}$. But what about those scary groups of people that might come for you if you refuse the wrong person at the door? They do not care at all about the computational hardness of Clique, and neither do their fists. What can you do? Well, in Norway most people do not have too many friends. In fact, it is quite unheard of that someone has more than $\Delta = 20$ friends. That means that we are trying to find a k-clique in a graph of maximum degree Δ. This can be done quite efficiently: if we guess one vertex v in the clique, then the remaining vertices in the clique must be among the Δ neighbors of v. Thus we can try all of the 2^{Δ} subsets of the neighbors of v, and return the largest clique that we found. The total running time of this algorithm is $\mathcal{O}(2^{\Delta} \cdot \Delta^2 \cdot n)$, which is quite feasible for $\Delta = 20$. Again it is possible to use complexity theoretic assumptions on the hardness of CNF-SAT to show that this algorithm is asymptotically optimal, up to multiplicative constants in the exponent.

What the algorithm above shows is that the Clique problem is FPT when the parameter is the maximum degree Δ of the input graph. At the same time Clique is probably not FPT when the parameter is the solution size k. Thus, the classification of the problem into "tractable" or "intractable" crucially depends on the choice of parameter. This makes a lot of sense; the more we know about our input instances, the more we can exploit algorithmically!

The art of parameterization

For typical optimization problems, one can immediately find a relevant parameter: the size of the solution we are looking for. In some cases, however, it is not completely obvious what we mean by the size of the solution. For example, consider the variant of Bar Fight Prevention where we want to reject at most k guests such that the number of conflicts is reduced to at most ℓ (as we believe that the bouncers at the bar can handle ℓ conflicts, but not more). Then we can parameterize either by k or by ℓ. We may even parameterize by both: then the goal is to find an FPT algorithm with running time $f(k, \ell) \cdot n^c$ for some computable function f depending only on k and ℓ. Thus the theory of parameterization and FPT algorithms can be extended to considering a set of parameters at the same time. Formally, however, one can express parameterization by k and ℓ simply by defining the value $k + \ell$ to be the parameter: an $f(k, \ell) \cdot n^c$ algorithm exists if and only if an $f(k + \ell) \cdot n^c$ algorithm exists.

The parameters k and ℓ in the extended Bar Fight Prevention example of the previous paragraph are related to the *objective* of the problem: they are parameters explicitly given in the input, defining the properties of the solution we are looking for. We get more examples of this type of parameter if we define variants of Bar Fight Prevention where we need to reject at most k guests such that, say, the number of conflicts decreases by p, or such that each accepted guest has conflicts with at most d other accepted guests, or such that the average number of conflicts per guest is at most a. Then the parameters p, d, and a are again explicitly given in the input, telling us what kind of solution we need to find. The parameter Δ (maximum degree of the graph) in the Clique example is a parameter of a very different type: it is not given explicitly in the input, but it is a *measure* of some property of the input instance. We defined and explored this particular measure because we believed that it is typically small in the input instances we care about: this parameter expresses some structural property of typical instances. We can identify and investigate any number of such parameters. For example, in problems involving graphs, we may parameterize by any structural parameter of the graph at hand. Say, if we believe that the problem is easy on planar graphs and the instances are "almost planar", then we may explore the parameterization by the genus of the graph (roughly speaking, a graph has genus g if it can be drawn without edge crossings on a sphere with g holes in it). A large part of Chapter 7 (and also Chapter 11) is devoted to parameterization by treewidth, which is a very important parameter measuring the "tree-likeness" of the graph. For problems involving satisfiability of Boolean formulas, we can have such parameters as the number of variables, or clauses, or the number of clauses that need to be satisfied, or that are allowed not to be satisfied. For problems involving a set of strings, one can parameterize by the maximum length of the strings, by the size of the alphabet, by the maximum number of distinct symbols appearing in each string, etc. In a problem involving a set of geometric objects (say, points in space, disks, or polygons), one may parameterize by the maximum number of vertices of each polygon or the dimension of the space where the problem is defined. For each problem, with a bit of creativity, one can come up with a large number of (combinations of) parameters worth studying.

For the same problem there can be multiple choices of parameters. Selecting the right parameter(s) for a particular problem is an art.

Parameterized complexity allows us to study how different parameters influence the complexity of the problem. A successful parameterization of a problem needs to satisfy two properties. First, we should have some reason to believe that the selected parameter (or combination of parameters) is typically small on input instances in some application. Second, we need efficient algorithms where the combinatorial explosion is restricted to the parameter(s), that is, we want the problem to be FPT with this parameterization. Finding good parameterizations is an art on its own and one may spend quite some time on analyzing different parameterizations of the same problem. However, in this book we focus more on explaining algorithmic techniques via carefully chosen illustrative examples, rather than discussing every possible aspect of a particular problem. Therefore, even though different parameters and parameterizations will appear throughout the book, we will not try to give a complete account of all known parameterizations and results for any concrete problem.

1.1 Formal definitions

We finish this chapter by leaving the realm of pub jokes and moving to more serious matters. Before we start explaining the techniques for designing parameterized algorithms, we need to introduce formal foundations of parameterized complexity. That is, we need to have rigorous definitions of what a parameterized problem is, and what it means that a parameterized problem belongs to a specific complexity class.

Definition 1.1. A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, k is called the *parameter*.

For example, an instance of Clique parameterized by the solution size is a pair (G, k), where we expect G to be an undirected graph encoded as a string over Σ, and k is a positive integer. That is, a pair (G, k) belongs to the Clique parameterized language if and only if the string G correctly encodes an undirected graph, which we will also denote by G, and moreover the graph

G contains a clique on k vertices. Similarly, an instance of the CNF-SAT problem (satisfiability of propositional formulas in CNF), parameterized by the number of variables, is a pair (φ, n), where we expect φ to be the input formula encoded as a string over Σ and n to be the number of variables of φ. That is, a pair (φ, n) belongs to the CNF-SAT parameterized language if and only if the string φ correctly encodes a CNF formula with n variables, and the formula is satisfiable.

We define the size of an instance (x, k) of a parameterized problem as $|x| + k$. One interpretation of this convention is that, when given to the algorithm on the input, the parameter k is encoded in unary.

Definition 1.2. A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *fixed-parameter tractable* (FPT) if there exists an algorithm $\mathcal{A}$ (called a *fixed-parameter algorithm*), a computable function $f\colon \mathbb{N} \to \mathbb{N}$, and a constant c such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^c$. The complexity class containing all fixed-parameter tractable problems is called FPT.

Before we go further, let us make some remarks about the function f in this definition. Observe that we assume f to be computable, as otherwise we would quickly run into trouble when developing complexity theory for fixed-parameter tractability. For technical reasons, it will be convenient to assume, from now on, that f is also nondecreasing. Observe that this assumption has no influence on the definition of fixed-parameter tractability as stated in Definition 1.2, since for every computable function $f\colon \mathbb{N} \to \mathbb{N}$ there exists a computable nondecreasing function $\bar{f}$ that is never smaller than f: we can simply take $\bar{f}(k) = \max_{i=0,1,\ldots,k} f(i)$. Also, for standard algorithmic results it is always the case that the bound on the running time is a nondecreasing function of the complexity measure, so this assumption is indeed satisfied in practice. However, the assumption about f being nondecreasing is formally needed in various situations, for example when performing reductions.

We now define the complexity class XP.

Definition 1.3. A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *slice-wise polynomial* (XP) if there exists an algorithm $\mathcal{A}$ and two computable functions $f, g\colon \mathbb{N} \to \mathbb{N}$ such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^{g(k)}$. The complexity class containing all slice-wise polynomial problems is called XP.

Again, we shall assume that the functions f, g in this definition are nondecreasing.

The definition of a parameterized problem, as well as the definitions of the classes FPT and XP, can easily be generalized to encompass multiple parameters. In this setting we simply allow k to be not just one nonnegative integer, but a vector of d nonnegative integers, for some fixed constant d. Then the functions f and g in the definitions of the complexity classes FPT and XP can depend on all these parameters.

Just as "polynomial time" and "polynomial-time algorithm" usually refer to time polynomial in the input size, the terms "FPT time" and "FPT algorithms" refer to time $f(k)$ times a polynomial in the input size. Here f is a computable function of k and the degree of the polynomial is independent of both n and k. The same holds for "XP time" and "XP algorithms", except that here the degree of the polynomial is allowed to depend on the parameter k, as long as it is upper bounded by $g(k)$ for some computable function g.

Observe that, given some parameterized problem L, the algorithm designer has essentially two different optimization goals when designing FPT algorithms for L. Since the running time has to be of the form $f(k) \cdot n^c$, one can:

- optimize the *parametric dependence* of the running time, i.e., try to design an algorithm where function f grows as slowly as possible; or
- optimize the *polynomial factor* in the running time, i.e., try to design an algorithm where constant c is as small as possible.

Both these goals are equally important, from both a theoretical and a practical point of view. Unfortunately, keeping track of and optimizing both factors of the running time can be a very difficult task. For this reason, most research on parameterized algorithms concentrates on optimizing one of the factors, and putting more focus on each of them constitutes one of the two dominant trends in parameterized complexity. Sometimes, when we are not interested in the exact value of the polynomial factor, we use the $\mathcal{O}^*$*-notation*, which suppresses factors polynomial in the input size. More precisely, a running time $\mathcal{O}^*(f(k))$ means that the running time is upper bounded by $f(k) \cdot n^{\mathcal{O}(1)}$, where n is the input size.

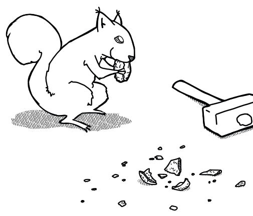
The theory of parameterized complexity has been pioneered by Downey and Fellows over the last two decades [148, 149, 150, 151, 153]. The main achievement of their work is a comprehensive complexity theory for parameterized problems, with appropriate notions of reduction and completeness. The primary goal is to understand the qualitative difference between fixed-parameter tractable problems, and problems that do not admit such efficient algorithms. The theory contains a rich "positive" toolkit of techniques for developing efficient parameterized algorithms, as well as a corresponding "negative" toolkit that supports a theory of parameterized intractability. This textbook is mostly devoted to a presentation of the positive toolkit: in Chapters 2 through 12 we present various algorithmic techniques for designing fixed-parameter tractable algorithms. As we have argued, the process of algorithm design has to use both toolkits in order to be able to conclude that certain research directions are pointless. Therefore, in Part III we give an introduction to lower bounds for parameterized problems.

Bibliographic notes

Downey and Fellows laid the foundation of parameterized complexity in the series of papers [1, 149, 150, 151]. The classic reference on parameterized complexity is the book of Downey and Fellows [153]. The new edition of this book [154] is a comprehensive overview of the state of the art in many areas of parameterized complexity. The book of Flum and Grohe [189] is an extensive introduction to the area with a strong emphasis on the complexity viewpoint. An introduction to basic algorithmic techniques in parameterized complexity up to 2006 is given in the book of Niedermeier [376]. The recent book [51] contains a collection of surveys on different areas of parameterized complexity.

Chapter 2
Kernelization

Kernelization is a systematic approach to study polynomial-time preprocessing algorithms. It is an important tool in the design of parameterized algorithms. In this chapter we explain basic kernelization techniques such as crown decomposition, the expansion lemma, the sunflower lemma, and linear programming. We illustrate these techniques by obtaining kernels for Vertex Cover, Feedback Arc Set in Tournaments, Edge Clique Cover, Maximum Satisfiability, *and d-*Hitting Set.



Preprocessing (data reduction or kernelization) is used universally in almost every practical computer implementation that aims to deal with an NP-hard problem. The goal of a preprocessing subroutine is to solve efficiently the “easy parts” of a problem instance and reduce it (shrink it) to its computationally difficult “core” structure (the *problem kernel* of the instance). In other words, the idea of this method is to reduce (but not necessarily solve) the given problem instance to an equivalent “smaller sized” instance in time polynomial in the input size. A slower exact algorithm can then be run on this smaller instance.

How can we measure the effectiveness of such a preprocessing subroutine? Suppose we define a useful preprocessing algorithm as one that runs in polynomial time and replaces an instance I with an equivalent instance that is at least one bit smaller. Then the existence of such an algorithm for an NP-hard problem would imply P= NP, making it unlikely that such an algorithm can be found. For a long time, there was no other suggestion for a formal definition of useful preprocessing, leaving the mathematical analysis of polynomial-time preprocessing algorithms largely neglected. But in the language of parameterized complexity, we can formulate a definition of useful preprocessing by demanding that large instances with a small parameter should be shrunk, while instances that are small compared to their parameter

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_2

do not have to be processed further. These ideas open up the “lost continent” of polynomial-time algorithms called kernelization.

In this chapter we illustrate some commonly used techniques to design kernelization algorithms through concrete examples. The next section, Section 2.1, provides formal definitions. In Section 2.2 we give kernelization algorithms based on so-called natural reduction rules. Section 2.3 introduces the concepts of crown decomposition and the expansion lemma, and illustrates it on Maximum Satisfiability. Section 2.5 studies tools based on linear programming and gives a kernel for Vertex Cover. Finally, we study the sunflower lemma in Section 2.6 and use it to obtain a polynomial kernel for d-Hitting Set.

2.1 Formal definitions

We now turn to the formal definition that captures the notion of kernelization. A *data reduction rule*, or simply, reduction rule, for a parameterized problem Q is a function $\phi\colon \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ that maps an instance (I,k) of Q to an equivalent instance (I',k') of Q such that ϕ is computable in time polynomial in $|I|$ and k. We say that two instances of Q are *equivalent* if $(I,k) \in Q$ if and only if $(I',k') \in Q$; this property of the reduction rule ϕ, that it translates an instance to an equivalent one, is sometimes referred to as the *safeness* or *soundness* of the reduction rule. In this book, we stick to the phrases: *a rule is safe* and *the safeness of a reduction rule*.

The general idea is to design a *preprocessing algorithm* that consecutively applies various data reduction rules in order to shrink the instance size as much as possible. Thus, such a preprocessing algorithm takes as input an instance $(I,k) \in \Sigma^* \times \mathbb{N}$ of Q, works in polynomial time, and returns an equivalent instance (I',k') of Q. In order to formalize the requirement that the output instance has to be small, we apply the main principle of Parameterized Complexity: The complexity is measured in terms of the parameter. Consequently, the *output size* of a preprocessing algorithm $\mathcal{A}$ is a function $\mathrm{size}_{\mathcal{A}}\colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined as follows:

$$\mathrm{size}_{\mathcal{A}}(k) = \sup\{|I'| + k' \ :\ (I',k') = \mathcal{A}(I,k),\ I \in \Sigma^*\}.$$

In other words, we look at all possible instances of Q with a fixed parameter k, and measure the supremum of the sizes of the output of $\mathcal{A}$ on these instances. Note that this supremum may be infinite; this happens when we do not have any bound on the size of $\mathcal{A}(I,k)$ in terms of the input parameter k only. *Kernelization algorithms* are exactly these preprocessing algorithms whose output size is finite and bounded by a computable function of the parameter.

Definition 2.1 (Kernelization, kernel). A *kernelization algorithm*, or simply a *kernel*, for a parameterized problem Q is an algorithm $\mathcal{A}$ that, given an instance (I, k) of Q, works in polynomial time and returns an equivalent instance (I', k') of Q. Moreover, we require that $\mathrm{size}_{\mathcal{A}}(k) \leq g(k)$ for some computable function $g\colon \mathbb{N} \to \mathbb{N}$.

The size requirement in this definition can be reformulated as follows: There exists a computable function $g(\cdot)$ such that whenever (I', k') is the output for an instance (I, k), then it holds that $|I'| + k' \leq g(k)$. If the upper bound $g(\cdot)$ is a polynomial (linear) function of the parameter, then we say that Q admits a *polynomial (linear) kernel*. We often abuse the notation and call the output of a kernelization algorithm the "reduced" equivalent instance, also a kernel.

In the course of this chapter, we will often encounter a situation when in some boundary cases we are able to completely resolve the considered problem instance, that is, correctly decide whether it is a yes-instance or a no-instance. Hence, for clarity, we allow the reductions (and, consequently, the kernelization algorithm) to return a yes/no answer instead of a reduced instance. Formally, to fit into the introduced definition of a kernel, in such cases the kernelization algorithm should instead return a constant-size trivial yes-instance or no-instance. Note that such instances exist for every parameterized language except for the empty one and its complement, and can be therefore hardcoded into the kernelization algorithm.

Recall that, given an instance (I, k) of Q, the size of the kernel is defined as the number of *bits* needed to encode the reduced equivalent instance I' plus the parameter value k'. However, when dealing with problems on graphs, hypergraphs, or formulas, often we would like to emphasize other aspects of output instances. For example, for a graph problem Q, we could say that Q admits a kernel with $\mathcal{O}(k^3)$ vertices and $\mathcal{O}(k^5)$ edges to emphasize the upper bound on the number of vertices and edges in the output instances. Similarly, for a problem defined on formulas, we could say that the problem admits a kernel with $\mathcal{O}(k)$ variables.

It is important to mention here that the early definitions of kernelization required that $k' \leq k$. On an intuitive level this makes sense, as the parameter k measures the complexity of the problem — thus the larger the k, the harder the problem. This requirement was subsequently relaxed, notably in the context of lower bounds. An advantage of the more liberal notion of kernelization is that it is robust with respect to polynomial transformations of the kernel. However, it limits the connection with practical preprocessing. All the kernels mentioned in this chapter respect the fact that the output parameter is at most the input parameter, that is, $k' \leq k$.

While usually in Computer Science we measure the efficiency of an algorithm by estimating its running time, the central measure of the efficiency of a kernelization algorithm is a bound on its output size. Although the actual running time of a kernelization algorithm is often very important for practical applications, in theory a kernelization algorithm is only required to run in polynomial time.

If we have a kernelization algorithm for a problem for which there is some algorithm (with any running time) to decide whether (I, k) is a yes-instance, then clearly the problem is FPT, as the size of the reduced instance I is simply a function of k (and independent of the input size n). However, a surprising result is that the converse is also true.

Lemma 2.2. *If a parameterized problem Q is* FPT *then it admits a kernelization algorithm.*

Proof. Since Q is FPT, there is an algorithm $\mathcal{A}$ deciding if $(I, k) \in Q$ in time $f(k) \cdot |I|^c$ for some computable function f and a constant c. We obtain a kernelization algorithm for Q as follows. Given an input (I, k), the kernelization algorithm runs $\mathcal{A}$ on (I, k), for at most $|I|^{c+1}$ steps. If it terminates with an answer, use that answer to return either that (I, k) is a yes-instance or that it is a no-instance. If $\mathcal{A}$ does not terminate within $|I|^{c+1}$ steps, then return (I, k) itself as the output of the kernelization algorithm. Observe that since $\mathcal{A}$ did not terminate in $|I|^{c+1}$ steps, we have that $f(k) \cdot |I|^c > |I|^{c+1}$, and thus $|I| < f(k)$. Consequently, we have $|I| + k \leq f(k) + k$, and we obtain a kernel of size at most $f(k) + k$; note that this upper bound is computable as $f(k)$ is a computable function. $\square$

Lemma 2.2 implies that a decidable problem admits a kernel if and only if it is fixed-parameter tractable. Thus, in a sense, kernelization can be another way of defining fixed-parameter tractability.

However, kernels obtained by this theoretical result are usually of exponential (or even worse) size, while problem-specific data reduction rules often achieve quadratic ($g(k) = \mathcal{O}(k^2)$) or even linear-size ($g(k) = \mathcal{O}(k)$) kernels. So a natural question for any concrete FPT problem is whether it admits a problem kernel that is bounded by a polynomial function of the parameter ($g(k) = k^{\mathcal{O}(1)}$). In this chapter we give polynomial kernels for several problems using some elementary methods. In Chapter 9, we give more advanced methods for obtaining kernels.

2.2 Some simple kernels

In this section we give kernelization algorithms for Vertex Cover and Feedback Arc Set in Tournaments (FAST) based on a few natural reduction rules.

2.2.1 Vertex Cover

Let G be a graph and $S \subseteq V(G)$. The set S is called a *vertex cover* if for every edge of G at least one of its endpoints is in S. In other words, the graph $G - S$ contains no edges and thus $V(G) \setminus S$ is an *independent set*. In the Vertex Cover problem, we are given a graph G and a positive integer k as input, and the objective is to check whether there exists a vertex cover of size at most k.

The first reduction rule is based on the following simple observation. For a given instance (G, k) of Vertex Cover, if the graph G has an isolated vertex, then this vertex does not cover any edge and thus its removal does not change the solution. This shows that the following rule is safe.

Reduction VC.1. If G contains an isolated vertex v, delete v from G. The new instance is $(G - v, k)$.

The second rule is based on the following natural observation:

If G contains a vertex v of degree more than k, then v should be in every vertex cover of size at most k.

Indeed, this is because if v is not in a vertex cover, then we need at least $k + 1$ vertices to cover edges incident to v. Thus our second rule is the following.

Reduction VC.2. If there is a vertex v of degree at least $k + 1$, then delete v (and its incident edges) from G and decrement the parameter k by 1. The new instance is $(G - v, k - 1)$.

Observe that exhaustive application of reductions VC.1 and VC.2 completely removes the vertices of degree 0 and degree at least $k + 1$. The next step is the following observation.

If a graph has maximum degree d, then a set of k vertices can cover at most kd edges.

This leads us to the following lemma.

Lemma 2.3. *If (G, k) is a yes-instance and none of the reduction rules VC.1, VC.2 is applicable to G, then $|V(G)| \leq k^2 + k$ and $|E(G)| \leq k^2$.*

Proof. Because we cannot apply Reductions VC.1 anymore on G, G has no isolated vertices. Thus for every vertex cover S of G, every vertex of $G - S$ should be adjacent to some vertex from S. Since we cannot apply Reductions VC.2, every vertex of G has degree at most k. It follows that

$|V(G-S)| \leq k|S|$ and hence $|V(G)| \leq (k+1)|S|$. Since (G,k) is a yes-instance, there is a vertex cover S of size at most k, so $|V(G)| \leq (k+1)k$. Also every edge of G is covered by some vertex from a vertex cover and every vertex can cover at most k edges. Hence if G has more than k^2 edges, this is again a no-instance. $\square$

Lemma 2.3 allows us to claim the final reduction rule that explicitly bounds the size of the kernel.

Reduction VC.3. Let (G,k) be an input instance such that Reductions VC.1 and VC.2 are not applicable to (G,k). If $k < 0$ and G has more than $k^2 + k$ vertices, or more than k^2 edges, then conclude that we are dealing with a no-instance.

Finally, we remark that all reduction rules are trivially applicable in linear time. Thus, we obtain the following theorem.

Theorem 2.4. Vertex Cover *admits a kernel with $\mathcal{O}(k^2)$ vertices and $\mathcal{O}(k^2)$ edges.*

2.2.2 Feedback Arc Set in Tournaments

In this section we discuss a kernel for the Feedback Arc Set in Tournaments problem. A *tournament* is a directed graph T such that for every pair of vertices $u, v \in V(T)$, exactly one of (u,v) or (v,u) is a directed edge (also often called an *arc*) of T. A set of edges A of a directed graph G is called a *feedback arc set* if every directed cycle of G contains an edge from A. In other words, the removal of A from G turns it into a directed acyclic graph. Very often, acyclic tournaments are called *transitive* (note that then $E(G)$ is a transitive relation). In the Feedback Arc Set in Tournaments problem we are given a tournament T and a nonnegative integer k. The objective is to decide whether T has a feedback arc set of size at most k.

For tournaments, the deletion of edges results in directed graphs which are not tournaments anymore. Because of that, it is much more convenient to use the characterization of a feedback arc set in terms of "reversing edges". We start with the following well-known result about *topological orderings* of directed acyclic graphs.

Lemma 2.5. *A directed graph G is acyclic if and only if it is possible to order its vertices in such a way such that for every directed edge (u,v), we have $u < v$.*

We leave the proof of Lemma 2.5 as an exercise; see Exercise 2.1. Given a directed graph G and a subset $F \subseteq E(G)$ of edges, we define $G \circledast F$ to be the directed graph obtained from G by reversing all the edges of F. That is, if $\text{rev}(F) = \{(u,v) \ : \ (v,u) \in F\}$, then for $G \circledast F$ the vertex set is $V(G)$

and the edge set $E(G \circledast F) = (E(G) \cup \mathrm{rev}(F)) \setminus F$. Lemma 2.5 implies the following.

Observation 2.6. Let G be a directed graph and let F be a subset of edges of G. If $G \circledast F$ is a directed acyclic graph then F is a feedback arc set of G.

The following lemma shows that, in some sense, the opposite direction of the statement in Observation 2.6 is also true. However, the minimality condition in Lemma 2.7 is essential, see Exercise 2.2.

Lemma 2.7. *Let G be a directed graph and F be a subset of $E(G)$. Then F is an inclusion-wise minimal feedback arc set of G if and only if F is an inclusion-wise minimal set of edges such that $G \circledast F$ is an acyclic directed graph.*

Proof. We first prove the forward direction of the lemma. Let F be an inclusion-wise minimal feedback arc set of G. Assume to the contrary that $G \circledast F$ has a directed cycle C. Then C cannot contain only edges of $E(G) \setminus F$, as that would contradict the fact that F is a feedback arc set. Let $f_1, f_2, \cdots, f_\ell$ be the edges of $C \cap \mathrm{rev}(F)$ in the order of their appearance on the cycle C, and let $e_i \in F$ be the edge f_i reversed. Since F is inclusion-wise minimal, for every e_i, there exists a directed cycle C_i in G such that $F \cap C_i = \{e_i\}$. Now consider the following closed walk W in G: we follow the cycle C, but whenever we are to traverse an edge $f_i \in \mathrm{rev}(F)$ (which is not present in G), we instead traverse the path $C_i - e_i$. By definition, W is a closed walk in G and, furthermore, note that W does not contain any edge of F. This contradicts the fact that F is a feedback arc set of G.

The minimality follows from Observation 2.6. That is, every set of edges F such that $G \circledast F$ is acyclic is also a feedback arc set of G, and thus, if F is not a minimal set such that $G \circledast F$ is acyclic, then it will contradict the fact that F is a minimal feedback arc set.

For the other direction, let F be an inclusion-wise minimal set of edges such that $G \circledast F$ is an acyclic directed graph. By Observation 2.6, F is a feedback arc set of G. Moreover, F is an inclusion-wise minimal feedback arc set, because if a proper subset F' of F is an inclusion-wise minimal feedback arc set of G, then by the already proved implication of the lemma, $G \circledast F'$ is an acyclic directed graph, a contradiction with the minimality of F. ☐

We are ready to give a kernel for Feedback Arc Set in Tournaments.

Theorem 2.8. Feedback Arc Set in Tournaments *admits a kernel with at most $k^2 + 2k$ vertices.*

Proof. Lemma 2.7 implies that a tournament T has a feedback arc set of size at most k if and only if it can be turned into an acyclic tournament by reversing directions of at most k edges. We will use this characterization for the kernel.

In what follows by a *triangle* we mean a directed cycle of length three. We give two simple reduction rules.

Reduction FAST.1. If an edge e is contained in at least $k+1$ triangles, then reverse e and reduce k by 1.

Reduction FAST.2. If a vertex v is not contained in any triangle, then delete v from T.

The rules follow similar guidelines as in the case of VERTEX COVER. In Reduction FAST.1, we greedily take into a solution an edge that participates in $k+1$ otherwise disjoint forbidden structures (here, triangles). In Reduction FAST.2, we discard vertices that do not participate in any forbidden structure, and should be irrelevant to the problem.

However, a formal proof of the safeness of Reduction FAST.2 is not immediate: we need to verify that deleting v and its incident edges does not make make a yes-instance out of a no-instance.

Note that after applying any of the two rules, the resulting graph is again a tournament. The first rule is safe because if we do not reverse e, we have to reverse at least one edge from each of $k+1$ triangles containing e. Thus e belongs to every feedback arc set of size at most k.

Let us now prove the safeness of the second rule. Let $X = N^+(v)$ be the set of heads of directed edges with tail v and let $Y = N^-(v)$ be the set of tails of directed edges with head v. Because T is a tournament, X and Y is a partition of $V(T) \setminus \{v\}$. Since v is not a part of any triangle in T, we have that there is no edge from X to Y (with head in Y and tail in X). Consequently, for any feedback arc set A_1 of tournament $T[X]$ and any feedback arc set A_2 of tournament $T[Y]$, the set $A_1 \cup A_2$ is a feedback arc set of T. As the reverse implication is trivial (for any feedback arc set A in T, $A \cap E(T[X])$ is a feedback arc set of $T[X]$, and $A \cap E(T[Y])$ is a feedback arc set of $T[Y]$), we have that (T, k) is a yes-instance if and only if $(T - v, k)$ is.

Finally, we show that every reduced yes-instance T, an instance on which none of the presented reduction rules are applicable, has at most $k(k+2)$ vertices. Let A be a feedback arc set of a reduced instance T of size at most k. For every edge $e \in A$, aside from the two endpoints of e, there are at most k vertices that are in triangles containing e — otherwise we would be able to apply Reduction FAST.1. Since every triangle in T contains an edge of A and every vertex of T is in a triangle, we have that T has at most $k(k+2)$ vertices.

Thus, given (T, k) we apply our reduction rules exhaustively and obtain an equivalent instance (T', k'). If T' has more than $k'^2 + k'$ vertices, then the algorithm returns that (T, k) is a no-instance, otherwise we get the desired kernel. This completes the proof of the theorem. $\square$

2.2.3 Edge Clique Cover

Not all FPT problems admit polynomial kernels. In the Edge Clique Cover problem, we are given a graph G and a nonnegative integer k, and the goal is to decide whether the edges of G can be covered by at most k cliques. In this section we give an exponential kernel for Edge Clique Cover. In Theorem 14.20 of Section *14.3.3, we remark that this simple kernel is essentially optimal.

Let us recall the reader that we use $N(v) = \{u : uv \in E(G)\}$ to denote the neighborhood of vertex v in G, and $N[v] = N(v) \cup \{v\}$ to denote the closed neighborhood of v. We apply the following data reduction rules in the given order (i.e., we always use the lowest-numbered rule that modifies the instance).

Reduction ECC.1. Remove isolated vertices.

Reduction ECC.2. If there is an isolated edge uv (a connected component that is just an edge), delete it and decrease k by 1. The new instance is $(G - \{u, v\}, k - 1)$.

Reduction ECC.3. If there is an edge uv whose endpoints have exactly the same closed neighborhood, that is, $N[u] = N[v]$, then delete v. The new instance is $(G - v, k)$.

The crux of the presented kernel for Edge Clique Cover is an observation that two true twins (vertices u and v with $N[u] = N[v]$) can be treated in exactly the same way in some optimum solution, and hence we can reduce them. Meanwhile, the vertices that are contained in exactly the same set of cliques in a feasible solution *have to* be true twins. This observation bounds the size of the kernel.

The safeness of the first two reductions is trivial, while the safeness of Reduction ECC.3 follows from the observation that a solution in $G - v$ can be extended to a solution in G by adding v to all the cliques containing u (see Exercise 2.3).

Theorem 2.9. Edge Clique Cover *admits a kernel with at most* 2^k *vertices.*

Proof. We start with the following claim.

Claim. If (G, k) is a reduced yes-instance, on which none of the presented reduction rules can be applied, then $|V(G)| \leq 2^k$.

Proof. Let $C_1, \ldots, C_k$ be an edge clique cover of G. We claim that G has at most 2^k vertices. Targeting a contradiction, let us assume that G has more than 2^k vertices. We assign to each vertex $v \in V(G)$ a binary vector b_v of length k, where bit i, $1 \leq i \leq k$, is set to 1 if and only if v is contained in clique C_i. Since there are only 2^k possible vectors, there must be $u \neq v \in V(G)$ with $b_u = b_v$. If b_u and b_v are zero vectors, the first rule applies; otherwise, u and v are contained in the same cliques. This means that u and v are adjacent and have the same neighborhood; thus either Reduction ECC.2 or Reduction ECC.3 applies. Hence, if G has more than 2^k vertices, at least one of the reduction rules can be applied to it, which is a contradiction to the initial assumption that G is reduced. This completes the proof of the claim. □

The kernelization algorithm works as follows. Given an instance (G, k), it applies Reductions ECC.1, ECC.2, and ECC.3 exhaustively. If the resulting graph has more than 2^k vertices the kernelization algorithm outputs that the input instance is a no-instance, else it outputs the reduced instance. □

2.3 Crown decomposition

Crown decomposition is a general kernelization technique that can be used to obtain kernels for many problems. The technique is based on the classical matching theorems of Kőnig and Hall.

Recall that for disjoint vertex subsets U, W of a graph G, a matching M is called *a matching of U into W* if every edge of M connects a vertex of U and a vertex of W and, moreover, every vertex of U is an endpoint of some edge of M. In this situation, we also say that M *saturates* U.

Definition 2.10 (Crown decomposition). A *crown decomposition* of a graph G is a partitioning of $V(G)$ into three parts C, H and R, such that

1. C is nonempty.
2. C is an independent set.
3. There are no edges between vertices of C and R. That is, H separates C and R.
4. Let E' be the set of edges between vertices of C and H. Then E' contains a matching of size $|H|$. In other words, G contains a matching of H into C.

The set C can be seen as a crown put on head H of the remaining part R, see Fig. 2.1. Note that the fact that E' contains a matching of size $|H|$ implies that there is a matching of H into C. This is a matching in the subgraph G', with the vertex set $C \cup H$ and the edge set E', saturating all the vertices of H.

For finding a crown decomposition in polynomial time, we use the following well known structural and algorithmic results. The first is a mini-max theorem due to Kőnig.

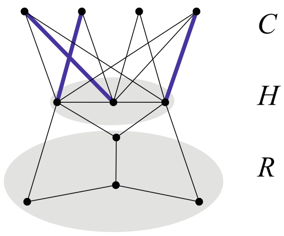


Fig. 2.1: Example of a crown decomposition. Set C is an independent set, H separates C and R, and there is a matching of H into C

Theorem 2.11 (Kőnig's theorem, [303]). *In every undirected bipartite graph the size of a maximum matching is equal to the size of a minimum vertex cover.*

Let us recall that a matching *saturates* a set of vertices S when every vertex in S is incident to an edge in the matching. The second classic result states that in bipartite graphs, a trivial necessary condition for the existence of a matching is also sufficient.

Theorem 2.12 (Hall's theorem, [256]). *Let G be an undirected bipartite graph with bipartition (V_1, V_2). The graph G has a matching saturating V_1 if and only if for all $X \subseteq V_1$, we have $|N(X)| \geq |X|$.*

The following theorem is due to Hopcroft and Karp [268]. The proof of the (nonstandard) second claim of the theorem is deferred to Exercise 2.21.

Theorem 2.13 (Hopcroft-Karp algorithm, [268]). *Let G be an undirected bipartite graph with bipartition V_1 and V_2, on n vertices and m edges. Then we can find a maximum matching as well as a minimum vertex cover of G in time $\mathcal{O}(m\sqrt{n})$. Furthermore, in time $\mathcal{O}(m\sqrt{n})$ either we can find a matching saturating V_1 or an inclusion-wise minimal set $X \subseteq V_1$ such that $|N(X)| < |X|$.*

The following lemma is the basis for kernelization algorithms using crown decomposition.

Lemma 2.14 (Crown lemma). *Let G be a graph without isolated vertices and with at least $3k + 1$ vertices. There is a polynomial-time algorithm that either*

- *finds a matching of size $k + 1$ in G; or*
- *finds a crown decomposition of G.*

Proof. We first find an inclusion-maximal matching M in G. This can be done by a greedy algorithm. If the size of M is $k + 1$, then we are done.

Hence, we assume that $|M| \leq k$, and let V_M be the endpoints of M. We have $|V_M| \leq 2k$. Because M is a maximal matching, the remaining set of vertices $I = V(G) \setminus V_M$ is an independent set.

Consider the bipartite graph G_{I,V_M} formed by edges of G between V_M and I. We compute a minimum-sized vertex cover X and a maximum sized matching M' of the bipartite graph G_{I,V_M} in polynomial time using Theorem 2.13. We can assume that $|M'| \leq k$, for otherwise we are done. Since $|X| = |M'|$ by Kőnig's theorem (Theorem 2.11), we infer that $|X| \leq k$.

If no vertex of X is in V_M, then $X \subseteq I$. We claim that $X = I$. For a contradiction assume that there is a vertex $w \in I \setminus X$. Because G has no isolated vertices there is an edge, say wz, incident to w in G_{I,V_M}. Since G_{I,V_M} is bipartite, we have that $z \in V_M$. However, X is a vertex cover of G_{I,V_M} such that $X \cap V_M = \emptyset$, which implies that $w \in X$. This is contrary to our assumption that $w \notin X$, thus proving that $X = I$. But then $|I| \leq |X| \leq k$, and G has at most

$$|I| + |V_M| \leq k + 2k = 3k$$

vertices, which is a contradiction.

Hence, $X \cap V_M \neq \emptyset$. We obtain a crown decomposition (C, H, R) as follows. Since $|X| = |M'|$, every edge of the matching M' has exactly one endpoint in X. Let M^* denote the subset of M' such that every edge from M^* has exactly one endpoint in $X \cap V_M$ and let V_{M^*} denote the set of endpoints of edges in M^*. We define head $H = X \cap V_M = X \cap V_{M^*}$, crown $C = V_{M^*} \cap I$, and the remaining part $R = V(G) \setminus (C \cup H) = V(G) \setminus V_{M^*}$. In other words, H is the set of endpoints of edges of M^* that are present in V_M and C is the set of endpoints of edges of M^* that are present in I. Obviously, C is an independent set and by construction, M^* is a matching of H into C. Furthermore, since X is a vertex cover of G_{I,V_M}, every vertex of C can be adjacent only to vertices of H and thus H separates C and R. This completes the proof. □

The crown lemma gives a relatively strong structural property of graphs with a small vertex cover (equivalently, a small maximum matching). If in a studied problem the parameter upper bounds the size of a vertex cover (maximum matching), then there is a big chance that the structural insight given by the crown lemma would help in developing a small kernel — quite often with number of vertices bounded linearly in the parameter.

We demonstrate the application of crown decompositions on kernelization for Vertex Cover and Maximum Satisfiability.

2.3.1 Vertex Cover

Consider a Vertex Cover instance (G, k). By an exhaustive application of Reduction VC.1, we may assume that G has no isolated vertices. If $|V(G)| > 3k$, we may apply the crown lemma to the graph G and integer k, obtaining either a matching of size $k+1$, or a crown decomposition $V(G) = C \cup H \cup R$. In the first case, the algorithm concludes that (G, k) is a no-instance.

In the latter case, let M be a matching of H into C. Observe that the matching M witnesses that, for every vertex cover X of the graph G, X contains at least $|M| = |H|$ vertices of $H \cup C$ to cover the edges of M. On the other hand, the set H covers all edges of G that are incident to $H \cup C$. Consequently, there exists a minimum vertex cover of G that contains H, and we may reduce (G, k) to $(G-H, k-|H|)$. Note that in the instance $(G-H, k-|H|)$, the vertices of C are isolated and will be reduced by Reduction VC.1.

As the crown lemma promises us that $H \neq \emptyset$, we can always reduce the graph as long as $|V(G)| > 3k$. Thus, we obtain the following.

Theorem 2.15. Vertex Cover *admits a kernel with at most* $3k$ *vertices.*

2.3.2 Maximum Satisfiability

For a second application of the crown decomposition, we look at the following parameterized version of Maximum Satisfiability. Given a CNF formula F, and a nonnegative integer k, decide whether F has a truth assignment satisfying at least k clauses.

Theorem 2.16. Maximum Satisfiability *admits a kernel with at most* k *variables and* $2k$ *clauses.*

Proof. Let φ be a CNF formula with n variables and m clauses. Let ψ be an arbitrary assignment to the variables and let $\neg\psi$ be the assignment obtained by complementing the assignment of ψ. That is, if ψ assigns $\delta \in \{\top, \bot\}$ to some variable x then $\neg\psi$ assigns $\neg\delta$ to x. Observe that either ψ or $\neg\psi$ satisfies at least $m/2$ clauses, since every clause is satisfied by ψ or $\neg\psi$ (or by both). This means that, if $m \geq 2k$, then (φ, k) is a yes-instance. In what follows we give a kernel with $n < k$ variables.

Let G_φ be the *variable-clause* incidence graph of φ. That is, G_φ is a bipartite graph with bipartition (X, Y), where X is the set of the variables of φ and Y is the set of clauses of φ. In G_φ there is an edge between a variable $x \in X$ and a clause $c \in Y$ if and only if either x, or its negation, is in c. If there is a matching of X into Y in G_φ, then there is a truth assignment satisfying at least $|X|$ clauses: we can set each variable in X in such a way that the clause matched to it becomes satisfied. Thus at least $|X|$ clauses are satisfied. Hence, in this case, if $k \leq |X|$, then (φ, k) is a yes-instance. Otherwise,

$k > |X| = n$, and we get the desired kernel. We now show that, if φ has at least $n \geq k$ variables, then we can, in polynomial time, either reduce φ to an equivalent smaller instance, or find an assignment to the variables satisfying at least k clauses (and conclude that we are dealing with a yes-instance).

Suppose φ has at least k variables. Using Hall's theorem and a polynomial-time algorithm computing a maximum-size matching (Theorems 2.12 and 2.13), we can in polynomial time find either a matching of X into Y or an inclusion-wise minimal set $C \subseteq X$ such that $|N(C)| < |C|$. As discussed in the previous paragraph, if we found a matching, then the instance is a yes-instance and we are done. So suppose we found a set C as described. Let H be $N(C)$ and $R = V(G_\varphi) \setminus (C \cup H)$. Clearly, $N(C) \subseteq H$, there are no edges between vertices of C and R and $G[C]$ is an independent set. Select an arbitrary $x \in C$. We have that there is a matching of $C \setminus \{x\}$ into H since $|N(C')| \geq |C'|$ for every $C' \subseteq C \setminus \{x\}$. Since $|C| > |H|$, we have that the matching from $C \setminus \{x\}$ to H is in fact a matching of H into C. Hence (C, H, R) is a crown decomposition of G_φ.

We prove that all clauses in H are satisfied in every assignment satisfying the maximum number of clauses. Indeed, consider any assignment ψ that does not satisfy all clauses in H. Fix any variable $x \in C$. For every variable y in $C \setminus \{x\}$ set the value of y so that the clause in H matched to y is satisfied. Let ψ' be the new assignment obtained from ψ in this manner. Since $N(C) \subseteq H$ and ψ' satisfies all clauses in H, more clauses are satisfied by ψ' than by ψ. Hence ψ cannot be an assignment satisfying the maximum number of clauses.

The argument above shows that (φ, k) is a yes-instance to Maximum Satisfiability if and only if $(\varphi \setminus H, k - |H|)$ is. This gives rise to the following simple reduction.

Reduction MSat.1. Let (φ, k) and H be as above. Then remove H from φ and decrease k by $|H|$. That is, $(\varphi \setminus H, k - |H|)$ is the new instance.

Repeated applications of Reduction MSat.1 and the arguments described above give the desired kernel. This completes the proof of the theorem. □

2.4 Expansion lemma

In the previous subsection, we described crown decomposition techniques based on the classical Hall's theorem. In this section, we introduce a powerful variation of Hall's theorem, which is called the expansion lemma. This lemma captures a certain property of neighborhood sets in graphs and can be used to obtain polynomial kernels for many graph problems. We apply this result to get an $\mathcal{O}(k^2)$ kernel for Feedback Vertex Set in Chapter 9.

A *q-star*, $q \geq 1$, is a graph with $q+1$ vertices, one vertex of degree q, called the *center*, and all other vertices of degree 1 adjacent to the center. Let G be a bipartite graph with vertex bipartition (A, B). For a positive integer q, a set of edges $M \subseteq E(G)$ is called by a *q-expansion of A into B* if

- every vertex of A is incident to exactly q edges of M;
- M saturates exactly $q|A|$ vertices in B.

Let us emphasize that a q-expansion saturates all vertices of A. Also, for every $u, v \in A$, $u \neq v$, the set of vertices E_u adjacent to u by edges of M does not intersect the set of vertices E_v adjacent to v via edges of M, see Fig. 2.2. Thus every vertex $v \in A$ could be thought of as the center of a star with its q leaves in B, with all these $|A|$ stars being vertex-disjoint. Furthermore, a collection of these stars is also a family of q edge-disjoint matchings, each saturating A.

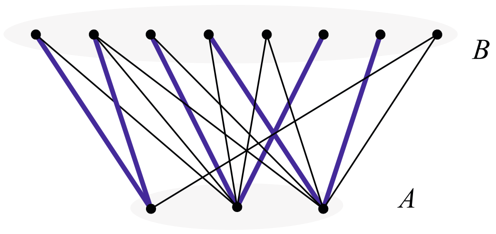


Fig. 2.2: Set A has a 2-expansion into B

Let us recall that, by Hall's theorem (Theorem 2.12), a bipartite graph with bipartition (A, B) has a matching of A into B if and only if $|N(X)| \geq |X|$ for all $X \subseteq A$. The following lemma is an extension of this result.

Lemma 2.17. *Let G be a bipartite graph with bipartition (A, B). Then there is a q-expansion from A into B if and only if $|N(X)| \geq q|X|$ for every $X \subseteq A$. Furthermore, if there is no q-expansion from A into B, then a set $X \subseteq A$ with $|N(X)| < q|X|$ can be found in polynomial time.*

Proof. If A has a q-expansion into B, then trivially $|N(X)| \geq q|X|$ for every $X \subseteq A$.

For the opposite direction, we construct a new bipartite graph G' with bipartition (A', B) from G by adding $(q-1)$ copies of all the vertices in A. For every vertex $v \in A$ all copies of v have the same neighborhood in B as v. We would like to prove that there is a matching M from A' into B in G'. If we prove this, then by identifying the endpoints of M corresponding to the copies of vertices from A, we obtain a q-expansion in G. It suffices to check that the assumptions of Hall's theorem are satisfied in G'. Assume otherwise, that there is a set $X \subseteq A'$ such that $|N_{G'}(X)| < |X|$. Without loss of generality, we can assume that if X contains some copy of a vertex v, then it contains all the copies of v, since including all the remaining copies increases $|X|$ but does not change $|N_{G'}(X)|$. Hence, the set X in A' naturally corresponds to the set X_A of size $|X|/q$ in A, the set of vertices whose copies are in X. But then $|N_G(X_A)| = |N_{G'}(X)| < |X| = q|X_A|$, which is a contradiction. Hence A' has a matching into B and thus A has a q-expansion into B.

For the algorithmic claim, note that, if there is no q-expansion from A into B, then we can use Theorem 2.13 to find a set $X \subseteq A'$ such that $|N_{G'}(X)| < |X|$, and the corresponding set X_A satisfies $|N_G(X_A)| < q|X_A|$. □

Finally, we are ready to prove a lemma analogous to Lemma 2.14.

Lemma 2.18. (Expansion lemma) *Let $q \geq 1$ be a positive integer and G be a bipartite graph with vertex bipartition (A, B) such that*

(i) $|B| \geq q|A|$, and
(ii) there are no isolated vertices in B.

Then there exist nonempty vertex sets $X \subseteq A$ and $Y \subseteq B$ such that

- *there is a q-expansion of X into Y, and*
- *no vertex in Y has a neighbor outside X, that is, $N(Y) \subseteq X$.*

Furthermore, the sets X and Y can be found in time polynomial in the size of G.

Note that the sets X, Y and $V(G) \setminus (X \cup Y)$ form a crown decomposition of G with a stronger property — every vertex of X is not only matched into Y, but there is a q-expansion of X into Y. We proceed with the proof of expansion lemma.

Proof. We proceed recursively, at every step decreasing the cardinality of A. When $|A| = 1$, the claim holds trivially by taking $X = A$ and $Y = B$.

We apply Lemma 2.17 to G. If A has a q-expansion into B, then we are done as we may again take $X = A$ and $Y = B$. Otherwise, we can in polynomial time find a (nonempty) set $Z \subseteq A$ such that $|N(Z)| < q|Z|$. We construct the graph G' by removing Z and $N(Z)$ from G. We claim that G' satisfies the assumptions of the lemma. Indeed, because we removed less than q times more vertices from B than from A, we have that (i) holds for G'. Moreover, every vertex from $B \setminus N(Z)$ has no neighbor in Z, and thus (ii) also holds for G'. Note that $Z \neq A$, because otherwise $N(A) = B$ (there are no isolated vertices in B) and $|B| \geq q|A|$. Hence, we recurse on the graph G' with bipartition $(A \setminus Z, B \setminus N(Z))$, obtaining nonempty sets $X \subseteq A \setminus Z$ and $Y \subseteq B \setminus N(Z)$ such that there is a q-expansion of X into Y and such that $N_{G'}(Y) \subseteq X$. Because $Y \subseteq B \setminus N(Z)$, we have that no vertex in Y has a neighbor in Z. Hence, $N_{G'}(Y) = N_G(Y) \subseteq X$ and the pair (X, Y) satisfies all the required properties. □

The expansion lemma is useful when the matching saturating the head part H in the crown lemma turns out to be not sufficient for a reduction, and we would like to have a few vertices of the crown C matched to a single vertex of the head H. For example, this is the case for the FEEDBACK VERTEX SET kernel presented in Section 9.1, where we need the case $q = 2$.

2.5 Kernels based on linear programming

In this section we design a $2k$-vertex kernel for VERTEX COVER exploiting the solution to a linear programming formulation of VERTEX COVER.

Many combinatorial problems can be expressed in the language of INTEGER LINEAR PROGRAMMING (ILP). In an INTEGER LINEAR PROGRAMMING instance, we are given a set of integer-valued variables, a set of linear inequalities (called *constraints*) and a linear *cost function*. The goal is to find an (integer) evaluation of the variables that satisfies all constraints, and minimizes or maximizes the value of the cost function.

Let us give an example on how to encode a VERTEX COVER instance (G, k) as an INTEGER LINEAR PROGRAMMING instance. We introduce $n = |V(G)|$ variables, one variable x_v for each vertex $v \in V(G)$. Setting variable x_v to 1 means that v is in the vertex cover, while setting x_v to 0 means that v is not in the vertex cover. To ensure that every edge is covered, we can introduce constraints $x_u + x_v \geq 1$ for every edge $uv \in E(G)$. The size of the vertex cover is given by $\sum_{v \in V(G)} x_v$. In the end, we obtain the following ILP formulation:

$$\begin{array}{lll} \text{minimize} & \sum_{v \in V(G)} x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for every } uv \in E(G), \\ & 0 \leq x_v \leq 1 & \text{for every } v \in V(G), \\ & x_v \in \mathbb{Z} & \text{for every } v \in V(G). \end{array} \tag{2.1}$$

Clearly, the optimal value of (2.1) is at most k if and only if G has a vertex cover of size at most k.

As we have just seen, INTEGER LINEAR PROGRAMMING is at least as hard as VERTEX COVER, so we do not expect it to be polynomial-time solvable. In fact, it is relatively easy to express many NP-hard problems in the language of INTEGER LINEAR PROGRAMMING. In Section 6.2 we discuss FPT algorithms for INTEGER LINEAR PROGRAMMING and their application in proving fixed-parameter tractability of other problems.

Here, we proceed in a different way: we relax the integrality requirement of INTEGER LINEAR PROGRAMMING, which is the main source of the hardness of this problem, to obtain LINEAR PROGRAMMING. That is, in LINEAR

Programming the instance looks exactly the same as in Integer Linear Programming, but the variables are allowed to take arbitrary real values, instead of just integers.

In the case of Vertex Cover, we relax (2.1) by dropping the constraint $x_v \in \mathbb{Z}$ for every $v \in V(G)$. In other words, we obtain the following Linear Programming instance. For a graph G, we call this relaxation LPVC(G).

$$\begin{array}{lll} \text{minimize} & \sum_{v \in V(G)} x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for every } uv \in E(G), \\ & 0 \leq x_v \leq 1 & \text{for every } v \in V(G). \end{array} \tag{2.2}$$

Note that constraints $x_v \leq 1$ can be omitted because every optimal solution of LPVC(G) satisfies these constraints.

Observe that in LPVC(G), a variable x_v can take fractional values in the interval $[0, 1]$, which corresponds to taking "part of the vertex v" into a vertex cover. Consider an example of G being a triangle. A minimum vertex cover of a triangle is of size 2, whereas in LPVC(G) we can take $x_v = \frac{1}{2}$ for every $v \in V(G)$, obtaining a feasible solution of cost $\frac{3}{2}$. Thus, LPVC(G) does not express exactly the Vertex Cover problem on graph G, but its optimum solution can still be useful to learn something about minimum vertex covers in G.

The main source of utility of Linear Programming comes from the fact that Linear Programming can be solved in polynomial time, even in some general cases where there are exponentially many constraints, accessed through an oracle. For this reason, Linear Programming has found abundant applications in approximation algorithms (for more on this topic, we refer to the book of Vazirani [427]). In this section, we use LP to design a small kernel for Vertex Cover. In Section 3.4, we will use LPVC(G) to obtain an FPT branching algorithm for Vertex Cover.

Let us now have a closer look at the relaxation LPVC(G). Fix an optimal solution $(x_v)_{v \in V(G)}$ of LPVC(G). In this solution the variables corresponding to vertices of G take values in the interval $[0, 1]$. We partition $V(G)$ according to these values into three sets as follows.

- $V_0 = \{v \in V(G) \ : \ x_v < \frac{1}{2}\}$,
- $V_{\frac{1}{2}} = \{v \in V(G) \ : \ x_v = \frac{1}{2}\}$,
- $V_1 = \{v \in V(G) \ : \ x_v > \frac{1}{2}\}$.

Theorem 2.19 (Nemhauser-Trotter theorem). *There is a minimum vertex cover S of G such that*

$$V_1 \subseteq S \subseteq V_1 \cup V_{\frac{1}{2}}.$$

Proof. Let $S^* \subseteq V(G)$ be a minimum vertex cover of G. Define

$$S = (S^* \setminus V_0) \cup V_1.$$

By the constraints of (2.2), every vertex of V_0 can have a neighbor only in V_1 and thus S is also a vertex cover of G. Moreover, $V_1 \subseteq S \subseteq V_1 \cup V_{\frac{1}{2}}$. It suffices to show that S is a *minimum* vertex cover. Assume the contrary, i.e., $|S| > |S^*|$. Since $|S| = |S^*| - |V_0 \cap S^*| + |V_1 \setminus S^*|$ we infer that

$$|V_0 \cap S^*| < |V_1 \setminus S^*|. \tag{2.3}$$

Let us define

$$\varepsilon = \min\{|x_v - \tfrac{1}{2}| \ : \ v \in V_0 \cup V_1\}.$$

We decrease the fractional values of vertices from $V_1 \setminus S^*$ by ε and increase the values of vertices from $V_0 \cap S^*$ by ε. In other words, we define a vector $(y_v)_{v \in V(G)}$ as

$$y_v = \begin{cases} x_v - \varepsilon & \text{if } v \in V_1 \setminus S^*, \\ x_v + \varepsilon & \text{if } v \in V_0 \cap S^*, \\ x_v & \text{otherwise.} \end{cases}$$

Note that $\varepsilon > 0$, because otherwise $V_0 = V_1 = \emptyset$, a contradiction with (2.3). This, together with (2.3), implies that

$$\sum_{v \in V(G)} y_v < \sum_{v \in V(G)} x_v. \tag{2.4}$$

Now we show that $(y_v)_{v \in V(G)}$ is a feasible solution, i.e., it satisfies the constraints of LPVC(G). Since $(x_v)_{v \in V(G)}$ is a feasible solution, by the definition of ε we get $0 \le y_v \le 1$ for every $v \in V(G)$. Consider an arbitrary edge $uv \in E(G)$. If none of the endpoints of uv belong to $V_1 \setminus S^*$, then both $y_u \ge x_u$ and $y_v \ge x_v$, so $y_u + y_v \ge x_u + x_v \ge 1$. Otherwise, by symmetry we can assume that $u \in V_1 \setminus S^*$, and hence $y_u = x_u - \varepsilon$. Because S^* is a vertex cover, we have that $v \in S^*$. If $v \in V_0 \cap S^*$, then

$$y_u + y_v = x_u - \varepsilon + x_v + \varepsilon = x_u + x_v \ge 1.$$

Otherwise, $v \in (V_{\frac{1}{2}} \cup V_1) \cap S^*$. Then $y_v \ge x_v \ge \frac{1}{2}$. Note also that $x_u - \varepsilon \ge \frac{1}{2}$ by the definition of ε. It follows that

$$y_u + y_v = x_u - \varepsilon + y_v \ge \frac{1}{2} + \frac{1}{2} = 1.$$

Thus $(y_v)_{v \in V(G)}$ is a feasible solution of LPVC(G) and hence (2.4) contradicts the optimality of $(x_v)_{v \in V(G)}$. □

Theorem 2.19 allows us to use the following reduction rule.

Reduction VC.4. Let $(x_v)_{v \in V(G)}$ be an optimum solution to LPVC(G) in a Vertex Cover instance (G, k) and let V_0, V_1 and $V_{\frac{1}{2}}$ be defined as above. If $\sum_{v \in V(G)} x_v > k$, then conclude that we are dealing with a no-instance. Otherwise, greedily take into the vertex cover the vertices of V_1. That is, delete all vertices of $V_0 \cup V_1$, and decrease k by $|V_1|$.

Let us now formally verify the safeness of Reduction VC.4.

Lemma 2.20. *Reduction VC.4 is safe.*

Proof. Clearly, if (G, k) is a yes-instance, then an optimum solution to LPVC(G) is of cost at most k. This proves the correctness of the step if we conclude that (G, k) is a no-instance.

Let $G' = G - (V_0 \cup V_1) = G[V_{\frac{1}{2}}]$ and $k' = k - |V_1|$. We claim that (G, k) is a yes-instance of Vertex Cover if and only if (G', k') is. By Theorem 2.19, we know that G has a vertex cover S of size at most k such that $V_1 \subseteq S \subseteq V_1 \cup V_{\frac{1}{2}}$. Then $S' = S \cap V_{\frac{1}{2}}$ is a vertex cover in G' and the size of S' is at most $k - |V_1| = k'$.

For the opposite direction, let S' be a vertex cover in G'. For every solution of LPVC(G), every edge with an endpoint from V_0 should have an endpoint in V_1. Hence, $S = S' \cup V_1$ is a vertex cover in G and the size of this vertex cover is at most $k' + |V_1| = k$. □

Reduction VC.4 leads to the following kernel for Vertex Cover.

Theorem 2.21. Vertex Cover *admits a kernel with at most* $2k$ *vertices.*

Proof. Let (G, k) be an instance of Vertex Cover. We solve LPVC(G) in polynomial time, and apply Reduction VC.4 to the obtained solution $(x_v)_{v \in V(G)}$, either concluding that we are dealing with a no-instance or obtaining an instance (G', k'). Lemma 2.20 guarantees the safeness of the reduction. For the size bound, observe that

$$|V(G')| = |V_{\frac{1}{2}}| = \sum_{v \in V_{\frac{1}{2}}} 2x_v \leq 2 \sum_{v \in V(G)} x_v \leq 2k.$$

□

While it is possible to solve linear programs in polynomial time, usually such solutions are less efficient than combinatorial algorithms. The specific structure of the LP-relaxation of the vertex cover problem (2.2) allows us to solve it by reducing to the problem of finding a maximum-size matching in a bipartite graph.

Lemma 2.22. *For a graph G with n vertices and m edges, the optimal (fractional) solution to the linear program* LPVC(G) *can be found in time* $\mathcal{O}(m\sqrt{n})$.

Proof. We reduce the problem of solving LPVC(G) to a problem of finding a minimum-size vertex cover in the following bipartite graph H. Its vertex set consists of two copies V_1 and V_2 of the vertex set of G. Thus, every vertex $v \in V(G)$ has two copies $v_1 \in V_1$ and $v_2 \in V_2$ in H. For every edge $uv \in E(H)$, we have edges u_1v_2 and v_1u_2 in H.

Using the Hopcroft-Karp algorithm (Theorem 2.13), we can find a minimum vertex cover S of H in time $\mathcal{O}(m\sqrt{n})$. We define a vector $(x_v)_{v\in V(G)}$ as follows: if both vertices v_1 and v_2 are in S, then $x_v = 1$. If exactly one of the vertices v_1 and v_2 is in S, we put $x_v = \frac{1}{2}$. We put $x_v = 0$ if none of the vertices v_1 and v_2 are in S. Thus

$$\sum_{v\in V(G)} x_v = \frac{|S|}{2}.$$

Since S is a vertex cover in H, we have that for every edge $uv \in E(G)$ at least two vertices from $\{u_1, u_2, v_1, v_2\}$ should be in S. Thus $x_u + x_v \geq 1$ and vector $(x_v)_{v\in V(G)}$ satisfies the constraints of LPVC(G).

To show that $(x_v)_{v\in V(G)}$ is an optimal solution of LPVC(G), we argue as follows. Let $(y_v)_{v\in V(G)}$ be an optimal solution of LPVC(G). For every vertex v_i, $i \in \{1,2\}$, of H, we assign the weight $\mathbf{w}(v_i) = y_v$. This weight assignment is a fractional vertex cover of H, i.e., for every edge $v_1u_2 \in E(H)$, $\mathbf{w}(v_1) + \mathbf{w}(u_2) \geq 1$. We have that

$$\sum_{v\in V(G)} y_v = \frac{1}{2}\sum_{v\in V(G)} (\mathbf{w}(v_1) + \mathbf{w}(v_2)).$$

On the other hand, the value $\sum_{v\in V(H)} \mathbf{w}(v)$ of any fractional solution of LPVC(H) is at least the size of a maximum matching M in H. A reader familiar with linear programming can see that this follows from weak duality; we also ask you to verify this fact in Exercise 2.24.

By Kőnig's theorem (Theorem 2.11), $|M| = |S|$. Hence

$$\sum_{v\in V(G)} y_v = \frac{1}{2}\sum_{v\in V(G)} (\mathbf{w}(v_1) + \mathbf{w}(v_2)) = \frac{1}{2}\sum_{v\in V(H)} \mathbf{w}(v) \geq \frac{|S|}{2} = \sum_{v\in V(G)} x_v.$$

Thus $(x_v)_{v\in V(G)}$ is an optimal solution of LPVC(G). □

We immediately obtain the following.

Corollary 2.23. *For a graph G with n vertices and m edges, the kernel of Theorem 2.21 can be found in time $\mathcal{O}(m\sqrt{n})$.*

The following proposition is another interesting consequence of the proof of Lemma 2.22.

Proposition 2.24. *Let G be a graph on n vertices and m edges. Then* LPVC(G) *has a half-integral optimal solution, i.e., all variables have values in the set $\{0, \frac{1}{2}, 1\}$. Furthermore, we can find a half-integral optimal solution in time $\mathcal{O}(m\sqrt{n})$.*

In short, we have proved properties of LPVC(G). There exists a half-integral optimal solution $(x_v)_{v\in V(G)}$ to LPVC(G), and it can be found efficiently. We can look at this solution as a partition of $V(G)$ into parts V_0, $V_{\frac{1}{2}}$, and V_1 with the following message: greedily take V_1 into a solution, do not take any vertex of V_0 into a solution, and in $V_{\frac{1}{2}}$, we do not know what to do and that is the hard part of the problem. However, as an optimum solution pays $\frac{1}{2}$ for every vertex of $V_{\frac{1}{2}}$, the hard part — the kernel of the problem — cannot have more than $2k$ vertices.

2.6 Sunflower lemma

In this section we introduce a classical result of Erdős and Rado and show some of its applications in kernelization. In the literature it is known as the sunflower lemma or as the Erdős-Rado lemma. We first define the terminology used in the statement of the lemma. A *sunflower* with k *petals* and a *core* Y is a collection of sets $S_1, \ldots, S_k$ such that $S_i \cap S_j = Y$ for all $i \neq j$; the sets $S_i \setminus Y$ are petals and we require *none of them to be empty*. Note that a family of pairwise disjoint sets is a sunflower (with an empty core).

Theorem 2.25 (Sunflower lemma). *Let $\mathcal{A}$ be a family of sets (without duplicates) over a universe U, such that each set in $\mathcal{A}$ has cardinality exactly d. If $|\mathcal{A}| > d!(k-1)^d$, then $\mathcal{A}$ contains a sunflower with k petals and such a sunflower can be computed in time polynomial in $|\mathcal{A}|$, $|U|$, and k.*

Proof. We prove the theorem by induction on d. For $d = 1$, i.e., for a family of singletons, the statement trivially holds. Let $d \geq 2$ and let $\mathcal{A}$ be a family of sets of cardinality at most d over a universe U such that $|\mathcal{A}| > d!(k-1)^d$.

Let $\mathcal{G} = \{S_1, \ldots, S_\ell\} \subseteq \mathcal{A}$ be an inclusion-wise maximal family of pairwise disjoint sets in $\mathcal{A}$. If $\ell \geq k$ then $\mathcal{G}$ is a sunflower with at least k petals. Thus we assume that $\ell < k$. Let $S = \bigcup_{i=1}^{\ell} S_i$. Then $|S| \leq d(k-1)$. Because $\mathcal{G}$ is maximal, every set $A \in \mathcal{A}$ intersects at least one set from $\mathcal{G}$, i.e., $A \cap S \neq \emptyset$. Therefore, there is an element $u \in U$ contained in at least

$$\frac{|\mathcal{A}|}{|S|} > \frac{d!(k-1)^d}{d(k-1)} = (d-1)!(k-1)^{d-1}$$

sets from $\mathcal{A}$. We take all sets of $\mathcal{A}$ containing such an element u, and construct a family $\mathcal{A}'$ of sets of cardinality $d-1$ by removing from each set the element u. Because $|\mathcal{A}'| > (d-1)!(k-1)^{d-1}$, by the induction hypothesis, $\mathcal{A}'$ contains a sunflower $\{S'_1, \ldots, S'_k\}$ with k petals. Then $\{S'_1 \cup \{u\}, \ldots, S'_k \cup \{u\}\}$ is a sunflower in $\mathcal{A}$ with k petals.

The proof can be easily transformed into a polynomial-time algorithm, as follows. Greedily select a maximal set of pairwise disjoint sets. If the size of this set is at least k, then return this set. Otherwise, find an element u contained in the maximum number of sets in $\mathcal{A}$, and call the algorithm recursively on sets of cardinality $d-1$, obtained from deleting u from the sets containing u. □

2.6.1 d-Hitting Set

As an application of the sunflower lemma, we give a kernel for d-Hitting Set. In this problem, we are given a family $\mathcal{A}$ of sets over a universe U, where each set in the family has cardinality at most d, and a positive integer k. The objective is to decide whether there is a subset $H \subseteq U$ of size at most k such that H contains at least one element from each set in $\mathcal{A}$.

Theorem 2.26. d-Hitting Set *admits a kernel with at most* $d!k^d$ *sets and at most* $d!k^d \cdot d^2$ *elements.*

Proof. The crucial observation is that if $\mathcal{A}$ contains a sunflower

$$S = \{S_1, \dots, S_{k+1}\}$$

of cardinality $k+1$, then every hitting set H of $\mathcal{A}$ of cardinality at most k intersects the core Y of the sunflower S. Indeed, if H does not intersect Y, it should intersect each of the $k+1$ disjoint petals $S_i \setminus Y$. This leads to the following reduction rule.

Reduction HS.1. Let $(U, \mathcal{A}, k)$ be an instance of d-Hitting Set and assume that $\mathcal{A}$ contains a sunflower $S = \{S_1, \dots, S_{k+1}\}$ of cardinality $k+1$ with core Y. Then return $(U', \mathcal{A}', k)$, where $\mathcal{A}' = (\mathcal{A} \setminus S) \cup \{Y\}$ is obtained from $\mathcal{A}$ by deleting all sets $\{S_1, \dots, S_{k+1}\}$ and by adding a new set Y and $U' = \bigcup_{X \in \mathcal{A}'} X$.

Note that when deleting sets we do not delete the elements contained in these sets but only those which do not belong to any set. Then the instances $(U, \mathcal{A}, k)$ and $(U', \mathcal{A}', k)$ are equivalent, i.e. $(U, \mathcal{A})$ contains a hitting set of size k if and only if $(U, \mathcal{A}')$ does.

The kernelization algorithm is as follows. If for some $d' \in \{1, \dots, d\}$ the number of sets in $\mathcal{A}$ of size exactly d' is more than $d'!k^{d'}$, then the kernelization algorithm applies the sunflower lemma to find a sunflower of size $k+1$, and applies Reduction HS.1 on this sunflower. It applies this procedure exhaustively, and obtains a new family of sets $\mathcal{A}'$ of size at most $d!k^d \cdot d$. If $\emptyset \in \mathcal{A}'$ (that is, at some point a sunflower with an empty core has been discovered), then the algorithm concludes that there is no hitting set of size at most k and returns that the given instance is a no-instance. Otherwise, every set contains at most d elements, and thus the number of elements in the kernel is at most $d!k^d \cdot d^2$. □

Exercises

2.1 (✎). Prove Lemma 2.5: A digraph is acyclic if and only if it is possible to order its vertices in such a way such that for every arc (u, v), we have $u < v$.

2.2 (✎). Give an example of a feedback arc set F in a tournament G, such that $G \circledast F$ is not acyclic.

2.3 (✎). Show that Reductions ECC.1, ECC.2, and ECC.3 are safe.

2.4 (✎). In the Point Line Cover problem, we are given a set of n points in the plane and an integer k, and the goal is to check if there exists a set of k lines on the plane that contain all the input points. Show a kernel for this problem with $\mathcal{O}(k^2)$ points.

2.5. A graph G is a *cluster graph* if every connected component of G is a clique. In the Cluster Editing problem, we are given as input a graph G and an integer k, and the objective is to check whether one can edit (add or delete) at most k edges in G to obtain a cluster graph. That is, we look for a set $F \subseteq \binom{V(G)}{2}$ of size at most k, such that the graph $(V(G), (E(G) \setminus F) \cup (F \setminus E(G)))$ is a cluster graph.

1. Show that a graph G is a cluster graph if and only if it does not have an induced path on three vertices (sequence of three vertices u, v, w such that uv and vw are edges and $uw \notin E(G)$).
2. Show a kernel for Cluster Editing with $\mathcal{O}(k^2)$ vertices.

2.6. In the Set Splitting problem, we are given a family of sets $\mathcal{F}$ over a universe U and a positive integer k, and the goal is to test whether there exists a coloring of U with two colors such that at least k sets in $\mathcal{F}$ are nonmonochromatic (that is, they contain vertices of both colors). Show that the problem admits a kernel with at most $2k$ sets and $\mathcal{O}(k^2)$ universe size.

2.7. In the Minimum Maximal Matching problem, we are given an undirected graph G and a positive integer k, and the objective is to decide whether there exists a maximal matching in G on at most k edges. Obtain a polynomial kernel for the problem (parameterized by k).

2.8. In the Min-Ones-2-SAT problem, we are given a 2-CNF formula ϕ and an integer k, and the objective is to decide whether there exists a satisfying assignment for ϕ with at most k variables set to true. Show that Min-Ones-2-SAT admits a polynomial kernel.

2.9. In the d-Bounded-Degree Deletion problem, we are given an undirected graph G and a positive integer k, and the task is to find at most k vertices whose removal decreases the maximum vertex degree of the graph to at most d. Obtain a kernel of size polynomial in k and d for the problem. (Observe that Vertex Cover is the case of $d = 0$.)

2.10. Show a kernel with $\mathcal{O}(k^2)$ vertices for the following problem: given a graph G and an integer k, check if G contains a subgraph with exactly k edges, whose vertices are all of odd degree in the subgraph.

2.11. A set of vertices D in an undirected graph G is called a *dominating set* if $N[D] = V(G)$. In the Dominating Set problem, we are given an undirected graph G and a positive integer k, and the objective is to test whether there exists a dominating set of size at most k. Show that Dominating Set admits a polynomial kernel on graphs where the length of the shortest cycle is at least 5. (What would you do with vertices with degree more than k? Note that unlike for the Vertex Cover problem, you cannot delete a vertex once you pick it in the solution.)

2.12. Show that FEEDBACK VERTEX SET admits a kernel with $\mathcal{O}(k)$ vertices on undirected regular graphs.

2.13. We say that an n-vertex digraph is *well-spread* if every vertex has indegree at least $\sqrt{n}$. Show that DIRECTED FEEDBACK ARC SET, restricted to well-spread digraphs, is FPT by obtaining a polynomial kernel for this problem. Does the problem remain FPT if we replace the lower bound on indegree by any monotonically increasing function of n (like $\log n$ or $\log\log\log n$)? Does the assertion hold if we replace indegree with outdegree? What about DIRECTED FEEDBACK VERTEX SET?

2.14. In the CONNECTED VERTEX COVER problem, we are given an undirected graph G and a positive integer k, and the objective is to decide whether there exists a vertex cover C of G such that $|C| \leq k$ and $G[C]$ is connected.

1. Explain where the kernelization procedure described in Theorem 2.4 for VERTEX COVER breaks down for the CONNECTED VERTEX COVER problem.
2. Show that the problem admits a kernel with at most $2^k + \mathcal{O}(k^2)$ vertices.
3. Show that if the input graph G does not contain a cycle of length 4 as a subgraph, then the problem admits a kernel with at most $\mathcal{O}(k^2)$ vertices.

2.15 (☠). Extend the argument of the previous exercise to show that, for every fixed $d \geq 2$, CONNECTED VERTEX COVER admits a kernel of size $\mathcal{O}(k^d)$ if restricted to graphs that do not contain the biclique $K_{d,d}$ as a subgraph.

2.16 (☠). A graph G is chordal if it contains no induced cycles of length more than 3, that is, every cycle of length at least 4 has a chord. In the CHORDAL COMPLETION problem, we are given an undirected graph G and a positive integer k, and the objective is to decide whether we can add at most k edges to G so that it becomes a chordal graph. Obtain a polynomial kernel for CHORDAL COMPLETION (parameterized by k).

2.17 (☠). In the EDGE DISJOINT CYCLE PACKING problem, we are given an undirected graph G and a positive integer k, and the objective is to test whether G has k pairwise edge disjoint cycles. Obtain a polynomial kernel for EDGE DISJOINT CYCLE PACKING (parameterized by k).

2.18 (☠). A *bisection* of a graph G with an even number of vertices is a partition of $V(G)$ into V_1 and V_2 such that $|V_1| = |V_2|$. The size of (V_1, V_2) is the number of edges with one endpoint in V_1 and the other in V_2. In the MAXIMUM BISECTION problem, we are given an undirected graph G with an even number of vertices and a positive integer k, and the objective is to test whether there exists a bisection of size at least k.

1. Show that every graph with m edges has a bisection of size at least $\lceil \frac{m}{2} \rceil$. Use this to show that MAXIMUM BISECTION admits a kernel with $2k$ edges.
2. Consider the following "above guarantee" variant of MAXIMUM BISECTION, where we are given an undirected graph G and a positive integer k, but the objective is to test whether there exists a bisection of size at least $\lceil \frac{m}{2} \rceil + k$. Show that the problem admits a kernel with $\mathcal{O}(k^2)$ vertices and $\mathcal{O}(k^3)$ edges.

2.19 (✐). Byteland, a country of area exactly n square miles, has been divided by the government into n regions, each of area exactly one square mile. Meanwhile, the army of Byteland divided its area into n military zones, each of area again exactly one square mile. Show that one can build n airports in Byteland, such that each region and each military zone contains one airport.

2.20. A magician and his assistant are performing the following magic trick. A volunteer from the audience picks five cards from a standard deck of 52 cards and then passes the deck to the assistant. The assistant shows to the magician, one by one in some order, four cards from the chosen set of five cards. Then, the magician guesses the remaining fifth card. Show that this magic trick can be performed without any help of magic.

2.21. Prove the second claim of Theorem 2.13.

2.22. In the Dual-Coloring problem, we are given an undirected graph G on n vertices and a positive integer k, and the objective is to test whether there exists a proper coloring of G with at most $n-k$ colors. Obtain a kernel with $\mathcal{O}(k)$ vertices for this problem using crown decomposition.

2.23 (☠). In the Max-Internal Spanning Tree problem, we are given an undirected graph G and a positive integer k, and the objective is to test whether there exists a spanning tree with at least k internal vertices. Obtain a kernel with $\mathcal{O}(k)$ vertices for Max-Internal Spanning Tree.

2.24 (✎). Let G be an undirected graph, let $(x_v)_{v \in V(G)}$ be any feasible solution to LPVC(G), and let M be a matching in G. Prove that $|M| \leq \sum_{v \in V(G)} x_v$.

2.25 (☠). Let G be a graph and let $(x_v)_{v \in V(G)}$ be an optimum solution to LPVC(G) (not necessarily a half-integral one). Define a vector $(y_v)_{v \in V(G)}$ as follows:

$$y_v = \begin{cases} 0 & \text{if } x_v < \frac{1}{2} \\ \frac{1}{2} & \text{if } x_v = \frac{1}{2} \\ 1 & \text{if } x_v > \frac{1}{2}. \end{cases}$$

Show that $(y_v)_{v \in V(G)}$ is also an optimum solution to LPVC(G).

2.26 (☠). In the Min-Ones-2-SAT, we are given a CNF formula, where every clause has exactly two literals, and an integer k, and the goal is to check if there exists a satisfying assignment of the input formula with at most k variables set to true. Show a kernel for this problem with at most $2k$ variables.

2.27 (✎). Consider a restriction of d-Hitting Set, called Ed-Hitting Set, where we require every set in the input family $\mathcal{A}$ to be of size *exactly* d. Show that this problem is not easier than the original d-Hitting Set problem, by showing how to transform a d-Hitting Set instance into an equivalent Ed-Hitting Set instance without changing the number of sets.

2.28. Show a kernel with at most $f(d)k^d$ sets (for some computable function f) for the Ed-Hitting Set problem, defined in the previous exercise.

2.29. In the d-Set Packing problem, we are given a family $\mathcal{A}$ of sets over a universe U, where each set in the family has cardinality at most d, and a positive integer k. The objective is to decide whether there are sets $S_1, \ldots, S_k \in \mathcal{A}$ that are pairwise disjoint. Use the sunflower lemma to obtain a kernel for d-Set Packing with $f(d)k^d$ sets, for some computable function d.

2.30. Consider a restriction of d-Set Packing, called Ed-Set Packing, where we require every set in the input family $\mathcal{A}$ to be of size *exactly* d. Show that this problem is not easier than the original d-Set Packing problem, by showing how to transform a d-Set Packing instance into an equivalent Ed-Set Packing instance without changing the number of sets.

2.31. A *split graph* is a graph in which the vertices can be partitioned into a clique and an independent set. In the Vertex Disjoint Paths problem, we are given an undirected graph G and k pairs of vertices (s_i, t_i), $i \in \{1, \ldots, k\}$, and the objective is to decide whether there exists paths P_i joining s_i to t_i such that these paths are pairwise vertex disjoint. Show that Vertex Disjoint Paths admits a polynomial kernel on split graphs (when parameterized by k).

2.32. Consider now the Vertex Disjoint Paths problem, defined in the previous exercise, restricted, for a fixed integer $d \geq 3$, to a class of graphs that does not contain a d-vertex path as an induced subgraph. Show that in this class the Vertex Disjoint Paths problem admits a kernel with $\mathcal{O}(k^{d-1})$ vertices and edges.

2.33. In the Cluster Vertex Deletion problem, we are given as input a graph G and a positive integer k, and the objective is to check whether there exists a set $S \subseteq V(G)$ of size at most k such that $G - S$ is a cluster graph. Show a kernel for Cluster Vertex Deletion with $\mathcal{O}(k^3)$ vertices.

2.34. An undirected graph G is called *perfect* if for every induced subgraph H of G, the size of the largest clique in H is same as the chromatic number of H. In the Odd Cycle Transversal problem, we are given an undirected graph G and a positive integer k, and the objective is to find at most k vertices whose removal makes the resulting graph bipartite. Obtain a kernel with $\mathcal{O}(k^2)$ vertices for Odd Cycle Transversal on perfect graphs.

2.35. In the Split Vertex Deletion problem, we are given an undirected graph G and a positive integer k and the objective is to test whether there exists a set $S \subseteq V(G)$ of size at most k such that $G - S$ is a split graph (see Exercise 2.31 for the definition).

1. Show that a graph is split if and only if it has no induced subgraph isomorphic to one of the following three graphs: a cycle on four or five vertices, or a pair of disjoint edges.
2. Give a kernel with $\mathcal{O}(k^5)$ vertices for Split Vertex Deletion.

2.36 (☠). In the Split Edge Deletion problem, we are given an undirected graph G and a positive integer k, and the objective is to test whether G can be transformed into a split graph by deleting at most k edges. Obtain a polynomial kernel for this problem (parameterized by k).

2.37 (☠). In the Ramsey problem, we are given as input a graph G and an integer k, and the objective is to test whether there exists in G an independent set or a clique of size at least k. Show that Ramsey is FPT.

2.38 (☠). A directed graph D is called *oriented* if there is no directed cycle of length at most 2. Show that the problem of testing whether an oriented digraph contains an induced directed acyclic subgraph on at least k vertices is FPT.

Hints

2.4 Consider the following reduction rule: if there exists a line that contains more than k input points, delete the points on this line and decrease k by 1.

2.5 Consider the following natural reduction rules:

1. delete a vertex that is not a part of any P_3 (induced path on three vertices);
2. if an edge uv is contained in at least $k+1$ different P_3s, then delete uv;
3. if a non-edge uv is contained in at least $k+1$ different P_3s, then add uv.

Show that, after exhaustive application of these rules, a yes-instance has $\mathcal{O}(k^2)$ vertices.

2.6 First, observe that one can discard any set in $\mathcal{F}$ that is of size at most 1. Second, observe that if every set in $\mathcal{F}$ is of size at least 2, then a random coloring of U has at least

$|\mathcal{F}|/2$ nonmonochromatic sets on average, and an instance with $|\mathcal{F}| \geq 2k$ is a yes-instance. Moreover, observe that if we are dealing with a yes-instance and $F \in \mathcal{F}$ is of size at least $2k$, then we can always tweak the solution coloring to color F nonmonochromatically: fix two differently colored vertices for $k-1$ nonmonochromatic sets in the solution, and color some two uncolored vertices of F with different colors. Use this observation to design a reduction rule that handles large sets in $\mathcal{F}$.

2.7 Observe that the endpoints of the matching in question form a vertex cover of the input graph. In particular, every vertex of degree larger than $2k$ needs to be an endpoint of a solution matching. Let X be the set of these large-degree vertices. Argue, similarly as in the case of $\mathcal{O}(k^2)$ kernel for Vertex Cover, that in a yes-instance, $G \setminus X$ has only few edges. Design a reduction rule to reduce the number of isolated vertices of $G \setminus X$.

2.8 Proceed similarly as in the $\mathcal{O}(k^2)$ kernel for Vertex Cover.

2.9 Proceed similarly as in the case of Vertex Cover. Argue that the vertices of degree larger than $d+k$ need to be included in the solution. Moreover, observe that you may delete isolated vertices, as well as edges connecting two vertices of degree at most d. Argue that, if no rule is applicable, then a yes-instance is of size bounded polynomially in $d+k$.

2.10 The important observation is that a matching of size k is a good subgraph. Hence, we may restrict ourselves to the case where we are additionally given a vertex cover X of the input graph of size at most $2k$. Moreover, assume that X is inclusion-wise minimal. To conclude, prove that, if a vertex $v \in X$ has at least k neighbors in $V(G) \setminus X$, then (G,k) is a yes-instance.

2.11 The main observation is that, since there is no 3-cycle nor 4-cycle in the graph, if $x, y \in N(v)$, then only v can dominate both x and y at once. In particular, every vertex of degree larger than k needs to be included in the solution.

However, you cannot easily delete such a vertex. Instead, mark it as "obligatory" and mark its neighbors as "dominated". Note now that you can delete a "dominated" vertex, as long as it has no unmarked neighbor and its deletion does not drop the degree of an "obligatory" vertex to k.

Prove that, in a yes-instance, if no rule is applicable, then the size is bounded polynomially in k. To this end, show that

1. any vertex can dominate at most k unmarked vertices, and, consequently, there are at most k^2 unmarked vertices;
2. there are at most k "obligatory" vertices;
3. every remaining "dominated" vertex can be charged to one unmarked or obligatory vertex in a manner that each unmarked or obligatory vertex is charged at most $k+1$ times.

2.12 Let (G,k) be a Feedback Vertex Set instance and assume G is d-regular. If $d \leq 2$, then solve (G,k) in polynomial time. Otherwise, observe that G has $dn/2$ edges and, if (G,k) is a yes-instance and X is a feedback vertex set of G of size at most k, then at most dk edges of G are incident to X and $G-X$ contains less than $n-k$ edges (since it is a forest). Consequently, $dn/2 \leq dk + n - k$, which gives $n = \mathcal{O}(k)$ for $d \geq 3$.

2.13 Show, using greedy arguments, that if every vertex in a digraph G has indegree at least d, then G contains d pairwise edge-disjoint cycles.

For the vertex-deletion variant, design a simple reduction that boosts up the indegree of every vertex without actually changing anything in the solution space.

2.14 Let X be the set of vertices of G of degree larger than k. Clearly, any connected vertex cover of G of size at most k needs to contain X. Moreover, as in the case of Vertex Cover, in a yes-instance there are only $\mathcal{O}(k^2)$ edges in $G-X$. However, we cannot easily discard the isolated vertices of $G-X$, as they may be used to make the solution connected.

To obtain an exponential kernel, note that in a yes-instance, $|X| \leq k$, and if we have two vertices u, v that are isolated in $G - X$, and $N_G(u) = N_G(v)$ (note that $N_G(u) \subseteq X$ for every u that is isolated in $G - X$), then we need only one of the vertices u, v in a Connected Vertex Cover solution. Hence, in a kernel, we need to keep:

1. $G[X]$, and all edges and non-isolated vertices of $G - X$;
2. for every $x \in X$, some $k + 1$ neighbors of x;
3. for every $Y \subseteq X$, one vertex u that is isolated in $G - X$ and $N_G(u) = Y$ (if there exists any such vertex).

For the last part of the exercise, note that in the presence of this assumption, no two vertices of X share more than one neighbor and, consequently, there are only $\mathcal{O}(|X|^2)$ sets $Y \subseteq X$ for which there exist $u \notin X$ with $N_G(u) = Y$.

2.15 We repeat the argument of the previous exercise, and bound the number of sets $Y \subseteq X$ for which we need to keep a vertex $u \in V(G) \setminus X$ with $N_G(u) = Y$. First, there are $\mathcal{O}(d|X|^{d-1})$ sets Y of size smaller than d. Second, charge every set Y of size at least d to one of its subset of size d. Since G does not contain $K_{d,d}$ as a subgraph, every subset X of size d is charged less than d times. Consequently, there are at most $(d-1)\binom{|X|}{d}$ vertices $u \in V(G) \setminus X$ such that $N_G(u) \subseteq X$ and $|N_G(u)| \geq d$.

2.16 The main observation is as follows: an induced cycle of length ℓ needs exactly $\ell - 3$ edges to become chordal. In particular, if a graph contains an induced cycle of length larger than $k + 3$, then the input instance is a no-instance, as we need more than k edges to triangulate the cycle in question.

First, prove the safeness of the following two reduction rules:

1. Delete any vertex that is not contained in any induced cycle in G.
2. A vertex x is a *friend* of a non-edge uv, if u, x, v are three consecutive vertices of some induced cycle in G. If $uv \notin E(G)$ has more than $2k$ friends, then add the edge uv and decrease k by one.

Second, consider the following procedure. Initiate A to be the vertex set of any inclusion-wise maximal family of pairwise vertex-disjoint induced cycles of length at most 4 in G. Then, as long as there exists an induced cycle of length at most 4 in G that contains two consecutive vertices in $V(G) \setminus A$, move these two vertices to A. Show, using a charging argument, that, in a yes-instance, the size of A remains $\mathcal{O}(k)$. Conclude that the size of a reduced yes-instance is bounded polynomially in k.

2.17 Design reduction rules that remove vertices of degree at most 2 (you may obtain a multigraph in the process). Prove that every n-vertex multigraph of minimum degree at least 3 has a cycle of length $\mathcal{O}(\log n)$. Use this to show a greedy argument that an n-vertex multigraph of minimum degree 3 has $\Omega(n^{\varepsilon})$ pairwise edge-disjoint cycles for some $\varepsilon > 0$.

2.18 Consider the following argument. Let $|V(G)| = 2n$ and pair the vertices of G arbitrarily: $V(G) = \{x_1, y_1, x_2, y_2, \ldots, x_n, y_n\}$. Consider the bisection (V_1, V_2) where, in each pair (x_i, y_i), one vertex goes to V_1 and the other goes to V_2, where the decision is made uniformly at random and independently of other pairs. Prove that, in expectation, the obtained bisection is of size at least $(m + \ell)/2$, where ℓ is the number of pairs (x_i, y_i) where $x_i y_i \in E(G)$.

Use the arguments in the previous paragraph to show not only the first point of the exercise, but also that the input instance is a yes-instance if it admits a matching of size $2k$. If this is not the case, then let X be the set of endpoints of a maximal matching in G; note that $|X| \leq 4k$.

First, using a variation of the argument of the first paragraph, prove that, if there exists $x \in X$ that has at least $2k$ neighbors and at least $2k$ non-neighbors outside X, then the input instance is a yes-instance. Second, show that in the absence of such a vertex, all but

$\mathcal{O}(k^2)$ vertices of $V(G) \setminus X$ have exactly the same neighborhood in X, and design a way to reduce them.

2.19 Construct the following bipartite graph: on one side there are regions of Byteland, on the second side there are military zones, and a region R is adjacent to a zone Z if $R \cap Z \neq \emptyset$. Show that this graph satisfies the condition of Hall's theorem and, consequently, contains a perfect matching.

2.20 Consider the following bipartite graph: on one side there are all $\binom{52}{5}$ sets of five cards (possibly chosen by the volunteer), and on the other side there are all $52 \cdot 51 \cdot 50 \cdot 49$ tuples of pairwise different four cards (possibly shown by the assistant). A set S is adjacent to a tuple T if all cards of T belong to S. Using Hall's theorem, show that this graph admits a matching saturating the side with all sets of five cards. This matching induces a strategy for the assistant and the magician.

We now show a relatively simple explicit strategy, so that you can impress your friends and perform this trick at some party. In every set of five cards, there are two cards of the same color, say a and b. Moreover, as there are 13 cards of the same color, the cards a and b differ by at most 6, that is, $a+i=b$ or $b+i=a$ for some $1 \leq i \leq 6$, assuming some cyclic order on the cards of the same color. Without loss of generality, assume $a+i=b$. The assistant first shows the card a to the magician. Then, using the remaining three cards, and some fixed total order on the whole deck of cards, the assistant shows the integer i (there are $3! = 6$ permutations of remaining three cards). Consequently, the magician knows the card b by knowing its color (the same as the first card show by the assistant) and the value of the card a and the number i.

2.21 Let M be a maximum matching, which you can find using the Hopcroft-Karp algorithm (the first part of Theorem 2.13). If M saturates V_1, then we are done. Otherwise, pick any $v \in V_1 \setminus V(M)$ (i.e., a vertex $v \in V_1$ that is not an endpoint of an edge of M) and consider all vertices of G that are reachable from v using alternating paths. (A path P is *alternating* if every second edge of P belongs to M.) Show that all vertices from V_1 that are reachable from v using alternating paths form an inclusion-wise minimal set X with $|N(X)| < |X|$.

2.22 Apply the crown lemma to $\bar{G}$, *the edge complement of* G ($\bar{G}$ has vertex set $V(G)$ and $uv \in E(\bar{G})$ if and only if $uv \notin E(G)$) and the parameter $k-1$. If it returns a matching M_0 of size k, then note that one can color the endpoints of each edge of M_0 with the same color, obtaining a coloring of G with $n-k$ colors. Otherwise, design a way to greedily color the head and the crown of the obtained crown decomposition.

2.23 Your main tool is the following variation of the crown lemma: if $V(G)$ is sufficiently large, then you can find either a matching of size $k+1$, or a crown decomposition $V(G) = C \cup H \cup R$, such that $G[H \cup C]$ admits a spanning tree where all vertices of H and $|H|-1$ vertices of C are of degree at least two. Prove it, and use it for the problem in question.

2.24 Observe that for every $uv \in M$ we have $x_u + x_v \geq 1$ and, moreover, all these inequalities for all edges of M contain different variables. In other words,

$$\sum_{v \in V(G)} \mathbf{w}(x_v) \geq \sum_{v \in V(M)} \mathbf{w}(x_v) = \sum_{vu \in M} (\mathbf{w}(x_v) + \mathbf{w}(x_u)) \geq \sum_{vu \in M} 1 = |M|.$$

2.25 Let $V_\delta = \{v \in V(G) : 0 < x_v < \frac{1}{2}\}$ and $V_{1-\delta} = \{v \in V(G) : \frac{1}{2} < x_v < 1\}$. For sufficiently small $\varepsilon > 0$, consider two operations: first, an operation of adding ε to all variables x_v for $v \in V_\delta$ and subtracting ε from x_v for $v \in V_{1-\delta}$, and second, an operation of adding ε to all variables x_v for $v \in V_{1-\delta}$ and subtracting ε from x_v for $v \in V_\delta$. Show that both these operations lead to feasible solutions to LPVC(G), as long as ε is small enough. Conclude that $|V_\delta| = |V_{1-\delta}|$, and that both operations lead to other *optimal* solutions to LPVC(G). Finally, observe that, by repeatedly applying the second operation, one can empty sets V_δ and $V_{1-\delta}$, and reach the vector $(y_v)_{v \in V(G)}$.

2.26 First, design natural reduction rules that enforce the following: for every variable x in the input formula φ, there exists a truth assignment satisfying φ that sets x to false, and a truth assignment satisfying φ that sets x to true. In other words, whenever a value of some variable is fixed in any satisfying assignment, fix this value and propagate it in the formula.

Then, consider the following *closure* of the formula φ: for every two-literal clause C that is satisfied in every truth assignment satisfying φ, add C to φ. Note that testing whether C is such a clause can be done in polynomial time: force two literals in C to be false and check if φ remains satisfiable. Moreover, observe that the sets of satisfying assignments for φ and φ' are equal.

Let φ' be the closure of φ. Consider the following auxiliary graph H: $V(H)$ is the set of variables of φ', and $xy \in E(H)$ iff the clause $x \vee y$ belongs to φ'. Clearly, if we take any truth assignment ψ satisfying φ, then $\psi^{-1}(\top)$ is a vertex cover of H. A somewhat surprising fact is that a partial converse is true: for every inclusion-wise minimal vertex cover X of H, the assignment ψ defined as $\psi(x) = \top$ if and only if $x \in X$ satisfies φ' (equivalently, φ). Note that such a claim would solve the exercise: we can apply the LP-based kernelization algorithm to Vertex Cover instance (H, k), and translate the reductions it makes back to the formula φ.

Below we prove the aforementioned claim in full detail. We encourage you to try to prove it on your own before reading.

Let X be a minimal vertex cover of H, and let ψ be defined as above. Take any clause C in φ' and consider three cases. If $C = x \vee y$, then $xy \in E(H)$, and, consequently, either x or y belongs to X. It follows from the definition of ψ that $\psi(x) = \top$ or $\psi(y) = \top$, and ψ satisfies C.

In the second case, $C = x \vee \neg y$. For a contradiction, assume that ψ does not satisfy C and, consequently, $x \notin X$ and $y \in X$. Since X is a minimal vertex cover, there exists $z \in N_H(y)$ such that $z \notin X$ and the clause $C' = y \vee z$ belongs to φ'. If $z = x$, then any satisfying assignment to φ' sets y to true, a contradiction to our first preprocessing step. Otherwise, the presence of C and C' implies that in any assignment ψ' satisfying φ' we have $\psi'(x) = \top$ or $\psi'(z) = \top$. Thus, $x \vee z$ is a clause of φ', and $xz \in E(H)$. However, neither x nor z belongs to X, a contradiction.

In the last case, $C = \neg x \vee \neg y$ and, again, we assume that ψ does not satisfy C, that is, $x, y \in X$. Since X is a minimal vertex cover, there exist $s \in N_H(x)$, $t \in N_H(y)$ such that $s, t \notin X$. It follows from the definition of H that the clauses $C_x = x \vee s$ and $C_y = y \vee t$ are present in φ'. If $s = t$, then the clauses C, C_x and C_y imply that t is set to true in any truth assignment satisfying φ', a contradiction to our first preprocessing step. If $s \neq t$, then observe that the clauses C, C_x and C_y imply that either s or t is set to true in any truth assignment satisfying φ' and, consequently, $s \vee t$ is a clause of φ' and $st \in E(H)$. However, $s, t \notin X$, a contradiction.

2.27 If $\emptyset \in \mathcal{A}$, then conclude that we are dealing with a no-instance. Otherwise, for every set $X \in \mathcal{A}$ of size $|X| < d$, create $d - |X|$ new elements and add them to X.

2.28 There are two ways different ways to solve this exercise. First, you can treat the input instance as a d-Hitting Set instance, proceed as in Section 2.6.1, and at the end apply the solution of Exercise 2.27 to the obtained kernel, in order to get an Ed-Hitting Set instance.

In a second approach, try to find a sunflower with $k + 2$ sets, instead of $k + 1$ as in Section 2.6.1. If a sunflower is found, then discard one of the sets: the remaining $k + 1$ sets still ensure that the core needs to be hit in any solution of size at most k.

2.29 Show that in a $(dk+2)$-sunflower, one set can be discarded without changing the answer to the problem.

2.30 Proceed as in Exercise 2.27: pad every set $X \in \mathcal{A}$ with $d-|X|$ newly created elements.

2.31 Show that, in a split graph, every path can be shortened to a path on at most four vertices. Thus, for every $1 \le i \le k$, we have a family $\mathcal{F}_i$ of vertex sets of possible paths between s_i and t_i, and this family is of size $\mathcal{O}(n^4)$. Interpret the problem as a d-Set Packing instance for some constant d and family $\mathcal{F} = \bigcup_{i=1}^{k} \mathcal{F}_i$. Run the kernelization algorithm from the previous exercise, and discard all vertices that are not contained in any set in the obtained kernel.

2.32 Show that, in a graph excluding a d-vertex path as an induced subgraph, every path in a solution can shortened to a path on at most $d-1$ vertices. Proceed then as in Exercise 2.31.

2.33 Let $\mathcal{A}$ be a family of all vertex sets of a P_3 (induced path on three vertices) in G. In this manner, Cluster Vertex Deletion becomes a 3-Hitting Set problem on family $\mathcal{A}$, as we need to hit all induced P_3s in G. Reduce $\mathcal{A}$, but not exactly as in the d-Hitting Set case: repeatedly find a $k+2$ sunflower and delete one of its elements from $\mathcal{A}$. Show that this reduction is safe for Cluster Vertex Deletion. Moreover, show that, if $\mathcal{A}'$ is the family after the reduction is exhaustively applied, then $(G[\bigcup \mathcal{A}'], k)$ is the desired kernel.

2.34 Use the following observation: a perfect graph is bipartite if and only if it does not contain a triangle. Thus, the problem reduces to hitting all triangles in the input graph, which is a 3-Hitting Set instance.

2.35 Proceed as in the case of Cluster Vertex Deletion: interpret a Split Vertex Deletion instance as a 5-Hitting Set instance.

2.36 Let $\{C_4, C_5, 2K_2\}$ be the set of *forbidden induced subgraphs* for split graphs. That is, a graph is a split graph if it contains none of these three graphs as an induced subgraph.

You may need (some of) the following reduction rules. (Note that the safeness of some of them is not so easy.)

1. The "standard" sunflower-like: if more than k forbidden induced subgraphs share a single edge (and otherwise are pairwise edge-disjoint), delete the edge in question.
2. The irrelevant vertex rule: if a vertex is not part of any forbidden induced subgraph, then delete the vertex in question. (Safeness is not obvious here!)
3. If two adjacent edges uv and uw are contained in more than k induced C_4s, then delete uv and uw, and replace them with edges va and wb, where a and b are new degree-1 vertices.
4. If two adjacent edges uv and uw are contained in more than k pairwise edge-disjoint induced C_5s, then delete uv and uw, and decrease k by 2.
5. If the edges v_1v_2, v_2v_3 and v_3v_4 are contained in more than k induced C_5s, delete v_2v_3 and decrease k by 1.

2.37 By induction, show that every graph on at least 4^k vertices has either a clique or an independent set on k vertices. Observe that this implies a kernel of exponential size for the problem.

2.38 Show that every tournament on n vertices has a transitive subtournament on $\mathcal{O}(\log n)$ vertices. Then, use this fact to show that every oriented directed graph on n vertices has an induced directed acyclic subgraph on $\log n$ vertices. Finally, obtain an exponential kernel for the considered problem.

Bibliographic notes

The history of preprocessing, such as that of applying reduction rules to simplify truth functions, can be traced back to the origins of computer science—the 1950 work of Quine [391]. A modern example showing the striking power of efficient preprocessing is the commercial integer linear program solver CPLEX. The name "lost continent of polynomial-time preprocessing" is due to Mike Fellows [175].

Lemma 2.2 on equivalence of kernelization and fixed-parameter tractability is due to Cai, Chen, Downey, and Fellows [66]. The reduction rules for Vertex Cover discussed in this chapter are attributed to Buss in [62] and are often referred to as Buss kernelization in the literature. A more refined set of reduction rules for Vertex Cover was introduced in [24]. The kernelization algorithm for FAST in this chapter follows the lines provided by Dom, Guo, Hüffner, Niedermeier and Truß [140]. The improved kernel with $(2+\varepsilon)k$ vertices, for $\varepsilon > 0$, is obtained by Bessy, Fomin, Gaspers, Paul, Perez, Saurabh, and Thomassé in [32]. The exponential kernel for Edge Clique Cover is taken from [234], see also Gyárfás [253]. Cygan, Kratsch, Pilipczuk, Pilipczuk and Wahlström [114] showed that Edge Clique Cover admits no kernel of polynomial size unless NP $\subseteq$ coNP/ poly (see Exercise 15.4, point 16). Actually, as we will see in Chapter 14 (Exercise 14.10), one can prove a stronger result: no subexponential kernel exists for this problem unless P $=$ NP .

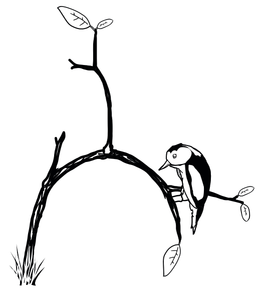
Kőnig's theorem (found also independently in a more general setting of weighted graphs by Egerváry) and Hall's theorem [256, 303] are classic results from graph theory, see also the book of Lovász and Plummer [338] for a general overview of matching theory. The crown rule was introduced by Chor, Fellows and Juedes in [94], see also [174]. Implementation issues of kernelization algorithms for vertex cover are discussed in [4]. The kernel for Maximum Satisfiability (Theorem 2.16) is taken from [323]. Abu-Khzam used crown decomposition to obtain a kernel for d-Hitting Set with at most $(2d-1)k^{d-1}+k$ elements [3] and for d-Set Packing with $\mathcal{O}(k^{d-1})$ elements [2]. Crown Decomposition and its variations were used to obtain kernels for different problems by Wang, Ning, Feng and Chen [431], Prieto and Sloper [388, 389], Fellows, Heggernes, Rosamond, Sloper and Telle [178], Moser [369], Chlebík and Chlebíková [93]. The expansion lemma, in a slightly different form, appears in the PhD thesis of Prieto [387], see also Thomassé [420, Theorem 2.3] and Halmos and Vaughan [257].

The Nemhauser-Trotter theorem is a classical result from combinatorial optimization [375]. Our proof of this theorem mimics the proof of Khuller from [289]. The application of the Nemhauser-Trotter theorem in kernelization was observed by Chen, Kanj and Jia [81]. The sunflower lemma is due to Erdős and Rado [167]. Our kernelization for d-Hitting Set follows the lines of [189].

Exercise 2.11 is from [392], Exercise 2.15 is from [121, 383], Exercise 2.16 is from [281], Exercise 2.18 is from [251] and Exercise 2.23 is from [190]. An improved kernel for the above guarantee variant of Maximum Bisection, discussed in Exercise 2.18, is obtained by Mnich and Zenklusen [364]. A polynomial kernel for Split Edge Deletion (Exercise 2.36) was first shown by Guo [240]. An improved kernel, as well as a smaller kernel for Split Vertex Deletion, was shown by Ghosh, Kolay, Kumar, Misra, Panolan, Rai, and Ramanujan [228]. It is worth noting that the very simple kernel of Exercise 2.4 is probably optimal by the result of Kratsch, Philip, and Ray [308].

Chapter 3
Bounded search trees

In this chapter we introduce a variant of exhaustive search, namely the method of bounded search trees. This is one of the most commonly used tools in the design of fixed-parameter algorithms. We illustrate this technique with algorithms for two different parameterizations of Vertex Cover*, as well as for the problems (undirected)* Feedback Vertex Set *and* Closest String*.*



Bounded search trees, or simply *branching*, is one of the simplest and most commonly used techniques in parameterized complexity that originates in the general idea of backtracking. The algorithm tries to build a feasible solution to the problem by making a sequence of decisions on its shape, such as whether to include some vertex into the solution or not. Whenever considering one such step, the algorithm investigates many possibilities for the decision, thus effectively *branching* into a number of subproblems that are solved one by one. In this manner the execution of a branching algorithm can be viewed as a *search tree*, which is traversed by the algorithm up to the point when a solution is discovered in one of the leaves. In order to justify the correctness of a branching algorithm, one needs to argue that in case of a yes-instance some sequence of decisions captured by the algorithm leads to a feasible solution. If the total size of the search tree is bounded by a function of the parameter alone, and every step takes polynomial time, then such a branching algorithm runs in FPT time. This is indeed the case for many natural backtracking algorithms.

More precisely, let I be an instance of a minimization problem (such as Vertex Cover). We associate a measure $\mu(I)$ with the instance I, which, in the case of FPT algorithms, is usually a function of k alone. In a branch step we generate from I simpler instances $I_1, \ldots, I_\ell$ ($\ell \geq 2$) of the same problem such that the following hold.

1. Every feasible solution S of I_i, $i \in \{1, \ldots, \ell\}$, corresponds to a feasible solution $h_i(S)$ of I. Moreover, the set
$$\Big\{h_i(S) \ : \ 1 \leq i \leq \ell \text{ and } S \text{ is a feasible solution of } I_i\Big\}$$
contains at least one optimum solution for I. Informally speaking, a branch step splits problem I into subproblems $I_1, \ldots, I_\ell$, possibly taking some (formally justified) greedy decisions.
2. The number ℓ is *small*, e.g., it is bounded by a function of $\mu(I)$ alone.
3. Furthermore, for every I_i, $i \in \{1, \ldots, \ell\}$, we have that $\mu(I_i) \leq \mu(I) - c$ for some constant $c > 0$. In other words, in every branch we *substantially* simplify the instance at hand.

In a branching algorithm, we recursively apply branching steps to instances $I_1, I_2, \ldots, I_\ell$, until they become simple or even trivial. Thus, we may see an execution of the algorithm as a *search tree*, where each recursive call corresponds to a node: the calls on instances $I_1, I_2, \ldots, I_\ell$ are children of the call on instance I. The second and third conditions allow us to bound the number of nodes in this search tree, assuming that the instances with non-positive measure are simple. Indeed, the third condition allows us to bound the depth of the search tree in terms of the measure of the original instance, while the second condition controls the number of branches below every node. Because of these properties, search trees of this kind are often called *bounded search trees*. A branching algorithm with a cleverly chosen branching step often offers a drastic improvement over a straightforward exhaustive search.

We now present a typical scheme of applying the idea of bounded search trees in the design of parameterized algorithms. We first identify, in polynomial time, a small (typically of size that is constant, or bounded by a function of the parameter) subset S of elements of which at least one must be in *some* or *every* feasible solution of the problem. Then we solve $|S|$ subproblems: for each element e of S, create one subproblem in which we include e in the solution, and solve the remaining task with a reduced parameter value. We also say that we *branch* on the element of S that belongs to the solution. Such search trees are analyzed by measuring the drop of the parameter in each branch. If we ensure that the parameter (or some measure bounded by a function of the parameter) decreases in each branch by at least a constant value, then we will be able to bound the depth of the search tree by a function of the parameter, which results in an FPT algorithm.

It is often convenient to think of branching as of "guessing" the right branch. That is, whenever performing a branching step, the algorithm guesses the right part of an (unknown) solution in the graph, by trying all possibilities. What we need to ensure is that there will be a sequence of guesses that uncovers the whole solution, and that the total time spent on wrong guesses is not too large.

We apply the idea of bounded search trees to Vertex Cover in Section 3.1. Section 3.2 briefly discusses methods of bounding the number of nodes of a search tree. In Section 3.3 we give a branching algorithm for Feedback Vertex Set in undirected graphs. Section 3.4 presents an algorithm for a different parameterization of Vertex Cover and shows how this algorithm implies algorithms for other parameterized problems such as Odd Cycle Transversal and Almost 2-SAT. Finally, in Section 3.5 we apply this technique to a non-graph problem, namely Closest String.

3.1 Vertex Cover

As the first example of branching, we use the strategy on Vertex Cover. In Chapter 2 (Lemma 2.23), we gave a kernelization algorithm which in time $\mathcal{O}(n\sqrt{m})$ constructs a kernel on at most $2k$ vertices. Kernelization can be easily combined with a brute-force algorithm to solve Vertex Cover in time $\mathcal{O}(n\sqrt{m}+4^k k^{\mathcal{O}(1)})$. Indeed, there are at most $2^{2k} = 4^k$ subsets of size at most k in a $2k$-vertex graph. Thus, by enumerating all vertex subsets of size at most k in the kernel and checking whether any of these subsets forms a vertex cover, we can solve the problem in time $\mathcal{O}(n\sqrt{m} + 4^k k^{\mathcal{O}(1)})$. We can easily obtain a better algorithm by branching. Actually, this algorithm was already presented in Chapter 1 under the cover of the Bar Fight Prevention problem.

Let (G, k) be a Vertex Cover instance. Our algorithm is based on the following two simple observations.

- For a vertex v, any vertex cover must contain either v or *all* of its neighbors $N(v)$.
- Vertex Cover becomes trivial (in particular, can be solved optimally in polynomial time) when the maximum degree of a graph is at most 1.

We now describe our recursive branching algorithm. Given an instance (G, k), we first find a vertex $v \in V(G)$ of maximum degree in G. If v is of degree 1, then every connected component of G is an isolated vertex or an edge, and the instance has a trivial solution. Otherwise, $|N(v)| \geq 2$ and we recursively branch on two cases by considering

either v, or $N(v)$ in the vertex cover.

In the branch where v is in the vertex cover, we can delete v and reduce the parameter by 1. In the second branch, we add $N(v)$ to the vertex cover, delete $N[v]$ from the graph and decrease k by $|N(v)| \geq 2$.

The running time of the algorithm is bounded by (the number of nodes in the search tree) $\times$ (time taken at each node).

Clearly, the time taken at each node is bounded by $n^{\mathcal{O}(1)}$. Thus, if $\tau(k)$ is the number of nodes in the search tree, then the total time used by the algorithm is at most $\tau(k)n^{\mathcal{O}(1)}$.

In fact, in every search tree $\mathcal{T}$ that corresponds to a run of a branching algorithm, every internal node of $\mathcal{T}$ has at least two children. Thus, if $\mathcal{T}$ has ℓ leaves, then the number of nodes in the search tree is at most $2\ell - 1$. Hence, to bound the running time of a branching algorithm, it is sufficient to bound the number of leaves in the corresponding search tree.

In our case, the tree $\mathcal{T}$ is the search tree of the algorithm when run with parameter k. Below its root, it has two subtrees: one for the same algorithm run with parameter $k-1$, and one recursive call with parameter at most $k-2$. The same pattern occurs deeper in $\mathcal{T}$. This means that if we define a function $T(k)$ using the recursive formula

$$T(i) = \begin{cases} T(i-1) + T(i-2) & \text{if } i \geq 2, \\ 1 & \text{otherwise,} \end{cases}$$

then the number of leaves of $\mathcal{T}$ is bounded by $T(k)$.

Using induction on k, we prove that $T(k)$ is bounded by 1.6181^k. Clearly, this is true for $k = 0$ and $k = 1$, so let us proceed for $k \geq 2$:

$$\begin{aligned} T(k) &= T(k-1) + T(k-2) \leq 1.6181^{k-1} + 1.6181^{k-2} \\ &\leq 1.6181^{k-2}(1.6181 + 1) \leq 1.6181^{k-2}(1.6181)^2 \leq 1.6181^k. \end{aligned}$$

This proves that the number of leaves is bounded by 1.6181^k. Combined with kernelization, we arrive at an algorithm solving Vertex Cover in time $\mathcal{O}(n\sqrt{m} + 1.6181^k k^{\mathcal{O}(1)})$.

A natural question is how did we know that 1.6181^k is a solution to the above recurrence. Suppose that we are looking for an upper bound on function $T(k)$ of the form $T(k) \leq c \cdot \lambda^k$, where $c > 0$, $\lambda > 1$ are some constants. Clearly, we can set constant c so that the initial conditions in the definition of $T(k)$ are satisfied. Then, we are left with proving, using induction, that this bound holds for every k. This boils down to proving that

$$c \cdot \lambda^k \geq c \cdot \lambda^{k-1} + c \cdot \lambda^{k-2}, \tag{3.1}$$

since then we will have

$$T(k) = T(k-1) + T(k-2) \leq c \cdot \lambda^{k-1} + c \cdot \lambda^{k-2} \leq c \cdot \lambda^k.$$

Observe that (3.1) is equivalent to $\lambda^2 \geq \lambda + 1$, so it makes sense to look for the lowest possible value of λ for which this inequality is satisfied; this is actually the one for which equality holds. By solving equation $\lambda^2 = \lambda + 1$ for $\lambda > 1$, we find that $\lambda = \frac{1+\sqrt{5}}{2} < 1.6181$, so for this value of λ the inductive proof works.

The running time of the above algorithm can be easily improved using the following argument, whose proof we leave as Exercise 3.1.

Proposition 3.1. Vertex Cover *can be solved optimally in polynomial time when the maximum degree of a graph is at most* 2.

Thus, we branch only on the vertices of degree at least 3, which immediately brings us to the following upper bound on the number of leaves in a search tree:

$$T(k) = \begin{cases} T(k-1) + T(k-3) & \text{if } k \geq 3, \\ 1 & \text{otherwise.} \end{cases}$$

Again, an upper bound of the form $c \cdot \lambda^k$ for the above recursive function can be obtained by finding the largest root of the polynomial equation $\lambda^3 = \lambda^2 + 1$. Using standard mathematical techniques (and/or symbolic algebra packages) the root is estimated to be at most 1.4656. Combined with kernelization, this gives us the following theorem.

Theorem 3.2. Vertex Cover *can be solved in time* $\mathcal{O}(n\sqrt{m} + 1.4656^k k^{\mathcal{O}(1)})$.

Can we apply a similar strategy for graphs of vertex degree at most 3? Well, this becomes more complicated as Vertex Cover is NP-hard on this class of graphs. But there are more involved branching strategies, and there are faster branching algorithms than the one given in Theorem 3.2.

3.2 How to solve recursive relations

For algorithms based on the bounded search tree technique, we need to bound the number of nodes in the search tree to obtain an upper bound on the running time of the algorithm. For this, recurrence relations are used. The most common case in parameterized branching algorithms is when we use linear recurrences with constant coefficients. There exists a standard technique to bound the number of nodes in the search tree for this case. If the algorithm solves a problem of size n with parameter k and calls itself recursively on problems with decreased parameters $k - d_1, k - d_2, \ldots, k - d_p$, then $(d_1, d_2, \ldots, d_p)$ is called the *branching vector* of this recursion. For example, we used a branching vector $(1, 2)$ to obtain the first algorithm for Vertex Cover in the previous section, and a branching vector $(1, 3)$ for the second one. For a branching vector $(d_1, d_2, \ldots, d_p)$, the upper bound $T(k)$

on the number of leaves in the search tree is given by the following linear recurrence:

$$T(k) = T(k-d_1) + T(k-d_2) + \cdots + T(k-d_p).$$

Again, for $k < d$, where $d = \max_{i=1,2,\ldots,p} d_i$, we put the initial condition $T(k) = 1$. Assuming that new subproblems with smaller parameters can be computed in time polynomial in n, the running time of such recursive algorithm is $T(k) \cdot n^{\mathcal{O}(1)}$.

If we now look for an upper bound of the form $T(k) \leq c \cdot \lambda^k$, then the inductive step boils down to proving the following inequality:

$$\lambda^k \geq \lambda^{k-d_1} + \lambda^{k-d_2} + \cdots + \lambda^{k-d_p}. \tag{3.2}$$

Inequality (3.2) can be rewritten as $P(\lambda) \geq 0$, where

$$P(\lambda) = \lambda^d - \lambda^{d-d_1} - \lambda^{d-d_2} - \cdots - \lambda^{d-d_p} \tag{3.3}$$

is the *characteristic polynomial* of the recurrence for $T(k)$ (recall $d = \max_{i=1,2,\ldots,p} d_i$). Using standard techniques of calculus it is not hard to show that if a polynomial P has a form as in (3.3), then P has a unique positive root λ_0, and moreover $P(\lambda) < 0$ for $0 < \lambda < \lambda_0$ and $P(\lambda) > 0$ for $\lambda > \lambda_0$. This means that λ_0 is the best possible value that can be used in an upper bound for $T(k)$. In the bibliographic notes we provide further references to the relevant literature on solving recursive formulas.

The root λ_0 is often called the *branching number* corresponding to the branching vector $(d_1, d_2, \ldots, d_p)$. Hence, the running time of the considered branching algorithm is bounded by $\lambda_0^k n^{\mathcal{O}(1)}$. In Table 3.1, we give branching numbers corresponding to branching vectors (i, j) for $i, j \in \{1, \ldots, 6\}$.

(i,j)	1	2	3	4	5	6
1	2.0000	1.6181	1.4656	1.3803	1.3248	1.2852
2		1.4143	1.3248	1.2721	1.2366	1.2107
3			1.2560	1.2208	1.1939	1.1740
4				1.1893	1.1674	1.1510
5					1.1487	1.1348
6						1.1225

Table 3.1: A table of branching numbers (rounded up)

Two natural questions arise:

- How good is the estimation of $T(k)$ using the exponent of the corresponding branching number?
- How well does $T(k)$ estimate the actual size of the search tree?

The answer to the first question is "it is good": up to a polynomial factor, the estimation is tight. The second question is much more difficult, since the actual way a branching procedure explores the search space may be more complex than our estimation of its behavior using recursive formulas. If, say, a branching algorithm uses several ways of branching into subproblems (so-called branching rules) that correspond to different branching vectors, and/or is combined with local reduction rules, then so far we do not know how to estimate the running time better than by using the branching number corresponding to the worst branching vector. However, the delicate interplay between different branching rules and reduction rules may lead to a much smaller tree than what follows from our imprecise estimations.

3.3 Feedback Vertex Set

For a given graph G and a set $X \subseteq V(G)$, we say that X is a *feedback vertex set* of G if $G - X$ is an acyclic graph (i.e., a forest). In the Feedback Vertex Set problem, we are given an undirected graph G and a nonnegative integer k, and the objective is to determine whether there exists a feedback vertex set of size at most k in G. In this section, we give a branching algorithm solving Feedback Vertex Set in time $k^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$.

It is more convenient for us to consider this problem in the more general setting of *multigraphs*, where the input graph G may contain multiple edges and loops. We note that both a double edge and a loop are cycles. We also use the convention that a loop at a vertex v contributes 2 to the degree of v.

We start with some simple reduction rules that clean up the graph. At any point, we use the lowest-numbered applicable rule. We first deal with the multigraph parts of G. Observe that any vertex with a loop needs to be contained in any solution set X.

Reduction FVS.1. If there is a loop at a vertex v, delete v from the graph and decrease k by 1.

Moreover, notice that the multiplicity of a multiple edge does not influence the set of feasible solutions to the instance (G, k).

Reduction FVS.2. If there is an edge of multiplicity larger than 2, reduce its multiplicity to 2.

We now reduce vertices of low degree. Any vertex of degree at most 1 does not participate in any cycle in G, so it can be deleted.

Reduction FVS.3. If there is a vertex v of degree at most 1, delete v.

Concerning vertices of degree 2, observe that, instead of including into the solution any such vertex, we may as well include one of its neighbors. This leads us to the following reduction.

Reduction FVS.4. If there is a vertex v of degree 2, delete v and connect its two neighbors by a new edge.

Two remarks are in place. First, a vertex v in Reduction FVS.4 cannot have a loop, as otherwise Reduction FVS.1 should be triggered on v instead. This ensures that the neighbors of v are distinct from v, hence the rule may be applied and the safeness argument works. Second, it is possible that v is incident to a double edge: in this case, the reduction rule deletes v and adds a loop to a sole "double" neighbor of v. Observe that in this case Reduction FVS.1 will trigger subsequently on this neighbor.

We remark that after exhaustively applying these four reduction rules, the resulting graph G

(P1) contains no loops,
(P2) has only single and double edges, and
(P3) has minimum vertex degree at least 3.

Moreover, all rules are trivially applicable in polynomial time. From now on we assume that in the input instance (G, k), graph G satisfies properties (P1)–(P3).

We remark that for the algorithm in this section, we do not need properties (P1) and (P2). However, we will need these properties later for the kernelization algorithm in Section 9.1.

Finally, we need to add a rule that stops the algorithm if we already exceeded our budget.

Reduction FVS.5. If $k < 0$, terminate the algorithm and conclude that (G, k) is a no-instance.

The intuition behind the algorithm we are going to present is as follows. Observe that if X is a feedback vertex set of G, then $G - X$ is a forest. However, $G - X$ has at most $|V(G)| - |X| - 1$ edges and thus $G - X$ cannot have "many" vertices of high degree. Thus, if we pick some $f(k)$ vertices with the highest degrees in the graph, then every solution of size at most k must contain one of these high-degree vertices. In what follows we make this intuition work.

Let $(v_1, v_2, \dots, v_n)$ be a descending ordering of $V(G)$ according to vertex degrees, i.e., $d(v_1) \geq d(v_2) \geq \cdots \geq d(v_n)$. Let $V_{3k} = \{v_1, \dots, v_{3k}\}$. Let us recall that the minimum vertex degree of G is at least 3. Our algorithm for Feedback Vertex Set is based on the following lemma.

Lemma 3.3. *Every feedback vertex set in G of size at most k contains at least one vertex of V_{3k}.*

Proof. To prove this lemma we need the following simple claim.

Claim 3.4. *For every feedback vertex set X of G,*

$$\sum_{v \in X} (d(v) - 1) \geq |E(G)| - |V(G)| + 1.$$

Proof. Graph $F = G - X$ is a forest and thus the number of edges in F is at most $|V(G)| - |X| - 1$. Every edge of $E(G) \setminus E(F)$ is incident to a vertex of X. Hence

$$\sum_{v \in X} d(v) + |V(G)| - |X| - 1 \geq |E(G)|.$$

⌟

Targeting a contradiction, let us assume that there is a feedback vertex set X of size at most k such that $X \cap V_{3k} = \emptyset$. By the choice of V_{3k}, for every $v \in X$, $d(v)$ is at most the minimum of vertex degrees from V_{3k}. Because $|X| \leq k$, by Claim 3.4 we have that

$$\sum_{i=1}^{3k} (d(v_i) - 1) \geq 3 \cdot \Big(\sum_{v \in X} (d(v) - 1) \Big) \geq 3 \cdot (|E(G)| - |V(G)| + 1).$$

In addition, we have that $X \subseteq V(G) \setminus V_{3k}$, and hence

$$\sum_{i > 3k} (d(v_i) - 1) \geq \sum_{v \in X} (d(v) - 1) \geq (|E(G)| - |V(G)| + 1).$$

Therefore,

$$\sum_{i=1}^{n} (d(v_i) - 1) \geq 4 \cdot (|E(G)| - |V(G)| + 1).$$

However, observe that $\sum_{i=1}^{n} d(v_i) = 2|E(G)|$: every edge is counted twice, once for each of its endpoints. Thus we obtain

$$4 \cdot (|E(G)| - |V(G)| + 1) \leq \sum_{i=1}^{n} (d(v_i) - 1) = 2|E(G)| - |V(G)|,$$

which implies that $2|E(G)| < 3|V(G)|$. However, this contradicts the fact that every vertex of G is of degree at least 3. □

We use Lemma 3.3 to obtain the following algorithm for Feedback Vertex Set.

Theorem 3.5. *There exists an algorithm for* Feedback Vertex Set *running in time $(3k)^k \cdot n^{\mathcal{O}(1)}$.*

Proof. Given an undirected graph G and an integer $k \geq 0$, the algorithm works as follows. It first applies Reductions FVS.1, FVS.2, FVS.3, FVS.4, and FVS.5 exhaustively. As the result, we either already conclude that we are dealing with a no-instance, or obtain an equivalent instance (G', k') such that G' has minimum degree at least 3 and $k' \leq k$. If G' is empty, then we conclude that we are dealing with a yes-instance, as $k' \geq 0$ and an empty set is a feasible solution. Otherwise, let $V_{3k'}$ be the set of $3k'$ vertices of G' with largest degrees. By Lemma 3.3, every solution X to the Feedback Vertex Set instance (G', k') contains at least one vertex from $V_{3k'}$. Therefore, we branch on the choice of one of these vertices, and for every vertex $v \in V_{3k'}$, we recursively apply the algorithm to solve the Feedback Vertex Set instance $(G' - v, k' - 1)$. If one of these branches returns a solution X', then clearly $X' \cup \{v\}$ is a feedback vertex set of size at most k' for G'. Else, we return that the given instance is a no-instance.

At every recursive call we decrease the parameter by 1, and thus the height of the search tree does not exceed k'. At every step we branch in at most $3k'$ subproblems. Hence the number of nodes in the search tree does not exceed $(3k')^{k'} \leq (3k)^k$. This concludes the proof. □

3.4 Vertex Cover Above LP

Recall the integer linear programming formulation of Vertex Cover and its relaxation LPVC(G):

$$\begin{array}{lll} \text{min} & \sum_{v \in V(G)} x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for every } uv \in E(G), \\ & 0 \leq x_v \leq 1 & \text{for every } v \in V(G). \end{array}$$

These programs were discussed in Section 2.2.1. If the minimum value of LPVC(G) is $\mathrm{vc}^*(G)$, then the size of a minimum vertex cover is at least $\mathrm{vc}^*(G)$. This leads to the following parameterization of Vertex Cover, which we call Vertex Cover Above LP: Given a graph G and an integer k, we ask for a vertex cover of G of size at most k, but instead of seeking an FPT algorithm parameterized by k as for Vertex Cover, the parameter now is $k - \mathrm{vc}^*(G)$. In other words, the goal of this section is to design an algorithm for Vertex Cover on an n-vertex graph G with running time $f(k - \mathrm{vc}^*(G)) \cdot n^{\mathcal{O}(1)}$ for some computable function f.

The parameterization by $k - \mathrm{vc}^*(G)$ falls into a more general theme of *above guarantee parameterization*, where, instead of parameterizing purely by the solution size k, we look for some (computable in polynomial time) lower bound ℓ for the solution size, and use a more refined parameter $k - \ell$, the *excess* above the lower bound. Such a parameterization makes perfect sense in problems where the solution size is often quite large and, consequently, FPT algorithms in parameterization by k may not be very efficient. In the Vertex Cover problem, the result of Section 2.5 — a kernel with at most $2k$ vertices — can be on one hand seen as a very good result, but on the other hand can be an evidence that the solution size parameterization for Vertex Cover may not be the most meaningful one, as the parameter can be quite large. A more striking example is the Maximum Satisfiability problem, studied in Section 2.3.2: here an instance with at least $2k$ clauses is trivially a yes-instance. In Section *9.2 we study an above guarantee parameterization of a variant of Maximum Satisfiability, namely Max-Er-SAT. In Exercise 9.3 we also ask for an FPT algorithm for Maximum Satisfiability parameterized by $k - m/2$, where m is the number of clauses in the input formula. The goal of this section is to study the above guarantee parameterization of Vertex Cover, where the lower bound is the cost of an optimum solution to an LP relaxation.

Before we describe the algorithm, we fix some notation. By *optimum solution* $\mathbf{x} = (x_v)_{v \in V(G)}$ to LPVC(G), we mean a feasible solution with $1 \geq x_v \geq 0$ for all $v \in V(G)$ that minimizes the objective function (sometimes called the *cost*) $\mathbf{w}(\mathbf{x}) = \sum_{v \in V(G)} x_v$. By Proposition 2.24, for any graph G there exists an optimum half-integral solution of LPVC(G), i.e., a solution with $x_v \in \{0, \frac{1}{2}, 1\}$ for all $v \in V(G)$, and such a solution can be found in polynomial time.

Let $\mathrm{vc}(G)$ denote the size of a minimum vertex cover of G. Clearly, $\mathrm{vc}(G) \geq \mathrm{vc}^*(G)$. For a half-integral solution $\mathbf{x} = (x_v)_{v \in V(G)}$ and $i \in \{0, \frac{1}{2}, 1\}$, we define $V_i^{\mathbf{x}} = \{v \in V \ : \ x_v = i\}$. We also say that $\mathbf{x} = \{x_v\}_{v \in V(G)}$ is *all-$\frac{1}{2}$-solution* if $x_v = \frac{1}{2}$ for every $v \in V(G)$. Because the all-$\frac{1}{2}$-solution is a feasible solution, we have that $\mathrm{vc}^*(G) \leq \frac{|V(G)|}{2}$. Furthermore, we define the *measure* of an instance (G, k) to be our parameter of interest $\mu(G, k) = k - \mathrm{vc}^*(G)$.

Recall that in Section 2.5 we have developed Reduction VC.4 for Vertex Cover. This reduction, if restricted to half-integral solutions, can be stated as follows: for an optimum half-integral LPVC(G) solution $\mathbf{x}$, we (a) conclude that the input instance (G, k) is a no-instance if $\mathbf{w}(\mathbf{x}) > k$; and (b) delete $V_0^{\mathbf{x}} \cup V_1^{\mathbf{x}}$ and decrease k by $|V_1^{\mathbf{x}}|$ otherwise. As we are now dealing with measure $\mu(G, k) = k - \mathrm{vc}^*(G)$, we need to understand how this parameter changes under Reduction VC.4.

Lemma 3.6. *Assume an instance (G', k') is created from an instance (G, k) by applying Reduction VC.4 to a half-integral optimum solution $\mathbf{x}$. Then $\mathrm{vc}^*(G) - \mathrm{vc}^*(G') = \mathrm{vc}(G) - \mathrm{vc}(G') = |V_1^{\mathbf{x}}| = k - k'$. In particular, $\mu(G', k') = \mu(G, k)$.*

We remark that, using Exercise 2.25, the statement of Lemma 3.6 is true for any optimum solution $\mathbf{x}$, not only a half-integral one (see Exercise 3.19). However, the proof for a half-integral solution is slightly simpler, and, thanks to Proposition 2.24, we may work only with half-integral solutions.

Proof (of Lemma 3.6). Observe that every edge of G incident to $V_0^{\mathbf{x}}$ has its second endpoint in $V_1^{\mathbf{x}}$. Hence, we have the following:

- For every vertex cover Y of G', $Y \cup V_1^{\mathbf{x}}$ is a vertex cover of G and, consequently, $\mathrm{vc}(G) \leq \mathrm{vc}(G') + |V_1^{\mathbf{x}}|$.
- For every feasible solution $\mathbf{y}'$ to LPVC(G'), if we define a vector $\mathbf{y} = (y_v)_{v \in V(G)}$ as $y_v = y'_v$ for every $v \in V(G')$ and $y_v = x_v$ for every $v \in V(G) \setminus V(G')$, then we obtain a feasible solution to LPVC(G) of cost $\mathbf{w}(\mathbf{y}') + |V_1^{\mathbf{x}}|$; consequently, $\mathrm{vc}^*(G) \leq \mathrm{vc}^*(G') + |V_1^{\mathbf{x}}|$.

Now it suffices to prove the reversed versions of the two inequalities obtained above. Theorem 2.19 ensures that $\mathrm{vc}(G') \leq \mathrm{vc}(G) - |V_1^{\mathbf{x}}|$. Moreover, since $\mathbf{x}$ restricted to $V(G')$ is a feasible solution to LPVC(G') of cost $\mathrm{vc}^*(G) - |V_1^{\mathbf{x}}|$, we have $\mathrm{vc}^*(G') \leq \mathrm{vc}^*(G) - |V_1^{\mathbf{x}}|$. □

In Theorem 2.21 we simply solved LPVC(G), and applied Reduction VC.4 to obtain a kernel with at most $2k$ vertices. In this section we would like to do something slightly stronger: to apply Reduction VC.4 as long as there exists some half-integral optimum solution $\mathbf{x}$ that is not the all-$\frac{1}{2}$-solution. Luckily, by a simple self-reduction trick, we can always detect such solutions.

Lemma 3.7. *Given a graph G, one can in $\mathcal{O}(mn^{3/2})$ time find an optimum solution to LPVC(G) which is not the all-$\frac{1}{2}$-solution, or correctly conclude that the all-$\frac{1}{2}$-solution is the unique optimum solution to LPVC(G). Moreover, the returned optimum solution is half-integral.*

Proof. First, use Proposition 2.24 to solve LPVC(G), obtaining an optimum half-integral solution $\mathbf{x}$. If $\mathbf{x}$ is not the all-$\frac{1}{2}$-solution, then return $\mathbf{x}$. Otherwise, proceed as follows.

For every $v \in V(G)$, use Proposition 2.24 again to solve LPVC$(G - v)$, obtaining an optimum half-integral solution $\mathbf{x}^v$. Define a vector $\mathbf{x}^{v,\circ}$ as $\mathbf{x}^v$, extended with a value $x_v = 1$. Note that $\mathbf{x}^{v,\circ}$ is a feasible solution to LPVC(G) of cost $\mathbf{w}(\mathbf{x}^{v,\circ}) = \mathbf{w}(\mathbf{x}^v) + 1 = \mathrm{vc}^*(G - v) + 1$. Thus, if for some $v \in V(G)$ we have $\mathbf{w}(\mathbf{x}^v) = \mathrm{vc}^*(G - v) \leq \mathrm{vc}^*(G) - 1$, then $\mathbf{x}^{v,\circ}$ is an optimum solution for LPVC(G). Moreover, $\mathbf{x}^{v,\circ}$ is half-integral, but is not equal to the all-$\frac{1}{2}$-solution due to the value at vertex v. Hence, we may return $\mathbf{x}^{v,\circ}$.

We are left with the case

$$\mathrm{vc}^*(G - v) > \mathrm{vc}^*(G) - 1 \quad \text{for every } v \in V(G). \tag{3.4}$$

We claim that in this case the all-$\frac{1}{2}$-solution is the unique solution to LPVC(G); note that such a claim would conclude the proof of the lemma, as the computation time used so far is bounded by $\mathcal{O}(mn^{3/2})$ ($n+1$ applications of Proposition 2.24). Observe that, due to Proposition 2.24, both $2\,\mathrm{vc}^*(G-v)$ and $2\,\mathrm{vc}^*(G)$ are integers and, consequently, we obtain the following strengthening of (3.4):

$$\mathrm{vc}^*(G - v) \geq \mathrm{vc}^*(G) - \frac{1}{2} \quad \text{for every } v \in V(G). \tag{3.5}$$

By contradiction, let $\mathbf{x}$ be an optimum solution to LPVC(G) that is not the all-$\frac{1}{2}$-solution. As the all-$\frac{1}{2}$-solution is an optimum solution to LPVC(G), for some $v \in V(G)$ we have $x_v > \frac{1}{2}$. However, then $\mathbf{x}$, restricted to $G-v$, is a feasible solution to LPVC($G-v$) of cost $\mathbf{w}(\mathbf{x})-x_v = \mathrm{vc}^*(G)-x_v < \mathrm{vc}^*(G)-\frac{1}{2}$, a contradiction to (3.5). $\square$

Lemma 3.7 allows us to enhance Reduction VC.4 to the following form.

Reduction VC.5. Invoke the algorithm of Lemma 3.7 and, if an optimum solution $\mathbf{x}$ is returned, then apply Reduction VC.4 to G and solution $\mathbf{x}$.

If Reduction VC.5 is not applicable, then we know not only that the all-$\frac{1}{2}$-solution is an optimum solution to LPVC(G), but also that this is the *unique* optimum solution. This property is crucial for our branching algorithm, encapsulated in the next theorem.

Theorem 3.8. *There exists an algorithm solving* Vertex Cover Above LP *in time* $4^{k-\mathrm{vc}^*(G)} \cdot n^{\mathcal{O}(1)}$.

Proof. Our algorithm for Vertex Cover Above LP is almost the same as the branching algorithm described for Vertex Cover in Section 3.1. After Reduction VC.5 is applied exhaustively, we pick an arbitrary vertex v in the graph and branch on it. In other words, in one branch, we add v into the vertex cover, decrease k by 1, and delete v from the graph, and in the other branch, we add $N(v)$ into the vertex cover, decrease k by $|N(v)|$, and delete $N[v]$ from the graph. The correctness of this algorithm follows from the safeness of Reduction VC.5 and the fact that the branching is exhaustive.

Although the algorithm is very similar to the one of Section 3.1, we analyze our algorithm in a completely different way, using the measure $\mu(G,k) = k - \mathrm{vc}^*(G)$. In Lemma 3.6 we have proved that Reduction VC.4 does not change the measure; as Reduction VC.5 is in fact an application of Reduction VC.4 to a specific half-integral solution $\mathbf{x}$, the same holds for Reduction VC.5. Hence, it remains to analyze how the measure changes in a branching step.

Consider first the case when we pick v to the vertex cover, obtaining an instance $(G',k') = (G-v, k-1)$. We claim that $\mathrm{vc}^*(G') \geq \mathrm{vc}^*(G)-\frac{1}{2}$. Suppose that this is not the case. Let $\mathbf{x}'$ be an optimum solution to LPVC(G'). We have $\mathbf{w}(\mathbf{x}') \leq \mathrm{vc}^*(G) - 1$ and we can obtain an optimum solution $\mathbf{x}''$ to LPVC(G) by extending $\mathbf{x}'$ with a new variable $x_v = 1$ corresponding to v. But this contradicts our assumption that the all-$\frac{1}{2}$-solution is the unique optimum solution to LPVC(G). Hence, $\mathrm{vc}^*(G') \geq \mathrm{vc}^*(G) - \frac{1}{2}$, which implies that $\mu(G',k') \leq \mu(G,k) - \frac{1}{2}$.

In the second case, let $p = |N(v)|$. Recall that here the new instance is $(G',k') = (G - N[v], k-p)$. We claim that $\mathrm{vc}^*(G') \geq \mathrm{vc}^*(G) - p + \frac{1}{2}$, which would imply again $\mu(G',k') \leq \mu(G,k) - \frac{1}{2}$. Assume the contrary: let $\mathbf{x}'$ be an optimum solution to LPVC(G') and suppose that $\mathbf{w}(\mathbf{x}') \leq \mathrm{vc}^*(G) - p$. Define a feasible solution $\mathbf{x}''$ to LPVC(G) by extending $\mathbf{x}'$ with $x_v = 0$ and $x_u = 1$ for every $u \in N(v)$. Clearly, $\mathbf{w}(\mathbf{x}'') = \mathbf{w}(\mathbf{x}') + p$, thus $\mathbf{x}''$ is an optimum solution to LPVC(G), contradicting the assumption that the all-$\frac{1}{2}$-solution is the unique optimum solution.

We have shown that the preprocessing rule does not increase the measure $\mu(G, k)$, and that the branching step results in a $(\frac{1}{2}, \frac{1}{2})$ decrease in $\mu(G, k)$. As a result, we obtain recurrence $T(\mu) \leq 2T(\mu - \frac{1}{2})$ for the number of leaves in the search tree. This recurrence solves to $4^{\mu} = 4^{k-\mathrm{vc}^*(G)}$, and we obtain a $4^{(k-\mathrm{vc}^*(G))} \cdot n^{\mathcal{O}(1)}$-time algorithm for Vertex Cover Above LP. ☐

Let us now consider a different lower bound on the size of a minimum vertex cover in a graph, namely the size of a maximum matching. Observe that, if graph G contains a matching M, then for $k < |M|$ the instance (G, k) is a trivial no-instance of Vertex Cover. Thus, for a graph G with a large maximum matching (e.g., when G has a perfect matching) the FPT algorithm for Vertex Cover of Section 3.1 is not practical, as in this case k has to be quite large.

This leads to a second above guarantee variant of the Vertex Cover problem, namely the Vertex Cover Above Matching problem. On input, we are given an undirected graph G, a maximum matching M and a positive integer k. As in Vertex Cover, the objective is to decide whether G has a vertex cover of size at most k; however, now the parameter is $k - |M|$. By the weak duality of linear programs, it follows that $\mathrm{vc}^*(G) \geq |M|$ (see Exercise 2.24 and the corresponding hint for a self-contained argument) and thus we have that $k - \mathrm{vc}^*(G) \leq k - |M|$. Consequently, any parameterized algorithm for Vertex Cover Above LP is also a parameterized algorithm for Vertex Cover Above Matching, and Theorem 3.8 yields the following interesting observation.

Theorem 3.9. Vertex Cover Above Matching *can be solved in time* $4^{k-|M|} \cdot n^{\mathcal{O}(1)}$.

The Vertex Cover Above Matching problem has been at the center of many developments in parameterized algorithms. The reason is that faster algorithms for this problem also yield faster algorithms for a host of other problems. Just to show its importance, we design algorithms for Odd Cycle Transversal and Almost 2-SAT by making use of the algorithm for Vertex Cover Above Matching.

A subset $X \subseteq V(G)$ is called an *odd cycle transversal* of G if $G - X$ is a bipartite graph. In Odd Cycle Transversal, we are given an undirected graph G with a positive integer k, and the goal is to determine whether G has an odd cycle transversal of size at most k.

For a given graph G, we define a new graph $\widetilde{G}$ as follows. Let $V_i = \{u_i : u \in V(G)\}$ for $i \in \{1, 2\}$. The vertex set $V(\widetilde{G})$ consists of two copies of $V(G)$, i.e. $V(\widetilde{G}) = V_1 \cup V_2$, and

$$E(\widetilde{G}) = \{u_1u_2 \ : \ u \in V(G)\} \cup \{u_iv_i \ : \ uv \in E(G), i \in \{1, 2\}\}.$$

In other words, $\widetilde{G}$ is obtained by taking two disjoint copies of G and by adding a perfect matching such that the endpoints of every matching edge are the copies of the same vertex. In particular, $M = \{u_1u_2 \,:\, u \in V(G)\}$ is a perfect matching in $\widetilde{G}$ and $|M| = n$. The graph $\widetilde{G}$ has some interesting properties which are useful for designing an algorithm for Odd Cycle Transversal.

Lemma 3.10. *Let G be a graph on n vertices. Then G has an odd cycle transversal of size at most k if and only if $\widetilde{G}$ has a vertex cover of size at most $n + k$.*

Proof. Let X be an odd cycle transversal of size at most k in G, and let (A, B) be a bipartition of $G - X$. For $i \in \{1, 2\}$, let A_i, B_i and X_i be the copies of A, B and X in V_i, respectively. Observe that $A_1 \cup B_2$ is an independent set in $\widetilde{G}$ and thus $A_2 \cup B_1 \cup X_1 \cup X_2$ is a vertex cover of $\widetilde{G}$. However,

$$(|A_2| + |B_1|) + |X_1| + |X_2| \leq (n - k) + k + k \leq n + k.$$

This completes the forward direction of the proof.

In the other direction, let Y be a vertex cover of $\widetilde{G}$ of size at most $n + k$. Then $I = V(\widetilde{G}) \setminus Y$ is an independent set of $\widetilde{G}$ of size at least $n - k$. Let $\widetilde{A} = I \cap V_1$ and $\widetilde{B} = I \cap V_2$. Let A and B denote the vertices corresponding to $\widetilde{A}$ and $\widetilde{B}$ in $V(G)$, respectively. Since $\widetilde{A} \cup \widetilde{B}$ is independent, we have that $A \cap B = \emptyset$. Therefore, $A \cup B$ is of size at least $n - k$. Moreover, $G[A \cup B]$ is bipartite, as $G[A]$ and $G[B]$ are independent sets. Thus, $V(G) \setminus (A \cup B)$ is an odd cycle transversal of size at most k for G. This completes the proof. □

Using Lemma 3.10 and Theorem 3.9 we get the following result for Odd Cycle Transversal.

Theorem 3.11. Odd Cycle Transversal *can be solved in time $4^k n^{\mathcal{O}(1)}$.*

Proof. Given an input (G, k) to Odd Cycle Transversal, we first construct the graph $\widetilde{G}$. Let $M = \{u_1u_2 \,:\, u \in V(G)\}$ be a perfect matching of $\widetilde{G}$. Now we apply Theorem 3.9 to the instance $(G, M, n+k)$ to check in time $4^{n+k-n} \cdot n^{\mathcal{O}(1)} = 4^k \cdot n^{\mathcal{O}(1)}$ whether $\widetilde{G}$ has a vertex cover of size at most $n+k$, where $n = |V(G)|$. If a vertex cover of size at most $n + k$ in $\widetilde{G}$ is found, then using Lemma 3.10 we obtain an odd cycle transversal of size at most k in G. Otherwise, by Lemma 3.10, G has no odd cycle transversal of size at most k. □

In fact, the fastest currently known FPT algorithm for Odd Cycle Transversal [328, 373] uses an improved version of the algorithm described in Theorem 3.9 as a subroutine. We will give a $3^k n^{\mathcal{O}(1)}$-time algorithm for this problem in the next chapter using a different technique, called iterative compression.

Next, we give an algorithm for Almost 2-SAT using the $4^k n^{\mathcal{O}(1)}$-time algorithm for Vertex Cover Above Matching. In the Almost 2-SAT problem, we are given a 2-CNF formula φ and an integer k, and the question is whether one can delete at most k clauses from φ to make it satisfiable.

In a variable-deletion variant, called Variable Deletion Almost 2-SAT, we instead allow deleting at most k variables; a variable is deleted together with all clauses containing it. (One can think of deleting a variable as setting it both to true and false at the same time, in this way satisfying all the clauses containing it.) Exercises 3.21 and 3.22 ask you to prove that these two variants are equivalent, and an $f(k)n^{\mathcal{O}(1)}$-time algorithm for one of them implies an $f(k)n^{\mathcal{O}(1)}$-time algorithm for the other one (with the same function f). Hence, it suffices to focus only on the variable-deletion variant.

Theorem 3.12. Variable Deletion Almost 2-SAT *can be solved in* $4^k n^{\mathcal{O}(1)}$ *time.*

Proof. We proceed similarly as in the case of Odd Cycle Transversal: in polynomial time, we construct a Vertex Cover Above Matching instance $(G, M, |M|+k)$ that is equivalent to the input Variable Deletion Almost 2-SAT instance (φ, k). Then the theorem follows from an application of the algorithm of Theorem 3.9 to the instance $(G, M, |M| + k)$.

We construct the graph G as follows. For every variable x, and for every literal $\ell \in \{x, \neg x\}$, we construct a vertex v_ℓ, and connect the vertices v_x and $v_{\neg x}$ by an edge. Let M be a set of edges defined in this step; note that M is a matching in G. The intuitive meaning of vertices v_ℓ is that picking v_ℓ into a vertex cover corresponds to valuating the variable of ℓ so that ℓ is true. For every edge $v_x v_{\neg x} \in M$, we expect to take one endpoint into a vertex cover, which corresponds to setting the value of variable x. However, in k edges $v_x v_{\neg x} \in M$ we are allowed to take both endpoints into a vertex cover, corresponding to deleting x or, equivalently, setting x both to true and false at once.

To encode clauses, we proceed as follows. For every binary clause $\ell_1 \vee \ell_2$, we add an edge $v_{\ell_1} v_{\ell_2}$, so that either v_{ℓ_1} or v_{ℓ_2} needs to be taken into a vertex cover. For every unary clause with literal ℓ, we add a new vertex of degree 1, adjacent to v_ℓ.

This concludes the construction of the graph G and the instance $(G, M, |M| +k)$. We refer to Fig. 3.1 for an illustration of the construction. We claim that (φ, k) is a yes-instance of Variable Deletion Almost 2-SAT if and only if $(G, M, |M| + k)$ is a yes-instance of Vertex Cover Above Matching.

In one direction, let X be a set of at most k variables of φ such that $\varphi - X$, the formula φ with all variables of X deleted, has a satisfying assignment ψ. We define a set $Y \subseteq V(G)$ as follows. First, for every $x \in X$, we put both v_x and $v_{\neg x}$ into Y. Second, for every variable $x \notin X$, we put v_x into Y if $\psi(x) = \top$ and otherwise, if $\psi(x) = \bot$, we put $v_{\neg x}$ into Y. Clearly $|Y| = |M| + |X| \leq |M| + k$.

It remains to argue that Y is a vertex cover of G. By construction, Y contains at least one endpoint of every edge in M. For the remaining edges, consider a clause C. Since X is a feasible solution to (φ, k), there exists a literal ℓ in C that is either evaluated to true by ψ, or its variable is deleted

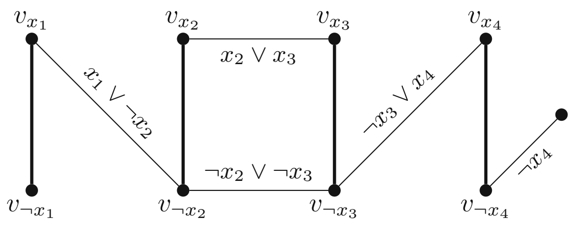


Fig. 3.1: The graph G for the formula $\varphi = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_4)$. The thick edges are the edges of the matching M

(belongs to X). In both cases, $v_\ell \in Y$ and the edge corresponding to the clause C is covered by Y.

In the other direction, let Y be a vertex cover of G, and assume $|Y| \leq |M| + k$. Without loss of generality, we may assume that $Y \subseteq V(M)$: if, for some unary clause $C = \ell$, the corresponding degree 1 neighbor of v_ℓ belongs to Y, we replace it with the vertex v_ℓ. Let X be the set of these variables x for which both v_x and $v_{\neg x}$ belong to Y. Since Y needs to contain at least one endpoint of every edge of M, we have $|X| \leq k$. Let us define an assignment ψ on the variables outside X as follows: $\psi(x) = \top$ if $v_x \in Y$, and $\psi(x) = \bot$ if $v_{\neg x} \in Y$. We claim that ψ is a satisfying assignment of $\varphi - X$; note that such a statement implies that X is a feasible solution to the Variable Deletion Almost 2-SAT instance (φ, k), concluding the proof of the theorem.

To this end, consider a clause C, and assume no variable of C belongs to X. If $C = \ell_1 \vee \ell_2$ is a binary clause, then the edge $v_{\ell_1} v_{\ell_2}$ ensures that either v_{ℓ_1} or v_{ℓ_2} belongs to Y, and the corresponding literal is evaluated to true in the assignment ψ. If $C = \ell$ is a unary clause, then the corresponding edge incident to v_ℓ, together with the assumption that $Y \subseteq V(M)$, implies that $v_\ell \in Y$ and ℓ is evaluated to true by ψ.

Consequently, (φ, k) is a yes-instance to Variable Deletion Almost 2-SAT if and only if $(G, M, |M| + k)$ is a yes-instance to Vertex Cover Above Matching. This concludes the proof of the theorem. $\square$

3.5 Closest String

In the last part of this chapter we give yet another example of branching algorithms, this time for a string problem called Closest String. Here, we are given k strings $x_1, \ldots, x_k$, each string over an alphabet Σ and of length L, and an integer d. The question is whether there exists a string y of length L over Σ such that $d_H(y, x_i) \leq d$ for all $i \in \{1, \ldots, k\}$. Here, $d_H(x, y)$ is the *Hamming distance* between strings x and y, that is, the number of positions where x and y differ. We call any such string y a *center string*. In this section we consider the parameterization by d, the maximum allowed distance between the center string and the input strings.

Let x be a string over alphabet Σ. We denote the letter on the pth position of x as $x[p]$. Thus $x = x[1]x[2]\cdots x[L]$ for a string of length L. We say that string x and y *differ* on the p-th position if $x[p] \neq y[p]$.

Given a set of k strings, each of length L, we can think of these strings as a $k \times L$ character matrix. By *columns* of the set of strings, we mean the columns of this matrix. That is, the j-th column is the sequence of letters $x_1[j], x_2[j], \ldots, x_k[j]$. We call a column *bad* if it contains at least two different symbols from alphabet Σ, and *good* otherwise. Clearly, if the j-th column is a good one, then we have an obvious greedy choice for the j-th letter of the solution: $y[j] = x_1[j] = x_2[j] = \ldots = x_k[j]$. Thus, we obtain the following reduction rule.

Reduction CS.1. Delete all good columns.

It is straightforward to implement Reduction CS.1 in linear time.

We now observe that we cannot have too many bad columns in a yes-instance.

Lemma 3.13. *For every yes-instance of* Closest String*, the corresponding $k \times L$ matrix contains at most kd bad columns.*

Proof. Fix a center string y. For every bad column j there exists a string $x_{i(j)}$ such that $x_{i(j)}[j] \neq y[j]$. Since every string x_i differs from y on at most d positions, for every i we have $i(j) = i$ for at most d positions j. Consequently, there are at most kd bad columns. □

Reduction CS.2. If there are more than kd bad columns, then conclude that we are dealing with a no-instance.

We now give an intuition behind the algorithm.

Fix a center string y. The idea of our algorithm is to start with one of the given strings, say x_1, as a “candidate string”, denoted by z. As long as there is a string x_i, $i \in \{1, \ldots, k\}$, such that $d_H(x_i, z) \geq d + 1$, then for at least one of the positions p where z and x_i differ, we have that $y[p] = x_i[p]$. Thus we can try recursively $d + 1$ ways to move the candidate z string “closer” to y; moving closer here means that we select a position p on which the candidate string z and x_i differ and set $z[p] := x_i[p]$. As at every step we move closer to y, and at the beginning $z = x_1$ and $d_H(x_1, y) \leq d$, we obtain a bounded number of possibilities.

Let us move to formal details.

Theorem 3.14. Closest String *can be solved in time $\mathcal{O}(kL + kd(d+1)^d)$.*

Proof. First, we apply Reductions CS.1 and CS.2. Unless we have resolved the instance already, we are left with k strings $x_1, x_2, \ldots, x_k$, each of length $L \leq kd$. The time spent so far is $\mathcal{O}(kL)$, that is, linear in the input size.

Recursively, we solve the following augmented problem: given a candidate string z and an integer $\ell \leq d$, determine whether there exists a center string y for the strings $x_1, x_2, \ldots, x_k$ with the additional property $d_H(y, z) \leq \ell$. Observe that, to solve Closest String on our instance, it suffices to solve our augmented problem for $z = x_1$ and $\ell = d$.

Given z and ℓ, we first perform the following checks. If z is a center string itself, then we return z. Otherwise, if $\ell = 0$, then we return that there is no such center string y. In the remaining case, $\ell > 0$ and there exists a string x_i with $d_H(x_i, z) > d$. Let $\mathcal{P}$ be a set of arbitrary $d+1$ positions on which x_i and z differ. Observe that for every center string y we have $y[p] = x_i[p]$ for at least one position $p \in \mathcal{P}$. Hence, we branch into $|\mathcal{P}| = d + 1$ subcases: for every $p \in \mathcal{P}$, we define z_p to be equal z except for position p where $z_p[p] = x_i[p]$, and we recursively solve our augmented problem for the pair $(z_p, \ell - 1)$. To show correctness of this branching, observe that if there exists a center string y with $d_H(z, y) \leq \ell$, then for a position $p \in \mathcal{P}$ satisfying $x_i[p] = y[p]$ we have $d_H(z_p, y) \leq d_H(z, y) - 1 \leq \ell - 1$.

Concerning the running time of the algorithm, note that we build a search tree of depth at most d, and every node of the search tree has at most $d + 1$ children. Thus, the size of the search tree does not exceed $\mathcal{O}((d+1)^d)$. With small technical work which we omit here, every step of the algorithm can be implemented in linear time. This completes the proof. ☐

Exercises

3.1 (✎). Prove Proposition 3.1.

3.2 (✎). Show that Clique and Independent Set, parameterized by the solution size k, are FPT on r-regular graphs for every fixed integer r. Also show that these problems are FPT with combined parameters $k + r$.

3.3. Show that any graph has at most 2^k inclusion-wise minimal vertex covers of size at most k. Furthermore, show that given G and k, we can enumerate all inclusion-wise minimal vertex covers of G of size at most k in time $2^k n^{\mathcal{O}(1)}$.

3.4 (✎). In the Cluster Vertex Deletion problem, we are given a graph G and an integer k, and the task is to delete at most k vertices from G to obtain a cluster graph (a disjoint union of cliques). Obtain a $3^k n^{\mathcal{O}(1)}$-time algorithm for Cluster Vertex Deletion.

3.5 (✎). In the Cluster Editing problem, we are given a graph G and an integer k, and the objective is to check whether we can turn G into a cluster graph (a disjoint union of cliques) by making at most k edge editions, where each edition is adding or deleting one edge. Obtain a $3^k n^{\mathcal{O}(1)}$-time algorithm for Cluster Editing.

3.6 (✎). An undirected graph G is called *perfect* if for every induced subgraph H of G, the size of the largest clique in H is the same as the chromatic number of H. In this exercise we consider the Odd Cycle Transversal problem, restricted to perfect graphs.

Recall that Exercise 2.34 asked for a kernel with $\mathcal{O}(k)$ vertices for this problem. In this exercise, we ask for a $3^k n^{\mathcal{O}(1)}$-time branching algorithm.

3.7. Let $\mathcal{F}$ be a set of graphs. We say that a graph G is $\mathcal{F}$*-free* if G does not contain any induced subgraph isomorphic to a graph in $\mathcal{F}$; in this context the elements of $\mathcal{F}$ are sometimes called *forbidden induced subgraphs*. For a fixed set $\mathcal{F}$, consider a problem where, given a graph G and an integer k, we ask to turn G into a $\mathcal{F}$-free graph by:

(vertex deletion) deleting at most k vertices;
(edge deletion) deleting at most k edges;
(completion) adding at most k edges;
(edition) performing at most k editions, where every edition is adding or deleting one edge.

Prove that, if $\mathcal{F}$ is finite, then for each of the four aforementioned problems there exists a $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$-time FPT algorithm. (Note that the constants hidden in the $\mathcal{O}()$-notation may depend on the set $\mathcal{F}$.)

3.8. In the Vertex Cover/OCT problem, we are given an undirected graph G, an integer ℓ, and an odd cycle transversal Z of size at most k, and the objective is to test whether G has a vertex cover of size at most ℓ. Show that Vertex Cover/OCT admits an algorithm with running time $2^k n^{\mathcal{O}(1)}$.

3.9. In this exercise we consider FPT algorithms for Feedback Arc Set in Tournaments and Feedback Vertex Set in Tournaments. Recall that a *tournament* is a directed graph, where every pair of vertices is connected by exactly one directed edge (in one of the directions).

1. Let G be a digraph that can be made into a tournament by adding at most $k \geq 2$ directed edges. Show that if G has a cycle then it has a directed cycle of length at most $3\sqrt{k}$.
2. Show that Feedback Arc Set in Tournaments admits a branching algorithm with running time $(3\sqrt{k})^k n^{\mathcal{O}(1)}$.
3. Show that Feedback Vertex Set in Tournaments admits a branching algorithm with running time $3^k n^{\mathcal{O}(1)}$.
4. Observe that, in the Feedback Arc Set in Tournaments problem, we can equivalently think of *reversing* an edge instead of deleting it. Use this observation to show a branching algorithm for Feedback Arc Set in Tournaments with running time $3^k n^{\mathcal{O}(1)}$.

3.10 (✎). A *bipartite tournament* is an orientation of a complete bipartite graph, meaning its vertex set is a union of two disjoint sets V_1 and V_2 and there is exactly one arc between every pair of vertices u and v such that $u \in V_1$ and $v \in V_2$.

1. Show that a bipartite tournament has a directed cycle if and only if it has a directed cycle on four vertices.
2. Show that Directed Feedback Vertex Set and Directed Feedback Arc Set admit algorithms with running time $4^k n^{\mathcal{O}(1)}$ on bipartite tournaments.

3.11 (☠). A graph is *chordal* if it does not contain a cycle on at least four vertices as an induced subgraph. A *triangulation* of a graph G is a set of edges whose addition turns G into a chordal graph. In the Chordal Completion problem, given a graph G and an integer k, we ask whether G admits a triangulation of size at most k.

1. Show that Chordal Completion admits an algorithm with running time $k^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$.
2. Show that there is a one-to-one correspondence between inclusion-wise minimal triangulations of a cycle with ℓ vertices and binary trees with $\ell - 2$ internal nodes. Using this correspondence show that a cycle on ℓ vertices has at most $4^{\ell-2}$ minimal triangulations.
3. Use the previous point to obtain an algorithm with running time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ for Chordal Completion.

You may use the fact that, given a graph G, one can in polynomial time check if G is a chordal graph and, if this is not the case, find in G an induced cycle on at least four vertices.

3.12. In the Min-Ones-r-SAT problem, we are given an r-CNF formula ϕ and an integer k, and the objective is to decide whether there exists a satisfying assignment for ϕ with at most k variables set to true. Show that Min-Ones-r-SAT admits an algorithm with running time $f(r,k)n^{\mathcal{O}(1)}$ for some computable function f.

3.13. In the Min-2-SAT problem, we are given a 2-CNF formula ϕ and an integer k, and the objective is to decide whether there exists an assignment for ϕ that satisfies *at most* k clauses. Show that Min-2-SAT can be solved in time $2^k n^{\mathcal{O}(1)}$.

3.14. In the Minimum Maximal Matching problem, we are given a graph G and an integer k, and the task is to check if G admits an (inclusion-wise) maximal matching with at most k edges.

1. Show that if G has a maximal matching of size at most k, then $V(M)$ is a vertex cover of size at most $2k$.
2. Let M be a maximal matching in G and let $X \subseteq V(M)$ be a minimal vertex cover in G. Furthermore, let M_1 be a maximum matching of $G[X]$ and M_2 be a maximum matching of $G[V(G) \setminus V(M_1)]$. Show that $M_1 \cup M_2$ is a maximal matching in G of size at most $|M|$.
3. Obtain a $4^k n^{\mathcal{O}(1)}$-time algorithm for Minimum Maximal Matching.

3.15 (☠). In the Max Leaf Spanning Tree problem, we are given a connected graph G and an integer k, and the objective is to test whether there exists a spanning tree of G with at least k leaves. Obtain an algorithm with running time $4^k n^{\mathcal{O}(1)}$ for Max Leaf Spanning Tree.

3.16 (☠). An *out-tree* is an oriented tree with only one vertex of indegree zero called the *root*. In the Directed Max Leaf problem, we are given an directed graph G and an integer k, and the objective is to test whether there exists an out-tree in G with at least k leaves (vertices of outdegree zero). Show that Directed Max Leaf admits an algorithm with running time $4^k n^{\mathcal{O}(1)}$.

3.17. Describe an algorithm running in time $\mathcal{O}(1.381^n)$ which finds the *number* of independent sets (or, equivalently, vertex covers) in a given n-vertex graph.

3.18. Show that if a graph on n vertices has minimum degree at least 3, then it contains a cycle of length at most $2\lceil \log n \rceil$. Use this to design a $(\log n)^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$-time algorithm for Feedback Vertex Set on undirected graphs. Is this an FPT algorithm for Feedback Vertex Set?

3.19. Prove that the statement of Lemma 3.6 is true for any optimum solution $\mathbf{x}$, not necessarily a half-integral one. (For a not half-integral solution $\mathbf{x}$, we define $V_1^{\mathbf{x}} = \{v \in V(G) \ : \ x_v > \frac{1}{2}\}$ and $V_0^{\mathbf{x}} = \{v \in V(G) \ : \ x_v < \frac{1}{2}\}$.)

3.20. A graph G is a *split graph* if $V(G)$ can be partitioned into sets C and I, such that C is a clique and I is an independent set in G. In the Split Vertex Deletion problem, given a graph G and an integer k, the task is to check if one can delete at most k vertices from G to obtain a split graph.

Give a polynomial-time algorithm that, given a Split Vertex Deletion instance (G, k), produces an equivalent Vertex Cover Above Matching instance $(G', M, |M| + k)$. Use this to design a $4^k n^{\mathcal{O}(1)}$-time algorithm for Split Vertex Deletion.

3.21. Show how to, given an Almost 2-SAT instance (φ, k), compute in polynomial time an equivalent instance (φ', k) of Variable Deletion Almost 2-SAT.

3.22. Show the reverse of the previous exercise. That is, show how to, given an instance (φ, k) of Variable Deletion Almost 2-SAT, compute in polynomial time an equivalent instance (φ', k) of Almost 2-SAT.

3.23 (☠). For an independent set I in a graph G, the *surplus* of I is defined as $|N(I)| - |I|$.

1. Show that a graph reduced with respect to Reduction VC.5 (i.e., this reduction cannot be further applied) does not admit independent sets with nonpositive surplus.
2. Show how to detect independent sets of surplus 1 in such a graph using the LP relaxation of Vertex Cover.
3. Design a reduction rule for Vertex Cover that handles a (given) independent set of surplus 1. How does the measure $\mu(G, k) = k - \text{vc}^*(G)$ behave in your reduction rule?
4. Show that, if a graph does not admit an independent set of surplus at most 1, then the branching of Theorem 3.8 has branching vector $(\frac{1}{2}, 1)$, and the algorithm runs in time $2.6181^{k-\text{vc}^*(G)} n^{\mathcal{O}(1)}$.

3.24 (🍃). Consider the Closest String problem, where in the input there are two strings x_i and x_j with $d_H(x_i, x_j) = 2d$. Show that in this case the Closest String problem can be solved in time $\mathcal{O}(kL + kd4^d)$.

3.25 (☠). Consider a generalization of Closest String where, apart from the strings $x_1, x_2, \ldots, x_k$, each over alphabet Σ and of length L, we are given integers $d_1, d_2, \ldots, d_k$, and we ask for a center string y of length L such that $d_H(y, x_i) \leq d_i$ for every $1 \leq i \leq k$. Consider the following recursive algorithm for this generalization.

1. First, try if x_1 is a center string. If this is the case, then return x_1 and finish.
2. Otherwise, if $d_1 = 0$, then return that there is no solution.
3. In the remaining case, take any $j \geq 2$ such that $d(x_1, x_j) > d_j$. Let $\mathcal{P}$ be the set of positions on which x_1 and x_j differ. If $|\mathcal{P}| > d_1 + d_j$, return that there is no solution.
4. Guess the values $y[p]$ for every $p \in \mathcal{P}$, where y is the center string we are looking for, in such a way that y differs from x_i on at most d_i positions from $\mathcal{P}$ for every $1 \leq i \leq k$.
5. For every guess $(y[p])_{p \in \mathcal{P}}$, define $d'_i = d_i - |\{p \in \mathcal{P} \ : \ y[p] \neq x_i[p]\}|$ and let x'_i be the string x_i with letters on positions of P deleted.
6. Observe that $x'_1 = x'_j$. Discard the string x'_j, and set $d'_1 := \min(d'_1, d'_j)$.
7. Recurse on the strings x'_i and integers d'_i, for every guess $(y[p])_{p \in \mathcal{P}}$.

Denote $d = \max_{1 \leq i \leq k} d_i$.

1. Show that in every recursive step it holds that $d'_1 < d_1/2$.
2. Use this fact to show that this algorithms solves the generalization of Closest String in time $2^{\mathcal{O}(d)} |\Sigma|^d (kL)^{\mathcal{O}(1)}$.

Hints

3.3 Consider the following branching algorithm: given an instance (G, k), pick an arbitrary edge $uv \in E(G)$ and branch on two instances $(G - u, k - 1)$ and $(G - v, k - 1)$. Stop when G becomes edgeless or $k < 0$. Show that in this manner you generate at most 2^k leaves of the search tree, and every minimal vertex cover of G of size at most k appears in some leaf.

3.4 Observe that a graph is a cluster graph if and only if it does not contain a path on three vertices (P_3) as an induced subgraph. As long as there exists a P_3 as an induced subgraph, branch by choosing one of its vertices to delete.

3.5 Proceed as in Exercise 3.4, and observe that you can break a P_3 in three different ways.

3.6 Observe that the class of perfect graphs is closed under taking induced subgraphs, and a perfect graph is bipartite if and only if it does not contain a triangle.

3.7 Observe that, if $\mathcal{F}$ is finite and fixed, then

1. we can in polynomial time verify if a graph G is $\mathcal{F}$-free and, if not, find an induced subgraph of G that is isomorphic to a member of $\mathcal{F}$;
2. there are only $\mathcal{O}(1)$ ways to break such a forbidden induced subgraph in all four considered problem variants.

3.8 For every $v \in Z$, branch on whether to include v or $N(v)$ into the desired vertex cover. Observe that the remaining graph is bipartite.

3.9 For the first point, observe that a shortest cycle in a directed graph cannot have any chord. Moreover, if ℓ is the length of some chordless cycle in D, then there are at least $\ell(\ell - 3)/2$ pairs of nonadjacent vertices in D. To solve the second point, show how to find such a chordless cycle, and branch choosing which edge to delete from it.

For the third and fourth point, observe that a tournament is acyclic if and only if it does not contain a directed triangle.

3.10 Use again the observation that in the Directed Feedback Arc Set problem you can alternatively reverse edges instead of deleting them. Thus, the graph remains a bipartite tournament in the course of the algorithm, and every directed cycle on four vertices can be broken in exactly four ways in both considered problems.

3.11 First, design an algorithm that either verifies that G is a chordal graph, or finds an induced cycle on at least four vertices in G. Second, observe that such a cycle on more than $k + 3$ vertices is a certificate that (G, k) is a no-instance. For the first point, it suffices to branch choosing one edge to add to the cycle found. For the last point, the branching should consider all minimal triangulations of the cycle.

3.12 Start with an assignment that sets all variables to false. As long as there exists an unsatisfied clause C, branch on C, in each subcases choosing one positive literal of C to be satisfied. In this way, you obtain an $r^k n^{\mathcal{O}(1)}$-time algorithm.

3.13 First, observe that if some variable appears only positively (or negatively), then we can assign its value in such a way that it does not satisfy any clause. Second, note that if the aforementioned preprocessing rule is not applicable, then branching on any variable makes at least one clause satisfied, decreasing the budget k by at least 1.

3.14 The first point is straightforward. The fact that $M_1 \cup M_2$ is a maximal matching follows directly from the fact that M_2 is a maximum matching in $G - V(M_1)$. For the cardinality bound, let $M_X \subseteq M$ be the set of edges of M that have both endpoints in

X. Observe that $|M| = |M_X| + |M \setminus M_X| = |M_X| + (|X| - 2|M_X|) = |X| - |M_X|$, while $|M_1| \geq |M_X|$ and $|M_2| \leq |X| - |V(M_1)| = |X| - 2|M_1|$. For the last point, use Exercise 3.3 to enumerate all minimal vertex covers of G of size at most $2k$.

3.15 Start with the following observation: if G is connected and has a subtree T, rooted at some vertex r, with at least k leaves, then G has also a spanning tree with at least k leaves. In our algorithm, we start by guessing a non-leaf vertex r of the tree in question, and consider all further trees as rooted in r.

At every node of the search tree we have a subtree T of G rooted at node r. Let $L(T)$ denote the set of leaves of T. Our objective in this branch is to find a subtree with k leaves (if it exists) that is a supergraph of T.

First, observe that if there exists a leaf $v \in L(T)$ with at least two neighbors in $V(G) \setminus V(T)$, then we can branch on v. In one subcase, we consider v being an internal vertex of the tree in question, adding all edges vu for $u \in N(v) \setminus V(T)$ to the tree T. In the second subcase, we consider v being a leaf vertex of the tree in question, deleting all edges vu for $u \in N(v) \setminus V(T)$ from the graph G. Show that this branching is correct. Observe that in every branching step we either fix one vertex to be a leaf, or increase the number of leaves of T. Hence, this branching leads to at most 2^{2k} leaves of the search tree.

We are left with the case when all vertices $v \in L(T)$ have at most one neighbor outside $V(T)$. Prove that in this case a similar branching is still valid: for a vertex $v \in L(T)$, either proclaim v a leaf, or fix as internal vertices both v and all its descendants, up to the closest vertex of degree at least three, inclusive.

In the leaves of the search tree we have trees T with $|L(T)| \geq k$ (where we report that (G, k) is a yes-instance), or trees T where all the leaves have been fixed as leaves, but still $|L(T)| < k$ (where we report no solution in this branch).

3.16 Adapt the solution of Exercise 3.15 to the directed case.

3.17 Note that there is a bijection between vertex covers and maximal independent sets, since the complement of a vertex cover is an independent set. Show a *counting* counterpart of Proposition 3.1 and design a branching algorithm where the number of leaves in the search tree is bounded by the solution of the recurrence $T(n) = T(n-1) + T(n-4)$. Solve the recurrence by applying the techniques described in Section 3.2 (use Table 3.1).

3.18 To prove the desired upper bound on the length of the shortest cycle, consider a breadth-first search tree of the input graph.

To show that the obtained algorithm is an FPT one, consider two possibilities. If $n \leq k^k$, then $\log n \leq k \log k$ and $(\log n)^k \leq 2^{\mathcal{O}(k \log k)}$. Otherwise, $k = \mathcal{O}(\log n / \log\log n)$ and $(\log n)^k = 2^{\mathcal{O}(\log n)} = n^{\mathcal{O}(1)}$. Alternatively, you may use the Cauchy-Schwarz inequality to observe that

$$(\log n)^k = 2^{k \log\log n} \leq 2^{\frac{k^2 + (\log\log n)^2}{2}} = 2^{k^2/2} \cdot 2^{(\log\log n)^2/2} = 2^{k^2/2} \cdot n^{o(1)}.$$

3.19 Apply Exercise 2.25 to $\mathbf{x}$, obtaining a half-integral solution $\mathbf{y}$. Observe that Reduction VC.4 behaves the same way when fed with the solution $\mathbf{x}$ and with the solution $\mathbf{y}$.

3.20 In short, modify the reduction of Lemma 3.10.

For a given a graph G, we define a new graph $\widetilde{G}$ as follows. Let $V_i = \{u_i \ : \ u \in V(G)\}$, $i \in \{1, 2\}$. The vertex set $V(\widetilde{G})$ consists of two copies of $V(G)$, i.e., $V(\widetilde{G}) = V_1 \cup V_2$ and

$$E(\widetilde{G}) = \{u_1 u_2 \ : \ u \in V(G)\} \cup \{u_1 v_1 \ : \ uv \in E(G)\} \cup \{u_2 v_2 \ : \ uv \notin E(G)\}.$$

In other words, $\widetilde{G}$ is obtained by taking a disjoint copy of G and a complement of G, and adding a perfect matching such that the endpoints of every matching edge are copies of the same vertex.

3.21 The main idea is to make, for every variable x, a different copy for every appearance of x in the formula φ. To create the formula φ', we first copy all the clauses of φ, replacing every variable x with one of its copies so that no copy is used twice, and then we add clauses enforcing equality between every pair of copies of the same variable. Show that a deletion of a clause C in the original instance corresponds to the deletion of one of the copies of a variable appearing in C in the new instance.

3.22 The main idea is to replace every variable x with its two copies, $x^\top$ and $x^\perp$: $x^\top$ has the meaning "Is x true?", whereas $x^\perp$ has the meaning "Is x false?". We connect them by a clause $\neg x^\top \vee \neg x^\perp$ to enforce that only one value of x is chosen. Moreover, we replace every literal x with $x^\top$, and every literal $\neg x$ with $x^\perp$. Argue that in the new instance (φ', k) of Almost 2-SAT there exists a minimum solution that deletes only clauses of the form $\neg x^\top \vee \neg x^\perp$. Observe that deleting such a clause corresponds to setting x to both true and false at once in the input formula φ, satisfying all the clauses containing x.

3.23 Let I be an independent set in G of surplus a. Define the following vector $\mathbf{x} = (x_v)_{v \in V(G)}$: $x_v = 0$ for $v \in I$, $x_v = 1$ for $v \in N(I)$ and $x_v = \frac{1}{2}$ for $v \notin N[I]$. Observe that $\mathbf{x}$ is a feasible solution to LPVC(G) of cost $\frac{|V(G)|}{2} + \frac{a}{2}$. This proves the first point.

Moreover, it also proves the following: if we pick any vertex v and solve LPVC(G) with an additional constraint $x_v = 0$ (in other words, solve LPVC($G - N[v]$) and extend it with $x_v = 0$ and $x_u = 1$ for every $u \in N(v)$), obtaining a half-integral solution $\mathbf{x}$, then the set $V_0^{\mathbf{x}}$ is an independent set of minimum surplus among all independent sets containing v. In particular, this gives us a way to detect independent sets with surplus 1, as well as shows that, if the minimum surplus of an independent set containing v is at least 2, then in a branch where we exclude v from a vertex cover, the measure $\mu(G, k)$ drops by at least 1.

It remains to argue about the third point, tackling with a reduction rule. To this end, prove the following: if Reduction VC.5 is not applicable, and I is of surplus 1, then there exists a minimum vertex cover of G that contains the whole I or the whole $N(I)$. Recall $|I| + 1 = |N(I)|$. If $G[N(I)]$ contains an edge, then we can greedily take $N(I)$ into the vertex cover. Otherwise, we can replace $N[I]$ with a single vertex incident to $N(N[I])$ and decrease k by $|I|$. A direct check shows that the measure $\mu(G, k)$ does not increase in these steps.

3.24 Observe that, in this case, for every center string y, $y[p]$ equals $x_i[p]$ whenever $x_i[p] = x_j[p]$, whereas for every position p with $x_i[p] \neq x_j[p]$ we have $y[p] = x_i[p]$ or $y[p] = x_j[p]$. Thus, there are two options for each of $2d$ positions where x_i and x_j differ.

3.25 For the first point, consider two cases depending on for how many positions $p \in \mathcal{P}$ the character $y[p]$ has been guessed so that $y[p] = x_j[p]$. If there are more than $d_1/2$ such positions, then x_1 differs on all of them from y, and we have $d_1' < d_1/2$. Show that in the second case, when there are at most $d_1/2$ such positions, it holds that $d_j' < d_1/2$. Use here the fact that $|\mathcal{P}| > d_j$.

For the second point, think of choosing $(y[p])_{p \in \mathcal{P}}$ as a three stage process. First, we choose an integer $0 \leq \ell \leq d_1$: y will differ from x_1 on exactly ℓ positions of $\mathcal{P}$. Second, we choose the set $\mathcal{Q}$ of these ℓ positions; there are $\binom{|\mathcal{P}|}{\ell} \leq \binom{d+d_1}{\ell}$ possibilities. Third, we choose $y[p]$ for every $p \in \mathcal{Q}$; there are $(|\Sigma| - 1)^\ell$ possibilities.

Using d_1 as a measure for the complexity of an instance, and using the first point, we obtain the following recurrence for time complexity.

$$\begin{aligned} T(d_1) &\leq \sum_{\ell=0}^{d_1} \binom{d+d_1}{\ell} (|\Sigma| - 1)^\ell \cdot T(\min(d_1 - \ell, d_1/2)) \\ &= \sum_{\ell=0}^{\lfloor d_1/2 \rfloor} \binom{d+d_1}{\ell} (|\Sigma| - 1)^\ell \cdot T(d_1/2) + \sum_{\ell=\lfloor d_1/2 \rfloor + 1}^{d_1} \binom{d+d_1}{\ell} (|\Sigma| - 1)^\ell \cdot T(d_1 - \ell). \end{aligned}$$

It remains to use the aforementioned recurrence to prove by induction that

$$T(d_1) \leq \binom{d+d_1}{d_1} 2^{6d_1} (|\Sigma|-1)^{d_1}.$$

(Constant 6 in the exponent is not optimal; it has been chosen so that the proof goes smoothly.)

Bibliographic notes

The history of branching algorithms can be traced back to the work of Davis and Putnam [126] (see also [125]) on the design and analysis of algorithms to solve some satisfiability problems. Solving linear recurrences or linearly recurrent sequences is the subject of many textbooks in discrete mathematics including the books of Rosen [406] and Graham, Knuth, and Patashnik [232], see also [300, p. 88]. Branching algorithms and techniques for their analysis are discussed in detail in the book of Fomin and Kratsch [197].

A branching algorithm with running time $\mathcal{O}(2^k(n+m))$ for Vertex Cover appears in the book of Mehlhorn [363]. After a long sequence of improvements, the current champion is the algorithm of Chen, Kanj, and Xia, running in time $\mathcal{O}(1.2738^k+kn)$ [82]. The branching algorithm for Feedback Vertex Set is from [409]. Our algorithm for Closest String follows Gramm, Niedermeier, and Rossmanith [235].

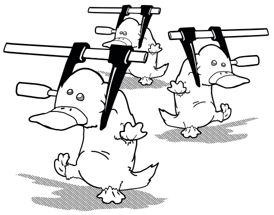
Parameterizations above and below guarantees were introduced by Mahajan and Raman [342]. Other interesting examples of algorithms for above and below guarantee parameterizations include [11, 104, 105, 107, 108, 247, 250, 290, 343].

Our algorithm for Vertex Cover Above LP, as well as Exercise 3.23, follows the outline from [373]. More involved reduction rules and a detailed analysis lead to an improved running time bound of $2.32^k n^{\mathcal{O}(1)}$ for Vertex Cover Above LP [328, 373]. The "LP-guided branching" paradigm was introduced by Cygan, Pilipczuk, Pilipczuk, and Wojtaszczyk [122] to solve Vertex Multiway Cut, and extended to a bigger set of problems by Wahlström [430]. The above algorithm is not only an example of the use of linear programming in parameterized algorithms, but also an example of the technique called "measure and conquer." In a more sophisticated use of this technique, one comes up with a measure which is a function of the input and the parameter, and whose objective is to measure the progress in each branching step. We refer to [194] and [197] for an introduction to measure and conquer, as well as its applications to the Vertex Cover and Dominating Set problems.

The observation that graph modification problems for hereditary graph classes with a finite set of forbidden induced subgraphs admit simple FPT algorithms (encapsulated as Exercise 3.7) is due to Cai [63].

Exercise 3.11 is from [281]; we note that there is also a subexponential parameterized algorithm for Chordal Completion [211]. Exercise 3.25 is from [341]. Exercises 3.15 and 3.16 are from [298].

Chapter 4 Iterative compression



In this chapter we introduce iterative compression, a simple yet very useful technique for designing fixed-parameter tractable algorithms. Using this technique we obtain FPT *algorithms for* Feedback Vertex Set in Tournaments*,* Feedback Vertex Set *and* Odd Cycle Transversal.

In 2004, Reed, Smith and Vetta [397] presented the first fixed-parameter tractable algorithm for Odd Cycle Transversal, running in time $3^k n^{\mathcal{O}(1)}$. This result is important not only because of the significance of the problem, but also because the proposed approach turned out to be a novel and generic technique, applicable in many other situations. Based on this new technique, called nowadays *iterative compression*, a number of FPT algorithms for several important problems have been obtained. Besides Odd Cycle Transversal, examples include Directed Feedback Vertex Set and Almost 2-SAT.

Typically, iterative compression algorithms are designed for parameterized minimization problems, where the goal is to find a small set of vertices or edges of the graph, whose removal makes the graph admit some global property. The upper bound on the size of this set is the parameter k. The main idea is to employ a so-called *compression routine*. A compression routine is an algorithm that, given a problem instance and a corresponding solution, either calculates a smaller solution or proves that the given solution is of the minimum size. Using a compression routine, one finds an optimal solution to the problem by iteratively building up the structure of the instance and compressing intermediate solutions.

The main point of the iterative compression technique is that if the compression routine runs in FPT time, then so does the whole algorithm. The strength of iterative compression is that it allows us to see the problem from a different viewpoint: The compression routine has not only the problem instance as input, but also a solution, which carries valuable structural information about the instance. Therefore, constructing a compression routine may be simpler than designing a direct FPT algorithm for the original problem.

While embedding the compression routine into the iteration framework is usually straightforward, finding the compression routine itself is not. Therefore, the art of iterative compression typically lies in the design of the compression routine. In this chapter we design algorithms for Feedback Vertex Set in Tournaments, Feedback Vertex Set and Odd Cycle Transversal using the method of iterative compression. An important case of Almost 2-SAT is covered in the exercises. This technique will be also applied in Chapter 7 to solve Planar Vertex Deletion, and in Chapter 8 for Directed Feedback Vertex Set.

4.1 Illustration of the basic technique

The technique described in this section is based on the following strategy.

Solution compression: First, apply some simple trick so that you can assume that a slightly too large solution is available. Then exploit the structure it imposes on the input graph to construct an optimal solution.

As a simple example of this approach, let us try to apply it to our favourite example problem Vertex Cover. The algorithm we are going to obtain now is much worse than the one obtained in the previous chapter, but it serves well for the illustration. Assume we are given a Vertex Cover instance (G, k). We use the well-known 2-approximation algorithm to obtain an approximate vertex cover Z. If $|Z| > 2k$, then we can clearly conclude that (G, k) is a no-instance, so assume otherwise. We are now going to exploit the structure that Z imposes on the graph G: Z is small, so we can afford some branching on Z, while at the same time $G - Z$ is edgeless.

The branching step is as follows: we branch in all possible ways an optimal solution X can intersect Z. Let $X_Z \subseteq Z$ be one such guess. We are searching now for a vertex cover X of size at most k, such that $X \cap Z = X_Z$. Let $W = Z \setminus X_Z$. If there is an edge in $G[W]$, then we can clearly conclude that our guess of X_Z was wrong. Otherwise, note that any vertex cover X satisfying $X \cap Z = X_Z$ needs to include $X_Z \cup N_G(W)$. Furthermore, since $G - Z$ is an independent set, $X_Z \cup N_G(W)$ is actually a vertex cover of G. Consequently, if for some choice of $X_Z \subseteq Z$ we have $|X_Z \cup N_G(W)| \leq k$, then we return a solution $X := X_Z \cup N_G(W)$, and otherwise we conclude that (G, k) is a no-instance. Thus, we obtain an algorithm solving Vertex Cover in time $2^{|Z|} n^{\mathcal{O}(1)} \leq 4^k n^{\mathcal{O}(1)}$.

Let us go back to the point when we have guessed the set X_Z and defined $W = Z \setminus X_Z$. In the previous paragraph we have in fact argued that the following Disjoint Vertex Cover problem is polynomial-time solvable: Does $G - X_Z$ contain a vertex cover of size at most $k - |X_Z|$ that is disjoint from W? Note here that W is a vertex cover of $G - X_Z$, giving us a lot of insight into the structure of $G - X_Z$.

In our example, the final dependency of the running time on k is much worse than the one obtained by branching algorithms in the previous chapter. One of the reasons is that we started with a large set Z, of size at most $2k$. Fortunately, there is an easy and very versatile way to obtain a set Z of size $k + 1$. This is exactly the main trick of *iterative compression*.

As the name suggests, in iterative compression we apply the compression step iteratively. To exemplify this idea on Vertex Cover, let us take an arbitrary ordering $(v_1, v_2, \ldots, v_n)$ of G. For $i \in \{1, \ldots, k\}$, we denote by G_i the subgraph of G induced by the first i vertices. For $i = k$, we can take the vertex set of G_i as a vertex cover X in G_i of size k. We proceed iteratively. Suppose that for some $i > k$, we have constructed a vertex cover X_i of G_i of size at most k. Then in graph G_{i+1}, set $Z_{i+1} := X_i \cup \{v_{i+1}\}$ is a vertex cover of size at most $k + 1$. If actually $|Z_{i+1}| \leq k$ then we are done: we can simply put $X_{i+1} = Z_{i+1}$ and proceed to the next iteration. Otherwise we have $|Z_{i+1}| = k + 1$, and we need to compress the too large solution. By applying the branching algorithm described above, i.e., solving $2^{|Z_{i+1}|} = 2^{k+1}$ instances of Disjoint Vertex Cover, in time $2^{k+1}n^{\mathcal{O}(1)}$, we can either find a vertex cover X_{i+1} of G_i of size at most k, or conclude that no such cover exists. If G_i does not admit a vertex of size at most k, then of course neither does G, and we may terminate the whole iteration and provide a negative answer to the problem. If X_{i+1} has been found, however, then we may proceed further to the graph G_{i+2} and so on. To conclude, observe that $G_n = G$, so at the last iteration we obtain a solution for the input Vertex Cover instance in time $2^k n^{\mathcal{O}(1)}$.

Combined with other ideas, this simple strategy becomes a powerful technique which can be used to solve different parameterized problems. Let us sketch how this method can be applied to a graph problem. The central idea here is to design an FPT algorithm which for a given $(k + 1)$-sized solution for a problem either compresses it to a solution of size at most k or proves that there is no solution of size at most k. This is known as the compression step of the algorithm. The method adopted usually is to begin with a subgraph that trivially admits a k-sized solution and then expand it iteratively. In any iteration, we try to find a compressed k-sized solution for the instance corresponding to the current subgraph. If we find such a solution, then by adding a vertex or an edge we obtain a solution to the next instance, but this solution can be too large by 1. To this solution we apply the compression step. We stop when we either obtain a solution of size at most k for the entire graph, or if some intermediate instance turns out to be incompressible. In order to stop in the case when some intermediate instance turns out to be incompressible, the problem must have the property that the optimum solution size in any intermediate instance is at most the optimum solution size in the whole graph.

4.1.1 A few generic steps

We start with explaining the generic steps of iterative compression that are common to most of the problems. We focus here on vertex subset problems; the steps for an edge subset problem are analogous. Suppose that we want to solve $(*)$-Compression, where the wildcard $(*)$ can be replaced by the name of the problem we are trying to solve. In $(*)$-Compression, as input we are given an instance of the problem $(*)$ with a solution of size $k+1$ and a positive integer k. The objective is to either find a solution of size at most k or conclude that no solution of size at most k exists. For example, for $(*)$ =Feedback Vertex Set, we are given a graph G and a feedback vertex set X of size $k+1$. The task is to decide whether G has a feedback vertex set of size at most k.

The first observation that holds for all the problems in this section is the following:

If there exists an algorithm solving $(*)$-Compression in time $f(k)\cdot n^c$, then there exists an algorithm solving problem $(*)$ in time $\mathcal{O}(f(k)\cdot n^{c+1})$.

We already explained how to prove such an observation for Vertex Cover. We will repeat it once again for Feedback Vertex Set in Tournaments and skip the proofs of this observation for Feedback Vertex Set and Odd Cycle Transversal.

Now we need to compress a solution Z of size $k+1$. In all our examples we follow the same strategy as we did for Vertex Cover. That is, for every $i \in \{0,\ldots,k\}$ and every subset X_Z of Z of size i, we solve the following Disjoint-$(*)$ problem: Either find a solution X to $(*)$ in $G - X_Z$ such that $|X| \leq k-i$, where X and $W := Z \setminus X_Z$ are disjoint, or conclude that this is impossible. Note that in the disjoint variant we have that $|W| = k-i+1$, so again the size of the solution W is one larger than the allowed budget; the difference now is that taking vertices from W is explicitly forbidden. We use the following observation, which follows from simple branching.

If there exists an algorithm solving Disjoint-$(*)$ in time $g(k)\cdot n^{\mathcal{O}(1)}$, then there exists an algorithm solving $(*)$-Compression in time

$$\sum_{i=0}^{k} \binom{k+1}{i} g(k-i) n^{\mathcal{O}(1)}.$$

In particular, if $g(k) = \alpha^k$, then $(*)$-Compression can be solved in time $(1+\alpha)^k n^{\mathcal{O}(1)}$.

Thus, the crucial part of the algorithms based on iterative compression lies in solving the disjoint version of the corresponding problem. We will provide the detailed proof of the above observation for Feedback Vertex Set in Tournaments. We will not repeat these arguments for Feedback Vertex Set and Odd Cycle Transversal, and explain only algorithms solving Disjoint-$(*)$ for these problems.

Finally, let us point out that when designing algorithms for the compression and disjoint versions of a problem, one cannot focus only on the decision version, where the task is just to determine whether a solution exists. This is because constructing an actual solution is needed to perform the next step of the iteration. This issue is almost never a real problem: either the algorithm actually finds the solution on the way, or one can use the standard method of self-reducibility to query a decision algorithm multiple times in order to reconstruct the solution. However, the reader should be aware of the caveat, especially when trying to estimate the precise polynomial factor of the running time of the algorithm.

4.2 Feedback Vertex Set in Tournaments

In this section we design an FPT algorithm for Feedback Vertex Set in Tournaments (FVST) using the methodology of iterative compression. In this problem, the input consists of a tournament T and a positive integer k, and the objective is to decide whether there exists a vertex set $X \subseteq V(T)$ of size at most k such that $T - X$ is a directed acyclic graph (equivalently, a transitive tournament). We call the solution X a *directed feedback vertex set*. Let us note that Feedback Vertex Set in Tournaments is a special case of Directed Feedback Vertex Set, where the input directed graph is restricted to being a tournament.

In what follows, using the example of Feedback Vertex Set in Tournaments we describe the steps that are common to most of the applications of iterative compression.

We first define the compression version of the problem, called Feedback Vertex Set in Tournaments Compression. In this problem, the input consists of a tournament T, a directed feedback vertex set Z of T of size $k+1$, and a positive integer k, and the objective is either to find a directed feedback vertex set of T of size at most k, or to conclude that no such set exists.

Suppose we can solve FEEDBACK VERTEX SET IN TOURNAMENTS COMPRESSION in time $f(k) \cdot n^{\mathcal{O}(1)}$. Given this, we show how to solve the original problem in $f(k) \cdot n^{\mathcal{O}(1)}$ time. We take an arbitrary ordering $(v_1, v_2, \ldots, v_n)$ of $V(T)$ and for every $i \in \{1, \ldots, n\}$ we define $V_i = \{v_1, \ldots, v_i\}$ and $T_i = T[V_i]$. Notice that

- V_k is a directed feedback vertex set of size k of T_k.
- If X is a directed feedback vertex set of T_i, then $X \cup \{v_{i+1}\}$ is a directed feedback vertex set of T_{i+1}.
- If T_i does not admit a directed feedback vertex set of size at most k, then neither does T.

These three facts together with an $f(k) \cdot n^c$-time algorithm for FEEDBACK VERTEX SET IN TOURNAMENTS COMPRESSION imply an $f(k) \cdot n^{c+1}$-time algorithm for FEEDBACK VERTEX SET IN TOURNAMENTS as follows. In tournament T_k set V_k is a directed feedback vertex set of size k. Suppose that for $i \geq k$ we have constructed a directed feedback vertex set X_i of T_i of size at most k. Then in T_{i+1}, set $Z_{i+1} := X_i \cup \{v_{i+1}\}$ is a directed feedback vertex set of size at most $k+1$. If actually $|Z_{i+1}| \leq k$, then we may proceed to the next iteration with $X_{i+1} = Z_{i+1}$. Otherwise we have $|Z_{i+1}| = k+1$. Then in time $f(k) \cdot n^c$ we can either construct a directed feedback vertex set X_{i+1} in T_{i+1} of size k, or deduce that (T, k) is a no-instance of FEEDBACK VERTEX SET IN TOURNAMENTS. By iterating at most n times, we obtain the following lemma.

Lemma 4.1. *The existence of an algorithm solving* FEEDBACK VERTEX SET IN TOURNAMENTS COMPRESSION *in time $f(k) \cdot n^c$ implies that* FEEDBACK VERTEX SET IN TOURNAMENTS *can be solved in time $\mathcal{O}(f(k) \cdot n^{c+1})$.*

In all applications of iterative compression one proves a lemma similar to Lemma 4.1. Hence, the bulk of the work goes into solving the compression version of the problem. We now discuss how to solve the compression problem by reducing it to a bounded number of instances of the following disjoint version of the problem: DISJOINT FEEDBACK VERTEX SET IN TOURNAMENTS. In this problem, the input consists of a tournament T together with a directed feedback vertex set W and the objective is either to find a directed feedback vertex set $X \subseteq V(T) \setminus W$ of size at most k, or to conclude that no such set exists.

Let Z be a directed feedback vertex set of size $k+1$ in a tournament T. To decide whether T contains a directed feedback vertex set X of size k (i.e., to solve a FEEDBACK VERTEX SET IN TOURNAMENTS COMPRESSION instance (G, Z, k)), we do the following. We guess the intersection of X with Z, that is, we guess the set $X_Z := X \cap Z$, delete X_Z from T and reduce parameter k by $|X_Z|$. For each guess of X_Z, we set $W := Z \setminus X_Z$ and solve DISJOINT FEEDBACK VERTEX SET IN TOURNAMENTS on the instance $(T - X_Z, W, k -$

$|X_Z|$). If for some guess X_Z we find a directed feedback vertex set X' of $T-X_Z$ of size at most $k-|X_Z|$ that is disjoint from W, then we output $X := X' \cup X_Z$. Otherwise, we conclude that the given instance of the compression problem is a no-instance. The number of all guesses is bounded by $\sum_{i=0}^{k} \binom{k+1}{i}$. So to obtain an FPT algorithm for Feedback Vertex Set in Tournaments Compression, it is sufficient to solve Disjoint Feedback Vertex Set in Tournaments in FPT time. This leads to the following lemma.

Lemma 4.2. *If there exists an algorithm solving* Disjoint Feedback Vertex Set in Tournaments *in time $g(k) \cdot n^{\mathcal{O}(1)}$, then there exists an algorithm solving* Feedback Vertex Set in Tournaments Compression *in time $\sum_{i=0}^{k} \binom{k+1}{i} g(k-i) \cdot n^{\mathcal{O}(1)}$. In particular, if $g(k) = \alpha^k$ for some fixed constant α, then the algorithm runs in time $(\alpha+1)^k \cdot n^{\mathcal{O}(1)}$.*

By Lemmas 4.1 and 4.2, we have that

Solving Feedback Vertex Set in Tournaments boils down to solving the disjoint variant of the problem.

There is nothing very particular about Feedback Vertex Set in Tournaments in Lemma 4.2. In many graph modification problems, for which the corresponding disjoint version can be solved in time $g(k) \cdot n^{\mathcal{O}(1)}$, one can obtain an algorithm for the original problem by proving a lemma analogous to Lemma 4.2. We need only the following two properties: (i) a way to deduce that if an intermediate instance is a no-instance, then the input instance is a no-instance as well; and (ii) a way to enhance a computed solution X_i of an intermediate instance G_i with a new vertex or edge to obtain a slightly larger solution Z_{i+1} of the next intermediate instance G_{i+1}.

4.2.1 *Solving* Disjoint Feedback Vertex Set in Tournaments *in polynomial time*

Our next step is to show that Disjoint Feedback Vertex Set in Tournaments can be solved in polynomial time. As we already discussed, together with Lemmas 4.1 and 4.2, this shows that Feedback Vertex Set in Tournaments can be solved in time $2^k n^{\mathcal{O}(1)}$.

For our proof we need the following simple facts about tournaments, which are left as an exercise.

Lemma 4.3. *Let T be a tournament. Then*

1. *T has a directed cycle if and only if T has a directed triangle;*

2. If T is acyclic then it has unique topological ordering. That is, there exists a unique ordering $\prec$ of the vertices of T such that for every directed edge (u, v), we have $u \prec v$ (that is, u appears before v in the ordering $\prec$).

We now describe the algorithm for the disjoint version of the problem.

Let (T, W, k) be an instance of Disjoint Feedback Vertex Set in Tournaments and let $A = V(T) \setminus W$. Recall that we have that $|W| = k + 1$. Clearly, we can assume that both $T[W]$ and $T[A]$ induce transitive tournaments, since otherwise (T, W, k) is a no-instance. By Lemma 4.3, in order to solve Disjoint Feedback Vertex Set in Tournaments it is sufficient to find a set of at most k vertices from A intersecting all the directed triangles in T. This observation gives us the following simple reduction.

Reduction FVST.1. If T contains a directed triangle x, y, z with exactly one vertex from A, say z, then delete z and reduce the parameter by 1. The new instance is $(T - \{z\}, W, k - 1)$.

Reduction FVST.1 simply says that all directed triangles in T with exactly one vertex in A can be eliminated by picking the corresponding vertex in A. Safeness of Reduction FVST.1 follows from the fact that we are not allowed to select any vertex from W.

Given an instance (T, W, k), we first apply Reduction FVST.1 exhaustively. So from now onwards assume that Reduction FVST.1 is no longer applicable. Since tournaments $T[W]$ and $T[A]$ are acyclic, by Lemma 4.3 we know that they both have unique topological orderings. Let the topological orderings of $T[W]$ and $T[A]$ be denoted by $\sigma = (w_1, \ldots, w_q)$ and ρ, respectively. Suppose X is a desired solution; then $T[W \cup (A \setminus X)]$ is a transitive tournament with the unique ordering such that when we restrict this ordering to W, we obtain σ, and when we restrict it to $A \setminus X$, we get restriction of ρ to $A \setminus X$. Since the ordering of σ is preserved in the ordering of $T[W \cup (A \setminus X)]$, our modified goal now is as follows.

Insert a maximum-sized subset of A into ordering $\sigma = (w_1, \ldots, w_q)$.

Every vertex $v \in A$ has a "natural position" in σ, say $p[v]$, defined as follows. Since Reduction FVST.1 is no longer applicable, for all $v \in A$, we have that $T[W \cup \{v\}]$ is acyclic and the position of v in σ is its position in the unique topological ordering of $T[W \cup \{v\}]$. Thus, there exists an integer $p[v]$ such that for $i < p[v]$, there is an arc from w_i to v and for all $i \geq p[v]$ there is an arc from v to w_i. Thus we get that

$$(v, w_i) \in E(T) \iff i \geq p[v]\,. \tag{4.1}$$

Observe that $p[v]$ is defined for all $v \in A$, it is unique, and $p[v] \in \{1, \ldots, q+1\}$.

We now construct an ordering π of A as follows: u is before v in π if and only if $p[u] < p[v]$ or $p[u] = p[v]$ and u is before v in the ordering ρ. That is, in the ordering π we take iteratively the sets $\{v \in A \ : \ p[v] = i\}$ for $i = 1, 2, \ldots, q+1$ and, within each set, we order the vertices according to ρ, the topological order of $T[A]$.

The main observation now is as follows:

1. In the transitive tournament $T - X$, the topological ordering of $T[A \setminus X]$ needs to be a restriction of π, because $T[W]$ remains in $T - X$ and σ is the topological ordering of $T[W]$.
2. On the other hand, the topological ordering of $T[A \setminus X]$ needs to be a restriction of ρ, the topological ordering of $T[A]$.

Consequently, it suffices to search for the longest common subsequence of π and ρ.

We next prove a lemma which formalizes the intuition suggested above.

Lemma 4.4. *Let $B \subseteq A$. Then $T[W \cup B]$ is acyclic if and only if the vertices of B form a common subsequence of ρ and π.*

Proof. By $\rho|_B$ and $\pi|_B$ we denote restrictions of ρ and π to B, respectively.

For the forward direction, suppose that $T[W \cup B]$ is acyclic. We show that $\rho|_B = \pi|_B$ and hence vertices of B form a common subsequence of ρ and π. Targeting a contradiction, assume that there exist $x, y \in B$ such that x appears before y in $\rho|_B$ and y appears before x in $\pi|_B$. Then $(x, y) \in E(T)$, and $p[y] \leq p[x]$. Moreover, if it was that $p[x] = p[y]$, then the order of x and y in π would be determined by ρ. Thus we conclude that $p[y] < p[x]$. By (4.1), we have that $(y, w_{p[y]}) \in E(T)$ and $(w_{p[y]}, x) \in E(T)$. Because of the directed edges (x, y), $(y, w_{p[y]})$, and $(w_{p[y]}, x)$, we have that $\{x, y, w_{p[y]}\}$ induces a directed triangle in $T[W \cup B]$, a contradiction.

Now we show the reverse direction of the proof. Assume that the vertices of B form a common subsequence of ρ and π. In particular, this means that $\rho|_B = \pi|_B$. To show that $T[W \cup B]$ is acyclic, by Lemma 4.3 it is sufficient to show that $T[W \cup B]$ contains no directed triangles. Since $T[W]$ and $T[A]$ are acyclic and there are no directed triangles with exactly two vertices in W (Reduction FVST.1), it follows that there can only be directed triangles with exactly two vertices in B. Since $\rho|_B = \pi|_B$, for all $x, y \in B$ with $(x, y) \in E(T)$, we have that $p[x] \leq p[y]$. Then by (4.1), there is no $w_i \in W$ with $(y, w_i) \in E(T)$ and $(w_i, x) \in E(T)$. Hence there is no directed triangle in $T[W \cup B]$, and thus it is acyclic. This completes the proof of the lemma. $\square$

We need the following known fact about the longest common subsequence problem, whose proof we leave as Exercise 4.2.

Lemma 4.5. *A longest common subsequence of two sequences with p and q elements can be found in time $\mathcal{O}(pq)$.*

We are ready to describe the algorithm now.

Lemma 4.6. Disjoint Feedback Vertex Set in Tournaments *is solvable in polynomial time.*

Proof. Let (T, W, k) be an instance of the problem. We use Reduction FVST.1 exhaustively. Let R be the set of vertices deleted by Reduction FVST.1, and let (T', W, k') be the reduced instance.

By Lemma 4.4, the optimal way to make T' acyclic by vertex deletions is exactly the same as that to make sequences ρ and π equal by vertex deletions. Thus, to find an optimal vertex deletion set, we can find a longest common subsequence of ρ and π, which can be done in polynomial time by Lemma 4.5. Let B be the vertices of a longest common subsequence of ρ and π and let $X := R \cup (V(T') \setminus B)$. If $|X| > k$; then (T, W, k) is a no-instance. Otherwise, X is the desired directed feedback vertex set of size at most k. $\square$

Lemmas 4.1 and 4.2 combined with Lemma 4.6 give the following result.

Theorem 4.7. Feedback Vertex Set in Tournaments *can be solved in time* $2^k n^{\mathcal{O}(1)}$.

4.3 Feedback Vertex Set

In this section we give an algorithm for the Feedback Vertex Set problem on undirected graphs using the method of iterative compression. Recall that X is a feedback vertex set of an undirected graph G if $G - X$ is a forest. We give only the algorithms for the disjoint version of the problem. As was discussed in Section 4.1.1, the existence of an algorithm with running time $\alpha^k n^{\mathcal{O}(1)}$ for the disjoint variant problem would yield that Feedback Vertex Set is solvable in time $(1+\alpha)^k n^{\mathcal{O}(1)}$.

We start by defining Disjoint Feedback Vertex Set. In this problem, as input we are given an undirected graph G, integer k and a feedback vertex set W in G of size $k+1$. The objective is to find a feedback vertex set $X \subseteq V(G) \setminus W$ of size at most k, or correctly conclude that no such feedback vertex set exists. We first give an algorithm for Disjoint Feedback Vertex Set running in time $4^k n^{\mathcal{O}(1)}$, and then we provide a faster algorithm with running time $\varphi^{2k} n^{\mathcal{O}(1)}$, where $\varphi = \frac{1+\sqrt{5}}{2} < 1.6181$ denotes the golden ratio.

4.3.1 First algorithm for Disjoint Feedback Vertex Set

Let (G, W, k) be an instance of Disjoint Feedback Vertex Set and let $H = G - W$. We first give a few reduction rules that simplify the input instance.

Reduction FVS*.1. Delete all the vertices of degree at most 1 in G.

Reduction FVS*.2. If there exists a vertex v in H such that $G[W \cup \{v\}]$ contains a cycle, then include v in the solution, delete v and decrease the parameter by 1. That is, the new instance is $(G - \{v\}, W, k - 1)$.

Reduction FVS*.3. If there is a vertex $v \in V(H)$ of degree 2 in G such that at least one neighbor of v in G is from $V(H)$, then delete this vertex and make its neighbors adjacent (even if they were adjacent before; the graph could become a multigraph now).

It is easy to see that Reductions FVS*.1, FVS*.2 and FVS*.3 are safe and that they produce an equivalent instance. Furthermore, all of them can be applied in polynomial time. Now we are ready to state the main lemma of this section.

Lemma 4.8. Disjoint Feedback Vertex Set *is solvable in time* $4^k n^{\mathcal{O}(1)}$.

Proof. We give only an algorithm for the decision variant of the problem, i.e., we only verify whether a solution exists or not. It is straightforward to modify the algorithm so that it actually finds a solution, provided there exists one.

We will follow a branching strategy with a nontrivial measure function. Let (G, W, k) be the input instance. If $G[W]$ is not a forest then return that (G, W, k) is a no-instance. So from now onwards we assume that $G[W]$ is indeed a forest. The algorithm first applies Reductions FVS*.1, FVS*.2 and FVS*.3 exhaustively. For clarity we denote the reduced instance (the one on which Reductions FVS*.1, FVS*.2 and FVS*.3 do not apply) by (G, W, k). If $k < 0$, then return that (G, W, k) is a no-instance.

From now onwards we assume that $k \geq 0$. Recall that H is a forest as W is a feedback vertex set. Thus H has a vertex x of degree at most 1. Furthermore, x has at least two neighbors in W, otherwise Reduction FVS*.1 or FVS*.3 would have been applied. Since Reduction FVS*.2 cannot be applied, we have that no two neighbors of x belong to the same connected component of $G[W]$. Now branch by including x in the solution in one branch and excluding it in the other branch. That is, we call the algorithm on instances $(G-\{x\}, W, k-1)$ and $(G, W \cup \{x\}, k)$. If one of these branches returns a solution, then we conclude that (G, W, k) is a yes-instance, otherwise (G, W, k) is a no-instance. The correctness of this algorithm follows from the safeness of our reductions and the fact that the branching is exhaustive.

To estimate the running time of the algorithm, for instance $I = (G, W, k)$, we define its measure

$$\mu(I) = k + \gamma(I),$$

where $\gamma(I)$ is the number of connected components of $G[W]$. Observe that Reductions FVS*.1, FVS*.2, and FVS*.3 do not increase the measure. How does $\mu(I)$ change when we branch? When we include x in the solution, k decreases by 1 and $\gamma(I)$ remains the same, and thus $\mu(I)$ decreases by 1. In the other branch, k remains the same, while x has neighbors in at least two connected components of W. Hence, when we include x in W, $\gamma(I)$ drops by at least 1. Thus, we have a branching vector $(1, 1)$ and the running time of our branching algorithm is $2^{\mu(I)}n^{\mathcal{O}(1)}$. Since at the beginning we have $\mu(I) \le k + |W| \le 2k + 1$, we obtain the desired running time. □

Similarly to Lemmas 4.1 and 4.2 (see also the discussion in Section 4.1.1), one can use iterative compression to show that the $\alpha^k n^{\mathcal{O}(1)}$-time algorithm for Disjoint Feedback Vertex Set can be used to solve Feedback Vertex Set in time $(1+\alpha)^k n^{\mathcal{O}(1)}$. Hence, Lemma 4.8 implies the following theorem.

Theorem 4.9. Feedback Vertex Set *can be solved in time* $5^k n^{\mathcal{O}(1)}$.

*4.3.2 *Faster algorithm for* Disjoint Feedback Vertex Set

Now we show how to modify the algorithm of Theorem 4.9 to obtain "almost" the fastest known deterministic algorithm for Feedback Vertex Set.

We start with a few definitions. Let (G, W, k) be an instance of Disjoint Feedback Vertex Set and $H = G - W$. A vertex $v \in V(H)$ is called a *tent* if its degree in G is 3 and all its neighbors are in W. In other words, v has degree 3 in G and it is an isolated vertex in H. We call a vertex $v \in V(H)$ *nice* if its degree in G is 2 and both its neighbors are in W. The following lemma explains why we are interested in nice vertices and tents.

Lemma 4.10. *Let* (G, W, k) *be an instance of* Disjoint Feedback Vertex Set *such that every vertex from* $V(H)$ *is either nice or a tent. Then* Disjoint Feedback Vertex Set *on* (G, W, k) *can be solved in polynomial time.*

The proof of Lemma 4.10 is based on a polynomial-time algorithm for the "matroid parity" problem for graphic matroids. We defer the proof of this Lemma to Section 12.2.2 and here we use it as a black box to obtain the desired algorithm for Feedback Vertex Set.

For the new algorithm we also need one more reduction rule apart from Reductions FVS*.1, FVS*.2 and FVS*.3.

Reduction FVS*.4. Let $v \in V(H)$ be a vertex of degree 3 in G, with one neighbor u in $V(H)$ and two neighbors in W. In particular, v is a leaf in H.

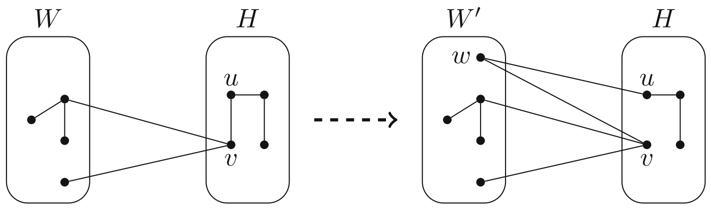


Fig. 4.1: Reduction FVS*.4

Subdivide the edge vu and move the newly introduced vertex, say w, to W. Let the new graph be named G' and $W' = W \cup \{w\}$. The new instance is (G', W', k). See Fig. 4.1.

The safeness of Reduction FVS*.4 follows from the fact that introducing an undeletable degree-2 vertex in the middle of an edge does not change the set of feasible solutions to Disjoint Feedback Vertex Set. Let us also observe that even though Reduction FVS*.4 increases the number of vertices, it also increases the number of tents. Furthermore, as we never add vertices to H, this rule can be applied at most $|V(H)|$ times.

Next we define the new measure which is used to estimate the running time of the new algorithm for Disjoint Feedback Vertex Set. With an instance $I = (G, W, k)$, we associate the measure

$$\mu(I) = k + \gamma(I) - \tau(I).$$

Here, $\gamma(I)$ denotes the number of connected components of $G[W]$ and $\tau(I)$ denotes the number of tents in G.

Next we show that our reduction rules do not increase the measure.

Lemma 4.11. *An application of any of the Reductions FVS*.1, FVS*.2, FVS*.3 and FVS*.4 does not increase* $\mu(I)$.

Proof. The proofs for Reductions FVS*.1, FVS*.2 and FVS*.3 are straightforward. As for Reduction FVS*.4, let $I = (G, W, k)$ be the considered instance. We add vertex w to W thus creating a new connected component and increasing γ by 1. We also create at least one tent at the vertex v (the vertex u may also become a tent), see Fig. 4.1. Thus we increase τ by at least 1. Thus, for the new instance I' we have that $\mu(I') \leq \mu(I)$. □

The important thing we have to keep in mind is that:

For a branching algorithm it is necessary that in all instances whose measure is nonpositive, it should be possible to solve the problem in polynomial time.

So let us now verify that this is the case for Disjoint Feedback Vertex Set. The next lemma implies that if for an instance I we have $\tau(I) \geq k+\gamma(I)$, that is, if the measure of I is nonpositive, then I is a no-instance. In fact, we prove a slightly stronger statement.

Lemma 4.12. *Let* $I = (G, W, k)$ *be an instance of* Disjoint Feedback Vertex Set. *If* $\tau(I) \geq k + \frac{\gamma(I)}{2}$, *then* I *is a no-instance.*

Proof. Suppose, for contradiction, that I is a yes-instance of the problem and let X be a feedback vertex set of size at most k disjoint from W. Thus $G' = G - X$ is a forest. Let $T \subseteq V(G) \setminus W$ be the set of tents in G. Now consider $G'[W \cup (T \setminus X)]$. Clearly, $G'[W \cup (T \setminus X)]$ is also a forest. Now in $G'[W \cup (T \setminus X)]$ we contract each connected component of $G[W]$ into a single vertex and obtain a forest, say $\hat{F}$. Then $\hat{F}$ has at most $\gamma(I)+|T \setminus X|$ vertices, and hence has at most $\gamma(I)+|T \setminus X|-1$ edges. However, the vertices of $T \setminus X$ form an independent set in $\hat{F}$ (as they are tents), and each of them is of degree exactly three in $\hat{F}$: the degree could not drop during the contraction step, since we just contracted some subtrees of the forest $G'[W \cup (T \setminus X)]$. Thus $3|T \setminus X| < \gamma(I) + |T \setminus X|$, and hence $2|T \setminus X| < \gamma(I)$. As $|X| \leq k$, we have that $\tau(I) = |T| < k + \frac{\gamma(I)}{2}$. This concludes the proof. □

Now we are ready to state the main technical lemma of this section.

Lemma 4.13. Disjoint Feedback Vertex Set *can be solved in time* $\varphi^{2k} n^{\mathcal{O}(1)}$, *where* $\varphi < 1.6181$ *is the golden ratio.*

Proof. Again, we give an algorithm only for the decision version of the problem, and turning it into a constructive algorithm is straightforward.

We follow a similar branching strategy, but this time we shall use our adjusted measure $\mu(I)$ that takes into account also tents. Let (G, W, k) be the input instance. If $G[W]$ is not a forest, then return that (G, W, k) is a no-instance. So from now onwards we assume that $G[W]$ is indeed a forest. The algorithm first applies Reductions FVS*.1, FVS*.2, FVS*.3 and FVS*.4 exhaustively. For clarity we denote the reduced instance (the one on which Reductions FVS*.1, FVS*.2, FVS*.3 and FVS*.4 do not apply) by (G, W, k). If $\mu(I) \leq 0$ then return that (G, W, k) is a no-instance. The correctness of this step follows from the fact that if $\mu(I) \leq 0$ then $\tau(I) \geq k+\gamma(I)$, and thus by Lemma 4.12, we have that the given instance is a no-instance.

From now onwards we assume that $\mu(I) > 0$. We now check whether every vertex in $V(G) \setminus W$ is either nice or a tent. If this is the case, we apply Lemma 4.10 and solve the problem in polynomial time. Otherwise, we move to the branching step of the algorithm. The algorithm has only one branching rule which is described below.

Pick a vertex x that is not a tent and has the maximum possible number of neighbors in W.

Now we branch by including x in the solution in one branch, and excluding it in the other branch. That is we call the algorithm on instances $(G - \{x\}, W, k-1)$ and $(G, W \cup \{x\}, k)$. If one of these branches is a yes-instance, then (G, W, k) is a yes-instance. Otherwise (G, W, k) is a no-instance. The correctness of this algorithm follows from the safeness of reductions and the fact that the branching is exhaustive.

To compute the running time of the algorithm, let us see how measure $\mu(I) = k + \gamma(I) - \tau(I)$ changes. By Lemma 4.11, we know that applications of Reductions FVS*.1, FVS*.2, FVS*.3 and FVS*.4 do not increase the measure. Now we will see how $\mu(I)$ changes when we branch. In the branch where we include x in the solution, we have that k decreases by 1, the number of tents does not decrease, and the number of connected components of $G[W]$ remains the same. Thus, $\mu(I)$ decreases by 1.

Now let us consider the other branch, where x is included into W. In this branch, k remains the same and $\tau(I)$ does not decrease. Since Lemma 4.10 is not applicable, we know that there exists a connected component of H that is not just a single tent, and there exists a vertex u in this component whose degree in H is at most one (since H is a forest). Furthermore, since Reductions FVS*.1, FVS*.2, FVS*.3 and FVS*.4 are not applicable, vertex u has at least three neighbors in W. Since in the branching algorithm we chose a vertex x that is not a tent and has the maximum possible number of neighbors in W, we have that x also has at least three neighbors in W. Since Reduction FVS*.2 is not applicable, when we include x to W, graph $G[W \cup \{x\}]$ remains a forest and its number of connected components $\gamma(I)$ drops by at least 2. Thus, $\mu(I)$ drops by at least 2.

The last two paragraphs show that the algorithm has a branching vector of $(1, 2)$. Hence, the algorithm solves Disjoint Feedback Vertex Set in time $\varphi^{\mu(I)} n^{\mathcal{O}(1)}$. Since initially we have that $\mu(I) \leq k + |W| \leq 2k + 1$, we obtain the claimed running time. $\square$

Pipelined with the iterative compression framework (see the discussion in Section 4.1.1), Lemma 4.13 implies the following theorem.

Theorem 4.14. Feedback Vertex Set *can be solved in time*

$$(1 + \varphi^2)^k n^{\mathcal{O}(1)} = 3.6181^k n^{\mathcal{O}(1)}.$$

4.4 Odd Cycle Transversal

In this section we give an algorithm for Odd Cycle Transversal using the method of iterative compression. Recall that X is an odd cycle transversal of G if graph $G - X$ is bipartite. Again, we follow the same generic steps as described in Section 4.1.1.

We start by defining Disjoint Odd Cycle Transversal. In this problem, on the input we are given an undirected graph G, an odd cycle transversal W of size $k+1$ and a positive integer k. The objective is to find an odd cycle transversal $X \subseteq V(G) \setminus W$ of size at most k, or to conclude that no such odd cycle transversal exists. We give an algorithm for Disjoint Odd Cycle Transversal running in time $2^k n^{\mathcal{O}(1)}$.

Our algorithm for Disjoint Odd Cycle Transversal will use the following annotated problem as a subroutine. In the Annotated Bipartite Coloring problem, we are given a *bipartite* graph G, two sets $B_1, B_2 \subseteq V(G)$, and an integer k, and the goal is to find a set X consisting of at most k vertices, such that $G - X$ has a proper 2-coloring $f : V(G) \setminus X \to \{1, 2\}$ (i.e., $f(u) \neq f(v)$ for every edge uv) that agrees with the sets B_1 and B_2, that is, $f(v) = i$ whenever $v \in B_i \setminus X$ and $i = 1, 2$.

An algorithm for Annotated Bipartite Coloring. Let (G, B_1, B_2, k) be an instance of Annotated Bipartite Coloring. We can view vertices of B_1 and B_2 as precolored vertices. We do not assume that this precoloring is proper, that is, a pair of adjacent vertices can be colored with the same color. Moreover, we do not assume that B_1 and B_2 are disjoint, thus some vertices can have both colors. We want to find a set X of size at most k such that in graph $G - X$ there is a *proper* 2-coloring extending precolored vertices. To find such a coloring we proceed as follows. We fix an arbitrary proper 2-coloring f^* of G, $f^* : V(G) \to \{1, 2\}$. Clearly, such a coloring exists as G is a bipartite graph. Let $B_i^* = (f^*)^{-1}(i)$ for $i = 1, 2$. The objective is to find a set S of at most k vertices such that $G - S$ has *another* 2-coloring f such that $B_i \setminus X$ is colored i for $i = 1, 2$. Observe that each vertex of $C := (B_1 \cap B_2^*) \cup (B_2 \cap B_1^*)$ should be either included in X, or have different colors with respect to f^* and f. That is, for every $v \in C \setminus X$, it holds that $f^*(v) \neq f(v)$, i.e., the vertices of $C \setminus X$ must change their colors. Similarly, each vertex of $R := (B_1 \cap B_1^*) \cup (B_2 \cap B_2^*)$ that is not included in X should keep its color. Thus, for every $v \in R$ it holds that $f^*(v) = f(v)$, unless $v \in X$. See the diagram on Fig. 4.2. The following lemma helps us in solving the annotated problem.

Lemma 4.15. *Let G be a bipartite graph and f^* be an arbitrary proper* 2-*coloring of G. Then set X is a solution for* Annotated Bipartite Coloring *if and only if X separates C and R, i.e., no component of $G - X$ contains vertices from both $C \setminus X$ and $R \setminus X$. Furthermore, such a set X of size k (provided it exists) can be found in time $\mathcal{O}(k(n + m))$.*

Proof. We first prove the forward direction of the proof. In a proper 2-coloring of $G - X$, say f, each vertex of $V(G) \setminus X$ either keeps the same color ($f(v) = f^*(v)$) or changes its color ($f(v) \neq f^*(v)$). Moreover, two adjacent vertices have to pick the same decision. Therefore, for every connected component of $G \setminus X$, either all vertices v satisfy $f(v) = f^*(v)$ or all vertices v satisfy $f(v) \neq f^*(v)$. Consequently, no connected component of $G \setminus X$ may contain

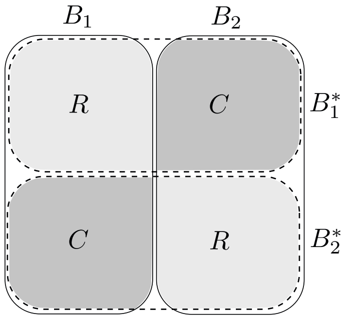


Fig. 4.2: Sets C and R

a vertex of both R (which needs to keep its color) and C (which needs to change its color).

For the backward direction, let X be a set separating R and C. Define the coloring $f : V(G) \setminus X \to \{1, 2\}$ as follows: first set $f = f^*$ and then flip the coloring of those components of $G - X$ that contain at least one vertex of $C \setminus X$. No vertex of R is flipped and thus this is the required 2-coloring.

To find a vertex set of size at most k separating C and R, one can use the classic max-flow min-cut techniques, e.g., by k iterations of the Ford-Fulkerson algorithm (see Theorem 8.2). Thus we obtain the promised running time bound. □

Back to Odd Cycle Transversal. Now we are going to use our algorithm for Annotated Bipartite Coloring to solve Odd Cycle Transversal by applying the iterative compression framework. As usual, to this end we first give an algorithm for the disjoint variant of the problem, Disjoint Odd Cycle Transversal. Recall that here we are given a graph G, an integer k, and a set W of $k+1$ vertices such that $G-W$ is bipartite. The objective is to find a set $X \subseteq V(G) \setminus W$ of at most k vertices such that $G-X$ is bipartite, that is, admits a proper 2-coloring f, or to conclude that no such set exists. Towards this we do as follows. Every vertex $v \in W$ remains in $G - X$ and hence $G[W]$ has to be bipartite, as otherwise we are clearly dealing with a no-instance. The idea is to guess the bipartition of W in $G - X$.

That is, we iterate over all proper 2-colorings $f_W : W \to \{1, 2\}$ of $G[W]$ and we look for a set $X \subseteq V(G) \setminus W$ such that $G - X$ admits a proper 2-coloring that extends f_W. Note that there are at most $2^{|W|} = 2^{k+1}$ choices for the coloring f_W. Let $B_i^W = f_W^{-1}(i)$ for $i = 1, 2$.

Observe that every vertex v in $B_2 := N_G(B_1^W) \cap (V(G) \setminus W)$ needs to be either colored 2 by f, or deleted (included in X). Similarly, every vertex v in $B_1 := N_G(B_2^W) \cap (V(G) \setminus W)$ needs to be either colored 1 by f, or deleted. Consequently, the task of finding the set X and the proper 2-coloring f

that extends f_W reduces to solving an Annotated Bipartite Coloring instance $(G - W, B_1, B_2, k)$.

By Lemma 4.15, this instance can be solved in $\mathcal{O}(k(n+m))$ time and hence the total running time for solving Disjoint Odd Cycle Transversal is $\mathcal{O}(2^k \cdot k(n+m))$. Thus we obtain the following lemma.

Lemma 4.16. Disjoint Odd Cycle Transversal *can be solved in time* $\mathcal{O}(2^k \cdot k(n+m))$.

Now Lemma 4.16, together with the iterative compression approach, yields the following theorem.

Theorem 4.17. Odd Cycle Transversal *is solvable in time* $\mathcal{O}(3^k \cdot kn(n+m))$.

Exercises

4.1 (✎). Prove Lemma 4.3.

4.2. Prove Lemma 4.5.

4.3. Obtain an algorithm for 3-Hitting Set running in time $2.4656^k n^{\mathcal{O}(1)}$ using iterative compression. Generalize this algorithm to obtain an algorithm for d-Hitting Set running in time $((d-1)+0.4656)^k n^{\mathcal{O}(1)}$.

4.4. An undirected graph G is called *perfect* if for every induced subgraph H of G, the size of the largest clique in H is the same as the chromatic number of H. We consider the Odd Cycle Transversal problem, restricted to perfect graphs.

Recall that Exercise 2.34 asked for a kernel with $\mathcal{O}(k)$ vertices, whereas Exercise 3.6 asked for a $3^k n^{\mathcal{O}(1)}$-time branching algorithm for this problem. Here we ask for a $2^k n^{\mathcal{O}(1)}$-time algorithm based on iterative compression.

4.5 (☠). In the Cluster Vertex Deletion problem, we are given a graph G and an integer k, and the task is to find a set X of at most k vertices of G such that $G - X$ is a cluster graph (a disjoint union of cliques). Using iterative compression, obtain an algorithm for Cluster Vertex Deletion running in time $2^k n^{\mathcal{O}(1)}$.

4.6. A graph G is a *split graph* if $V(G)$ can be partitioned into sets C and I, such that C is a clique and I is an independent set. In the Split Vertex Deletion problem, given a graph G and an integer k, the task is to check if one can delete at most k vertices from G to obtain a split graph.

Recall that Exercise 3.20 asked for a $4^k n^{\mathcal{O}(1)}$-time algorithm for this problem, via a reduction to Vertex Cover Above Matching. Here we ask for a $2^k n^{\mathcal{O}(1)}$-time algorithm for this problem using iterative compression.

4.7 (☠). A set $X \subseteq V(G)$ of an undirected graph G is called *an independent feedback vertex set* if $G[X]$ is independent and $G - X$ is acyclic. In the Independent Feedback Vertex Set problem, we are given as input a graph G and a positive integer k, and the objective is to test whether there exists in G an independent feedback vertex set of size at most k. Show that Independent Feedback Vertex Set is in FPT by obtaining an algorithm with running time $5^k n^{\mathcal{O}(1)}$.

4.8. Obtain a polynomial kernel for Disjoint Feedback Vertex Set. Improve it to a kernel with $\mathcal{O}(k)$ vertices.

4.9 (☠). The Edge Bipartization problem is the edge-deletion variant of Odd Cycle Transversal: one is given a graph G and an integer k, and the task is to remove at most k edges from G in order to make G bipartite. Give a $2^k \cdot n^{\mathcal{O}(1)}$-time algorithm for Edge Bipartization using iterative compression.

In the next few exercises we study the Variable Deletion Almost 2-SAT problem: given a Boolean formula φ in CNF, with every clause containing at most two literals, and an integer k, the task is to delete at most k variables from φ to make it satisfiable; a variable is deleted together with all clauses containing it. It is sometimes useful to think of deleting a variable as setting it to both true and false at once, so that it satisfies all the clauses where it appears.

In Theorem 3.12 we have shown an FPT algorithm for this problem, by a reduction to Vertex Cover Above Matching. Furthermore, Exercises 3.21 and 3.22 asked you to prove that Variable Deletion Almost 2-SAT is equivalent to the (more standard) clause-deletion variant called simply Almost 2-SAT, where we are allowed only to delete k clauses. Here, we study relations between Odd Cycle Transversal and Variable Deletion Almost 2-SAT, and develop a different FPT algorithm for Variable Deletion Almost 2-SAT, based on iterative compression.

4.10. Express a given Odd Cycle Transversal instance (G, k) as an instance (φ, k) of Variable Deletion Almost 2-SAT.

4.11. Consider the following annotated variant of Variable Deletion Almost 2-SAT, which we call Annotated Satisfiable Almost 2-SAT: Given a *satisfiable* formula φ, two sets of variables $V^\top$ and $V^\perp$, and an integer k, the task is to check if there exists a set X of at most k variables of φ and a satisfying assignment ψ of $\varphi - X$ such that $\psi(x) = \top$ for every $x \in V^\top \setminus X$ and $\psi(x) = \perp$ for every $x \in V^\perp \setminus X$. Assuming that there exists a $c^k n^{\mathcal{O}(1)}$-time FPT algorithm for Annotated Satisfiable Almost 2-SAT for some constant $c > 1$, show that Variable Deletion Almost 2-SAT admits a $(1+2c)^k n^{\mathcal{O}(1)}$-time algorithm.

4.12. Express a given Annotated Bipartite Coloring instance (G, B_1, B_2, k) as an instance $(\varphi, V^\top, V^\perp, k)$ of Annotated Satisfiable Almost 2-SAT.

4.13. Consider the following operation of *flipping a variable* x in a Variable Deletion Almost 2-SAT or an Annotated Satisfiable Almost 2-SAT instance: we replace every literal x with $\neg x$ and every literal $\neg x$ with x. Furthermore, in the case of Annotated Satisfiable Almost 2-SAT, if $x \in V^\alpha$ for some $\alpha \in \{\top, \perp\}$, then we move x to $V^{\neg\alpha}$.

Using the flipping operation, show that it suffices to consider only Annotated Satisfiable Almost 2-SAT instances where an assignment that assigns false to every variable satisfies the input formula φ. That is, there are no clauses of the form $x \vee y$.

4.14 (☠). Consider the following vertex-deletion variant of the Digraph Pair Cut problem with a sink: given a directed graph G, designated vertices $s, t \in V(G)$, a family of pairs of vertices $\mathcal{F} \subseteq \binom{V(G)}{2}$, and an integer k, check if there exists a set $X \subseteq V(G) \setminus \{s, t\}$ of size at most k such that t is unreachable from s in $G - X$ and every pair in $\mathcal{F}$ either contains a vertex of X or contains a vertex that is unreachable from s in $G - X$. Exercise 8.19 asks you to design a $2^k n^{\mathcal{O}(1)}$-time algorithm for this problem.

1. Deduce from Exercise 4.13 that, when considering the Annotated Satisfiable Almost 2-SAT problem, one can consider only instances where all the clauses are of the form $x \Rightarrow y$ or $\neg x \vee \neg y$. (Remember unary clauses.)
2. Use this insight to design a $2^k n^{\mathcal{O}(1)}$-time algorithm for Annotated Satisfiable Almost 2-SAT, using the aforementioned algorithm of Exercise 8.19 as a subroutine.
3. Conclude that such a result, together with a solution to Exercise 4.11, gives a $5^k n^{\mathcal{O}(1)}$-time algorithm for Variable Deletion Almost 2-SAT.

Hints

4.2 This problem is a standard example for dynamic programming. Let w_p and w_q be two sequences, of length p and q, respectively. For every $0 \leq i \leq p$ and $0 \leq j \leq q$, we define $T[i, j]$ to be the length of the longest common subsequence of substrings $w_p[1, \dots, i]$ and $w_q[1, \dots, j]$. Then, $T[i, j]$ equals the maximum of: $T[i-1, j]$, $T[i, j-1]$ and, if $w_p[i] = w_q[j]$, $1 + T[i-1, j-1]$.

4.3 In the case of 3-Hitting Set, observe that the disjoint version of the problem is essentially a Vertex Cover instance, as every set of size 3 contains at least one undeletable element. Then, use the algorithm of Theorem 3.2. In the general case, generalize this observation: the disjoint version of the problem is also a d-Hitting Set instance, but every set lost at least one element.

4.4 Reduce the disjoint version of the problem to a minimum vertex cover problem on an auxiliary bipartite graph. Some insight from Exercise 4.3 may help you.

4.5 The exercise boils down to a task of showing that the disjoint version of the problem is polynomial-time solvable.

First, observe that a graph is a cluster graph if and only if it does not have an induced path on three vertices. In the disjoint version of the problem, given (G, S, k) such that $|S| = k + 1$ and $G - S$ is a cluster graph, we are looking for a solution of size at most k such that it is disjoint from S.

- We can assume that the subgraph induced on S is a cluster graph. Why?
- What can you do if there is a vertex in $V(G) \setminus S$ that is adjacent to two of the clusters of $G[S]$, the subgraph induced on S? Or a vertex in $V(G) \setminus S$ that is adjacent to some, but not all vertices of one of the clusters of $G[S]$?
- Now we can assume that every vertex in $V(G) \setminus S$ is either completely nonadjacent to S, or completely adjacent to exactly one cluster of $G[S]$. Argue that the resulting problem (of finding a subset of vertices to be deleted from $V(G) \setminus S$ to make the resulting graph a cluster graph) can be solved by using an algorithm for finding a maximum matching of minimum cost in a bipartite graph.

4.6 Again, the task reduces to showing that the disjoint version of the problem is polynomial-time solvable.

The main observation is that a split graph on n vertices has at most n^2 partitions of the vertex set into the clique and independent set part: for a fixed one partition, at most one vertex can be moved from the clique side to the independent one, and at most one vertex can be moved in the opposite direction. (In fact one can show that it has at most $\mathcal{O}(n)$ split partitions.) Hence, you can afford guessing the "correct" partition of both the undeletable and the deletable part of the instance at hand.

4.7 In short, mimic the $3.619^k n^{\mathcal{O}(1)}$-time algorithm for Feedback Vertex Set.

That is, design a $4^k n^{\mathcal{O}(1)}$-time algorithm for the disjoint version of the problem in the following manner. Assume we are given an instance (G, W, k) of the following disjoint version of Independent Feedback Vertex Set: $G - W$ is a forest, $|W| = k + 1$, and we seek an independent feedback vertex set X of G that is of size at most k and is disjoint from W. Note that we do not insist that $G[W]$ be independent.

A *tent* is a vertex $v \notin W$, of degree 2, with both neighbors in W. For an instance $I = (G, W, k)$, we let $\tau(I)$ be the number of tents, $\gamma(I)$ be the number of connected components of $G[W]$, and $\mu(I) = k + \gamma(I) - \tau(I)$ be the potential of I. Proceed with the following steps.

1. Adapt the reduction rules that deal with vertices of degree 1 and with vertices $v \notin W$ that have more than one incident edge connecting them to the same connected component of $G[W]$.

2. Design a reduction that deals with vertices $v \notin W$ of degree 2 that have exactly one neighbor in W. Note that you cannot simply move them to W, due to the condition of solutions being independent. Instead, proceed similarly as we did in the Feedback Vertex Set algorithm for vertices of degree 3, with exactly two neighbors in W.
3. Finally, prove that the potential is positive on a yes-instance by adapting the proof of Lemma 4.12.

4.9 To perform the iterative compression step, observe that picking one endpoint of every edge of a solution to Edge Bipartization yields an odd cycle transversal of the graph. Use this observation to reduce the problem to designing a compression routine for the following task: you are given a graph G with an odd cycle transversal W of size $k+1$, and the task is to find a solution to Edge Bipartization of size at most k, or to conclude that no such solution exists. Branch into 2^{k+1} subproblems, guessing the right bipartition of W in the bipartite graph after removing the edges of the solution. Then reduce verifying existence of a solution in a branch to an auxiliary polynomial-time solvable cut problem, using similar ideas as with sets C and R in the algorithm for Odd Cycle Transversal.

4.8 Apply the reduction rules given for the disjoint version of the problem. Argue that if there are too many vertices (like $k^{\mathcal{O}(1)}$ or $\mathcal{O}(k)$) of degree at most 2 in the deletable part, then the given instance is a no-instance. Else, use a standard tree-based argument to show that the instance is small.

4.10 Treat every vertex as a variable. For every edge $xy \in E(G)$, introduce clauses $x \vee y$ and $\neg x \vee \neg y$. This concludes the construction of the formula φ. To show correctness, observe that the clauses for an edge xy force x and y to attain different values in a satisfying assignment.

4.11 Using iterative compression, the task boils down to designing a $(2c)^k n^{\mathcal{O}(1)}$-time algorithm for the disjoint version of Variable Deletion Almost 2-SAT. Thus, we have an instance (φ, W, k), where W is a set of $k+1$ variables from φ such that $\varphi - W$ is satisfiable, and we seek a solution X to Variable Deletion Almost 2-SAT on (φ, k) that is disjoint from W.

At the cost of 2^{k+1} subcases, we guess the values of variables of W in some satisfying assignment of $\varphi - X$. Now, we can do the following cleaning: discard all clauses satisfied by the guessed assignment, and discard the current branch if there exists a clause with all literals evaluated to false by the guessed assignment. Observe that every remaining clause $C = (\ell_1 \vee \ell_2)$ that contains a literal with a variable from W (say, ℓ_1) satisfies the following: ℓ_1 is evaluated to false by the guessed assignment, and the second literal ℓ_2 contains a variable not in W. For every such clause C, we put the variable of ℓ_2 into $V^\top$ if ℓ_2 is a positive literal, and otherwise we put the variable of ℓ_2 into $V^\bot$. Argue that it remains to solve an Annotated Satisfiable Almost 2-SAT instance $(\varphi - W, V^\top, V^\bot, k)$.

4.12 Proceed as in Exercise 4.10.

4.14 The first and the last point are straightforward. Here we elaborate on the second point.

We express the task of solving the input instance $(\varphi, V^\top, V^\bot, k)$ of Annotated Satisfiable Almost 2-SAT as a vertex-deletion Digraph Pair Cut with a sink instance $(G, s, t, \mathcal{F}, k)$ as follows.

1. Start with vertices s and t.
2. For every variable x, introduce a single vertex x.
3. For every clause $x \Rightarrow y$ (i.e., $\neg x \vee y$), introduce an edge (x, y).
4. For every clause $\neg x \vee \neg y$, introduce a pair $\{x, y\}$ to the family $\mathcal{F}$.
5. For every vertex $y \in V^\top$, we introduce an edge (s, y).
6. For every vertex $y \in V^\bot$, we introduce an edge (y, t).

Argue that solutions to the aforementioned node-deletion Digraph Pair Cut instance $(G, s, \mathcal{F}, k)$ are in one-to-one correspondence with the solutions to the original instance of Annotated Satisfiable Almost 2-SAT. The intuition is that the nodes x reachable from s after the solution is removed correspond to the variables that are set to true in a satisfying assignment.

Bibliographic notes

The idea of iterative compression was first introduced by Reed, Smith, and Vetta [397] to prove that Odd Cycle Transversal is fixed-parameter tractable. The technique has been used in several FPT algorithms for various problems, including Directed Feedback Vertex Set [85] and Almost 2-SAT [394].

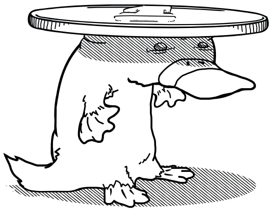
The algorithm for Feedback Vertex Set in Tournaments of Theorem 4.7 is due to Dom, Guo, Hüffner, Niedermeier, and Truß [140]. For the Feedback Vertex Set problem, the parametric dependency in the running time has been systematically improved over the years [393, 241, 127, 77, 70], resulting in the current best deterministic algorithm running in time $3.592^k n^{\mathcal{O}(1)}$ [301]. If we allow randomization, then the problem admits an algorithm with running time $3^k \cdot n^{\mathcal{O}(1)}$ [118] — we will actually see this algorithm in Section 11.2.1). Our simpler $5^k n^{\mathcal{O}(1)}$-time algorithm for Feedback Vertex Set follows the work of Chen, Fomin, Liu, Lu, and Villanger [77]. The description of the $3.619^k n^{\mathcal{O}(1)}$ algorithm follows Kociumaka and Pilipczuk [301]. The observation that Disjoint Feedback Vertex Set instance becomes polynomial-time solvable if every deletable vertex is nice or a tent is due to Cao, Chen, and Liu [70].

The algorithm for Cluster Vertex Deletion from Exercise 4.5 originates in the work of Hüffner, Komusiewicz, Moser, and Niedermeier [271]. A $2^k \cdot n^{\mathcal{O}(1)}$ algorithm for Edge Bipartization was first given by Guo, Gramm, Hüffner, Niedermeier, and Wernicke [241]; their algorithm is actually quite different from the one sketched in the hint to Exercise 4.9. The parameterized complexity of Almost 2-SAT was open for some time until Razgon and O'Sullivan gave a $15^k n^{\mathcal{O}(1)}$ algorithm. The current fastest algorithm for this problem runs in time $2.315^k n^{\mathcal{O}(1)}$ [328]. The algorithm for Almost 2-SAT from Exercise 4.14 originates in the work of Kratsch and Wahlström [310]; they gave a polynomial kernel for the problem, and the algorithm follows from their ideas implicitly.

In this book we will again use iterative compression to solve Planar Vertex Deletion in Chapter 7, and Directed Feedback Vertex Set in Chapter 8.

Chapter 5
Randomized methods in parameterized algorithms

In this chapter, we study how the powerful paradigm of randomization can help in designing FPT *algorithms. We introduce the general techniques of color coding, divide and color, and chromatic coding, and show the use of these techniques on the problems* LONGEST PATH *and* d-CLUSTERING. *We discuss also how these randomized algorithms can be derandomized using standard techniques.*



We give a few commonly used approaches to design randomized FPT algorithms.

In Section 5.1, we start with a simple example of an algorithm for FEEDBACK VERTEX SET. Here, we essentially observe that, after applying a few easy reduction rules, a large fraction of the edges of the graph needs to be adjacent to the solution vertices — and, consequently, taking into the solution a randomly chosen endpoint of a randomly chosen edge leads to a good success probability.

Then, in Section 5.2, we move to the very successful technique of *color coding*, where one randomly colors a universe (e.g., the vertex set of the input graph) with a carefully chosen number of colors and argue that, with sufficient probability, a solution we are looking for is somehow "properly colored". The problem-dependant notion of "properly colored" helps us resolve the problem if we only look for properly colored solutions. For resolving the colored version of the problem, we use basic tools learned in the previous chapters, such as bounded search trees. In this section we start with the classic example of a color coding algorithm for LONGEST PATH. Then, we show how a random partition (i.e., a random coloring with two colors) can be used to highlight a solution, using an example of the SUBGRAPH ISOMORPHISM problem in bounded degree graphs. Moreover, we present the so-called *divide and color* technique for LONGEST PATH, where a recursive random partitioning scheme allows us to obtain a better running time than the simple color coding approach. We conclude this section with a more advanced example of a *chromatic coding* approach for d-CLUSTERING.

DOI 10.1007/978-3-319-21275-3_5

Finally, in Section 5.6, we briefly discuss the methods of derandomizing algorithms based on color coding. We recall the necessary results on constructing pseudorandom objects such as splitters, perfect hash families, universal sets and small k-wise independent sample spaces, and show how to replace the random coloring step of (variants of) the color coding approach by iterating over a carefully chosen pseudorandom object.

Let us briefly recall the principles of randomization in algorithms. We assume that the algorithm is given access, apart from the input, to a stream of *random bits*. If the algorithm reads at most r random bits on the given input, then the probability space is the set of all 2^r possible strings of random bits read by the algorithm, with uniform probability distribution (i.e., each bit is chosen independently and uniformly at random). Whenever we say that "an algorithm does X with probability at least/at most p", we mean the probability measured in this probability space.

Consider an algorithm for a decision problem which given a no-instance always returns "no," and given a yes-instance returns "yes" with probability $p \in (0,1)$. Such an algorithm is called a one-sided error Monte Carlo algorithm with false negatives.

Sometimes we would like to improve the success probability p, especially in our applications where we often have bounds like $p = 1/2^{\mathcal{O}(k)}$, or, more generally, $p = 1/f(k)$ for some computable function f. It is not difficult to see that we can improve the success probability at the price of worse running time. Namely, we get a new algorithm by repeating the original algorithm t times and returning "no" only if the original algorithm returned "no" in each of the t repetitions. Clearly, given a no-instance the new algorithm still returns "no." However, given a yes-instance, it returns "no" only if all t repetitions returned an incorrect "no" answer, which has probability at most

$$(1-p)^t \leq (e^{-p})^t = 1/e^{pt},$$

where we used the well-known inequality $1 + x \leq e^x$. It follows that the new success probability is at least $1 - 1/e^{pt}$. Note that in particular it suffices to put $t = \lceil \frac{1}{p} \rceil$ to get constant success probability.[1]

If a one-sided error Monte Carlo algorithm has success probability at least p, then repeating it independently $\lceil \frac{1}{p} \rceil$ times gives constant success probability. In particular, if $p = 1/f(k)$ for some computable function f, then we get an FPT one-sided error Monte Carlo algorithm with additional $f(k)$ overhead in the running time bound.

[1] In this chapter, by "constant success probability" or "constant error probability" we mean that the success probability of an algorithm is lower bounded by a positive universal constant (or, equivalently, the error probability is upper bounded by some universal constant strictly smaller than 1).

In the area of polynomial-time algorithms, quite often we develop an algorithm that does something useful (e.g., solves the problem) with probability at least p, where $1/p$ is bounded polynomially in the input size. If this is the case, then we can repeat the algorithm a polynomial number of times, obtaining a constant error probability. In the FPT world, the same principle applies; however, now the threshold of "useful probability" becomes $1/(f(k)n^{\mathcal{O}(1)})$. That is, if we develop an algorithm that runs in FPT time and solves the problem with probability at least $1/(f(k)n^{\mathcal{O}(1)})$, then repeating the algorithm $f(k)n^{\mathcal{O}(1)}$ times gives us constant error probability, while still maintaining an FPT running time bound. This is the goal of most of the algorithms in this chapter.

5.1 A simple randomized algorithm for Feedback Vertex Set

In this section, we design a simple randomized algorithm for Feedback Vertex Set. As discussed in Section 3.3, when tackling with Feedback Vertex Set, it is more convenient to work with multigraphs, not only simple graphs. Recall that both a double edge and a loop are cycles, and we use the convention that a loop at a vertex v contributes *2* to the degree of v.

The following crucial lemma observes that, once we apply to the input instance basic reduction rules that deal with low-degree vertices, many edges of the graph are incident to the solution vertices.

Lemma 5.1. *Let G be a multigraph on n vertices, with minimum degree at least* 3*. Then, for every feedback vertex set X of G, more than half of the edges of G have at least one endpoint in X.*

Proof. Let $H = G - X$. Since every edge in $E(G) \setminus E(H)$ is incident to a vertex in X, the claim of the lemma is equivalent to $|E(G) \setminus E(H)| > |E(H)|$. However, since H is a forest, $|V(H)| > |E(H)|$ and it suffices to show that $|E(G) \setminus E(H)| > |V(H)|$.

Let J denote the set of edges with one endpoint in X and the other in $V(H)$. Let $V_{\leq 1}$, V_2 and $V_{\geq 3}$ denote the set of vertices in $V(H)$ such that they have degree at most 1, exactly 2, and at least 3 in H, respectively. (Note that we use here the degree of a vertex in the forest H, not in the input graph G.) Since G has minimum degree at least 3, every vertex in $V_{\leq 1}$ contributes at least two distinct edges to J. Similarly, each vertex in V_2 contributes at least one edge to J. As H is a forest, we have also that $|V_{\geq 3}| < |V_{\leq 1}|$. Putting all these bounds together, we obtain:

$$\begin{aligned}|E(G) \setminus E(H)| &\geq |J| \\ &\geq 2|V_{\leq 1}| + |V_2| > |V_{\leq 1}| + |V_2| + |V_{\geq 3}| \\ &= |V(H)|.\end{aligned}$$

This concludes the proof of the lemma. □

Recall that in Section 3.3, we have developed the simple reduction rules FVS.1–FVS.5 that reduce all vertices of degree at most 2 in the input graph. Lemma 5.1 says that, once such a reduction has been performed, a majority of the edges have at least one endpoint in a solution. Hence, if we pick an edge uniformly at random, and then independently pick its random endpoint, with probability at least $1/4$ we would pick a vertex from the solution. By iterating this process, we obtain an algorithm that solves the input Feedback Vertex Set instance in polynomial time, with probability at least 4^{-k}.

Theorem 5.2. *There exists a polynomial-time randomized algorithm that, given a* Feedback Vertex Set *instance* (G, k)*, either reports a failure or finds a feedback vertex set in* G *of size at most* k*. Moreover, if the algorithm is given a yes-instance, it returns a solution with probability at least* 4^{-k}*.*

Proof. We describe the algorithm as a recursive procedure that, given a graph G and an integer k, aims at a feedback vertex set of G of size at most k.

We first apply exhaustively Reductions FVS.1–FVS.5 to the Feedback Vertex Set instance (G, k). If the reductions conclude that we are dealing with a no-instance, then we report a failure. Otherwise, let (G', k') be the reduced instance. Note that $0 \leq k' \leq k$ and G' has minimum degree at least 3. Let X_0 be the set of vertices deleted due to Reduction FVS.1. Note that the vertices of X_0 have been qualified as mandatory vertices for a feedback vertex set in G, that is, there exists a feedback vertex set in G of minimum possible size that contains X_0, and for any feedback vertex set X' of G', $X' \cup X_0$ is a feedback vertex set of G. Moreover, $|X_0| = k - k'$.

If G' is an empty graph, then we return X_0; note that in this case $|X_0| \leq k$ as $k' \geq 0$, and X_0 is a feedback vertex set of G. Otherwise, we pick an edge e of G' uniformly at random (i.e., each edge is chosen with probability $1/|E(G')|$), and choose one endpoint of e independently and uniformly at random. Let v be the chosen endpoint. We recurse on $(G' - v, k' - 1)$. If the recursive step returns a failure, then we return a failure as well. If the recursive step returns a feedback vertex set X', then we return $X := X' \cup \{v\} \cup X_0$.

Note that in the second case the set X' is a feedback vertex set of $G' - v$ of size at most $k' - 1$. First, observe that the size bound on X' implies $|X| \leq k$. Second, we infer that $X' \cup \{v\}$ is a feedback vertex set of G' and, consequently, X is a feedback vertex set of G. Hence, the algorithm always reports a failure or returns a feedback vertex set of G of size at most k. It remains to argue about the probability bound; we prove it by induction on k.

Assume that there exists a feedback vertex set X of G of size at most k. By the analysis of the reduction rules of Section 3.3, Reduction FVS.5 is not triggered, the instance (G', k') is computed and there exists a feedback vertex set X' of G' of size at most k'. If G' is empty, then the algorithm returns X_0 deterministically. Otherwise, since G' has minimum degree at least 3, Lemma 5.1 implies that, with probability larger than 1/2, the edge e has at least one endpoint in X'. Consequently, with probability larger than 1/4, the chosen vertex v belongs to X', and $X' \setminus \{v\}$ is a feedback vertex set of size at most $k' - 1$ in the graph $G' - v$. By the inductive hypothesis, the recursive call finds a feedback vertex set of $G' - v$ of size at most $k' - 1$ (not necessarily the set $X' \setminus \{v\}$) with probability at least $4^{-(k'-1)}$. Hence, a feedback vertex set of G of size at most k is found with probability at least $\frac{1}{4} \cdot 4^{-(k'-1)} = 4^{-k'} \geq 4^{-k}$. This concludes the inductive step, and finishes the proof of the theorem. $\square$

As discussed at the beginning of this section, we can repeat the algorithm of Theorem 5.2 independently 4^k times to obtain a constant error probability.

Corollary 5.3. *There exists a randomized algorithm that, given a* FEEDBACK VERTEX SET *instance (G, k), in time $4^k n^{\mathcal{O}(1)}$ either reports a failure or finds a feedback vertex set in G of size at most k. Moreover, if the algorithm is given a yes-instance, it returns a solution with a constant probability.*

5.2 Color coding

The technique of color coding was introduced by Alon, Yuster and Zwick to handle the problem of detecting a small subgraph in a large input graph. More formally, given a k-vertex "pattern" graph H and an n-vertex input graph G, the goal is to find a subgraph of G isomorphic to H. A brute-force approach solves this problem in time roughly $\mathcal{O}(n^k)$; the color coding technique approach allows us to obtain an FPT running time bound of $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ in the case when H is a forest or, more generally, when H is of constant treewidth (for more on treewidth, we refer you to Chapter 7). We remark here that, as we discuss in Chapter 13, such an improvement is most likely not possible in general, as the case of H being a k-vertex clique is conjectured to be hard.

The idea behind the color coding technique is to randomly color the entire graph with a set of colors with the number of colors chosen in a way that, if the smaller graph does exist in this graph as a subgraph, then with high probability it will be colored in a way that we can find it *efficiently*. In what follows we mostly focus on a simplest, but very instructive case of H being a k-vertex path.

We remark that most algorithms based on the color coding technique can be derandomized using splitters and similar pseudorandom objects. We discuss these methods in Section 5.6.

5.2.1 A color coding algorithm for Longest Path

In the Longest Path problem, we are given a graph G and a positive integer k as input, and the objective is to test whether there exists a simple path on k vertices in the graph G. Note that this corresponds to the aforementioned "pattern" graph search problem with H being a path on k vertices (henceforth called a *k-path* for brevity).

Observe that finding a *walk* on k vertices in a directed graph is a simple task; the hardness of the Longest Path problem lies in the requirement that we look for a *simple path.* A direct approach involves keeping track of the vertices already visited, requiring an $\binom{n}{k}$ factor in the running time bound. The color coding technique is exactly the trick to avoid such a dependency.

Color the vertices uniformly at random from $\{1, \ldots, k\}$, and find a path on k vertices, if it exists, whose all colors are pairwise distinct.

The essence of the color coding technique is the observation that, if we color some universe with k colors uniformly at random, then a given k-element subset is colored with distinct colors with sufficient probability.

Lemma 5.4. *Let U be a set of size n, and let $X \subseteq U$ be a subset of size k. Let $\chi : U \to [k]$ be a coloring of the elements of U, chosen uniformly at random (i.e., each element of U is colored with one of k colors uniformly and independently at random). Then the probability that the elements of X are colored with pairwise distinct colors is at least e^{-k}.*

Proof. There are k^n possible colorings χ, and $k!k^{n-k}$ of them are injective on X. The lemma follows from the well-known inequality $k! > (k/e)^k$. $\square$

Hence, the color coding step reduces the case of finding a k-path in a graph to finding a *colorful* k-path in a vertex-colored graph. (In what follows, a path is called *colorful* if all vertices of the path are colored with pairwise distinct colors.) Observe that this step replaces the $\binom{n}{k}$ factor, needed to keep track of the used vertices in a brute-force approach, with a much better 2^k factor, needed to keep track of used *colors.* As the next lemma shows, in the case of finding a colorful path, the algorithm is relatively simple.

Lemma 5.5. *Let G be a directed or an undirected graph, and let $\chi : V(G) \to [k]$ be a coloring of its vertices with k colors. There exists a deterministic algorithm that checks in time $2^k n^{\mathcal{O}(1)}$ whether G contains a colorful path on k vertices and, if this is the case, returns one such path.*

Proof. Let $V_1, \ldots, V_k$ be a partitioning of $V(G)$ such that all vertices in V_i are colored i. We apply dynamic programming: for a *nonempty* subset S of $\{1, \ldots, k\}$ and a vertex $u \in \bigcup_{i \in S} V_i$, we define the Boolean value PATH(S, u)

to be equal to true if there is a colorful path with vertices colored with all colors from S and with an endpoint in u. For $|S| = 1$, note that PATH(S, u) is true for any $u \in V(G)$ if and only if $S = \{\chi(u)\}$. For $|S| > 1$, the following recurrence holds:

$$\text{PATH}(S, u) = \begin{cases} \bigvee\{\text{PATH}(S \setminus \{\chi(u)\}, v) \ : \ uv \in E(G)\} & \text{if } \chi(u) \in S \\ \text{False} & \text{otherwise.} \end{cases}$$

Indeed, if there is a colorful path ending at u using all colors from S, then there has to be colorful path ending at a neighbor v of u and using all colors from $S \setminus \{\chi(u)\}$.

Clearly, all values of PATH can be computed in time $2^k n^{\mathcal{O}(1)}$ by applying the above recurrence, and, moreover, there exists a colorful k-path in G if and only if PATH$([k], v)$ is true for some vertex $v \in V(G)$. Furthermore, a colorful path can be retrieved using the standard technique of backlinks in dynamic programming. □

We now combine Lemmas 5.4 and 5.5 to obtain the main result of this section.

Theorem 5.6. *There exists a randomized algorithm that, given a* Longest Path *instance (G, k), in time $(2e)^k n^{\mathcal{O}(1)}$ either reports a failure or finds a path on k vertices in G. Moreover, if the algorithm is given a yes-instance, it returns a solution with a constant probability.*

Proof. We show an algorithm that runs in time $2^k n^{\mathcal{O}(1)}$, and, given a yes-instance, returns a solution with probability at least e^{-k}. Clearly, by repeating the algorithm independently e^k times, we obtain the running time bound and success probability guarantee promised by the theorem statement.

Given an input instance (G, k), we uniformly at random color the vertices of $V(G)$ with colors $[k]$. That is, every vertex is colored independently with one color from the set $[k]$ with uniform probability. Denote the obtained coloring by $\chi : V(G) \to [k]$. We run the algorithm of Lemma 5.5 on the graph G with coloring χ. If it returns a colorful path, then we return this path as a simple path in G. Otherwise, we report failure.

Clearly, any path returned by the algorithm is a k-path in G. It remains to bound the probability of finding a path in the case (G, k) is a yes-instance.

To this end, suppose G has a path P on k vertices. By Lemma 5.4, P becomes a colorful path in the coloring χ with probability at least e^{-k}. If this is the case, the algorithm of Lemma 5.5 finds a colorful path (not necessarily P itself), and the algorithm returns a k-path in G. This concludes the proof of the theorem. □

5.3 Random separation

In this section we give a variant of the color coding technique that is particularly useful in designing parameterized algorithms on graphs of bounded degree. The technique, usually called *random separation* in the literature, boils down to a simple but fruitful observation that in some cases if we randomly partition the vertex or edge set of a graph into two sets, the solution we are looking for gets highlighted with good probability.

We illustrate the method on the Subgraph Isomorphism problem, where we are given an n-vertex graph G and a k-vertex graph H, and the objective is to test whether there exists a subgraph $\widehat{H}$ of G such that H is isomorphic to $\widehat{H}$. Observe that Longest Path is a special case of Subgraph Isomorphism where H is a path on k vertices. Exercise 5.2 asks you to generalize the color coding approach for Longest Path to Tree Subgraph Isomorphism, where H is restricted to being a tree, whereas Exercise 7.48 takes this generalization further, and assumes H has bounded treewidth (the notion of treewidth, which measures resemblance of a graph to a tree, is the topic of Chapter 7). Also, observe that the Clique problem is a special case of Subgraph Isomorphism, where H is a clique on k vertices. It is believed that Clique is not FPT (see Chapter 13), and, consequently, we do not expect that the general Subgraph Isomorphism problem is FPT when parameterized by k.

In this section we restrict ourselves to graphs of bounded degree, and show that if the degree of G is bounded by d, then Subgraph Isomorphism can be solved in time $f(d,k)n^{\mathcal{O}(1)}$ for some computable function f. In other words, Subgraph Isomorphism is FPT when parameterized by both k and d.

The idea of random separation is to color edges (vertices) randomly such that the edges (vertices) of the solution are colored with one color and the edges (vertices) that are adjacent to edges (vertices) of the solution subgraph get colored with a different color. That is, we separate the solution subgraph from its neighboring edges and vertices using a random coloring.

In Subgraph Isomorphism, if we perform a successful coloring step, then every connected component of the pattern graph H corresponds to one connected component of the subgraph induced by the first color in the colored graph G.

Let us now focus on the Subgraph Isomorphism problem. We first do a sanity test and check whether the maximum degree of H is at most d; if this is not the case, then we immediately report that the given instance is a no-instance. Let us color independently every edge of G in one of two colors, say red and blue (denoted by R and B), with probability $\frac{1}{2}$ each. Denote the obtained random coloring by $\chi : E(G) \to \{R, B\}$. Suppose that (G, H)

is a yes-instance. That is, there exists a subgraph $\widehat{H}$ in G such that H is isomorphic to $\widehat{H}$.

Let Γ denote the set of edges that are incident to some vertex of $V(\widehat{H})$, but do not belong to $E(\widehat{H})$. We say that a coloring χ is *successful* if both the following two conditions hold:

1. every edge of $E(\widehat{H})$ is colored red, that is, $E(\widehat{H}) \subseteq \chi^{-1}(R)$; and
2. every edge of Γ is colored blue, that is, $\Gamma \subseteq \chi^{-1}(B)$.

Observe that $E(\widehat{H})$ and Γ are disjoint. Thus, the two aforementioned conditions are independent. Furthermore, since every edge of $E(\widehat{H}) \cup \Gamma$ is incident to a vertex of $V(\widehat{H})$, $|V(\widehat{H})| = |V(H)| = k$, and the maximum degree of G is at most d, we obtain

$$|E(\widehat{H})| + |\Gamma| \leq d|V(\widehat{H})| \leq dk.$$

Consequently, the probability that χ is successful is at least

$$\frac{1}{2^{|E(\widehat{H})|+|\Gamma|}} \geq \frac{1}{2^{dk}}.$$

Let G_R be the subgraph of G containing only those edges that have been colored red, that is, $G_R = (V(G), \chi^{-1}(R))$. The core observation now is as follows: if χ is successful, then $\widehat{H}$ is a subgraph of G_R as well and, moreover, $\widehat{H}$ consists of a number of connected components of G_R.

For simplicity assume first that H is connected. Thus, if χ is successful, then $\widehat{H}$ is a connected component of G_R. Consequently, we can go over all connected components C of G_R with exactly k vertices and test if C is isomorphic to H. Such a single test can be done by brute force in time $k!k^{\mathcal{O}(1)}$, or we can use a more involved algorithm for Graph Isomorphism on bounded degree graphs that runs in time $k^{\mathcal{O}(d \log d)}$ on k-vertex graphs of maximum degree d.

Let us now consider the general case when H is not necessarily connected. Let $H_1, H_2, \ldots, H_p$ be the connected components of H, and let $C_1, C_2, \ldots, C_r$ be the connected components of G_R. We construct an auxiliary bipartite graph $B(H, G_R)$, where every vertex corresponds to a connected component H_i or a connected component C_j, and H_i is adjacent to C_j if they have at most k vertices each and are isomorphic. Observe that the graph $B(H, G_R)$ can be constructed in time $k!n^{\mathcal{O}(1)}$ or in time $k^{\mathcal{O}(d \log d)}n^{\mathcal{O}(1)}$, depending on the Graph Isomorphism algorithm we use, and the subgraph $\widehat{H}$ isomorphic to H corresponds to a matching in $B(H, G_R)$ that saturates $\{H_1, H_2, \ldots, H_p\}$. (Note that every component of $B(H, G_R)$ is a complete bipartite graph, so the matching can be found using a trivial algorithm.) Consequently, we can check if there is a subgraph of G_R isomorphic to H that consists of a number of connected components of G_R in time $k!n^{\mathcal{O}(1)}$ or in time $k^{\mathcal{O}(d \log d)}n^{\mathcal{O}(1)}$. This completes the description the algorithm.

We have proved that the probability that χ is successful is at least 2^{-dk}. Hence, to obtain a Monte Carlo algorithm with false negatives we repeat the above procedure 2^{dk} times, and obtain the following result:

Theorem 5.7. *There exist Monte Carlo algorithms with false negatives that solve the input* Subgraph Isomorphism *instance* (G, H) *in time* $2^{dk}k!n^{\mathcal{O}(1)}$ *and in time* $2^{dk}k^{\mathcal{O}(d \log d)}n^{\mathcal{O}(1)}$. *Here,* $|V(G)| = n$, $|V(H)| = k$, *and the maximum degree of* G *is bounded by* d.

*5.4 A divide and color algorithm for Longest Path

We now improve the running time bound of the algorithm for Longest Path, by using a different choice of coloring. The idea is inspired by the "divide and conquer" approach: one of the basic algorithmic techniques to design polynomial-time algorithms. In other words, we will see a technique that is an amalgamation of divide and conquer and color coding. Recall that for color coding we used k colors to make the subgraph we are seeking colorful. A natural question is: Do we always need k colors, or in some cases can we reduce the number of colors and hence the randomness used in the algorithm? Some problems can naturally be decomposed into disjoint pieces and solutions to individual pieces can be combined together to get the complete solution. The idea of *divide and color* is to use randomization to separate the pieces. For example, in the case of graph problems, we will randomly partition all vertices (or edges) of a graph into the left and the right side, and show that the structure we were looking for has been conveniently split between the sides. We solve the problem recursively on a graph induced on left and right sides separately and then combine them to get the structure we are searching for.

Thus, the workflow of the algorithm is a bit different from the one in the previous color coding example. We perform the partition step at every node of the recursion tree, solving recursively the same problem in subinstances. The leaves of the recursion tree correspond to trivial instances with $k = \mathcal{O}(1)$, where no involved work is needed. On the other hand, the work needed to glue the information obtained from subinstances to obtain a solution for the current instance can be seen as a variant of dynamic programming.

The basic idea of the divide and color technique for the Longest Path problem is to randomly assign each vertex of the graph G to either a set L (left) or another set R (right) with equal probability and thus obtain a partitioning of the vertices of the input graph. By doing this, we hope that there is a k-path P such that the first $\lceil \frac{k}{2} \rceil$ vertices of P are in L and the last $\lfloor \frac{k}{2} \rfloor$ vertices of P are in R. Observe that for a fixed k-path

```
Simple-Randomized-Paths($X, \ell$)
Input:  A subset $X \subseteq V(G)$ and an integer $\ell$, $1 \le \ell \le k$
Output: $\widehat{D}_{X,\ell}$

1. If $\ell = 1$, then return $\widehat{D}_{X,\ell}[v, v] = \top$ for all $v \in X$, $\bot$ otherwise.
2. Uniformly at random partition $X$ into $L$ and $R$.
3. $\widehat{D}_{L,\lceil \frac{\ell}{2} \rceil}$:=Simple-Randomized-Paths($L, \lceil \frac{\ell}{2} \rceil$)
4. $\widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$:=Simple-Randomized-Paths($R, \lfloor \frac{\ell}{2} \rfloor$)
5. Return $\widehat{D}_{X,\ell} := \widehat{D}_{L,\lceil \frac{\ell}{2} \rceil} \bowtie \widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$
```

Fig. 5.1: A simple algorithm to find a path on k vertices

this happens with probability exactly 2^{-k}. After the partitioning we recurse on the two induced graphs $G[L]$ and $G[R]$.

However, the naive implementation of this scheme gives a success probability worse than $2^{-\mathcal{O}(k)}$, and hence cannot be used to obtain an algorithm with running time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ and constant success probability. We need two additional ideas to make this scheme work. First, we need to define the recursion step in a way that it considers all possible pairs of endpoints for the path. We present first an algorithm implementing this idea. Then we improve the algorithm further by observing that we can do better than selecting a single random partition in each recursion step and then repeating the whole algorithm several times to increase the probability of success. Instead, in each recursive step, we create several random partitions and make several recursive calls.

In order to combine the subpaths obtained from $G[L]$ and $G[R]$, we need more information than just one k-vertex path from each side. For a given subset $X \subseteq V(G)$, a number $\ell \in \{1, \ldots, k\}$ and a pair of vertices u and v of X let $D_{X,\ell}[u, v]$ denote a Boolean value equal to true if and only if there exists an ℓ-vertex path from u to v in $G[X]$. Note that for a fixed subset X, we can consider $D_{X,\ell}$ as an $|X| \times |X|$ matrix. In what follows, we describe a procedure Simple-Randomized-Paths(X, ℓ) that computes a matrix $\widehat{D}_{X,\ell}$. The relation between $\widehat{D}_{X,\ell}$ and $D_{X,\ell}$ is the following. If $\widehat{D}_{X,\ell}[u, v]$ is true for some $u, v \in X$, then so is $D_{X,\ell}[u, v]$. The crux of the method will be in ensuring that if $D_{X,\ell}[u, v]$ is true for some fixed u and v, then $\widehat{D}_{X,\ell}[u, v]$ is also true with sufficiently high probability. Thus, we will get a one-sided error Monte Carlo algorithm.

Given a partition (L, R) of X, an $|L| \times |L|$ matrix A, and an $|R| \times |R|$ matrix B, we define $A \bowtie B$ as an $|X| \times |X|$ Boolean matrix D. For every pair of vertices $u, v \in X$ we have $D[u, v]$ equal to true if and only if $u \in L$, $v \in R$

and there is an edge $xy \in E(G[X])$ such that $x \in L$, $y \in R$ and both $A[u,x]$ and $B[y,v]$ are set to true. Then, $(D_{L,\lceil \frac{\ell}{2} \rceil} \bowtie D_{R,\lfloor \frac{\ell}{2} \rfloor})[u,v]$ is true if and only if there exists an ℓ-path P in $G[X]$ from u to v, whose first $\lceil \frac{\ell}{2} \rceil$ vertices belong to L and the remaining $\lfloor \frac{\ell}{2} \rfloor$ vertices belong to R. In particular, $(D_{L,\lceil \frac{\ell}{2} \rceil} \bowtie D_{R,\lfloor \frac{\ell}{2} \rfloor})[u,v]$ implies $D_{X,\ell}[u,v]$. The main observation is that if $D_{X,\ell}[u,v]$ is true, then one can hope that the random choice of L and R partitions the vertices of the path P correctly, i.e., so that $(D_{L,\lceil \frac{\ell}{2} \rceil} \bowtie D_{R,\lfloor \frac{\ell}{2} \rfloor})[u,v]$ is also true with some sufficiently large probability. Thus, we compute the matrices $\widehat{D}_{L,\lceil \frac{\ell}{2} \rceil}$ and $\widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$ recursively by invoking Simple-Randomized-Paths$(L, \lceil \frac{\ell}{2} \rceil)$ and Simple-Randomized-Paths$(R, \lfloor \frac{\ell}{2} \rfloor)$, respectively, and then return $\widehat{D}_{X,\ell} := \widehat{D}_{L,\lceil \frac{\ell}{2} \rceil} \bowtie \widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$. For the base case in this recurrence, we take $\ell = 1$ and compute $\widehat{D}_{X,1} = D_{X,1}$ directly from the definition.

Let us analyze the probability that, if $D_{X,\ell}[u,v]$ is true for some $u, v \in V(G)$, then $\widehat{D}_{X,\ell}[u,v]$ is also true. For fixed ℓ, by p_ℓ denote the infinimum of this probability for all graphs G, all sets $X \subseteq V(G)$, and all vertices $u, v \in X$. If $\ell = 1$, then $\widehat{D}_{X,\ell} = D_{X,\ell}$ and, consequently, $p_1 = 1$. Otherwise, let P be any ℓ-path in $G[X]$ with endpoints u and v; we now analyze how the path P can be "detected" by the algorithm. Let x be the $\lceil \frac{\ell}{2} \rceil$-th vertex on P (counting from u), and let y be the successor of x on P. Moreover, let P_L be the subpath of P between u and x, inclusive, and let P_R be the subpath of P between y and v, inclusive. Note that, P_L has $\lceil \frac{\ell}{2} \rceil$ vertices, P_R has $\lfloor \frac{\ell}{2} \rfloor$ vertices and, as $\ell > 1$, $\lfloor \frac{\ell}{2} \rfloor \leq \lceil \frac{\ell}{2} \rceil < \ell$.

Observe that, with probability $2^{-\ell}$, all vertices of P_L are assigned to L and all vertices of P_R are assigned to R. Moreover, if this is the case, then both $D_{L,\lceil \frac{\ell}{2} \rceil}[u,x]$ and $D_{R,\lfloor \frac{\ell}{2} \rfloor}[y,v]$ are true, with the paths P_L and P_R being the respective witnesses. Consequently, $\widehat{D}_{L,\lceil \frac{\ell}{2} \rceil}[u,x]$ and $\widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}[y,v]$ are both true with probability at least $p_{\lceil \frac{\ell}{2} \rceil} p_{\lfloor \frac{\ell}{2} \rfloor}$. If this is the case, $\widehat{D}_{X,\ell}[u,v]$ is set to true by the definition of the $\bowtie$ product. Hence, we obtain the following recursive bound:

$$p_\ell \geq 2^{-\ell} p_{\lceil \frac{\ell}{2} \rceil} p_{\lfloor \frac{\ell}{2} \rfloor}. \tag{5.1}$$

By solving the recurrence (5.1), we obtain $p_\ell = 2^{-\mathcal{O}(\ell \log \ell)}$. To see this, observe that, in the search for the path P for the value $D_{X,\ell}[u,v]$, there are $\ell - 1$ partitions that the algorithm Simple-Randomized-Paths needs to find correctly: one partition of all ℓ vertices, two partitions of roughly $\ell/2$ vertices, four partitions of roughly $\ell/4$ vertices, etc. That is, for each $i = 0, 1, \ldots, \lfloor \log \ell \rfloor - 1$, the algorithm needs to make, roughly 2^i times, a correct partition of roughly $k/2^i$ vertices. Consequently, the success probability of the algorithm is given by

$$p_\ell \sim \prod_{i=0}^{\log \ell - 1} \left(\frac{1}{2^{\ell/2^i}} \right)^{2^i} = 2^{-\mathcal{O}(\ell \log \ell)}.$$

```
Faster-Randomized-Paths($X, \ell$)
Input: A subset $X \subseteq V(G)$ and an integer $\ell$, $1 \leq \ell \leq k$
Output: $D_{X,\ell}$

1. If $\ell = 1$, then return $D_{X,\ell}[v, v] = \top$ for all $v \in X$, $\bot$ otherwise.
2. Set $\widehat{D}_{X,\ell}[u, v]$ to false for any $u, v \in X$.
3. Repeat the following $2^\ell \log(4k)$ times
    (a) Uniformly at random partition $X$ into $L$ and $R$.
    (b) $\widehat{D}_{L,\lceil \frac{\ell}{2} \rceil}$:=Faster-Randomized-Paths($L, \lceil \frac{\ell}{2} \rceil$)
    (c) $\widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$:=Faster-Randomized-Paths($R, \lfloor \frac{\ell}{2} \rfloor$)
    (d) Compute $\widehat{D}'_{X,\ell} := \widehat{D}_{L,\lceil \frac{\ell}{2} \rceil} \bowtie \widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$ and update
        $\widehat{D}_{X,\ell}[u, v] := \widehat{D}_{X,\ell}[u, v] \vee \widehat{D}'_{X,\ell}[u, v]$ for every $u, v \in V(G)$.
4. Return $\widehat{D}_{X,\ell}$.
```

Fig. 5.2: A faster algorithm to find a path on k-vertices

Thus, to achieve a constant success probability, we need to run the algorithm Simple-Randomized-Paths($V(G), k$) $2^{\mathcal{O}(k \log k)}$ number of times, obtaining a significantly worse running time bound than the one of Theorem 5.6.

Let us now try to improve the running time of our algorithm. The main idea is that instead of repeating the entire process $2^{\mathcal{O}(k \log k)}$ times, we split the repetitions between the recursive calls. More precisely, we choose some function $f(\ell, k)$ and, in each recursive call, we try not only one random partition of $V(G)$ into L and R, but $f(\ell, k)$ partitions, each chosen independently at random. Recall that ℓ denotes the argument of the recursive call (the length of paths we are currently looking for), whereas k denotes the argument of the root call to Faster-Randomized-Paths (the length of a path we are looking for in the entire algorithm). In this manner, we increase the running time of the algorithm, but at the same time we increase the success probability. Let us now analyze what function $f(\ell, k)$ guarantees constant success probability.

Recall that a fixed partition of the vertices into L and R suits our needs for recursion (i.e., correctly partitions the vertex set of one fixed ℓ-path) with probability $2^{-\ell}$. Hence, if we try $f(\ell, k)$ partitions chosen independently at random, the probability that none of the $f(\ell, k)$ partitions suits our needs is at most

$$\left(1 - \frac{1}{2^\ell}\right)^{f(\ell,k)}. \tag{5.2}$$

Fix a k-vertex path P witnessing that $D_{V(G),k}[u, v]$ is true for some fixed $u, v \in V(G)$. The main observation is that, although the entire recursion tree is huge — as we shall later see, it will be of size roughly 4^k — we do not need to be successful in all nodes of the recursion tree. First, we need to guess correctly at least one partition in the root node of the recursion tree. Then, we may limit ourselves to the two recursive calls made for that one particular correct partition at the root node, and insist that in these two recursive calls we guess at least one correct partition of the corresponding subpaths of length $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$. If we proceed with such a reasoning up to the leaves of the recursion, we infer that we need to guess correctly at least one partition at exactly $k-1$ nodes. Consequently, we obtain constant success probability if the value of the expression (5.2) is at most $\frac{1}{2(k-1)}$ for every $1 \leq \ell \leq k$. To achieve this bound, observe that we can choose $f(\ell, k) = 2^\ell \log(4k)$. Indeed, since $1 + x \leq e^x$ for every real x,

$$\left(1 - \frac{1}{2^\ell}\right)^{2^\ell \log(4k)} \leq \left(e^{-\frac{1}{2^\ell}}\right)^{2^\ell \log(4k)} = \frac{1}{e^{\log(4k)}} \leq \frac{1}{2k}.$$

Since the probability space in the algorithm is quite complex, some readers may find the argumentation in the previous paragraph too informal. For sake of completeness, we provide a formal argumentation in the next lemma.

Lemma 5.8. *If (G, k) is a yes-instance and $f(\ell, k) = 2^\ell \log(4k)$ (i.e., the value of the expression (5.2) is at most $\frac{1}{2(k-1)}$ for every $1 \leq \ell \leq k$), then with probability at least $\frac{1}{2}$ there exists a pair of vertices $u, v \in V(G)$ with $\widehat{D}_{V(G),k}[u, v]$ equal to true.*

Proof. Consider a set $X \subseteq V(G)$, vertices $u, v \in V(G)$, an integer $1 \leq \ell \leq k$, and an ℓ-vertex path P in $G[X]$ with endpoints u and v. Clearly, P witnesses that $D_{X,\ell}[u, v] = 1$. Let p_ℓ^k denote the infinimum, for fixed k and ℓ, over all choices of G, X, u, and v such that $D_{X,\ell}[u, v] = 1$, of the probability that a call to Faster-Randomized-Paths(X, ℓ) on instance (G, k) returns $\widehat{D}_{X,\ell}[u, v] = 1$. Let $s = \frac{1}{2(k-1)}$. We prove by induction on ℓ that for every $1 \leq \ell \leq k$ we have $p_\ell^k \geq 1 - (\ell - 1)s$; note that the lemma follows from such a claim for $\ell = k$. In the base case, observe that $p_1^k = 1$ as $\widehat{D}_{X,1} = D_{X,1}$ for every $X \subseteq V(G)$.

For the induction step, consider a path P in $G[X]$, with $\ell > 1$ vertices and endpoints u and v. Observe that at the call Faster-Randomized-Paths(X, ℓ) to set $\widehat{D}_{X,\ell}[u, v] = 1$ it suffices to (a) at least once guess a correct partition for the path P; and (b) for the first such correct partition (L, R) compute $\widehat{D}_{L,\lceil \ell/2 \rceil}[u, x] = 1$ and $\widehat{D}_{R,\lfloor \ell/2 \rfloor}[y, v] = 1$, where x is the $\lceil \frac{\ell}{2} \rceil$-th vertex on P (counting from u), and y is the successor of x on P. Consequently,

$$\begin{aligned} p_\ell^k &\geq (1-s) p_{\lceil \ell/2 \rceil}^k p_{\lfloor \ell/2 \rfloor}^k \\ &\geq (1-s) \cdot (1 - (\lceil \ell/2 \rceil - 1)s) \cdot (1 - (\lfloor \ell/2 \rfloor - 1)s) \\ &\geq 1 - s - (\lceil \ell/2 \rceil - 1)s - (\lfloor \ell/2 \rfloor - 1)s \\ &= 1 - (\ell - 1)s. \end{aligned}$$

Here, the second inequality follows from the inductive assumption, as $\ell > 1$ implies $\lfloor \ell/2 \rfloor \leq \lceil \ell/2 \rceil < \ell$. $\square$

For ease of presentation, the modified algorithm with improved error probability is given on Fig. 5.2.

Let us now analyze the running time of the algorithm: due to the choice of $f(\ell, k) = 2^\ell \log(4k)$, it is no longer polynomial. We have the following recursive formula:

$$T(n, \ell, k) \leq 2^\ell \log(4k) \left(T\left(n, \left\lceil \frac{\ell}{2} \right\rceil, k\right) + T\left(n, \left\lfloor \frac{\ell}{2} \right\rfloor, k\right) \right) + n^{\mathcal{O}(1)}. \quad (5.3)$$

This solves to $T(n, \ell, k) = 4^{\ell + o(\ell + k)} n^{\mathcal{O}(1)}$, and we obtain the following improvement upon Theorem 5.6.

Theorem 5.9. *There exists a randomized algorithm that, given a* LONGEST PATH *instance* (G, k), *in time* $4^{k+o(k)} n^{\mathcal{O}(1)}$ *either reports a failure or finds a path on* k *vertices in* G. *Moreover, if the algorithm is given a yes-instance, it returns a solution with a constant probability.*

5.5 A chromatic coding algorithm for d-CLUSTERING

We now move our attention to *edge modification problems*, where we are to modify the input graph G by adding or deleting a small number of edges to obtain a desired property (i.e., make a graph belong to a prescribed graph class). The idea of *chromatic coding* (or *color and conquer*, as it is sometimes referred to in the literature) is to randomly color the *vertices* of the input graph, with the hope that for every edge uv of the solution, i.e., the set of edges that have to be added/deleted, the colors assigned to u and v will differ. Once this random coloring step is successfully completed, the problem instance seems to have a nice structure which can be exploited for algorithmic applications. In particular, after a successful coloring, we can observe the following.

1. All the solution edges go across color classes.
2. Therefore, no modification happens within a color class, hence the subgraph induced by each color class is already a member of the target graph class.

While in some cases such properties significantly help us in resolving the problem at hand, the crux of the chromatic coding approach is that, in a yes-instance where the number of modifications in the solution is bounded by k, the number of colors that guarantee the desired property with good probability is *sublinear* in k. Consequently, the algorithms obtained via chromatic coding are usually *subexponential*, i.e., they run in $2^{o(k)}n^{\mathcal{O}(1)}$ time.

As we discuss in Chapter 14, subexponential parameterized algorithms are relatively rare.

Probability of a good coloring. Before we dive into a specific exemplary problem for chromatic coding, let us formally analyze the random coloring step — as it is generic for all algorithms using this approach. For a graph G, we say that a coloring $\chi : V(G) \to [q]$ *properly colors* $E(G)$ if for every $uv \in E(G)$ we have $\chi(u) \neq \chi(v)$.

Lemma 5.10. *If the vertices of a simple graph G on k edges are colored independently and uniformly at random with $\lceil \sqrt{8k} \rceil$ colors, then the probability that $E(G)$ is properly colored is at least $2^{-\sqrt{k/2}}$.*

Proof. Let $n = |V(G)|$. Consider the sequence of vertices generated by the following procedure.

1. Initialize $G_0 = G$.
2. For every $i = 1, 2, \ldots, n$, repeat the following procedure. Let v_i be a vertex of minimum possible degree in G_{i-1}. Remove v_i from G_{i-1} to obtain the graph G_i (i.e., define $G_i := G_{i-1} - v_i$).

Observe that G_n is empty.

Let d_i be the degree of v_i in G_{i-1}. Since we are dealing with simple graphs, for every $i = 1, \ldots, n$ we have

$$d_i \leq |V(G_{i-1})| - 1. \tag{5.4}$$

Moreover, since v_i is of minimum possible degree in G_{i-1}, we have

$$2k = 2|E(G)| \geq 2|E(G_{i-1})| \geq d_i \cdot |V(G_{i-1})|. \tag{5.5}$$

By merging (5.4) and (5.5) we obtain $2k \geq d_i^2$, that is, $d_i \leq \sqrt{2k}$.

Let us now consider the process of the random choice of the coloring χ by iteratively coloring the vertices in the order $v_n, v_{n-1}, \ldots, v_1$; each vertex is colored independently and uniformly at random with one of $q := \lceil \sqrt{8k} \rceil$ colors. Let P_i denote the event when the edge set $E(G_i)$ is properly colored. Clearly, $\Pr(P_n) = 1$ and $P_j \subseteq P_i$ for $j \leq i$. Let us see what happens when we color the vertex v_i, $1 \leq i \leq n$. There are d_i edges connecting v_i with $V(G_i)$, thus at most d_i possible values of $\chi(v_i)$ would cause some edge v_iv_j, $j > i$, to be incorrectly colored (i.e., $\chi(v_i) = \chi(v_j)$). Consequently, we obtain

$$\Pr(P_{i-1} \mid P_i) \geq \frac{q - d_i}{q} = 1 - \frac{d_i}{q} \geq 2^{-2d_i/q}, \tag{5.6}$$

where the last inequality follows from the estimate $1-x \geq 2^{-2x}$ for $0 \leq x \leq \frac{1}{2}$ (recall that $d_i \leq \sqrt{2k}$, that is, $\frac{d_i}{q} \leq \frac{1}{2}$).

Observe that P_0 is the event when the edge set $E(G_0) = E(G)$ is properly colored. Moreover, $\sum_{i=1}^{n} d_i = k$, as every edge is counted exactly once in the sum $\sum_{i=1}^{n} d_i$. By applying $P_{i-1} \subseteq P_i$ and (5.6) for every $1 \leq i \leq n$ we obtain:

$$\begin{aligned}
\Pr(P_0) &= \Pr(P_0 \mid P_1)\Pr(P_1) \\
&= \Pr(P_0 \mid P_1)\Pr(P_1 \mid P_2)\Pr(P_2) \\
&\quad\vdots \\
&= \Pr(P_n)\prod_{i=1}^{n}\Pr(P_{i-1} \mid P_i) \\
&\geq \prod_{i=1}^{n} 2^{-2d_i/q} = 2^{-2\sum_{i=1}^{n} d_i/q} \\
&= 2^{-2\cdot k/\lceil\sqrt{8k}\rceil} \geq 2^{-\sqrt{k/2}}.
\end{aligned}$$

□

In chromatic coding,

- we use $\mathcal{O}(\sqrt{k})$ colors, and
- the success probability is $2^{-\mathcal{O}(\sqrt{k})}$ (Lemma 5.10).

Therefore, if the running time of the algorithm is single-exponential in the number of colors, i.e., for c colors it runs in time $2^{\mathcal{O}(c)}n^{\mathcal{O}(1)}$, then $2^{\mathcal{O}(\sqrt{k})}$ repetitions give a $2^{\mathcal{O}(\sqrt{k})}n^{\mathcal{O}(1)}$-time algorithm with constant error probability. Contrast this with color coding, where

- we use $\mathcal{O}(k)$ colors, and
- the success probability is $2^{-\mathcal{O}(k)}$.

Thus if the running time of the algorithm is single-exponential in the number of colors, then $2^{\mathcal{O}(k)}$ repetitions give a $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$-time algorithm with constant error probability.

The d-Clustering problem. Armed with Lemma 5.10, we exemplify the chromatic coding approach on the d-Clustering problem.

A graph H is called an *ℓ-cluster graph* if H has ℓ connected components, and every connected component of H is a complete graph (clique). A graph H is a *cluster graph* if it is an ℓ-cluster graph for some ℓ. For a graph G, by *adjacencies in G* we mean pairs in $\binom{V(G)}{2}$. For a subset $A \subseteq \binom{V(G)}{2}$, by *modifying the adjacencies A in G* we mean an operation $G \oplus A$ that creates a graph with vertex set $V(G \oplus A) = V(G)$ and edge set $E(G \oplus A) =$

$(E(G) \setminus A) \cup (A \setminus E(G))$. In other words, for any $uv \in A$, if $uv \in E(G)$ then we delete uv from G, and otherwise we add uv to G. In the d-CLUSTERING problem, we are given an undirected graph G and a nonnegative integer k, and we ask whether there exists a set A of at most k adjacencies in G such that $G \oplus A$ is a d-cluster graph. Such a set A is henceforth called a *solution* to the instance (G, k). We remark here that the integer d is a part of the problem definition, and is considered constant through the algorithm.

In the first step of our algorithm, given an instance (G, k) of d-CLUSTERING, we pick $q := \lceil \sqrt{8k} \rceil$ and randomly color the vertices of G with q colors. Let $\chi : V(G) \to [q]$ be the obtained coloring. We say that a set of adjacencies $A \subseteq \binom{V(G)}{2}$ is *properly colored* if χ properly colors the graph $(V(G), A)$. Lemma 5.10 ensures that, if (G, k) is a yes-instance and A is a solution to (G, k), A is properly colored with sufficiently high probability. Our goal now is to show an algorithm that efficiently seeks for a properly colored solution.

Solving a colored instance. Assume we are given a d-CLUSTERING instance (G, k) and a coloring χ. We start with the following simple observation.

Lemma 5.11. *Let G be a graph and $\chi : V(G) \to [q]$ be a coloring function. Furthermore, let V_i denote the vertices of G that are colored i. If there exists a solution A in G that is properly colored by χ, then for every V_i, $G[V_i]$ is an ℓ-cluster graph for some $\ell \leq d$.*

Proof. Since every edge in A has endpoints of different colors, each of the subgraphs $G[V_i]$ is an induced subgraph of $G \oplus A$. But $G \oplus A$ is a d-cluster graph, and every induced subgraph of a cluster graph is also a cluster graph. Hence, $G[V_i]$ is a cluster graph with at most d components. □

We now proceed to the description of the algorithm. Suppose the given instance has a properly colored solution A. Lemma 5.11 implies that $G[V_i]$ is an ℓ-cluster graph for some $\ell \leq d$ and it remains so in $G \oplus A$. Therefore, every component of $G[V_i]$ is a clique and it lies in one of the d components of the graph $G \oplus A$; for each such component let us guess where it lies in $G \oplus A$. Since the total number of components of the subgraphs $G[V_i]$ is bounded by $dq = d\lceil \sqrt{8k} \rceil$, we have at most $d^{d\lceil \sqrt{8k} \rceil} = 2^{\mathcal{O}(d \log d \sqrt{k})}$ choices. Finally, it simply remains to check if the number of edges to be added and deleted to be consistent with the guess is at most k. All we need to do is to add to A any pair of adjacent vertices that are guessed to be in different components of $G \oplus A$ and any pair of nonadjacent vertices that are guessed to be in the same component of $G \oplus A$. Hence, we obtain the following.

Lemma 5.12. *Given an instance (G, k) of d-CLUSTERING with a coloring $\chi : V(G) \to [\lceil \sqrt{8k} \rceil]$, we can test if there is a properly colored solution in time $2^{\mathcal{O}(d \log d \sqrt{k})} n^{\mathcal{O}(1)}$.*

Using a simple dynamic-programming approach one can show the following (see Exercise 5.7).

Lemma 5.13. *Given an instance (G, k) of d-*Clustering *with a coloring $\chi : V(G) \to [\lceil\sqrt{8k}\rceil]$, we can test if there is a properly colored solution in time $2^{\mathcal{O}(d\sqrt{k})}n^{\mathcal{O}(1)}$.*

Combining Lemmas 5.10 and 5.13, we get that there is a randomized FPT algorithm for d-Clustering running in time $2^{\mathcal{O}(d\sqrt{k})}n^{\mathcal{O}(1)}$ with a success probability of at least $2^{-\mathcal{O}(\sqrt{k})}$. Thus, repeating the above algorithm independently $2^{\mathcal{O}(\sqrt{k})}$ times we obtain a constant success probability. This leads to the following theorem.

Theorem 5.14. *There exists a randomized algorithm that, given an instance (G, k) of d-*Clustering*, in time $2^{\mathcal{O}(d\sqrt{k})}n^{\mathcal{O}(1)}$ either reports a failure or finds a solution to (G, k). Moreover, if the algorithm is given a yes-instance, it returns a solution with a constant probability.*

5.6 Derandomization

It turns out that all presented algorithms based on some variant of the color coding technique can be efficiently derandomized. The goal of this section is to show basic tools for such derandomization.

The basic idea of derandomization is as follows: instead of picking a random coloring $\chi : [n] \to [k]$, we deterministically construct a family $\mathcal{F}$ of functions $f : [n] \to [k]$ such that it is guaranteed that one of the functions from $\mathcal{F}$ has the property that we hope to attain by choosing a random coloring χ.

Quite surprisingly, it turns out that such families $\mathcal{F}$ of small cardinality exist and can be constructed efficiently. In particular, if we estimated the probability that a random coloring χ satisfies the desired property by $1/f(k)$, the size of the corresponding family $\mathcal{F}$ is usually not much larger than $f(k) \log n$. Consequently, we are able to derandomize most of the algorithms based on variants of the color coding technique with only small loss of efficiency.

We limit ourselves only to formulating the appropriate results on constructions of pseudorandom objects we need, and showing how to derandomize the algorithms presented earlier in this section. The actual methods behind the constructions, although very interesting, are beyond the scope of this book.

5.6.1 Basic pseudorandom objects

The basic pseudorandom object we need is called a *splitter.*

Definition 5.15. An (n, k, ℓ)-*splitter* $\mathcal{F}$ is a family of functions from $[n]$ to $[\ell]$ such that for every set $S \subseteq [n]$ of size k there exists a function $f \in \mathcal{F}$ that *splits* S *evenly.* That is, for every $1 \leq j, j' \leq \ell$, $|f^{-1}(j) \cap S|$ and $|f^{-1}(j') \cap S|$ differ by at most 1.

One can view a function f in a splitter $\mathcal{F}$ as a coloring of $[n]$ into ℓ colors; the set S is split evenly if f uses on S each color roughly the same number of times. Note that for $\ell \geq k$, the notion of "splitting S evenly" reduces to f being injective on S. Indeed, if $|f^{-1}(j) \cap S| \geq 2$ for some j, then for every j' we have $|f^{-1}(j') \cap S| \geq 1$ and hence $|S| \geq |f^{-1}([\ell]) \cap S| \geq \ell + 1 > |S|$, a contradiction.

Our basic building block for derandomization is the following construction.

Theorem 5.16 ([15]). *For any $n, k \geq 1$ one can construct an (n, k, k^2)-splitter of size $k^{\mathcal{O}(1)} \log n$ in time $k^{\mathcal{O}(1)} n \log n$.*

We emphasize here that the size of the splitter provided by Theorem 5.16 is very small: it is polynomial in k, and depends only logarithmically on n (Exercise 5.18 asks you to show that this dependence on n is optimal.) This is achieved by allowing the members of a splitter to have a relatively large codomain, namely of size k^2.

However, observe that the case of a much smaller codomain, namely $\ell = k$, directly corresponds to the color coding algorithm for Longest Path: we were coloring the vertex set of a graph G with k colors, hoping to be injective on the vertex set of one fixed k-path in G. This special case of a splitter is known as a perfect hash family.

Definition 5.17. An (n, k, k)-splitter is called an (n, k)-*perfect hash family.*

To obtain a perfect hash family, we need to allow the splitter to be significantly larger than in Theorem 5.16.

Theorem 5.18 ([372]). *For any $n, k \geq 1$ one can construct an (n, k)-perfect hash family of size $e^k k^{\mathcal{O}(\log k)} \log n$ in time $e^k k^{\mathcal{O}(\log k)} n \log n$.*

The proof of Theorem 5.18 is derived by a composition of an (n, k, k^2)-splitter $\mathcal{F}_1$ obtained from Theorem 5.16 with an explicit construction of a small (k^2, k, k)-splitter $\mathcal{F}_2$, i.e., the resulting (n, k)-perfect hash family is $\mathcal{F} = \{f_2 \circ f_1 \ : \ f_1 \in \mathcal{F}_1, f_2 \in \mathcal{F}_2\}$. (We denote by $f_2 \circ f_1$ the composition of the two functions obtained by performing f_1 first and then f_2, that is, $(f_2 \circ f_1)(x) = f_2(f_1(x))$.)

Observe that the actual size bound of Theorem 5.18 is very close to the inverse of the probability that a random function $[n] \to [k]$ is injective on a given set $S \subseteq [n]$ of size k (Lemma 5.4). Hence, if we replace repeating a color coding algorithm e^k times with iterating over all elements of an (n, k)-perfect hash family of Theorem 5.18, we obtain a deterministic algorithm with only a very slight increase in the running time bound.

For the random separation and divide and color algorithms, recall that we needed to capture a specific partition either of the set $\Gamma \cup E(\widehat{H})$, or of the vertex set of a path P in question. Hence, we need a notion that is able to capture all partitions of a given small subset of the universe.

Definition 5.19. An (n, k)-*universal set* is a family $\mathcal{U}$ of subsets of $[n]$ such that for any $S \subseteq [n]$ of size k, the family $\{A \cap S \ : \ A \in \mathcal{U}\}$ contains all 2^k subsets of S.

Similar to the case of perfect hash families, it is possible to obtain a universal set of size very close to the inverse of the probability that a random subset of $[n]$ has a desired intersection with a fixed set $S \subseteq [n]$ of size k.

Theorem 5.20 ([372]). *For any $n, k \geq 1$ one can construct an (n, k)-universal set of size $2^k k^{\mathcal{O}(\log k)} \log n$ in time $2^k k^{\mathcal{O}(\log k)} n \log n$.*

As in the case of Theorem 5.18, the proof of Theorem 5.20 boils down to an explicit construction of a (k^2, k)-universal set and composing it with a splitter of Theorem 5.16.

Finally, for the chromatic coding approach, we need to revise the coloring step and observe that in fact we do not need the colors of vertices to be independent in general; limited independence is sufficient. This brings us to the following definition.

Definition 5.21 (k-wise independent). A family $H_{n,k,q}$ of functions from $[n]$ to $[q]$ is called a *k-wise independent sample space* if, for every k positions $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and every tuple $\alpha \in [q]^k$, we have

$$\Pr((f(i_1), f(i_2), \ldots, f(i_k)) = \alpha) = q^{-k}$$

where the function $f \in H_{n,k,q}$ is chosen uniformly at random.

Fortunately, there are known constructions of k-wise independent sample spaces of small size.

Theorem 5.22 ([10]). *There exists a k-wise independent sample space $H_{n,k,q}$ of size $\mathcal{O}(n^k)$ and it can be constructed efficiently in time linear in the output size.*

In the next section we will see that to derandomize chromatic coding it suffices to modify the 2-independent sample space from Theorem 5.22.

5.6.2 Derandomization of algorithms based on variants of color coding

We now show how to derandomize the algorithms from Section 5.2 using the pseudorandom objects of Section 5.6.1. For the sake of completeness, we first repeat the more or less straightforward arguments for color coding and divide and conquer. Then, we show a more involved argument for chromatic coding.

Color coding and Longest Path**.** Let (G, k) be the input instance for Longest Path, where $n = |V(G)|$. Instead of taking a random coloring χ of $V(G)$, we use Theorem 5.18 to construct an (n, k)-perfect hash family $\mathcal{F}$. Then, for each $f \in \mathcal{F}$, we proceed as before, that is, we invoke the dynamic-programming algorithm of Lemma 5.5 for the coloring $\chi := f$. The properties of an (n, k)-perfect hash family $\mathcal{F}$ ensure that, if there exists a k-path P in G, then there exists $f \in \mathcal{F}$ that is injective on $V(P)$ and, consequently, the algorithm of Lemma 5.5 finds a colorful path for the coloring $\chi := f$. Hence, we obtain the following deterministic algorithm.

Theorem 5.23. Longest Path *can be solved in time* $(2e)^k k^{\mathcal{O}(\log k)} n^{\mathcal{O}(1)}$ *by a deterministic algorithm.*

We emphasize here that the almost-optimal size bound of Theorem 5.18 makes the running time of the algorithm of Theorem 5.23 only slightly worse than that of Theorem 5.6.

Divide and color and Longest Path**.** Let (G, k) again be the input instance for Longest Path, where $n = |V(G)|$. We derandomize the procedure Faster-Randomized-Paths(G, k) of Section *5.4 as follows: instead of taking $f(k)$ random partitions of $V(G)$, we use Theorem 5.20 to compute an (n, k)-universal set $\mathcal{U}$ and recurse on all partitions $(L := A, R := V(G) \setminus A)$ for $A \in \mathcal{U}$. By the properties of a universal set, for every $u, v \in V(G)$ and every k-path P from u to v in G, there exists $A \in \mathcal{U}$ where $A \cap V(P)$ consists of the first $\lceil \frac{k}{2} \rceil$ vertices of P. Consequently, there always exists at least one recursive step where the path P is properly partitioned, and the algorithm actually computes $\widehat{D}_{V(G),k} = D_{V(G),k}$.

As for the running time bound, observe that the size bound of $\mathcal{U}$ promised by Theorem 5.20 is very close to the choice $f(k) = 2^k \log(4k)$ we made in Section *5.4. Consequently, if we solve the appropriate variant of recursive formula (5.3) we obtain an algorithm with almost no increase in the running time (in fact, all increase is hidden in the big-$\mathcal{O}$ notation).

Theorem 5.24. Longest Path *can be solved in time* $4^{k+o(k)} n^{\mathcal{O}(1)}$ *by a deterministic algorithm.*

The derandomization of the random separation algorithm for Subgraph Isomorphism in graphs of bounded degree, presented in Section 5.3, is straightforward and postponed to Exercise 5.19.

5.6.2.1 A derandomized chromatic coding algorithm for *d*-Clustering

Let (G, k) be the input instance for d-Clustering, where $n = |V(G)|$. Following previous examples, we would like to replace the random choice of a coloring of $V(G)$ into $q = \mathcal{O}(\sqrt{k})$ colors with an iteration over some pseudorandom object. Recall that in Section 5.5, the essential property we were hoping for was that the obtained coloring χ properly colors the solution A. This leads to the following definition:

Definition 5.25. A (n, k, q)-*coloring family* is a family $\mathcal{F}$ of functions from $[n]$ to $[q]$ with the following property: for every graph G on the vertex set $[n]$ with at most k edges, there exists a function $f \in \mathcal{F}$ that properly colors $E(G)$.

Thus, to derandomize the algorithm of Theorem 5.14, we need to provide an efficient way of constructing a small (n, k, q)-coloring family for some $q = \mathcal{O}(\sqrt{k})$. The basic idea is that a 2-wise independent sample space $H_{n,k,q}$ should be close to our needs for $q = c\sqrt{k}$ and sufficiently large constant c.

As in the case of construction of an (n, k)-perfect hash family or an (n, k)-universal set, it suffices to focus on the case $n = k^2$; the general case can be resolved by composing with an (n, k, k^2)-splitter of Theorem 5.16. Hence, we start with the following explicit construction.

Lemma 5.26. *For any $k \geq 1$, there exists a $(k^2, k, 2\lceil\sqrt{k}\rceil)$-coloring family $\mathcal{F}$ of size $2^{\mathcal{O}(\sqrt{k}\log k)}$ that can be constructed in time linear in its size.*

Proof. Denote $q = \lceil\sqrt{k}\rceil$. We use Theorem 5.22 to obtain a 2-wise independent sample space $\mathcal{G} := H_{k^2,2,q}$. Note that the size of $\mathcal{G}$ is bounded by $\mathcal{O}(k^4)$. Recall that every element $g \in \mathcal{G}$ is a function $g : [k^2] \to [q]$.

We can now describe the required family $\mathcal{F}$. For each $g \in \mathcal{G}$ and each subset $T \subset [k^2]$ of size $|T| = q$, we define a function $f_{g,T} \in \mathcal{F}$ as follows. Suppose $T = \{i_1, i_2, \ldots, i_q\}$, with $i_1 < i_2 < \ldots < i_q$. For $1 \leq j \leq q$, we define $f(i_j) = q + j$, and $f(i) = g(i)$ if $i \notin T$. Note that $f(i) \in [2q]$ for any $i \in [k^2]$, and the size of $\mathcal{F}$ is at most

$$\binom{k^2}{q}|\mathcal{G}| = 2^{\mathcal{O}(\sqrt{k}\log k)}.$$

To complete the proof we have to show that for every graph G on the set of vertices $[k^2]$ with at most k edges, there is an $f \in \mathcal{F}$ that properly colors $E(G)$. Fix such a graph G.

We use the probabilistic method, i.e., we choose T and g in the definition of the function $f_{g,T}$ in a random way, so that $f_{g,T}$ provides the required coloring for G with positive probability, which implies the existence of the desired function in $\mathcal{F}$. The idea can be sketched as follows. The function g is chosen at random in $\mathcal{G}$, and is used to properly color all but at most q

edges of $E(G)$. The set T is chosen to contain at least one endpoint of each of these edges, and the vertices in the set T will be re-colored by a unique color that is used only once by $f_{g,T}$. Then surely any edge incident to T will have different colors on its endpoints. Using the properties of $\mathcal{G}$ we now prove that with positive probability the number of edges $uv \in E(G)$ for which $g(u) = g(v)$ (henceforth called *monochromatic edges*) is bounded by $\sqrt{k}$.

Claim 5.27. *If the vertices of G are colored by a function g chosen at random from $\mathcal{G}$, then the expected number of monochromatic edges is at most $k/q \leq \sqrt{k}$.*

Proof. Fix an edge e in the graph G and $j \in [q]$. As g maps the vertices in a pairwise independent manner, the probability that both the endpoints of e get mapped to j is precisely $\frac{1}{q^2}$. There are q possibilities for j and hence the probability that e is monochromatic is $\frac{1}{q}$. Let X be the random variable denoting the number of monochromatic edges. By linearity of expectation, the expected value of X is at most $k \cdot \frac{1}{q} \leq \sqrt{k}$. ⌟

Returning to the proof of the lemma, observe that by the above claim, with positive probability the number of monochromatic edges is upper bounded by $q = \lceil \sqrt{k} \rceil$. Fix a $g \in \mathcal{G}$ for which this holds and let T be a set of q vertices containing at least one endpoint of every monochromatic edge. Consider the function $f_{g,T}$. As mentioned above, $f_{g,T}$ colors each of the vertices in T by a unique color, which is used only once by $f_{g,T}$, and hence we only need to consider the coloring $f_{g,T}$ restricted to $G \setminus T$. However all edges of $G \setminus T$ are properly colored by g and $f_{g,T}$ coincides with g on $G \setminus T$. Hence $f_{g,T}$ properly colors $E(G)$, completing the proof of the lemma. □

Now we are ready to state the main result of this section.

Theorem 5.28. *For any $n, k \geq 1$, there exists an $(n, k, 2\lceil \sqrt{k} \rceil)$-coloring family $\mathcal{F}$ of size $2^{\mathcal{O}(\sqrt{k} \log k)} \log n$ that can be constructed in time $2^{\mathcal{O}(\sqrt{k} \log k)} n \log n$.*

Proof. First, use Theorem 5.16 to obtain an (n, k, k^2)-splitter $\mathcal{F}_1$. Second, use Lemma 5.26 to obtain a $(k^2, k, \lceil \sqrt{k} \rceil)$-coloring family $\mathcal{F}_2$. Finally, define $\mathcal{F} := \{f_2 \circ f_1 \ : \ f_1 \in \mathcal{F}_1, f_2 \in \mathcal{F}_2\}$. A direct check shows that $\mathcal{F}$ satisfies all the desired properties. □

As announced at the beginning of this section, if we replace the random choice of a coloring in Theorem 5.14 with an iteration over the coloring family given by Theorem 5.28, we obtain a deterministic subexponential time algorithm for d-Clustering.

Theorem 5.29. *d-Clustering can be solved in time $2^{\mathcal{O}(\sqrt{k}(d+\log k))} n^{\mathcal{O}(1)}$.*

Note that for $d = \mathcal{O}(1)$, the derandomization step resulted in an additional $\log k$ factor in the exponent, as compared to Theorem 5.14.

Exercises

5.1. In the Triangle Packing problem, we are given an undirected graph G and a positive integer k, and the objective is to test whether G has k-vertex disjoint triangles. Using color coding show that the problem admits an algorithm with running time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$.

5.2. In the Tree Subgraph Isomorphism, we are given an undirected graph G and a tree T on k vertices, and the objective is to decide whether there exists a subgraph in G that is isomorphic to T. Obtain a $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$-time algorithm for the problem using color coding.

5.3. Consider the following problem: given an undirected graph G and positive integers k and q, find a set X of at most k vertices such that $G \setminus X$ has at least two components of size at least q. Show that this problem can be solved in time $2^{\mathcal{O}(q+k)}n^{\mathcal{O}(1)}$.

5.4 (☠). Assuming $q > k$, solve the problem from the previous problem in randomized time $q^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$. Can you derandomize your algorithm without any significant loss in the running time?

5.5 (🍃). Show formally the bound $p_\ell = 2^{-\mathcal{O}(\ell \log \ell)}$ in the analysis of the procedure Simple-Randomized-Paths(X, ℓ) in Section *5.4. That is, show that for sufficiently large c, $p_\ell = 2^{-c\ell \log \ell}$ satisfies (5.1).

5.6 (🍃). Solve formally the recurrence (5.3).

5.7. Prove Lemma 5.13.

5.8. Give a randomized FPT algorithm for the problem of deciding whether a given undirected graph contains a cycle of length at least k. Your algorithm should have running time $c^k n^{\mathcal{O}(1)}$. Note that a graph may not have any cycle of length exactly k, but contain a much longer cycle. Derandomize your algorithm using perfect hash families.

5.9 (☠). Give a randomized FPT algorithm for the problem of deciding whether a given directed graph contains a cycle of length at least k. Your algorithm should use color coding with k^2 colors, and have running time $2^{\mathcal{O}(k^2)}n^{\mathcal{O}(1)}$. Then improve the running time to $k^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ by only using colors $\{1, \ldots, k+1\}$, and assigning color $k+1$ with probability $1 - \frac{1}{k^2}$. Finally, derandomize your algorithm using (n, k^2, k^4)-splitters.

5.10. In the Set Splitting problem, we are given a family of sets $\mathcal{F}$ over a universe U and a positive integer k, and the goal is to test whether there exists a coloring of U with two colors such that at least k sets in $\mathcal{F}$ are not monochromatic (that is, they contain vertices of both colors).

1. Obtain a randomized FPT algorithm with running time $2^k(|U| + |\mathcal{F}|)^{\mathcal{O}(1)}$.
2. Using universal sets derandomize your algorithm and obtain a running time bound $4^k(|U| + |\mathcal{F}|)^{\mathcal{O}(1)}$.

5.11. In the Partial Vertex Cover problem, we are given an undirected graph G and positive integers k and t, and the goal is to check whether there exists a set $X \subseteq V(G)$ of size at most k such that at least t edges of G are incident to vertices on X. Obtain an algorithm with running time $2^{\mathcal{O}(t)}n^{\mathcal{O}(1)}$ for the problem. (We show in Chapter 13 that this problem is W[1]-hard parameterized by the solution size k.)

5.12. In the Partial Dominating Set problem, we are given an undirected graph G and positive integers k and t, and the goal is to check whether there exists a set $X \subseteq V(G)$ of size at most k such that $|N_G[X]| \geq t$ (that is, X dominates at least t vertices). Obtain an algorithm with running time $2^{\mathcal{O}(t)}n^{\mathcal{O}(1)}$ for the problem. (We show in Chapter 13 that this problem is W[1]-hard parameterized by the solution size k.)

5.13. A graph H is called *r-colorable* if there exists a function $\chi : V(H) \to [r]$ such that $\chi(u) \neq \chi(v)$ for every $uv \in E(H)$. Consider the following problem: given a perfect graph G and integers k and r, check whether G admits an r-colorable induced subgraph on at least k vertices. Show an algorithm for this problem with running time $f(k,r)n^{\mathcal{O}(1)}$. You could use the fact that we can find a maximum independent set in perfect graphs in polynomial time.

5.14. In the Pseudo Achromatic Number problem, we are given an undirected graph G and a positive integer k, and the goal is to check whether the vertices of G can be partitioned into k groups such that each pair of groups is connected by at least one edge. Obtain a randomized algorithm for Pseudo Achromatic Number running in time $2^{\mathcal{O}(k^2 \log k)}n^{\mathcal{O}(1)}$.

5.15. Consider a slightly different approach for Subgraph Isomorphism on graphs of bounded degree, where we randomly color vertices (instead of edges) with two colors.

1. Show that this approach leads to a $2^{(d+1)k}k!n^{\mathcal{O}(1)}$-time Monte Carlo algorithm with false negatives.
2. Improve the dependency on d in the running time of the algorithm to $d^{\mathcal{O}(k)}k!n^{\mathcal{O}(1)}$.

5.16. In the Split Edge Deletion problem, we are given an undirected graph G and a positive integer k, and the objective is to test whether there exists a set $S \subseteq E(G)$ of size at most k such that $G \setminus S$ is a split graph. Using chromatic coding show that the problem admits a $2^{\mathcal{O}(\sqrt{k}\log k)}n^{\mathcal{O}(1)}$-time algorithm.

5.17 (☠). Show an algorithm for the Split Edge Deletion problem (defined in the previous exercise) with running time $2^{\mathcal{O}(\sqrt{k})}n^{\mathcal{O}(1)}$.

5.18 (✎). Show that, for every $n, \ell \geq 2$, any $(n, 2, \ell)$-splitter needs to contain at least $\log_\ell n$ elements. In other words, show that the $\log n$ dependency in the size bound of Theorem 5.16 is optimal.

5.19 (✎). Give a deterministic version of Theorem 5.7 using (n,k)-universal sets.

5.20 (✎). Give a deterministic version of the first algorithm developed in Exercise 5.15. Your algorithm should run in time $2^{(d+1)k+o(dk)}k!n^{\mathcal{O}(1)}$.

5.21 (☠). Give a deterministic version of the second algorithm developed in Exercise 5.15. Your algorithm should run in time $2^{\mathcal{O}(d(\log d+\log k))}n^{\mathcal{O}(1)}$.

5.22 (✎). Show formally the running time bound of Theorem 5.24. That is, formulate and solve the corresponding variant of recurrence (5.3).

5.23 (✎). Complete the proof of Theorem 5.28: verify that the size of the obtained family $\mathcal{F}$ satisfies the promised bound, and that $\mathcal{F}$ is in fact a $(n, k, 2\lceil\sqrt{k}\rceil)$-coloring family.

Hints

5.2 The coloring step is exactly the same as in the Longest Path example: color the vertices of G with k colors, hoping to make the subgraph in question colorful; let V_i be the set of vertices colored i. Then, design a dynamic-programming algorithm to find a colorful subgraph isomorphic to T. To this end, root T at an arbitrary vertex and, for every $x \in V(T)$, denote by T_x the subtree of T rooted at x. By the dynamic-programming approach, for every $x \in V(T)$, for every $v \in V(G)$, and for every set S of exactly $|V(T_x)|-1$

colors, check whether there exists a colorful subgraph of $G[\bigcup_{i \in S} V_i \cup \{v\}]$ isomorphic to T_x, where the vertex v corresponds to the root x.

5.3 Randomly partition $V(G)$ into two parts and hope that the vertices of the solution X will be placed on the left, while a q-vertex connected subgraph of each of the two large components of $G \setminus X$ is placed on the right.

5.4 Use the same approach as in the previous problem, but play with probabilities: assign each vertex to the left with probability $1/q$, and to the right with the remaining probability. To derandomize the algorithm, use a $(n, 2q+k, (2q+k)^2)$-splitter.

5.8 Prove the following observation: if a graph G contains a cycle of length at least $2k$, then, after contracting an arbitrary edge, it still contains a cycle of length at least k. Use this, together with color coding, to get the algorithm.

5.9 Note that contracting a single edge may create a directed cycle even if there was none before contraction. Prove that contracting all of the edges of a directed cycle C cannot turn a no-instance into a yes-instance. Prove that if G has a cycle of length at least t before contracting C, then G has a cycle of length at least $t/|C|$ after contracting C. Use this, together with color coding, to get the algorithm.

To improve the running time from $2^{\mathcal{O}(k^2)}$ to $k^{\mathcal{O}(k)}$ observe that if G contains a cycle C such that the k first vertices of C are colored 1,2,. . .,k, and the remaining vertices are colored $k+1$, then we may find a cycle of length at least k in G in polynomial time.

5.10 Fix a solution $f : U \to \{1, 2\}$ and k sets $F_1, F_2, \ldots, F_k \in \mathcal{F}$ that are not monochromatic in the coloring f. For each set F_i, fix two elements $a_{i,1}$, $a_{i,2} \in F_i$ such that $f(a_{i,1}) \neq f(a_{i,2})$. Denote $A := \bigcup_{i=1}^{k} \{a_{i,1}, a_{i,2}\}$. Prove that $\Omega(2^{|A|-k})$ colorings $g : A \to \{1, 2\}$ satisfy $g(a_{i,1}) \neq g(a_{i,2})$ for every $1 \leq i \leq k$.

5.11 Perform the color coding step with t colors on the set $E(G)$. Then, a *colorful solution* is a set of at most k vertices that is adjacent to at least one edge of every color.

5.12 Proceed in a way similar to the solution of the previous problem. Perform the color coding step with t colors on the set $V(G)$. Then, a *colorful solution* is a set X of at most k vertices such that $N_G[X]$ contains at least one vertex of every color.

5.13 First, observe that, without loss of generality, we may look for an induced subgraph with exactly k vertices. Second, show that a random coloring of G into r colors correctly colors the subgraph in question with good probability. Finally, observe that, given a colored instance, it suffices to find a maximum size independent set in each color class independently.

5.14 Prove that a random coloring of $V(G)$ into k colors does the job with good probability.

5.15 Let $\widehat{H}$ be a subgraph of G isomorphic to H. A coloring $\chi : V(G) \to \{R, B\}$ is successful if $V(\widehat{H}) \subseteq \chi^{-1}(R)$ but $N_G(V(\widehat{H})) \subseteq \chi^{-1}(B)$. Observe that $|V(\widehat{H})| = k$ and $|N_G(V(\widehat{H}))| \leq dk$ by the degree bound. The first point follows by using a brute-force algorithm to check isomorphism in the construction of the graph $B(H, G_R)$ where $G_R = G[\chi^{-1}(R)]$.

For the second point, the key idea is to bias the probability: since our bound for $|V(\widehat{H})|$ is much better than the one for $|N_G(V(\widehat{H}))|$, we may color independently every vertex red with probability $1/d$ and blue with probability $1 - 1/d$. In this manner, χ is successful with probability:

$$\left(\frac{1}{d}\right)^{|V(\widehat{H})|} \cdot \left(1 - \frac{1}{d}\right)^{|N_G(V(\widehat{H}))|} \geq d^{-k} \cdot \left(1 - \frac{1}{d}\right)^{dk} = d^{-k} \cdot 2^{-\mathcal{O}(k)}.$$

5.16 Perform chromatic coding as in the case of d-Clustering. In the second phase, use the fact that every n-vertex split graph has only $\mathcal{O}(n)$ possible partitions of the vertex set into the clique and independent set parts. To obtain the $2^{\mathcal{O}(\sqrt{k}\log k)}n^{\mathcal{O}(1)}$ running time bound, you need to either obtain a polynomial kernel for the problem (which is done in Exercise 2.36), or wrap the algorithm with iterative compression.

5.17 The crucial observation is the following: every n-vertex graph on at most k edges has at most $2^{\mathcal{O}(\sqrt{k})}n$ subgraphs being complete graphs. To see this claim, consider the ordering $v_1, v_2, \ldots, v_n$ of $V(G)$ defined in the proof of Lemma 5.10. To identify a complete subgraph H of G, it suffices to give (a) a vertex $v_i \in V(H)$ of lowest possible index (n choices), and (b) the set $V(H) \setminus \{v_i\} = N_G(v_i) \cap V(G_i)$ ($2^{d_i} = 2^{\mathcal{O}(\sqrt{k})}$ choices, as v_i has degree d_i in $V(G_{i-1})$ and $d_i = \mathcal{O}(\sqrt{k})$).

To utilize the aforementioned observation, proceed with iterative compression, adding vertices one by one to the graph G. Assume we have a solution F to Split Edge Deletion on $(G - v, k)$ and we would like find a solution F' to Split Edge Deletion on (G, k). Furthermore, let $V(G) \setminus \{v\} = C \uplus I$ be (any) partition of the vertex set of the split graph $G - v - F$ into the clique and independent set parts. Observe that, if $V(G) = C' \uplus I'$ is (any) partition of the vertex set of the split graph $G - F'$ into the clique and independent set parts, then $I \setminus I'$ induces a clique in G. Moreover, $E(G[I]) \subseteq F$ and, consequently, $G[I]$ has at most k edges. Thus, there are $2^{\mathcal{O}(\sqrt{k})}n$ choices for $I \setminus I'$. Hence, we can guess $I \setminus I'$, as well as whether the new vertex v belongs to C' or I'. With this information, it is straightforward to deduce the remaining parts of C' and I'.

5.18 Observe that if $n > \ell^{|\mathcal{F}|}$ in a $(n, 2, \ell)$-splitter $\mathcal{F}$, then there are two elements $a, b \in [n]$ such that for every $f \in \mathcal{F}$ we have $f(a) = f(b)$.

5.19 Observe that you need the correct partition of $E(\widehat{H}) \cup \Gamma$, which is of size at most dk. Thus, you need a $(|E(G)|, p)$-universal set for every $k \le p \le dk$.

5.20 Observe that you need the correct partition of $N_G[V(\widehat{H})]$, which is of size at most $(d+1)k$. Thus, you need a $(|V(G)|, p)$-universal set for every $k \le p \le (d+1)k$.

5.21 The biased separation, used in the solution for Exercise 5.15, can be usually efficiently derandomized using a splitter from Theorem 5.16.

Recall that we need to correctly separate $N_G[V(\widehat{H})]$, which is of size at most $(d+1)k$. We first guess $p = |N_G[V(\widehat{H})]| \le (d+1)k$ and we construct a (n, p, p^2)-splitter $\mathcal{F}$ of polynomial size. Then, we iterate through every function $f \in \mathcal{F}$ and every set $X \subseteq [p^2]$ of size exactly k and consider a coloring $\chi : V(G) \to \{R, B\}$ defined as: $\chi(v) = R$ if and only if $f(v) \in X$. By the definition of a splitter, there exists $f \in \mathcal{F}$ that is injective on $N_G[V(\widehat{H})]$ and we consider a set $X = f(V(\widehat{H}))$. For this choice of f and X, the coloring χ is successful. Finally, note that the size of $\mathcal{F}$ is polynomial, whereas there are $\binom{p^2}{k} = 2^{\mathcal{O}(k(\log d + \log k))}$ choices for the set X.

Intuitively, in this approach we use a generic splitter whose size is negligible in the running time, and then we mimic the bias of the separation by guessing a small set in the (largish) codomain of the splitter.

Bibliographic notes

The randomized $4^k n^{\mathcal{O}(1)}$ algorithm for Feedback Vertex Set is due to Becker, Bar-Yehuda, and Geiger [29]. Longest Path was studied intensively within the parameterized complexity paradigm. Monien [368] and Bodlaender [44] showed that the problem is fixed-parameter tractable with running time $2^{\mathcal{O}(k\log k)}n^{\mathcal{O}(1)}$. This led Papadimitriou and Yannakakis [381] to conjecture that the problem is solvable in polynomial time for $k = \log n$. This conjecture was resolved in a seminal paper of Alon, Yuster and Zwick [15], who introduced the method of color coding and obtained the first algorithm with running time $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ for the problem. The divide and color algorithm decribed in this chapter was obtained independently by Kneis, Mölle, Richter and Rossmanith [299] and Chen, Lu, Sze and Zhang [86] (see also [83]). After that, there were several breakthrough ideas leading to faster and faster *randomized* algorithms based on algebraic tools, see Chapter 10. Theorem 5.24, i.e., the derandomization of divide and color using universal sets in time $\mathcal{O}(4^{k+o(k)}) n^{\mathcal{O}(1)}$ is due to Chen et al. [86] (see also [83]). The fastest deterministic algorithm so far, based on fast computation of representative sets, runs in time $2.619^k n^{\mathcal{O}(1)}$ and is presented in Chapter 12.

Longest Path is a special case of the Subgraph Isomorphism problem, where for a given n-vertex graph G and k-vertex graph F, the question is whether G contains a subgraph isomorphic to F. In addition to Longest Path, parameterized algorithms for two other variants of Subgraph Isomorphism, when F is a tree, and, more generally, when F is a graph of treewidth at most t, were studied in the literature. Alon, Yuster and Zwick [15] showed that Subgraph Isomorphism, when the treewidth of the pattern graph is bounded by t, is solvable in time $2^{\mathcal{O}(k)} n^{\mathcal{O}(t)}$. Cohen, Fomin, Gutin, Kim, Saurabh, and Yeo [96] gave a randomized algorithm that for an input digraph D decides in time $5.704^k n^{\mathcal{O}(1)}$ if D contains a given out-tree with k vertices. They also showed how to derandomize the algorithm in time $6.14^k n^{\mathcal{O}(1)}$. Amini, Fomin and Saurabh [16] introduced an inclusion–exclusion based approach in the classic color-coding and gave a randomized $5.4^k n^{\mathcal{O}(t)}$-time algorithm and a deterministic $5.4^{k+o(k)} n^{\mathcal{O}(t)}$ time algorithm for the case when F has treewidth at most t. Koutis and Williams [306] generalized their algebraic approach for Longest Path to Tree Subgraph Isomorphism and obtained a randomized algorithm running in time $2^k n^{\mathcal{O}(1)}$ for Tree Subgraph Isomorphism. This result was extended in [204] by a randomized algorithm for Subgraph Isomorphism running in time $2^k n^{\mathcal{O}(t)}$, when the treewidth of the pattern graph H is at most t. If we restrict ourselves to deterministic algorithms, the fastest known algorithm runs in time $2.851^k n^{\mathcal{O}(t)}$ for Subgraph Isomorphism if the treewidth of H is at most t [206].

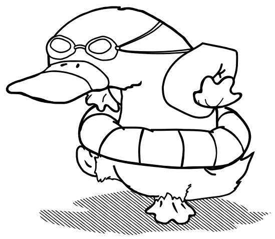
The random separation technique was introduced by Cai, Chan and Chan [65]. The $n^{\mathcal{O}(d \log d)}$-time algorithm for Graph Isomorphism in n-vertex graphs of maximum degree at most d is due to Luks [340].

Chromatic coding was introduced by Alon, Lokshtanov and Saurabh [14] for giving a subexponential algorithm for Feedback Arc Set in Tournaments. For algorithms with improved running time and related problems on tournaments see also [171, 209, 283, 186]. Recently, subexponential algorithms for several edge completion problems, like completing to split [228], interval [42], proper interval [43], chordal [211] or trivially perfect graphs [158] with at most k edges, were discovered. The fastest algorithm for d-Clustering runs in time $2^{\mathcal{O}(\sqrt{dk})} + n^{\mathcal{O}(1)}$ [199].

Most of the cited results for splitters, hash families and universal sets are due to Naor, Schulman and Srinivasan [372], with parts already present in the work of Alon, Yuster and Zwick [15]. In particular, the $\mathcal{O}(n \log n)$ dependency on n in the running time bounds follows from the combination of the (n, k, k^2)-splitter construction of [15] and further constructions of [372]. A similar result with worse size bounds can be obtained using the results of Fredman, Komlós and Szemerédi [216]. A generalization of the discussed derandomization objects, useful for designing approximate counting algorithms, was developed by Alon and Gutner [12, 13]. The construction of universal coloring families is taken from [14].

Chapter 6 Miscellaneous

In this chapter, we gather a few algorithmic tools that we feel are important, but do not fit into any of the previous chapters. First, we discuss exponential-time dynamic-programming algorithms that consider all subsets of a certain set when defining the subproblems. Second, we introduce the Integer Linear Programming Feasibility *problem, formulate the classical results on how to solve it in the case of a bounded number of variables, and show an example of its application in fixed-parameter algorithms. Finally, we discuss the algorithmic implications of the Robertson-Seymour theorem on graph minors.*



The chapter consists of three independent sections, each tackling a different tool in parameterized algorithms. Since a full exposition of, say, the algorithmic applications of the graph minors project would constitute a large volume in itself, we have decided to give only a glimpse of the topic in each of the three sections.

In Section 6.1, we present a common theme in many exponential-time algorithms: a dynamic-programming algorithm, where the subproblems are defined by considering all subsets of a certain set of elements (and hence the number of subproblems and the running time are exponential in the number of elements). As a first example, we give a simple $2^{|U|}(|U|+|\mathcal{F}|)^{\mathcal{O}(1)}$-time dynamic-programming algorithm for Set Cover with a family of sets $\mathcal{F}$ over a universe U. Then, we obtain a $3^{|K|}n^{\mathcal{O}(1)}$ algorithm for Steiner Tree, where K is the set of terminal vertices. Using additional ideas, we will improve the running time of this algorithm in Section 10.1.2.

Section 6.2 introduces yet another tool to design fixed-parameter algorithms, namely integer linear programs. In turns out that many NP-hard problems can be expressed in the language of Integer Linear Programming. For example, in Section 2.5 we have seen how a Vertex Cover instance can be encoded as an Integer Linear Programming instance.

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_6

In 1983, Lenstra showed that INTEGER LINEAR PROGRAMMING is fixed-parameter tractable when parameterized by the dimension of the space, i.e., the number of variables. (Naturally, Lenstra did not use the terminology of fixed-parameter tractability, as it was introduced much later.) Thus, Lenstra's result gives us a very general tool for proving fixed-parameter tractability of various problems. Note that a similar phenomenon happens in the world of polynomial-time algorithms, where a large number of tractable problems, including the problems of finding a shortest path or a maximum flow, can be represented in the language of LINEAR PROGRAMMING. In Section 6.2, we state the fastest known algorithm for INTEGER LINEAR PROGRAMMING, and exemplify its usage on the IMBALANCE problem.

In Section 6.3, we move to the theory of graph minors. The Graph Minors project of Robertson and Seymour, developed in the last three decades, resulted not only in achieving its main goal — proving Wagner's conjecture, asserting that the class of all graphs is well-quasi-ordered by the minor relation — but also flourished with other tools, techniques and insights that turned out to have plenty of algorithmic implications. In this chapter, we restrict ourselves only to the implications of the Robertson-Seymour theorem in parameterized complexity, and we show how theorems from Graph Minors immediately imply fixed-parameter tractability of such problems as detecting the Euler genus of a graph. Robertson-Seymour theory also gives us pretext to discuss the notion of *nonuniform* fixed-parameter algorithms. We remark here that the next chapter, Chapter 7, is entirely devoted to the algorithmic usage of the notion of treewidth, a different offspring of the Graph Minors project.

6.1 Dynamic programming over subsets

In this section, we give two examples of dynamic-programming algorithms over families of sets. Our examples are SET COVER and STEINER TREE.

6.1.1 SET COVER

Let $\mathcal{F}$ be a family of sets over a universe U. For a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ and a subset $U' \subseteq U$, we say that $\mathcal{F}'$ *covers* U' if every element of U' belongs to some set of $\mathcal{F}'$, that is, $U' \subseteq \bigcup \mathcal{F}'$. In the SET COVER problem, we are given a family of sets $\mathcal{F}$ over a universe U and a positive integer k, and the task is to check whether there exists a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ of size at most k such that $\mathcal{F}'$ covers U. We give an algorithm for SET COVER that runs in time $2^{|U|}(|U| + |\mathcal{F}|)^{\mathcal{O}(1)}$. In fact, this algorithm does not use the value of k, and finds the minimum possible cardinality of a family $\mathcal{F}' \subseteq \mathcal{F}$ that covers U.

Theorem 6.1. *Given a* Set Cover *instance* $(U, \mathcal{F}, k)$*, the minimum possible size of a subfamily* $\mathcal{F}' \subseteq \mathcal{F}$ *that covers* U *can be found in time* $2^{|U|}(|U| + |\mathcal{F}|)^{\mathcal{O}(1)}$*.*

Proof. Let $\mathcal{F} = \{F_1, F_2, \ldots, F_{|\mathcal{F}|}\}$. We define the dynamic-programming table as follows: for every subset $X \subseteq U$ and for every integer $0 \leq j \leq |\mathcal{F}|$, we define $T[X, j]$ as the minimum possible size of a subset $\mathcal{F}' \subseteq \{F_1, F_2, \ldots, F_j\}$ that covers X. (Henceforth, we call such a family $\mathcal{F}'$ a *valid candidate* for the entry $T[X, j]$.) If no such subset $\mathcal{F}'$ exists (i.e., if $X \not\subseteq \bigcup_{i=1}^{j} F_i$), then $T[X, j] = +\infty$.

In our dynamic-programming algorithm, we compute all $2^{|U|}(|\mathcal{F}| + 1)$ values $T[X, j]$. To achieve this goal, we need to show (a) base cases, in our case values $T[X, j]$ for $j = 0$; (b) recursive computations, in our case how to compute the value $T[X, j]$ knowing values $T[X', j']$ for $j' < j$.

For the base case, observe that $T[\emptyset, 0] = 0$ while $T[X, 0] = +\infty$ for $X \neq \emptyset$.

For the recursive computations, let $X \subseteq U$ and $0 < j \leq |\mathcal{F}|$; we show that

$$T[X, j] = \min(T[X, j-1], 1 + T[X \setminus F_j, j-1]). \tag{6.1}$$

We prove (6.1) by showing inequalities in both directions. In one direction, let $\mathcal{F}' \subseteq \{F_1, F_2, \ldots, F_j\}$ be a family of minimum size that covers X. We distinguish two cases. If $F_j \notin \mathcal{F}'$, then note that $\mathcal{F}'$ is also a valid candidate for the entry $T[X, j-1]$ (i.e., $\mathcal{F}' \subseteq \{F_1, F_2, \ldots, F_{j-1}\}$ and $\mathcal{F}'$ covers X). If $F_j \in \mathcal{F}'$, then $\mathcal{F}' \setminus \{F_j\}$ is a valid candidate for the entry $T[X \setminus F_j, j-1]$. In the other direction, observe that any valid candidate $\mathcal{F}'$ for the entry $T[X, j-1]$ is also a valid candidate for $T[X, j]$ and, moreover, for every valid candidate $\mathcal{F}'$ for $T[X \setminus F_j, j-1]$, the family $\mathcal{F}' \cup \{F_j\}$ is a valid candidate for $T[X, j]$. This finishes the proof of (6.1).

By using (6.1), we compute all values $T[X, j]$ for $X \subseteq U$ and $0 \leq j \leq |\mathcal{F}|$ within the promised time bound. Finally, observe that $T[U, |\mathcal{F}|]$ is the answer we are looking for: the minimum size of a family $\mathcal{F}' \subseteq \{F_1, F_2, \ldots, F_{|\mathcal{F}|}\} = \mathcal{F}$ that covers U. □

We remark that, although the dynamic-programming algorithm of Theorem 6.1 is very simple, we suspect that the exponential dependency on $|U|$, that is, the term $2^{|U|}$, is optimal. However, there is no known reduction that supports this claim with the Strong Exponential Time Hypothesis (discussed in Chapter 14).

6.1.2 Steiner Tree

Let G be an undirected graph on n vertices and $K \subseteq V(G)$ be a set of *terminals*. A *Steiner tree* for K in G is a connected subgraph H of G containing K, that is, $K \subseteq V(H)$. As we will always look for a Steiner tree of minimum possible size or weight, without loss of generality, we may assume that we focus only on subgraphs H of G that are trees. The vertices of $V(H) \setminus K$ are called *Steiner vertices* of H. In the (weighted) Steiner Tree problem, we are given an undirected graph G, a weight function $\mathbf{w}\colon E(G) \to \mathbb{R}_{>0}$ and a subset of terminals $K \subseteq V(G)$, and the goal is to find a Steiner tree H for K in G whose weight $\mathbf{w}(H) = \sum_{e\in E(H)} \mathbf{w}(e)$ is minimized. Observe that if the graph G is unweighted (i.e., $\mathbf{w}(e) = 1$ for every $e \in E(G)$), then we in fact optimize the number of edges of H, and we may equivalently optimize the number of Steiner vertices of H.

For a pair of vertices $u, v \in V(G)$, by $\mathrm{dist}(u, v)$ we denote the cost of a shortest path between u and v in G (i.e., a path of minimum total weight). Let us remind the reader that, for any two vertices u, v, the value $\mathrm{dist}(u, v)$ is computable in polynomial time, say by making use of Dijkstra's algorithm.

The goal of this section is to design a dynamic-programming algorithm for Steiner Tree with running time $3^{|K|}n^{\mathcal{O}(1)}$, where $n = |V(G)|$.

We first perform some *preprocessing steps*. First, assume $|K| > 1$, as otherwise the input instance is trivial. Second, without loss of generality, we may assume that G is connected: a Steiner tree for K exists in G only if all terminals of K belong to the same connected component of G and, if this is the case, then we may focus only on this particular connected component. This assumption ensures that, whenever we talk about some minimum weight Steiner tree or a shortest path, such a tree or path exists in G (i.e., we do not minimize over an empty set). Third, we may assume that each terminal in K is of degree exactly 1 in G and its sole neighbor is not a terminal. To achieve this property, for every terminal $t \in K$, we attach a new neighbor t' of degree 1, that is, we create a new vertex t' and an edge tt' of some fixed weight, say 1. Observe that, if $|K| > 1$, then the Steiner trees in the original graph are in one-to-one correspondence with the Steiner trees in the modified graphs.

We start with defining a table for dynamic programming. For every nonempty subset $D \subseteq K$ and every vertex $v \in V(G) \setminus K$, let $T[D, v]$ be the minimum possible weight of a Steiner tree for $D \cup \{v\}$ in G.

The intuitive idea is as follows: for every subset of terminals D, and for every vertex $v \in V(G) \setminus K$, we consider the possibility that in the optimal Steiner tree H for K, there is a subtree of H that contains D and is attached to the rest of the tree H through the vertex v. For $|D| > 1$, such a subtree decomposes into two smaller subtrees rooted at some vertex u (possibly $u = v$), and a shortest path between v and u. We are able to build such subtrees for larger and larger sets D through the dynamic-programming algorithm, filling up the table $T[D, v]$.

The base case for computing the values $T[D, v]$ is where $|D| = 1$. Observe that, if $D = \{t\}$, then a Steiner tree of minimum weight for $D \cup \{v\} = \{v, t\}$

is a shortest path between v and t in the graph G. Consequently, we can fill in $T[\{t\}, v] = \mathrm{dist}(t, v)$ for every $t \in K$ and $v \in V(G) \setminus K$.

In the next lemma, we show a recursive formula for computing the values $T[D, v]$ for larger sets D.

Lemma 6.2. *For every $D \subseteq K$ of size at least 2, and every $v \in V(G) \setminus K$, the following holds*

$$T[D, v] = \min_{\substack{u \in V(G) \setminus K \\ \emptyset \neq D' \subsetneq D}} \left\{ T[D', u] + T[D \setminus D', u] + \mathrm{dist}(v, u) \right\}. \tag{6.2}$$

Proof. We prove (6.2) by showing inequalities in both directions.

In one direction, fix $u \in V(G)$ and $\emptyset \neq D' \subsetneq D$. Let H_1 be the tree witnessing the value $T[D', u]$, that is, H_1 is a Steiner tree for $D' \cup \{u\}$ in G of minimum possible weight. Similarly, define H_2 for the value $T[D \setminus D', u]$. Moreover, let P be a shortest path between v and u in G. Observe that $H_1 \cup H_2 \cup P$ is a connected subgraph of G that contains $D \cup \{v\}$ and is of weight

$$\mathbf{w}(H_1 \cup H_2 \cup P) \leq \mathbf{w}(H_1) + \mathbf{w}(H_2) + \mathbf{w}(P) = T[D', u] + T[D \setminus D', u] + \mathrm{dist}(v, u).$$

Thus

$$T[D, v] \leq \min_{\substack{u \in V(G) \setminus K \\ \emptyset \neq D' \subsetneq D}} \left\{ T[D', u] + T[D \setminus D', u] + \mathrm{dist}(v, u) \right\}.$$

In the opposite direction, let H be a Steiner tree for $D \cup \{v\}$ in G of minimum possible weight. Let us root the tree H in the vertex v, and let u_0 be the vertex of H that has at least two children and, among such vertices, is closest to the root. An existence of such a vertex follows from the assumptions that $|D| \geq 2$ and that every terminal vertex is of degree 1. Moreover, since every terminal of K is of degree 1 in G, we have $u_0 \notin K$. Let u_1 be an arbitrarily chosen child of u_0 in the tree H. We decompose H into the following three edge-disjoint subgraphs:

1. P is the path between u_0 and v in H;
2. H_1 is the subtree of H rooted at u_1, together with the edge $u_0 u_1$;
3. H_2 consists of the remaining edges of H, that is, the entire subtree of H rooted at u_0, except for the descendants of u_1 (that are contained in H_1). See Fig. 6.1.

Let $D' = V(H_1) \cap K$ be the terminals in the tree H_1. Since every terminal is of degree 1 in G, we have $D \setminus D' = V(H_2) \cap K$. Observe that, as H is of minimum possible weight, $D' \neq \emptyset$, as otherwise $H \setminus H_1$ is a Steiner tree for $D \cup \{v\}$ in G. Similarly, we have $D' \subsetneq D$ as otherwise $H \setminus H_2$ is a Steiner tree for $D \cup \{v\}$ in G. Furthermore, note that from the optimality of H it follows that $\mathbf{w}(H_1) = T[D', u_0]$, $\mathbf{w}(H_2) = T[D \setminus D', u_0]$ and, moreover, P is a shortest path between u_0 and v. Consequently,

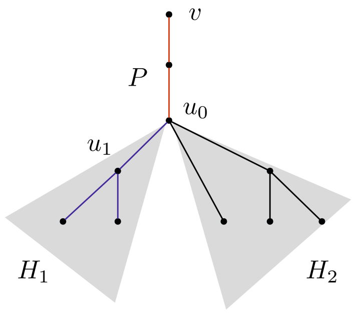


Fig. 6.1: Decomposition of H

$$\begin{aligned} T[D,v] = \mathbf{w}(H) &= T[D',u_0] + T[D \setminus D', u_0] + \mathrm{dist}(v,u_0) \\ &\geq \min_{\substack{u \in V(G)\setminus K \\ \emptyset \neq D' \subsetneq D}} \{T[D',u] + T[D \setminus D', u] + \mathrm{dist}(v,u)\}. \end{aligned}$$

This finishes the proof of the lemma. □

With the insight of Lemma 6.2, we can now prove the main result of this section.

Theorem 6.3. Steiner Tree *can be solved in time* $3^{|K|}n^{\mathcal{O}(1)}$.

Proof. Let (G,w,K) be an instance of Steiner Tree after the preprocessing steps have been performed. We compute all values of $T[D,v]$ in the increasing order of the cardinality of the set D. As discussed earlier, in the base case we have $T[\{t\},v] = \mathrm{dist}(t,v)$ for every $t \in K$ and $v \in V(G) \setminus K$. For larger sets D, we compute $T[D,v]$ using (6.2); note that in this formula we use values of $T[D',u]$ and $T[D \setminus D',u]$, and both D' and $D \setminus D'$ are proper subsets of D. In this manner, a fixed value $T[D,v]$ can be computed in time $2^{|D|}n^{\mathcal{O}(1)}$. Consequently, all values $T[D,v]$ are computed in time

$$\sum_{v \in V(G)\setminus K} \sum_{D \subseteq K} 2^{|D|} n^{\mathcal{O}(1)} \leq n \sum_{j=2}^{|K|} \binom{|K|}{j} 2^j n^{\mathcal{O}(1)} = 3^{|K|} n^{\mathcal{O}(1)}.$$

Finally, observe that, if the preprocessing steps have been performed, then any Steiner tree for K in $V(G)$ needs to contain at least one Steiner point and, consequently, the minimum possible weight of such a Steiner tree equals $\min_{v \in V(G)\setminus K} T[K,v]$. □

We will see in Section 10.1.2 how the result of Theorem 6.3 can be improved.

6.2 Integer Linear Programming

In this section, we take a closer look at parameterized complexity of Integer Linear Programming.

We start with some definitions. In the Integer Linear Programming Feasibility problem, the input consists of p variables $x_1, x_2, \ldots, x_p$, and a set of m inequalities of the following form:

$$\begin{array}{ccccc} a_{1,1}x_1 + a_{1,2}x_2 & + \ldots + & a_{1,p}x_p & \leq b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 & + \ldots + & a_{2,p}x_p & \leq b_2 \\ \vdots \quad\quad \vdots & \ddots & \vdots & \vdots \\ a_{m,1}x_1 + a_{m,2}x_2 & + \ldots + & a_{m,p}x_p & \leq b_m \end{array}$$

where every coefficient $a_{i,j}$ and b_i is required to be an integer. The task is to check whether one can choose *integer* values for every variable x_i so that all inequalities are satisfiable.

Equivalently, one may look at an Integer Linear Programming Feasibility instance as a matrix $A \in \mathbb{Z}^{m \times p}$ and a vector $b \in \mathbb{Z}^m$; the task is to check whether there exists a vector $x \in \mathbb{Z}^p$ such that $Ax \leq b$. We assume that an input of Integer Linear Programming Feasibility is given in binary and thus the size of the input is the number of bits in its binary representation.

In typical applications, when we want to solve a concrete algorithmic problem by formulating it as Integer Linear Programming Feasibility, we may assume that the absolute values of the variables in the solution are polynomially bounded by the size n of the instance we want to solve. Therefore, if the Integer Linear Programming Feasibility instance has p variables, then we may solve it in time $n^{\mathcal{O}(p)}$ by brute force. The main technical tool that we use in this section is the fact that Integer Linear Programming Feasibility is fixed-parameter tractable parameterized by the number of variables. This opens up the possibility of obtaining FPT results by translating the problem into an instance of Integer Linear Programming Feasibility with bounded number of variables.

Theorem 6.4 ([280],[319],[215]). *An* Integer Linear Programming Feasibility *instance of size L with p variables can be solved using $\mathcal{O}(p^{2.5p+o(p)} \cdot L)$ arithmetic operations and space polynomial in L.*

In other words, Theorem 6.4 says that Integer Linear Programming Feasibility is fixed-parameter tractable when parameterized by p, with a relatively good (slightly super-exponential) dependence on the parameter and linear dependence on the input size.

In some applications, it is more convenient to work with an *optimization* version of the Integer Linear Programming Feasibility problem, namely Integer Linear Programming. Although this problem has been already briefly discussed in Section 2.5, let us now recall its definition. In the Integer Linear Programming problem, apart from the standard input for Integer Linear Programming Feasibility (i.e., a matrix $A \in \mathbb{Z}^{m\times p}$ and a vector $b \in \mathbb{Z}^m$) we are given a vector $c \in \mathbb{Z}^p$, and our goal is to find a vector $x \in \mathbb{Z}^p$ satisfying all the aforementioned inequalities (i.e., $Ax \le b$) that *minimizes* the *objective function* $c \cdot x$ (the scalar product of c and x).

Using binary search, it is easy to derive an algorithm for Integer Linear Programming using Theorem 6.4.

Theorem 6.5. *An* Integer Linear Programming *instance of size L with p variables can be solved using*

$$\mathcal{O}(p^{2.5p+o(p)} \cdot (L + \log M_x) \log(M_x M_c))$$

arithmetic operations and space polynomial in $L + \log M_x$, where M_x is an upper bound on the absolute value a variable can take in a solution, and M_c is the largest absolute value of a coefficient in the vector c.

Proof. Observe that the absolute value of the objective function is at most pM_xM_c as long as the variables have absolute values at most M_x. We perform a binary search to find the minimum value of the objective function. That is, for a fixed integer threshold $-pM_xM_c \le t \le pM_xM_c$, we add an inequality $cx \le t$ to the system $Ax \le b$ and apply the algorithm of Theorem 6.4 to the obtained Integer Linear Programming Feasibility instance. The instance has size $\mathcal{O}(L + p\log(pM_xM_c)) = \mathcal{O}(p(L + \log M_x))$, and hence each application of Theorem 6.4 runs in time $\mathcal{O}(p^{2.5p+o(p)} \cdot (L + \log M_x))$. Consequently, we are able to find an optimum value t_0 of the objective function within the promised bound on the running time. Moreover, any solution to the Integer Linear Programming Feasibility instance consisting of the system $Ax \le b$ with an additional inequality $cx \le t_0$ is an optimal solution to the input Integer Linear Programming instance. □

6.2.1 The example of Imbalance

Now we exemplify the usage of Theorem 6.5 on the Imbalance problem.

In order to define the problem itself, we need to introduce some notation. Let G be an n-vertex undirected graph. An *ordering* of $V(G)$ is any bijective function $\pi\colon V(G) \to \{1, 2, \ldots, n\}$. For $v \in V(G)$, we define $L_\pi(v) = \{u \in N(v) : \pi(u) < \pi(v)\}$ and $R_\pi(v) = \{u \in N(v) : \pi(u) > \pi(v)\} = N(v) \setminus L_\pi(v)$. Thus $L_\pi(v)$ is the set of vertices preceding v, and $R_\pi(v)$ is the set of vertices succeeding v in π. The *imbalance at vertex v* is defined as $\iota_\pi(v) = ||L_\pi(v)| - |R_\pi(v)||$, and the *imbalance of the ordering π* equals $\iota(\pi) = \sum_{v\in V(G)} \iota_\pi(v)$.

In the Imbalance problem, parameterized by the vertex cover number of a graph, we are given a graph G together with its vertex cover X of size k, and the task is to find an ordering π of $V(G)$ minimizing its imbalance. The parameter is k, the size of the provided vertex cover, as opposed to — maybe more natural after most of the previous examples in this book — the parameterization by solution size (the value of the objective function, the imbalance of the ordering in question). We remark that there is an FPT algorithm for Imbalance parameterized by the imbalance of the ordering, but such an algorithm is not the goal of this section.

The Imbalance problem, parameterized by the vertex cover number, falls into the wide topic of so-called *structural parameterizations*. Here, instead of taking the most natural "solution size" parameterization (as was the case, e.g., for Vertex Cover or Feedback Vertex Set in previous chapters), we pick as a parameter some structural measure of the instance (most usually, a graph) at hand. In parameterized complexity, the choice of the parameter comes from the application: we would like to solve efficiently instances for which the parameter is small. Hence, studying a computational problem both using the "solution size" parameterization and different structural parameters, such as the vertex cover number, refines our understanding of the matter by considering complementary angles: we design efficient algorithms for different classes of input instances. Furthermore, from a theoretical point of view, different parameterizations often offer different interesting insights into the studied problem. This is yet another example of flexibility provided by parameterized complexity in choosing parameters.

We remark that this is not the only example of structural parameterization in this book. The entire Chapter 7 is devoted to *treewidth*, an important graph parameter that measures the resemblance of a graph to a tree. Moreover, in Section 15.2.4 we show an example of kernelization lower bound for Clique, parameterized by the vertex cover number.

We remark that in the case of parameterization by the vertex cover number, it is not necessary that we require that some vertex cover is provided along with the input graph, but instead we can compute a 2-approximation of a minimum vertex cover. In this manner, we obtain a parameter that is at most twice as large. Another approach would be to use an FPT algorithm to compute a vertex cover of size at most k. However, in the case of several other structural parameters of the input graph (e.g., the size of a dominating set) we do not have a good approximation or FPT algorithm to fall back on, and hence, in this book, we define all problems with structural parameters by insisting on providing the promised structure in the input.

We now show how to apply Theorem 6.5 in order to give an FPT algorithm for Imbalance parameterized by the size of the provided vertex cover of G.

We are looking for an ordering π for which $\iota(\pi)$ is minimized. In order to do this, we loop over all possible orderings (bijections) $\pi_X : X \to \{1, 2, \ldots, k\}$ of the vertex cover X and, for every such ordering π_X, we find the best ordering π of $V(G)$ that *agrees* with π_X: We say that π and π_X *agree* if for all $u, v \in X$

we have that $\pi_X(u) < \pi_X(v)$ if and only of $\pi(u) < \pi(v)$. In other words, the relative ordering π imposes on X is precisely π_X. Thus, at a cost of a factor of $k!$ in the running time, we can assume that there exists an optimal ordering π such that $X = \{u_1, u_2, \ldots, u_k\}$ and $\pi(u_1) < \pi(u_2) < \ldots < \pi(u_k)$. For every $0 \leq i \leq k$, we define $X_i = \{u_1, u_2, \ldots, u_i\}$.

Because X is a vertex cover, the set $I = V(G) \setminus X$ is independent. We associate a *type* with each vertex in I as follows.

Definition 6.6. The *type* of a vertex $v \in I$ is the set $N(v) \subseteq X$. For a type $S \subseteq X$, the set $I(S)$ is the set of all vertices in I of type S.

Notice that the number of different types does not exceed the number of subsets of X, which is 2^k.

Observe that every vertex of I is either mapped between two vertices of X, to the left of u_1, or to the right of u_k by an optimal ordering π. We say that a vertex $v \in I$ is at *location* 0 if $\pi(v) < \pi(u_1)$ and *at location* i if i is the largest integer such that $\pi(u_i) < \pi(v)$. The set of vertices that are at location i is denoted by L_i. We define the *inner order* of π at location i to be the restriction of π to L_i.

The task of finding an optimal permutation can be divided into two parts. The first part is to partition the set I into $L_0, \ldots, L_k$, while the second part consists of finding an optimal inner order at all locations. One should notice that partitioning I into $L_0, \ldots, L_k$ amounts to deciding *how many* vertices of each type are at location i for each i. Moreover, observe that the second part of the task, namely permuting the inner order of π at location i, in fact does not change the imbalance at any single vertex. This is due to the fact that the vertices of I, and thus all vertices at location i, are pairwise nonadjacent. Hence, the inner orders are in fact irrelevant and finding the optimal ordering of the vertices thus reduces to the first part of finding the right partition of I into $L_0, \ldots, L_k$. Our goal for the rest of this section is to phrase this task as an Integer Linear Programming instance.

For a type S and location i, we let x_S^i be a variable that encodes the number of vertices of type S that are at location i. Also, for every vertex u_i in X, we introduce a variable y_i that represents the lower bound on the imbalance of u_i.

Let us now describe the inequalities. First, in order to represent a feasible permutation, all the variables must be nonnegative. Second, the variables x_S^i have to be consistent with the fact that we have $|I(S)|$ vertices of type S, that is, $\sum_{i=0}^k x_S^i = |I(S)|$ for every type S.

For every vertex u_i of the vertex cover X, we let $e_i = |N(u_i) \cap X_{i-1}| - |N(u_i) \cap (X \setminus X_i)|$. For every $u_i \in X$ we have a constraint

$$y_i \geq \left| e_i + \sum_{\substack{S \subseteq X \\ u_i \in S}} \left(\sum_{j=0}^{i-1} x_S^j - \sum_{j=i}^{k} x_S^j \right) \right| . \tag{6.3}$$

Finally, for every type S and location i, let the constant z_S^i be equal to the imbalance at a vertex of type S if it is placed at location i. That is,

$$z_S^i = \left||S \cap X_i| - |S \cap (X \setminus X_i)|\right|.$$

We are now ready to formulate our Integer Linear Programming instance.

$$\begin{aligned}
\min \ & \sum_{i=1}^{k} y_i + \sum_{i=0}^{k} \sum_{S \subseteq X} z_S^i x_S^i & \\
\text{s.t.} \ & \sum_{i=0}^{k} x_S^i = |I(S)| & \forall S \subseteq X \\
& y_i \geq e_i + \sum_{\substack{S \subseteq X \\ u_i \in S}} \left(\sum_{j=0}^{i-1} x_S^j - \sum_{j=i}^{k} x_S^j \right) & \forall 1 \leq i \leq k \\
& y_i \geq -e_i - \sum_{\substack{S \subseteq X \\ u_i \in S}} \left(\sum_{j=0}^{i-1} x_S^j - \sum_{j=i}^{k} x_S^j \right) & \forall 1 \leq i \leq k \\
& x_S^i \geq 0 & \forall 0 \leq i \leq k, S \subseteq X.
\end{aligned}$$

A few remarks are in place. First, since (6.3) is not a linear constraint, we have represented it as two constraints in the Integer Linear Programming instance.

Second, formally, in the integer linear program above, we do not require y_i to be *exactly* the imbalance at u_i, but to be *at least* this imbalance. However, the minimization criterion forces y_i to be actually equal to the imbalance at u_i.

Finally, observe that the maximum possible value of a variable x_S^i is less than n, the maximum possible value of a variable y_i in an optimal solution (where the inequality (6.3) is in fact an equality) is less than n, and the maximum absolute value of a coefficient z_S^i is, again, less than n. Consequently, an application of Theorem 6.5 on our Integer Linear Programming instance runs in $2^{2^{\mathcal{O}(k)}} n^{\mathcal{O}(1)}$ time, and we obtain the following result.

Theorem 6.7. *The* Imbalance *problem, parameterized by the size of a provided vertex cover of the input graph, is fixed-parameter tractable.*

6.3 Graph minors and the Robertson-Seymour theorem

In this section, we discuss the minor relation in graphs and the algorithmic usage of the Robertson-Seymour theorem.

For a graph G and an edge $uv \in G$, we define the operation of *contracting edge uv* as follows: we delete vertices u and v from G, and add a new vertex w_{uv} adjacent to $(N_G(u) \cup N_G(v)) \setminus \{u, v\}$ (i.e., to all vertices that u or v was adjacent to in G). Observe that contraction defined as above does not introduce any multiple edges or loops (note that we used a different definition when treating Feedback Vertex Set in Section 3.3). Let G/uv be the graph obtained by contracting edge uv in G.

We say that a graph H is a *minor* of G, denoted by $H \leq_m G$, if H can be obtained from some subgraph of G by a series of edge contractions. Equivalently, we may say that H is a minor of G if H can be obtained from G itself by a series of edge deletions, edge contractions and vertex deletions.

There is also a different definition that in some cases is more convenient: H is a minor of G if for every $h \in V(H)$ we can assign a nonempty *branch set* $V_h \subseteq V(G)$, such that

(a) $G[V_h]$ is connected;
(b) for different $g, h \in V(H)$, the branch sets V_g and V_h are disjoint; and
(c) for every $gh \in E(H)$ there exists an edge $v_g v_h \in E(G)$ such that $v_g \in V_g$ and $v_h \in V_h$.

Such a family $(V_h)_{h \in V(H)}$ of branch sets is called a *minor model of H in G*. Exercise 6.12 asks the reader to check the equivalence of the aforementioned definitions.

The minor relation in some sense preserves the topological properties of a graph, for example, whether a graph is a planar graph.

Let us now clarify what we mean by a *planar graph*, or a graph embedded into the plane. First, instead of embedding into the plane we will equivalently embed our graphs into a sphere: in this manner, we do not distinguish unnecessarily the outer face of the embedding. Formally, an *embedding* of a graph G into a sphere is a mapping that maps (a) injectively each vertex of G into a point of the sphere, and (b) each edge uv of G into a Jordan curve connecting the images of u and v, such that the curves are pairwise disjoint (except for the endpoints) and do not pass through any other image of a vertex. A *face* is a connected component of the complement of the image of G in the sphere; if G is connected, each face is homeomorphic to an open disc. A *planar graph* is a graph that admits an embedding into a sphere, and a *plane graph* is a planar graph together with one fixed embedding.

It is easy to observe the following.

Proposition 6.8. *A minor of a planar graph is also planar.*

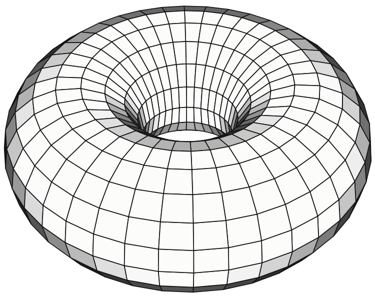


Fig. 6.2: Torus

Indeed, removing edges and vertices surely cannot make a planar graph non-planar and it is not difficult to modify a planar embedding to express the contraction of an edge.

A statement similar to Proposition 6.8 holds when we generalize planar graphs to graphs drawn on some other, more complicated, but fixed surface. In this chapter, we discuss graphs drawn on surfaces only very briefly and informally, as they only serve here as a demonstration of certain concepts in the theory of graph minors. It is well known that a graph can be drawn on the plane if and only if it can be drawn on the sphere. However, there are graphs that can be drawn on the torus (the doughnut-shaped surfaced shown on Fig. 6.2), but not on the sphere — the clique K_5 is such a graph. One way to construct surfaces is to add some number of "handles" to a sphere: for example, we may consider the torus as a sphere with a handle attached to it. There are other, more weird surfaces, such as the Klein bottle or the projective plane, that cannot be obtained this way and where we need to introduce an orientation-changing construction called a *cross-cap*, along with handles. We do not go into the exact definitions and details of these constructions, all we want to state here is that if a graph G is embeddable in a surface Σ, and H is a minor of G, then H is embeddable on Σ as well. For readers interested in Topological Graph Theory, we provide references to sources in the bibliographic notes at the end of this chapter.

It will be very important for some of the applications that certain structural parameters are monotone with respect to taking minors: for example, the minimum size of a vertex cover or feedback vertex set is not increased when taking the minor of a graph. The proof of the following proposition is left as an exercise (Exercise 6.13).

Proposition 6.9. *Let G be a graph, let $X \subseteq V(G)$ be a vertex cover (feedback vertex set) of G, and let H be a minor of G. Then there exists a vertex cover (feedback vertex set) of H of size at most $|X|$.*

Another example of a graph property that cannot increase if we go to a minor of a graph is treewidth, discussed in Chapter 7.

Although the minor relation is already interesting in itself, the main thrust to its usability is given by the following seminal result of Robertson and Seymour.

Theorem 6.10 (Robertson and Seymour). *The class of all graphs is well-quasi-ordered by the minor relation. That is, in any infinite family of graphs, there are two graphs such that one is a minor of the other.*

The statement of the Robertson-Seymour theorem above is sometimes referred to in the literature as *the Wagner's conjecture* or *the graph minors theorem*. In what follows, we stick to the name *Robertson-Seymour theorem*. As we shall see in the rest of this section, the Robertson-Seymour theorem is a very powerful tool in parameterized complexity.

Consider a class of graphs $\mathcal{G}$. We say that $\mathcal{G}$ is *closed under taking minors* or *minor-closed* if for every $G \in \mathcal{G}$ and every minor H of G, the graph H belongs to $\mathcal{G}$ as well. For example, by Proposition 6.8, the class of planar graphs, or more generally, graphs embeddable on a surface of fixed genus g, is minor-closed. Proposition 6.9 asserts that, for every fixed k, the class of graphs admitting a vertex cover of size at most k or a feedback vertex set of size k, is minor-closed.

We observe now the following crucial corollary of the Robertson-Seymour theorem.

Corollary 6.11. *For every minor-closed graph class $\mathcal{G}$, there exists a* finite *set* $\mathrm{Forb}(\mathcal{G})$ *of graphs with the following property: for every graph G, graph G belongs to $\mathcal{G}$ if and only if there does not exist a minor of G that is isomorphic to a member of* $\mathrm{Forb}(\mathcal{G})$.

Proof. Define $\mathrm{Forb}(\mathcal{G})$ to be the set of minor-minimal elements of the complement of $\mathcal{G}$: a graph G is in $\mathrm{Forb}(\mathcal{G})$ if G is *not* in $\mathcal{G}$, but every proper minor of G is in $\mathcal{G}$. We keep only one representative of every isomorphism class. By definition, $\mathrm{Forb}(\mathcal{G})$ has the desired property; it remains to check only its finiteness. However, if $\mathrm{Forb}(\mathcal{G})$ is infinite, then the Robertson-Seymour theorem implies that there exists $H, G \in \mathrm{Forb}(\mathcal{G})$ such that $H \leq_m G$. However, then G is not minimal with respect to the minor relation, contradicting the definition of $\mathrm{Forb}(\mathcal{G})$. $\square$

The family $\mathrm{Forb}(\mathcal{G})$ is often called *the family of minimal forbidden minors for $\mathcal{G}$*.

For example, by Wagner's theorem, a graph is planar if and only if it does not contain K_5 and $K_{3,3}$ as a minor. Thus for planar graphs Corollary 6.11 is not so interesting. However, already for the class of graphs embeddable in a torus (see Fig 6.2) the situation is not that clear. By Corollary 6.11, we know that the family of forbidden minors for this class is finite, but their number can be enormous and it is not clear how we can find them efficiently.

Currently, more than 16,000 graphs are known from the family of forbidden minors for the class of graphs embeddable in the torus. More generally, for graphs embeddable on a surface of genus g, i.e., a sphere with g handles, we only know that the number of forbidden minimal minors for this class of graphs is bounded by some function of g. Similarly, by Corollary 6.11, for every fixed k, the class of graphs admitting a vertex cover of size at most k has a forbidden family of minimal minors of size $f(k)$ for some function f.

It is important to note here that the proof of Corollary 6.11 is nonconstructive, i.e., while we know that set $\mathrm{Forb}(\mathcal{G})$ is finite, we do not know how to construct it and we do not know how to estimate its size.

We need one more algorithmic result of the Graph Minors project.

Theorem 6.12 (Robertson and Seymour). *There exists a computable function f and an algorithm that, for given graphs H and G, checks in time $f(H)|V(G)|^3$ whether $H \leq_m G$.*

Corollary 6.11 with Theorem 6.12 implies that every minor-closed class can be recognized in polynomial time.

Theorem 6.13. *Let $\mathcal{G}$ be a minor-closed graph class. There is a constant $c_{\mathcal{G}}$ depending on the class $\mathcal{G}$ only, such that for any n-vertex graph G, deciding whether $G \in \mathcal{G}$ can be done in time $c_{\mathcal{G}} \cdot n^3$.*

Proof. By Corollary 6.11, there exists a family $\mathrm{Forb}(\mathcal{G})$ that characterizes $\mathcal{G}$ and whose size and the sizes of its elements are at most some constant $c'_{\mathcal{G}}$ depending only on $\mathcal{G}$. Consider the following algorithm: iterate through all graphs $H \in \mathrm{Forb}(\mathcal{G})$ and use Theorem 6.12 to check if $H \leq_m G$. If this is the case for at least one graph $H \in \mathrm{Forb}(\mathcal{G})$, then $G \notin \mathcal{G}$ and we answer that $G \notin \mathcal{G}$. Otherwise, by the properties of $\mathrm{Forb}(\mathcal{G})$, we have that $G \in \mathcal{G}$.

Let us bound the running time of the algorithm. By Theorem 6.12, for every $H \in \mathrm{Forb}(\mathcal{G})$, we check whether H is a minor of G in time $f(H) \cdot n^3 \leq f(c'_{\mathcal{G}}) \cdot n^3$. The size of $\mathrm{Forb}(\mathcal{G})$ does not exceed $c'_{\mathcal{G}}$, and by putting $c_{\mathcal{G}} = c'_{\mathcal{G}} \cdot f(c'_{\mathcal{G}})$, we conclude the proof of the theorem. $\square$

Let us now apply Theorem 6.13 to Vertex Cover. Let (G, k) be a Vertex Cover instance. By Proposition 6.9, the class $\mathcal{G}_k$ of graphs admitting a vertex cover of size at most k is minor-closed. Thus by Theorem 6.13, for every $k \geq 0$, there is an algorithm deciding if a graph G is in $\mathcal{G}_k$ in time $c_k \cdot n^3$, where constant c_k depends only $\mathcal{G}_k$, that is $c_k = f(k)$ for some function f. Have we just shown another proof that Vertex Cover is fixed-parameter tractable?

Well, not exactly. Given only the integer k, we do not know the family $\mathrm{Forb}(\mathcal{G}_k)$ — Corollary 6.11 only asserts its existence and finiteness — and the knowledge of this family or at least its size is essential for further steps. In general, it seems hard to get around the problem of obtaining the forbidden minors: for example, it was shown by Fellows and Langston in [184] that, given only oracle access to a membership test for a minor-closed class $\mathcal{G}$, one cannot compute Forb($\mathcal{G}$). However, we have at least proved the following statement, which is somewhat weaker than fixed-parameter tractability. It should be noted that we obtained this result without any algorithmic insight into the problem: we only used the general tool Theorem 6.13 and the fact that the vertex cover number does not increase when taking minors.

Corollary 6.14. *For every k, there is a constant c_k and an algorithm that, given an n-vertex graph G, checks in time $c_k \cdot n^3$ if G admits a vertex cover of size at most k.*

Corollary 6.14 asserts the existence of a family of algorithms, one for every k, and the forbidden minors Forb($\mathcal{G}_k$) are hard-coded into the algorithm for the parameter k. Such a conclusion as in Corollary 6.14 is called *nonuniform fixed-parameter tractability.* .

Definition 6.15. We say that a parameterized problem Q is *nonuniformly fixed-parameter tractable* if there exists a constant α, a function $f : \mathbb{N} \to \mathbb{N}$, and a collection of algorithms $(\mathcal{A}_k)_{k\in\mathbb{N}}$ such that the following holds. For every $k \in \mathbb{N}$ and every input x, the algorithm $\mathcal{A}_k$ accepts input x if and only if (x, k) is a yes-instance of Q, and the running time of $\mathcal{A}_k$ on x is at most $f(k)|x|^\alpha$.

The above notion should be contrasted with our definition of fixed-parameter tractability, where we require that there exists a *single* algorithm that takes k on input and works for all values of k. To emphasize the difference, our "standard" FPT algorithms are sometimes called *(strongly) uniform.*

Let us now generalize Corollary 6.14 to a wider set of problems. Let $\mathcal{G}$ be a graph class. Many interesting problems can be defined as special cases of the following generic $\mathcal{G}$ Vertex Deletion problem: for a given graph G and integer k, does there exist a set X of at most k vertices of G such that $G - X \in \mathcal{G}$? For example, when $\mathcal{G}$ is the class of graphs without edges, $\mathcal{G}$ Vertex Deletion is Vertex Cover. When $\mathcal{G}$ is the class of forests, $\mathcal{G}$ Vertex Deletion is Feedback Vertex Set, and when $\mathcal{G}$ is planar, $\mathcal{G}$ Vertex Deletion is Planar Vertex Deletion, that is, the problem of whether a given graph G can be turned into a planar graph by deleting at most k vertices (we study this problem in detail in Section *7.8).

From a graph-theoretical point of view, the following formulation seems more natural. For a graph class $\mathcal{G}$ and an integer k, let $\mathcal{G} + k\mathrm{v}$ be a class of graphs defined as follows:

$$G \in \mathcal{G} + k\mathrm{v} \quad \text{if and only if} \quad \exists X \subseteq V(G) : (|X| \leq k) \wedge (G - X \in \mathcal{G}).$$

That is, $\mathcal{G} + k\mathrm{v}$ are exactly these graphs G for which (G, k) is a yes-instance to $\mathcal{G}$ Vertex Deletion. For this reason, the $\mathcal{G}$ Vertex Deletion problem is sometimes called $\mathcal{G} + k\mathrm{v}$ Recognition.

Proposition 6.9 generalizes to the following statement (see Exercise 6.14):

Proposition 6.16. *For every minor-closed graph class $\mathcal{G}$ and for every integer k, the class $\mathcal{G} + k\mathrm{v}$ is also minor-closed.*

Let us note that the only property of Vertex Cover we have used to derive Corollary 6.14 is that the class of graphs with vertex cover at most k is minor-closed. Thus we have the following theorem.

Theorem 6.17. *For every minor-closed class $\mathcal{G}$, the problem $\mathcal{G}$* Vertex Deletion *is nonuniformly fixed-parameter tractable, when parameterized by k.*

Although the notion of nonuniform fixed-parameter tractability evidently captures more problems than the standard, uniform one,[1] we believe that there is no significant difference among the "interesting" problems. In particular, it is very likely that the theory of fixed-parameter intractability and W[1]-hardness described in Chapter 13 excludes also nonuniform FPT algorithms. That is, our conjecture is that no W[1]-hard problem is nonuniformly fixed-parameter tractable.

Thus while Theorem 6.13 does not bring us to an FPT algorithm, it is very convenient to use it to make an "educated guess" that a problem is FPT.

In many cases, after using Theorem 6.13 to establish that a problem is nonuniformly FPT and hence very likely to be uniformly FPT as well, we can use other, more concrete techniques to prove that the problem is indeed uniformly FPT. For example, Vertex Cover and Feedback Vertex Set are special cases of $\mathcal{G}$ Vertex Deletion and thus by Theorem 6.17 are nonuniformly FPT. On the other hand, we already saw FPT algorithms for these problems in Chapter 3. For Planar Vertex Deletion, we give an FPT algorithm in Section *7.8. Another example is the Graph Genus problem: for a given graph G and integer k decide whether G can be embedded in a surface of genus k, i.e. a surface with k handles. The class of graphs of genus at most k is minor-closed and thus Graph Genus is nonuniformly fixed-parameter tractable. An explicit FPT algorithm for Graph Genus has been given by Mohar [365]. There are also some general techniques for making the algorithms resulting from Theorem 6.13 uniform, but they work only for a limited set of problems [183, 5].

In Exercises 6.16 and 6.17, we give two other examples of usages of the Robertson-Seymour theorem: Planar Diameter Improvement (does graph G have a planar supergraph with diameter at most k?) and Linkless Embedding (does graph G have an embedding in three-dimensional space

[1] The standard examples for P vs. P / poly apply here: any undecidable unary language, parameterized by the length of the input, does not admit a (uniform) FPT algorithm (since it is undecidable), but there is a trivial nonuniform one, as every algorithm $\mathcal{A}_k$ is supposed to handle only one input.

such that at most k cycles can link pairwise?). In both cases, the existence of a *uniform* FPT algorithm remains open.

Exercises

6.1 (✎). Using dynamic programming over the subsets, obtain an algorithm for Chromatic Number on n-vertex graphs running in time $3^n n^{\mathcal{O}(1)}$.

6.2 (✎). Using dynamic-programming over subsets, show that the Hamiltonian Cycle problem on an n-vertex graph can be solved in time $2^n n^{\mathcal{O}(1)}$.

6.3. For an $n \times n$ matrix $A = (a_{i,j})_{1 \le i,j \le n}$, the *permanent* of A is the value $\mathrm{perm}(A) = \sum_\sigma \prod_{i=1}^n a_{i,\sigma(i)}$, where the sum extends over all permutations σ of $\{1, 2, \ldots, n\}$. (This is very similar to the definition of the determinant, but we do not have here the 1 or -1 factor depending on the number of inversions in the permutation.) Using dynamic programming over subsets, show how to compute the permanent of a given $n \times n$ matrix in time $2^n n^{\mathcal{O}(1)}$.

6.4. Using dynamic programming over subsets, show that Directed Feedback Arc Set on n-vertex graphs can be solved in time $2^n n^{\mathcal{O}(1)}$.

6.5 (☠). Let G be an undirected bipartite graph on n vertices. Show that we can find a minimum size dominating set in G in time $2^{n/2} n^{\mathcal{O}(1)}$.

6.6 (☠). Show that Connected Vertex Cover admits an algorithm with running time $6^k n^{\mathcal{O}(1)}$. You might need to use an algorithm for Steiner Tree of Theorem 6.3.

6.7. Given a directed graph G, a set of terminals $K \subseteq V(G)$ and a root $r \in V(G)$, Directed Steiner Tree asks for a directed tree rooted at r such that every terminal in K is reachable from r on the tree. Obtain a $3^{|K|} n^{\mathcal{O}(1)}$-time algorithm for Directed Steiner Tree.

6.8 (☠). Consider the following restricted variant of Steiner Tree: assume G is a plane graph, and all terminals lie on the infinite face of G. Show how to enhance the dynamic-programming algorithm of Theorem 6.3 to run in polynomial time in this restricted case.

6.9 (☠). Improve the algorithm of Theorem 6.3 so that the factor in the running time bound that depends on the size of the input graph equals the running time of a single-source shortest-path algorithm. That is, obtain a $3^{|K|}|K|^{\mathcal{O}(1)}(n+m)$-time algorithm for unweighted graphs and $3^{|K|}|K|^{\mathcal{O}(1)}(n \log n + m)$-time for the general, weighted case.

In the next two exercises we revisit the Closest String problem, which was already considered in Chapter 3. Recall that in this problem we are given a set of k strings $x_1, x_2, \ldots, x_k$ over alphabet Σ, each of length L, and an integer d. The task is to find a string y of length L such that the Hamming distance between y and x_i is at most d, for every $1 \le i \le k$. In Section 3.5 we designed an FPT algorithm for this problem parameterized by d, while in Exercise 3.25 you were asked to design an algorithm that is faster when the alphabet size is small. This time, we consider the parameterization by k, the number of strings.

6.10 (☠). Prove that Closest String is fixed-parameter tractable when parameterized by k and $|\Sigma|$.

6.11 (☠). Refine the solution of the previous exercise to show that Closest String is fixed-parameter tractable when parameterized by k only.

6.12 (✎). Prove that the two definitions of the minor relation, mentioned in Section 6.3, are equivalent.

6.13 (✎). Show that vertex cover and feedback vertex set are minor-closed parameters. In other words, for any minor H of graph G, if G admits a vertex cover (feedback vertex set) of size at most k, then H admits a vertex cover (feedback vertex set) of size at most k as well.

6.14. Prove Proposition 6.16.

6.15 (✎). In the Max Leaf Subtree problem, we are given a graph G together with an integer k and the question is whether there is a subtree T of G with at least k leaves.

1. Show that (G, k) is a yes-instance if and only if $K_{1,k}$ (a graph with a center vertex connected to k degree-one vertices) is a minor of G.
2. Deduce that Max Leaf Subtree is nonuniformly fixed-parameter tractable, when parameterized by k.

6.16. In the Planar Diameter Improvement problem, the input consists of a planar graph G and an integer k, and the task is to check if there exists a supergraph of G that is still planar, and at the same time has diameter at most k.

1. Prove that this problem is nonuniformly fixed-parameter tractable, when parameterized by k.
2. Show that it suffices to consider supergraphs only G' of G with $V(G') = V(G)$; that is, it only makes sense to add edges to G, and adding new vertices does not help.

6.17. Let G be a graph embedded into $\mathbb{R}^3$ (that is, every vertex of G corresponds to some point in $\mathbb{R}^3$, and every edge of G corresponds to some sufficiently regular curve connecting its endpoints; the edges/vertices do not intersect unless it is imposed by the definition). Two vertex-disjoint cycles C_1 and C_2 are said to be *linked* in this embedding if the corresponding closed curves in $\mathbb{R}^3$ cannot be separated by a continuous deformation (i.e., they look like two consecutive links of a chain). A family $\mathcal{C}$ of pairwise vertex-disjoint cycles is *pairwise linked* if every two distinct cycles from the family are linked. In the Linkless Embedding problem, given a graph G and an integer k, we ask whether there exists an embedding of G into $\mathbb{R}^3$ such that any pairwise linked family of cycles in G has size at most k. Prove that this problem is nonuniformly fixed-parameter tractable, when parameterized by k.

6.18. In the Face Cover problem, we are given a planar graph G and an integer k, and the task is to check if there exists a planar embedding of G and a set of at most k faces in this embedding, such that every vertex of G lies on one of the chosen faces. Prove that this problem is nonuniformly fixed-parameter tractable, when parameterized by k.

6.19. In the Cycle Packing problem the input is a graph G with an integer k and the task is to determine whether there exist k cycles in G that are pairwise vertex disjoint. Prove that this problem is nonuniformly fixed-parameter tractable, when parameterized by k.

Hints

6.1 In the dynamic-programming table, for every $X \subseteq V(G)$, define $T[X]$ to be the minimum possible number of colors needed to color $G[X]$. For the base case, observe that $T[X] = 1$ if and only if X is a nonempty independent set in G.

6.2 Fix a vertex $s \in V(G)$. In the dynamic-programming table, for every set $X \subseteq V(G)$ that contains s, and every $t \in V(G) \setminus X$, define $T[X, t]$ to be a Boolean value indicating whether there exists in G a simple path with endpoints s and t and vertex set $X \cup \{t\}$.

6.3 In the dynamic-programming table, for every $X \subseteq \{1, 2, \ldots, n\}$, define $T[X]$ to be the permanent of the $|X| \times |X|$ matrix $(a_{i,j})_{1 \leq i \leq |X|, j \in X}$.

6.4 In the dynamic-programming table, for every $X \subseteq V(G)$, define $T[X]$ to be the minimum possible number of edges of $G[X]$ whose deletion makes $G[X]$ acyclic. For a recursive formula, for fixed set X, consider all possible choices of a vertex $v \in X$ that is the last vertex in the topological ordering of $G[X]$ after the solution edges have been deleted.

6.5 Let A and B be the bipartition sides of the input bipartite graph G and without loss of generality assume $|A| \leq |B|$, in particular, $|A| \leq n/2$. In general, perform a dynamic-programming algorithm like that of Theorem 6.1 for Set Cover instance with universe A and set family $\{N_G(b) \ : \ b \in B\}$. However, to ensure that the side B is dominated, and at the same time allow some vertices of A to be chosen into a dominating set, change the base case as follows. Let D be a minimum size dominating set in G and observe that $B \setminus N_G(D \cap A) \subseteq D$. For every possible guess of $D_A \subseteq A$ for the set $D \cap A$, we would like to allow in the dynamic-programming routine an option of taking $D_A \cup (B \setminus N_G(D_A))$ into the solution; from the point of view of the Set Cover instance with universe A, this translates to covering $D_A \cup N_G(B \setminus N_G(D_A))$ at cost $|D_A| + |B \setminus N_G(D_A)|$. To achieve this goal, we compute the values $T[X, 0]$ as follows.

1. First set $T[X, 0] = +\infty$ for every $X \subseteq A$.
2. For every $D_A \subseteq A$, define $X = D_A \cup N_G(B \setminus N_G(D_A))$ and set $T[X, 0] := \min(T[X, 0], |D_A| + |B \setminus N_G(D_A)|)$.
3. For every $X \subseteq A$ in the *decreasing* order of the size of X, and for every $v \in A \setminus X$, set $T[X, 0] := \min(T[X, 0], T[X \cup \{v\}, 0])$.

In this manner, $T[X, 0]$ equals a minimum possible size of a set $D_A \cup (B \setminus N_G(D_A))$ that dominates X, over all $D_A \subseteq A$.

6.6 Observe that Connected Vertex Cover can be seen as a two-stage problem: first choose some vertex cover K of the input graph G, of size at most k, and then connect K using minimum size Steiner tree for K. Furthermore, note that the straightforward branching algorithm for Vertex Cover runs in $2^k n^{\mathcal{O}(1)}$ time and in fact outputs all inclusion-wise minimal vertex covers of G of size at most k.

6.8 The main observation is as follows: it suffices to consider only sets $D \subseteq K$ that consist of a number of terminals that appear *consecutively* on the infinite face of the input graph. In this way, there are only $\mathcal{O}(|K|^2)$ sets D to consider.

6.9 The main idea is to perform the computations in a different order: for a fixed choice of sets D' and D, we would like to compute

$$T'[D, D', v] = \min_{w \in V(G) \setminus K} T[D', w] + T[D \setminus D', w] + \mathrm{dist}_G(v, w).$$

for all vertices $v \in V(G) \setminus K$ using a single run of a single source shortest path algorithm.

To this end, create an auxiliary graph G' as follows: create a new vertex s and for every $w \in V(G) \setminus K$ create an edge sw of weight $T[D', w] + T[D \setminus D', w]$. Observe that for every $v \in V(G)$, we have $\mathrm{dist}_{G'}(s, v) = T'[D, D', v]$. Hence, to compute the values $T'[D, D', v]$ for all vertices v, it suffices to run a single source shortest path algorithm from s in the graph G'.

To obtain linear time in the unweighted case, observe that in this case all weights of the edges sw are positive integers of order $\mathcal{O}(n)$, and still a single source shortest path algorithm can be performed in linear time.

6.10 We shall use integer linear programming, so as to avoid confusion with variables we use notation $s_1, s_2, \ldots, s_k$ for the input strings.

For every position $1 \le j \le L$, let $s_i[j]$ be the j-th letter of the string s_i, and let $\mathbf{s}^j = (s_1[j], s_2[j], \ldots, s_k[j]) \in \Sigma^k$ be the *tuple at position* j. Define an Integer Linear Programming instance using the following variables: for every $\mathbf{s} \in \Sigma^k$ and for every $\sigma \in \Sigma$, the variable $x_{\mathbf{s},\sigma}$ counts the number of positions j such that $\mathbf{s}^j = \mathbf{s}$ and, in the solution s, the letter at position j is exactly σ.

6.11 Improve the approach of the previous exercise in the following way: we say that two tuples $\mathbf{s}, \mathbf{s}' \in \Sigma^k$ are *equivalent* if, for every $1 \le i, i' \le k$, we have $\mathbf{s}(i) = \mathbf{s}(i')$ if and only if $\mathbf{s}'(i) = \mathbf{s}'(i')$. Show that the variables corresponding to equivalent tuples can be merged (you need to be careful here with the index σ) and you can have only $k^{\mathcal{O}(k)}$ variables in your Integer Linear Programming instance.

6.14 Let $G \in \mathcal{G} + k\mathrm{v}$ and let $X \subseteq V(G)$ be such that $|X| \le k$ and $G - X \in \mathcal{G}$. Let H be a minor of G, and let $(V_h)_{h \in V(H)}$ be a minor model of H in G. Define $Y = \{h \in V(H) : X \cap V_h \neq \emptyset\}$. Clearly, $|Y| \le |X| \le k$. Let $Z = \bigcup_{h \in Y} V_h \subseteq V(G)$. Observe that $G - Z$ is a subgraph of $G - X$. Moreover, $(V_h)_{h \in V(H) \setminus Y}$ is a minor model of $H - Y$ in $G - Z$. Since $\mathcal{G}$ is minor-closed, we have $H - Y \in \mathcal{G}$ and, consequently, $H \in \mathcal{G} + k\mathrm{v}$.

6.16 The main observation is that edge contraction cannot increase a diameter. Deduce from this fact that, for every fixed k, the class of graphs G for which (G, k) is a yes-instance to Planar Diameter Improvement is minor-closed.

6.19 Again, show that for fixed k, the class of graphs G for which (G, k) is a yes-instance is minor-closed.

Bibliographic notes

The algorithm for Set Cover is taken from [198]. The optimality of this algorithm is discussed in [112]. For a comprehensive discussion on exponential-time algorithms for computing the permanent, we refer the reader to [38] (cf. Exercise 6.3). The algorithm for Steiner Tree is the classical Dreyfus-Wagner algorithm from [159]. The first improvement of this algorithm is due to Fuchs, Kern, Mölle, Richter, Rossmanith, and Wang [219] down to $(2+\varepsilon)^k n^{f(1/\varepsilon)}$. This was improved to $2^k n^{\mathcal{O}(1)}$ by Björklund, Husfeldt, Kaski, and Koivisto [37]; see also Section 10.1.2 for a polynomial-space algorithm of Nederlof [374]. The fact that the Dreyfus-Wagner algorithm becomes polynomial if we restrict ourselves to planar graphs with all terminals on the infinite face (Exercise 6.8) was first observed by Erickson, Monma, and Veinott [168]. The algorithm of Exercise 6.6 is from [245]. After a long line of improvements [245, 367, 187, 367, 34, 118], the currently fastest algorithm for Connected Vertex Cover running in deterministic time $2^k n^{\mathcal{O}(1)}$ is due to Cygan [111], and is conjectured to be optimal [112].

Lenstra [319] showed that Integer Linear Programming Feasibility is FPT with running time doubly exponential in p. Later, Kannan [280] provided an algorithm for

Integer Linear Programming Feasibility running in time $p^{\mathcal{O}(p)}$. The algorithm uses Minkowski's Convex Body theorem and other results from the geometry of numbers. A bottleneck in this algorithm was that it required space exponential in p. Using the method of simultaneous Diophantine approximation, Frank and Tardos [215] describe preprocessing techniques, using which it is shown that Lenstra's and Kannan's algorithms can be made to run in polynomial space. They also slightly improve the running time of the algorithm. For our purposes, we use this algorithm. Later, a randomized algorithm for Integer Linear Programming Feasibility was provided by Clarkson; we refer the reader to [95] for further details. The result of Lenstra was extended by Khachiyan and Porkolab [288] to semidefinite integer programming.

An FPT algorithm for Imbalance parameterized by the objective function (the imbalance of the ordering in question) is given in [327]. The algorithm for Imbalance parameterized by vertex cover (Theorem 6.7) is from [185]. A similar approach works for several other vertex ordering problems.

The Graph Minors project of Robertson and Seymour spans over 23 papers, most of the work published in the Journal of Combinatorial Theory Ser. B. A nice overview of this project is contained in the book of Diestel [138]. Interestingly enough, while it was not the main goal of the project, many of the tools developed by Robertson and Seymour to settle Wagner's conjecture are used for a very broad class of algorithmic problems. Kuratowski's theorem is the classical theorem in graph theory [314]. It was proved originally for topological minors; the statement in terms of minors is due to Wagner [428]. The lower bound on the size of a partial list of graph minors for toroidal graphs is taken from the work of Gagarin, Myrvold, and Chambers [222]. We recommend the book of Mohar and Thomas [366] for more background on topological graph theory.

To circumvent the discussed issue of nonuniformity, Fellows and Langston [33, 180, 182, 181, 183, 184] have developed some general techniques using self-reducibility and a graph-theoretic generalization of the Myhill-Nerode theorem of formal language theory [183], to algorithmically construct the set of forbidden minors along the way. Thereby, for some of the considered problems they proved uniform fixed-parameter tractability.

The (uniform) fixed-parameter algorithm for Graph Genus was given by Mohar [365]. There are several constructive algorithms for Planar Vertex Deletion [358, 285, 277], with the current fastest algorithm by Jansen, Lokshtanov, and Saurabh [277] having running time $k^{\mathcal{O}(k)}n$.

An embedding of a graph G into $\mathbb{R}^3$ is called *linkless* if every two vertex-disjoint cycles of G can be separated from each other by a continuous transformation (i.e., they do not look like two consecutive links of a chain). In the Linkless Embedding problem, defined in Exercise 6.17, the graphs admitting a linkless embedding are yes-instances for $k \leq 1$. An explicit family of forbidden subgraphs for graphs admitting a linkless embedding was first conjectured by Sachs, and proved to be correct by Robertson, Seymour and Thomas [404]. There has been substantial work (e.g., [287] and [267]) to understand the computational complexity of finding a linkless embedding of a graph.

Chapter 7
Treewidth

The treewidth of a graph is one of the most frequently used tools in parameterized algorithms. Intuitively, treewidth measures how well the structure of a graph can be captured by a tree-like structural decomposition. When the treewidth of a graph is small, or equivalently the graph admits a good tree decomposition, then many problems intractable on general graphs become efficiently solvable. In this chapter we introduce treewidth and present its main applications in parameterized complexity. We explain how good tree decompositions can be exploited to design fast dynamic-programming algorithms. We also show how treewidth can be used as a tool in more advanced techniques, like shifting strategies, bidimensionality, or the irrelevant vertex approach.

In Section 6.3, we gave a brief overview on how the deep theory of Graph Minors of Robertson and Seymour can be used to obtain nonconstructive FPT algorithms. One of the tools defined by Robertson and Seymour in their work was the treewidth of a graph. Very roughly, treewidth captures how similar a graph is to a tree. There are many ways to define "tree-likeness" of a graph; for example, one could measure the number of cycles, or the number of vertices needed to be removed in order to make the graph acyclic. However, it appears that the approach most useful from algorithmic and graph theoretical perspectives, is to view tree-likeness of a graph G as the existence of a structural decomposition of G into pieces of bounded size that are connected in a tree-like fashion. This intuitive concept is formalized via the notions of a *tree decomposition* and the *treewidth* of a graph; the latter is a quantitative measure of how good a tree decomposition we can possibly obtain.

Treewidth is a fundamental tool used in various graph algorithms. In parameterized complexity, the following win/win approach is commonly ex-

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_7

ploited. Given some computational problem, let us try to construct a good tree decomposition of the input graph. If we succeed, then we can use dynamic programming to solve the problem efficiently. On the other hand, if we fail and the treewidth is large, then there is a reason for this outcome. This reason has the form of a combinatorial obstacle embedded in the graph that forbids us to decompose it expeditiously. However, the existence of such an obstacle can also be used algorithmically. For example, for some problems like VERTEX COVER or FEEDBACK VERTEX SET, we can immediately conclude that we are dealing with a no-instance in case the treewidth is large. For other problems, like LONGEST PATH, large treewidth implies that we are working with a yes-instance. In more complicated cases, one can examine the structure of the obstacle in the hope of finding a so-called irrelevant vertex or edge, whose removal does not change the answer to the problem. Thus, regardless of whether the initial construction of a good tree decomposition succeeded or failed, we win: we solve the problem by dynamic programming, or we are able to immediately provide the answer, or we can simplify the problem and restart the algorithm.

We start the chapter by slowly introducing treewidth and tree decompositions, and simultaneously showing connections to the idea of dynamic programming on the structure of a graph. In Section 7.1 we build the intuition by explaining, on a working example of WEIGHTED INDEPENDENT SET, how dynamic-programming procedures can be designed on trees and on subgraphs of grids. These examples bring us naturally to the definitions of path decompositions and pathwidth, and of tree decompositions and treewidth; these topics are discussed in Section 7.2. In Section 7.3 we provide the full framework of dynamic programming on tree decompositions. We consider carefully three exemplary problems: WEIGHTED INDEPENDENT SET, DOMINATING SET, and STEINER TREE; the examples of DOMINATING SET and STEINER TREE will be developed further in Chapter 11.

Section 7.4 is devoted to connections between graphs of small treewidth and monadic second-order logic on graphs. In particular, we discuss a powerful meta-theorem of Courcelle, which establishes the tractability of decision problems definable in Monadic Second-Order logic on graphs of bounded treewidth. Furthermore, we also give an extension of Courcelle's theorem to optimization problems.

In Section 7.5 we present a more combinatorial point of view on pathwidth and treewidth, by providing connections between these graph parameters, various search games on graphs, and classes of interval and chordal graphs. While these discussions are not directly relevant to the remaining part of this chapter, we think that they give an insight into the nature of pathwidth and treewidth that is invaluable when working with them.

In Section 7.6 we address the question of how to compute a reasonably good tree decomposition of a graph. More precisely, we present an approximation algorithm that, given an n-vertex graph G and a parameter k, works in time $\mathcal{O}(8^k k^2 \cdot n^2)$ and either constructs a tree decomposition of G of width at most $4k+4$, or reports correctly that the treewidth of G is more than k. Application of this algorithm is usually the first step of the classic win/win framework described in the beginning of this section.

We then move to using treewidth and tree decompositions as tools in more involved techniques. Section 7.7 is devoted to applications of treewidth for designing FPT algorithms on planar graphs. Most of these applications are based on deep structural results about obstacles to admitting good tree decompositions. More precisely, in Section 7.7.1 we introduce the so-called Excluded Grid Theorem and some of its variants, which states that a graph of large treewidth contains a large grid as minor. In the case of planar graphs, the theorem gives a very tight relation between the treewidth and the size of the largest grid minor that can be found in a graph. This fact is then exploited in Section 7.7.2, where we introduce the powerful framework of *bidimensionality*, using which one can derive parameterized algorithms on planar graphs with subexponential parametric dependence of the running time. In Section 7.7.3 we discuss the parameterized variant of the shifting technique; a reader familiar with basic results on approximation algorithms may have seen this method from a different angle. We apply the technique to give fixed-parameter tractable algorithms on planar graphs for SUBGRAPH ISOMORPHISM and MINIMUM BISECTION.

In Section *7.8 we give an FPT algorithm for a problem where a more advanced version of the treewidth win/win approach is implemented. More precisely, we provide an FPT algorithm for PLANAR VERTEX DELETION, the problem of deleting at most k vertices to obtain a planar graph. The crux of this algorithm is to design an *irrelevant vertex rule*: to prove that if a large grid minor can be found in the graph, one can identify a vertex that can be safely deleted without changing the answer to the problem. This technique is very powerful, but also requires attention to many technical details. For this reason, some technical steps of the correctness proof are omitted.

The last part of this chapter, Section 7.9, gives an overview of other graph parameters related to treewidth, such as branchwidth and rankwidth.

7.1 Trees, narrow grids, and dynamic programming

Imagine that you want to have a party and invite some of your colleagues from work to come to your place. When preparing the list of invitations, you would like to maximize the *total fun factor* of the invited people. However, from experience you know that there is not much fun when your direct boss is also invited. As you want everybody at the party to have fun, you would rather avoid such a situation for any of the invited colleagues.

We model this problem as follows. Assume that job relationships in your company are represented by a rooted tree T. Vertices of the tree represent your colleagues, and each $v \in V(T)$ is assigned a nonnegative weight $\mathbf{w}(v)$

that represents the amount of contributed fun for a particular person. The task is to find the maximum weight of an independent set in T, that is, of a set of pairwise nonadjacent vertices. We shall call this problem Weighted Independent Set.

This problem can be easily solved on trees by making use of dynamic programming. As usual, we solve a large number of subproblems that depend on each other. The answer to the problem shall be the value of a single, topmost subproblem. Assume that r is the root of the tree T (which corresponds to the superior of all the employees, probably the CEO). For a vertex v of T, let T_v be the subtree of T rooted at v. For the vertex v we define the following two values:

- Let $A[v]$ be the maximum possible weight of an independent set in T_v.
- Let $B[v]$ be the maximum possible weight of an independent set in T_v that does not contain v.

Clearly, the answer to the whole problem is the value of $A[r]$.

Values of $A[v]$ and $B[v]$ for leaves of T are equal to $\mathbf{w}(v)$ and 0, respectively. For other vertices, the values are calculated in a bottom-up order. Assume that $v_1, \ldots, v_q$ are the children of v. Then we can use the following recursive formulas:

$$B[v] = \sum_{i=1}^{q} A[v_i]$$

and

$$A[v] = \max \Big\{ B[v], \mathbf{w}(v) + \sum_{i=1}^{q} B[v_i] \Big\}.$$

Intuitively, the correctness of these formulas can be explained as follows. We know that $B[v]$ stores the maximum possible weight of an independent set in T_v that does not contain v and, thus, the independent set we are seeking is contained in $T_{v_1}, \ldots, T_{v_q}$. Furthermore, since T_v is a tree, there is no edge between vertices of two distinct subtrees among $T_{v_1}, \ldots, T_{v_q}$. This in turn implies that the maximum possible weight of an independent set of T_v that does not contain v is the sum of maximum possible weights of an independent set of T_{v_i}, $i \in \{1, \ldots, q\}$. The formula for $A[v]$ is justified by the fact that an independent set in T_v of the maximum possible weight either contains v, which is taken care of by the term $\mathbf{w}(v) + \sum_{i=1}^{q} B[v_i]$, or does not contain v, which is taken care of by the term $B[v]$. Therefore, what remains to do is to calculate the values of $A[v]$ and $B[v]$ in a bottom-up manner in the tree T, and finally read the answer from $A[r]$. This procedure can be clearly implemented in linear time. Let us remark that with a slight modification of the algorithm, using a standard method of remembering the origin of computed values as backlinks, within the same running time one can find not only the maximum possible weight, but also the corresponding independent set.

Let us now try to solve Weighted Independent Set on a different class of graphs. The problem with trees is that they are inherently "thin", so let us

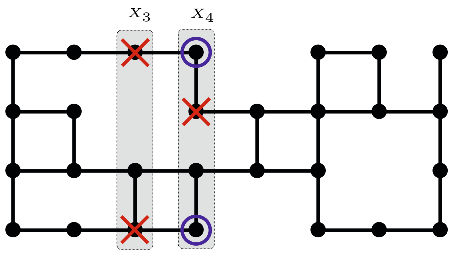


Fig. 7.1: A subgraph of a 4×8 grid, with the third and the fourth columns highlighted. When computing $c[4, Y]$ for the forbidden set $Y = \{(2, 4)\}$ (crossed out in the fourth column), one of the sets S we consider is $S = \{(1, 4), (4, 4)\}$, depicted with blue circles. Then, when looking at the previous column we need to forbid picking neighbors $(1, 3)$ and $(4, 3)$ (crossed out in the third column), since this would violate the independence constraint

try to look at graphs that are "thicker" in nature, like grids. Since the class of grids is not very broad, let us rather focus on subgraphs of grids. More precisely, assume that we are given a graph G that is a subgraph of a $k \times N$ grid. The vertex set of a $k \times N$ grid consists of all pairs of the form (i, j) for $1 \leq i \leq k$ and $1 \leq j \leq N$, and two pairs (i_1, j_1) and (i_2, j_2) are adjacent if and only if $|i_1 - i_2| + |j_1 - j_2| = 1$. Graph G is a subgraph of an $k \times N$ grid, which means that some vertices and edges of the grid can be missing in G. Figure 7.1 presents an example of a subgraph of a 4×8 grid. In our considerations, we will think of the number of rows k as quite small (say, $k = 10$), while N, the number of columns, can be very large (say, $N = 10^6$). Of course, since we are trying to solve the Weighted Independent Set problem, every vertex $v \in V(G)$ is assigned its weight $\mathbf{w}(v)$.

Let X_j be the j-th column of G, that is, $X_j = V(G) \cap \{(i, j) : 1 \leq i \leq k\}$. Moreover, for $1 \leq j \leq N$, let $G_j = G[X_1 \cup X_2 \cup \ldots \cup X_j]$ be the graph induced by the first j columns. We would like to run a dynamic programming-algorithm that sweeps the grid from left to right, column by column. In the case of trees, we recognized two possible situations that were handled differently in the two dynamic-programming tables: either the root of the subtree was allowed to be picked into an independent set, or it was forbidden. Mimicking this idea, let us define the following function $c[j, Y]$ that we shall compute in the algorithm. For $Y \subseteq X_j$, we take the following definition:

$$c[j, Y] = \text{maximum possible weight of an independent set in } G_j - Y.$$

In other words, we are examining graph G_j, and we look for the best possible independent set that avoids picking vertices from Y. We now move on to explaining how the values of $c[j, Y]$ will be computed.

For $j = 1$, the situation is very simple. For every $Y \subseteq X_1$, we iterate through all possible subsets $S \subseteq X_1 \setminus Y$, and for each of them we check whether it is an independent set. Then $c[1, Y]$ is the largest weight among the candidates (independent sets) that passed this test. Since for each $Y \subseteq X_1$ we iterate through all possible subsets of $X_1 \setminus Y$, in total, for all the Ys, the number of checks is bounded by

$$\sum_{\ell=0}^{|X_1|} \binom{|X_1|}{\ell} 2^{|X_1|-\ell} = 3^{|X_1|} \leq 3^k. \tag{7.1}$$

Here, factor $\binom{|X_1|}{\ell}$ comes from the choice of Y (the sum iterates over $\ell = |Y|$), while factor $2^{|X_1|-\ell}$ represents the number of choices for $S \subseteq X_1 \setminus Y$. Since every check can be implemented in $k^{\mathcal{O}(1)}$ time (assuming that for every vertex we store a list of its neighbors), the total time spent on computing values $c[1, \cdot]$ is $3^k \cdot k^{\mathcal{O}(1)}$.

We now show how to compute the values of $c[j, \cdot]$ depending on the precomputed values of $c[j-1, \cdot]$, for $j > 1$. Let us look at one value $c[j, Y]$ for some $Y \subseteq X_j$. Similarly, for $j = 1$, we should iterate through all the possible ways a maximum weight independent set intersects column X_j. This intersection should be independent, of course, and moreover if we pick some $v \in X_j \setminus Y$ to the independent set, then this choice forbids choosing its neighbor in the previous column X_{j-1} (providing this neighbor exists). But for column X_{j-1} we have precomputed answers for *all* possible combinations of forbidden vertices. Hence, we can easily read from the precomputed values what is the best possible weight of an extension of the considered intersection with X_j to the previous columns. All in all, we arrive at the following recursive formula:

$$c[j, Y] = \max_{\substack{S \subseteq X_i \setminus Y \\ S \text{ is independent}}} \Big\{ \mathbf{w}(S) + c[j-1, N(S) \cap X_{j-1}] \Big\}; \tag{7.2}$$

here $\mathbf{w}(S) = \sum_{v \in S} \mathbf{w}(v)$. Again, when applying (7.2) for every $Y \subseteq X_i$ we iterate through all the subsets of $X_j \setminus Y$. Therefore, as in (7.1) we obtain that the total number of sets S checked, for all the sets Y, is at most 3^k. Each S is processed in $k^{\mathcal{O}(1)}$ time, so the total time spent on computing values $c[j, \cdot]$ is $3^k \cdot k^{\mathcal{O}(1)}$.

To wrap up, we first compute the values $c[1, \cdot]$, then iteratively compute values $c[j, \cdot]$ for $j \in \{2, 3, \ldots, N\}$ using (7.2), and conclude by observing that the answer to the problem is equal to $c[N, \emptyset]$. As argued, each iteration takes time $3^k \cdot k^{\mathcal{O}(1)}$, so the whole algorithm runs in time $3^k \cdot k^{\mathcal{O}(1)} \cdot N$.

Let us now step back and look at the second algorithm that we designed. Basically, the only property of G that we really used is that $V(G)$ can be partitioned into a sequence of small subsets (columns, in our case), such that edges of G can connect only two consecutive subsets. In other words, G has a "linear structure of separators", such that each separator separates the part lying on the left of it from the part on the right. In the algorithm we actually used the fact that the columns are disjoint, but this was not that crucial. The main point was that only two consecutive columns can interact.

Let us take a closer look at the choice of the definition of the table $c[j, Y]$. Let I be an independent set in a graph G that is a subgraph of a $k \times N$ grid. For fixed column number j, we can look at the index Y in the cell $c[j, Y]$ as the succinct representation of the interaction between $I \cap \bigcup_{a=1}^{j} X_a$ and $I \cap \bigcup_{a=j+1}^{N} X_a$. More precisely, assume that $Y = X_j \cap N(I \cap X_{j+1})$ and, consequently, the size of $C = I \cap \bigcup_{a=1}^{j} X_a$ is one of the candidates for the value $c[j, Y]$. Furthermore, let C' be any other candidate for the value $c[j, Y]$, that is, let C' be any independent set in $G_j \setminus Y$. Observe that then $(I \setminus C) \cup C'$ is also an independent set in G. In other words, when going from the column j to the column $j+1$, the only information we need to remember is the set Y of vertices "reserved as potential neighbors", and, besides the set Y, we do not care how exactly the solution looked so far.

If we now try to lift these ideas to general graphs, then the obvious thing to do is to try to merge the algorithms for trees and for subgraphs of grids into one. As $k \times N$ grids are just "fat" paths, we should define the notion of trees of "fatness" k. This is exactly the idea behind the notion of treewidth. In the next section we shall first define parameter *pathwidth*, which encapsulates in a more general manner the concepts that we used for the algorithm on subgraphs of grids. From pathwidth there will be just one step to the definition of *treewidth*, which also encompasses the case of trees, and which is the main parameter we shall be working with in this chapter.

7.2 Path and tree decompositions

We have gathered already enough intuition so that we are ready to introduce formally the main notions of this chapter, namely path and tree decompositions.

Path decompositions. A *path decomposition* of a graph G is a sequence $\mathcal{P} = (X_1, X_2, \ldots, X_r)$ of *bags*, where $X_i \subseteq V(G)$ for each $i \in \{1, 2, \ldots, r\}$, such that the following conditions hold:

(P1) $\bigcup_{i=1}^{r} X_i = V(G)$. In other words, every vertex of G is in at least one bag.
(P2) For every $uv \in E(G)$, there exists $\ell \in \{1, 2, \ldots, r\}$ such that the bag X_ℓ contains both u and v.
(P3) For every $u \in V(G)$, if $u \in X_i \cap X_k$ for some $i \leq k$, then $u \in X_j$ also for each j such that $i \leq j \leq k$. In other words, the indices of the bags containing u form an interval in $\{1, 2, \ldots, r\}$.

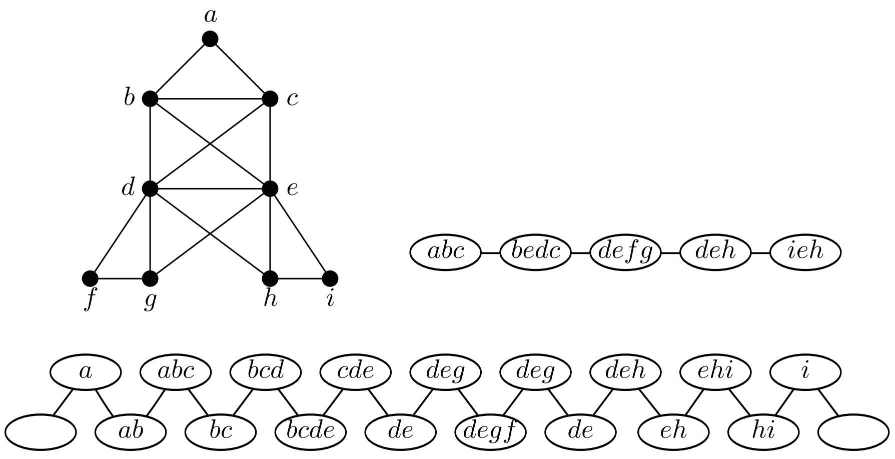


Fig. 7.2: A graph, its path and nice path decompositions

See Fig. 7.2 for example of a path decomposition. The *width* of a path decomposition $(X_1, X_2, \ldots, X_r)$ is $\max_{1 \le i \le r} |X_i| - 1$. The *pathwidth* of a graph G, denoted by $\mathrm{pw}(G)$, is the minimum possible width of a path decomposition of G. The reason for subtracting 1 in the definition of the width of the path decomposition is to ensure that the pathwidth of a path with at least one edge is 1, not 2. Similarly, we subtract in the definition of treewidth to ensure that the treewidth of a tree is 1.

We can also interpret the bags of a path decomposition as nodes of a path and two consecutive bags correspond to two adjacent nodes of the path. This interpretation will become handy when we introduce tree decomposition.

For us the most crucial property of path decompositions is that they define a sequence of separators in the graph. In the following, we will say that (A, B) is a *separation* of a graph G if $A \cup B = V(G)$ and there is no edge between $A \setminus B$ and $B \setminus A$. Then $A \cap B$ is a *separator* of this separation, and $|A \cap B|$ is the *order* of the separation. Note that any path in G that begins in A and ends in B must contain at least one vertex of the separator $A \cap B$. Also, for a subset $A \subseteq V(G)$ we define the *border* of A, denoted by $\partial(A)$, as the set of those vertices of A that have a neighbor in $V(G) \setminus A$. Note that $(A, (V(G) \setminus A) \cup \partial(A))$ is a separation with separator $\partial(A)$. Let us remark that by Lemma 7.1, each of the bags X_i separates vertices in the bags before i with the vertices of the bags following after i.

Lemma 7.1. *Let $(X_1, X_2, \ldots, X_r)$ be a path decomposition of a graph G. Then for every $j \in \{1, \ldots, r-1\}$ it holds that $\partial(\bigcup_{i=1}^{j} X_i) \subseteq X_j \cap X_{j+1}$. In other words, $(\bigcup_{i=1}^{j} X_i, \bigcup_{i=j+1}^{r} X_i)$ is a separation of G with separator $X_j \cap X_{j+1}$.*

Proof. Let us fix j and let $(A, B) = (\bigcup_{i=1}^{j} X_i, \bigcup_{i=j+1}^{r} X_i)$. We first show that $\partial(\bigcup_{i=1}^{j} X_i) = \partial(A) \subseteq X_j \cap X_{j+1}$. Targeting a contradiction, let us assume that there is a $u \in \partial(A)$ such that $u \notin X_j \cap X_{j+1}$. This means that there is an edge $uv \in E(G)$ such that $u \in A$, $v \notin A$ but also $u \notin X_j \cap X_{j+1}$. Let i be the largest index such that $u \in X_i$ and k be the smallest index such that $v \in X_k$. Since $u \in A$ and $u \notin X_j \cap X_{j+1}$, (P3) implies that $i \leq j$. Since $v \notin A$, we have also $k \geq j+1$. Therefore $i < k$. On the other hand, by (P2) there should be a bag X_ℓ containing both u and v. We obtain that $\ell \leq i < k \leq \ell$, which is a contradiction. The fact that $A \cap B = X_j \cap X_{j+1}$ follows immediately from (P3). □

Note that we can always assume that no two consecutive bags of a path decomposition are equal, since removing one of such bags does not violate any property of a path decomposition. Thus, a path decomposition $(X_1, X_2, \ldots, X_r)$ of width p naturally defines a sequence of separations $(\bigcup_{i=1}^{j} X_i, \bigcup_{i=j+1}^{r} X_i)$. Each of these separations has order at most p, because the intersection of two different sets of size at most $p+1$ has size at most p.

We now introduce sort of a "canonical" form of a path decomposition, which will be useful in the dynamic-programming algorithms presented in later sections. A path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_r)$ of a graph G is *nice* if

- $X_1 = X_r = \emptyset$, and
- for every $i \in \{1, 2, \ldots, r-1\}$ there is either a vertex $v \notin X_i$ such that $X_{i+1} = X_i \cup \{v\}$, or there is a vertex $w \in X_i$ such that $X_{i+1} = X_i \setminus \{w\}$.

See Fig. 7.2 for example.

Bags of the form $X_{i+1} = X_i \cup \{v\}$ shall be called *introduce bags* (or *introduce nodes*, if we view the path decomposition as a path rather than a sequence). Similarly, bags of the form $X_{i+1} = X_i \setminus \{w\}$ shall be called *forget bags* (*forget nodes*). We will also say that X_{i+1} *introduces* v or *forgets* w. Let us note that because of (P3), every vertex of G gets introduced and becomes forgotten exactly once in a nice path decomposition, and hence we have that r, the total number of bags, is exactly equal to $2|V(G)|+1$. It turns out that every path decomposition can be turned into a nice path decomposition of at most the same width.

Lemma 7.2. *If a graph G admits a path decomposition of width at most p, then it also admits a nice path decomposition of width at most p. Moreover, given a path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_r)$ of G of width at most p, one can in time $\mathcal{O}(p^2 \cdot \max(r, |V(G)|))$ compute a nice path decomposition of G of width at most p.*

The reader is asked to prove Lemma 7.2 in Exercise 7.1.

Tree decompositions. A tree decomposition is a generalization of a path decomposition. Formally, a *tree decomposition* of a graph G is a pair $\mathcal{T} =$

$(T, \{X_t\}_{t \in V(T)})$, where T is a tree whose every node t is assigned a vertex subset $X_t \subseteq V(G)$, called a bag, such that the following three conditions hold:

(T1) $\bigcup_{t \in V(T)} X_t = V(G)$. In other words, every vertex of G is in at least one bag.
(T2) For every $uv \in E(G)$, there exists a node t of T such that bag X_t contains both u and v.
(T3) For every $u \in V(G)$, the set $T_u = \{t \in V(T) \ : \ u \in X_t\}$, i.e., the set of nodes whose corresponding bags contain u, induces a connected subtree of T.

The *width* of tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ equals $\max_{t \in V(T)} |X_t| - 1$, that is, the maximum size of its bag minus 1. The *treewidth* of a graph G, denoted by $\mathrm{tw}(G)$, is the minimum possible width of a tree decomposition of G.

To distinguish between the vertices of the decomposition tree T and the vertices of the graph G, we will refer to the vertices of T as *nodes*. As we mentioned before, path decompositions correspond exactly to tree decompositions with the additional requirement that T has to be a path.

Let us give several examples of how small or large can be the treewidth of particular graph classes. Forests and trees are of treewidth at most 1 and cycles have treewidth 2, see Exercise 7.8. The treewidth of an outerplanar graph, which is a graph that can be drawn in the plane in such manner that all its vertices are on one face, is at most 2, see Exercise 7.12. On the other hand, the treewidth of planar graphs can be arbitrarily large. For example, as we will see later, the treewidth of a $t \times t$ grid is t. Interestingly, for every planar graph H, there is a constant c_H, such that for every graph G excluding H as a minor, the treewidth of G does not exceed c_H, see Exercise 7.36.

While planar graphs can have arbitrarily large treewidths, as we will see later, still the treewidth of an n-vertex planar graph is sublinear, more precisely $\mathcal{O}(\sqrt{n})$. The treewidth of an n-vertex clique K_n is $n-1$ and of a complete bipartite graph $K_{n,m}$ is $\min\{m, n\} - 1$. Expander graphs serve as an example of a sparse graph class with treewidth $\Omega(n)$, see Exercise 7.34.

In Lemma 7.1, we have proved that for every pair of adjacent nodes of a path decomposition, the intersection of the corresponding bags is a separator that separates the left part of the decomposition from the right part. The following lemma establishes a similar separation property of bags of a tree decomposition. Its proof is similar to the proof of Lemma 7.1 and is left to the reader as Exercise 7.5.

Lemma 7.3. *Let $(T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of a graph G and let ab be an edge of T. The forest $T - ab$ obtained from T by deleting edge ab consists of two connected components T_a (containing a) and T_b (containing b). Let $A = \bigcup_{t \in V(T_a)} X_t$ and $B = \bigcup_{t \in V(T_b)} X_t$. Then $\partial(A), \partial(B) \subseteq X_a \cap X_b$. Equivalently, (A, B) is a separation of G with separator $X_a \cap X_b$.*

Again, note that we can always assume that the bags corresponding to two adjacent nodes in a tree decomposition are not the same, since in such situation we could contract the edge between them, keeping the same bag in the node resulting from the contraction. Thus, if $(T, \{X_t\}_{t \in V(T)})$ has width t, then each of the separations given by Lemma 7.3 has order at most t.

Similarly to nice path decompositions, we can also define nice tree decompositions of graphs. It will be convenient to think of nice tree decompositions as rooted trees. That is, for a tree decomposition $(T, \{X_t\}_{t \in V(T)})$ we distinguish one vertex r of T which will be the root of T. This introduces natural parent-child and ancestor-descendant relations in the tree T. We will say that such a rooted tree decomposition $(T, \{X_t\}_{t \in V(T)})$ is *nice* if the following conditions are satisfied:

- $X_r = \emptyset$ and $X_\ell = \emptyset$ for every leaf ℓ of T. In other words, all the leaves as well as the root contain empty bags.
- Every non-leaf node of T is of one of the following three types:
  - **Introduce node**: a node t with exactly one child t' such that $X_t = X_{t'} \cup \{v\}$ for some vertex $v \notin X_{t'}$; we say that v is *introduced* at t.
  - **Forget node**: a node t with exactly one child t' such that $X_t = X_{t'} \setminus \{w\}$ for some vertex $w \in X_{t'}$; we say that w is *forgotten* at t.
  - **Join node**: a node t with two children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$.

At first glance the condition that the root and the leaves contain empty bags might seem unnatural. As we will see later, this property helps to streamline designing dynamic-programming algorithms on tree decompositions, which is the primary motivation for introducing nice tree decompositions. Note also that, by property (T3) of a tree decomposition, every vertex of $V(G)$ is forgotten only once, but may be introduced several times.

Also, note that we have assumed that the bags at a join node are equal to the bags of the children, sacrificing the previous observation that any separation induced by an edge of the tree T is of order at most t. The increase of the size of the separators from t to $t+1$ has negligible effect on the asymptotic running times of the algorithms, while nice tree decompositions turn out to be very convenient for describing the details of the algorithms.

The following result is an analogue of Lemma 7.2 for nice tree decompositions. The reader is asked to prove it in Exercise 7.2.

Lemma 7.4. *If a graph G admits a tree decomposition of width at most k, then it also admits a nice tree decomposition of width at most k. Moreover, given a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G of width at most k, one can in time $\mathcal{O}(k^2 \cdot \max(|V(T)|, |V(G)|))$ compute a nice tree decomposition of G of width at most k that has at most $\mathcal{O}(k|V(G)|)$ nodes.*

Due to Lemma 7.4, we will assume that all the nice tree decompositions used by our algorithms have $\mathcal{O}(k|V(G)|)$ nodes.

Note that in a general setting a good path/tree decomposition of an input graph is not known in advance. Hence, in the dynamic-programming algorithms we will always assume that such a decomposition is provided on the input together with the graph. For this reason, we will need to address separately how to compute its path/tree decomposition of optimum or near-optimum width, so that an efficient dynamic-programming algorithm can be employed on it. In this book we shall not answer this question for path decompositions and pathwidth. However, in Section 7.6 we present algorithms for computing tree decompositions of (approximately) optimum width.

7.3 Dynamic programming on graphs of bounded treewidth

In this section we give examples of dynamic-programming-based algorithms on graphs of bounded treewidth.

7.3.1 Weighted Independent Set

In Section 7.1 we gave two dynamic-programming routines for the Weighted Independent Set problem. The first of them worked on trees, and the second of them worked on subgraphs of grids. Essentially, in both cases the main idea was to define subproblems for parts of graphs separated by small separators. In the case of trees, every vertex of a tree separates the subtree rooted at it from the rest of the graph. Thus, choices made by the solution in the subtree are independent of what happens outside it. The algorithm for subgraphs of grids exploited the same principle: every column of the grid separates the part of the graph on the left of it from the part on the right.

It seems natural to combine these ideas for designing algorithms for graphs of bounded treewidth. Let us focus on our running example of the Weighted Independent Set problem, and let $\mathcal{T} = (T, \{X_t\}_{t\in V(T)})$ be a tree decomposition of the input n-vertex graph G that has width at most k. By applying Lemma 7.4 we can assume that $\mathcal{T}$ is a nice tree decomposition. Recall that then T is rooted at some node r. For a node t of T, let V_t be the union of all the bags present in the subtree of T rooted at t, including X_t. Provided that $t \neq r$ we can apply Lemma 7.3 to the edge of T between t and its parent, and infer that $\partial(V_t) \subseteq X_t$. The same conclusion is trivial when $t = r$, since then $V_r = V(G)$ and $\partial(V_r) = \emptyset$. This exactly formalizes the intuition that the subgraph induced by V_t can communicate with the rest of the graph only via bag X_t, which is of small size.

Extending our intuition from the algorithms of Section 7.1, we would like to define subproblems depending on the interaction between the solution and bag X_t. Consider the following: Let I_1, I_2 be two independent sets of G such that $I_1 \cap X_t = I_2 \cap X_t$. Let us add the weights of vertices of I_1 and I_2 that are contained in V_t, and suppose that it turned out that $\mathbf{w}(I_1 \cap V_t) > \mathbf{w}(I_2 \cap V_t)$. Observe that then solution I_2 is suboptimal for the following reason. We can obtain a solution I_2' from I_2 by replacing $I_2 \cap V_t$ with $I_1 \cap V_t$. The fact that X_t separates $V_t \setminus X_t$ from the rest of the graph, and that I_1 and I_2 have the same intersection with X_t, implies that I_2' is still an independent set. On the other hand, $\mathbf{w}(I_1 \cap V_t) > \mathbf{w}(I_2 \cap V_t)$ implies that $\mathbf{w}(I_2') > \mathbf{w}(I_2)$.

Therefore, among independent sets I satisfying $I \cap X_t = S$ for some fixed S, all the maximum-weight solutions have exactly the same weight of the part contained in V_t. This weight corresponds to the maximum possible value for the following subproblem: given $S \subseteq X_t$, we look for a maximum-weight extension $\widehat{S} \supseteq S$ such that $\widehat{S} \subseteq V_t$, $\widehat{S} \cap X_t = S$, and $\widehat{S}$ is independent. Indeed, the argument from the previous paragraph shows that for any solution I with $I \cap X_t = S$, the part of the solution contained in V_t can be safely replaced with the best possible partial solution $\widehat{S}$ — this replacement preserves independence and can only increase the weight. Observe that the number of subproblems is small: for every node t, we have only $2^{|X_t|}$ subproblems. Also, we do not need to remember $\widehat{S}$ explicitly; remembering its weight will suffice.

Hence, we now mimic the bottom-up dynamic programming that we performed for trees. For every node t and every $S \subseteq X_t$, define the following value:

$$\begin{aligned} c[t, S] = {} & \text{maximum possible weight of a set } \widehat{S} \text{ such that} \\ & S \subseteq \widehat{S} \subseteq V_t,\ \widehat{S} \cap X_t = S, \text{ and } \widehat{S} \text{ is independent.} \end{aligned}$$

If no such set $\widehat{S}$ exists, then we put $c[t, S] = -\infty$; note that this happens if and only if S is not independent itself. Also $c[r, \emptyset]$ is exactly the maximum weight of an independent set in G; this is due to the fact that $V_r = V(G)$ and $X_r = \emptyset$.

The reader can now observe that this definition of function $c[\cdot, \cdot]$ differs from the one used in Section 7.1. While there we were just *forbidding* vertices from some set Y from being used, now we fix *exactly* how a solution is supposed to interact with a bag. Usually, fixing the exact interaction of the solution with a bag is a more generic approach to designing dynamic-programming algorithms on tree decompositions. However, it must be admitted that tweaking the definition slightly (e.g., by relaxing the exactness condition to allow or forbid usage of a subset of vertices) often leads to a simpler description. This is actually the case also in the example of WEIGHTED INDEPENDENT SET. In Exercise 7.17 the reader is asked to work out the details of a dynamic program with the definition of a state mimicking the one used in Section 7.1.

We now move on to presenting how the values of $c[\cdot, \cdot]$ are computed. Thanks to the definition of a nice tree decomposition, there are only a few simple ways in which a bag at some node can relate to the bags of the children of this node. Therefore, if we are fortunate we can compute the values at each node t based on the values computed for the children of t. This will be done by giving recursive formulas for $c[t,S]$. The case when t is a leaf corresponds to the base case of the recurrence, whereas values for $c[\cdot,\cdot]$ for a non-leaf node t depend on the values of $c[\cdot,\cdot]$ for the children of t. By applying the formulas in a bottom-up manner on T we will finally compute $c[r,\emptyset]$, which is the value we are looking for.

Leaf node. If t is a leaf node, then we have only one value $c[t,\emptyset]=0$.

Introduce node. Suppose t is an introduce node with child t' such that $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$. Let S be any subset of X_t. If S is not independent, then we can immediately put $c[t,S] = -\infty$; hence assume otherwise. Then we claim that the following formula holds:

$$c[t,S] = \begin{cases} c[t',S] & \text{if } v \notin S; \\ c[t',S\setminus\{v\}] + \mathbf{w}(v) & \text{otherwise.} \end{cases} \tag{7.3}$$

To prove formally that this formula holds, consider first the case when $v \notin S$. Then the families of sets $\widehat{S}$ considered in the definitions of $c[t,S]$ and of $c[t',S]$ are equal, which immediately implies that $c[t,S]=c[t',S]$.

Consider the case when $v \in S$, and let $\widehat{S}$ be a set for which the maximum is attained in the definition of $c[t,S]$. Then it follows that $\widehat{S}\setminus\{v\}$ is one of the sets considered in the definition of $c[t',S\setminus\{v\}]$, which implies that $c[t',S\setminus\{v\}] \geq \mathbf{w}(\widehat{S}\setminus\{v\}) = \mathbf{w}(\widehat{S}) - \mathbf{w}(v) = c[t,S]-\mathbf{w}(v)$. Consequently,

$$c[t,S] \leq c[t',S\setminus\{v\}] + \mathbf{w}(v). \tag{7.4}$$

On the other hand, let $\widehat{S}'$ be a set for which the maximum is attained in the definition of $c[t',S\setminus\{v\}]$. Since we assumed that S is independent, we have that v does not have any neighbor in $S\setminus\{v\} = \widehat{S}'\cap X_{t'}$. Moreover, by Lemma 7.3, v does not have any neighbors in $V_{t'}\setminus X_{t'}$, which is a superset of $\widehat{S}'\setminus X_{t'}$. We conclude that v does not have any neighbors in $\widehat{S}'$, which means that $\widehat{S}'\cup\{v\}$ is an independent set. Since this set intersects with X_t exactly at S, it is considered in the definition of $c[t,S]$ and we have that

$$c[t,S] \geq \mathbf{w}(\widehat{S}'\cup\{v\}) = \mathbf{w}(\widehat{S}') + \mathbf{w}(v) = c[t',S\setminus\{v\}] + \mathbf{w}(v). \tag{7.5}$$

Concluding, (7.4) and (7.5) together prove (7.3) for the case $v \in S$.

Forget node. Suppose t is a forget node with child t' such that $X_t = X_{t'}\setminus\{w\}$ for some $w \in X_{t'}$. Let S be any subset of X_t; again we assume that S is independent, since otherwise we put $c[t,S]=-\infty$. We claim that the following formula holds:

$$c[t,S] = \max\Big\{c[t',S], c[t',S\cup\{w\}]\Big\}. \tag{7.6}$$

We now give a formal proof of this formula. Let $\widehat{S}$ be a set for which the maximum is attained in the definition of $c[t,S]$. If $w \notin \widehat{S}$, then $\widehat{S}$ is one of the sets considered in the definition of $c[t',S]$, and hence $c[t',S] \geq \mathbf{w}(\widehat{S}) = c[t,S]$. However, if $w \in \widehat{S}$ then $\widehat{S}$ is one of the sets considered in the definition of $c[t', S \cup \{w\}]$, and then $c[t', S \cup \{w\}] \geq \mathbf{w}(\widehat{S}) = c[t,S]$. As exactly one of these alternatives happens, we can infer that the following holds always:

$$c[t,S] \leq \max\Big\{c[t',S], c[t', S \cup \{w\}]\Big\}. \tag{7.7}$$

On the other hand, observe that each set that is considered in the definition of $c[t',S]$ is also considered in the definition of $c[t,S]$, and the same holds also for $c[t', S \cup \{w\}]$. This means that $c[t,S] \geq c[t',S]$ and $c[t,S] \geq c[t', S \cup \{w\}]$. These two inequalities prove that

$$c[t,S] \geq \max\Big\{c[t',S], c[t', S \cup \{w\}]\Big\}. \tag{7.8}$$

The combination of (7.7) and (7.8) proves that formula (7.6) indeed holds.

Join node. Finally, suppose that t is a join node with children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$. Let S be any subset of X_t; as before, we can assume that S is independent. The claimed recursive formula is as follows:

$$c[t,S] = c[t_1,S] + c[t_2,S] - \mathbf{w}(S). \tag{7.9}$$

We now prove formally that this formula holds. First take $\widehat{S}$ to be a set for which the maximum is attained in the definition of $c[t,S]$. Let $\widehat{S}_1 = \widehat{S} \cap V_{t_1}$ and $\widehat{S}_2 = \widehat{S} \cap V_{t_2}$. Observe that $\widehat{S}_1$ is independent and $\widehat{S}_1 \cap X_{t_1} = S$, so this set is considered in the definition of $c[t_1,S]$. Consequently $c[t_1,S] \geq \mathbf{w}(\widehat{S}_1)$, and analogously $c[t_2,S] \geq \mathbf{w}(\widehat{S}_2)$. Since $\widehat{S}_1 \cap \widehat{S}_2 = S$, we obtain the following:

$$c[t,S] = \mathbf{w}(\widehat{S}) = \mathbf{w}(\widehat{S}_1) + \mathbf{w}(\widehat{S}_2) - \mathbf{w}(S) \leq c[t_1,S] + c[t_2,S] - \mathbf{w}(S). \tag{7.10}$$

On the other hand, let $\widehat{S}'_1$ be a set for which the maximum is attained in the definition of $c[t_1,S]$, and similarly define $\widehat{S}'_2$ for $c[t_2,S]$. By Lemma 7.3 we have that there is no edge between vertices of $V_{t_1} \setminus X_t$ and $V_{t_2} \setminus X_t$, which implies that the set $\widehat{S}' := \widehat{S}'_1 \cup \widehat{S}'_2$ is independent. Moreover $\widehat{S}' \cap X_t = S$, which implies that $\widehat{S}'$ is one of the sets considered in the definition of $c[t,S]$. Consequently,

$$c[t,S] \geq \mathbf{w}(\widehat{S}') = \mathbf{w}(\widehat{S}'_1) + \mathbf{w}(\widehat{S}'_2) - \mathbf{w}(S) = c[t_1,S] + c[t_2,S] - \mathbf{w}(S). \tag{7.11}$$

From (7.10) and (7.11) we infer that (7.9) indeed holds.

This concludes the description and the proof of correctness of the recursive formulas for computing the values of $c[\cdot,\cdot]$. Let us now wrap up the whole algorithm and estimate the running time. Recall that we are working on a tree decomposition of width at most k, which means that $|X_t| \leq k+1$ for every node t. Thus at node t we compute $2^{|X_t|} \leq 2^{k+1}$ values of $c[t, S]$. However, we have to be careful when estimating the time needed for computing each of these values.

Of course, we could just say that using the recursive formulas and any graph representation we can compute each value $c[t, S]$ in $n^{\mathcal{O}(1)}$ time, but then we would end up with an algorithm running in time $2^k \cdot n^{\mathcal{O}(1)}$. It is easy to see that all the operations needed to compute one value can be performed in $k^{\mathcal{O}(1)}$ time, apart from checking adjacency of a pair of vertices. We need it, for example, to verify that a set S is independent. However, a straightforward implementation of an adjacency check runs in $\mathcal{O}(n)$ time, which would add an additional $\mathcal{O}(n)$ factor to the running time of the algorithm. Nonetheless, as G is a graph of treewidth at most k, it is possible to construct a data structure in time $k^{\mathcal{O}(1)}n$ that allows performing adjacency queries in time $\mathcal{O}(k)$. The reader is asked to construct such a data structure in Exercise 7.16.

Wrapping up, for every node t it takes time $2^k \cdot k^{\mathcal{O}(1)}$ to compute all the values $c[t, S]$. Since we can assume that the number of nodes of the given tree decompositions is $\mathcal{O}(kn)$ (see Lemma 7.4), the total running time of the algorithm is $2^k \cdot k^{\mathcal{O}(1)} \cdot n$. Hence, we obtain the following theorem.

Theorem 7.5. *Let G be an n-vertex graph with weights on vertices given together with its tree decomposition of width at most k. Then the* WEIGHTED INDEPENDENT SET *problem in G is solvable in time $2^k \cdot k^{\mathcal{O}(1)} \cdot n$.*

Again, using the standard technique of backlinks, i.e., memorizing for every cell of table $c[\cdot, \cdot]$ how its value was obtained, we can reconstruct the solution (i.e., an independent set with the maximum possible weight) within the same asymptotic running time. The same will hold also for all the other dynamic-programming algorithms given in this section.

Since a graph has a vertex cover of size at most ℓ if and only if it has an independent set of size at least $n - \ell$, we immediately obtain the following corollary.

Corollary 7.6. *Let G be an n-vertex graph given together with its tree decomposition of width at most k. Then one can solve the* VERTEX COVER *problem in G in time $2^k \cdot k^{\mathcal{O}(1)} \cdot n$.*

The algorithm of Theorem 7.5 seems actually quite simple, so it is very natural to ask if its running time could be improved. We will see in Chapter 14, Theorem 14.38, that under some reasonable assumptions the upper bound of Theorem 7.5 is tight.

The reader might wonder now why we gave all the formal details of this dynamic-programming algorithm, even though most of them were straightforward. Our point is that despite the naturalness of many dynamic-programming algorithms on tree decompositions, proving their correctness formally needs a lot of attention and care with regard to the details. We tried to show which implications need to be given in order to obtain a complete and formal proof, and what kind of arguments can be used along the way.

Most often, proving correctness boils down to showing two inequalities for each type of node: one relating an optimum solution for the node to some solutions for its children, and the second showing the reverse correspondence. It is usual that a precise definition of a state of the dynamic program together with function c denoting its value already suggests natural recursive formulas for c. Proving correctness of these formulas is usually a straightforward and tedious task, even though it can be technically challenging.

For this reason, in the next dynamic-programming routines we usually only give a precise definition of a state, function c, and the recursive formulas for computing c. We resort to presenting a short rationale for why the formulas are correct, leaving the full double-implication proof to the reader. We suggest that the reader always performs such double-implication proofs for his or her dynamic-programming routines, even though all the formal details might not always appear in the final write-up of the algorithm. Performing such formal proofs can highlight the key arguments needed in the correctness proof, and is the best way of uncovering possible problems and mistakes.

Another issue that could be raised is why we should not go further and also estimate the polynomial factor depending on k in the running time of the algorithm, which is now stated just as $k^{\mathcal{O}(1)}$. We refrain from this, since the actual value of this factor depends on the following:

- How fast we can implement all the low-level details of computing formulas for $c[t, S]$, e.g., iteration through subsets of vertices of a bag. This in particular depends on how we organize the structure of G and $\mathcal{T}$ in the memory.
- How fast we can access the exponential-size memory needed for storing the dynamic-programming table $c[\cdot, \cdot]$.

Answers to these questions depend on low-level details of the implementation and on the precise definition of the assumed model of RAM computations. While optimization of the $k^{\mathcal{O}(1)}$ factor is an important and a nontrivial problem in practice, this issue is beyond the theoretical scope of this book. For this reason, we adopt a pragmatic policy of not stating such polynomial factors explicitly for dynamic-programming routines. In the bibliographic notes we provide some further references to the literature that considers this issue.

7.3.2 Dominating Set

Our next example is the Dominating Set problem. Recall that a set of vertices D is a dominating set in graph G if $V(G) = N[D]$. The goal is to provide a dynamic-programming algorithm on a tree decomposition that determines the minimum possible size of a dominating set of the input graph G. Again, n will denote the number of vertices of the input graph G, and k is an upper bound on the width of the given tree decomposition $(T, \{X_t\}_{t \in V(T)})$ of G.

In this algorithm we will use a refined variant of nice tree decompositions. In the algorithm of the previous section, for every node t we were essentially interested in the best partial solutions in the graph $G[V_t]$. Thus, whenever a vertex v was introduced in some node t, we simultaneously introduced also all the edges connecting it to other vertices of X_t. We will now add a new type of a node called an *introduce edge node.* This modification enables us to add edges one by one, which often helps in simplifying the description of the algorithm.

Formally, in this extended version of a nice tree decomposition we have both introduce vertex nodes and introduce edge nodes. Introduce vertex nodes correspond to introduce nodes in the standard sense, while introduce edge nodes are defined as follows.

- **Introduce edge node**: a node t, labeled with an edge $uv \in E(G)$ such that $u, v \in X_t$, and with exactly one child t' such that $X_t = X_{t'}$. We say that edge uv is *introduced* at t.

We additionally require that every edge of $E(G)$ is introduced exactly once in the whole decomposition. Leaf nodes, forget nodes and join nodes are defined just as previously. Given a standard nice tree decomposition, it can be easily transformed to this variant as follows. Observe that condition (T3) implies that, in a nice tree decomposition, for every vertex $v \in V(G)$, there exists a unique highest node $t(v)$ such that $v \in X_{t(v)}$; moreover, the parent of $X_{t(v)}$ is a forget node that forgets v. Consider an edge $uv \in E(G)$, and observe that (T2) implies that $t(v)$ is an ancestor of $t(u)$ or $t(u)$ is an ancestor of $t(v)$. Without loss of generality assume the former, and observe that we may insert the introduce edge bag that introduces uv between $t(u)$ and its parent (which forgets u). This transformation, for every edge $uv \in E(G)$, can be easily implemented in time $k^{\mathcal{O}(1)}n$ by a single top-down transversal of the tree decomposition. Moreover, the obtained tree decomposition still has $\mathcal{O}(kn)$ nodes, since a graph of treewidth at most k has at most kn edges (see Exercise 7.15).

With each node t of the tree decomposition we associate a subgraph G_t of G defined as follows:

$$G_t = \Big(V_t, E_t = \{e \ : \ e \text{ is introduced in the subtree rooted at } t\}\Big).$$

While in the previous section the subproblems for t were defined on the graph $G[V_t]$, now we will define subproblems for the graph G_t.

We are ready to define the subproblems formally. For Weighted Independent Set, we were computing partial solutions according to how a maximum weight independent set intersects a bag of the tree decomposition. For domination the situation is more complicated. Here we have to distinguish not only if a vertex is in the dominating set or not, but also if it is dominated. A *coloring* of bag X_t is a mapping $f\colon X_t \to \{0, \hat{0}, 1\}$ assigning three different colors to vertices of the bag.

- **Black**, represented by 1. The meaning is that all black vertices have to be contained in the partial solution in G_t.
- **White**, represented by 0. The meaning is that all white vertices are not contained in the partial solution and must be dominated by it.
- **Grey**, represented by $\hat{0}$. The meaning is that all grey vertices are not contained in the partial solution, but do not have to be dominated by it.

The reason why we need to distinguish between white and grey vertices is that some vertices of a bag can be dominated by vertices or via edges which are not introduced so far. Therefore, we also need to consider subproblems where some vertices of the bag are not required to be dominated, since such subproblems can be essential for constructing the optimum solution. Let us stress the fact that we do not forbid grey vertices to be dominated — we just do not care whether they are dominated or not.

For a node t, there are $3^{|X_t|}$ colorings of X_t; these colorings form the space of states at node t. For a coloring f of X_t, we denote by $c[t, f]$ the minimum size of a set $D \subseteq V_t$ such that

- $D \cap X_t = f^{-1}(1)$, which is the set of vertices of X_t colored black.
- Every vertex of $V_t \setminus f^{-1}(\hat{0})$ either is in D or is adjacent in G_t to a vertex of D. That is, D dominates all vertices of V_t in graph G_t, except possibly some grey vertices in X_t.

We call such a set D a *minimum compatible set* for t and f. If no minimum compatible set for t and f exists, we put $c[t, f] = +\infty$. Note that the size of a minimum dominating set in G is exactly the value of $c[r, \emptyset]$ where r is the root of the tree decomposition. This is because we have $G = G_r$ and $X_r = \emptyset$, which means that for X_r we have only one coloring: the empty function.

It will be convenient to use the following notation. For a subset $X \subseteq V(G)$, consider a coloring $f : X \to \{0, \hat{0}, 1\}$. For a vertex $v \in V(G)$ and a color $\alpha \in \{0, \hat{0}, 1\}$ we define a new coloring $f_{v\to\alpha} : X \cup \{v\} \to \{0, \hat{0}, 1\}$ as follows:

$$f_{v\to\alpha}(x) = \begin{cases} f(x) & \text{when } x \neq v, \\ \alpha & \text{when } x = v. \end{cases}$$

For a coloring f of X and $Y \subseteq X$, we use $f|_Y$ to denote the restriction of f to Y.

We now proceed to present the recursive formulas for the values of c. As we mentioned in the previous section, we give only the formulas and short arguments for their correctness, leaving the full proof to the reader.

Leaf node. For a leaf node t we have that $X_t = \emptyset$. Hence there is only one, empty coloring, and we have $c[t, \emptyset] = 0$.

Introduce vertex node. Let t be an introduce node with a child t' such that $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$. Since this node does not introduce edges to G_t, the computation will be very simple — v is isolated in G_t. Hence, we just need to be sure that we do not introduce an isolated white vertex, since then we have $c[t, f] = +\infty$. That is, for every coloring f of X_t we can put

$$c[t,f] = \begin{cases} +\infty & \text{when } f(v) = 0, \\ c[t', f|_{X_{t'}}] & \text{when } f(v) = \hat{0}, \\ 1 + c[t', f|_{X_{t'}}] & \text{when } f(v) = 1. \end{cases}$$

Introduce edge node. Let t be an introduce edge node labeled with an edge uv and let t' be the child of t. Let f be a coloring of X_t. Then sets D compatible for t and f should be almost exactly the sets that are compatible for t' and f, apart from the fact that the edge uv can additionally help in domination. That is, if f colors u black and v white, then when taking the precomputed solution for t' we can relax the color of v from white to grey — in the solution for t' we do not need to require any domination constraint on v, since v will get dominated by u anyways. The same conclusion can be drawn when u is colored white and v is colored black. Therefore, we have the following formulas:

$$c[t,f] = \begin{cases} c[t', f_{v \to \hat{0}}] & \text{when } (f(u), f(v)) = (1, 0), \\ c[t', f_{u \to \hat{0}}] & \text{when } (f(u), f(v)) = (0, 1), \\ c[t', f] & \text{otherwise.} \end{cases}$$

Forget node. Let t be a forget node with a child t' such that $X_t = X_{t'} \setminus \{w\}$ for some $w \in X_{t'}$. Note that the definition of compatible sets for t and f requires that vertex w be dominated, so every set D compatible for t and f is also compatible for t' and $f_{w \to 1}$ (if $w \in D$) or $f_{w \to 0}$ (if $w \notin D$). On the other hand, every set compatible for t' and any of these two colorings is also compatible for t and f. This justifies the following recursive formula:

$$c[t,f] = \min \left\{ c[t', f_{w \to 1}], c[t', f_{w \to 0}] \right\}.$$

Join node. Let t be a join node with children t_1 and t_2. Recall that $X_t = X_{t_1} = X_{t_2}$. We say that colorings f_1 of X_{t_1} and f_2 of X_{t_2} are *consistent* with a coloring f of X_t if for every $v \in X_t$ the following conditions hold

(*i*) $f(v) = 1$ if and only if $f_1(v) = f_2(v) = 1$,
(*ii*) $f(v) = 0$ if and only if $(f_1(v), f_2(v)) \in \{(\hat{0}, 0), (0, \hat{0})\}$,
(*iii*) $f(v) = \hat{0}$ if and only if $f_1(v) = f_2(v) = \hat{0}$.

On one hand, if D is a compatible set for f and t, then $D_1 := D \cap V_{t_1}$ and $D_2 := D \cap V_{t_2}$ are compatible sets for t_1 and f_1, and t_2 and f_2, for some colorings f_1, f_2 that are consistent with f. Namely, for every vertex v that is white in f we make it white either in f_1 or in f_2, depending on whether it is dominated by D_1 in G_{t_1} or by D_2 in G_{t_2} (if v is dominated by both D_1 and D_2, then both options are correct). On the other hand, if D_1 is compatible for t_1 and f_1 and D_2 is compatible for t_2 and f_2, for some colorings f_1, f_2 that are consistent with f, then it is easy to see that $D := D_1 \cup D_2$ is compatible for t and f. Since for such D_1, D_2 we have that $D \cap X_t = D_1 \cap X_{t_1} = D_2 \cap X_{t_2} = f^{-1}(1)$, it follows that $|D| = |D_1| + |D_2| - |f^{-1}(1)|$. Consequently, we can infer the following recursive formula:

$$c[t, f] = \min_{f_1, f_2} \left\{ c[t_1, f_1] + c[t_2, f_2] - |f^{-1}(1)| \right\}, \tag{7.12}$$

where the minimum is taken over all colorings f_1, f_2 consistent with f.

This finishes the description of the recursive formulas for the values of c. Let us analyze the running time of the algorithm. Clearly, the time needed to process each leaf node, introduce vertex/edge node or forget node is $3^k \cdot k^{\mathcal{O}(1)}$, providing that we again use the data structure for adjacency queries of Exercise 7.16. However, computing the values of c in a join node is more time consuming. The computation can be implemented as follows. Note that if a pair f_1, f_2 is consistent with f, then for every $v \in X_t$ we have

$$(f(v), f_1(v), f_2(v)) \in \{(1, 1, 1), (0, 0, \hat{0}), (0, \hat{0}, 0), (\hat{0}, \hat{0}, \hat{0})\}.$$

It follows that there are exactly $4^{|X_t|}$ triples of colorings (f, f_1, f_2) such that f_1 and f_2 are consistent with f, since for every vertex v we have four possibilities for $(f(v), f_1(v), f_2(v))$. We iterate through all these triples, and for each triple (f, f_1, f_2) we include the contribution from f_1, f_2 to the value of $c[t, f]$ according to (7.12). In other words, we first put $c[t, f] = +\infty$ for all colorings f of X_t, and then for every considered triple (f, f_1, f_2) we replace the current value of $c[t, f]$ with $c[t_1, f_1] + c[t_2, f_2] - |f^{-1}(1)|$ in case the latter value is smaller. As $|X_t| \le k + 1$, it follows that the algorithm spends $4^k \cdot k^{\mathcal{O}(1)}$ time for every join node. Since we assume that the number of nodes in a nice tree decomposition is $\mathcal{O}(kn)$ (see Exercise 7.15), we derive the following theorem.

Theorem 7.7. *Let G be an n-vertex graph given together with its tree decomposition of width at most k. Then one can solve the* Dominating Set *problem in G in time $4^k \cdot k^{\mathcal{O}(1)} \cdot n$.*

In Chapter 11, we show how one can use more clever techniques in the computations for the join node in order to reduce the exponential dependence on the treewidth from 4^k to 3^k.

7.3.3 Steiner Tree

Our last example is the Steiner Tree problem. We are given an undirected graph G and a set of vertices $K \subseteq V(G)$, called *terminals*. The goal is to find a subtree H of G of the minimum possible size (that is, with the minimum possible number of edges) that connects all the terminals. Again, assume that $n = |V(G)|$ and that we are given a nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G of width at most k. We will use the same variant of a nice tree decomposition as in Section 7.3.2, that is, with introduce edge nodes.

While the exponential dependence on the treewidth in the algorithms we discussed so far is single-exponential, for Steiner Tree the situation will be different. In what follows, we give an algorithm with running time $k^{\mathcal{O}(k)} \cdot n$. While a single-exponential algorithm for Steiner Tree actually exists, it requires more advanced techniques that will be discussed in Chapter 11.

In order to make the description of the algorithm simpler, we will make one small adjustment to the given decomposition. Let us pick an arbitrary terminal $u^\star \in K$ and let us add it to every bag of the decomposition $\mathcal{T}$. Then the width of $\mathcal{T}$ increases by at most 1, and bags at the root and at all the leaves are equal to $\{u^\star\}$. The idea behind this simple tweak is to make sure that every bag of $\mathcal{T}$ contains at least one terminal. This will be helpful in the definition of the state of the dynamic program.

Let H be a Steiner tree connecting K and let t be a node of $\mathcal{T}$. The part of H contained in G_t is a forest F with several connected components, see Fig. 7.3. Note that this part is never empty, because X_t contains at least one terminal. Observe that, since H is connected and X_t contains a terminal, each connected component of F intersects X_t. Moreover, every terminal from $K \cap V_t$ should belong to some connected component of F. We try to encode all this information by keeping, for each subset $X \subseteq X_t$ and each partition $\mathcal{P}$ of X, the minimum size of a forest F in G_t such that

(a) $K \cap V_t \subseteq V(F)$, i.e., F spans all terminals from V_t,
(b) $V(F) \cap X_t = X$, and
(c) the intersections of X_t with vertex sets of connected components of F form exactly the partition $\mathcal{P}$ of X.

When we introduce a new vertex or join partial solution (at join nodes), the connected components of partial solutions could merge and thus we need to keep track of the updated partition into connected components.

More precisely, we introduce the following function. For a bag X_t, a set $X \subseteq X_t$ (a set of vertices touched by a Steiner tree), and a partition $\mathcal{P} = \{P_1, P_2, \dots, P_q\}$ of X, the value $c[t, X, \mathcal{P}]$ is the minimum possible number of edges of a forest F in G_t such that:

- F has exactly q connected components that can be ordered as $C_1, \dots, C_q$ so that $P_s = V(C_s) \cap X_t$ for each $s \in \{1, \dots, q\}$. Thus the partition $\mathcal{P}$ corresponds to connected components of F.

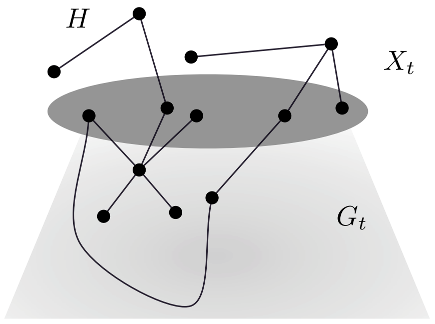


Fig. 7.3: Steiner tree H intersecting bag X_t and graph G_t

- $X_t \cap V(F) = X$. That is, vertices of $X_t \setminus X$ are untouched by F.
- Every terminal vertex from $K \cap V_t$ is in $V(F)$.

A forest F conforming to this definition will be called *compatible* for $(t, X, \mathcal{P})$. If no compatible forest F exists, we put $c[t, X, \mathcal{P}] = +\infty$.

Note that the size of an optimum Steiner tree is exactly $c[r, \{u^\star\}, \{\{u^\star\}\}]$, where r is the root of the decomposition $\mathcal{T}$. This is because we have that $X_r = \{u^\star\}$. We now provide recursive formulas to compute the values of c.

Leaf node. If t is a leaf node, then $X_t = \{u^\star\}$. Since $u^\star \in K$, we have $c[t, \emptyset, \emptyset] = +\infty$ and $c[t, \{u^\star\}, \{\{u^\star\}\}] = 0$.

Introduce vertex node. Suppose that t is an introduce vertex node with a child t' such that $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$. Recall that we have *not introduced* any edges adjacent to v so far, so v is isolated in G_t. Hence, for every set $X \subseteq X_t$ and partition $\mathcal{P} = \{P_1, P_2, \ldots, P_q\}$ of X we do the following. If v is a terminal, then it has to be in X. Moreover, if v is in X, then $\{v\}$ should be a block of $\mathcal{P}$, that is, v should be in its own connected component. If any of these conditions is not satisfied, we put $c[t, X, \mathcal{P}] = +\infty$. Otherwise we have the following recursive formula:

$$c[t, X, \mathcal{P}] = \begin{cases} c[t', X \setminus \{v\}, \mathcal{P} \setminus \{\{v\}\}] & \text{if } v \in X, \\ c[t', X, \mathcal{P}] & \text{otherwise.} \end{cases}$$

Introduce edge node. Suppose that t is an introduce edge node that introduces an edge uv, and let t' be the child of t. For every set $X \subseteq X_t$ and partition $\mathcal{P} = \{P_1, P_2, \ldots, P_q\}$ of X we consider three cases. If $u \notin X$ or $v \notin X$, then we cannot include uv into the tree under construction, as one of its endpoints has been already determined not to be touched by the tree. Hence in this case $c[t, X, \mathcal{P}] = c[t', X, \mathcal{P}]$. The same happens if u and v are both in X, but are not in the same block of $\mathcal{P}$. Assume then that u and v are

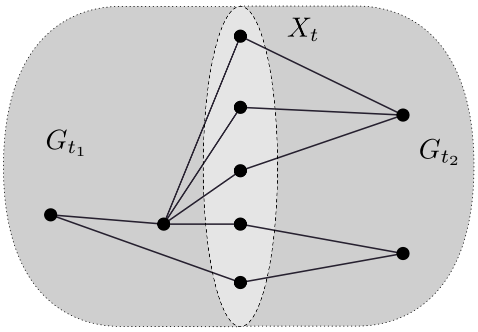


Fig. 7.4: Undesirable merge of partial solutions in a join node

both in X and they actually are in the same block of $\mathcal{P}$. Then the edge uv either has been picked to the solution, or has not been picked. If not, then we should look at the same partition $\mathcal{P}$ at t', and otherwise the block of u and v in $\mathcal{P}$ should have been obtained from merging two smaller blocks, one containing u and the second containing v. Hence

$$c[t, X, \mathcal{P}] = \min\left\{\min_{\mathcal{P}'} c[t', X, \mathcal{P}'] + 1, c[t', X, \mathcal{P}]\right\},$$

where in the inner minimum we consider all partitions $\mathcal{P}'$ of X in which u and v are in separate blocks (as otherwise adding uv would create a cycle) such that after merging the blocks of u and v we obtain the partition $\mathcal{P}$.

Forget node. Suppose that t is a forget node with a child t' such that $X_t = X_{t'} \setminus \{w\}$ for some $w \in X_{t'}$. Consider any set $X \subseteq X_t$ and partition $\mathcal{P} = \{P_1, P_2, \ldots, P_q\}$ of X. The solution for X and $\mathcal{P}$ might either use the vertex w, in which case it should be added to one of existing blocks of $\mathcal{P}$, or not use it, in which case we should simply look on the same partition of the same X. Hence

$$c[t, X, \mathcal{P}] = \min\left\{\min_{\mathcal{P}'} c[t', X \cup \{w\}, \mathcal{P}'], c[t', X, \mathcal{P}]\right\},$$

where the inner minimum is taken over all partitions $\mathcal{P}'$ of $X \cup \{w\}$ that are obtained from $\mathcal{P}$ by adding w to one of the existing blocks.

Join node. Suppose t is a join node with children t_1 and t_2. Recall that then $X_t = X_{t_1} = X_{t_2}$. In essence, in the computation for t we need to encode merging two partial solutions: one originating from G_{t_1} and the second originating from G_{t_2}. When merging two partial solutions, however, we have to be careful because such a merge can create cycles, see Fig. 7.4.

To avoid cycles while merging, we introduce an auxiliary structure.[1] For a partition $\mathcal{P}$ of X let $G_{\mathcal{P}}$ be a forest with a vertex set X, such that the set of connected components in $G_{\mathcal{P}}$ corresponds exactly to $\mathcal{P}$. In other words, for each block of $\mathcal{P}$ there is a tree in $G_{\mathcal{P}}$ with the same vertex set. We say that a partition $\mathcal{P} = \{P_1, P_2, \ldots, P_q\}$ of X is an *acyclic merge* of partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ if the merge of two forests $G_{\mathcal{P}_1}$ and $G_{\mathcal{P}_2}$ (treated as a multigraph) is a forest whose family of connected components is exactly $\mathcal{P}$.

Thus we have the following formula:

$$c[t, X, \mathcal{P}] = \min_{\mathcal{P}_1, \mathcal{P}_2} c[t_1, X, \mathcal{P}_1] + c[t_2, X, \mathcal{P}_2],$$

where in the minimum we consider all pairs of partitions $\mathcal{P}_1, \mathcal{P}_2$ such that $\mathcal{P}$ is an acyclic merge of them.

This concludes the description of the recursive formulas for the values of c. We proceed to estimate the running time. Recall that every bag of the decomposition has size at most $k+2$. Hence, the number of states per node is at most $2^{k+2} \cdot (k+2)^{k+2} = k^{\mathcal{O}(k)}$, since for a node t there are $2^{|X_t|}$ subsets $X \subseteq X_t$ and at most $|X|^{|X|}$ partitions of X. The computation of a value for every state requires considering at most all the pairs of states for some other nodes, which means that each value can be computed in time $(k^{\mathcal{O}(k)})^2 = k^{\mathcal{O}(k)}$. Thus, up to a factor polynomial in k, which is anyhow dominated by the $\mathcal{O}$-notation in the exponent, for every node the running time of computing the values of c is $k^{\mathcal{O}(k)}$. We conclude with the following theorem.

Theorem 7.8. *Let G be an n-vertex graph, let $K \subseteq V(G)$ be a given set of terminals, and assume that G is given together with its tree decomposition of width at most k. Then one can find the minimum possible number of edges of a Steiner tree connecting K in time $k^{\mathcal{O}(k)} \cdot n$.*

Algorithms similar to the dynamic programming of Theorem 7.8 can be used to solve many problems in time $k^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$. Essentially, such a running time appears for problems with connectivity requirements, since then it is natural to keep in the dynamic programming state a partition of a subset of the bag. Since the number of such partitions is at most $k^{\mathcal{O}(k)}$, this factor appears naturally in the running time.

For convenience, we now state formally the most prominent problems that can be solved in *single-exponential time* when parameterized by treewidth (i.e., in time $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$), and those that can be solved in *slightly super-exponential time* (i.e., $k^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$). We leave designing the remaining algorithms from the following theorems as two exercises: Exercise 7.18 and Exercise 7.19.

[1] One could avoid the cycle detection by relaxing the definition of $c[t, X, \mathcal{P}]$ and dropping the assumption that F is a forest. However, as in other problems, like Feedback Vertex Set, cycle checking is essential, we show a common solution here.

Let us recall that the MaxCut problem asks for a partition of $V(G)$ into sets A and B such that the number of edges between A and B is maximized. The q-Coloring problem asks whether G can be properly colored using q colors, while in Chromatic Number the question is to find the minimum possible number of colors needed to properly color G. Longest Path and Longest Cycle ask for the existence of a path/cycle on at least ℓ vertices in G, for a given integer ℓ. Similarly, in Cycle Packing the question is whether one can find ℓ vertex-disjoint cycles in G. Problems Connected Vertex Cover, Connected Dominating Set, Connected Feedback Vertex Set differ from their standard (non-connected) variants by additionally requiring that the solution vertex cover/dominating set/feedback vertex set induces a connected graph in G.

Theorem 7.9. *Let G be an n-vertex graph given together with its tree decomposition of width at most k. Then in G one can solve*

- Vertex Cover *and* Independent Set *in time* $2^k \cdot k^{\mathcal{O}(1)} \cdot n$,
- Dominating Set *in time* $4^k \cdot k^{\mathcal{O}(1)} \cdot n$,
- Odd Cycle Transversal *in time* $3^k \cdot k^{\mathcal{O}(1)} \cdot n$,
- MaxCut *in time* $2^k \cdot k^{\mathcal{O}(1)} \cdot n$,
- q-Coloring *in time* $q^k \cdot k^{\mathcal{O}(1)} \cdot n$.

Theorem 7.10. *Let G be an n-vertex graph given together with its tree decomposition of width at most k. Then one can solve each of the following problems in G in time $k^{\mathcal{O}(k)} \cdot n$:*

- Steiner Tree,
- Feedback Vertex Set,
- Hamiltonian Path *and* Longest Path,
- Hamiltonian Cycle *and* Longest Cycle,
- Chromatic Number,
- Cycle Packing,
- Connected Vertex Cover,
- Connected Dominating Set,
- Connected Feedback Vertex Set.

Let us also note that the dependence on k for many of the problems listed in Theorem 7.10 will be improved to single-exponential in Chapter 11. Also, the running time of the algorithm for Dominating Set will be improved from $4^k \cdot k^{\mathcal{O}(1)} \cdot n$ to $3^k \cdot k^{\mathcal{O}(1)} \cdot n$.

As the reader probably observed, dynamic-programming algorithms on graphs of bounded treewidth are very similar to each other.

The main challenge for most of the problems is to understand what information to store at nodes of the tree decomposition. Obtaining formulas for forget, introduce and join nodes can be a tedious task, but is usually straightforward once a precise definition of a state is established.

It is also worth giving an example of a problem which cannot be solved efficiently on graphs of small treewidth. In the STEINER FOREST problem we are given a graph G and a set of pairs $(s_1, t_1), \ldots, (s_p, t_p)$. The task is to find a minimum subgraph F of G such that each of the pairs is connected in F. STEINER TREE is a special case of STEINER FOREST with $s_1 = \cdots = s_p$. The intuition behind why a standard dynamic programming approach on graphs of constant treewidth does not work for this problem is as follows. Suppose that we have a set of vertices $S = \{s_i\}_{1 \le i \le p}$ separated from vertices $T = \{t_i\}_{1 \le i \le p}$ by a bag of tree decomposition of size 2. Since all paths from S to T must pass through the bag of size 2, the final solution F contains at most two connected components. Any partial solution divides the vertices of S into two parts based on which vertex in the separating bag they are connected to. Thus it seems that to keep track of all partial solutions for the problem, we have to compute all possible ways the set S can be partitioned into two subsets corresponding to connected components of F, which is 2^p. Since p does not depend on the treewidth of G, we are in trouble. In fact, this intuition is supported by the fact that STEINER FOREST is NP-hard on graphs of treewidth at most 3, see [26]. In Section 13.6, we also give examples of problems which are W[1]-hard parameterized by the treewidth of the input graph.

Finding faster dynamic-programming strategies can be an interesting challenge and we will discuss several nontrivial examples of such algorithms in Chapter 11. However, if one is not particularly interested in obtaining the best possible running time, then there exist meta-techniques for designing dynamic-programming algorithms on tree decompositions. These tools we discuss in the next section.

7.4 Treewidth and monadic second-order logic

As we have seen in the previous sections, many optimization problems are fixed-parameter tractable when parameterized by the treewidth. Algorithms for this parameterization are in the vast majority derived using the paradigm of dynamic programming. One needs to understand how to succinctly represent necessary information about a subtree of the decomposition in a dynamic-programming table. If the size of this table turns out to be bounded by a function of the treewidth only, possibly with some polynomial factors in the total graph size, then there is hope that a bottom-up dynamic-programming procedure can compute the complete answer to the problem. In other words, the final outcome can be obtained by consecutively assembling information about the behavior of the problem in larger and larger subtrees of the decomposition.

The standard approach to formalizing this concept is via *tree automata*. This leads to a deep theory linking logic on graphs, tree automata, and treewidth. In this section, we touch only the surface of this subject by highlighting the most important results, namely Courcelle's theorem and its optimization variant.

Intuitively, Courcelle's theorem provides a unified description of properties of a problem that make it amenable to dynamic programming over a tree decomposition. This description comes via a form of a logical formalism called *Monadic Second-Order logic on graphs*. Slightly more precisely, the theorem and its variants state that problems expressible in this formalism are always fixed-parameter tractable when parameterized by treewidth. Before we proceed to stating Courcelle's theorem formally, we need to understand first how Monadic Second-Order logic on graphs works.

7.4.1 Monadic second-order logic on graphs

$\mathbf{MSO_2}$ for dummies. The logic we are about to introduce is called $\mathbf{MSO_2}$. Instead of providing immediately the formal description of this logic, we first give an example of an $\mathbf{MSO_2}$ formula in order to work out the main concepts. Consider the following formula $\mathbf{conn}(X)$, which verifies that a subset X of vertices of a graph $G = (V, E)$ induces a connected subgraph.

$$\begin{aligned}\mathbf{conn}(X) = \quad & \forall_{Y \subseteq V} [(\exists_{u \in X}\, u \in Y \wedge \exists_{v \in X}\, v \notin Y) \\ & \Rightarrow (\exists_{e \in E}\, \exists_{u \in X}\, \exists_{v \in X}\, \mathbf{inc}(u, e) \wedge \mathbf{inc}(v, e) \wedge u \in Y \wedge v \notin Y)].\end{aligned}$$

Now, we rewrite this formula in English.

For every subset of vertices Y, if X contains both a vertex from Y and a vertex outside of Y, then there exists an edge e whose endpoints u, v both belong to X, but one of them is in Y and the other is outside of Y.

One can easily see that this condition is equivalent to the connectivity of $G[X]$: the vertex set of G cannot be partitioned into Y and $V(G) \setminus Y$ in such a manner that X is partitioned nontrivially and no edge of $G[X]$ crosses the partition.

As we see on this example, $\mathbf{MSO_2}$ is a formal language of expressing properties of graphs and objects inside these graphs, such as vertices, edges, or subsets of them. A formula φ of $\mathbf{MSO_2}$ is nothing else but a string over some mysterious symbols, which we shall decode in the next few paragraphs.

One may think that a formula defines a *program* that can be run on an input graph, similarly as, say, a `C++` program can be run on some text input.[2] A `C++` program is just a sequence of instructions following some syntax, and an $\mathbf{MSO}_2$ formula is just a sequence of symbols constructed using a specified set of rules. A `C++` program can be run on multiple different inputs, and may provide different results of the computation. Similarly, an $\mathbf{MSO}_2$ formula may be evaluated in different graphs, and it can give different outcomes. More precisely, an $\mathbf{MSO}_2$ formula can be *true* in a graph, or *false*. The result of an application of a formula to a graph will be called the *evaluation* of the formula in the graph.

Similarly to `C++` programs, $\mathbf{MSO}_2$ formulas have *variables* which represent different objects in the graph. Generally, we shall have four *types* of variables: variables for single vertices, for single edges, for subsets of vertices, and for subsets of edges; the last type was not used in formula $\mathbf{conn}(X)$. At each point of the process of evaluation of the formula, every variable is *evaluated* to some object of appropriate type.

Note that a formula can have "parameters": variables that are given from "outside", whose properties we verify in the graph. In the $\mathbf{conn}(X)$ example such a parameter is X, the vertex subset whose connectivity is being tested. Such variables will be called *free variables* of the formula. Note that in order to properly evaluate the formula in a graph, we need to be given the evaluation of these variables. Most often, we will assume that the input graph is *equipped* with evaluation of all the free variables of the considered $\mathbf{MSO}_2$ formula, which means that these evaluations are provided together with the graph.

If we already have some variables in the formula, we can test their mutual interaction. As we have seen in the $\mathbf{conn}(X)$ example, we can for instance check whether some vertex u belongs to some vertex subset Y ($u \in Y$), or whether an edge e is incident to a vertex u ($\mathbf{inc}(u,e)$). These checks can be combined using standard Boolean operators such as $\neg$ (negation, logical NOT), $\wedge$ (conjunction, logical AND), $\vee$ (disjunction, logical OR), $\Rightarrow$ (implication).

The crucial concept that makes $\mathbf{MSO}_2$ useful for expressing graph properties are *quantifiers*. They can be seen as counterparts of *loops* in standard programming languages. We have two types of quantifiers, $\forall$ and $\exists$. Each quantifier is applied to some *subformula* ψ, which in the programming language analogy is just a block of code bound by the loop. Moreover, every quantifier introduces a new variable over which it iterates. This variable can be then used in the subformula.

Quantifier $\forall$ is called the *universal quantifier*. Suppose we write a formula $\forall_{v \in V}\, \psi$, where ψ is some subformula that uses variable v. This formula should be then read as "For every vertex v in the graph, ψ holds." In other words, quantifier $\forall_{v \in V}$ iterates through all possible evaluations of variable v to a vertex of the graph, and for each of them it is checked whether ψ is indeed

[2] For a reader familiar with the paradigm of *functional programming* (languages like *Lisp* or *Haskell*), an analogy with any functional language would be more appropriate.

true. If this is the case for *every* evaluation of v, then the whole formula $\forall_{v \in V}\, \psi$ is true; otherwise it is false.

Quantifier $\exists$, called the *existential quantifier*, works sort of similarly. Formula $\exists_{v \in V}\, \psi$ should be read as "There exists a vertex v in the graph, such that ψ holds." This means that $\exists_{v \in V}$ iterates through all possible evaluations of variable v to a vertex of the graph, and verifies whether there is at least one for which ψ is true.

Of course, here we just showed examples of quantification over variables for single vertices, but we can also quantify over variables for single edges (e.g., $\forall_{e \in E}/\exists_{e \in E}$), vertex subsets (e.g., $\forall_{X \subseteq V}/\exists_{X \subseteq V}$), or edge subsets (e.g., $\forall_{C \subseteq E}/\exists_{C \subseteq E}$). Standard Boolean operators can be also used to combine larger formulas; see for instance our use of the implication in formula $\mathbf{conn}(X)$.

We hope that the reader already understands the basic idea of $\mathbf{MSO_2}$ as a (programming) language for expressing graph properties. We now proceed to explaining formally the syntax of $\mathbf{MSO_2}$ (how formulas can be constructed), and the semantics (how formulas are evaluated). Fortunately, they are much simpler than for `C++`.

Syntax and semantics of $\mathbf{MSO_2}$. Formulas of $\mathbf{MSO_2}$ can use four types of variables: for single vertices, single edges, subsets of vertices, and subsets of edges. The subscript 2 in $\mathbf{MSO_2}$ exactly signifies that quantification over edge subsets is also allowed. If we forbid this type of quantification, we arrive at a weaker logic $\mathbf{MSO_1}$. The vertex/edge subset variables are called *monadic* variables.

Every formula φ of $\mathbf{MSO_2}$ can have free variables, which often will be written in parentheses besides the formula. More precisely, whenever we write a formula of $\mathbf{MSO_2}$, we should always keep in mind what variables are assumed to be existent in the context in which this formula will be used. The sequence of these variables is called the *signature* over which the formula is written;[3] following our programming language analogy, this is the environment in which the formula is being defined. Then variables from the signature can be used in φ as free variables. The signature will be denoted by Σ. Note that for every variable in the signature we need to know what is its type.

In order to evaluate a formula φ over signature Σ in a graph G, we need to know how the variables of Σ are evaluated in G. By Σ^G we will denote the sequence of evaluations of variables from Σ. Evaluation of a single variable x will be denoted by x^G. Graph G and Σ^G together shall be called the *structure* in which φ is being evaluated. If φ is true in structure $\langle G, \Sigma^G \rangle$, then we shall denote it by

$$\langle G, \Sigma^G \rangle \models \varphi,$$

[3] A reader familiar with the foundations of logic in computer science will certainly see that we are slightly tweaking some definitions so that they fit to our small example. We do it in order to simplify the description of $\mathbf{MSO_2}$ while maintaining basic compliance with the general literature. In the bibliographic notes we provide pointers to more rigorous introductions to logic in computer science, where the notions of *signature* and *structure* are introduced properly.

which should be read as “Structure $\langle G, \Sigma^G \rangle$ is a model for φ.”

Formulas of $\mathbf{MSO}_2$ are constructed inductively from smaller subformulas. We first describe the smallest building blocks, called *atomic formulas.*

- If $u \in \Sigma$ is a vertex (edge) variable and $X \in \Sigma$ is a vertex (edge) set variable, then we can write formula $u \in X$. The semantics is standard: the formula is true if and only if $u^G \in X^G$.
- If $u \in \Sigma$ is a vertex variable and $e \in \Sigma$ is an edge variable, then we can write formula $\mathbf{inc}(u, e)$. The semantics is that the formula is true if and only if u^G is an endpoint of e^G.
- For any two variables $x, y \in \Sigma$ of the same type, we can write formula $x = y$. This formula is true in the structure if and only if $x^G = y^G$.

Now that we know the basic building blocks, we can start to create larger formulas. As described before, we can use standard Boolean operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$ working as follows. Suppose that φ_1, φ_2 are two formulas over the same signature Σ. Then we can write the following formulas, also over Σ

- Formula $\neg\varphi_1$, where $\langle G, \Sigma^G \rangle \models \neg\varphi_1$ if and only if $\langle G, \Sigma^G \rangle \not\models \varphi_1$.
- Formula $\varphi_1 \wedge \varphi_2$, where $\langle G, \Sigma^G \rangle \models \varphi_1 \wedge \varphi_2$ if and only if $\langle G, \Sigma^G \rangle \models \varphi_1$ and $\langle G, \Sigma^G \rangle \models \varphi_2$.
- Formula $\varphi_1 \vee \varphi_2$, where $\langle G, \Sigma^G \rangle \models \varphi_1 \vee \varphi_2$ if and only if $\langle G, \Sigma^G \rangle \models \varphi_1$ or $\langle G, \Sigma^G \rangle \models \varphi_2$.
- Formula $\varphi_1 \Rightarrow \varphi_2$, where $\langle G, \Sigma^G \rangle \models \varphi_1 \Rightarrow \varphi_2$ if and only if $\langle G, \Sigma^G \rangle \models \varphi_1$ implies that $\langle G, \Sigma^G \rangle \models \varphi_2$.

Finally, we can use quantifiers. For concreteness, suppose we have a formula ψ over signature Σ' that contains some vertex variable v. Let $\Sigma = \Sigma' \setminus \{v\}$. Then we can write the following formulas over Σ:

- Formula $\varphi_\forall = \forall_{v \in V}\, \psi$. Then $\langle G, \Sigma^G \rangle \models \varphi_\forall$ if and only if *for every* vertex $v^G \in V(G)$, it holds that $\langle G, \Sigma^G, v^G \rangle \models \psi$.
- Formula $\varphi_\exists = \exists_{v \in V}\, \psi$. Then $\langle G, \Sigma^G \rangle \models \varphi_\exists$ if and only if there *exists* a vertex $v^G \in V(G)$ such that $\langle G, \Sigma^G, v^G \rangle \models \psi$.

Similarly, we can perform quantification over variables for single edges ($\forall_{e \in E}/\exists_{e \in E}$), vertex subsets ($\forall_{X \subseteq V}/\exists_{X \subseteq V}$), and edge subsets ($\forall_{C \subseteq E}/\exists_{C \subseteq E}$). The semantics is defined analogously.

Observe that in formula $\mathbf{conn}(X)$ we used a couple of notation “hacks” that simplified the formula, but were formally not compliant to the syntax described above. We namely allow some shorthands to streamline writing formulas. Firstly, we allow simple shortcuts in the quantifiers. For instance, $\exists_{v \in X}\, \psi$ is equivalent to $\exists_{v \in V}\, (v \in X) \wedge \psi$ and $\forall_{v \in X}\, \psi$ is equivalent to $\forall_{v \in V}\, (v \in X) \Rightarrow \psi$. We can also merge a number of similar quantifiers into one, e.g., $\exists_{X_1, X_2 \subseteq V}$ is the same as $\exists_{X_1 \subseteq V} \exists_{X_2 \subseteq V}$. Another construct that we can use is the subset relation $X \subseteq Y$: it can be expressed as $\forall_{v \in V} (v \in X) \Rightarrow (v \in Y)$, and similarly for edge subsets. We can also express the adjacency relation between two vertex variables: $\mathbf{adj}(u, v) = (u \neq v) \wedge (\exists_{e \in E}\, \mathbf{inc}(u, e) \wedge \mathbf{inc}(v, e))$.

Finally, we use $x \neq y$ for $\neg(x = y)$ and $x \notin X$ for $\neg(x \in X)$. The reader is encouraged to use his or her own shorthands whenever it is beneficial.

Examples. Let us now provide two more complicated examples of graph properties expressible in $\mathbf{MSO_2}$. We have already seen how to express that a subset of vertices induces a connected graph. Let us now look at 3-colorability. To express this property, we need to quantify the existence of three vertex subsets X_1, X_2, X_3 which form a partition of V, and where each of them is an independent set.

$$\begin{aligned}\mathbf{3colorability} = \quad & \exists_{X_1,X_2,X_3 \subseteq V}\, \mathbf{partition}(X_1, X_2, X_3) \wedge \\ & \mathbf{indp}(X_1) \wedge \mathbf{indp}(X_2) \wedge \mathbf{indp}(X_3).\end{aligned}$$

Here, **partition** and **indp** are two auxiliary subformulas. Formula **partition** has three vertex subset variables X_1, X_2, X_3 and verifies that (X_1, X_2, X_3) is a partition of the vertex set V. Formula **indp** verifies that a given subset of vertices is independent.

$$\begin{aligned}\mathbf{partition}(X_1, X_2, X_3) = \quad & \forall_{v \in V}\, [(v \in X_1 \wedge v \notin X_2 \wedge v \notin X_3) \\ & \vee(v \notin X_1 \wedge v \in X_2 \wedge v \notin X_3) \\ & \vee(v \notin X_1 \wedge v \notin X_2 \wedge v \in X_3)];\end{aligned}$$

$$\mathbf{indp}(X) = \quad \forall_{u,v \in X}\, \neg\mathbf{adj}(u, v).$$

Second, let us look at Hamiltonicity: we would like to write a formula that is true in a graph G if and only if G admits a Hamiltonian cycle. For this, let us quantify existentially a subset of edges C that is supposed to comprise the edges of the Hamiltonian cycle we look for. Then we need to verify that (a) C induces a connected graph, and (b) every vertex of V is adjacent to exactly two different edges of C.

$$\mathbf{hamiltonicity} = \quad \exists_{C \subseteq E}\, \mathbf{connE}(C) \wedge \forall_{v \in V}\, \mathbf{deg2}(v, C).$$

Here, $\mathbf{connE}(C)$ is an auxiliary formula that checks whether the graph (V, C) is connected (using similar ideas as for $\mathbf{conn}(X)$), and $\mathbf{deg2}(v, C)$ verifies that vertex v has exactly two adjacent edges belonging to C:

$$\begin{aligned}\mathbf{connE}(C) = \quad & \forall_{Y \subseteq V} \quad [(\exists_{u \in V}\, u \in Y \wedge \exists_{v \in V}\, v \notin Y) \\ & \Rightarrow (\exists_{e \in C}\, \exists_{u \in Y}\, \exists_{v \notin Y}\, \mathbf{inc}(u, e) \wedge \mathbf{inc}(v, e))];\end{aligned}$$

$$\begin{aligned}\mathbf{deg2}(v, C) = \quad & \exists_{e_1, e_2 \in C}\, [(e_1 \neq e_2) \wedge \mathbf{inc}(v, e_1) \wedge \mathbf{inc}(v, e_2) \wedge \\ & (\forall_{e_3 \in C}\, \mathbf{inc}(v, e_3) \Rightarrow (e_1 = e_3 \vee e_2 = e_3))].\end{aligned}$$

7.4.2 Courcelle's theorem

In the following, for a formula φ by $||\varphi||$ we denote the length of the encoding of φ as a string.

Theorem 7.11 (Courcelle's theorem, [98]). *Assume that φ is a formula of* $\mathbf{MSO_2}$ *and G is an n-vertex graph equipped with evaluation of all the free variables of φ. Suppose, moreover, that a tree decomposition of G of width t is provided. Then there exists an algorithm that verifies whether φ is satisfied in G in time $f(||\varphi||, t) \cdot n$, for some computable function f.*

The proof of Courcelle's theorem is beyond the scope of this book, and we refer to other sources for a comprehensive presentation. As we will see later, the requirement that G be given together with its tree decomposition is not necessary, since an optimal tree decomposition of G can be computed within the same complexity bounds. Algorithms computing treewidth will be discussed in the next section and in the bibliographic notes.

Recall that in the previous section we constructed formulas **3colorability** and **hamiltonicity** that are satisfied in G if and only if G is 3-colorable or has a Hamiltonian cycle, respectively. If we now apply Courcelle's theorem to these constant-size formulas, we immediately obtain as a corollary that testing these two properties of graphs is fixed-parameter tractable when parameterized by treewidth.

Let us now focus on the Vertex Cover problem: given a graph G and integer k, we would like to verify whether G admits a vertex cover of size at most k. The natural way of expressing this property in $\mathbf{MSO_2}$ is to quantify existentially k vertex variables, representing vertices of the vertex cover, and then verify that every edge of G has one of the quantified vertices as an endpoint. However, observe that the length of such a formula depends linearly on k. This means that a direct application of Courcelle's theorem gives only an $f(k,t) \cdot n$ algorithm, and not an $f(t) \cdot n$ algorithm as was the case for the dynamic-programming routine of Corollary 7.6. Note that the existence of an $f(k,t) \cdot n$ algorithm is only a very weak conclusion, because as we already have seen in Section 3.1, even the simplest branching algorithm for Vertex Cover runs in time $\mathcal{O}(2^k \cdot (n+m))$.

Therefore, we would rather have the following optimization variant of the theorem. Formula φ has some free monadic (vertex or edge) variables $X_1, X_2, \ldots, X_p$, which correspond to the sets we seek in the graph. In the Vertex Cover example we would have one vertex subset variable X that represents the vertex cover. Formula φ verifies that the variables $X_1, X_2, \ldots, X_p$ satisfy all the requested properties; for instance, that X indeed covers every edge of the graph. Then the problem is to find an evaluation of variables $X_1, X_2, \ldots, X_p$ that minimizes/maximizes the value of some arithmetic expression $\alpha(|X_1|, |X_2|, \ldots, |X_p|)$ depending on the cardinalities of these sets, subject to $\varphi(X_1, X_2, \ldots, X_p)$ being true. We will focus on α

being an *affine function*, that is, $\alpha(x_1, x_2, \ldots, x_p) = a_0 + \sum_{i=1}^{p} a_i x_i$ for some $a_0, a_1, \ldots, a_p \in \mathbb{R}$.

The following theorem states that such an optimization version of Courcelle's theorem indeed holds.

Theorem 7.12 ([19]). *Let φ be an $\mathbf{MSO}_2$ formula with p free monadic variables $X_1, X_2, \ldots, X_p$, and let $\alpha(x_1, x_2, \ldots, x_p)$ be an affine function. Assume that we are given an n-vertex graph G together with its tree decomposition of width t, and suppose G is equipped with evaluation of all the free variables of φ apart from $X_1, X_2, \ldots, X_p$. Then there exists an algorithm that in $f(||\varphi||, t) \cdot n$ finds the minimum and maximum value of $\alpha(|X_1|, |X_2|, \ldots, |X_p|)$ for sets $X_1, X_2, \ldots, X_p$ for which $\varphi(X_1, X_2, \ldots, X_p)$ is true, where f is some computable function.*

To conclude our Vertex Cover example, we can now write a simple constant-length formula $\mathbf{vcover}(X)$ that verifies that X is a vertex cover of G: $\mathbf{vcover}(X) = \forall_{e \in E} \exists_{x \in X} \mathbf{inc}(x, e)$. Then we can apply Theorem 7.12 to **vcover** and $\alpha(|X|) = |X|$, and infer that finding the minimum cardinality of a vertex cover can be done in $f(t) \cdot n$ time, for some function f.

Note that both in Theorem 7.11 and in Theorem 7.12 we allow the formula to have some additional free variables, whose evaluation is provided together with the graph. This feature can be very useful whenever in the considered problem the graph comes together with some predefined objects, e.g., terminals in the Steiner Tree problem. Observe that we can easily write an $\mathbf{MSO}_2$ formula $\mathbf{Steiner}(K, F)$ for a vertex set variable K and edge set variable F, which is true if and only if the edges from F form a Steiner tree connecting K. Then we can apply Theorem 7.12 to minimize the cardinality of F subject to $\mathbf{Steiner}(K, F)$ being true, where the vertex subset K is given together with the input graph. Thus we can basically re-prove Theorem 7.8, however without any explicit bound on the running time.

To conclude, let us deliberate briefly on the function f in the bound on the running time of algorithms provided by Theorems 7.11 and 7.12. Unfortunately, it can be proved that this function has to be nonelementary; in simple words, it cannot by bounded by a folded c times exponential function for any constant c. Generally, the main reason why the running time must be so high is the possibility of having alternating sequences of quantifiers in the formula φ. Slightly more precisely, we can define the *quantifier alternation* of a formula φ to be the maximum length of an alternating sequence of nested quantifiers in φ, i.e., $\forall \exists \forall \exists \ldots$ (we omit some technicalities in this definition). Then it can be argued that formulas of quantifier alternation at most q give rise to algorithms with at most c-times exponential function f, where c depends linearly on q. However, tracing the exact bound on f even for simple formulas φ is generally very hard, and depends on the actual proof of the theorem that is used. This exact bound is also likely to be much higher than optimal. For this reason, Courcelle's theorem and its variants should be regarded primarily as classification tools, whereas designing efficient dynamic-programming routines on tree decompositions requires "getting your hands dirty" and constructing the algorithm explicitly.

7.5 Graph searching, interval and chordal graphs

In this section we discuss alternative interpretations of treewidth and pathwidth, connected to the concept of *graph searching*, and to the classes of *interval* and *chordal* graphs. We also provide some tools for certifying that treewidth and pathwidth of a graph is at least/at most some value. While these results are not directly related to the algorithmic topics discussed in the book, they provide combinatorial intuition about the considered graph parameters that can be helpful when working with them.

Alternative characterizations of pathwidth. We start with the concept of graph searching. Suppose that G is a graph representing a network of tunnels where an agile and omniscient fugitive with unbounded speed is hiding. The network is being searched by a team of searchers, whose goal is to find the fugitive. A *move* of the search team can be one of the following:

- *Placement*: We can take a searcher from the pool of free searchers and place her on some vertex v.
- *Removal*: We can remove a searcher from a vertex v and put her back to the pool of free searchers.

Initially all the searchers are free. A *search program* is a sequence of moves of searchers. While the searchers perform their program, the fugitive, who is not visible to them, can move between the nodes of the network at unbounded speed. She cannot, however, pass through the vertices occupied by the searchers, and is caught if a searcher is placed on a vertex she currently occupies. Searchers win if they launch a search program that guarantees catching the fugitive. The fugitive wins if she can escape the searchers indefinitely.

Another interpretation of the same game is that searchers are cleaning a network of tunnels contaminated with a poisonous gas. Initially all the edges (tunnels) are contaminated, and an edge becomes clear if both its endpoints are occupied by searchers. The gas, however, spreads immediately between edges through vertices that are not occupied by any searcher. If a cleaned edge becomes contaminated in this manner, we say that it is *recontaminated*. The goal of the searchers is to clean the whole network using a search program.

These interpretations can be easily seen to be equivalent,[4] and they lead to the following definition of a graph parameter. The *node search number* of a graph G is the minimum number of searchers required to clean G from gas

[4] Strictly speaking, there is a boring corner case of nonempty edgeless graph, where no gas is present but one needs a searcher to visit one by one all vertices to catch the fugitive. In what follows we ignore this corner case, as all further results consider only a positive number of searchers.

or, equivalently, to catch an invisible fugitive in G. For concreteness, from now on we will work with the gas cleaning interpretation.

An important question in graph searching is the following: if k searchers can clean a graph, can they do it in a *monotone* way, i.e., in such a way that no recontamination occurs? The answer is given by the following theorem of LaPaugh; its proof lies beyond the scope of this book.

Theorem 7.13 ([315]). *For any graph G, if k searchers can clear G, then k searchers can clear G in such a manner that no edge gets recontaminated.*

Let us now proceed to the class of interval graphs. A graph G is an *interval graph* if and only if one can associate with each vertex $v \in V(G)$ a closed interval $I_v = [l_v, r_v]$, $l_v \leq r_v$, on the real line, such that for all $u, v \in V(G)$, $u \neq v$, we have that $uv \in E(G)$ if and only if $I_u \cap I_v \neq \emptyset$. The family of intervals $\mathcal{I} = \{I_v\}_{v \in V}$ is called an *interval representation* or an *interval model* of G. By applying simple transformations to the interval representation at hand, it is easy to see that every interval graph on n vertices has an interval representation in which the left endpoints are distinct integers $1, 2, \ldots, n$. We will call such a representation *canonical*.

Recall that a graph G' is a *supergraph* of a graph G if $V(G) \subseteq V(G')$ and $E(G) \subseteq E(G')$. We define the *interval width* of a graph G as the minimum over all interval supergraphs G' of G of the maximum clique size in G'. That is,

$$\text{interval-width}(G) = \min\left\{\omega(G') \ : \ G \subseteq G' \wedge G' \text{ is an interval graph}\right\}.$$

Here, $\omega(G')$ denotes the maximum size of a clique in G'. In other words, the interval width of G is at most k if and only if there is an interval supergraph of G with the maximum clique size at most k. Note that in this definition we can assume that $V(G') = V(G)$, since the class of interval graphs is closed under taking induced subgraphs.

We are ready to state equivalence of all the introduced interpretations of pathwidth.

Theorem 7.14. *For any graph G and any $k \geq 0$, the following conditions are equivalent:*

(i) *The node search number of G is at most $k+1$.*
(ii) *The interval width of G is at most $k+1$.*
(iii) *The pathwidth of G is at most k.*

Proof. $(i) \Rightarrow (ii)$. Without loss of generality, we can remove from G all isolated vertices. Assume that there exists a search program that cleans G using at most $k+1$ searchers. By Theorem 7.13, we can assume that this search is monotone, i.e., no edge becomes recontaminated. Suppose the program uses p moves; we will naturally index them with $1, 2, \ldots, p$. Let $x_1, x_2, \ldots, x_p$ be the sequence of vertices on which searchers are placed/from which searchers are removed in consecutive moves. Since G has no isolated vertices, each vertex of G has to be occupied at some point (to clear the edges incident to it) and hence it appears in the sequence $x_1, x_2, \ldots, x_p$. Also, without loss of generality we assume that each placed searcher is eventually removed, since at the end we can always remove all the searchers.

We now claim that we can assume that for every vertex $v \in V(G)$, a searcher is placed on v exactly once and removed from v exactly once. Let us look at the first move j when an edge incident to v is cleaned. Note that it does not necessarily hold that $x_j = v$, but vertex v has to be occupied after the j-th move is performed. Let $j_1 \leq j$ be the latest move when a searcher is placed at v before or at move j; in particular $x_{j_1} = v$. Let $j_2 > j$ be the first move when a searcher is removed from v after move j; in particular $x_{j_2} = v$. Since j is the first move when an edge incident to v is cleaned, we have that before j_1 all the edges incident to v are contaminated. Therefore, we can remove from the search program any moves at vertex v that are before j_1, since they actually do not clean anything. On the other hand, observe that at move j_2 all the edges incident to v must be cleaned, since otherwise a recontamination would occur. These edges stay clean to the end of the search, by the monotonicity of our search program. Therefore, we can remove from the search program all the moves at v that occur after j_2, since they actually do not clean anything.

For every vertex v, we define $\ell(v)$ as the first move when a searcher is placed on v, and $r(v)$ as the move when a searcher is removed from v. Now with each vertex v we associate an interval $I_v = [\ell(v), r(v) - 1]$; note that integers contained in I_v are exactly moves after which v is occupied. The intersection graph G_I of intervals $\mathcal{I} = \{I_v\}_{v \in V}$ is, of course, an interval graph. This graph is also a supergraph of G for the following reason. Since every edge uv is cleared, there is always a move j_{uv} after which both u and v are occupied by searchers. Then $j_{uv} \in I_u \cap I_v$, which means that I_u and I_v have nonempty intersection. Finally, the maximum size of a clique in G_I is the maximum number of intervals intersecting in one point, which is at most the number of searchers used in the search.

(ii) $\Rightarrow$ *(iii)*. Let G_I be an interval supergraph of G with the maximum clique size at most $k+1$, and let $\mathcal{I} = \{I_v\}_{v \in V}$ be the canonical representation of G_I. For $i \in \{1, \ldots, n\}$, we define X_i as the set of vertices v of G whose corresponding intervals I_v contain i. All intervals associated with vertices of X_i pairwise intersect at i, which means that X_i is a clique in G. Consequently $|X_i| \leq k + 1$. We claim that $(X_1, \ldots, X_n)$ is a path decomposition of G. Indeed, property (P1) holds trivially. For property (P2), because G_I is a supergraph of G, for every edge $uv \in E(G)$ we have $I_u \cap I_v \neq \emptyset$. Since left endpoints of intervals of $\mathcal{I}$ are $1, 2, \ldots, n$, there exists also some $i \in \{1, 2, \ldots, n\}$ that belongs to $I_u \cap I_v$. Consequently X_i contains both u and v. For property (P3), from the definition of sets X_i it follows that, for every vertex v, the sets containing v form an interval in $\{1, 2, \ldots, n\}$.

(iii) $\Rightarrow$ *(i)*. Let $\mathcal{P} = (X_1, X_2, \ldots, X_r)$ be a path decomposition of G of width at most k. We define the following search program for $k+1$ searchers. First, we place searchers on X_1. Then, iteratively for $i = 2, 3, \ldots, r$, we move searchers from X_{i-1} to X_i by first removing all the searchers from $X_{i-1} \setminus X_i$, and then placing searchers on $X_i \setminus X_{i-1}$. Note that in this manner the searchers are all the time placed on the vertices of $X_{i-1} \cap X_i$. Moreover, since the sizes of bags X_i are upper bounded by $k+1$, we use at most $k+1$ searchers at a time. We now prove that this search program cleans the graph.

By property (P2), for every edge $uv \in E(G)$ there exists a bag X_i such that $\{u, v\} \subseteq X_i$. Hence, every edge gets cleaned at some point. It remains to argue that no recontamination happens during implementation of this search program. For the sake of contradiction, suppose that recontamination happens for the first time when a searcher is removed from some vertex $u \in X_{i-1} \setminus X_i$, for some $i \geq 2$. This means that u has to be incident to some edge uv that is contaminated at this point. Again by the property (P2), there exists some bag X_j such that $\{u, v\} \in X_j$. However, since $u \in X_{i-1}$ and $u \notin X_i$, then by property (P3) u can only appear in bags with indices $1, 2, \ldots, i-1$. Hence $j \leq i-1$, which means that edge uv has been actually already cleaned before, when searchers were placed on the whole bag X_j. Since we assumed that we consider the first time when a recontamination happens, we infer that edge uv must have remained clean up to this point, a contradiction. $\square$

Alternative characterizations of treewidth. A similar set of equivalent characterizations is known for the treewidth. The role of interval graphs is now played by chordal graphs. Let us recall that a graph is *chordal* if it does not contain an induced cycle of length more than 3, i.e., every cycle of length more than 3 has a chord. Sometimes chordal graphs are called triangulated. It is easy to prove that interval graphs are chordal, see Exercise 7.27.

We define the *chordal width* of a graph G as the minimum over all chordal supergraphs G' of G of the maximum clique size in G'. That is,

$$\text{chordal-width}(G) = \min \{\omega(G') \ : \ G \subseteq G' \wedge G' \text{ is a chordal graph}\}.$$

As we will see soon, the chordal width is equal to treewidth (up to a ± 1 summand), but first we need to introduce the notion of graph searching related to the treewidth.

The rules of the search game for treewidth are very similar to the node searching game for pathwidth. Again, a team of searchers, called in this setting *cops*, tries to capture in a graph an agile and omniscient fugitive, called in this setting a *robber*. The main difference is that the cops actually know where the robber is localized. At the beginning of the game cops place themselves at some vertices, and then the robber chooses a vertex to start her escape. The game is played in rounds. Let us imagine that cops are equipped with helicopters. At the beginning of each round, some subset of cops take off in their helicopters, declaring where they will land at the end of the round. While the cops are in the air, the robber may run freely in the graph; however, she cannot pass through vertices that are occupied by cops that are not airborne. After the robber runs to her new location, being always a vertex, the airborne cops land on pre-declared vertices. As before, the cops win if they have a procedure to capture the robber by landing on her location, and the robber wins if she can avoid cops indefinitely.

Intuitively, the main difference is that if cops are placed at some set $S \subseteq V(G)$, then they know in which connected component of $G - S$ the robber resides. Hence, they may concentrate the chase in this component. In the node search game the searchers lack this knowledge, and hence the search program must be oblivious to the location of the fugitive. It is easy to see that when cops do not see the robber, then this reduces to an equivalent variant of the node search game. However, if they see the robber, they can gain a lot. For example, two cops can catch a visible robber on any tree, but catching an invisible fugitive on an n-vertex tree can require $\log_3 n$ searchers; see Exercise 7.10.

Finally, let us introduce the third alternative characterization of treewidth, which is useful in showing lower bounds on this graph parameter. We say that two subsets A and B of $V(G)$ *touch* if either they have a vertex in common, or there is an edge with one endpoint in A and second in B. A *bramble* is the family of pairwise touching connected vertex sets in G. A subset $C \subseteq V(G)$ *covers* a bramble $\mathcal{B}$ if it intersects every element of $\mathcal{B}$. The least number of vertices covering bramble $\mathcal{B}$ is the *order* of $\mathcal{B}$.

It is not difficult to see that if G has a bramble $\mathcal{B}$ of order $k+1$, then k cops cannot catch the robber. Indeed, during the chase the robber maintains the invariant that there is always some $X \in \mathcal{B}$ that is free of cops, and in which she resides. This can be clearly maintained at the beginning of the chase, since for every set of initial position of cops there is an element X of $\mathcal{B}$ that will not contain any cops, which the robber can choose for the start. During every round, the robber examines the positions of all the cops after landing. Again, there must be some $Y \in \mathcal{B}$ that will be disjoint from the set of these positions. If $X = Y$ then the robber does not need to move, and otherwise she runs through X (which is connected and free from cops at this point) to a vertex shared with Y or to an edge connecting X and Y. In this manner the robber can get through to the set Y, which is free from the cops both at this point and after landing.

This reasoning shows that the maximum order of a bramble in a graph G provides a lower bound on the number of cops needed in the search game on G, and thus on the treewidth of G. The following deep result of Seymour and Thomas shows that this lower bound is in fact tight. We state this theorem without a proof.

Theorem 7.15 ([414]). *For every $k \geq 0$ and graph G, the treewidth of G is at least k if and only if G contains a bramble of order at least $k+1$.*

Now we have gathered three alternative characterizations of treewidth, and we can state the analogue of Theorem 7.14 for this graph parameter. The result is given without a proof because of its technicality, but the main steps are discussed in Exercise 7.28.

Theorem 7.16. *For any graph G and any $k \geq 0$, the following conditions are equivalent:*

(*i*) *The treewidth of G is at most k.*
(*ii*) *The chordal width of G is at most $k+1$.*
(*iii*) *$k+1$ cops can catch a visible robber on G.*
(*iv*) *There is no bramble of order larger than $k+1$ in G.*

7.6 Computing treewidth

In all the applications of the treewidth we discussed in this chapter, we were assuming that a tree decomposition of small width is given as part of the input. A natural question is how fast we can find such a decomposition. Unfortunately, it turns out that it is NP-hard to compute the treewidth of a given graph, so we need to resort to FPT or approximation algorithms.

In order to make our considerations more precise, let us consider the Treewidth problem defined as follows: we are given a graph G and an integer k, and the task is to determine whether the treewidth of G is at most k. There is a "simple" argument why Treewidth is fixed-parameter tractable: It follows from the results of the Graph Minors theory of Robertson and Seymour, briefly surveyed in Chapter 6. More precisely, it is easy to see that treewidth is a minor-monotone parameter in the following sense: for every graph G and its minor H, it holds that $\mathrm{tw}(H) \leq \mathrm{tw}(G)$ (see Exercise 7.7). Hence if we define $\mathcal{G}_k$ to be the class of graphs of treewidth at most k, then $\mathcal{G}_k$ is closed under taking minors. By Corollary 6.11, there exists a set of forbidden minors $\mathrm{Forb}(\mathcal{G}_k)$, which depends on k only, such that a graph has treewidth k if and only if it does not contain any graph from $\mathrm{Forb}(\mathcal{G}_k)$ as a minor. Hence, we can apply the algorithm of Theorem 6.12 to each $H \in \mathrm{Forb}(\mathcal{G}_k)$ separately, verifying in $f(H) \cdot |V(G)|^3$ time whether it is contained in G as a minor. Since $\mathrm{Forb}(\mathcal{G}_k)$ depends only on k, both in terms of its cardinality and the maximum size of a member, this algorithm runs in $g(k) \cdot |V(G)|^3$ for some function g. Similarly to the algorithms discussed in Section 6.3, this gives only a nonuniform FPT algorithm — Corollary 6.11 does not provide us any means of constructing the family $\mathcal{G}_k$, or even bounding its size.

Alternatively, one could approach the Treewidth problem using the graph searching game that we discussed in Section 7.5. This direction quite easily leads to an algorithm with running time $n^{\mathcal{O}(k)}$; the reader is asked to construct it in Exercise 7.26.

It is, nonetheless, possible to obtain a uniform, constructive algorithm for the TREEWIDTH problem. However, this requires much more involved techniques and technical effort. In the following theorem we state the celebrated algorithm of Bodlaender that shows that TREEWIDTH is FPT.

Theorem 7.17 ([45]). *There exists an algorithm that, given an n-vertex graph G and integer k, runs in time $k^{\mathcal{O}(k^3)} \cdot n$ and either constructs a tree decomposition of G of width at most k, or concludes that* $\mathrm{tw}(G) > k$.

Generally, in the literature there is a variety of algorithms for computing good tree decompositions of graphs. The algorithm of Theorem 7.17 is the fastest known *exact* parameterized algorithm, meaning that it computes the exact value of the treewidth. Most other works follow the direction of *approximating* treewidth. In other words, we relax the requirement that the returned decomposition has width at most k by allowing the algorithm to output a decomposition with a slightly larger width. In exchange we would like to obtain better parameter dependence than $k^{\mathcal{O}(k^3)}$, which would be a dominating factor in most applications.

In this section we shall present the classical approximation algorithm for treewidth that originates in the work of Robertson and Seymour. More precisely, we prove the following result.

Theorem 7.18. *There exists an algorithm that, given an n-vertex graph G and integer k, runs in time $\mathcal{O}(8^k k^2 \cdot n^2)$ and either constructs a tree decomposition of G of width at most $4k+4$, or concludes that* $\mathrm{tw}(G) > k$.

We remark that the running time of algorithm of Theorem 7.17 depends linearly on the size of the graph, whereas in Theorem 7.18 the dependence is quadratic. Although in this book we usually do not investigate precisely the polynomial factors in the running time bounds, we would like to emphasize that this particular difference is important in applications, as the running time of most dynamic-programming algorithms on tree decompositions depends linearly on the graph size, and, consequently, the n^2 factor coming from Theorem 7.18 is a bottleneck. To cope with this issue, very recently a 5-approximation algorithm running in time $2^{\mathcal{O}(k)} \cdot n$ has been developed (see also the bibliographic notes at the end of the chapter); however, the techniques needed to obtain such a result are beyond the scope of this book.

The algorithm of Theorem 7.18 is important not only because it provides an efficient way of finding a reasonable tree decomposition of a graph, but also because the general strategy employed in this algorithm has been reused multiple times in approximation algorithms for other structural graph parameters. The crux of this approach can be summarized as follows.

- It is easy to see that every n-vertex tree T contains a vertex v such that every connected component of $T - v$ has at most $\frac{n}{2}$ vertices.

A similar fact can be proved for graphs of bounded treewidth. More precisely, graphs of treewidth at most k have balanced separators of size at most $k+1$: it is possible to remove $k+1$ vertices from the graph so that every connected component left is "significantly" smaller than the original graph.

- The algorithm decomposes the graph recursively. At each step we try to decompose some part of a graph that has only ck vertices on its boundary, for some constant c.
- The crucial step is to find a small separator of the currently processed part H with the following property: the separator splits H into two pieces H_1, H_2 so that the boundary of H gets partitioned evenly between H_1 and H_2. If for some $\alpha > 1$, each of H_1 and H_2 contains only, say, $\frac{ck}{\alpha}$ vertices of $\partial(H)$, then we have $|\partial(H_1)|, |\partial(H_2)| \leq \frac{ck}{\alpha} + (k+1)$; the summand $k+1$ comes from the separator. If $\frac{ck}{\alpha} + (k+1) \leq ck$, then we can recurse into pieces H_1 and H_2.

Before we proceed to the proof of Theorem 7.18, we need to introduce some auxiliary results about balanced separators and separations in graphs of small treewidth.

7.6.1 Balanced separators and separations

From now on we will work with the notions of separators and separations in graphs; we have briefly discussed these concepts in Section 7.2. Let us recall the main points and develop further these definitions. A pair of vertex subsets (A, B) is a *separation* in graph G if $A \cup B = V(G)$ and there is no edge between $A \setminus B$ and $B \setminus A$. The *separator* of this separation is $A \cap B$, and the *order* of separation (A, B) is equal to $|A \cap B|$.

We say that (A, B) *separates* two sets X, Y if $X \subseteq A$ and $Y \subseteq B$. Note that then every X-Y path, i.e., a path between a vertex from X and a vertex from Y, has to pass through at least one vertex of $A \cap B$. Given this observation, we may say that a set $C \subseteq V(G)$ *separates* two vertex sets X and Y if every X-Y path contains a vertex of C. Note that in this definition we allow C to contain vertices from X or Y. Given C that separates X and Y, it is easy to construct a separation (A, B) that separates X and Y and has C as the separator.

For two vertex sets X and Y, by $\mu(X, Y)$ we denote the minimum size of a separator separating X and Y, or, equivalently, the minimum order of a separation separating X and Y. Whenever the graph we refer to is not clear from the context, we put it as a subscript of the μ symbol. By the classic theorem of Menger, $\mu(X, Y)$ is equal to the maximum number of vertex-disjoint X-Y paths. The value of $\mu(X, Y)$ can be computed in polynomial time by any algorithm for computing the maximum flow in a graph. Moreover, within the same running time we can obtain both a separation of order $\mu(X,Y)$ separating X and Y, as well as a family of $\mu(X,Y)$ vertex-disjoint X-Y paths (see also Theorems 8.2, 8.4 and 8.5).

We now move to our first auxiliary result, which states that graphs of treewidth at most k have balanced separators of size at most $k+1$. We prove a slightly more general statement of this fact. Assume that $\mathbf{w}\colon V(G) \to \mathbb{R}_{\geq 0}$ is a nonnegative weight function on the vertices of G. For a set $X \subseteq V(G)$, let us define $\mathbf{w}(X) = \sum_{u \in X} \mathbf{w}(u)$. Let $\alpha \in (0,1)$ be any constant. We say that a set $X \subseteq V(G)$ is an *α-balanced separator* in G if for every connected component D of $G - X$, it holds that $\mathbf{w}(D) \leq \alpha \cdot \mathbf{w}(V(G))$. Informally speaking, a balanced separator breaks the graph into pieces whose weights constitute only a constant fraction of the original weight of the graph.

Like trees, graphs of "small" treewidth have "small" balanced separators. This property is heavily exploited in numerous algorithmic applications of treewidth.

If we put uniform weights on vertices, then balanced separators split the graph more or less evenly with respect to the cardinalities of the obtained pieces. Recall, however, that in the presented strategy for the algorithm of Theorem 7.18 we would like to split evenly some smaller subset of vertices, which is the boundary of the currently processed part of the graph. Therefore, in our applications we will put weights 1 on some vertices of the graph, and 0 on the other. Then balanced separators split evenly the set of vertices that are assigned weight 1. In other words, what we need is a weighted version of balanced separators.

We now prove the fact that graphs of small treewidth admit small balanced separators.

Lemma 7.19. *Assume G is a graph of treewidth at most k, and consider a nonnegative weight function $\mathbf{w}\colon V(G) \to \mathbb{R}_{\geq 0}$ on the vertices of G. Then in G there exists a $\frac{1}{2}$-balanced separator X of size at most $k+1$.*

Proof. Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of G of width at most k. We prove that one of the bags of $\mathcal{T}$ is a $\frac{1}{2}$-balanced separator we are looking for. We start by rooting T at an arbitrarily chosen node r. For a node t of T, let V_t be the set of vertices of G contained in the bags of the subtree T_t of T rooted at node t (including t itself). Let us select a node t of T such that:

- $\mathbf{w}(V_t) \geq \frac{1}{2}\mathbf{w}(V(G))$, and subject to
- t is at the maximum distance in T from r.

Observe that such a vertex t exists because root r satisfies the first condition.

We claim that X_t is a $\frac{1}{2}$-balanced separator in G. Let $t_1, \ldots, t_p$ be the children of t in T (possibly $p = 0$). Observe that every connected component D of $V(G) \setminus X_t$ is entirely contained either in $V(G) \setminus V_t$, or in $V_{t_i} \setminus X_t$ for some $i \in \{1, 2, \ldots, p\}$. Since $\mathbf{w}(V_t) \geq \frac{1}{2}\mathbf{w}(V(G))$, then $\mathbf{w}(V(G) \setminus V_t) \leq \frac{1}{2}\mathbf{w}(V(G))$. On the other hand, by the choice of t we have that $\mathbf{w}(V_{t_i}) < \frac{1}{2}\mathbf{w}(V(G))$ for every $i \in \{1, \ldots, p\}$. Consequently, in both cases D is entirely contained in a set whose weight is at most $\frac{1}{2}\mathbf{w}(V(G))$, which means that $\mathbf{w}(D) \leq \frac{1}{2}\mathbf{w}(V(G))$. □

The proof of Lemma 7.19 shows that if a tree decomposition of G of width at most k is given, one can find such a $\frac{1}{2}$-balanced separator in polynomial time. Moreover, the found separator is one of the bags of the given decomposition. However, in the remainder of this section we will only need the existential statement.

Recall that in the sketched strategy we were interested in splitting the currently processed subgraph into two even parts. However, after deleting the vertices of the separator given by Lemma 7.19 the graph can have more than two connected components. For this reason, we would like to group these pieces (components) into two roughly equal halves. More formally, we say that a separation (A, B) of G is an *α-balanced separation* if $\mathbf{w}(A \setminus B) \leq \alpha \cdot \mathbf{w}(V(G))$ and $\mathbf{w}(B \setminus A) \leq \alpha \cdot \mathbf{w}(V(G))$. The next lemma shows that we can focus on balanced separations instead of separators, at a cost of relaxing $\frac{1}{2}$ to $\frac{2}{3}$.

Lemma 7.20. *Assume G is a graph of treewidth at most k, and consider a nonnegative weight function $\mathbf{w}\colon V(G) \to \mathbb{R}_{\geq 0}$ on the vertices of G. Then in G there exists a $\frac{2}{3}$-balanced separation (A, B) of order at most $k + 1$.*

Proof. Let X be a $\frac{1}{2}$-balanced separator in G of size at most $k + 1$, given by Lemma 7.19. Let $D_1, D_2, \ldots, D_p$ be the vertex sets of connected components of $G - X$. For $i \in \{1, 2, \ldots, p\}$, let $a_i = \mathbf{w}(D_i)$; recall that we know that $a_i \leq \frac{1}{2}\mathbf{w}(V(G))$. By reordering the components if necessary, assume that $a_1 \geq a_2 \geq \ldots \geq a_p$.

Let q be the smallest index such that $\sum_{i=1}^{q} a_i \geq \frac{1}{3}\mathbf{w}(V(G))$, or $q = p$ if no such index exists. We claim that $\sum_{i=1}^{q} a_i \leq \frac{2}{3}\mathbf{w}(V(G))$. This is clearly true if $\sum_{i=1}^{p} a_i < \frac{1}{3}\mathbf{w}(V(G))$. Also, if $q = 1$, then $\sum_{i=1}^{q} a_i = a_1 \leq \frac{1}{2}\mathbf{w}(V(G))$ and the claim holds in this situation as well. Therefore, assume that $\sum_{i=1}^{q} a_i \geq \frac{1}{3}\mathbf{w}(V(G))$ and $q > 1$. By the minimality of q, we have that $\sum_{i=1}^{q-1} a_i < \frac{1}{3}\mathbf{w}(V(G))$. Hence $a_q \leq a_{q-1} \leq \sum_{i=1}^{q-1} a_i < \frac{1}{3}\mathbf{w}(V(G))$, so $\sum_{i=1}^{q} a_i = a_q + \sum_{i=1}^{q-1} a_i < \frac{1}{3}\mathbf{w}(V(G)) + \frac{1}{3}\mathbf{w}(V(G)) = \frac{2}{3}\mathbf{w}(V(G))$. This proves the claim.

We now define $A = X \cup \bigcup_{i=1}^{q} D_i$ and $B = X \cup \bigcup_{i=q+1}^{p} D_i$. Clearly (A, B) is a separation of G with X being the separator, so it has order at most $k+1$. By the claim from the last paragraph, we have that $\mathbf{w}(A \setminus B) = \sum_{i=1}^{q} a_i \leq \frac{2}{3}\mathbf{w}(V(G))$. Moreover, either we have that $p = q$ and $B \setminus A = \emptyset$, or $p < q$ and then $\sum_{i=1}^{q} a_i \geq \frac{1}{3}\mathbf{w}(V(G))$. In the first case $\mathbf{w}(B \setminus A) = 0$, and in the latter case we have that $\mathbf{w}(B \setminus A) = \sum_{i=q+1}^{p} a_i \leq \mathbf{w}(V(G)) - \frac{1}{3}\mathbf{w}(V(G)) =$

$\frac{2}{3}\mathbf{w}(V(G))$. We conclude that (A, B) is a $\frac{2}{3}$-balanced separation in G of order at most $k + 1$. □

From Lemma 7.20 we can infer the following corollary, which will be our main tool in the proof of Theorem 7.18.

Corollary 7.21. *Let G be a graph of treewidth at most k and let $S \subseteq V(G)$ be a vertex subset with $|S| = 3k + 4$. Then there exists a partition (S_A, S_B) of S such that $k + 2 \leq |S_A|, |S_B| \leq 2k + 2$ and $\mu_G(S_A, S_B) \leq k + 1$.*

Proof. Let us define the following weight function $\mathbf{w}$: $\mathbf{w}(u) = 1$ if $u \in S$, and $\mathbf{w}(u) = 0$ otherwise. Let (A, B) be a separation provided by Lemma 7.20. We know that (A, B) has order at most $k+1$ and that $|(A \setminus B) \cap S|, |(B \setminus A) \cap S| \leq \frac{2}{3}|S| = 2k + \frac{8}{3}$. Since these cardinalities are integral, we in fact have that $|(A \setminus B) \cap S|, |(B \setminus A) \cap S| \leq 2k + 2$.

We now define the partition (S_A, S_B). We first put $(A \setminus B) \cap S$ into S_A and $(B \setminus A) \cap S$ into S_B; in this manner we have at most $2k + 2$ vertices in S_A and at most $2k + 2$ vertices in S_B. Then we iteratively examine vertices of $(A \cap B) \cap S$, and each vertex $u \in (A \cap B) \cap S$ is assigned to S_A or to S_B, depending on which of them is currently smaller (or arbitrarily, if they are equal). Since $|S| = 3k + 4 \leq 2 \cdot (2k + 2)$, in this manner none of the sets S_A or S_B can get larger than $2k + 2$.

Thus we obtain that $|S_A| \leq 2k + 2$, so also $|S_B| = |S| - |S_A| \geq k + 2$. Symmetrically $|S_A| \geq k+2$. Finally, observe that separation (A, B) separates S_A and S_B, which proves that $\mu_G(S_A, S_B) \leq |A \cap B| \leq k + 1$. □

7.6.2 An FPT approximation algorithm for treewidth

Armed with our understanding of balanced separators in graphs of bounded treewidth, we can proceed to the proof of Theorem 7.18.

Proof (of Theorem 7.18). We will assume that the input graph G is connected, as otherwise we can apply the algorithm to each connected component of G separately, and connect the obtained tree decompositions arbitrarily. Furthermore, we assume that G has at most kn edges, since otherwise we can immediately conclude that $\mathrm{tw}(G) > k$; see Exercise 7.15. Let $m = |E(G)|$.

We present a recursive procedure `decompose`(W, S) for $S \subsetneq W \subseteq V(G)$, which tries to decompose the subgraph $G[W]$ in such a way that the set S is contained in one bag of the decomposition. The procedure will work under the assumption that the following invariants are satisfied:

(i) $|S| \leq 3k + 4$ and $W \setminus S \neq \emptyset$;
(ii) both $G[W]$ and $G[W \setminus S]$ are connected;
(iii) $S = N_G(W \setminus S)$. In other words, every vertex of S is adjacent to some vertex of $W \setminus S$, and the vertices of $W \setminus S$ do not have neighbors outside W.

These invariants exactly express the properties that we want to maintain during the algorithm: at each point we process some connected subgraph of G, which communicates with the rest of G only via a small interface S. The output of procedure `decompose`(W, S) is a rooted tree decomposition $\mathcal{T}_{W,S}$ of the graph $G[W]$, which has width at most $4k + 4$ and satisfies $S \subseteq X_r$, where r is the root of $\mathcal{T}_{W,S}$. The whole algorithm then boils down to calling `decompose`$(V(G), \emptyset)$; note that the invariants in this call are satisfied since we assumed G to be connected.

We now explain how procedure `decompose`(W, S) is implemented. The first step is to construct a set $\widehat{S}$ with the following properties:

(a) $S \subsetneq \widehat{S} \subseteq W$, that is, $\widehat{S}$ is a proper superset of S;
(b) $|\widehat{S}| \leq 4k + 5$;
(c) every connected component of $G[W \setminus \widehat{S}]$ is adjacent to at most $3k + 4$ vertices of $\widehat{S}$.

Intuitively, the set $\widehat{S}$ is going to form the root bag of the constructed decomposition. Property (b) is needed to ensure small width, and property (c) will be essential for maintaining invariant (i) when we recurse into connected components of $G[W \setminus \widehat{S}]$. So far it is not clear why we require that $\widehat{S}$ contains at least one more vertex than S. This property will be very helpful when estimating the running time of the algorithm. In particular, this will allow us to show that the number of nodes in the constructed tree decomposition is at most n.

Construction of $\widehat{S}$ is very simple if $|S| < 3k + 4$. We just pick an arbitrary vertex $u \in W \setminus S$ (which exists since $W \setminus S \neq \emptyset$), and we put $\widehat{S} = S \cup \{u\}$. Then $|\widehat{S}| \leq 3k + 4$. It is straightforward to verify that properties (a), (b), and (c) are satisfied in this case.

Assume then that $|S| = 3k + 4$. If $\mathrm{tw}(G) \leq k$, then also $\mathrm{tw}(G[W]) \leq k$, so by Lemma 7.21, there exists a partition (S_A, S_B) of S such that $k + 2 \leq |S_A|, |S_B| \leq 2k + 2$ and $\mu_{G[W]}(S_A, S_B)$, the minimum size of a separator separating S_A and S_B in $G[W]$, is at most $k + 1$. The algorithm iterates through all the 2^{3k+4} partitions of S into two parts, and applies a max-flow algorithm to verify whether the minimum order of a separation of $G[W]$ separating the sides does not exceed $k + 1$. If no partition satisfying the properties expected from (S_A, S_B) has been found, then by Lemma 7.21 we can safely conclude that $\mathrm{tw}(G[W]) > k$. Hence $\mathrm{tw}(G) > k$, and the whole algorithm can be terminated.

Assume then that a partition (S_A, S_B) satisfying the required properties has been obtained. This means that there exists some separation (A, B) in $G[W]$ that has order at most $k + 1$ and separates S_A from S_B; see Fig. 7.5. Moreover, this separation was computed during the application of a max-flow algorithm when partition (S_A, S_B) was considered. We now put $\widehat{S} = S \cup (A \cap B)$. Clearly $|\widehat{S}| \leq |S| + |A \cap B| \leq 3k + 4 + k + 1 = 4k + 5$, so property (b) is satisfied. For property (c), let D be the vertex set of a connected

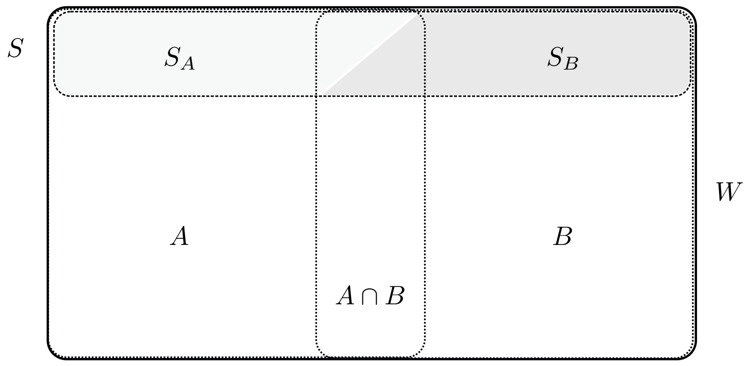


Fig. 7.5: Situation in procedure decompose(W, S) when $|S| = 3k + 4$

component of $G[W \setminus \widehat{S}]$. Since D is connected in $G[W]$ and disjoint from $A \cap B$, it follows that $D \subseteq A \setminus B$ or $D \subseteq B \setminus A$. Assume the former; the proof for the second case is symmetric. Then the vertices of $\widehat{S}$ that are adjacent to D can be only contained in $(A \setminus B) \cap S$ or in $A \cap B$. However, we have that

$$|(A \setminus B) \cap S| + |A \cap B| \leq |S_A| + |A \cap B| \leq (2k + 2) + (k + 1) = 3k + 3,$$

so property (c) indeed holds.

For property (a), we just need to check that $\widehat{S}$ contains at least one more vertex than S. Since $|S_A|, |S_B| \geq k + 2$ and $|A \cap B| \leq k + 1$, sets $S_A \setminus (A \cap B)$ and $S_B \setminus (A \cap B)$ are nonempty. Let $u_A \in S_A \setminus (A \cap B)$ and $u_B \in S_B \setminus (A \cap B)$ be arbitrarily chosen vertices. By invariants (ii) and (iii), there is a path P in $G[W]$ that starts in u_A, ends in u_B, and whose internal vertices are contained in $W \setminus S$. Indeed, u_A has a neighbor u'_A in $W \setminus S$, u_B has a neighbor u'_B in $W \setminus S$, whereas u'_A and u'_B can be connected by a path inside $G[W \setminus S]$ due to the connectivity of this graph. As (A, B) separates S_A from S_B, path P contains at least one vertex from $A \cap B$. This vertex can be neither u_A nor u_B, since these vertices do not belong to $A \cap B$. Hence $A \cap B$ contains at least one internal vertex of P, which belongs to $W \setminus S$; this proves property (a).

Once the set $\widehat{S}$ is constructed, procedure decompose(W, S) can be completed easily. Let $D_1, D_2, \ldots, D_p$ be the vertex sets of connected components of $G[W \setminus \widehat{S}]$ (possibly $p = 0$). For each $i = 1, 2, \ldots, p$, we call recursively procedure decompose$(N_G[D_i], N_G(D_i))$. Satisfaction of invariants for these calls follows directly from the definition of sets D_i, invariant (iii) for the call decompose(W, S), and properties (a) and (c) of $\widehat{S}$. Let $\mathcal{T}_i$ be the tree decomposition obtained from the call decompose$(N_G[D_i], N_G(D_i))$, and let r_i be its root; recall that X_{r_i} contains $N(D_i)$. We now obtain a tree decomposition $\mathcal{T}_{W,S}$ of $G[W]$ as follows: create a root r with bag $X_r = \widehat{S}$, and for every $i = 1, 2, \ldots, p$ attach $\mathcal{T}_i$ below r using edge rr_i. From the construction it easily follows that $\mathcal{T}_{W,S}$ is a tree decomposition of $G[W]$. Moreover, it has width at most $4k+4$, since $|\widehat{S}| \le 4k+5$ (property (b)) and every tree decomposition $\mathcal{T}_i$ has width at most $4k+4$. Finally, we have that $X_r = \widehat{S} \supseteq S$. Hence $\mathcal{T}_{W,S}$ has all the required properties and can be output as the result of `decompose`(W,S). Note that $p = 0$ represents the base case; in this situation the tree decomposition $\mathcal{T}_{W,S}$ of $G[W]$ is a single node with the bag $X_r = \widehat{S}$.

This concludes the description of the algorithm, and we are left with estimating its running time. For every call of `decompose`(W,S) when $|S| = 3k+4$, we iterate through 2^{3k+4} partitions of S into two sides. For each partition we apply, say, the Ford-Fulkerson algorithm in the graph $G[W]$ to compute the maximum vertex flow between the sides. This algorithm runs a number of iterations; each iteration takes $\mathcal{O}(n+m)$ time and either concludes that the currently found flow is maximum, or augments it by 1. Since we are interested only in situations when the maximum flow is of size at most $k+1$, we may terminate the computation after $k+2$ iterations (see also Theorem 8.2). Hence, each application of the Ford-Fulkerson algorithm takes time $\mathcal{O}(k(n+m)) = \mathcal{O}(k^2 n)$, since $m \le kn$. In total, this gives time $\mathcal{O}(2^{3k+4}k^2 \cdot n)$ for computing set $\widehat{S}$. All the other operations performed in `decompose`(W,S), like partitioning $G[W \setminus \widehat{S}]$ into connected components or joining the decompositions obtained in the recursive calls, can be easily implemented in $\mathcal{O}(n+m) = \mathcal{O}(kn)$ time. In total, operations performed in procedure `decompose`(W,S) (excluding subcalls) take time $\mathcal{O}(2^{3k+4}k^2 \cdot n) = \mathcal{O}(8^k k^2 \cdot n)$.

Hence, it remains to prove an upper bound on how many times procedure `decompose`(W,S) is called. Observe that each call of `decompose`(W,S) creates exactly one new node of the output tree decomposition of G (the root bag $X_r = \widehat{S}$), so equivalently we can bound the total number of nodes constructed by the algorithm. This is precisely the point when we will use property (a) of the set $\widehat{S}$ computed in each call. Let $\mathcal{T}$ be the final tree decomposition of G output by the algorithm. For each call `decompose`(W,S), we have that set $\widehat{S}$, which is the bag at the root of the returned decomposition $\mathcal{T}_{W,S}$, contains at least one vertex u that does not belong to S. Consequently, the root of $\mathcal{T}_{W,S}$ is the unique top-most node of $\mathcal{T}$ whose bag contains u. Since for every call `decompose`(W,S) we can find such a vertex u, and these vertices are pairwise different, we infer that the total number of constructed nodes does not exceed the total number of vertices in the graph, which is n. Therefore, procedure `decompose`(W,S) is called at most n times, and the whole algorithm runs in time $\mathcal{O}(8^k k^2 \cdot n^2)$. □

The algorithm of Theorem 7.18 can give a negative answer only in two cases: if it finds out that $m > kn$, or when it identifies a set S of size $3k+4$ which does not satisfy the statement of Lemma 7.21 in graph $G[W]$, and consequently also in graph G. Distinguishing the case when $m > kn$ was needed only for the purpose of achieving better running time; if we drop this case, then the algorithm is still correct, but works in $\mathcal{O}(8^k k \cdot nm)$ time. Hence, after this small adjustment the algorithm of Theorem 7.18 provides an obstacle in the graph whenever it is incapable of constructing a decomposition. This obstacle is a set S with $|S| = 3k + 4$ that does not satisfy the statement of Corollary 7.21 in G.

This property of a set S is a variant of the notion of *well-linkedness*. The exact definition of a well-linked set is slightly different, and the statement of Corollary 7.21 is contrived to its algorithmic usage in the proof of Theorem 7.18. While in this book we use a well-linked set only in the treewidth-approximation algorithm as a certificate for large treewidth, this notion has found a lot of other important algorithmic applications. In the bibliographic notes we provide pointers to literature giving a broader presentation of this subject.

7.7 Win/win approaches and planar problems

We start from formalizing the idea of a win/win approach, already sketched in the introduction of this chapter. Let us focus on the Vertex Cover problem; we would like to prove that this problem is FPT using treewidth, ignoring for a moment the existence of a simple $2^k \cdot n^{\mathcal{O}(1)}$ branching strategy.

Observe first that if a graph admits a vertex cover X of size k, then it has treewidth at most k. Indeed, one can create a simple tree decomposition (even a path decomposition) of width k, where every bag contains the whole set X and one of the vertices outside X, and the bags are arranged into a path arbitrarily. Given an instance (G, k) of Vertex Cover, let us apply the algorithm of Theorem 7.18 to G and parameter k; this takes time $\mathcal{O}(8^k k^2 \cdot n^2)$, where $n = |V(G)|$. If the algorithm concludes that $\mathrm{tw}(G) > k$, then we can conclude that there is no vertex cover of G of size at most k. Otherwise, we have a tree decomposition of G of width at most $4k + 4$ at hand. Hence, we can apply the dynamic-programming routine of Corollary 7.6 to solve the problem in $2^{4k+4} k^{\mathcal{O}(1)} \cdot n$ time. Thus, the total running time of the algorithm is $\mathcal{O}(8^k k^2 \cdot n^2 + 16^k k^{\mathcal{O}(1)} \cdot n)$.

This simple trick works well for Vertex Cover, Feedback Vertex Set, or even the more general Treewidth-η Modulator problem, where for a fixed constant η the task is to decide whether a given graph G can be turned into a graph of treewidth at most η by deleting at most k vertices. Essentially, we just need two properties: that a graph in a yes-instance always has treewidth bounded in terms of k, and that the problem admits an FPT algorithm when parameterized by treewidth. While the second property usually holds, the first one seems very restrictive. For example, consider the Cycle Packing problem: given graph G and integer k, we need to determine whether G contains at least k vertex-disjoint cycles. This problem is in some sense dual to Feedback Vertex Set. So far we do not see any link between the treewidth of a graph and the maximum number of cycles that can be packed in it. The intuition, however, is that a graph of large treewidth has a very complicated, entangled structure that should allow us to find as many cycles as we like.

In order to follow this direction, we need to take a closer look at combinatorial structures that can be always found in graphs of large treewidth. Equivalently, we can examine obstacles that prevent the existence of tree decompositions of small width. So far we know two types of such structures. The first ones are brambles that we discussed in Section 7.5. However, these objects are quite difficult to grasp and understand, and moreover we do not know efficient procedures for constructing brambles of large order. The second ones are well-linked sets, or in our notation, sets S that do not satisfy the statement of Corollary 7.21. Their obvious advantage is that we get such a set S directly from the approximation algorithm of Theorem 7.18 in case of its failure. While it is still unclear why the existence of this obstacle implies the possibility of packing many cycles in the graph, there is at least some hope that from a well-linked set one could extract a more robust obstacle that would be easier to use.

This intuition is indeed true, and leads to a powerful result called the *Excluded Grid Theorem*: graphs of large treewidth contain large grids as minors. While the proof of this result is beyond the scope this book, we shall state it formally and derive some algorithmic consequences using a win/win approach. Of particular interest for us are the corollaries for planar graphs. For this class of graphs the relation between the treewidth and grid minors is very tight, which enables us to design many efficient approximation schemes, kernelization procedures, and, most importantly for us, fixed-parameter tractable algorithms.

7.7.1 Grid theorems

As we have discussed in the beginning of Section 7.6, treewidth is a minor-closed parameter and hence the class $\mathcal{G}_t$ comprising graphs of treewidth at most t is closed under taking minors.[5] From the Graph Minors theorem it follows that the property of having treewidth at most t can be characterized by a finite set of forbidden minors $\mathrm{Forb}(\mathcal{G}_t)$. That is, a graph G has treewidth at most t if and only if it does not contain any graph $H \in \mathrm{Forb}(\mathcal{G}_t)$ as a minor. But what do graphs of $\mathrm{Forb}(\mathcal{G}_t)$ look like? We apparently do not know the answer to this question. However, we would like to get some "approximate characterization" for graphs having treewidth at most t that is more tractable.

The obvious first attempt is to look at minors of complete graphs. It is easy to see that if a graph G contains K_t as a minor, then $\mathrm{tw}(G) \geq t-1$; see Exercise 7.8 . However, it is the converse implication that would

[5] As in this section we work with parameterized problems with parameter denoted by k, from now on we switch to a convention of denoting the width of a tree decomposition by t.

be more valuable to us: it would be perfect if, for some function g, the fact that $\mathrm{tw}(G) > g(t)$ would imply the existence of a K_t minor model in G. This implication is unfortunately not true, as we will see in the next few paragraphs.

Consider a large grid. More precisely, for a positive integer t, a $t \times t$ *grid* $\boxplus_t$ is a graph with vertex set $\{(x,y) : x, y \in \{1,\ldots,t\}\}$. Thus $\boxplus_t$ has exactly t^2 vertices. Two different vertices (x,y) and (x',y') are adjacent if and only if $|x - x'| + |y - y'| = 1$. See the left graph in Fig. 7.6.

It is easy to show that the treewidth of $\boxplus_t$ is at most t. For example, $t+1$ cops can always catch the fugitive by sweeping the grid column by column. It is harder to prove that the treewidth of $\boxplus_t$ is exactly t. Here Theorem 7.15 is very handy. The cross C_{ij} of the grid is the union of the vertices of the i-th column and the j-th row. The set of crosses of $\boxplus_t$ forms a bramble of order t, hence the treewidth is at least $t-1$. With a bit more work, one can construct a bramble of order $t+1$ in $\boxplus_t$, and thus prove that $\mathrm{tw}(\boxplus_t) = t$; see Exercise 7.37. On the other hand, a grid is a planar graph, so by Kuratowski's theorem, it does not admit a K_5-minor. Therefore, K_5-minor-free graphs can be of arbitrarily large treewidth.

Our first attempt failed on the example of grids, so let us try to refine it by taking grid minors as obstacles, instead of clique minors. Since $\mathrm{tw}(\boxplus_t) = t$, a graph that contains a $t \times t$ grid as a minor must have treewidth at least t. What is more surprising, is that now the converse implication is true: there exists a function g such that every graph of treewidth larger than $g(t)$ contains a $t \times t$ grid as a minor.

This fundamental result was first proved by Robertson and Seymour. They did not give any explicit bounds on function g, but a bound following from their proof is astronomical. The first proof providing an explicit bound on g was later given by Robertson, Seymour, and Thomas; they showed that one can take $g(t) = 2^{\mathcal{O}(t^5)}$. There was a line of subsequent improvements decreasing the polynomial function in the exponent. However, whether g can be bounded by a polynomial function of t was open for many years. This open problem was resolved in 2013 by Chekuri and Chuzhoy in the affirmative. More precisely, they proved the following result.

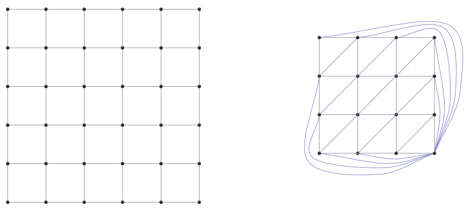


Fig. 7.6: Example of a 6×6 grid $\boxplus_6$ and a triangulated grid Γ_4

Theorem 7.22 (Excluded grid theorem, [73]). *There exists a function $g(t) = \mathcal{O}(t^{98+o(1)})$ such that every graph of treewidth larger than $g(t)$ contains $\boxplus_t$ as a minor.*

Theorem 7.22 gives the tightest relationship known so far between the treewidth and the size of a grid minor. It is worth remarking that its proof is constructive: there exists a randomized polynomial-time algorithm that either constructs a $\boxplus_t$ minor model, or finds a tree decomposition of the input graph of width at most $g(t)$.

Note that Theorem 7.22 provides us the relationship between treewidth and packing cycles in a graph, which we discussed in the introduction of this section. Indeed, if a graph G contains a $\boxplus_t$ minor model for $t = 2\lceil\sqrt{k}\rceil$, then in this model one can find $(\lceil\sqrt{k}\rceil)^2 \geq k$ vertex-disjoint cycles: we just need to partition $\boxplus_t$ into 2×2 subgrids, and find a cycle in each of them. Hence, whenever a graph G has treewidth larger than $g(t) = \mathcal{O}(k^{49+o(1)})$, then we are certain that one can pack k vertex-disjoint cycles in G. As a result, also for Cycle Packing we can apply the same win/win approach as described before. Namely, we run the approximation algorithm for parameter $g(t)$. If $\mathrm{tw}(G) > g(t)$, then we conclude that (G, k) is a yes-instance. Otherwise, we obtain a tree decomposition of G of width at most $4g(t) + 4$, on which we can employ dynamic programming (see Exercise 7.19). This shows that the Cycle Packing problem is fixed-parameter tractable.

If we assume that graph G is planar, then it is possible to get a much tighter relationship between the treewidth of G and the size of the largest grid minor that G contains. The following theorem is due to Robertson, Seymour and Thomas; we present here a version with refined constants due to Gu and Tamaki.

Theorem 7.23 (Planar excluded grid theorem, [239, 403]). *Let t be a nonnegative integer. Then every planar graph G of treewidth at least $9t/2$ contains $\boxplus_t$ as a minor. Furthermore, for every $\varepsilon > 0$ there exists an $\mathcal{O}(n^2)$ algorithm that, for a given n-vertex planar graph G and integer t, either outputs a tree decomposition of G of width at most $(9/2 + \varepsilon)t$, or constructs a minor model of $\boxplus_t$ in G.*

In other words, for planar graphs the function g is *linear*. Since $\boxplus_t$ has t^2 vertices, every graph containing $\boxplus_t$ has at least t^2 vertices. Therefore, Theorem 7.23 immediately implies the following corollary.

Corollary 7.24. *The treewidth of an n-vertex planar graph G is less than $\frac{9}{2}\left\lceil\sqrt{n+1}\right\rceil$. Moreover, for any $\varepsilon > 0$, a tree decomposition of G of width at most $(\frac{9}{2} + \varepsilon)\left\lceil\sqrt{n+1}\right\rceil$ can be constructed in $\mathcal{O}(n^2)$ time.*

For us it is more important that the planar excluded grid theorem allow us to identify many parameterized problems on planar graphs with parameter k, such that the treewidth of the graph in every yes-instance/no-instance is $\mathcal{O}(\sqrt{k})$. For example, a quick reader probably can already see that this will be the case for the Cycle Packing problem. This requires inspecting the general win/win strategy for this problem that we explained before, and replacing function g provided by Theorem 7.22 with the one provided by Theorem 7.23. We shall delve deeper into this direction in Section 7.7.2, but now we proceed to one more variant of the excluded grid theorem.

The variant we are about to introduce considers edge contractions instead of minors. For this, we shall define a new family of graphs which play the role of grids. For an integer $t > 0$, the graph Γ_t is obtained from the grid $\boxplus_t$ by adding, for all $1 \leq x, y \leq t-1$, the edge $(x+1, y), (x, y+1)$, and additionally making vertex (t,t) adjacent to all the other vertices (x, y) with $x \in \{1, t\}$ or $y \in \{1, t\}$, i.e., to the whole border of $\boxplus_t$. Graph Γ_4 is the graph in the right panel of Fig. 7.6.

Theorem 7.25 (Planar excluded grid theorem for edge contractions). *For every connected planar graph G and integer $t \geq 0$, if $\mathrm{tw}(G) \geq 9t+5$ then G contains Γ_t as a contraction. Furthermore, for every $\varepsilon > 0$ there exists an $\mathcal{O}(n^2)$ algorithm that, given a connected planar n-vertex graph G and integer t, either outputs a tree decomposition of G of width $(9+\varepsilon)t+5$ or a set of edges whose contraction in G results in Γ_t.*

Proof (sketch). By Theorem 7.23, if the treewidth of G is at least $9t + 5$, then G contains $\boxplus_{2t+1}$ as a minor. This implies that, after a sequence of vertex deletions, edge deletions and edge contractions, G can be transformed to $\boxplus_{2t+1}$. Let us examine this sequence, omit all edge deletions, and replace every deletion of some vertex v with an edge contraction between v and one of its neighbors (such a neighbor exists by the connectivity of G). It is easy to see that this sequence of edge contractions transforms G into graph H which is a partially triangulated $(2t + 1) \times (2t + 1)$ grid, that is, a planar graph obtained from grid $\boxplus_{2t+1}$ by adding some edges. We construct Γ_t from H by contracting edges as shown in Fig. 7.7. More precisely, we first contract the whole border of the grid so that it becomes one vertex adjacent to the whole border of the inner $(2t - 1) \times (2t - 1)$ subgrid. Then we contract this vertex onto the corner of the subgrid. At the end, we contract edges in the subgrid using the zig-zag pattern depicted in Fig. 7.7 to obtain the same diagonal in every cell. The final observation is that we must have obtained exactly the graph Γ_t with no extra edges, since adding any edge to Γ_t spoils its planarity—recall that we started from a planar graph G and applied only edge contractions. □

7.7.2 Bidimensionality

Introduction to bidimensionality. The planar excluded grid theorem (Theorem 7.23) provides a powerful tool for designing algorithms on planar

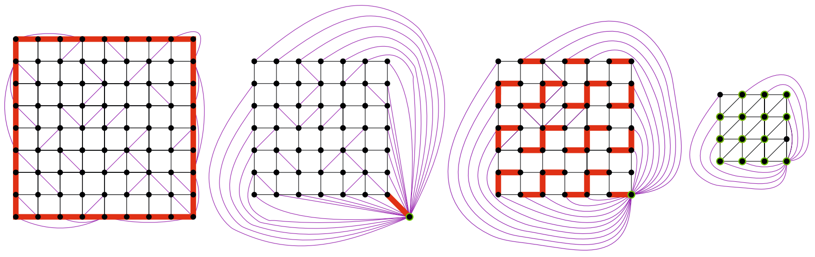


Fig. 7.7: The steps of the proof of Theorem 7.25. The two first steps are the boundary contraction of a partial triangulation of $\boxplus_9$. The third step is the contraction to Γ_4

graphs. In all these algorithms we use again the win/win approach. We first approximate the treewidth of a given planar graph. If the treewidth turns out to be small, we use dynamic programing to find a solution. Otherwise, we know that our graph contains a large grid as a minor, and using this we should be able to conclude the right answer to the instance.

Let us first give a few examples of this strategy. In all these examples, by making use of the planar excluded grid theorem we obtain algorithms with running time *subexponential* in the parameter.

Let us first look at Planar Vertex Cover, i.e., for a given planar graph G and parameter k, we need to determine whether there exists a vertex cover of G of size at most k. We need to answer the following three simple questions.

(i) How small can be a vertex cover of $\boxplus_t$? It is easy to check that $\boxplus_t$ contains a matching of size $\lfloor t^2/2 \rfloor$, and hence every vertex cover of $\boxplus_t$ is of cardinality at least $\lfloor t^2/2 \rfloor$.
(ii) Given a tree decomposition of width t of G, how fast can we solve Vertex Cover? By Corollary 7.6, this can be done in time $2^t \cdot t^{\mathcal{O}(1)} \cdot n$.
(iii) Is Vertex Cover minor-closed? In other words, is it true that for every minor H of graph G, the vertex cover of H does not exceed the vertex cover of G?

As was observed in Chapter 6.3, if the class of graphs with vertex cover at most k is minor-closed, i.e., a graph G has a vertex cover of size at most k, then the same holds for every minor of G. Thus, if G contains $\boxplus_t$ as a minor for some $t \geq \sqrt{2k+2}$, then by (i) G has no vertex cover of size k. By the planar excluded grid theorem, this means that the treewidth of a planar graph admitting a vertex cover of size k is smaller than $\frac{9}{2}\sqrt{2k+2}$.

We summarize the above discussion with the following algorithm. For $t = \lceil \sqrt{2k+2} \rceil$ and some $\varepsilon > 0$, by making use of the constructive part of Theorem 7.23 we either compute in time $\mathcal{O}(n^2)$ a tree decomposition of G

of width at most $(\frac{9}{2}+\varepsilon)t$, or we conclude that G has $\boxplus_t$ as a minor. In the second case, we infer that G has no vertex cover of size at most k. However, if a tree decomposition has been constructed, then by (ii) we can solve the problem in time $2^{(\frac{9}{2}+\varepsilon)\lceil\sqrt{2k+2}\rceil} \cdot k^{\mathcal{O}(1)} \cdot n = 2^{\mathcal{O}(\sqrt{k})} \cdot n$. The total running time of the algorithm is hence $2^{\mathcal{O}(\sqrt{k})} \cdot n + \mathcal{O}(n^2)$.

It is instructive to extract the properties of Planar Vertex Cover which were essential for obtaining a subexponential parameterized algorithm.

(P1) The size of any solution in $\boxplus_t$ is of order $\Omega(t^2)$.
(P2) Given a tree decomposition of width t, the problem can be solved in time $2^{\mathcal{O}(t)} \cdot n^{\mathcal{O}(1)}$.
(P3) The problem is minor-monotone, i.e., if G has a solution of size at most k, then every minor of G also has a solution of size at most k.

In this argument we have used the fact that in $\boxplus_t$ the size of a minimum solution is *lower-bounded* by a quadratic function of t. We could apply the same strategy for maximization problems where $\boxplus_t$ admits a solution of quadratic size. For instance, consider the Planar Longest Path and Planar Longest Cycle problems: given a planar graph G and integer k, we are asked to determine whether G contains a path/cycle on at least k vertices. It is easy to see that $\boxplus_t$ contains a path on t^2 vertices and a cycle on t^2-1 vertices (or even t^2, if t is even). Moreover, both these problems are minor-monotone in the following sense: if a graph H contains a path/cycle on at least k vertices, then so does every graph containing H as a minor. Finally, both these problems can be solved in time $t^{\mathcal{O}(t)} \cdot n^{\mathcal{O}(1)}$ when a tree decomposition of width t is available; see Exercise 7.19. Consequently, we can apply the same win/win approach to obtain $2^{\mathcal{O}(\sqrt{k}\log k)} \cdot n^{\mathcal{O}(1)}$-time algorithms for Planar Longest Path and Planar Longest Cycle — the only difference is that when a large grid minor is found, we give a positive answer instead of negative.

This suggests that properties (P1), (P2), and (P3) may constitute a base for a more general meta-result, which would apply to other problems as well. But before we formalize this idea, let us consider a few more examples.

Our next problem is Planar Dominating Set. It is easy to check that the problem satisfies (P1), and we also proved that it satisfies (P2) (see Theorem 7.7). However, (P3) does not hold. For example, to dominate vertices of a 3ℓ-vertex cycle C we need ℓ vertices. But if we add to C a universal vertex adjacent to all the vertices of C, then the new graph contains C as a minor, but has a dominating set of size 1. Therefore, the approach that worked for Planar Vertex Cover cannot be applied directly to Planar Dominating Set.

However, while not being closed under taking of minors, domination is closed under edge contractions. That is, if G can be dominated by k vertices, then any graph obtained from G by contracting some of the edges can be also dominated by k vertices. What we need here is a structure similar to a grid to which every planar graph of large treewidth can be contracted. But this is exactly what Theorem 7.25 provides us!

We thus need to answer the following question: what is the minimum possible size of a dominating set in Γ_t? We do not know the exact formula (and this can be a difficult question to answer), but obtaining an asymptotic lower bound is easy. Every vertex except the lower-right corner (t, t) can dominate at most nine vertices. The lower-right corner vertex dominates only $4t - 4$ vertices. Thus, every dominating set of Γ_t is of size at least $\frac{t^2-4t+4}{9} = \Omega(t^2)$.

In this manner, using Theorem 7.25 we conclude that every connected planar graph with dominating set of size at most k has treewidth $\mathcal{O}(\sqrt{k})$. We can now combine this observation with the dynamic programming of Theorem 7.7. Using the same win/win approach, this gives us a $2^{\mathcal{O}(\sqrt{k})} \cdot n^{\mathcal{O}(1)}$-time algorithm for Planar Dominating Set. And again, the essential properties that we used are: (P1′) the size of a minimum solution in Γ_t is of order $\Omega(t^2)$; (P2′) the problem can be solved in single-exponential time when parameterized by the treewidth of the input graph; and (P3′) the problem is closed under edge contraction (that is, the solution does not increase when edges are contracted).

Formalizing the framework. We have already gathered enough intuition to be able to put the framework into more formal terms. In the following, we restrict our attention to vertex-subset problems. Edge-subset problems can be defined similarly and same arguments will work for them, hence we do not discuss them here.

Let ϕ be a computable function which takes on input a graph G and a set $S \subseteq V(G)$, and outputs **true** or **false**. The interpretation of ϕ is that it defines the space of *feasible solutions* S for a graph G, by returning a Boolean value denoting whether S is feasible or not. For example, for the Dominating Set problem we would have that $\phi(G, S) =$ **true** if and only if $N[S] = V(G)$. Similarly, for Independent Set we would have that $\phi(G, S) =$ **true** if and only if no two vertices of S are adjacent.

For a function ϕ, we define two parameterized problems, called *vertex-subset problems*: ϕ-Minimization and ϕ-Maximization. In both problems the input consists of a graph G and a parameter k. ϕ-Minimization asks whether there exists a set $S \subseteq V(G)$ such that $|S| \leq k$ and $\phi(G, S) =$ **true**. Similarly, ϕ-Maximization asks whether there exists a set $S \subseteq V(G)$ such that $|S| \geq k$ and $\phi(G, S) =$ **true**.

Obviously, problems like Dominating Set, Independent Set, or Vertex Cover are vertex-subset problems for appropriate functions ϕ. We note, however, that this notion captures also many other problems that, at first glance, have only little resemblance to the definition. A good example here is the familiar Cycle Packing problem. To see how Cycle Packing can be understood as a maximization vertex-subset problem, observe that a graph G contains k vertex-disjoint cycles if and only if there exists a set $S \subseteq V(G)$ of size at least k with the following property: one can find in G a family of vertex-disjoint cycles such that each vertex of S is contained in exactly one of them. Hence, we can set $\phi(G,S)$ to be true if and only if exactly this property is satisfied for G and S. Contrary to the examples of Dominating Set and Independent Set, in this case it is NP-hard to check whether $\phi(G,S)$ is true for a given graph G and set S. But since we require ϕ only to be computable, this definition shows that Cycle Packing is a vertex-subset problem. Another examples are Longest Path and Longest Cycle; recall that in these problems we ask for the existence of a path/cycle on at least k vertices in a given graph. Here, we can take $\phi(G,S)$ to be true if and only if $G[S]$ contains a Hamiltonian path/cycle.

Let us consider some ϕ-Minimization problem Q. Observe that $(G,k) \in Q$ implies that $(G,k') \in Q$ for all $k' \geq k$. Similarly, if Q is a ϕ-Maximization problem, then we have that $(G,k) \in Q$ implies that $(G,k') \in Q$ for all $k' \leq k$. Thus, the notion of optimality is well defined for vertex-subset problems.

Definition 7.26. For a ϕ-Minimization problem Q, we define

$$OPT_Q(G) = \min\{k \ : \ (G,k) \in Q\}.$$

If there is no k such that $(G,k) \in Q$, then we put $OPT_Q(G) = +\infty$.

For a ϕ-Maximization problem Q, we define

$$OPT_Q(G) = \max\{k \ : \ (G,k) \in Q\}.$$

If there is no k such that $(G,k) \in Q$, then we put $OPT_Q(G) = -\infty$.

We say that a (minimization or maximization) vertex-subset problem Q is *contraction-closed* if for every H that is a contraction of some graph G, we have that $OPT_Q(H) \leq OPT_Q(G)$. Similarly, a vertex-subset problem Q is *minor-closed* if the same inequality holds for all minors H of G. Observe that the property of being contraction-closed or minor-closed can be checked by examining single graph operations for these containment notions: a contraction of a single edge in the case of contractions, and vertex deletion, edge deletion, and edge contraction in the case of minors.

We are now ready to define bidimensional problems.

Definition 7.27 (Bidimensional problem). A vertex-subset problem Q is *bidimensional* if it is contraction-closed, and there exists a constant $c > 0$ such that $OPT_Q(\Gamma_t) \geq ct^2$ for every $t > 0$.

As already pointed out in Definition 7.27, for the sake of simplicity from now on we will focus only on *contraction bidimensionality.* That is, we will work with contraction-closed problems and graphs Γ_t, rather than with minor-closed problems and grids $\boxplus_t$. Being contraction-closed is a strictly weaker requirement than being minor-closed, whereas graphs Γ_t and $\boxplus_t$ do not differ that much when it comes to arguing about lower/upper bounds on the optimum value for them. For this reason, on planar graphs working with contraction bidimensionality is usually the preferable way. We remark here that differences between contraction bidimensionality and minor bidimensionality become more significant when one generalizes the theory to H-minor-free graphs. We briefly discuss this topic at the end of this section.

It is usually straightforward to determine whether a given problem Q is bidimensional according to Definition 7.27. For example, let us take $Q =$Vertex Cover. Contracting an edge does not increase the size of a minimum vertex cover, so the problem is contraction-closed. And as we already observed, $OPT_Q(\boxplus_t) \geq \lfloor t^2/2 \rfloor$. Thus also $OPT_Q(\Gamma_t) \geq \lfloor t^2/2 \rfloor$, so Vertex Cover is bidimensional. Similarly, Feedback Vertex Set, Induced Matching, Cycle Packing, Scattered Set for a fixed value of d, Longest Path, Dominating Set, and r-Center are bidimensional as well; see Exercise 7.40.

The crucial property that we used in our examples was that whenever the answer to the problem cannot be determined immediately, the input graph has treewidth roughly $\sqrt{k}$. This is exactly what makes bidimensionality useful for algorithmic applications.

Lemma 7.28 (Parameter-treewidth bound). *Let Q be a bidimensional problem. Then there exists a constant α_Q such that for any connected planar graph G it holds that $\mathrm{tw}(G) \leq \alpha_Q \cdot \sqrt{OPT_Q(G)}$. Furthermore, there exists a polynomial-time algorithm that for a given G constructs a tree decomposition of G of width at most $\alpha_Q \cdot \sqrt{OPT_Q(G)}$.*

Proof. Consider a bidimensional problem Q and a connected planar graph G. Let t be the maximum integer such that G contains Γ_t as a contraction. Since Q is bidimensional, it is also contraction-closed, and hence $OPT_Q(G) \geq OPT_Q(\Gamma_t) \geq ct^2$. By Theorem 7.25, the treewidth of G is at most $9t+5 \leq 14t$. Thus $OPT_Q(G) \geq c \cdot \frac{\mathrm{tw}(G)^2}{14^2}$ and the first statement of the lemma follows for $\alpha_Q = 14/\sqrt{c}$. To obtain the second, constructive statement, one can take $\alpha_Q = 15/\sqrt{c}$ and apply the algorithm of Theorem 7.25 for $\varepsilon = 1$ and increasing values of parameter t (starting from 1), up to the point when a tree decomposition is returned. $\square$

The next theorem follows almost directly from Lemma 7.28.

Theorem 7.29. *Let Q be a bidimensional problem such that Q can be solved in time $2^{\mathcal{O}(t)} \cdot n^{\mathcal{O}(1)}$ when a tree decomposition of the input graph G of width t is provided. Then Q is solvable in time $2^{\mathcal{O}(\sqrt{k})} \cdot n^{\mathcal{O}(1)}$ on connected planar graphs.*

Proof. The proof of the theorem is identical to the algorithms we described for Vertex Cover and Dominating Set. Let (G, k) be an input instance of Q. Using Lemma 7.28, we construct in polynomial time a tree decomposition G of width $t \leq \alpha_Q \cdot \sqrt{OPT_Q(G)}$. If $t > \alpha_Q\sqrt{k}$, then we infer that $k < OPT_Q(G)$. In such a situation we can immediately conclude that (G, k) is a no-instance of Q, provided Q is a minimization problem, or a yes-instance of Q, if Q is a maximization problem. Otherwise, we have in hand a tree decomposition of G of width at most $\alpha_Q\sqrt{k}$, and we can solve the problem in time $2^{\mathcal{O}(\sqrt{k})} \cdot n^{\mathcal{O}(1)}$ using the assumed treewidth-based algorithm. □

Observe that if the assumed algorithm working on a tree decomposition had a slightly different running time, say $t^{\mathcal{O}(t)} \cdot n^{\mathcal{O}(1)}$, then we would need to make a respective adjustment in the running time of the algorithm given by Theorem 7.29. In the aforementioned case we would obtain an algorithm with running time $k^{\mathcal{O}(\sqrt{k})} \cdot n^{\mathcal{O}(1)}$.

Let us remark that the requirement of connectivity of G in Theorem 7.29 is necessary. In Exercise 7.42 we give an example of a bidimensional problem Q such that Lemma 7.28 does not hold for Q and disconnected graphs. On the other hand, for most natural problems, in particular problems listed in Corollaries 7.30 and 7.31, we do not need the input graph G to be connected. This is due to the fact that each of these problems is monotone under removal of connected components. In other words, for every connected component C of G we have that $OPT_Q(G - C) \leq OPT_Q(G)$. It is easy to see that if problem Q has this property, then the requirement of connectivity can be lifted in Lemma 7.28 and Theorem 7.29.

We can now combine the facts that many considered problems are both bidimensional and solvable in single-exponential time when parameterized by treewidth, see Theorem 7.9 and Exercises 7.40 and 7.23. Thus, we obtain the following corollary of Theorem 7.29.

Corollary 7.30. *The following parameterized problems can be solved in time $2^{\mathcal{O}(\sqrt{k})}n^{\mathcal{O}(1)}$ on planar graphs:*

- Vertex Cover*,*
- Independent Set*,*
- Dominating Set*,*
- Scattered Set *for fixed d,*
- Induced Matching*, and*
- r-Center *for fixed r.*

Similarly, by combining Lemma 7.28 with Theorem 7.10, we obtain the following corollary.

Corollary 7.31. *The following parameterized problems can be solved in time $k^{\mathcal{O}(\sqrt{k})}n^{\mathcal{O}(1)}$ on planar graphs:*

- Feedback Vertex Set*,*

- Longest Path *and* Longest Cycle*,*
- Cycle Packing*,*
- Connected Vertex Cover*,*
- Connected Dominating Set*, and*
- Connected Feedback Vertex Set*.*

In Chapter 11 we will see how to improve the running time of dynamic programming on a tree decomposition to $2^{\mathcal{O}(t)} \cdot n^{\mathcal{O}(1)}$ for almost all the problems listed in Corollary 7.31, except Cycle Packing. Hence, for these problems the running time on planar graphs can be improved to $2^{\mathcal{O}(\sqrt{k})}n^{\mathcal{O}(1)}$, as stated in Corollary 11.14. Actually, this can be done also for Cycle Packing, even though in Chapter 14 we will learn that the running time of $t^{\mathcal{O}(t)} \cdot n^{\mathcal{O}(1)}$ on general graphs cannot be improved under reasonable complexity assumptions (see Theorem 14.19). We can namely use the fact that the algorithm is run on a *planar* graph of treewidth at most t, and this property can be exploited algorithmically via the technique of *Catalan structures*. This improvement, however, is beyond the scope of this book.

Beyond planarity. Let us briefly mention possible extensions of bidimensionality to more general classes of graphs. The planar excluded grid theorem (Theorem 7.23) can be generalized to graphs excluding some fixed graph H as a minor, i.e., H-minor-free graphs. More precisely, Demaine and Hajiaghayi [134] proved that for every fixed graph H and integer $t > 0$, every H-minor-free graph G of treewidth more than $\alpha_H t$ contains $\boxplus_t$ as a minor, where α_H is a constant depending on H only.

Using this, it is possible to show that the treewidth-parameter bound $\mathrm{tw}(G) \leq \alpha_Q \cdot \sqrt{OPT_Q(G)}$ holds for much more general classes of apex-minor-free graphs. An apex graph is a graph obtained from a planar graph G by adding one vertex and making it adjacent to an arbitrary subset of vertices of G. Then a class of graphs is apex-minor-free if every graph in this class does not contain some fixed apex graph as a minor. Thus for example, the bidimensional arguments imply a subexponential parameterized algorithm for Dominating Set on apex-minor-free graphs, but do not imply such an algorithm for general H-minor-free graphs. While Dominating Set is solvable in subexponential parameterized time on H-minor-free graphs, the algorithm requires additional ideas.

If we relax the notion of (contraction) bidimensionality to minor bidimensionality, i.e., we require that the problem Q is minor-closed and $OPT_Q(\boxplus_t) = \Omega(t^2)$, then for minor-bidimensional problems the treewidth-parameter bound holds even for H-minor-free graphs. Therefore for example Vertex Cover or Feedback Vertex Set admit subexponential parameterized algorithms on graphs excluding a fixed minor because they are minor-bidimensional.

7.7.3 Shifting technique

We now present another technique for obtaining fixed-parameter tractable algorithms for problems on planar graphs using treewidth. The main idea of the approach originates in the work on approximation schemes on planar graphs, pioneered in the 1980s by Baker. The methodology is widely used in modern approximation algorithms, and is called the *shifting technique*, or simply *Baker's technique.* In this section we present a parameterized counterpart of this framework.

For a vertex v of a graph G and integer $r \geq 0$, by G_v^r we denote the subgraph of G induced by vertices within distance at most r from v in G. By Theorem 7.25, we have the following corollary.

Corollary 7.32. *Let G be a planar graph, v be an arbitrary vertex, and r be a nonnegative integer. Then* $\mathrm{tw}(G_v^r) \leq 18r + 13$.

Proof. For the sake of contradiction, suppose that $\mathrm{tw}(G_v^r) \geq 18r + 14$. Since G_v^r is connected and planar, by Theorem 7.25 we infer that G_v^r can be contracted to Γ_{2r+1}. It is easy to see that for any vertex of Γ_{2r+1}, in particular for the vertex to which v was contracted, there is another vertex of Γ_{2r+1} at distance at least $r + 1$ from this vertex. Since contraction of edges does not increase the distances between vertices, this implies that there is a vertex of G_v^r at distance at least $r + 1$ from v, which is a contradiction. □

It is possible to prove a better bound on the dependence of treewidth on r in planar graphs than the one stated in Corollary 7.32. The following result is due to Robertson and Seymour.

Theorem 7.33 ([398]). *Let G be a planar graph, v be an arbitrary vertex, and r be a nonnegative integer. Then* $\mathrm{tw}(G_v^r) \leq 3r + 1$. *Moreover, a tree decomposition of G_v^r of width at most $3r+1$ can be constructed in polynomial time.*

By Theorem 7.33, we have the following corollary.

Corollary 7.34. *Let v be a vertex of a planar graph G, and for $i \geq 0$ let L_i be the set of vertices of G that are at distance exactly i from v. Then for any $i, j \geq 1$, the treewidth of the subgraph $G_{i,i+j-1} = G[L_i \cup L_{i+1} \cup \cdots \cup L_{i+j-1}]$ does not exceed $3j + 1$. Moreover, a tree decomposition of $G_{i,i+j-1}$ of width at most $3j + 1$ can be computed in polynomial time.*

Proof. Consider the graph G_v^{i+j-1}, and note that G_v^{i-1} is its induced subgraph that is connected. Contract the whole subgraph G_v^{i-1} to one vertex $v^\star$, and observe that the obtained planar graph H is exactly $G_{i,i+j-1}$ with additional vertex $v^\star$. However, in H all the vertices are at distance at most j from $v^\star$, since in G every vertex from $L_i \cup L_{i+1} \cup \cdots \cup L_{i+j-1}$ is at distance at most j from some vertex of L_{i-1}. By Theorem 7.33, we can compute in polynomial time a tree decomposition of H of width at most $3j + 1$. As $G_{i,i+j-1}$

is an induced subgraph of H, we can easily adjust it to a tree decomposition of $G_{i,i+j-1}$ of width at most $3j+1$. □

The basic idea behind the shifting technique is as follows.

- Pick a vertex v of the input planar graph G, and run a breadth-first search (BFS) from v in G.
- By Corollary 7.34, for any choice of $i, j \geq 1$, the treewidth of the subgraph $G_{i,i+j-1}$ induced by vertices on levels $i, i+1, \ldots, i+j-1$ of BFS does not exceed $3j+1$.
- Assume that in our problem we are looking for a set S of size k that has some property. Suppose for a moment that such S exists. If we take $j = k+1$, then we are sure that there exists a number $q \in \{0, 1, \ldots, k\}$ such that levels with indices of the form $a(k+1)+q$ for $a = 0, 1, 2, \ldots$ do not contain any vertex of S. We can remove these layers from G, thus obtaining a graph G_q that is a disjoint union of graphs $G_{(k+1)a+q+1,(k+1)(a+1)+q-1}$ for $a = 0, 1, 2, \ldots$, plus possibly G_v^{q-1} if $q > 0$. Note that G_q still contains the whole set S. However, using Theorem 7.33 and Corollary 7.34 we can construct a tree decomposition of G_q of width at most $3k+1$.
- To find a solution S, we iterate through all possible $q \in \{0, 1, \ldots, k\}$, and for each q we run a dynamic-programming algorithm on the obtained tree decomposition of G_q. If a solution exists, it survives in at least one graph G_q and can be uncovered by the algorithm.

We formalize the above idea in the following lemma.

Lemma 7.35. *Let G be a planar graph and let $k \geq 0$ be an integer. Then the vertex set of G can be partitioned into $k+1$ sets (possibly empty) in such a manner that any k of these sets induce a graph of treewidth at most $3k+1$ in G. Moreover, such a partition, together with tree decompositions of width at most $3k+1$ of respective graphs, can be found in polynomial time.*

Proof. Let us give a proof first for the case of connected graphs. Assume then that we are given a connected planar graph G and a nonnegative integer k. We select an arbitrary vertex $v \in V(G)$ and run a breadth-first search (BFS) from v. For $j \in \{0, \ldots, k\}$, we define S_j to be the set of vertices contained on levels $a(k+1)+j$ of the BFS, for $a = 0, 1, 2, \ldots$. Observe that thus $(S_j)_{0 \leq j \leq k}$ is a partition of $V(G)$. Moreover, the graph $G - S_j$ is a disjoint union of graphs $G_{(k+1)a+j+1,(k+1)(a+1)+j-1}$ for $a = 0, 1, 2, \ldots$, plus possibly G_v^{j-1} provided that $j > 0$. By Theorem 7.33 and Corollary 7.34, for each of these graphs we can construct a tree decomposition of width at most $3k+1$; connecting these tree decompositions arbitrarily yields a tree decomposition of $G - S_j$ of width at most $3k+1$. Hence, partition $(S_j)_{0 \leq j \leq k}$ satisfies the required property and we have proved lemma for connected graphs.

Let G be a (non-connected) graph and let $G_1, G_2, \dots, G_c$ be its connected components . Then we apply the statement to each connected component G_i separately, thus obtaining a partition $(S^i_j)_{0 \le j \le k}$ of $V(G_i)$ that satisfies the required property. Then for $j \in \{0, 1, \dots, k\}$ take $S_j = \bigcup_{i=1}^{c} S^i_j$. It is easy to verify that partition $(S_j)_{0 \le j \le k}$ of $V(G)$ satisfies the required property. □

Sometimes it is more convenient to use the following edge variant of Lemma 7.35. The proof of the following statement can be proved in a similar manner as Lemma 7.35, and we leave it as Exercise 7.47.

Lemma 7.36. *Let G be a planar graph and k be a nonnegative integer. Then the edge set of G can be partitioned into $k+1$ sets (possibly empty) such that any k of these sets induce a graph of treewidth at most $3k+4$ in G. Moreover, such a partition, together with tree decompositions of width at most $3k+4$ of respective graphs, can be found in polynomial time.*

We now give two examples of applications of the shifting technique. Our first example is Subgraph Isomorphism. In this problem we are given two graphs, *host* graph G and *pattern* graph H, and we ask whether H is isomorphic to a subgraph of G. Since Clique is a special case of Subgraph Isomorphism, the problem is W[1]-hard on general graphs when parameterized by the size of the pattern graph. It is possible to show using color coding that the problem can be solved in time $f(|V(H)|)|V(G)|^{\mathcal{O}(\mathrm{tw}(H))}$ for some function f, see Exercise 7.48. In other words, the problem is FPT if the pattern graph has constant treewidth. However, we now prove that if both the host and the pattern graph are planar, then the problem is FPT when parameterized by $|V(H)|$, without any further requirements.

As a building block here we need the following result about solving Subgraph Isomorphism on graphs of bounded treewidth. The lemma can be proved by either designing an explicit dynamic-programming algorithm, or using Courcelle's theorem. We leave its proof as Exercise 7.49.

Lemma 7.37. *There exists an algorithm that solves a given instance (G, H) of* Subgraph Isomorphism *in time $f(|V(H)|, t) \cdot |V(G)|^{\mathcal{O}(1)}$, where f is some function and t is the width of a given tree decomposition of G.*

Let us remark that the running time of the algorithm has to depend on $|V(H)|$; in other words, parameterization just by the treewidth will not work. It is in fact known that Subgraph Isomorphism is NP-hard on forests, i.e., graphs of treewidth 1. In bibliographic notes we discuss further literature on different parameterizations of Subgraph Isomorphism.

Let us note that if we search for a connected subgraph H, then Lemmas 7.33 and 7.37 are sufficient to find a solution: for each vertex v of a planar graph, we try if a subgraph induced by the vertices of the ball of radius $|V(H)|$ centered in v contains H as a subgraph. Because each such subgraph is of treewidth at most $3|V(H)|+1$, this will give a time $f(|V(H)|) \cdot |V(G)|^{\mathcal{O}(1)}$ algorithm solving Subgraph Isomorphism. However, if H is not connected, we need to use shifting technique.

Theorem 7.38. *There exists an algorithm that, given an instance (G, H) of* Subgraph Isomorphism *where G and H are planar, solves this instance in time $f(|V(H)|) \cdot |V(G)|^{\mathcal{O}(1)}$, for some function f.*

Proof. Let $k = |V(H)|$. Using Lemma 7.35, we construct in polynomial time a partition $S_0 \cup \cdots \cup S_k$ of $V(G)$ such that for every $i \in \{0, \ldots, k\}$, graph $G - S_i$ has a tree decomposition of width at most $3k + 1$. These tree decompositions are also provided by the algorithm of Lemma 7.35.

By Lemma 7.37, we can solve Subgraph Isomorphism in time $f(k) \cdot n^{\mathcal{O}(1)}$ on each $G - S_j$. Because we partition the vertex set of G into $k + 1$ sets, for every k-vertex subset X of G there exists $j \in \{0, \ldots, k\}$ such that $X \cap S_j = \emptyset$. Therefore, if G contains H as a subgraph, then for at least one value of j, $G - S_j$ also contains H as a subgraph. This means that by trying each of the graphs $G - S_j$ for $j \in \{0, \ldots, k\}$, we will find a copy of H in G, provided there exists one. $\square$

The running time of the algorithm of Theorem 7.38 depends on how fast we can solve Subgraph Isomorphism on graphs of bounded treewidth. In bibliographic notes we sketch the current state of the art on this issue.

Our second example concerns the Minimum Bisection problem. For a given n-vertex graph G and integer k, the task is to decide whether there exists a partition of $V(G)$ into sets A and B, such that $\lfloor n/2 \rfloor \leq |A|, |B| \leq \lceil n/2 \rceil$ and the number of edges with one endpoint in A and the second in B is at most k. In other words, we are looking for a balanced partition (A, B) with an (A, B)-cut of size at most k. Such a partition (A, B) will be called a *k-bisection* of G.

It will be convenient to work with a slightly more general variant of the problem. In the following, we will assume that G can be a multigraph, i.e., it can have multiple edges between the same pair of vertices. Moreover, the graph comes together with a weight function $\mathbf{w} : V(G) \to \mathbb{Z}_{\geq 0}$ on vertices, and from a k-bisection (A, B) we will require that $\lfloor \mathbf{w}(V(G))/2 \rfloor \leq \mathbf{w}(A), \mathbf{w}(B) \leq \lceil \mathbf{w}(V(G))/2 \rceil$. Of course, by putting unit weights we arrive at the original problem.

The goal is to prove that Minimum Bisection on planar graphs is FPT when parameterized by k. This time we shall use yet another variant of Lemma 7.35. The proof of the following result is slightly more difficult than that of Lemma 7.35 or Lemma 7.36. We leave it to the reader as Exercise 7.50.

Lemma 7.39. *Let G be a planar graph and k be a nonnegative integer. Then the edge set of G can be partitioned into $k + 1$ sets such that after contracting edges of any of these sets, the resulting graph admits a tree decomposition of width at most ck, for some constant $c > 0$. Moreover, such a partition, together with tree decompositions of width at most ck of respective graphs, can be found in polynomial time.*

As a building block for FPT algorithm on planar graphs, we need the following lemma about fixed parameterized tractability of Minimum Bisection on graphs of bounded treewidth. Its proof is again left to the reader as Exercise 7.51.

Lemma 7.40. Minimum Bisection *can be solved in time* $2^t \cdot W \cdot n^{\mathcal{O}(1)}$ *on an* n*-vertex multigraph given together with its tree decomposition of width* t*. Here,* W *is the maximum weight of a vertex.*

Note that here we are referring to tree decompositions of multigraphs, but the definitions of treewidth and tree decompositions can be naturally extended to multigraphs as well. We now proceed to the main result; note that again we state it for the generalized weighted problem.

Theorem 7.41. Minimum Bisection *on planar graphs can be solved in time* $2^{\mathcal{O}(k)} \cdot W \cdot n^{\mathcal{O}(1)}$*, where* W *is the maximum weight of a vertex.*

Proof. We use Lemma 7.39 to partition the set of edges of the input planar graph G into sets $S_0 \cup \cdots \cup S_k$. Note here that Lemma 7.39 is formally stated only for simple graphs, but we may extend it to multigraphs by putting copies of the same edge always inside the same set S_j.

Suppose that there exists a k-bisection (A, B) of G, and let F be the set of edges between A and B. Then at least one of the sets S_j is disjoint from F. Let us contract all the edges of S_j, keeping multiple edges but removing created loops. Moreover, whenever we contract some edge uv, we define the weight of the resulting vertex as $\mathbf{w}(u)+\mathbf{w}(v)$. Let G_j be the obtained multigraph. Since F is disjoint from S_j, during this contraction we could have just contracted some parts of $G[A]$ and some parts of $G[B]$. Therefore, the new multigraph G_j also admits a k-bisection (A', B'), where A', B' comprise vertices originating in subsets of A and B, respectively.

On the other hand, if for any G_j we find some k-bisection (A', B'), then uncontracting the edges of S_j yields a k-bisection (A, B) of G. These two observations show that G admits a k-bisection if and only if at least one of the multigraphs G_j does. However, Lemma 7.39 provided us a tree decomposition of each G_j of width $\mathcal{O}(k)$. Hence, we can apply the algorithm of Lemma 7.40 to each G_j, and thus solve the input instance (G, k) in time $2^{\mathcal{O}(k)} \cdot W \cdot n^{\mathcal{O}(1)}$. Note here that the maximum weight of a vertex in each G_j is at most nW. □

Let us remark that the only properties of planar graphs that we used here were Theorem 7.33 and the fact that planar graphs are closed under the operation of taking minors. We say that a class of graphs $\mathcal{G}$ is of *bounded local treewidth* if there exists function f such that for every graph $G \in \mathcal{G}$ and every vertex v of G, it holds that $\mathrm{tw}(G_v^r) \leq f(r)$. For example, Theorem 7.33 shows that planar graphs are of bounded local treewidth. Graphs of maximum degree d for a constant d are also of bounded local treewidth for the following reason: for every v and r, graph G_v^r contains at most $d^r + 1$ vertices, so also $\mathrm{tw}(G_v^r) \leq d^r$. However, this class is not closed under taking minors.

Our arguments used for planar graphs can be trivially extended to minor-closed graph classes of bounded local treewidth. It was shown by Eppstein [165] that a minor-closed class of graphs is of bounded local treewidth if and only if all its members exclude some fixed apex graph as a minor. Recall that an apex graph is a graph obtained from a planar graph G by adding one vertex and making it adjacent to an arbitrary subset of vertices of G. Then a class of graphs is apex-minor-free if every graph in this class does not contain some fixed apex graph as a minor. Demaine and Hajiaghayi [134] refined the result of Eppstein by showing that for every apex-minor-free graph class, function f in the definition of bounded local treewidth is actually linear.

*7.8 Irrelevant vertex technique

In the Planar Vertex Deletion problem we are given a graph G and an integer k. The question is whether there exists a vertex subset D of size at most k such that $G - D$ is planar. A set D with this property will be henceforth called a *planar deletion set*.

The class of planar graph is minor-closed and due to that Planar Vertex Deletion is a special case of $\mathcal{G}$ Vertex Deletion for $\mathcal{G}$ the class of planar graphs. By Theorem 6.17, we have that Planar Vertex Deletion is nonuniformly FPT. In this section we give a constructive FPT algorithm for Planar Vertex Deletion. However, more interesting than the algorithm itself is the method of obtaining it. The algorithm is namely based on the *irrelevant vertex technique.*

This technique originates from the work of Robertson and Seymour on their famous FPT algorithm for the Vertex Disjoint Paths problem. In this problem, we are given a graph G and a set of k pairs of terminal vertices, $\{(s_1, t_1), \ldots, (s_k, t_k)\}$, and the task is to find k vertex-disjoint paths connecting all pairs of terminals. The algorithm of Robertson and Seymour is very complicated and uses deep structural theorems from the theory of Graph Minors. On a very general level, the algorithm can be roughly summarized as follows.

As long as the treewidth of G is large in terms of k, it is possible to find a vertex v that is solution-irrelevant: every collection of k paths of any solution can be rerouted to an equivalent one that avoids v. Thus v can be safely removed from the graph without changing the answer to the instance. By repeating this argument exhaustively, we eventually reduce the treewidth of the graph to some function of k. Then the problem can be solved in FPT time using dynamic programming.

Since the work of Robertson and Seymour, the irrelevant vertex technique has been used multiple times in various parameterized algorithms. While the general win/win approach is natural, the devil lies in the detail. Most of the algorithms based on irrelevant edge/vertex techniques are very nontrivial and involve a large number of technical details. The algorithm for Planar Vertex Deletion is one of the simplest examples of applications of this approach, as it is possible to give an almost complete proof in just a few pages. Formally, we shall prove the following theorem.

Theorem 7.42. Planar Vertex Deletion *can be solved in time* $2^{\mathcal{O}(k^2 \log k)} n^{\mathcal{O}(1)}$.

The remaining part of this section is devoted to the proof of the theorem. We shall implement the following plan.

- We use iterative compression (see Chapter 4) in a standard manner and reduce the problem to Disjoint Planar Vertex Deletion: For a given graph G and vertex set D of size $k+1$ such that $G-D$ is planar, we ask if there is a set D' of size k such that $G-D'$ is planar and $D \cap D' = \emptyset$.
- The only black box ingredient which we do not explain is the fact that on graphs of treewidth t one can solve Disjoint Planar Vertex Deletion in time $2^{\mathcal{O}(t \log t)} n$ using dynamic programming.
- If the treewidth of $G-D$ is smaller than some threshold depending on k, then we solve Disjoint Planar Vertex Deletion by dynamic programming. Otherwise, $G-D$ contains a large grid as a minor.
- If the grid is sufficiently large, then it contains $k+2$ "concentric" cycles "centered" at some vertex v. Moreover, the outermost cycle C separates all the other cycles from the rest of the graph, and the connected component of $G-C$ containing all these cycles is planar. Then one can show that v is irrelevant and can be safely deleted from the graph; this is the most difficult part of the proof.

We start by introducing a criterion of planarity that will be used later. Let C be a simple cycle in graph G. A *C-bridge* in G is a subgraph of G which is either a single edge that is a chord of C (together with its endpoints), or a connected component of $G - V(C)$ together with all the edges connecting it with C (and their endpoints). Observe that if G is not connected, then by definition every connected component of G that does not contain C is a C-bridge. If B is a C-bridge, then the vertices of $V(C) \cap V(B)$ are the *attachment points* of B. Two C-bridges B_1, B_2 *overlap* if at least one of the following conditions is satisfied:

(a) B_1 and B_2 have at least three attachment points in common, or
(b) cycle C contains distinct vertices a, b, c, d (in this cyclic order) such that a and c are attachment points of B_1, while b and d are attachment points of B_2.

For a graph G with a cycle C, we define an *overlap graph* $O(G, C)$ that has the C-bridges in G as vertices, and two C-bridges are connected by an edge if they overlap. If G has a plane embedding, then in this embedding cycle C partitions the rest of the plane into two disjoint open sets, the *faces* of C. Observe that if two C-bridges B_1 and B_2 overlap, then they cannot be both drawn in the same face of C without intersection. This implies that if G is indeed planar, then the overlap graph $O(G,C)$ has to be bipartite: sides of the bipartition of $O(G,C)$ correspond to C-bridges that are placed inside and outside the cycle C.

The following criterion of planarity states that the condition that $O(G,C)$ is bipartite is not only necessary for G to be planar, but also sufficient. We leave its proof as Exercise 7.43.

Lemma 7.43. *Let C be a simple cycle in G. Then G is planar if and only if (i) for every C-bridge B, graph $C \cup B$ is planar, and (ii) the overlap graph $O(G,C)$ is bipartite.*

We now introduce the notation for separators in planar graphs. Let G be a connected graph, and let u, v be two distinct vertices. A set X is a *(u,v)-separator* if $u, v \notin X$, but every u-v path contains a vertex of X. Note that in this definition we assume that the separator does not contain either u or v, contrary to separators that we used in Section 7.6. The connected component of $G-X$ that contains v is called the *v-component*, and its vertex set will be denoted by $R(v,X)$. Equivalently, $R(v,X)$ is the set of all the vertices of G that are reachable from v via paths that avoid X. We say that X is a *connected separator* if $G[X]$ is connected, and is a *cyclic separator* if $G[X]$ contains a Hamiltonian cycle. Of course, every cyclic separator is also connected.

The following lemma will be a technical tool needed in delicate reasonings about finding an irrelevant vertex.

Lemma 7.44. *Let G be a connected graph, and suppose $u, v \in V(G)$ are two distinct vertices. Let X_1 and X_2 be two disjoint connected (u,v)-separators and let $R_i = R(v, X_i)$ for $i = 1, 2$. Then either $R_1 \cup X_1 \subseteq R_2$ or $R_2 \cup X_2 \subseteq R_1$.*

Proof. Let P be any walk in G that starts at v and ends in u. Since X_1 and X_2 separate u from v, P meets both a vertex of X_1 and a vertex of X_2. Let $h_1(P)$ be the first vertex on P that belongs to $X_1 \cup X_2$, and let $h_2(P)$ be the last such vertex; in this definition P is traversed from v to u.

We claim that if $h_1(P) \in X_1$ then $h_2(P) \in X_2$, and if $h_1(P) \in X_2$ then $h_2(P) \in X_1$. These statements are symmetric, so let us prove only the first implication. Assume, for the sake of contradiction, that $h_2(P) \in X_1$ (recall that X_1 and X_2 are disjoint). Then consider the following v-u walk in G: we first traverse P from v to $h_1(P)$, then by connectivity of X_1 we can travel through $G[X_1]$ to $h_2(P)$, and finally we traverse the end part of P from $h_2(P)$ to u. Since X_1 is disjoint from X_2, by the definition of $h_1(P)$ and $h_2(P)$ we infer that this walk avoids X_2. This is a contradiction with X_2 being a (u,v)-separator.

Let us fix any v-u walk P and suppose that $h_1(P) \in X_1$ and $h_2(P) \in X_2$. We will prove that in this case $R_1 \cup X_1 \subseteq R_2$; the second case when $h_1(P) \in X_2$ and $h_2(P) \in X_1$ leads to the conclusion that $R_2 \cup X_2 \subseteq R_1$ in a symmetric manner. Take any other v-u walk P'. We claim that for P' it also holds that $h_1(P') \in X_1$ and $h_2(P') \in X_2$. For the sake of contradiction, assume that $h_1(P') \in X_2$ and $h_2(P') \in X_1$. Consider the following v-u walk P'': we first traverse P from v to $h_1(P)$, then by connectivity of X_1 we can travel through $G[X_1]$ to $h_2(P')$, and finally we traverse the end part of P' from $h_2(P')$ to u. Again, by the same arguments as before walk P'' avoids X_2, which is a contradiction to X_2 being a (u,v)-separator.

We now know that for every v-u walk P it holds that $h_1(P) \in X_1$ and $h_2(P) \in X_2$. We shall prove that $R_1 \subseteq R_2$. Observe that this will prove also that $X_1 \subseteq R_2$, since X_1 is connected, disjoint from X_2, and adjacent to R_1 via at least one edge due to the connectivity of G. Take any $w \in R_1$; we need to prove that $w \in R_2$ as well. Let P_w be any v-w walk that is entirely contained in $G[R_1]$. Prolong P_w to a v-u walk P'_w by appending an arbitrarily chosen w-u walk. Observe now that no vertex of P_w can belong to X_2 — in such a situation we would have that $h_1(P'_w) \in X_2$, since no vertex of P_w belongs to X_1. Hence path P_w omits X_2, which proves that $w \in R_2$. As w was chosen arbitrarily, this concludes the proof. □

We are now ready to state the irrelevant vertex rule.

Definition 7.45 (Irrelevant vertex). Let G be a graph and k be an integer. A vertex v of G is called an *irrelevant vertex* if for every vertex set D of size at most k, $G - D$ is planar if and only if $G - (D \cup \{v\})$ is planar.

Thus if v is an irrelevant vertex, then (G,k) is a yes-instance if and only if $(G-v,k)$ is. The following lemma establishes a criterion of being an irrelevant vertex, and is the crucial ingredient of the algorithm.

Lemma 7.46. *Let (G,k) be an instance of* Planar Vertex Deletion. *Suppose v is a vertex of G such that there exists a vertex $u \neq v$ and a sequence of pairwise disjoint cyclic (v,u)-separators $X_1, X_2, \dots, X_{k+2}$ with the following properties:*

(a) for every $i \in \{1,\dots,k+1\}$, X_i is a subset of the $R(v,X_{i+1})$, and
(b) the graph $G[R(v,X_{k+2}) \cup X_{k+2}]$ is planar.

Then v is an irrelevant vertex for the instance (G,k).

Proof. Let v be a vertex of G that satisfies the conditions of the lemma. We show that for every set D of size at most k, $G - D$ is planar if and only if $G - (D \cup \{v\})$ is planar.

If $G - D$ is planar then clearly $G - (D \cup \{v\})$ is planar. Suppose then that $G - (D \cup \{v\})$ is planar. By the conditions of the lemma, there exists a vertex u, $u \neq v$, and vertex-disjoint cyclic (v,u)-separators $X_1, X_2, \dots, X_{k+2}$ satisfying properties (a) and (b). For $i = 1, 2, \dots, k+2$, let $R_i = R(v, X_i)$.

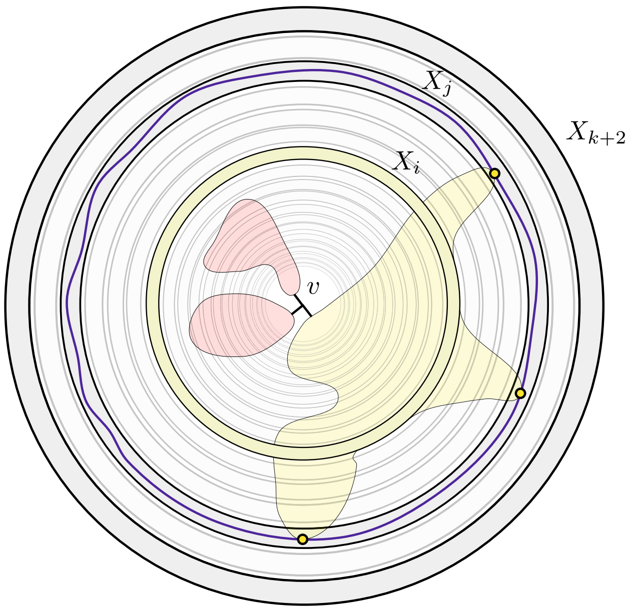


Fig. 7.8: Situation in the proof of Lemma 7.46. Cycle C is depicted in blue. C-bridges B_q in $G-(D\cup\{v\})$ that are originate in B_v are depicted in yellow and red. The yellow C-bridge is the one that contains X_i; it may not exist, but if it exists then it is unique. The other C-bridges (depicted red) are entirely contained in R_i, and hence they do not have any attachment points on C

Since separators X_i are pairwise disjoint and connected, from Lemma 7.44 and property (a) we infer the following chain of inclusions:

$$\{v\} \subseteq R_1 \subseteq R_1 \cup X_1 \subseteq R_2 \subseteq R_2 \cup X_2 \subseteq \ldots \subseteq R_{k+2} \subseteq R_{k+2} \cup X_{k+2}.$$

As $|D| \leq k$, there are two separators X_i and X_j that contain no vertices from D. Assume without loss of generality that $i < j$; then we have also $R_i \cup X_i \subseteq R_j$.

Let C be a Hamiltonian cycle in $G[X_j]$. In order to show that $G-D$ is planar, we want to apply the planarity criterion of Lemma 7.43 to cycle C and graph $G-D$. First of all, every C-bridge B in $G-D$ that does not contain v is also a C-bridge in $G-(D\cup\{v\})$, and hence $B\cup C$ is planar. Also, if B_v is a C-bridge of $G-D$ that contains v, then $B_v \cup C$ is a subgraph of $G[R_j \cup X_j]$, which in turn is a subgraph of the planar graph $G[R_{k+2} \cup X_{k+2}]$ (property (b)). Hence $B_v \cup C$ is also planar.

Now we focus on the overlap graph $O(G-D, C)$. Let B_v be the C-bridge of $G-D$ that contains v. Then $B_v = B_1 \cup \cdots \cup B_p \cup \{v\}$, where B_q for $q \in \{1, 2, \ldots, p\}$ are some C-bridges in $G-(D\cup\{v\})$. Consider one B_q. Since

$R_i \cup X_i \subseteq R_j$ and R_i is a connected component of $G - X_i$, it follows that B_q can have a neighbor in X_j only if B_q contains some vertex of X_i. Hence, every B_q that has an attachment point on C must intersect X_i. However, separator X_i is connected and disjoint from D, so there is a unique C-bridge in $G - (D \cup \{v\})$ that contains the whole X_i. Therefore, we infer that among $B_1, \ldots, B_p$ there can be at most one C-bridge that actually has some attachment points on C—this is the C-bridge that contains X_i—and all the other B_qs are completely contained in R_i. As these other C-bridges do not have attachment points, they are isolated vertices in $O(G - (D \cup \{v\}), X)$. On the other hand, the C-bridge B_q that contains X_i, provided that it exists, has the same attachment points on C as C-bridge B_v in $G - D$.

Concluding, graph $O(G - (D \cup \{v\}), X)$ can differ from $O(G - D, X)$ only by having some additional isolated vertices, which correspond to C-bridges adjacent to v that are contained in R_i. Because $O(G - (D \cup \{v\}), X)$ is bipartite, we have that $O(G - D, X)$ is also bipartite. Thus all the conditions required by Lemma 7.43 are satisfied, and we can conclude that $G - D$ is planar. □

Now that we are armed with the irrelevant vertex rule, we can proceed to the algorithm itself. Essentially, the strategy is to try to find a situation where Lemma 7.46 can be applied, provided that the treewidth of the graph is large. Note that, due to property (b) from the statement of Lemma 7.46, we will need to identify a reasonably large part of the graph that is planar. Therefore, it would be perfect if we could work on a graph where a large planar part has been already identified. However, this is exactly the structure given to us as an input in the problem Disjoint Planar Vertex Deletion. That is, $G - D$ is planar. Thus, to prove Theorem 7.42 it is sufficient to show the following lemma.

Lemma 7.47. Disjoint Planar Vertex Deletion *on an n-vertex graph can be solved in time $2^{\mathcal{O}(k^2 \log k)} n^{\mathcal{O}(1)}$. Moreover, in the case of a positive answer, the algorithm can also return a planar deletion set of size at most k.*

Proof. The idea of the algorithm is as follows. Graph $G - D$ is planar. Let us apply the algorithm of Theorem 7.23 to graph $G - D$ for $\varepsilon = \frac{1}{2}$ and some parameter $f(k)$ to be defined later. If we find a tree decomposition of $G - D$ of width at most $5f(k)$, then we can create a tree decomposition of G of width at most $5f(k) + |D| \leq 5f(k) + k + 1$ by simply adding D to every bag. In this situation we can use dynamic programming to solve the problem in FPT time. Otherwise, if the algorithm of Theorem 7.23 fails to construct a tree decomposition, then it returns an $f(k) \times f(k)$ grid minor in $G - D$. We will then show that G contains an irrelevant vertex which can be identified in polynomial time. Hence, we can identify and delete irrelevant vertices up to the point when the problem can be solved by dynamic programming on a tree decomposition of bounded width.

It is not difficult to show that Planar Vertex Deletion admits an FPT dynamic-programming algorithm when parameterized by the width of a given tree decomposition of the input graph. For example, one can make use of the optimization version of Courcelle's theorem (Theorem 7.12): we write an $\mathbf{MSO}_2$ formula $\varphi(D)$ for a free vertex set variable D, which verifies that $G - D$ is planar by checking that it contains neither a K_5-minor nor a $K_{3,3}$-minor. Then minimizing $|D|$ corresponds to finding the minimum possible size of a planar deletion set in G. In this argument we can also easily handle annotations in Disjoint Planar Vertex Deletion, since we may additionally require in the formula that D has to be disjoint from the set of forbidden vertices. This gives some FPT algorithm for Disjoint Planar Vertex Deletion. However, obtaining explicit and efficient bounds on the running time requires additional ideas and more technical effort. In our algorithm we shall use the following result, which we do not prove here.

Lemma 7.48 ([277]). Disjoint Planar Vertex Deletion *can be solved in time $2^{\mathcal{O}(t \log t)} n$ on an n-vertex graph given together with its tree decomposition of width at most t. Moreover, in the case of a positive answer, the algorithm can also return a planar deletion set D that has size at most k and is disjoint from the forbidden vertices.*

We have already all the tools to deal with the bounded-treewidth case, so now let us try to find a place to apply Lemma 7.46 if the treewidth is large. Let us fix $p = 2k + 5$ and $f(k) = (p + 4)(k + 2)$, and let H be the $f(k) \times f(k)$ grid. Recall that if the algorithm of Theorem 7.23 fails to construct a tree decomposition of small width of $G - D$, it provides a minor model of H in $G - D$. Let this minor model be $(I_w)_{w \in V(H)}$; recall that every branch set I_w is connected and all the branch sets are pairwise disjoint. Moreover, let G_H be the connected component of $G - D$ that contains H. From the graph G we construct a new graph G_c by performing the following steps:

- First, delete all the vertices of $G - D$ apart from the component G_H.
- Then, for every $w \in V(H)$ contract $G_H[I_w]$ into a single vertex, which will be called $\eta(w)$.
- Iteratively take a vertex $u \in V(G_H)$ that does not belong to any branch set I_w, and contract u onto any of its neighbor (such a neighbor exists due to connectivity of G_H). Perform this operation up to the point when there are no more original vertices of $V(G_H)$ left.

In this manner, $V(G_c)$ consists of set D and set $\{\eta(w) \ : \ w \in V(H)\}$ that induces in G_c a graph $\widehat{H}$ that is a supergraph of grid H. In other words, G_c consists of a partially triangulated $f(k) \times f(k)$ grid $\widehat{H}$ and a set of apices D that can have quite arbitrary neighborhoods in $\widehat{H}$. For $w \in V(H)$, let J_w be the set of vertices of G_H that got contracted onto w. Note that $I_w \subseteq J_w$, $G'[J_w]$ is connected, and $(J_w)_{w \in V(H)}$ forms a partition of $V(G_H)$.

After these operations we more-or-less see a large, flat area (planar) in the graph, which could possibly serve as a place where Lemma 7.46 is applied. However, we do not control the interaction between D and $\widehat{H}$. The goal now is to show that this interaction cannot be too complicated, and in fact there

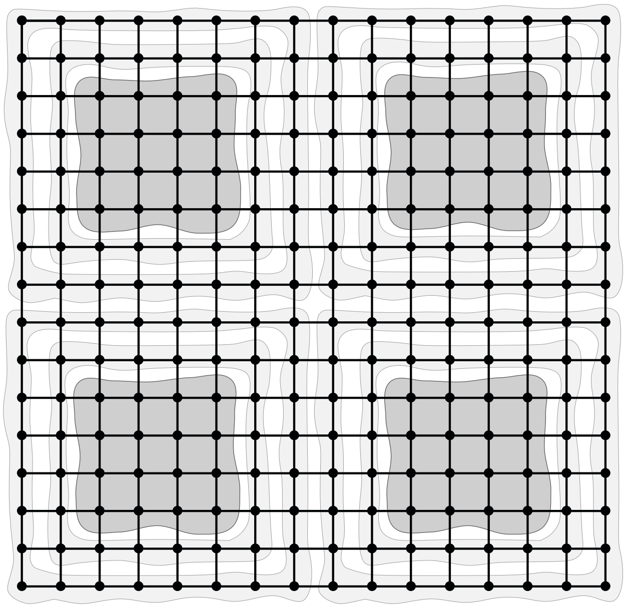


Fig. 7.9: Choosing the family of subgrids $\mathcal{P}$ in a large grid H. The grids of $\mathcal{P}$ are depicted in dark grey, and their 1-frames and 2-frames are depicted in light grey

exists a large enough subgrid of $\widehat{H}$ that is completely nonadjacent to D. From now on, by somewhat abusing the notation we will identify grid H with its subgraph in $\widehat{H}$ via mapping η; note that thus $V(H) = V(\widehat{H})$, and $\widehat{H}$ can only have some additional edges.

Let us partition grid H into smaller grids. Since H is an $f(k) \times f(k)$ grid for $(p+4)(k+2)$, we can partition it into $(k+2)^2$ grids of size $(p+4) \times (p+4)$. Now for each of these subgrids perform twice the following operation: delete the whole boundary of the grid, thus "peeling" twice a frame around the grid. If M is the obtained $p \times p$ subgrid, then by 2-*frame* and 1-*frame* of M, denoted by $F_2(M)$ and $F_1(M)$, we mean the boundaries removed in the first and in the second step, respectively. Thus, the 2-frame of M induces in H a cycle of length $4(p+3)$ around M, and the 1-frame induces a cycle of length $4(p+1)$ around M. See Fig. 7.9 for a visualization of this operation. Let $\mathcal{P}$ be the set of obtained $p \times p$ grids M (i.e., those after peeling the frames).

We now prove the following claim, which bounds the interaction between a single vertex of D and the grids from $\mathcal{P}$.

Claim 7.49. *If* (G, k, D) *is a yes-instance of* Disjoint Planar Vertex Deletion, *then in graph* G_c *no vertex of* D *can have neighbors in more than* $k+1$ *grids from* $\mathcal{P}$.

Proof. For the sake of contradiction, assume that there is a set $D' \subseteq V(G) \setminus D$ such that $|D'| \leq k$ and $G - D'$ is planar, and suppose that there exists a vertex $x \in D$ that in G_c has neighbors in $k+2$ of $p \times p$ grids from $\mathcal{P}$. We now examine what happens with set D' during the operations that transform G to G_c. Let D_c be the set of those vertices of G_c onto which at least one vertex of D' was contracted. In other words, whenever we contract an edge, we consider the resulting vertex to be in D' if and only if one of the original endpoints was in D', and we define D_c to be the set of vertices of G_c that end up being in D' after all these contractions. It follows that $|D_c| \leq |D'| \leq k$, D_c is still disjoint from D, and $G_c - D_c$ is a planar graph.

Since D_c is of size at most k, there exist at least two $p \times p$ subgrids from $\mathcal{P}$, say Y and Z, which are adjacent to x in G_c and such that D_c does not contain any vertex of Y, or of Z, or of 1- and 2-frames of Y and Z. Let y be an arbitrary neighbor of x in Y, and similarly define $z \in Z$.

It is easy to see that for any two distinct $p \times p$ grids of $\mathcal{P}$, there are at least $p > k$ vertex-disjoint paths in H that connect these grids. One of these paths does not contain any vertex from D_c. Thus, in H there exists a path P from y to z that contains no vertex of D_c: we first travel inside Y from y to the border of Y, then we use a D_c-free path connecting Y and Z, and finally we get to z by travelling inside Z.

Let us stop for a moment and examine the current situation (see also Fig. 7.10). We have identified two subgrids Y and Z that are completely free from D_c together with their frames. Moreover, these two subgrids can be connected inside H by a path that is also free from D_c. So far the picture is planar, but recall that we have an additional vertex x that is adjacent both to $y \in Y$ and to $z \in Z$. This vertex thus creates a "handle" that connects two completely different regions of grid H. As we assumed that $x \in D$ and D_c is disjoint from D, we have that $x \notin D_c$, and hence this additional handle is also disjoint from D_c. Now the intuition is that the obtained structure is inherently non-planar, and therefore it cannot be contained in the planar graph $G_c - D_c$. To prove this formally, the easiest way is to exhibit a model of a K_5-minor in G_c that does not use any vertex of D_c. The construction is depicted in Fig. 7.10.

We start with constructing a minor of K_5 without one edge in $Y \cup F_1(Y) \cup F_2(Y)$. Recall that x is adjacent to $y \in Y$ and $z \in Z$. Define the following branch sets $I_1, I_2, \ldots, I_5$. First, take $I_1 = \{y\}$. Then, it is easy to see that one can partition $(Y \cup F_1(Y)) \setminus \{y\}$ into three connected sets I_2, I_3, I_4 such that each of them is adjacent both to y and to $F_2(Y)$. Finally, take $I_5 = F_2(Y)$. Thus all the pairs of branch sets are adjacent, apart from I_1 and I_5. To create this additional adjacency, we use vertex x. More precisely, we extend the branch set I_1 by including x, z, and all the vertices from the path P traversed from z up to the point when it meets $F_2(Y)$. In this manner, branch set I_1 is still connected, and it becomes adjacent to I_5. In this construction we used only vertices that do not belong to D_c, which means that $I_1, I_2, \ldots, I_5$

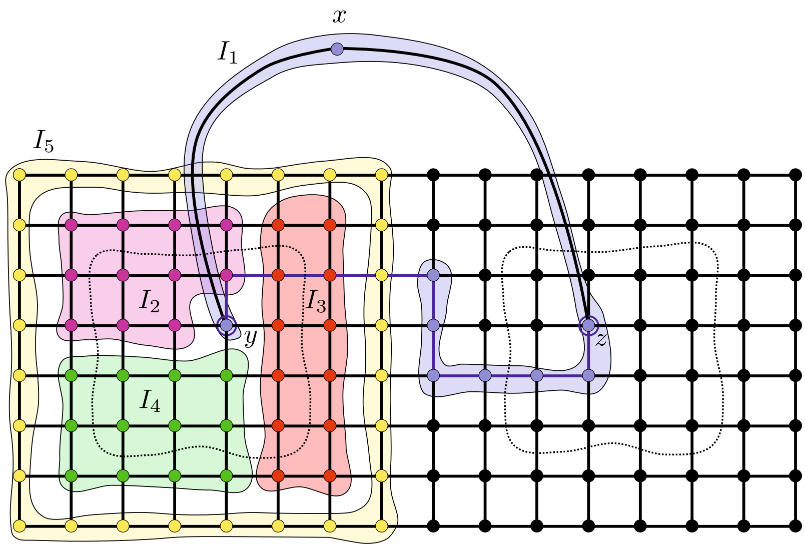


Fig. 7.10: Construction of a K_5-minor using x and adjacent subgrids Y and Z. Path P connecting y and z is depicted in blue.

form a K_5-minor model in $G_c - D_c$. This is a contradiction with D_c being a planar deletion set for G_c. ⌟

By Claim 7.49, in graph G_c every vertex of D has neighbors in at most $k+1$ subgrids from $\mathcal{P}$, or otherwise we may immediately terminate the subproblem by providing a negative answer. Since $|\mathcal{P}| = (k+2)^2$ and $|D| \leq k+1$, this means that we can find at least one $p \times p$ subgrid $M \in \mathcal{P}$ that has no neighbors in D. Let v_c be the middle vertex of M, and let v be any vertex of J_{v_c}. We claim that v is irrelevant, and to prove this we will need to verify the prerequisites of Lemma 7.46.

Since $p = 2k+5$, we can find $k+2$ vertex sets $N_1, N_2, \ldots, N_{k+2} \subseteq V(M)$ such that N_i induces the i-th cycle in M around v_c, counting from v_c. In other words, sets $N_{k+2}, N_{k+1}, \ldots, N_1$ are obtained from M by iteratively taking the boundary of the grid as the next N_i, and removing it, up to the point when only v_c is left. If we now uncontract the vertices of N_i, then we obtain a connected set $Y_i = \bigcup_{w \in V(N_i)} J_w$. For each $i \in \{1, 2, \ldots, k+2\}$, let C_i be a cycle inside $G'[Y_i]$ that visits consecutive branch sets J_w for $w \in V(N_i)$ in the same order as the cycle induced by N_i in M. Let $X_i = V(C_i)$, and let us fix any vertex $u \in V(G_H) \setminus \bigcup_{w \in V(M)} J_w$; such a vertex exists since M is not the only grid from $\mathcal{P}$.

Claim 7.50. *Vertices v and u and sets $X_1, X_2, \ldots, X_{k+2}$ satisfy the prerequisites of Lemma 7.46 for the instance (G,k) of* Planar Vertex Deletion. *Consequently, vertex v is irrelevant for (G,k).*

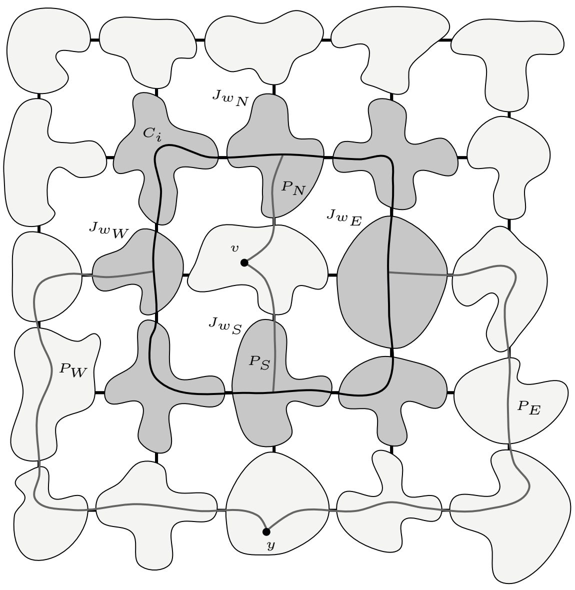


Fig. 7.11: The crucial argument in the proof of Claim 7.50: paths P_N, P_E, P_S, and P_W certify that v and y have to be placed on different faces of C_i in any planar drawing of G_H

Proof. The two prerequisites that are trivial are that each X_i induces a graph with a Hamiltonian cycle, and that $X_i \subseteq R(v, X_{i+1})$. The first follows directly from the definition, and the second follows from the existence of connections between branch sets $(J_w)_{w \in V(H)}$ implied by M.

Therefore, we need to prove that (i) each X_i is indeed a (u,v)-separator, and (ii) $G'[X_{k+2} \cup R(v, X_{k+2})]$ is planar. For $i \in \{0,1,2,\dots,k+2\}$, let $N_{\leq i} = \{v\} \cup N_1 \cup N_2 \cup \ldots \cup N_i$ and let W_i be the set of vertices of G_H contained in branch sets J_w for $w \in N_{\leq i}$. To prove (i) and (ii), we shall show that for each $i \geq 1$, it holds that $R(v, X_i) \subseteq W_i$. In other words, all the vertices that can be connected to v via paths avoiding X_i are actually contained in G_H, and moreover get contracted onto vertices contained in the union of the deepest i layers in M. Note that this statement implies (i), since $u \notin W_{k+2}$, and also it implies (ii), since then $G[X_{k+2} \cup R(v, X_{k+2})]$ is an induced subgraph of the planar graph $G - D$.

Consider any $y \in V(G_H) \setminus W_i$; then in G_c vertex y gets contracted onto some $w_y \notin N_{\leq i}$. Choose four vertices w_N, w_E, w_S and w_W, placed in the interiors of the north, east, south, and west side of the cycle $M[N_i]$, respectively; see Fig. 7.11. Since M does not touch the boundary of H (it has at least two frames peeled), it can be easily seen that in G_H we can find four paths P_N, P_E, P_S and P_W such that the following holds:

- P_W and P_E connect y with the cycle C_i such that both P_E and P_W travel only through branch sets J_w for $w \notin N_{\leq i}$, apart from J_{w_E} in the case of P_E, and J_{w_W} in the case of P_W. Then P_E meets C_i inside J_{w_E} and P_W meets C_i inside J_{w_W}.
- P_N and P_S connect v with the cycle C_i such that both P_N and P_S travel only through branch sets J_w for $w \in N_{\leq i-1}$, apart from J_{w_N} in the case of P_N, and J_{w_S} in the case of P_S. Then P_N meets C_i inside J_{w_N} and P_S meets C_i inside J_{w_S}.

Note that the family of branch sets visited by $P_W \cup P_E$ is disjoint from the family of branch sets visited by $P_N \cup P_S$. Hence, subgraphs $P_W \cup P_E$ and $P_N \cup P_S$ are vertex-disjoint.

Consider now subgraph $G_P = P_N \cup P_E \cup P_S \cup P_W \cup C_i$ of G_H. G_P is obviously planar as a subgraph of a planar graph, and it has two C_i-bridges: the first $P_N \cup P_S$ contains v, and the second $P_E \cup P_W$ contains y. Each of these C_i-bridges has two attachment points, and all these four attachment points interleave. It follows that for every planar embedding of G_P, vertices v and y must be drawn on different faces of the cycle C_i. Since G_P is a subgraph of G_H, the same holds for G_H as well.

We are now ready to prove that $R(v, X_i) \subseteq W_i$. For the sake of contradiction, suppose that some vertex $y \notin W_i$ can be reached from v via a path P that avoids X_i. By taking the first vertex outside W_i on P, we may assume without loss of generality that all the other vertices on P belong to W_i. Let y' be the predecessor of y on P; then $y' \in W_i$. Now examine where y can be localized in graph G. It cannot belong to D, since we know that in G_c all the vertices of D are not adjacent to any vertex of M, and in particular not to the vertex onto which y' was contracted. Also, obviously it cannot belong to a different component of $G - D$ than G_H, since $y' \in W_i \subseteq V(G_H)$. Therefore, we are left with the case when $y \in V(G_H) \setminus W_i$. But then we know that v and y must be drawn on different faces of C_i, so every path contained in G_H that connects v and y has to intersect X_i. This should in particular hold for P, a contradiction. ⌟

By Claim 7.50 we conclude that vertex v is irrelevant, so we can safely delete it and proceed. Note that by the definition of being irrelevant, removal of v is also safe in the disjoint variant of the problem that we are currently solving.

The reader may wonder now why we gave so elaborate a proof of Claim 7.50, even though its correctness is "obvious from the picture". Well, as the proof shows, formal justification of this fact is far from being trivial, and requires a lot of attention to avoid hand-waving. This is a very common phenomenon when working with planar graphs: facts that seem straightforward in a picture, turn out to be troublesome, or even incorrect, when a formal argument needs to be given. A good approach here is to try to identify the key formal argument that exploits planarity. In our case this argument was the observation that in G_H, set X_i has to separate v from every vertex $y \in V(G_H) \setminus W_i$, because v and y have to lie on different faces of C_i in every planar embedding of G_H. Once the key argument is identified and understood, the rest of the proof follows easily.

To summarize, we solve Disjoint Planar Vertex Deletion with the following algorithm.

- In polynomial time we are able either to conclude that this is a no-instance, or to identify an irrelevant vertex and delete it. We repeat this procedure until the treewidth of $G - D$ becomes at most $5f(k)$, and the corresponding tree decomposition can be constructed.
- When we have at hand a tree decomposition of $G - D$ of width at most $5f(k)$, we use Lemma 7.48 to solve the subproblem in time $2^{\mathcal{O}(f(k) \log f(k))} n$.

Since $f(k) = \mathcal{O}(k^2)$, the total running time of our algorithm solving Disjoint Planar Vertex Deletion is $2^{\mathcal{O}(f(k) \log f(k))} \cdot n^{\mathcal{O}(1)} = 2^{\mathcal{O}(k^2 \log k)} \cdot n^{\mathcal{O}(1)}$. □

As discussed before, Lemma 7.47 with a standard application of the iterative compression technique (see Section 4.1.1) concludes the proof of Theorem 7.42. We remark that with more ideas, the running time of the algorithm can be improved; see the bibliographic notes.

7.9 Beyond treewidth

Given all the presented applications of treewidth, it is natural to ask if one could design other measures of structural complexity of graphs that would be equally useful from the algorithmic viewpoint. So far it seems that treewidth and pathwidth are the most successful concepts, but several other approaches also turned out to be fruitful in different settings. In this section we review briefly two examples of other width measures: branchwidth and rankwidth. While the first one is tightly related to treewidth, the second one goes beyond it.

Branch decompositions and f-width. It will be convenient to introduce a more generic approach to defining structural width measures of combinatorial objects. Assume we are working with some finite universe U, and suppose we are given a function f defined on subsets of U that, for a subset $X \subseteq U$, is supposed to measure the complexity of interaction between X and $U \setminus X$. More precisely, $f\colon 2^U \to \mathbb{R}_{\geq 0}$ is a function that satisfies the following two conditions:

- *Symmetry*: for each $X \subseteq U$, it holds that $f(X) = f(U \setminus X)$;
- *Fairness*: $f(\emptyset) = f(U) = 0$.

Function f will be also called a *cut function*, or a *connectivity function*.

Given some cut function f on U, we may define a class of structural decompositions of U. A *branch decomposition* of U is a pair (T, η), where T is a tree with $|U|$ leaves whose all internal nodes have degree 3, and η is a bijection from U to the leaves of T. Consider any edge $e \in E(T)$. If we remove e from T, then T gets split into two subtrees, and thus the set of leaves of T is also partitioned into two sets. Leaves of T correspond one-to-one to elements of U, so let (X_e, Y_e) be the corresponding partition of U. We define the *width* of edge e in (T, η) as $f(X_e) = f(Y_e)$; this equality holds due to the symmetry condition on f. The *width* of (T, η) is the maximum width among its edges. The *f-width* of set U, denoted by $w_f(U)$, is equal to the minimum possible width of its branch decomposition. If $|U| = 1$ then there is no decomposition, so we put $w_f(U) = 0$.

This definition is generic in the sense that we may consider different combinatorial objects U and various cut functions f, thus obtaining a full variety of possible structural parameters. Both branchwidth and rankwidth are defined as f-widths for appropriately chosen U and f. However, the definition of f-width can be used to define structural width parameters of not only graphs, but also hypergraphs and matroids. Also, one often assumes that the cut function f is additionally *submodular*, i.e., it satisfies property $f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y)$ for all $X, Y \subseteq U$. This property provides powerful tools for approximating the f-width of a given set. In the bibliographic notes we give some pointers to literature on this subject.

Branchwidth. To define branchwidth of a graph G, we put $U = E(G)$, the edge set of G. For every $X \subseteq E(G)$, its *border* $\delta(X)$ is the set of those vertices of G that are incident both to an edge of X and to an edge of $E(G) \setminus X$. The *branchwidth* of G, denoted by $\mathrm{bw}(G)$, is the f-width of $U = E(G)$ with cut function $f(X) = |\delta(X)|$.

It turns out that branchwidth is in some sense an equivalent definition of treewidth, expressed in the formalism of branch decompositions. More precisely, for every graph G with $\mathrm{bw}(G) > 1$ it holds that

$$\mathrm{bw}(G) \leq \mathrm{tw}(G) + 1 \leq \frac{3}{2}\,\mathrm{bw}(G). \tag{7.13}$$

In many situations branchwidth turns out to be more convenient to work with than treewidth, for example when it comes to some dynamic-programming algorithms. More importantly, branchwidth seems to behave more robustly on planar graphs. In particular, branchwidth of a planar graph can be computed in polynomial time, whereas the computational complexity of treewidth on planar graphs remains a longstanding open question. Also, in the proof of the Planar Excluded Grid Theorem (Theorem 7.23) it is more convenient to relate the size of the largest grid minor in a graph to its branchwidth rather than to treewidth, for instance when one tries to refine/improve constants. Actually, the version of Gu and Tamaki that we gave here was originally stated for branchwidth, and then the bound was translated for treewidth using (7.13).

It is worth mentioning that branchwidth does not use the notion of vertices and hence generalizes to matroids while treewidth is tied to vertices.

Rankwidth. The motivation of rankwidth comes from the observation that many computational problems are tractable on classes of graphs that are dense, but structured. An obvious example of such graphs are complete graphs, but one can also allow a little bit more freedom in the structure. For instance, take the class of *cographs*, that is, graphs that can be constructed from single vertices using two types of operations: (a) taking a disjoint union of two constructed cographs, and (b) taking a disjoint union of two constructed cographs and adding a complete bipartite graph between their vertex sets.

This recursive definition of cographs allows us to solve many computational problems using, again, the principle of dynamic programming. Essentially, we exploit the fact that the *algebraic structure* of cographs is simple — the adjacency relation between pieces in their decomposition is trivial: full or empty. However, cographs are dense in the sense that the number of edges may be quadratic in the number of vertices. Hence, their treewidth can be even linear, and all the tools that we developed in this chapter may not be applicable at all. Therefore, it is natural to introduce a new type of structural decomposition, where the main measure of width would not be the cardinality of a separator, but rather the complexity of the adjacency relation between a part of the decomposition and the rest of the graph.

Historically, the first width parameter whose goal was to capture this phenomenon was *cliquewidth.* We do not give here its formal definition due to its technicality. Intuitively, a graph is of cliquewidth at most k if it can be built from single vertices by consecutively joining already constructed parts of the graph, and in each constructed part the vertices can be partitioned into at most k types such that vertices of the same type will be indistinguishable in later steps of the construction. The combinatorics of cliquewidth, while convenient for designing dynamic-programming algorithms, turned out to be very difficult to handle from the point of view of computing or approximating this graph parameter. For this reason, rankwidth has been introduced. Rankwidth is equivalent to cliquewidth in the sense that its value is both lower- and upper-bounded by some functions of cliquewidth. However, for rankwidth it is much easier to design exact and approximation algorithms.

We now proceed to formal definitions. For a vertex set $X \subseteq V(G)$, we define its *cut-rank* $\rho_G(X)$ as follows. Consider an $|X| \times |V(G) \setminus X|$ matrix $B_G(X) = (b_{x,y})_{x \in X,\, y \in V(G) \setminus X}$ with rows indexed by vertices of X and columns indexed by vertices of $V(G) \setminus X$. Entry $b_{x,y}$ is equal to 1 if x and y are adjacent, and 0 otherwise. Thus, $B_G(X)$ is exactly the adjacency matrix of the bipartite graph induced by G between X and $V(G) \setminus X$. We define $\rho_G(X)$ to be the rank of $B_G(X)$, when treated as a matrix over the binary field GF(2). Then the *rankwidth* of G, denoted by $\mathrm{rw}(G)$, is equal to the ρ_G-width of $V(G)$.

It is easy to prove that for any graph G it holds that $\mathrm{rw}(G) \leq \mathrm{tw}(G)$, see Exercise 7.53. However, as expected, treewidth cannot be bounded by any function of rankwidth. To see this, observe that the rankwidth of a complete graph on n vertices is at most 1, while its treewidth is $n-1$. Hence, classes of graphs that have bounded treewidth have also bounded rankwidth, but not vice versa.

Designing dynamic-programming routines for graphs of bounded rankwidth are similar to those of treewidth. Instead of exploiting the property that the already processed part of the graph communicates with the rest via a small separator, we use the fact that the matrix encoding adjacency relation between the processed part and the rest has small rank. This enables us to partition the processed vertices into a bounded number of classes that have exactly the same neighborhood in the rest of the graph. For some problems, vertices of the same type can be treated as indistinguishable, which leads to reducing the number of states of the dynamic program to a function of k.

As in the case of treewidth, dynamic-programming algorithms on graphs of bounded rankwidth can be explained by a meta-theorem similar to Courcelle's theorem. For this, we need to weaken the $\mathbf{MSO}_2$ logic that we described in Section 7.4.1 to $\mathbf{MSO}_1$ logic, where we forbid quantifying over subsets of edges of the graph. Then, analogues of Theorems 7.11 and 7.12 hold for $\mathbf{MSO}_1$ and graphs of bounded rankwidth. It should not be very surprising that rankwidth, as a parameter upper-bounded by treewidth, supports fixed-parameter tractability of a smaller class of problems than treewidth. Indeed, some problems whose fixed-parameter tractability when parameterized by treewidth follows from Theorem 7.12, turn out to be W[1]-hard when parameterized by rankwidth, and hence unlikely to be FPT. Examples include Hamiltonian Cycle and Edge Dominating Set; again, it is not a coincidence that these are problems where quantification over a subset of edges is essential.

When it comes to the complexity of computing rankwidth, the situation is very similar to treewidth. Hliněný and Oum [263] gave an $f(k) \cdot n^{\mathcal{O}(1)}$-time algorithm computing a branch decomposition of rankwidth at most k, or correctly reporting that the rankwidth of a given graph is more than k. An approximation algorithm with factor 3 for rankwidth, running in time $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$, was given by Oum [377].

Exercises

7.1. Prove Lemma 7.2.

7.2. Prove Lemma 7.4.

7.3 (☠). Find an example showing that the bound $\mathcal{O}(k|V(G)|)$ on the number of nodes of a nice tree decomposition cannot be strengthened to $\mathcal{O}(|V(G)|)$.

7.4. Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of graph G, t be a node of T, and X_t be the corresponding bag. Show that for every connected component C of $G - X_t$, the vertices of C are contained in bags of exactly one of the connected components of $T - t$.

7.5. Prove Lemma 7.3.

7.6. Prove that every clique of a graph is contained in some bag of its tree decomposition. Infer that $\mathrm{tw}(G) \geq \omega(G) - 1$, where $\omega(G)$ denotes the maximum size of a clique in G.

7.7 (✎). Show that treewidth is a minor-monotone parameter, i.e., for every minor H of a graph G, $\mathrm{tw}(H) \leq \mathrm{tw}(G)$.

7.8 (✎). What is the treewidth of (a) a complete graph; (b) a complete bipartite graph; (c) a forest; (d) a cycle?

7.9 (✎). Show that the treewidth of a graph G is equal to the maximum treewidth of its biconnected components.

7.10 (☠). Show that the pathwidth of an n-vertex tree is at most $\lceil \log n \rceil$. Construct a class of trees of pathwidth k and $\mathcal{O}(3^k)$ vertices.

7.11 (☠). Prove that a graph has treewidth at most 2 if and only if it does not contain K_4 as a minor.

7.12. A graph is outerplanar if it can be embedded in the plane in such manner that all its vertices are on one face. What values can treewidth of an outerplanar graph have?

7.13 (✎). Prove that the treewidth of a simple graph cannot increase after subdividing any of its edges. Show that in the case of multigraphs the same holds, with the exception that the treewidth can possibly increase from 1 to 2.

7.14. A graph G is called *d-degenerate* if every subgraph of G contains a vertex of degree at most d. Prove that graphs of treewidth k are k-degenerate.

7.15. Let G be an n-vertex graph of treewidth at most k. Prove that the number of edges in G is at most kn.

7.16 (☠). For a graph G given together with its tree decomposition of width t, construct in time $t^{\mathcal{O}(1)}n$ a data structure such that for any two vertices $x, y \in V(G)$, it is possible to check in time $\mathcal{O}(t)$ if x and y are adjacent. You should *not* use any results on hashing, hash tables, etc.

7.17 (✎). Remodel the dynamic-programming algorithm of Section 7.3.1 so that it uses the following definition of the value of a state: for $Y \subseteq X_t$, value $c[t, Y]$ is equal to the maximum possible weight of an independent set in $G[V_t \setminus Y]$. How complicated are the recursive formulas, compared to the algorithm given in Section 7.3.1?

7.18. Construct the remaining algorithms listed in Theorem 7.9. That is, show algorithms that, given an n-vertex graph together with its tree decomposition of width at most k, solve:

- Odd Cycle Transversal in time $3^k \cdot k^{\mathcal{O}(1)} \cdot n$,
- MaxCut in time $2^k \cdot k^{\mathcal{O}(1)} \cdot n$,
- q-Coloring in time $q^k \cdot k^{\mathcal{O}(1)} \cdot n$.

7.19. Construct the remaining algorithms listed in Theorem 7.10. That is, show that the following problems can be solved in time $k^{\mathcal{O}(k)} \cdot n$ on an n-vertex graph given together with its tree decomposition of width at most k:

- Feedback Vertex Set,
- Hamiltonian Path and Longest Path,
- Hamiltonian Cycle and Longest Cycle,
- Chromatic Number,
- Cycle Packing,
- Connected Vertex Cover,
- Connected Dominating Set,
- Connected Feedback Vertex Set.

7.20. List Coloring is a generalization of Vertex Coloring: given a graph G, a set of colors C, and a list function $L : V(G) \to 2^C$ (that is, a subset of colors $L(v)$ for each vertex v), the task is to assign a color $c(v) \in L(v)$ to each vertex $v \in V(G)$ such that adjacent vertices receive different colors. Show that on an n-vertex graph G, List Coloring can be solved in time $n^{\mathcal{O}(\mathrm{tw}(G))}$.

7.21 (☠). Consider the following problem: given a graph G, find a minimum set of vertices $X \subseteq V(G)$ such that $G - X$ does not contain a cycle on four vertices as a subgraph. Show how to solve this problem in time $2^{\mathcal{O}(k^2)} n^{\mathcal{O}(1)}$, where k is the treewidth of G.

7.22 (☠). Consider the following problem: given a graph G, find an induced subgraph H of G of maximum possible number of vertices, such that the size of the largest independent set in H is strictly smaller than the size of the largest independent set in G. Show how to solve this problem in time $2^{2^{\mathcal{O}(k)}} n^{\mathcal{O}(1)}$, where k is the treewidth of G.

7.23. In the following problems, assume that the input graph G is given together with its tree decomposition of width at most k.

- In the Induced Matching problem, given a graph G and an integer ℓ, we ask if there is a subset of 2ℓ vertices in G inducing a matching. Show that Induced Matching can be solved in time $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$.
- (☠) For a fixed integer $r \geq 1$, the r-Center problem, given a graph G and an integer ℓ, asks to find ℓ vertices such that every other vertex of G is at distance at most r from some of these vertices. Show that r-Center can be solved in time $(r+1)^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$.
- (☠) Scattered Set, given a graph G and integers ℓ and d, asks for at least ℓ vertices that are at pairwise distance at least d. Show that the problem can be solved in time $d^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$.

7.24. Obtain an algorithm for Vertex Cover running in time $1.3803^k k^{\mathcal{O}(1)} + \mathcal{O}(m\sqrt{n})$ by combining branching on degree 4 vertices, the $2k$ vertex kernel of Theorem 2.21, and the fact that a graph on n vertices of maximum degree 3 has pathwidth at most $\frac{n}{6} + o(n)$, and a path decomposition of such width can be constructed in polynomial time (see [195]).

7.25. Show that for a fixed graph H, the property "graph G does not contain H as a minor" is expressible in $\mathbf{MSO}_2$.

7.26. Using the cops and robber game, give an algorithm deciding in time $n^{\mathcal{O}(k)}$ if the treewidth of a graph G is at most k.

7.27 (✎). Show that every interval graph has no induced cycles of length more than 3.

7.28. This exercise consists of the crucial steps used to prove Theorem 7.16.

1. For a pair of vertices u, v from the same connected component of graph G, a vertex set S is a *(u, v)-separator* if $u, v \notin S$ and u and v are in different connected components of $G - S$. A (u, v)-separator is *minimal*, if it does not contain any other (u, v)-separator as a proper subset. Finally, a set S is a *minimal separator* if S is a minimal (u, v)-separator for some $u, v \in V(G)$. Let us remark that a minimal separator S can properly contain another minimal separator S'; this can happen if S' separates other pair of vertices than S. A connected component C of $G - S$ is *full* if $S = N(C)$. Show that every minimal separator S has at least two full components.

2. Show that every minimal separator of a chordal graph is a clique.
3. (Dirac's Lemma) A vertex v is *simplicial* if its closed neighborhood $N[v]$ is a clique. Show that every chordal graph G on at least two vertices has at least two simplicial vertices. Moreover, if G is not complete, then it has at least two nonadjacent simplicial vertices.
4. Prove that every chordal graph G admits a tree decomposition such that every bag of the decomposition is a maximal clique of G.
5. Prove Theorem 7.16.

7.29. We define a *k-tree* inductively. A clique on $k+1$ vertices is a k-tree. A new k-tree G can be obtained from a smaller k-tree G' by adding a new vertex and making it adjacent to k vertices of G' that form a clique in G'. Show that every k-tree is a chordal graph of treewidth k. Prove that for every graph G and integer k, G is a subgraph of a k-tree if and only if $\mathrm{tw}(G) \le k$.

7.30. Show that every maximal treewidth k-graph G, i.e., a graph such that adding any missing edge to G increases its treewidth, is a *k-tree*.

7.31. We say that a graph G is an *intersection graph of subtrees of a tree* if there exists a tree H and a collection $(H_v)_{v \in V(G)}$ of subtrees of H such that $uv \in E(G)$ if and only if $V(H_u) \cap V(H_v) \neq \emptyset$. Show that a graph is chordal if and only if it is an intersection graph of subtrees of a tree.

7.32. Let G be an n-vertex graph, and let $\sigma = (v_1, v_2, \ldots, v_n)$ be an ordering of its vertices. We define the *width* of σ, denoted by t_σ, as follows:

$$t_\sigma = \max_{i=0,1,\ldots,n} |\partial(\{v_1, v_2, \ldots, v_i\})|.$$

The *vertex separation number* of a graph G, denoted by $\mathrm{vsn}(G)$, is equal to the minimum possible width of an ordering of $V(G)$. Prove that for every graph G, its vertex separation number is equal to its pathwidth.

7.33 (✎). It seems that in the algorithm of Theorem 7.18 we could directly use $\frac{1}{2}$-balanced separators given by Lemma 7.19 instead of $\frac{2}{3}$-balanced separations given by Lemma 7.20, and in this manner we could reduce the approximation factor from 4 to 3. Explain what is the problem with this approach.

7.34. An n-vertex graph G is called an *α-edge-expander* if for every set $X \subseteq V(G)$ of size at most $n/2$ there are at least $\alpha|X|$ edges of G that have exactly one endpoint in X. Show that the treewidth of an n-vertex d-regular α-edge-expander is $\Omega(n\alpha/d)$ (in particular, linear in n if α and d are constants).

7.35. Show that the dependency on k in the Excluded Grid Theorem needs to be $\Omega(k^2)$. That is, show a graph of treewidth $\Omega(k^2)$ that does not contain a $k \times k$ grid as a minor.

7.36 (☠). Let H be a planar graph. Show that there is a constant c_H such that the treewidth of every H-minor-free graph is at most c_H. Show that planarity requirement in the statement of the exercise is crucial, i.e., that K_5-minor-free graphs can be of any treewidth.

7.37 (☠). Prove that for every $t > 1$, $\boxplus_t$ contains a bramble of order $t+1$ and thus $\mathrm{tw}(\boxplus_t) = t$.

7.38. Prove the following version of the classic result of Lipton and Tarjan on separators in planar graphs. For any planar n-vertex graph G and $W \subseteq V(G)$, there is a set $S \subseteq V(G)$ of size at most $\frac{9}{2}\sqrt{n+1}$ such that every connected component of $G - S$ contains at most $\frac{|W|}{2}$ vertices of W.

7.39 (✎). Using Corollary 7.24, prove that on an n-vertex planar graph G problems listed in Theorem 7.9 can be solved in time $2^{\mathcal{O}(\sqrt{n})}$, while problems listed in Theorem 7.10 can be solved in time $2^{\mathcal{O}(\sqrt{n}\log n)}$.

7.40. Show that the following problems are bidimensional: Feedback Vertex Set, Induced Matching, Cycle Packing, Scattered Set for a fixed value of d, Longest Path, Dominating Set, and r-Center for a fixed r.

7.41. Show that Scattered Set is FPT on planar graphs, when parameterized by $k+d$. For a constant d, give an algorithm with running time $2^{\mathcal{O}(\sqrt{k})}n^{\mathcal{O}(1)}$.

7.42. Define a vertex-subset maximization problem Q with the following property: for a graph G with the maximum size of an independent set $\alpha(G)$ and the number of isolated vertices $\iota(G)$, it holds that $OPT_Q(G) = \max\{0, \alpha(G) - 2\iota(G)\}$. Prove that Q is bidimensional and construct a family of planar graphs G_k such that for every $k \geq 1$, $OPT_Q(G_k) = 0$ and $\mathrm{tw}(G_k) \geq k$. Infer that in Lemma 7.28 the connectivity requirement is necessary.

7.43. Prove Lemma 7.43.

7.44 (☠). The input of the Partial Vertex Cover problem is a graph G with two integers k and s, and the task is to check if G contains a set of at most k vertices that cover at least s edges. Show that on planar graphs Partial Vertex Cover is FPT when parameterized by k. Give a subexponential $2^{\mathcal{O}(\sqrt{k})}n^{\mathcal{O}(1)}$-time algorithm for Partial Vertex Cover on planar graphs.

7.45 (☠). In the Tree Spanner problem, given a connected graph G and an integer k, the task is to decide if G contains a spanning tree H such that for every pair of vertices u, v of G, the distance between u and v in H is at most k times the distance between u and v in G. Show that on planar graphs Tree Spanner is FPT when parameterized by k.

7.46. Prove that the bounds of Corollary 7.32 and Theorem 7.33 cannot be strengthened to hold for pathwidth instead of treewidth. In other words, find an example of a planar graph of constant diameter that has unbounded pathwidth.

7.47. Prove Lemma 7.36.

7.48 (☠). Show that Subgraph Isomorphism can be solved in time $f(|V(H)|)|V(G)|^{\mathcal{O}(\mathrm{tw}(H))}$ for some function f.

7.49. Prove Lemma 7.37.

7.50 (☠). Prove Lemma 7.39.

7.51. Prove Lemma 7.40.

7.52 (✎). Prove that cographs have rankwidth at most 1.

7.53. Prove that for any graph G it holds that $\mathrm{rw}(G) \leq \mathrm{tw}(G) + 1$.

7.54. A *rooted forest* is a union of pairwise disjoint rooted trees. A *depth* of a rooted forest is the maximum number of vertices in any leaf-to-root path. An *embedding of a graph G into a rooted forest H* is an injective function $f : V(G) \to V(H)$ such that for every edge $uv \in E(G)$, $f(u)$ is a descendant of $f(v)$ or $f(v)$ is a descendant of $f(u)$. The *treedepth* of a graph G is equal to the minimum integer d, for which there exists a rooted forest H of depth d and an embedding of G into H.

Show that the pathwidth of a nonempty graph is always smaller than its treedepth.

7.55 (☠). Let G be a graph and consider the following cut function $\mu_G(X)$ defined for $X \subseteq V(G)$. We look at the bipartite graph $G_X = (V(G), E(X, V(G) \setminus X))$ induced in G by the edges between X and $V(G) \setminus X$. Then $\mu_G(X)$ is equal to the size of the maximum matching in this graph. The *MM-width* (*maximum-matching-width*) of a graph G, denoted by $\mathrm{mmw}(G)$, is equal to the μ_G-width of $V(G)$.

Prove that there exist constants K_1, K_2 such that for every graph G it holds that $K_1 \cdot \mathrm{tw}(G) \leq \mathrm{mmw}(G) \leq K_2 \cdot \mathrm{tw}(G)$.

Hints

7.1 Let $\mathcal{P} = (X_1, X_2, \ldots, X_r)$ be a path decomposition. First, add bags $X_0 = X_{r+1} = \emptyset$ at the beginning and at the end of the sequence. Second, for every $0 \leq i \leq r$, insert a few bags between X_i and X_{i+1}: first forget, one by one, all elements of $X_i \setminus X_{i+1}$, and then introduce, one by one, all elements of $X_{i+1} \setminus X_i$. Finally, collapse neighboring equal bags.

7.2 Proceed in a way similar to Exercise 7.1. By introducing some intermediate introduce and forget bags, ensure that every node of the tree with at least two children has its bag equal to all bags of its children. Then, split such a node into a number of join nodes.

7.3 Fix some integers k and n, where n is much larger than k. Consider a graph G consisting of:

1. a clique K on k vertices;
2. n vertices $a_1, a_2, \ldots, a_n$, where each a_i is adjacent to all the vertices of K;
3. n vertices $b_1, b_2, \ldots, b_n$, where each b_i is adjacent only to vertex a_i.

Argue that in any nice tree decomposition of G of width at most k, every vertex of K is introduced roughly n times, once for every pair a_i, b_i.

7.4 Let $v \in C$ and assume v belongs to some bag of a connected component T_v of $T - t$. By (T3), and since $v \notin X_t$, *all* bags that contain v are bags at some nodes of T_v. Together with (T2), this implies that for every edge $uv \in E(G - X_t)$, both u and v need to appear in bags of only one connected component of $T - t$.

7.5 Let $S = X_a \cap X_b$. Pick any $v_a \in V_a \setminus S$ and $v_b \in V_b \setminus S$. If v_a belong to some bag of T_b, then, by (T3), $v_a \in X_a$ and $v_a \in X_b$, hence $v_a \in S$, a contradiction. Thus, v_a appears only in some bags of T_a; similarly we infer that v_b appears only in some bags of T_b. Hence, by (T2), $v_a v_b \notin E(G)$.

7.6 Let C be the vertex set of a clique in G. Use Lemma 7.3: for every edge $st \in E(T)$ of a tree decomposition $(T, \{X_t\}_{t \in V(T)})$, $C \subseteq V_s$ or $C \subseteq V_t$ (where V_s and V_t are defined as in Lemma 7.3 and Exercise 7.5). Orient the edge st towards an endpoint $\alpha \in \{s, t\}$ such that $C \subseteq V_\alpha$. Argue that, if all edges of $E(T)$ have been oriented in such a manner, then C is contained in X_α for any $\alpha \in V(T)$ with outdegree zero, and that at least one such α exists.

7.7 Let H be a minor of G, let $(V_h)_{h \in V(H)}$ be a minor model of H in G, and let $(T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of G. For every $t \in V(T)$, let $X'_t = \{h \in V(H) : V_h \cap X_t \neq \emptyset\}$. Argue that $(T, \{X'_t\}_{t \in V(T)})$ is a tree decomposition of H of width not larger than the width of $(T, \{X_t\}_{t \in V(T)})$.

7.8 The treewidth of K_n is $n - 1$, the lower bound is provided by Exercise 7.6. The treewidth of $K_{a,b}$ is $\min(a, b)$, the lower bound is provided by Exercise 7.7 since $K_{\min(a,b)+1}$ is a minor of $K_{a,b}$. The treewidth of a forest is 1 if it contains at least one edge.

7.9 Observe that it is easy to combine tree decompositions of biconnected components of a graph into one tree decomposition, since their bags can coincide only in single vertices.

7.10 For the upper bound, consider the following procedure. In an n-vertex tree G, find a vertex v such that every connected components of $G - v$ has at most $n/2$ vertices. Recursively compute path decompositions of the components of $G - v$, each of width at most $\lceil \log n/2 \rceil = \lceil \log n \rceil - 1$. Arrange them one after another, adding v to every bag in the decomposition.

For the lower bound, using graph searching argue that if a tree G has a vertex v such that at least three connected component of $G - v$ have pathwidth at least k, then G has pathwidth at least $k + 1$.

7.11 By Exercise 7.7, and since $\mathrm{tw}(K_4) = 3$, we have that every graph containing K_4 as a minor has treewidth at least 3. The other direction is much more difficult: you need to argue, by some case analysis, that in a bramble of order 4 you can always find the desired minor. Alternatively, you may use the decomposition of graphs into 3-connected components of Tutte; see [110]. If you follow this direction, you will essentially need to prove the following statement: every 3-connected graph that has more than three vertices contains K_4 as a minor.

7.12 An isolated vertex, a tree and a cycle are outerplanar graphs, so the treewidth of an outerplanar graph may be equal to 0, 1, or 2. Argue that it is always at most 2, for example in the following way.

1. Assume that all vertices of an outerplanar graph G lie on the infinite face.
2. By Exercise 7.9, assume G is biconnected.
3. Triangulate G, that is, add some edges to G, drawn inside finite faces of G, such that every finite face of G is a triangle.
4. Let G^* be the dual of G, and let $f^* \in V(G^*)$ be the infinite face. Observe that $G^* - f^*$ is a tree. Argue that, if you assign to every $f \in V(G^* - f^*)$ a bag X_f consisting of the three vertices of G that lie on the face f, then you obtain a valid tree decomposition of G.

Alternatively, one may prove that outerplanar graphs are closed under taking minors, and use Exercise 7.11 combined with an observation that K_4 is not outerplanar.

7.14 Since graphs of treewidth at most k are closed under taking subgraphs, it suffices to show that every graph of treewidth at most k contains a vertex of degree at most k. Consider a nice tree decomposition of a graph of treewidth at most k, and examine a vertex whose forget node is furthest from the root.

7.15 Prove that a d-degenerate n-vertex graph has at most dn edges, and use Exercise 7.14.

7.16 *Solution 1*: Turn the given tree decomposition into a nice tree decomposition $(T, \{X_t\}_{t \in V(T)})$ of G of width t. For every $v \in V(G)$, let $t(v)$ be the uppermost bag in T that contains v. Observe that, for every $uv \in V(G)$, either $u \in X_{t(v)}$ or $v \in X_{t(u)}$. Hence, it suffices to remember $A_v := N_G(v) \cap X_{t(v)}$ for every $v \in V(G)$, and, for every query (u, v), check if $u \in A_v$ or $v \in A_u$.

Solution 2: Prove that for every d-degenerate graph G one can in linear time compute an orientation of edges of G such that every vertex has outdegree at most d (while the indegree can be unbounded). Then for every vertex $v \in V(G)$ we can remember its outneighbors in this orientation. To check whether u and v are adjacent, it suffices to verify whether u is among the outneighbors of v (at most d checks) or v is among the outneighbors of u (again, at most d checks). This approach works for d-degenerate graphs, and by Exercise 7.14 we know that graphs of treewidth at most k are k-degenerate.

7.19 For Chromatic Number, argue that you need at most $k + 1$ colors. For other problems, proceed similarly as in the dynamic-programming algorithm for Steiner Tree, given in Section 7.3.3. In the cases of HAMILTONIAN PATH, LONGEST PATH, LONGEST CYCLE, and CYCLE PACKING, you will need matchings between vertices of the bag instead of general partitions of the bag.

7.21 In the state of the dynamic program, keep for every pair u, v of vertices in a bag, whether there exists a common neighbor of u and v among the forgotten vertices that was not hit by the solution. It is quite nontrivial that this information is sufficient for ensuring that all C_4s are hit, so be very careful when arguing that your algorithm is correct.

7.22 Consider a node t of a tree decomposition $(T, \{X_t\}_{t \in V(T)})$ of the input graph G. Let V_t be the union of all bags of descendants of t in T. In the dynamic-programming algorithm, for every $S \subseteq X_t$ and every family $\mathcal{A}$ of subsets of $X_t \setminus S$, we would like to compute a minimum cardinality of a set $\widehat{S} \subseteq V_t$ such that $\widehat{S} \cap X_t = S$ and, moreover, the following holds: for every independent set I in $(G - \widehat{S})[V_t]$ such that there does not exist an independent set I' in $G[V_t]$ with $V_t \cap I = V_t \cap I'$ and $|I| < |I'|$, the set $V_t \cap I$ belongs to $\mathcal{A}$.

7.23 For INDUCED MATCHING, define the state by partitioning the bag at t into three sets: (a) vertices that have to be matched inside G_t; (b) vertices that will not be matched, and play no role; (c) vertices that are "reserved" for matching edges that are yet to be introduced, and hence we require them be nonadjacent to any vertex picked to be matched, or reserved. For later points, think of the vertices of type (c) as sort of a "prediction": the state of the dynamic program encapsulates prediction of what will be the behavior in the rest of the graph, and requires the partial solution to be prepared for this behavior.

For r-CENTER, use the prediction method: for every vertex of the bag, encapsulate in the dynamic programming state (a) what the distance is to the nearest center from a partial solution inside G_t (this part is called *history*), and (b) what the distance will be to the nearest center in the whole graph, after the whole solution is uncovered (this part is called *prediction*). Remember to check in forget nodes that the prediction is consistent with the final history.

For SCATTERED SET, use the same prediction approach as for r-CENTER.

7.25 The idea is to encode the statement that G contains a minor model of H. We first quantify the existence of vertex sets V_h for $h \in V(H)$. Then, we need to check that (a) every V_h is connected; (b) the sets V_h are pairwise disjoint; (c) if $h_1 h_2 \in E(H)$, then there exists $v_1 v_2 \in E(G)$ with $v_1 \in V_{h_1}$ and $v_2 \in V_{h_2}$. All three properties are easy to express in $\mathbf{MSO}_2$.

7.26 Let us first introduce a classic approach to resolving problems about games on graphs. The idea is to represent all configurations of the game and their dependencies by an auxiliary game played on a directed graph called the *arena*. The node set V of the arena is partitioned into two sets of nodes V_1 and V_2. It also has a specified node $v \in V$ corresponding to the initial position of the game, and a subset of nodes $F \subseteq V$ corresponding to the final positions. In the auxiliary game there are two players, Player 1 and Player 2, who alternately move a token along arcs of the arena. The game starts by placing the token on v, and then the game alternates in rounds. At each round, if the token is on a vertex from V_i, $i = 1, 2$, then the i-th player slides the token from its current position along an arc to a new position, where the choice of the arc depends on the player. Player 1 wins if she manages to slide the token to a node belonging to F, and Player 2 wins if she can avoid this indefinitely.

Prove that there exists an algorithm with running time depending polynomially on the size of the arena, which determines which player wins the game. Then take the cops-and-robber game on the input n-vertex graph G, and encode it as an instance of the auxiliary game on an arena of size $n^{\mathcal{O}(k)}$. Positions of the arena should reflect possible configurations in the cops-and-robber game, i.e., locations of the cops and of the robber, whereas arcs should reflect their possible actions.

7.28

1. If S is a minimal (u, v)-separator, then the connected components of $G-S$ that contain u and v are full.
2. Use the fact that every minimal separator S has at least two full components.
3. Use induction on the number of vertices in G. If G has two nonadjacent vertices v_1 and v_2, then a minimal (v_1, v_2)-separator S is a clique. Use the induction assumption to find a simplicial vertex in each of the full components.
4. Let v be a simplicial vertex of G. Use induction on the number of vertices n and construct a tree decomposition of G from a tree decomposition of $G - v$.
5. $(i) \Rightarrow (ii)$. Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of width k. Let H be the supergraph of G obtained from G by transforming every bag of $\mathcal{T}$ into a clique. Then $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ is also a tree decomposition of H of width k, and thus the maximum clique size of H does not exceed $k+1$. Show that H is chordal. $(ii) \Rightarrow (iii)$. Here you need to construct a search strategy for the cops. This strategy imitates the strategy of two cops on a tree. $(iii) \Rightarrow (iv)$. We have discussed in the chapter that a bramble of order more than $k+1$ gives a strategy for the robber to avoid $k+1$ cops. $(iv) \Rightarrow (i)$. This is the (harder) half of Theorem 7.15.

7.29 Proceed by induction, using Theorem 7.16.

7.31 In one direction, argue that if G is an intersection graph of subtrees of a tree, then every cycle in G of length at least 4 has a chord. In the other direction, use the fact that every chordal graph G has a tree decomposition $(T, \{X_t\}_{t \in V(T)})$ where every bag induces a clique (Exercise 7.28). Define $H_v = T[\{t \in V(T) \ : \ v \in X_t\}]$ for every $v \in V(G)$. Argue that the tree T with subtrees $(H_v)_{v \in V(G)}$ witnesses that G is an intersection graph of subtrees of a tree.

7.32 Prove two inequalities, that $\mathrm{vsn}(G) \le \mathrm{pw}(G)$ and that $\mathrm{pw}(G) \le \mathrm{vsn}(G)$. In both cases, you can make a direct construction that takes an ordering/path decomposition of optimum width, and builds the second object of the same width. However, it can also be convenient to use other characterizations of pathwidth, given in Theorem 7.14. For instance, you may give a strategy for searchers that exploits the existence of a path decomposition of small width.

7.34 Let G be an n-vertex d-regular α-edge-expander and let X be a $\frac{1}{2}$-balanced separator in G for a uniform weight function, given by Lemma 7.19. Let $D_1, \ldots, D_r$ be the connected components of $G - X$. Since $|D_i| \le n/2$ for every $1 \le i \le r$, we have that at least $\alpha|D_i|$ edges connect D_i and X. However, since G is d-regular, at most $d|X|$ edges leave X. Consequently,

$$\begin{aligned}\sum_{i=1}^{r} \alpha|D_i| &\le d|X|,\\ \alpha(n - |X|) &\le d|X|,\\ n\frac{\alpha}{d+\alpha} &\le |X|.\end{aligned}$$

By Lemma 7.19, $n\frac{\alpha}{d+\alpha} - 1$ is a lower bound for the treewidth of G.

7.35 Consider a clique on $k^2 - 1$ vertices.

7.36 Show that planar graph H is a minor of a $|V(H)|^{\mathcal{O}(1)} \times |V(H)|^{\mathcal{O}(1)}$ grid and then use the Excluded Grid Theorem.

7.37 In Section 7.7.1 we have already given a bramble of order t. Play with this example to squeeze one more element of the bramble.

7.38 Use Lemma 7.19 and Corollary 7.24.

7.41 For an FPT algorithm parameterized by $k + d$, you may use either bidimensionality, or a modification of the shifting technique where you remove d consecutive layers. For a subexponential algorithm for a constant d, use bidimensionality.

7.44 To obtain any fixed-parameter tractable algorithm for parameterization by k, you can use the shifting technique. Getting a subexponential one is more tricky.

The problem is not bidimensional, however, it can be reduced to a bidimensional problem. We order the vertices $V(G) = \{v_1, v_2, \ldots, v_n\}$ nonincreasingly according to their degrees. Show that, if (G, k, s) is a yes-instance, then there is a solution X such that for some r it holds that $X \subseteq \{v_1, v_2, \ldots, v_r\}$, and, moreover, X is a dominating set in the graph $G[\{v_1, v_2, \ldots, v_r\}]$. After guessing the number r, we can use bidimensionality.

7.45 Prove that a sufficiently large grid (in terms of k) does not admit a k-spanner, and hence, intuitively, every yes-instance of the problem should have bounded treewidth. Also finding a dynamic programming on a tree decomposition is nontrivial. Here, you may find it useful to prove that it suffices to prove the spanner property only on single edges. A caveat: the problem is not contraction-closed; it is possible to contract edges and increase the parameter. While grid arguments will work at the end, we have to be careful here.

7.46 Consider a tree obtained from Exercise 7.10, equipped with a universal vertex.

7.50 First prove (say, by making use of the Planar Excluded Grid Theorem) that for every planar graph G of treewidth t, the treewidth of its dual graph G^* does not exceed ct for some constant c. In fact, it is possible to show that $\mathrm{tw}(G) \leq \mathrm{tw}(G^*) + 1$ (see [361]) but this is quite involved. Then deletion of edges in G corresponds to contraction of edges in G^*, and we can use Lemma 7.36.

7.48 First, perform a color coding step: every vertex $v \in V(G)$ is colored with a color $c(v) \in V(H)$, and we look for a subgraph isomorphic to H, whose vertices have appropriate colors. Second, compute a tree decomposition $(T, \{X_t\}_{t \in V(T)})$ of H of width $\mathrm{tw}(H)$. Third, perform the following dynamic programming: for every $t \in V(T)$ and mapping $\phi\colon X_t \to V(G)$ such that $c(\phi(h)) = h$ for every $h \in X_t$, compute whether there exists a (color-preserving) homomorphism $\hat{\phi}$ from $H[V_t]$ to G that extends ϕ. Here, color-preserving means that $c(\hat{\phi}(h)) = h$ for every h, and V_t denotes the union of all bags of descendants of t in the tree T.

7.53 Take a nice tree decomposition $(T, \{X_t\}_{t \in V(T)})$ of G of optimum width, and try to turn it to a branch decomposition (T', η) of at most the same rankwidth. You will need to get rid of leaf nodes and introduce nodes, so just collapse them. For forget nodes do the following: whenever a vertex w is forgotten at node t, add a pendant leaf t' to t and put $\eta(w) = t'$. After further small adjustments, you will obtain a valid branch decomposition. Prove that its rankwidth is at most the width of the original tree decomposition using the following observation: removal of each $e \in E(T')$ splits the vertices of $V(G)$ into sets X_e and Y_e, such that X_e is the set of vertices forgotten in some subtree of the original tree decomposition.

7.54 Intuitively, we sweep the forest H from left to right.

Let f be an embedding of G into a rooted forest H of depth d. Order the trees of H arbitrarily, and order the leaves of every tree of H in a natural way, obtaining a total order on all the leaves of H. For every leaf h of H, let Y_h be the set of all vertices on the path from h to the root of the tree containing h. Argue that sets $f^{-1}(Y_h)$ form a path decomposition of G of width at most $d - 1$.

7.55 For the left inequality $K_1 \cdot \mathrm{tw}(G) \leq \mathrm{mmw}(G)$, without loss of generality assume G has no isolated vertices, and let (T, η) be a branch decomposition of $V(G)$ that yields the optimum μ_G-width. We will construct a tree decomposition $(T, \{X_t\}_{t \in V(T)})$ of G (i.e., using the same tree T) of width similar to the μ_G-width of (T, η). Pick any edge $e \in E(T)$, and let G_e be the bipartite graph G_{X_e} where (X_e, Y_e) is the partition of $V(G)$ obtained by removing edge e from T. Let $W_e \subseteq V(G)$ be defined as follows: $v \in W_e$ if and only if for *every* maximum matching M in G_e, v is an endpoint of some edge in M. Show that W_e is a vertex cover of G_e, and moreover its cardinality is at most twice the cardinality of the maximum matching in G_e. For every $t \in V(T)$, let X_t be the union of sets W_e, where e iterates over edges e incident to t (there is always one or three such edges).

Now verify that $(T, \{X_t\}_{t \in V(T)})$ is a tree decomposition of G. Property (T1) is obvious. For property (T3), prove that the definition of the set W_e is monotone in the following sense: if v is a vertex of W_e that belongs to, say, the left side of G_e, then v stays in W_e even after moving any number of other vertices from the left side to the right side (this corresponds to changing the graph G_e when moving between two adjacent edges of T). For property (T2), focus on one edge $uv \in E(G)$ and look at the path in T between the leaves corresponding to the vertices u and v. Let $e_1, e_2, \ldots, e_r$ be the edges of this path, and assume they appear in this order. Argue, using (T3) and the fact that W_e is a vertex cover of G_e for each $e \in E(T)$, that there exists an index $1 \le i_0 < r$ such that $u \in W_{e_i}$ for every $i \le i_0$ and $v \in W_{e_i}$ for every $i > i_0$. Then u, v are both contained in the bag at the common endpoint of e_{i_0} and e_{i_0+1}.

For the right inequality $\mathrm{mmw}(G) \le K_2 \cdot \mathrm{tw}(G)$, use the same construction as in Exercise 7.53.

Bibliographic notes

The concept of the treewidth of a graph was rediscovered several times in different settings. Here we follow the definition of Robertson and Seymour [398, 399, 400]. Equivalent notions were introduced earlier by Halin [254] and by Bertelè and Brioschi [31]. See also the work of Arnborg and Proskurowski [20]. The proof that computing the treewidth of a graph is NP-hard is due to Arnborg, Corneil, and Proskurowski [18].

Treewidth-based dynamic-programming algorithm for Dominating Set with running time $4^t \cdot t^{\mathcal{O}(1)} \cdot n$ (i.e., Theorem 7.7) was first given by Alber, Bodlaender, Fernau, Kloks, and Niedermeier [8]. For origins of the idea of a nice tree decomposition we refer to the book of Kloks [297]. Techniques for reducing the polynomial factor depending on t in dynamic programming for different problems are discussed by Bodlaender, Bonsma, and Lokshtanov [47] (for Independent Set and MaxCut), and by Chimani, Mutzel, and Zey [88] (for variants of Steiner Tree).

Courcelle's theorem was proved by Courcelle in [98, 99]. The presented extension to optimization problems (Theorem 7.12) is due to Arnborg, Lagergren, and Seese [19]. A similar framework was independently proposed by Borie, Parker, and Tovey [58]. Another example of a powerful meta-theorem exploring relations between logic and combinatorial structures is the theorem of Frick and Grohe [218] that all first-order definable properties on planar graphs can be decided in linear time. See the surveys of Grohe and Kreutzer [237, 313] on algorithmic meta-theorems.

There are large numbers of different variants of search games studied in the literature, based on the discrete model we presented here, and modified by restricting or enhancing the abilities of the searchers or of the fugitive. The first variant of the search number of a graph was introduced by Kirousis and Papadimitriou in [292, 293], and is called *node search*. The variant of cops and robber games related to treewidth was defined by Seymour and Thomas [414]. There is an alternative game-theoretic approach to treewidth due to Dendris, Kirousis, and Thilikos [137], which exploits perfect elimination orderings of chordal graphs. The annotated bibliography [210] contains more references on graph searching and its relation to different width measures of graphs.

The book of Golumbic [231] discusses different algorithmic and structural properties of classes of perfect graphs, and in particular of chordal and interval graphs. Dirac's Lemma (part of Exercise 7.28) is due to Dirac [139]. Theorem 7.15 was proved by Seymour and Thomas in [414]; an accessible exposition can be found in the book of Diestel [138].

The definition of the vertex separation number (see Exercise 7.32) is due to Lengauer [318]. This notion was studied due to its connections with graph searching, gate-matrix layout, and other graph parameters [164, 292, 293, 362]. The proof that vertex separation number is equal to pathwidth (Exercise 7.32) is attributed to Kinnersley [291].

The first FPT approximation algorithm for computing treewidth is due to Robertson and Seymour [399]. Our description of the algorithm mostly follows Reed [396]. We remark that a different exposition of essentially the same algorithm can be found in the book of Kleinberg and Tardos [296].

A large variety of parameterized and approximation algorithms for treewidth can be found in the literature. These algorithms differ in approximation ratios, the exponential dependence of k, and the polynomial dependence of the input size. The best known fully polynomial-time approximation algorithm is due to Feige, Hajiaghayi, and Lee [172]; the approximation ratio is $\mathcal{O}(\sqrt{\log OPT})$ and the running time is polynomial in n. Wu, Austrin, Pitassi and Liu [438] showed that, unless the Small Set Expansion Conjecture fails, one cannot obtain a constant-factor approximation algorithm running in polynomial time.

The best known exact algorithm for treewidth, stated as Theorem 7.17, is due to Bodlaender [45]. The crucial component of this algorithm, namely a dynamic-programming procedure for computing treewidth exactly given a slightly too wide tree decomposition, was given by Bodlaender and Kloks [54]. There are several approximation algorithms, including the one given in this chapter, with better dependence on k, but worse dependence of n. One such example is the $(3+2/3)$-approximation algorithm of Amir [17] that runs in time $\mathcal{O}(2^{3.6982k}k^3 \cdot n^2)$. A 5-approximation algorithm running in time $2^{\mathcal{O}(k)} \cdot n$, i.e., single-exponential (as a function of treewidth) and linear (in the input size), was given by Bodlaender, Drange, Dregi, Fomin, Lokshtanov, and Pilipczuk [52].

The first proof of the Excluded Grid Minor Theorem (Theorem 7.22) is due to Robertson and Seymour [400]. There were several improvements on the exponential dependence on treewidth of the size of the grid minor: by Robertson, Seymour and Thomas [403], by Kawarabayashi and Kobayashi [286], and by Leaf and Seymour [317]. It was open for many years whether a polynomial relation could be established between the treewidth of a graph and the size of its largest grid minor. This question was resolved positively by Chekuri and Chuzhoy in [73], as stated in Theorem 7.22. The best known lower bound for function $g(t)$ in the Excluded Grid Minor Theorem is $\Omega(t^2 \log t)$, given by Robertson, Seymour, and Thomas [403]; this refines by a $\log t$ factor the lower bound given in Exercise 7.35.

The Planar Excluded Grid Theorem (Theorem 7.23) was first proved by Robertson, Seymour and Thomas [403]; their proof gave constant 6 in the relation between treewidth and the largest grid minor in a graph. The refined version with constant $\frac{9}{2}$ that we used in this chapter is due to Gu and Tamaki [239]. We refer you to the work of Gu and Tamaki for an overview of other approximation algorithms for treewidth and branchwidth on planar graphs. The contraction variant of the Planar Excluded Grid Theorem (Theorem 7.25) can be found in [193]; Figure 7.7 is borrowed from [193], by consent of the authors. Demaine and Hajiaghayi extended the Planar Excluded Grid Theorem to H-minor-free graphs in [134].

Bidimensionality was first introduced by Demaine, Fomin, Hajiaghayi, and Thilikos in [131]; see also surveys [133, 145]. As we mentioned, the framework can also be used to obtain kernelization algorithms as well as efficient polynomial-time approximation schemes (EPTASes). For kernelization algorithms based on bidimensional arguments we refer you to [200, 207, 208]. Applications for designing EPTASes are discussed in [132, 202]. The ideas of bidimensionality can also be extended to geometric graphs [205], and to directed planar graphs [144]. The subexponential algorithm for Partial Vertex Cover (Exercise 7.44) is from [203]. For Exercise 7.45, parameterized algorithms for Tree Spanner on planar, and more generally on apex-minor-free graphs, are given in [156]. The mentioned technique of Catalan structures was used by Arora, Grign, Karger, Klein, and Woloszyn in [22] and by Dorn, Penninx, Bodlaender, and Fomin in [147]. In Chapter 14 we discuss matching lower bounds on running times of algorithms obtained using bidimensionality.

The shifting technique is one of the most common techniques for designing PTASes on planar graphs. It dates back to the works of Baker [23] and Hochbaum and Maass [265]. Grohe showed in [236] how the shifting technique can be extended to H-minor-free graphs using tools from the theory of Graph Minors. Theorem 7.33 is due to Robertson and Seymour and is from [398]. There is also a very similar notion of k-outerplanarity, see the survey of Bodlaender [46]. Lemma 7.39 is due to Klein [294]; see also an extension of such a decomposition by Demaine, Hajiaghayi, and Kawarabayashi to H-minor-free graphs [135]. For the complexity of Subgraph Isomorphism under different parameterizations, we refer you to the work of Marx and Pilipczuk [356]. Eppstein [165] gave the first linear-time algorithm for Subgraph Isomorphism on planar graphs, with running time $k^{\mathcal{O}(k)}n$, where k is the size of the pattern graph. Dorn [143] improved this running time to $2^{\mathcal{O}(k)}n$. To the best of our knowledge, the existence of a polynomial-time algorithm for Minimum Bisection on planar graphs remains open. For general graphs the problem is NP-hard, but fixed-parameter tractable when parameterized by k [115].

The proof of the planarity criterion of Lemma 7.43 can be found in [421, Thm 3.8]. The irrelevant vertex technique was first used in the work of Robertson and Seymour on the Vertex Disjoint Paths problem [399, 402]. There are several constructive algorithms for Planar Vertex Deletion, see e.g. [285, 358]. The fastest known so far runs in time $k^{\mathcal{O}(k)}n$ and is due to Jansen, Lokshtanov, and Saurabh [277]. Other examples of applications of the irrelevant vertex technique can be found in [6, 116, 230, 238, 353].

Branchwidth of a graph was first introduced by Robertson and Seymour in [401]. This work also contains the proof of the linear relation between treewidth and branchwidth, i.e., equation (7.13), and introduces the concept already in the wider setting of hypergraphs. The polynomial-time algorithm for computing the branchwidth of a planar graph is due to Seymour and Thomas [415].

Oum and Seymour [378] gave an FPT 3-approximation algorithm for computing the f-width of any universe U, under the assumptions that f is submodular and provided to the algorithm as an oracle. This work also discusses applications to matroid branchwidth.

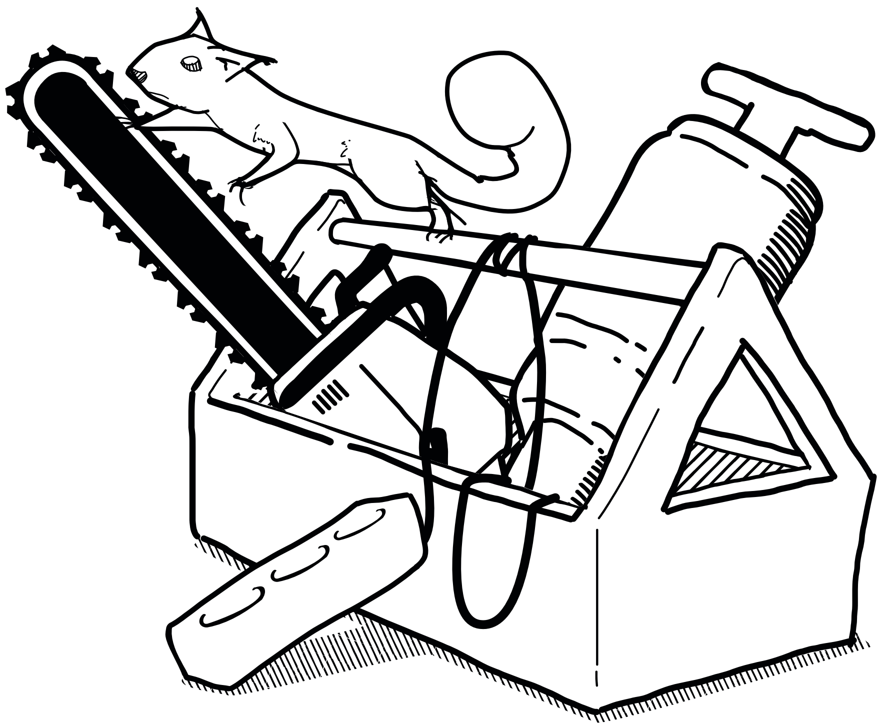
Cliquewidth was first introduced by Courcelle and Olariu [102]. Connections between $\mathbf{MSO}_1$ logic and graphs of bounded cliquewidth, resembling links between $\mathbf{MSO}_2$ and treewidth, were discovered by Courcelle, Makowsky, and Rotics [101]. Rankwidth was introduced by Oum and Seymour [378] as a tool to approximate cliquewidth. Since rankwidth and cliquewidth are bounded by functions of each other, the same tractability results for model checking $\mathbf{MSO}_1$ hold for rankwidth and for cliquewidth. From the general approximation algorithm of Oum and Seymour [378] it follows that rankwidth admits an FPT 3-approximation algorithm. In the specific setting of rankwidth, the running time and the approximation ratio were subsequently improved by Oum [377]. Hliněný and Oum [263] gave an exact FPT algorithm deciding if the rankwidth of a graph is at most k; this algorithm can also be used for matroid branchwidth.

Exercise 7.55 originates in the PhD thesis of Vatshelle [426]. Lower bounds on dynamic-programming algorithms for cliquewidth, and thus also for rankwidth, were investigated by Fomin, Golovach, Lokshtanov, and Saurabh [192, 191].

Further reading. The book of Diestel [138] contains a nice overview of the role the treewidth is playing in the Graph Minors project. We also recommend the following surveys of Reed [396, 395]. The proof of Courcelle's theorem and its applications are discussed in the book of Flum and Grohe [189]. The book of Courcelle and Engelfriet [100] is a thorough description of monadic second-order logic on graphs. For a general introduction to mathematical logic,

logic in computer science, and elements of model theory, we refer you to a wide range of available textbooks, e.g., [30, 266]. An extensive overview of different algorithmic and combinatorial aspects of treewidth as well as its relation to other graph parameters is given in the survey of Bodlaender [46] and in the book of Kloks [297]. An overview of different extensions of treewidth and rankwidth to directed graphs, matroids, hypergraphs, etc., is provided in the survey of Hliněný, Oum, Seese, and Gottlob [264].

Part II
Advanced algorithmic techniques



Chapter 8
Finding cuts and separators

The notion of important cuts and the related combinatorial bounds give a useful tool for showing the fixed-parameter tractability of problems where a graph has to be cut into certain parts. We discuss how this technique can be used to show that Edge Multiway Cut *and* Directed Feedback Vertex Set *are FPT. Random sampling of important separators is a recent extension of this method, making it applicable to a wider range of problems. We illustrate how this extension works on a clustering problem called* (p, q)-Partition*.*

Problems related to cutting a graph into parts satisfying certain properties or related to separating different parts of the graph from each other form a classical area of graph theory and combinatorial optimization, with strong motivation coming from applications. Many different versions of these problems have been studied in the literature: one may remove sets of edges or vertices in a directed or undirected graph; the goal can be separating two or more terminals from each other, cutting the graph into a certain number of parts, perhaps with constraints on the sizes of the parts, etc. Despite some notable exceptions (e.g., minimum $s - t$ cut, minimum multiway cut in planar graphs with fixed number of terminals), most of these problems are NP-hard. In this chapter, we investigate the fixed-parameter tractability of some of these problems parameterized by the size of the solution, that is, the size of the cut that we remove from the graph (one could parameterize these problems also by, for example, the number of terminals, while leaving the size of the solution unbounded, but such parameterizations are not the focus of this chapter). It turns out that small cuts have certain very interesting extremal combinatorial aspects that can be exploited in FPT algorithms. The notion of important cuts formalizes this extremal property and gives a very convenient tool for the treatment of these problems.

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_8

There is another class of problems that are closely related to cut problems: transversal problems, where we have to select edges/vertices to hit certain objects in the graph. For example, the Odd Cycle Transversal problem asks for a set of k vertices hitting every odd cycle in the graph. As we have seen in Section 4.4, iterative compression can be used to transform Odd Cycle Transversal into a series of minimum cut problems. The Directed Feedback Vertex Set problem asks for a set of k vertices hitting every directed cycle in a graph. The fixed-parameter tractability of Directed Feedback Vertex Set was a longstanding open problem; its solution was eventually reached by a combination of iterative compression and solving a directed cut problem (essentially) using important cuts. There are other examples where the study of a transversal problem reveals that it can be turned into an appropriate cut problem.

There is a particular difficulty that we face in the presentation of the basic results for the cut problems. There are many variants: we can delete vertices or edges, the graph can be directed or undirected, we may add weights (e.g., to forbid certain edges/vertices to be in the solution). While the basic theory is very similar for these variants, they require different notation and slightly different proofs. In this chapter, rather than presenting the most general form of these results, we mostly focus on the undirected edge versions, as they are the most intuitive and notationally cleanest. Then we go to the directed edge versions to treat certain problems specific to directed graphs (e.g., Directed Feedback Vertex Set) and finish the chapter by briefly commenting on the vertex variants of these results.

We assume basic familiarity with the concepts of cuts, flows, and algorithms for finding minimum cuts. Nevertheless, Section 8.1 reviews some of the finer points of minimum cuts in the form we need later. Section 8.2 introduces the notion of important cuts and presents the bound on their number and an algorithm to enumerate them. Then Section 8.3 uses important cuts to solve problems such as Edge Multiway Cut.

Very recently, a new way of using important cuts was introduced in one of the first FPT algorithms for Multicut. The “random sampling of important separators” technique allows us to restrict our attention to solutions that have a certain structural property, which can make the search for the solution much easier or can reduce it to some other problem. After the initial application for multicut problems, several other results used this technique as an ingredient of the solution. Unfortunately, most of these results are either on directed graphs (where the application of the technique is more delicate) or use several other nontrivial steps, making their presentation beyond the scope of this textbook. Section 8.4 presents the technique on a clustering problem where the task is to partition the graphs into classes of size at most p such that there are at most q edges leaving each class. This example is somewhat atypical (as it is not directly a transversal problem), but we can use it to demonstrate the random sampling technique in a self-contained way.

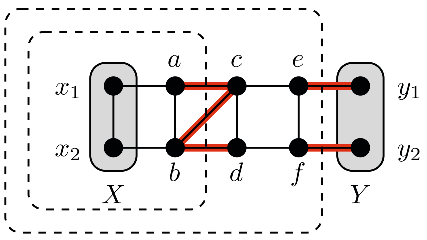


Fig. 8.1: The set $\Delta(\{x_1, x_2, a, b, c, d, e, f\}) = \{ey_1, fy_2\}$ is a minimum (X, Y)-cut (and hence minimal); the set $\Delta(\{x_1, x_2, a, b\}) = \{ac, bc, bd\}$ is a minimal (X, Y)-cut, but not minimum. The set $\Delta(\{x_1, x_2, a, c, d\}) = \{x_2b, ab, bc, bd, ce, df\}$ is an (X, Y)-cut, but not minimal

Section 8.5 generalizes the notion of important cuts to directed graphs and states the corresponding results without proofs. We point out that there are significant differences in the directed setting: the way important cuts were used to solve EDGE MULTIWAY CUT in undirected graphs does not generalize to directed graphs. Nevertheless, we can solve a certain directed cut problem called SKEW EDGE MULTICUT using important cuts. In Section 8.6, a combination of the algorithm for SKEW EDGE MULTICUT and iterative compression is used to show the fixed-parameter tractability of DIRECTED FEEDBACK VERTEX SET and DIRECTED FEEDBACK ARC SET.

Section 8.7 discusses the version of the results for vertex-deletion problems. We define the notions required for handling vertex-deletion problems and state the most important results. We do not give any proofs in this section, as they are mostly very similar to their edge-removal counterparts.

8.1 Minimum cuts

An (X, Y)-*cut* is a set S of edges that separates X and Y from each other, that is, $G \setminus S$ has no $X-Y$ path. We need to distinguish between two different notions of minimality. An (X, Y)-cut S is a *minimum* (X, Y)-cut if there is no (X, Y)-cut S' with $|S'| < |S|$. An (X, Y)-cut is *(inclusion-wise) minimal* if there is no (X, Y)-cut S' with $S' \subset S$ (see Fig. 8.1). Observe that every minimum cut is minimal, but not necessarily the other way around. We allow parallel edges in this section: this is essentially the same as having arbitrary integer weights on the edges.

It will be convenient to look at minimal (X, Y)-cuts from a different perspective, viewing them as edges on the boundary of a certain set of vertices. If G is an undirected graph and $R \subseteq V(G)$ is a set of vertices, then we denote by $\Delta_G(R)$ the set of edges with exactly one endpoint in R, and we denote

$d_G(R) = |\Delta_G(R)|$ (we omit the subscript G if it is clear from the context). Let S be a minimal (X, Y)-cut in G and let R be the set of vertices reachable from X in $G \setminus S$; clearly, we have $X \subseteq R \subseteq V(G) \setminus Y$. Then it is easy to see that S is precisely $\Delta(R)$. Indeed, every such edge has to be in S (otherwise a vertex of $V(G) \setminus R$ would be reachable from X) and S cannot have an edge with both endpoints in R or both endpoints in $V(G) \setminus R$, as omitting any such edge would not change the fact that the set is an (X, Y)-cut, contradicting minimality.

Proposition 8.1. *If S is a minimal (X, Y)-cut in G, then $S = \Delta_G(R)$, where R is the set of vertices reachable from X in $G \setminus S$.*

Therefore, we may always characterize a minimal (X, Y)-cut S as $\Delta(R)$ for some set $X \subseteq R \subseteq V(G) \setminus Y$. Let us also note that $\Delta(R)$ is an (X, Y)-cut for every such set R with $X \subseteq R \subseteq V(G) \setminus Y$, but not necessarily a minimal (X, Y)-cut (see Fig. 8.1).

The well-known maximum flow and minimum cut duality implies that the size of the minimum (X, Y)-cut is the same as the maximum number of pairwise edge-disjoint $X - Y$ paths. Classical maximum flow algorithms can be used to find a minimum cut and a corresponding collection of edge-disjoint $X - Y$ paths of the same size. We do not review the history of these algorithms and their running times here, as in our setting we usually want to find a cut of size at most k, where k is assumed to be a small constant. Therefore, the following fact is sufficient for our purposes: each round of the algorithm of Ford and Fulkerson takes linear time, and k rounds are sufficient to decide if there is an (X, Y)-cut of size at most k. We state this fact in the following form.

Theorem 8.2. *Given a graph G with n vertices and m edges, disjoint sets $X, Y \subseteq V(G)$, and an integer k, there is an $\mathcal{O}(k(n + m))$-time algorithm that either*

- *correctly concludes that there is no (X, Y)-cut of size at most k, or*
- *returns a minimum (X, Y)-cut $\Delta(R)$ and a collection of $|\Delta(R)|$ pairwise edge-disjoint $X - Y$ paths.*

Submodular set functions play an essential role in many areas of combinatorial optimization, and they are especially important for problems involving cuts and connectivity. Let $f \colon 2^{V(G)} \to \mathbb{R}$ be a set function assigning a real number to each subset of vertices of a graph G. We say that f is *submodular* if it satisfies the following inequality for every $A, B \subseteq V(G)$:

$$f(A) + f(B) \geq f(A \cap B) + f(A \cup B). \tag{8.1}$$

We will use the well-known fact that the function $d_G(X) = |\Delta_G(X)|$ is submodular.

Theorem 8.3. *The function d_G is submodular for every undirected graph G.*

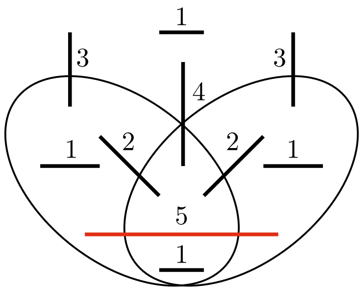


Fig. 8.2: The different types of edges in the proof of Theorem 8.3

Proof. Let us classify each edge e according to the location of its endpoints (see Fig. 8.2) and calculate its contribution to the two sides of (8.1):

1. If both endpoints of e are in $A \cap B$, in $A \setminus B$, in $B \setminus A$, or in $V(G) \setminus (A \cup B)$, then e contributes 0 to both sides.
2. If one endpoint of e is in $A \cap B$, and the other is either in $A \setminus B$ or in $B \setminus A$, then e contributes 1 to both sides.
3. If one endpoint of e is in $V(G) \setminus (A \cup B)$, and the other is either in $A \setminus B$ or in $B \setminus A$, then e contributes 1 to both sides.
4. If e is between $A \cap B$ and $V(G) \setminus (A \cup B)$, then e contributes 2 to both sides.
5. If e is between $A \setminus B$ and $B \setminus A$, then e contributes 2 to the left-hand side and 0 to the right-hand side.

As the contribution of each edge e to the left-hand side is at least as much as its contribution to the right-hand side, inequality (8.1) follows. □

A reason why submodularity of d_G is particularly relevant to cut problems is that if $\Delta(A)$ and $\Delta(B)$ are both (X, Y)-cuts, then $\Delta(A \cap B)$ and $\Delta(A \cup B)$ are both (X, Y)-cuts: indeed, $A \cap B$ and $A \cup B$ both contain X and are disjoint from Y. Therefore, we can interpret Theorem 8.3 as saying that if we have two (X, Y)-cuts $\Delta(A)$, $\Delta(B)$ of a certain size, then two new (X, Y)-cuts $\Delta(A \cap B)$, $\Delta(A \cup B)$ can be created and there is a bound on their total size.

The minimum (X, Y)-cut is not necessarily unique; in fact, a graph can have a large number of minimum (X, Y)-cuts. Suppose, for example, that $X = \{x\}$, $Y = \{y\}$, and k paths connect x and y, each of length n. Then selecting one edge from each path gives a minimum (X, Y)-cut, hence there are n^k different minimum (X, Y)-cuts. However, as we show below, there is a unique minimum (X, Y)-cut $\Delta(R_{\min})$ that is closest to X and a unique minimum (X, Y)-cut $\Delta(R_{\max})$ closest to Y, in the sense that the sets $R_{\min}$ and $R_{\max}$ are minimum and maximum possible, respectively (see Fig. 8.3(a)). The proof of this statement follows from an easy application of the submodularity of d_G.

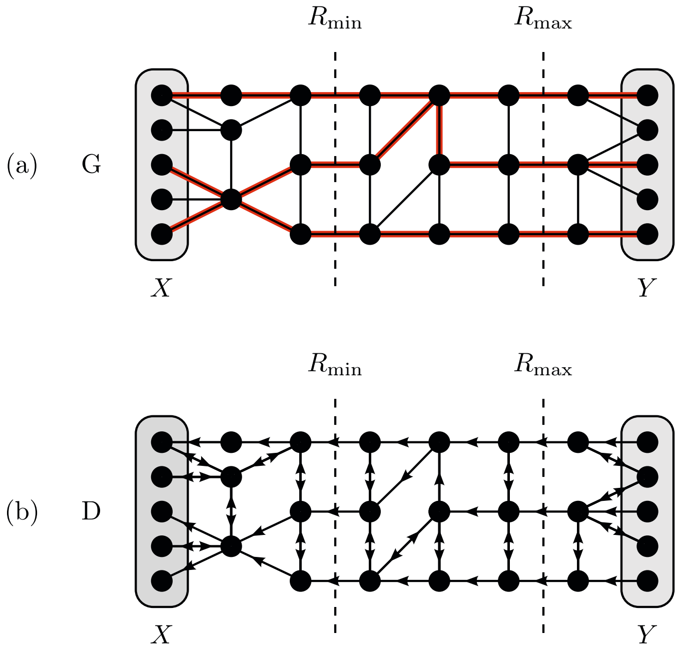


Fig. 8.3: (a) A graph G with three edge-disjoint (X, Y)-paths and the (X, Y)-cuts $R_{\min}$ and $R_{\max}$ of Theorem 8.4. (b) The corresponding residual directed graph D defined in the proof of Theorem 8.5

Theorem 8.4. *Let G be a graph and $X, Y \subseteq V(G)$ two disjoint sets of vertices. There are two minimum (X, Y)-cuts $\Delta(R_{\min})$ and $\Delta(R_{\max})$ such that if $\Delta(R)$ is a minimum (X, Y)-cut, then $R_{\min} \subseteq R \subseteq R_{\max}$.*

Proof. Consider the collection $\mathcal{R}$ of every set $R \subseteq V(G)$ for which $\Delta(R)$ is a minimum (X, Y)-cut. We show that there is a unique inclusion-wise minimal set $R_{\min}$ and a unique inclusion-wise maximal set $R_{\max}$ in $\mathcal{R}$. Suppose for contradiction that $\Delta(R_1)$ and $\Delta(R_2)$ are minimum cuts for two inclusion-wise minimal sets $R_1 \neq R_2$ of $\mathcal{R}$. By (8.1), we have

$$d_G(R_1) + d_G(R_2) \geq d_G(R_1 \cap R_2) + d_G(R_1 \cup R_2).$$

If λ is the minimum (X, Y)-cut size, then the left-hand side is exactly 2λ, hence the right-hand side is at most 2λ. Observe that $\Delta(R_1 \cap R_2)$ and $\Delta(R_1 \cup R_2)$ are both (X, Y)-cuts. Taking into account that λ is the minimum (X, Y)-cut size, the right-hand side is also exactly 2λ, with both terms being exactly λ. That is, $\Delta(R_1 \cap R_2)$ is a minimum (X, Y)-cut. Now $R_1 \neq R_2$ implies that $R_1 \cap R_2 \subset R_1, R_2$, contradicting the assumption that both R_1 and R_2 are inclusion-wise minimal in $\mathcal{R}$.

The same argument gives a contradiction if $R_1 \neq R_2$ are inclusion-wise maximal sets of the collection: then we observe that $\Delta(R_1 \cup R_2)$ is also a minimum (X,Y)-cut. $\square$

While the proof of Theorem 8.4 is not algorithmic, one can find, for example, $R_{\min}$ by repeatedly adding vertices to Y as long as this does not increase the minimum cut size (Exercise 8.5). However, there is also a linear-time algorithm for finding these sets.

Theorem 8.5. *Let G be a graph with n vertices and m edges, and $X, Y \subseteq V(G)$ be two disjoint sets of vertices. Let k be the size of the minimum (X,Y)-cut. The sets $R_{\min}$ and $R_{\max}$ of Theorem 8.4 can be found in time $\mathcal{O}(k(n+m))$.*

Proof. Let us invoke the algorithm of Theorem 8.2 and let $P_1, \ldots, P_k$ be the pairwise edge-disjoint $X-Y$ paths returned by the algorithm. We build the *residual* directed graph D as follows. If edge xy of G is not used by any of the paths P_i, then we introduce both (x,y) and (y,x) into D. If edge xy of G is used by some P_i in such a way that x is closer to X on path P_i, then we introduce the directed edge[1] (y,x) into D (see Fig. 8.3(b)).

We show that $R_{\min}$ is the set of vertices reachable from X in the residual graph D and $R_{\max}$ is the set of vertices from which Y is *not* reachable in D.

Let $\Delta_G(R)$ be a minimum (X,Y)-cut of G. As $\Delta_G(R)$ is an (X,Y)-cut of size k, each of the k paths $P_1, \ldots, P_k$ uses exactly one edge of $\Delta_G(R)$. This means that after P_i leaves R, it never returns to R. Therefore, if P_i uses an edge $ab \in \Delta_G(R)$ with $a \in R$ and $b \notin R$, then a is closer to X on P_i. This implies that (a,b) is not an edge of D. As this is true for every edge of the cut $\Delta_G(R)$, we get that $V(G) \setminus R$ is not reachable from X in D; in particular, Y is not reachable.

Let $R_{\min}$ be the set of vertices reachable from X in D. We have shown in the previous paragraph that Y is not reachable from X in D (that is, $X \subseteq R_{\min} \subseteq V(G) \setminus Y$), hence $\Delta_G(R_{\min})$ is an (X,Y)-cut of G. If we can show that this cut is a minimum (X,Y)-cut, then we are done: we have shown that if $\Delta_G(R)$ is a minimum (X,Y)-cut, then $V(G) \setminus R$ is not reachable from X in D, implying that $V(G) \setminus R \subseteq V(G) \setminus R_{\min}$ and hence $R_{\min} \subseteq R$.

Every path P_i uses at least one edge of the (X,Y)-cut $\Delta_G(R_{\min})$. Moreover, P_i cannot use more than one edge of the cut: if P_i leaves $R_{\min}$ and later returns to $R_{\min}$ on an edge ab with $a \notin R_{\min}$, $b \in R_{\min}$ and a closer to X on P_i, then (b,a) is an edge of D and it follows that a is also reachable from

[1] The reader might find introducing the edge (x,y) instead of (y,x) more natural and indeed the rest of the proof would work just as well after appropriate changes. However, here we follow the standard definition of residual graphs used in network flow algorithms.

X in D, contradicting $a \notin R_{\min}$. Therefore, $\Delta_G(R_{\min})$ can have at most k edges, implying that it is a minimum (X,Y)-cut. Therefore, $R_{\min}$ satisfies the requirements.

A symmetrical argument shows that the set $R_{\max}$ containing all vertices from which Y is not reachable in D satisfies the requirements. $\square$

8.2 Important cuts

Most of the results of this chapter are based on the following definition.

Definition 8.6 (Important cut). Let G be an undirected graph and let $X, Y \subseteq V(G)$ be two disjoint sets of vertices. Let $S \subseteq E(G)$ be an (X,Y)-cut and let R be the set of vertices reachable from X in $G \setminus S$. We say that S is an *important* (X,Y)*-cut* if it is inclusion-wise minimal and there is no (X,Y)-cut S' with $|S'| \leq |S|$ such that $R \subset R'$, where R' is the set of vertices reachable from X in $G \setminus S'$.

Note that the definition is not symmetrical: an important (X,Y)-cut is not necessarily an important (Y,X)-cut.

An intuitive interpretation of Definition 8.6 is that we want to minimize the size of the (X,Y)-cut and at the same time we want to maximize the set of vertices that remain reachable from X after removing the cut. The important (X,Y)-cuts are the (X,Y)-cuts that are "Pareto efficient" with respect to these two objectives: increasing the set of vertices reachable from X requires strictly increasing the size of the cut.

Let us point out that we do not want the number of vertices reachable from X to be maximal, we just want that this set of vertices be *inclusion-wise maximal* (i.e., we have $R \subset R'$ and *not* $|R| < |R'|$ in the definition).

The following proposition formalizes an immediate consequence of the definition. This is the property of important (X,Y)-cuts that we use in the algorithms.

Proposition 8.7. *Let G be an undirected graph and $X, Y \subseteq V(G)$ two disjoint sets of vertices. Let S be an (X,Y)-cut and let R be the set of vertices reachable from X in $G \setminus S$. Then there is an important (X,Y)-cut $S' = \Delta(R')$ (possibly, $S' = S$) such that $|S'| \leq |S|$ and $R \subseteq R'$.*

Proof. By definition, every (X,Y)-cut has a subset that is an inclusion-wise minimal (X,Y)-cut. Let $S^* \subseteq S$ be a minimal (X,Y)-cut and let $R^* \supseteq R$ be the set of vertices reachable from X in $G \setminus S^*$. If S^* is an important (X,Y)-cut, then we are done. Otherwise, there is an (X,Y)-cut $S' = \Delta(R')$ for some

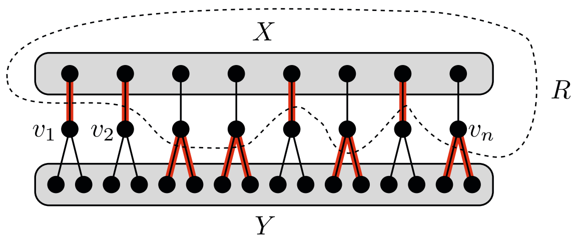


Fig. 8.4: A graph where every minimal (X, Y)-cut is an important (X, Y)-cut

$R' \supset R^* \supseteq R$ and $|S'| \leq |S^*| \leq |S|$. If S' is an important (X, Y)-cut, then we are done. Otherwise, we can repeat the argument: each time we strictly increase the set of vertices reachable from X and the size of the cut does not increase. Eventually, the process has to stop and we obtain an important (X, Y)-cut satisfying the required properties. □

To check whether a given (X, Y)-cut $\Delta(R)$ is important, one needs to check whether there is another (X, Y)-cut of the same size "after" R, that is, whether there is an $(R \cup v, Y)$-cut of size not larger for some $v \in V(G) \setminus R$. This can be done efficiently using Theorem 8.5.

Proposition 8.8. *Given a graph G with n vertices and m edges, two disjoint sets $X, Y \subseteq V(G)$, and an (X, Y)-cut $\Delta(R)$ of size k, it can be tested in time $\mathcal{O}(k(n + m))$ whether $\Delta(R)$ is an important cut.*

Proof. Observe that $\Delta(R)$ is an important $X - Y$ cut if and only if it is the unique minimum (R, Y) cut. Therefore, if we compute the minimum (R, Y)-cut $\Delta(R_{\max})$ using the algorithm of Theorem 8.5, then $\Delta(R)$ is an important (X, Y)-cut if and only if $R = R_{\max}$. □

Theorem 8.4 shows that $\Delta(R_{\max})$ is the unique minimum (X, Y)-cut that is an important (X, Y)-cut: we have $R_{\max} \supset R$ for every other minimum (X, Y)-cut $\Delta(R)$. However, for sizes larger than the minimum cut size, there can be a large number of incomparable (X, Y)-cuts of the same size. Consider the example in Fig. 8.4. Any (X, Y)-cut $\Delta(R)$ is an important (X, Y)-cut: the only way to extend the set R is to move some vertex $v_i \in V(G) \setminus (R \cup Y)$ into the set R, but then the cut size increases by 1. Therefore, the graph has exactly $\binom{n}{x}$ important (X, Y)-cuts of size $n + x$.

The main result of the section is a bound on the number of important cuts of size at most k that depends only on k and is independent of the size of the graph. Moreover, we show that the important separators can be efficiently enumerated. We need first the following simple observations, whose proofs are given as Exercise 8.7.

Proposition 8.9. *Let G be a graph, $X, Y \subseteq V(G)$ be two disjoint sets of vertices, and $S = \Delta(R)$ be an important (X, Y)-cut.*

1. *For every $e \in S$, the set $S \setminus \{e\}$ is an important (X,Y)-cut in $G \setminus e$.*
2. *If S is an (X',Y)-cut for some $X' \supset X$, then S is an important (X',Y)-cut.*

The first key observation in bounding the number of important (X,Y)-cuts is that every important (X,Y)-cut is after the set $R_{\max}$ of Theorem 8.4.

Lemma 8.10. *If $\Delta(R)$ is an important (X,Y)-cut, then $R_{\max} \subseteq R$.*

Proof. Let us apply the submodular inequality (8.1) on $R_{\max}$ and R:

$$d_G(R_{\max}) + d_G(R) \geq d_G(R_{\max} \cap R) + d_G(R_{\max} \cup R).$$

Let λ be the minimum (X,Y)-cut size. The first term $d_G(R_{\max})$ on the left-hand side is exactly λ (as $\Delta_G(R_{\max})$ is a minimum (X,Y)-cut). Furthermore, $\Delta_G(R_{\max} \cap R)$ is an (X,Y)-cut, hence we have that the first term on the right-hand side is at least λ. It follows then that the second term of the left-hand side is at least the second term on the right-hand side, that is, $d_G(R) \geq d_G(R \cup R_{\max})$. If $R_{\max} \not\subseteq R$, then $R_{\max} \cup R$ is a proper superset of R. However, then the (X,Y)-cut $\Delta_G(R_{\max} \cup R)$ of size at most $d_G(R)$ contradicts the assumption that $\Delta_G(R)$ is an important (X,Y)-cut. Thus we have proved that $R_{\max} \subseteq R$. □

We are now ready to present the bound on the number of important cuts.

Theorem 8.11. *Let $X, Y \subseteq V(G)$ be two disjoint sets of vertices in graph G and let $k \geq 0$ be an integer. There are at most 4^k important (X,Y)-cuts of size at most k.*

Proof. We prove that there are at most $2^{2k-\lambda}$ important (X,Y)-cuts of size at most k, where λ is the size of the smallest (X,Y)-cut. Clearly, this implies the upper bound 4^k claimed in the theorem. The statement is proved by induction on $2k - \lambda$. If $\lambda > k$, then there is no (X,Y)-cut of size k, and therefore the statement holds if $2k - \lambda < 0$. Also, if $\lambda = 0$ and $k \geq 0$, then there is a unique important (X,Y)-cut of size at most k: the empty set.

The proof is by branching on an edge xy leaving $R_{\max}$: an important (X,Y)-cut either contains xy or not. In both cases, we can recurse on an instance where the measure $2k - \lambda$ is strictly smaller.

Let $\Delta(R)$ be an important (X,Y)-cut and let $\Delta(R_{\max})$ be the minimum (X,Y)-cut defined by Theorem 8.4. By Lemma 8.10, we have $R_{\max} \subseteq R$. As we have assumed $\lambda > 0$, there is at least one edge xy with $x \in R_{\max} \subseteq R$ and $y \notin R_{\max}$. Then vertex y is either in R or not. If $y \notin R$, then xy is an edge of the cut $\Delta(R)$ and then $S \setminus \{xy\}$ is an important (X,Y)-cut in $G' = G \setminus xy$ of size at most $k' := k - 1$ (Prop. 8.9(1)). Removing an edge can decrease the size of the minimum (X, Y)-cut size by at most 1, hence the size λ' of the minimum (X, Y)-cut in G' is at least $\lambda - 1$. Therefore, $2k' - \lambda' < 2k - \lambda$ and the induction hypothesis implies that there are at most $2^{2k'-\lambda'} \leq 2^{2k-\lambda-1}$ important (X, Y)-cuts of size k' in G', and hence at most that many important (X, Y)-cuts of size k in G that contain the edge xy.

Let us count now the important (X, Y)-cuts not containing the edge xy. As $R_{\max} \subseteq R$, the fact that xy is not in the cut implies that even $R_{\max} \cup \{y\} \subseteq R$ is true. Let $X' = R_{\max} \cup \{y\}$; it follows that $\Delta(R)$ is an (X', Y)-cut and in fact an important (X', Y)-cut by Prop. 8.9(2). There is no (X', Y)-cut $\Delta(R')$ of size λ: such a cut would be an (X, Y)-cut with $R_{\max} \subset R_{\max} \cup \{y\} \subseteq R'$, contradicting the definition of $R_{\max}$. Thus the minimum size λ' of an (X', Y)-cut is greater than λ. It follows by the induction assumption that the number of important (X', Y)-cuts of size at most k is at most $2^{2k-\lambda'} \leq 2^{2k-\lambda-1}$, which is a bound on the number of important (X, Y)-cuts of size k in G that do not contain xy.

Adding the bounds in the two cases, we get the required bound $2^{2k-\lambda}$. $\square$

The main use of the bound in Theorem 8.11 is for algorithms based on branching: as the number of important (X, Y)-cuts is bounded by a function of k, we can afford to branch on selecting one of them (see for example Theorem 8.19). The following refined bound is very helpful for a tighter bound on the size of the search tree for such algorithms. Its proof is essentially the same as the proof of Theorem 8.11 with an appropriate induction statement (see Exercise 8.9).

Lemma 8.12. *Let G be an undirected graph and let $X, Y \subseteq V(G)$ be two disjoint sets of vertices. If $\mathcal{S}$ is the set of all important (X, Y)-cuts, then $\sum_{S \in \mathcal{S}} 4^{-|S|} \leq 1$.*

Observe that Lemma 8.12 implies Theorem 8.11: the contribution of each important (X, Y)-cut to the sum is at least 4^{-k}, hence the fact that the sum is at most 1 implies that there can be at most 4^k such cuts.

So far, Theorem 8.11 and Lemma 8.12 are purely combinatorial statements bounding the number of important cuts. However, an algorithm for enumerating all of the important cuts follows from the proof of Theorem 8.11. First, we can compute $R_{\max}$ using the algorithm of Theorem 8.5. Pick an arbitrary edge $xy \in \Delta(R_{\max})$. Then we branch on whether edge xy is in the important cut or not, and recursively find all possible important cuts for both cases. The search tree of the algorithm has size at most 4^k and the work to be done in each node is $\mathcal{O}(k(n+m))$. Therefore, the total running time of the branching algorithm is $\mathcal{O}(4^k \cdot k(n+m))$ and it returns at most 4^k cuts.

However, there is an unfortunate technical detail with the algorithm described above. The algorithm enumerates a superset of all important cuts: by our analysis, every important cut is found, but there is no guarantee that all the constructed cuts are important (see Exercise 8.8). We present three different ways of handling this issue. The most obvious solution is to perform a filtering phase where we use Proposition 8.8 to check for each returned cut whether it is important. This filtering phase takes time $\mathcal{O}(4^k \cdot k(n+m))$.

Theorem 8.13. *Let $X, Y \subseteq V(G)$ be two disjoint sets of vertices in a graph G with n vertices and m edges, and let $k \geq 0$ be an integer. The set of all important (X,Y)-cuts of size at most k can be enumerated in time $\mathcal{O}(4^k \cdot k \cdot (n+m))$.*

When using the refined bound of Lemma 8.12 to bound the running time of a branching algorithm, we need that the time spent in each node of the search tree be proportional to the number of directions into which we branch. This is unfortunately not true if we use Theorem 8.13 to enumerate all the important cuts: the 4^k factor appears in the time bound, even if the number of actual important cuts returned is much less than 4^k. To obtain an algorithm that does not suffer from this problem, let $\mathcal{S}'_k$ be the set of all (X,Y)-cuts found before the filtering. Observe that if the algorithm considers only branches where $k \geq \lambda$ (otherwise there is no (X,Y)-cut of size at most k), then every leaf node has $\lambda = 0$ and hence produces a cut, which means that the recursion tree has $|\mathcal{S}'_k|$ leaves. Furthermore, the height of the recursion tree is at most k, thus the recursion tree has $\mathcal{O}(k|\mathcal{S}'_k|)$ nodes. The work to be done in each node is $\mathcal{O}(k(n+m))$. Therefore, we obtain the following bound on the running time.

Theorem 8.14. *Let $X, Y \subseteq V(G)$ be two disjoint sets of vertices in graph G with n vertices and m edges, let $k \geq 0$ be an integer. One can enumerate a superset $\mathcal{S}'_k$ of every important (X,Y)-cut of size at most k in time $\mathcal{O}(|\mathcal{S}'_k| \cdot k^2 \cdot (n+m))$. Moreover, the set $\mathcal{S}'_k$ satisfies $\sum_{S \in \mathcal{S}'_k} 4^{-|S|} \leq 1$.*

Theorem 8.14 gives a more refined bound on the running time than Theorem 8.13. It may appear somewhat confusing, however, that the algorithm of Theorem 8.14 returns cuts that are not (X,Y)-cuts. What Theorem 8.14 really says is that we have good bounds both on the running time and on the number of cuts returned *even if* we consider these cuts and do not restrict the output to the important (X,Y)-cuts. One may prefer a cleaner statement about the existence of an algorithm that returns only important (X,Y)-cuts and the running time depends linearly on the size of the output. It is possible to obtain such an algorithm, but it requires an additional combinatorial argument. We need to ensure that the size of the recursion tree is $\mathcal{O}(k|\mathcal{S}_k|)$, where $\mathcal{S}_k$ is the set of all important (X,Y)-cuts of size at most k. We achieve this by checking before each recursion step whether at least one important (X,Y)-cut of size at most k will be returned; if not, then there is no need to perform the recursion. This way, we ensure that each leaf of the recursion tree returns an *important* (X,Y)-cut, hence the size of the recursion tree can be bounded by $\mathcal{O}(k|\mathcal{S}_k|)$. The check before the recursion is made possible by the following lemma.

Lemma 8.15. *Let $X, Y \subseteq V(G)$ be two disjoint sets of vertices in a graph G with n vertices and m edges, let $k \geq 0$ be an integer, and let $Z \subseteq \Delta(X)$*

be a set of edges. It can be checked in time $\mathcal{O}(k(n+m))$ whether there is an important (X,Y)-cut S of size at most k with $Z \subseteq S$.

Proof. Let $G' = G \setminus Z$, let $C = \Delta_{G'}(R'_{\max})$ be the minimum (X,Y)-cut of G' given by Theorem 8.4. We claim that G has an important (X,Y)-cut of size at most k containing Z if and only $Z \cup C$ is such a cut. Suppose that there is a $D \subseteq E(G')$ such that $Z \cup D$ is an important (X,Y)-cut of size at most k. As $Z \cup D$ is a minimal (X,Y)-cut, we can write it as $\Delta_G(R_D)$ for some set $R_D \supseteq X$. Each edge of $Z \subseteq \Delta_G(X)$ has an endpoint not in X; let P contain all these endpoints. Observe that $Z \subseteq \Delta_G(R_D)$ implies that R_D is disjoint from P.

Clearly, $Z \cup C$ is an (X,Y)-cut in G, thus if it is not important, then Proposition 8.7 implies that there is an important (X,Y)-cut $S = \Delta(R_S)$ with $|S| \le |Z \cup C|$ and $R'_{\max} \subseteq R_S$. If $Z \subseteq S$, then $S \setminus Z = \Delta_{G'}(R_S)$ is an (X,Y)-cut in G' with $|S \setminus Z| \le |C|$. By the definition of $R'_{\max}$, this is only possible if $|S \setminus Z| = |C|$ and $R_S = R'_{\max}$, which would imply $S = Z \cup C$. Therefore, at least one edge of Z is not in S, which implies that R_S contains at least one vertex of P.

Consider now the submodular inequality on R_S and R_D.

$$d_G(R_S) + d_G(R_D) \ge d_G(R_S \cap R_D) + d_G(R_S \cup R_D)$$

Observe that $\Delta_G(R_S \cup R_D)$ is an (X,Y)-cut and $R_S \cup R_D$ is a proper superset of R_D (as R_D is disjoint from P and R_S intersects P). Thus the fact that $Z \cup D$ is an important (X,Y)-cut implies $d_G(R_S \cup R_D) > d_G(R_D)$. It follows that $d_G(R_S \cap R_D) < d_G(R_S) \le |Z \cup C|$ has to hold. Note that $Z \subseteq \Delta_G(R_S \cap R_D)$ as $X \subseteq R_S \cap R_D$ and R_D (and hence $R_S \cap R_D$) is disjoint from P. Therefore, $\Delta_G(R_S \cap R_D)$ can be written as $Z \cup C^*$ for some $C^* \subseteq E(G')$ disjoint from Z. Now $|Z \cup C^*| = d_G(R_S \cap R_D) < |Z \cup C|$ implies $|C^*| < |C|$. This contradicts the assumption that C is a minimum (X,Y)-cut in G'. □

Equipped with Lemma 8.15, one can modify the branching algorithm in a way that it enumerates only important (X,Y)-cuts. (Exercise 8.10 asks for working out the details of this algorithm.)

Theorem 8.16. *Let $X, Y \subseteq V(G)$ be two disjoint sets of vertices in a graph G with n vertices and m edges, and let $k \ge 0$ be an integer. The set $\mathcal{S}_k$ of all important (X,Y)-cuts of size at most k can be enumerated in time $\mathcal{O}(|\mathcal{S}_k| \cdot k^2 \cdot (n+m))$.*

The reader might wonder how tight the bound 4^k in Theorem 8.11 is. The example in Fig. 8.4 already showed that the number of important cuts of size at most k can be exponential in k. We can show that the bound 4^k is tight up to polynomial factors; in particular, the base of the exponent has to be 4. Consider a rooted complete binary tree T with $n > k$ levels; let X contain only the root and Y contain all the leaves (see Fig. 8.5). Then every rooted full binary subtree T' with k leaves gives rise to an important (X,Y)-cut of

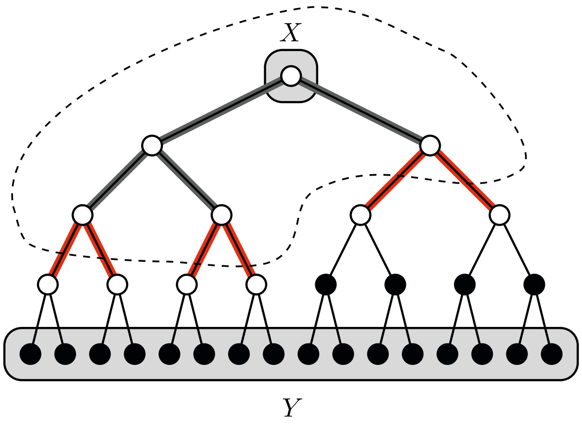


Fig. 8.5: A graph with $\Theta(4^k/k^{3/2})$ important (X, Y)-cuts of size k: every full binary subtree with k leaves gives rise to an important (X, Y)-cut of size k. The figure shows a subtree with six leaves and the corresponding (X, Y)-cut of size 6 (red edges)

size k. Indeed, the cut $\Delta(R)$ obtained by removing the edges incident to the leaves of T' is an important (X, Y)-cut, as moving more vertices to R would clearly increase the cut size. It is well known that the number of full subtrees with exactly k leaves of a rooted complete binary tree is precisely the Catalan number $C_k = (1/k)\binom{2k-2}{k-1} = \Theta(4^k/k^{3/2})$.

As an application of the bound on the number of important cuts, we can prove the following surprisingly simple, but still nontrivial combinatorial result: only a bounded number of edges incident to Y are relevant to (X, Y)-cuts of size at most k.

Lemma 8.17. *Let G be an undirected graph and let $X, Y \subseteq V(G)$ be two disjoint sets of vertices. The union of all minimal (X, Y)-cuts of size at most k contains at most $k \cdot 4^k$ edges incident to Y.*

Proof. Let F contain an edge e if it is incident to a vertex of Y and it appears in an (X, Y)-important cut of size at most k. By Theorem 8.11, we have $|F| \leq 4^k \cdot k$. To finish the proof of the lemma, we show that if an edge e incident to Y appears in some minimal (X, Y)-cut $\Delta(R)$ of size at most k, then $e \in F$. Proposition 8.7 implies that there is an important (X, Y)-cut $\Delta(R')$ of size at most k with $R \subseteq R'$. Edge e has an endpoint $x \in R$ and an endpoint $y \notin R$. As $R \cap Y = \emptyset$, the endpoint y of e has to be in Y. Now we have $x \in R \subseteq R'$ and $y \in Y \subseteq V(G) \setminus R'$, hence e is an edge of $\Delta(R')$ as well, implying $e \in F$. $\square$

8.3 EDGE MULTIWAY CUT

Let G be a graph and $T \subseteq V(G)$ be a set of terminals. An *edge multiway cut* is a set S of edges such that every component of $G \setminus S$ contains at most one vertex of T. Given a graph G, terminals T, and an integer k, the EDGE MULTIWAY CUT problem asks if a multiway cut of size at most k exists. If $|T| = 2$, that is, $T = \{t_1, t_2\}$, then EDGE MULTIWAY CUT is the problem of finding a (t_1, t_2)-cut of size at most k, hence it is polynomial-time solvable. However, the problem becomes NP-hard already for $|T| = 3$ terminals [124].

In this section, we show that EDGE MULTIWAY CUT is FPT parameterized by k. The following observation connects EDGE MULTIWAY CUT and the concept of important cuts:

Lemma 8.18 (Pushing lemma for EDGE MULTIWAY CUT**).** *Let $t \in T$ be an arbitrary terminal in an undirected graph G. If G has a multiway cut S, then it also has a multiway cut S^* with $|S^*| \leq |S|$ such that S^* contains an important $(t, T \setminus t)$-cut.*

Proof. If t is separated from $T \setminus t$ in G, then the statement trivially holds, as the empty set is an important $(t, T\setminus t)$-cut. Otherwise, let R be the component of $G \setminus S$ containing t. As S is a multiway cut, R is disjoint from $T \setminus t$ and hence $S_R = \Delta(R)$ is a $(t, T \setminus t)$-cut contained in S. By Proposition 8.7, there is an important $(t, T \setminus t)$-cut $S' = \Delta(R')$ with $R \subseteq R'$ and $|S'| \leq |S_R|$ (see Fig. 8.6). We claim that $S^* = (S \setminus S_R) \cup S'$, which has size at most $|S|$, is also a multiway cut, proving the statement of the lemma.

Replacing the $(t, T\setminus t)$-cut S_R in the solution with an important $(t, T\setminus t)$-cut S' cannot break the solution: as it is closer to $T \setminus t$, it can be even more helpful in separating the terminals in $T \setminus t$ from each other.

Clearly, there is no path between t and any other terminal of $T \setminus t$ in $G \setminus S^*$, as $S' \subseteq S^*$ is a $(t, T \setminus t)$-cut. Suppose therefore that there is a path P between two distinct terminals $t_1, t_2 \in T \setminus t$ in $G \setminus S^*$. If P goes through a vertex of $R \subseteq R'$, then it goes through at least one edge of the $(t, T \setminus t)$ cut $S' = \Delta(R')$, which is a subset of S^*, a contradiction. Therefore, we may assume that P is disjoint from R. Then P does not go through any edge of $S_R = \Delta(R)$ and therefore the fact that P is disjoint from S^* implies that it is disjoint from S as well, contradicting the assumption that S is a multiway cut. □

Using this observation, we can solve the problem by branching on the choice of an important cut and including it into the solution:

Theorem 8.19. EDGE MULTIWAY CUT *on a graph with n vertices and m edges can be solved in time $\mathcal{O}(4^k \cdot k^3 \cdot (n + m))$.*

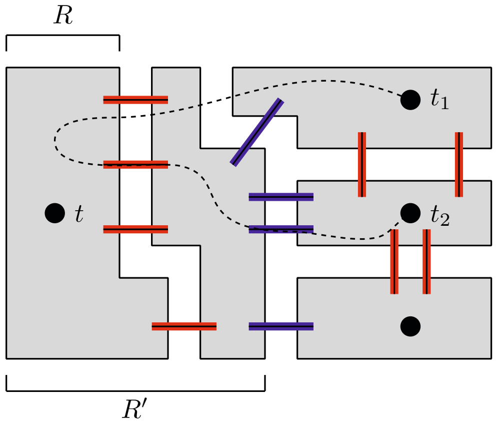


Fig. 8.6: Replacing $\Delta(R)$ with $\Delta(R')$ in the proof of Lemma 8.18 does not break the solution: if a path connecting $t_1, t_2 \in T \setminus t$ enters R, then it has to use an edge of $\Delta(R')$ as well

Proof. We solve the problem by a recursive branching algorithm. If all the terminals are separated from each other, then we are done. Otherwise, let $t \in T$ be a terminal not separated from the rest of the terminals. Let us use the algorithm of Theorem 8.16 to construct the set $\mathcal{S}_k$ consisting of every important $(t, T \setminus t)$-cut of size at most k. By Lemma 8.18, there is a solution that contains one of these cuts. Therefore, we branch on the choice of one of these cuts: for every important cut $S' \in \mathcal{S}_k$, we recursively solve the Edge Multiway Cut instance $(G \setminus S', T, k - |S'|)$. If one of these branches returns a solution S, then clearly $S \cup S'$ is a multiway cut of size at most k in G.

The correctness of the algorithm is clear from Lemma 8.18. We claim that the search tree explored by the algorithm has at most 4^k leaves. We prove this by induction on k, thus let us assume that the statement is true for every value less than k. This means that we know that the recursive call $(G \setminus S', T, k - |S'|)$ explores a search tree with at most $4^{k-|S'|}$ leaves. Note that the value of $k - |S'|$ is always nonnegative and the claim is trivial for $k = 0$. Using Lemma 8.12, we can bound the number of leaves of the search tree by

$$\sum_{S' \in \mathcal{S}_k} 4^{k-|S'|} \leq 4^k \cdot \sum_{S' \in \mathcal{S}_k} 4^{-|S'|} \leq 4^k.$$

Therefore, the total time spent at the leaves can be bounded by $\mathcal{O}(4^k k(n + m))$. As the height of the search tree is at most k, it has $\mathcal{O}(k4^k)$ nodes. If we use the algorithm of Theorem 8.16 to enumerate $\mathcal{S}_k$, then the time spent at an internal node with t children is $\mathcal{O}(tk^2(n + m))$. Summing the number of children for every internal node is exactly the number of edges, and hence can be bounded by $\mathcal{O}(k4^k)$. Therefore, the total work in the internal nodes is $\mathcal{O}(4^k k^3(n+m))$. □

A slight generalization of Edge Multiway Cut is Edge Multiway Cut for Sets, where the input is a graph G, pairwise disjoint sets $T_1, \dots, T_p$, and an integer k; the task is to find a set S of edges that pairwise separates these sets from each other, that is, there is no $T_i - T_j$ path in $G \setminus S$ for any $i \neq j$. Edge Multiway Cut for Sets can be reduced to Edge Multiway Cut simply by consolidating each set into a single vertex.

Theorem 8.20. Edge Multiway Cut for Sets *on a graph with n vertices and m edges can be solved in time $\mathcal{O}(4^k \cdot k^3 \cdot (n+m))$.*

Proof. We construct a graph G' as follows. The vertex set of G' is $(V(G) \setminus \bigcup_{i=1}^{p} T_i) \cup T$, where $T = \{v_1, \dots, v_p\}$. For every edge $xy \in E(G)$, there is a corresponding edge in G': if endpoint x or y is in T_i, then it is replaced by v_i. It is easy to see that there is a one-to-one correspondence between the solutions of the Edge Multiway Cut for Sets instance $(G, (T_1, \dots, T_p), k)$ and the Edge Multiway Cut instance (G', T, k). □

Another well-studied generalization of Edge Multiway Cut can be obtained if, instead of requiring that all the terminals be separated from each other, we require that a specified set of pairs of terminals be separated from each other. The input in the Edge Multicut problem is a graph G, pairs $(s_1, t_1), \dots, (s_\ell, t_\ell)$, and an integer k; the task is to find a set S of at most k edges such that $G \setminus S$ has no $s_i - t_i$ path for any $1 \le i \le \ell$. If we have a bounded number of pairs of terminals, Edge Multicut can be reduced to Edge Multiway Cut for Sets: we can guess how the components of $G \setminus S$ for the solution S partition the 2ℓ vertices s_i, t_i $(1 \le i \le \ell)$ and solve the resulting Edge Multiway Cut for Sets instance. The 2ℓ terminals can be partitioned in at most $(2\ell)^{2\ell}$ different ways, thus we can reduce Edge Multicut into at most that many instances of Edge Multiway Cut for Sets. This shows that Edge Multicut is FPT with combined parameters k and ℓ. We may also observe that the removal of at most k edges can partition a component of G into at most $(k+1)$ components, hence we need to consider at most $(k+1)^{2\ell}$ partitions.

Theorem 8.21. Edge Multicut *on a graph with n vertices and m edges can be solved in time $\mathcal{O}((2\ell)^{2\ell} \cdot 4^k \cdot k^3(n+m))$ or in time $\mathcal{O}((k+1)^{2\ell} \cdot 4^k \cdot k^3(n+m))$.*

It is a more challenging question whether the problem is FPT parameterized by k (the size of the solution) only. The proof of this requires several nontrivial technical steps and is beyond the scope of this chapter. We remark that one of the papers proving the fixed-parameter tractability of Edge Multicut was the paper introducing the random sampling of important separators technique [357].

Theorem 8.22 ([59, 357]). Edge Multicut *on an n-vertex graph can be solved in time $2^{\mathcal{O}(k^3)} \cdot n^{\mathcal{O}(1)}$.*

8.4 (p, q)-clustering

The typical goal in clustering problems is to group a set of objects such that, roughly speaking, similar objects appear together in the same group. There are many different ways of defining the input and the objective of clustering problems: depending on how similarity of objects is interpreted, what constraints we have on the number and size of the clusters, and how the quality of the clustering is measured, we can define very different problems of this form. Often a clustering problem is defined in terms of a (weighted) graph: the objects are the vertices and the presence of a (weighted) edge indicates similarity of the two objects and the absence of an edge indicates that the two objects are not considered to be similar.

In this section, we present an algorithm for a particular clustering problem on graphs that is motivated by partitioning circuits into several field programmable gate arrays (FPGAs). We say that a set $C \subseteq V(G)$ is a (p, q)-*cluster* if $|C| \leq p$ and $d(C) \leq q$. A (p, q)-*partition* of G is a partition of $V(G)$ into (p, q)-clusters. Given a graph G and integers p and q, the (p, q)-PARTITION problem asks if G has a (p, q)-partition. The main result of this section is showing that (p, q)-PARTITION is FPT parameterized by q. The proof is based on the technique of random sampling of important separators and serves as a self-contained demonstration of this technique.

- An uncrossing argument shows that the trivial necessary condition that every vertex is in a (p, q)-cluster is also a sufficient condition.
- Checking whether v is in a (p, q)-cluster is trivial in $n^{\mathcal{O}(q)}$ time, but an FPT algorithm parameterized by q is more challenging.
- The crucial observation is that a minimal (p, q)-cluster containing v is surrounded by important cuts.
- A randomized reduction allows us to reduce finding a (p, q)-cluster containing v to a knapsack-like problem.
- The derandomization of the reduction uses splitters, introduced in Section 5.6.

A necessary condition for the existence of (p, q)-partition is that for every vertex $v \in V(G)$ there exists a (p, q)-cluster that contains v. Very surprisingly, it turns out that this trivial necessary condition is actually sufficient for the existence of a (p, q)-partition. The proof (Lemma 8.24) needs a variant of submodularity: we say that a set function $f : 2^{V(G)} \to \mathbb{R}$ is *posimodular* if it satisfies the following inequality for every $A, B \subseteq V(G)$:

$$f(A) + f(B) \geq f(A \setminus B) + f(B \setminus A). \tag{8.2}$$

This inequality is very similar to the submodular inequality (8.1) and the proof that d_G has this property is a case analysis similar to the proof of Theorem 8.3. The only difference is that edges of type 4 contribute 2 to the left-hand side and 0 to the right-hand side, while edges of type 5 contribute 2 to both sides.

Theorem 8.23. *The function d_G is posimodular for every undirected graph G.*

More generally, it can be shown that every symmetric submodular function (that is, when $f(X) = f(V(G) \setminus X)$ for every $X \subseteq V(G)$) is posimodular (see Exercise 8.3).

We are now ready to prove that every vertex being in a (p,q)-cluster is a sufficient condition for the existence of a (p,q)-partition.

Lemma 8.24. *Let G be an undirected graph and let $p, q \geq 0$ be two integers. If every $v \in V(G)$ is contained in some (p,q)-cluster, then G has a (p,q)-partition. Furthermore, given a set of (p,q)-clusters $C_1, \ldots, C_n$ whose union is $V(G)$, a (p,q)-partition can be found in polynomial time.*

Proof. Let us consider a collection $C_1, \ldots, C_n$ of (p,q)-clusters whose union is $V(G)$. If the sets are pairwise disjoint, then they form a partition of $V(G)$ and we are done. If $C_i \subseteq C_j$, then the union remains $V(G)$ even after throwing away C_i. Thus we can assume that no set is contained in another.

The posimodularity of the function d_G allows us to uncross two clusters C_i and C_j if they intersect.

Suppose that C_i and C_j intersect. Now either $d(C_i) \geq d(C_i \setminus C_j)$ or $d(C_j) \geq d(C_j \setminus C_i)$ must be true: it is not possible that both $d(C_i) < d(C_i \setminus C_j)$ and $d(C_j) < d(C_j \setminus C_i)$ hold, as this would violate the posimodularity of d. Suppose that $d(C_j) \geq d(C_j \setminus C_i)$. Now the set $C_j \setminus C_i$ is also a (p,q)-cluster: we have $d(C_j \setminus C_i) \leq d(C_j) \leq q$ by assumption and $|C_j \setminus C_i| < |C_j| \leq p$. Thus we can replace C_j by $C_j \setminus C_i$ in the collection: it will remain true that the union of the clusters is $V(G)$. Similarly, if $d(C_i) \geq d(C_i \setminus C_j)$, then we can replace C_i by $C_i \setminus C_j$.

Repeating these steps (throwing away subsets and resolving intersections), we eventually arrive at a pairwise-disjoint collection of (p,q)-clusters. Each step decreases the number of cluster pairs (C_i, C_j) that have nonempty intersection. Therefore, this process terminates after a polynomial number of steps. □

The proof of Lemma 8.24 might suggest that we can obtain a partition by simply taking, for every vertex v, a (p,q)-cluster C_v that is inclusion-wise minimal with respect to containing v. However, such clusters can still cross. For example, consider a graph on vertices a, b, c, d where every pair of vertices except a and d are adjacent. Suppose that $p = 3$, $q = 2$. Then $\{a,b,c\}$ is a minimal cluster containing b (as more than two edges are going out of each of $\{b\}$, $\{b,c\}$, and $\{a,b\}$) and $\{b,c,d\}$ is a minimal cluster containing c. Thus unless we choose the minimal clusters more carefully in a coordinated way, they are not guaranteed to form a partition. In other words, there are two symmetric solutions $(\{a, b, c\}, \{d\})$ and $(\{a\}, \{b, c, d\})$ for the problem, and the clustering algorithm has to break this symmetry somehow.

In light of Lemma 8.24, it is sufficient to find a (p, q)-cluster C_v for each vertex $v \in V(G)$. If there is a vertex v for which there is no such cluster C_v, then obviously there is no (p, q)-partition; if we have such a C_v for every vertex v, then Lemma 8.24 gives us a (p, q)-partition in polynomial time. Therefore, in the rest of the section, we are studying the (p, q)-Cluster problem, where, given a graph G, a vertex $v \in V(G)$, and integers p and q, it has to be decided if there is a (p, q)-cluster containing v.

For fixed q, the (p, q)-Cluster problem can be solved by brute force: enumerate every set F of at most q edges and check if the component of $G \setminus F$ containing v is a (p, q)-cluster. If C_v is a (p, q)-cluster containing v, then we find it when $F = \Delta(C_v)$ is considered by the enumeration procedure.

Theorem 8.25. *(p, q)-Cluster can be solved in time $n^{\mathcal{O}(q)}$ on an n-vertex graph.*

The main result of the section is showing that it is possible to solve (p, q)-Cluster (and hence (p, q)-Partition) more efficiently: in fact, it is fixed-parameter tractable parameterized by q. By Lemma 8.24, all we need to show is that (p, q)-Cluster is fixed-parameter tractable parameterized by q. We introduce a somewhat technical variant of (p, q)-Cluster, the Satellite Problem, which is polynomial-time solvable. Then we show how to solve (p, q)-Cluster using an algorithm for the Satellite Problem.

The input of the Satellite Problem is a graph G, integers p, q, a vertex $v \in V(G)$, and a partition $(V_0, V_1, \dots, V_r)$ of $V(G)$ such that $v \in V_0$ and there is no edge between V_i and V_j for any $1 \leq i < j \leq r$. The task is to find a (p, q)-cluster C satisfying $V_0 \subseteq C$ such that for every $1 \leq i \leq r$, either $C \cap V_i = \emptyset$ or $V_i \subseteq C$ (see Fig. 8.7).

Since the sets $\{V_i\}$ form a partition of $V(G)$, we have $r \leq n$. For every V_i $(1 \leq i \leq r)$, we have to decide whether to include or exclude it from the solution cluster C. If we exclude V_i from C, then $d(C)$ increases by $d(V_i)$, the number of edges between V_0 and V_i. If we include V_i into C, then $|C|$ increases by $|C|$.

To solve Satellite Problem, we need to solve the knapsack-like problem of including sufficiently many V_i's such that $d(C) \leq q$, but not including too many to ensure $|C| \leq p$.

Lemma 8.26. *The* Satellite Problem *can be solved in polynomial time.*

Proof. For a subset S of $\{1, \dots, r\}$, we define $C(S) = V_0 \cup \bigcup_{i \in S} V_i$. Notice that $d(C(S)) = d(V_0) - \sum_{i \in S} d(V_i)$. Hence, we can reformulate the Satellite

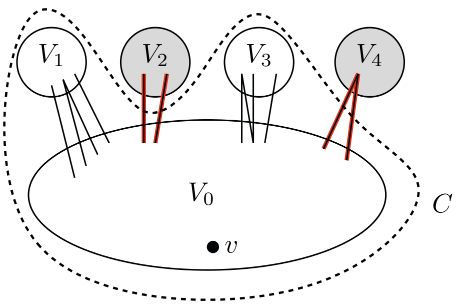


Fig. 8.7: Instance of SATELLITE PROBLEM with a solution C. Excluding V_2 and V_4 from C decreased the size of C by the grey area, but increased $d(C)$ by the red edges

PROBLEM as finding a subset S of $\{1, \ldots, r\}$ such that $\sum_{i \in S} d(V_i) \geq d(V_0) - q$ and $\sum_{i \in S} |V_i| \leq p - |V_0|$. Thus, we can associate with every i an item with value $d(V_i)$ and weight $|V_i|$. The objective is to find a set of items with total value at least $\mathbf{v}_{\text{target}} := d(V_0) - q$ and total weight at most $\mathbf{w}_{\max} := p - |V_0|$. This problem is known as KNAPSACK and can be solved in polynomial time by a classical dynamic-programming algorithm in time polynomial in the number r of items and the maximum weight $\mathbf{w}_{\max}$ (assuming that the weights are positive integers). In our case, both r and $\mathbf{w}_{\max}$ are polynomial in the size of the input, hence a polynomial-time algorithm for SATELLITE PROBLEM follows.

For completeness, we briefly sketch how the dynamic-programming algorithm works. For $0 \leq i \leq r$ and $0 \leq j \leq \mathbf{w}_{\max}$, we define $T[i, j]$ to be the maximum value of a set $S \subseteq \{1, \ldots, i\}$ that has total weight at most j. By definition, $T[0, j] = 0$ for every j. Assuming that we have computed $T[i-1, j]$ for every j, we can then compute $T[i, j]$ for every j using the following recurrence relation:

$$T[i, j] = \max\{T[i-1, j], T[i-1, j - |V_i|] + d(V_i)\}.$$

That is, the optimal set S either does not include item i (in which case it is also an optimal set for $T[i-1, j]$), or includes item i (in which case removing item i decreases the value by $d(V_i)$, decreases the weight bound by $|V_i|$, and what remains should be optimal for $T[i-1, j - |V_i|]$). After computing every value $T[i, j]$, we can check whether v is in a suitable cluster by checking whether $T[r, \mathbf{w}_{\max}] \geq \mathbf{v}_{\text{target}}$ holds. $\square$

What remains to be shown is how to reduce (p, q)-CLUSTER to the SATELLITE PROBLEM. We first present a randomized version of this reduction as it is cleaner and conceptually simpler. Then we discuss how splitters, introduced in Section 5.6, can be used to derandomize the reduction.

At this point it is highly unclear how to reduce (p, q)-Cluster to the Satellite Problem. In the latter problem, the parts V_i that may or may not be included into a cluster are very well behaved (disjoint and nonadjacent), whereas in (p, q)-Cluster we do not have any insight yet into the structure of connected components of $G \setminus C$, where C is the cluster we are looking for.

First, the crucial observation is that we may assume that each connected component of $G \setminus C$ is surrounded by an important cut, leading to only an FPT number of candidates for such components. Second, we randomly filter out a fraction of the candidates, so that we can obtain a Satellite Problem instance from the remaining ones.

The following definition connects the notion of important cuts with our problem.

Definition 8.27 (Important set). We say that a set $X \subseteq V(G)$, $v \notin X$ is *important* if

1. $d(X) \leq q$,
2. $G[X]$ is connected,
3. there is no $Y \supset X$, $v \notin Y$ such that $d(Y) \leq d(X)$ and $G[Y]$ is connected.

It is easy to see that X is an important set if and only if $\Delta(X)$ is an important (u, v)-cut of size at most q for every $u \in X$. Thus we can use Theorem 8.11 to enumerate every important set, and Lemma 8.12 to give an upper bound on the number of important sets. The following lemma establishes the connection between important sets and finding (p, q)-clusters: we can assume that the components of $G \setminus C$ for the solution C are important sets.

Lemma 8.28. *Let C be an inclusion-wise minimal (p, q)-cluster containing v. Then every component of $G \setminus C$ is an important set.*

Proof. Let X be a component of $G \setminus C$. It is clear that X satisfies the first two properties of Definition 8.27 (note that $\Delta(X) \subseteq \Delta(C)$ implies $d(X) \leq d(C) \leq q$). Thus let us suppose that there is a $Y \supset X$, $v \notin Y$ such that $d(Y) \leq d(X)$ and $G[Y]$ is connected. Let $C' := C \setminus Y$. Note that C' is a proper subset of C: every neighbor of X is in C, thus a connected superset of X has to contain at least one vertex of C. It is easy to see that C' is a (p, q)-cluster: we have $\Delta(C') \subseteq (\Delta(C) \setminus \Delta(X)) \cup \Delta(Y)$ and therefore $d(C') \leq d(C) - d(X) + d(Y) \leq d(C) \leq q$ and $|C'| < |C| \leq p$. This contradicts the minimality of C. $\square$

We are now ready to present the randomized version of the reduction.

Lemma 8.29. *Given an n-vertex graph G, vertex $v \in V(G)$, and integers p and q, we can construct in time $2^{\mathcal{O}(q)} \cdot n^{\mathcal{O}(1)}$ an instance I of the* Satellite Problem *such that*

- *If some (p, q)-cluster contains v, then I is a yes-instance with probability $2^{-\mathcal{O}(q)}$,*
- *If there is no (p, q)-cluster containing v, then I is a no-instance.*

Proof. We may assume that G is connected: if there is a (p, q)-cluster containing v, then there is such cluster contained in the connected component of v.

For every $u \in V(G)$, $u \neq v$, let us use the algorithm of Theorem 8.11 to enumerate every important (u, v)-cut of size at most q. For every such cut S, let us put the component K of $G \setminus S$ containing u into the collection $\mathcal{X}$. Note that the same component K can be obtained for more than one vertex u, but we put only one copy into $\mathcal{X}$.

Let $\mathcal{X}'$ be a subset of $\mathcal{X}$, where each member K of $\mathcal{X}$ is chosen with probability $4^{-d(K)}$ independently at random. Let Z be the union of the sets in $\mathcal{X}'$, let $V_1, \ldots, V_r$ be the connected components of $G[Z]$, and let $V_0 = V(G) \setminus Z$. It is clear that V_0, V_1, $\ldots$, V_r describe an instance I of the Satellite Problem, and a solution for I gives a (p, q)-cluster containing v. Thus we only need to show that if there is a (p, q)-cluster C containing v, then I is a yes-instance with probability $2^{-\mathcal{O}(q)}$.

We show that the reduction works if the union Z of the selected important sets satisfies two constraints: it has to cover every component of $G \setminus C$ and it has to be disjoint from the vertices on the boundary of C. The probability of the event that these constraints are satisfied is $2^{-\mathcal{O}(q)}$.

Let C be an inclusion-wise minimal (p, q)-cluster containing v. We define $\partial(C)$ to be the *border* of C: the set of vertices in C with a neighbor in $V(G) \setminus C$, or in other words, the vertices of C incident to $\Delta(C)$. Let $K_1, \ldots, K_t$ be the components of $G \setminus C$. Note that every edge of $\Delta(C)$ enters some K_i, thus $\sum_{i=1}^{t} d(K_i) = d(C) \leq q$. By Lemma 8.28, every K_i is an important set, and hence it is in $\mathcal{X}$. Consider the following two events:

(E1) Every component K_i of $G \setminus C$ is in $\mathcal{X}'$ (and hence $K_i \subseteq Z$).
(E2) $Z \cap \partial(C) = \emptyset$.

The probability that (E1) holds is $\prod_{i=1}^{t} 4^{-d(K_i)} = 4^{-\sum_{i=1}^{t} d(K_i)} \geq 4^{-q}$. Event (E2) holds if for every $w \in \partial(C)$, no set $K \in \mathcal{X}$ with $w \in K$ is selected into $\mathcal{X}'$. It follows directly from the definition of important cuts that for every $K \in \mathcal{X}$ with $w \in K$, the set $\Delta(K)$ is an important (w, v)-cut. Thus by Lemma 8.12, $\sum_{K \in \mathcal{X}, w \in K} 4^{-|d(K)|} \leq 1$. To bound the probability of the event $Z \cap \partial(C) = \emptyset$, we have to bound the probability that no important set intersecting $\partial(C)$ is selected:

$$\begin{aligned}
\prod_{\substack{K\in\mathcal{X}\\ K\cap\partial(C)\neq\emptyset}} (1-4^{-d(K)}) &\geq \prod_{w\in\partial(C)} \prod_{\substack{K\in\mathcal{X}\\ w\in K}} (1-4^{-d(K)})\\
&\geq \prod_{w\in\partial(C)} \prod_{\substack{K\in\mathcal{X}\\ w\in K}} \exp\left(\frac{-4^{-d(K)}}{(1-4^{-d(K)})}\right)\\
&\geq \prod_{w\in\partial(C)} \prod_{\substack{K\in\mathcal{X}\\ w\in K}} \exp\left(-\frac{4}{3}\cdot 4^{-d(K)}\right)\\
&= \prod_{w\in\partial(C)} \exp\left(-\frac{4}{3}\cdot \sum_{\substack{K\in\mathcal{X}\\ w\in K}} 4^{-d(K)}\right)\\
&\geq (e^{-\frac{4}{3}})^{|\partial(C)|} \geq e^{-4q/3}.
\end{aligned}$$

In the first inequality, we use the fact that every term is less than 1 and every term on the right-hand side appears at least once on the left-hand side. In the second inequality, we use the fact that $1+x \geq \exp(x/(1+x))$ for every $x > -1$. In the third inequality, we use the fact that $d(K) \geq 1$ follows from the assumption that the graph G is connected, hence $1-4^{-d(K)} \geq 3/4$ holds.

Events (E1) and (E2) are independent: (E1) is a statement about the selection of a specific subcollection $A \subseteq \mathcal{X}$ of at most q sets that are disjoint from $\partial(C)$, while (E2) is a statement about not selecting any member of a specific subcollection $B \subseteq \mathcal{X}$ of at most $|\partial(C)|\cdot 4^q$ sets intersecting S. Thus, with probability at least $2^{-\mathcal{O}(q)}$, both (E1) and (E2) hold.

Suppose that both (E1) and (E2) hold. We show that the corresponding instance I of the Satellite Problem is a yes-instance. In this case, every component K_i of $G\setminus C$ is a component V_j of $G[Z]$: $K_i \subseteq Z$ by (E1) and every neighbor of K_i is outside Z. Thus C is a solution of I, as it can be obtained as the union of V_0 and some components of $G[Z]$. □

Lemma 8.29 gives a randomized reduction from (p,q)-Cluster to the Satellite Problem, which is polynomial-time solvable (Lemma 8.26). Therefore, there is a randomized algorithm for (p,q)-Cluster with running time $2^{\mathcal{O}(q)}\cdot n^{\mathcal{O}(1)}$ and success probability $p_{\text{correct}} = 2^{-\mathcal{O}(q)}$. By repeating the algorithm $\lceil 1/p_{\text{correct}}\rceil = 2^{\mathcal{O}(q)}$ times, the probability of a false answer decreases to $(1-p_{\text{correct}})^{\lceil 1/p_{\text{correct}}\rceil} \leq 1/e$ (using $1-x \leq e^{-x}$).

Corollary 8.30. *There is a $2^{\mathcal{O}(q)}\cdot n^{\mathcal{O}(1)}$ time randomized algorithm for (p,q)-*Cluster *with constant error probability.*

To derandomize the proof of Lemma 8.29 and to obtain a deterministic version of Corollary 8.30, we use splitters, which were introduced in Section 5.6. We present here only a simpler version of the derandomization, where the dependence on q in the running time is of the form $2^{\mathcal{O}(q^2)}$. It is possible to improve this dependence to $2^{\mathcal{O}(q)}$, matching the bound in Corollary 8.30.

However, the details are somewhat technical and hence we omit here the description of this improvement.

Recall that Theorem 5.16 provided us with an (n, k, k^2)-*splitter* of size $\mathcal{O}(k^6 \log k \log n)$: a family $\mathcal{F}$ of functions from $[n]$ to $[k^2]$ such that for any subset $X \subseteq [n]$ with $|X| = k$, one of the functions f in the family is injective on X.

The main idea of the derandomization is to replace the random selection of the subfamily $\mathcal{X}'$ by a deterministic selection based on the functions in a splitter.

Theorem 8.31. (p, q)-CLUSTER *can be solved in time* $2^{\mathcal{O}(q^2)} \cdot n^{\mathcal{O}(1)}$.

Proof. In the algorithm of Lemma 8.29, a random subset of a universe $\mathcal{X}$ of size $s = |\mathcal{X}| \leq 4^q \cdot n$ is selected. If the (p, q)-CLUSTER problem has a solution C, then there is a collection $A \subseteq \mathcal{X}$ of at most $a := q$ sets and a collection $B \subseteq \mathcal{X}$ of at most $b := q \cdot 4^q$ sets such that if every set in A is selected and no set in B is selected, then (E1) and (E2) hold. Instead of selecting a random subset, we try every function f in an $(s, a + b, (a + b)^2)$-splitter $\mathcal{F}$ (obtained through Theorem 5.16), and every subset $F \subseteq [(a + b)^2]$ of size a (there are $\binom{(a+b)^2}{a} = 2^{\mathcal{O}(q^2)}$) such sets F). For a particular choice of f and F, we select those sets $S \in \mathcal{X}$ into $\mathcal{X}'$ for which $f(S) \in F$. The size of the splitter $\mathcal{F}$ is $2^{\mathcal{O}(q)} \cdot \log n$ and the number of possibilities for F is $2^{\mathcal{O}(q^2)}$. Therefore, we construct $2^{\mathcal{O}(q^2)} \cdot \log n$ instances of the SATELLITE PROBLEM.

By the definition of the splitter, there will be a function f that is injective on $A \cup B$, and there is a subset F such that $f(S) \in F$ for every set S in A and $f(S) \notin F$ for every set S in B. For such an f and F, the selection will ensure that (E1) and (E2) hold. This means that the constructed instance of the SATELLITE PROBLEM corresponding to f and F has a solution as well. Thus solving every constructed instance of the SATELLITE PROBLEM in polynomial time gives a $2^{\mathcal{O}(q^2)} \cdot n^{\mathcal{O}(1)}$ algorithm for (p, q)-CLUSTER. □

One can obtain a more efficient derandomization by a slight change of the construction in the proof of Theorem 8.31 and a more careful analysis. This gives a deterministic algorithm with $2^{\mathcal{O}(q)}$ dependence on q.

Theorem 8.32 ([324]). (p, q)-CLUSTER *can be solved in time* $2^{\mathcal{O}(q)} \cdot n^{\mathcal{O}(1)}$.

Finally, we have shown in Lemma 8.24 that there is a reduction from (p, q)-PARTITION to (p, q)-CLUSTER, hence fixed-parameter tractability follows for (p, q)-PARTITION as well.

Theorem 8.33. (p, q)-PARTITION *can be solved in time* $2^{\mathcal{O}(q)} \cdot n^{\mathcal{O}(1)}$.

8.5 Directed graphs

Problems on directed graphs are notoriously more difficult than problems on undirected graphs. This phenomenon has been observed equally often in the area of polynomial-time algorithms, approximability, and fixed-parameter tractability. Let us see if the techniques based on important cuts survive the generalization to directed graphs.

Given a directed graph G and two disjoint sets of vertices $X, Y \subseteq V(G)$, a *directed* (X,Y)*-cut* is a subset S of edges such that $G \setminus S$ has no directed path from X to Y (but it may have directed paths from Y to X). We denote by $\Delta^+_G(R)$ the set of edges starting in R and ending in $V(G) \setminus R$. As for undirected graphs, every minimal (X,Y)-cut S can be expressed as $\Delta^+_G(R)$ for some $X \subseteq R \subseteq V(G) \setminus R$. Important cuts can be defined analogously for directed graphs.

Definition 8.34 (Important cut in directed graphs). Let G be a directed graph and let $X, Y \subseteq V(G)$ be two disjoint sets of vertices. Let $S \subseteq E(G)$ be an (X,Y)-cut, and let R be the set of vertices reachable from X in $G \setminus S$. We say that S is an *important* (X,Y)*-cut* if it is minimal and there is no (X,Y)-cut S' with $|S'| \leq |S|$ such that $R \subset R'$, where R' is the set of vertices reachable from X in $G \setminus S'$.

Proposition 8.7 generalizes to directed graphs in a straightforward way.

Proposition 8.35. *Let G be a directed graph and $X, Y \subseteq V(G)$ be two disjoint sets of vertices. Let S be an (X,Y)-cut and let R be the set of vertices reachable from X in $G \setminus S$. Then there is an important (X,Y)-cut $S' = \Delta^+(R')$ (possibly, $S' = S$) such that $|S'| \leq |S|$ and $R \subseteq R'$.*

We state without proof that the bound of 4^k of Theorem 8.11 holds also for directed graphs.

Theorem 8.36. *Let $X, Y \subseteq V(G)$ be two sets of vertices in a directed graph G with n vertices and m edges, let $k \geq 0$ be an integer, and let $\mathcal{S}_k$ be the set of all (X,Y)-important cuts of size at most k. Then $|\mathcal{S}_k| \leq 4^k$ and $\mathcal{S}_k$ can be constructed in time $\mathcal{O}(|\mathcal{S}_k| \cdot k^2(n+m))$.*

Also, an analogue of the bound of Lemma 8.12 holds for directed important cuts.

Lemma 8.37. *Let G be a directed graph and let $X, Y \subseteq V(G)$ be two disjoint sets of vertices. If $\mathcal{S}$ is the set of all important (X,Y)-cuts, then $\sum_{S \in \mathcal{S}} 4^{-|S|} \leq 1$.*

Given a directed graph G, a set $T \subseteq V(G)$ of terminals, and an integer k, the Directed Edge Multiway Cut problem asks for a set S of at most k edges such that $G \setminus S$ has no directed $t_1 \to t_2$ path for any two distinct $t_1, t_2 \in T$. Theorem 8.36 gives us some hope that we would be able to use the techniques of Section 8.3 for undirected Edge Multiway Cut also for

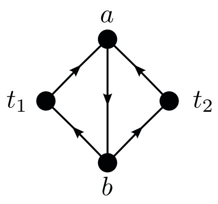


Fig. 8.8: The unique important (t_1, t_2)-cut is the edge (b, t_2), the unique important (t_2, t_1)-cut is the edge (b, t_1), but the unique minimum edge multiway cut is the edge (a, b)

directed graphs. However, the pushing lemma (Lemma 8.18) is not true on directed graphs, even for $|T| = 2$; see Fig. 8.8 for a simple counterexample where the unique minimum multiway cut does not use any important cut between the two terminals. The particular point where the proof of Lemma 8.18 breaks down in the directed setting is where a $T \setminus t \to t$ path appears after replacing S with S^*. This means that a straightforward generalization of Theorem 8.19 to directed graphs is not possible. Nevertheless, using different arguments (in particular, using the random sampling of important separators technique), it is possible to show that Directed Edge Multiway Cut is FPT parameterized by the size k of the solution.

Theorem 8.38 ([91, 89]). Directed Edge Multiway Cut *on an n-vertex graph can be solved in time $2^{\mathcal{O}(k^2)} \cdot n^{\mathcal{O}(1)}$.*

The input of the Directed Edge Multicut problem is a directed graph G, a set of pairs (s_1, t_1), ..., (s_ℓ, t_ℓ), and an integer k, the task is to find a set S of at most k edges such that $G \setminus S$ has no directed $s_i \to t_i$ path for any $1 \leq i \leq k$. This problem is more general than Directed Edge Multiway Cut: the requirement that no terminal be reachable from any other terminal in $G \setminus S$ can be expressed by a set of $|T|(|T| - 1)$ pairs (s_i, t_i).

In contrast to the undirected version (Theorem 8.22), Directed Edge Multicut is W[1]-hard parameterized by k, even on directed acyclic graphs. But the problem is interesting even for small values of ℓ. The argument of Theorem 8.21 (essentially, guessing how the components of $G \setminus S$ partition the terminals) no longer works, as the relation of the terminals in $G \setminus S$ can be more complicated in a directed graph than just a simple partition into connected components. The case $\ell = 2$ can be reduced to Directed Edge Multiway Cut in a simple way (see Exercise 8.14), thus Theorem 8.38 implies that Directed Edge Multicut for $\ell = 2$ is FPT parameterized by k. The case of a fixed $\ell \geq 3$ and the case of jointly parameterizing by ℓ and k are open for general directed graphs. However, there is a positive answer for directed acyclic graphs.

Theorem 8.39 ([309]). Directed Edge Multiway Cut *on directed acyclic graphs is FPT parameterized by ℓ and k.*

There is a variant of directed multicut where the pushing argument does work. Skew Edge Multicut has the same input as Directed Edge Multicut, but now the task is to find a set S of at most k edges such that $G \setminus S$ has no directed $s_i \to t_j$ path for any $i \geq j$. While this problem is somewhat unnatural, it will be an important ingredient in the algorithm for Directed Feedback Vertex Set in Section 8.6.

Lemma 8.40 (Pushing lemma for Skew Edge Multicut**).** *Let $(G, ((s_1, t_1), \dots, (s_\ell, t_\ell)), k)$ be an instance of* Skew Edge Multicut*. If the instance has a solution S, then it has a solution S^* with $|S^*| \leq |S|$ that contains an important $(s_\ell, \{t_1, \dots, t_\ell\})$-cut.*

Proof. Let $T = \{t_1, \dots, t_\ell\}$ and let R be the set of vertices reachable from s_ℓ in $G \setminus S$. As S is a solution, R is disjoint from T and hence $\Delta^+(R)$ is an (s_ℓ, T)-cut. If $S_R = \Delta^+(R)$ is an important (s_ℓ, T)-cut, then we are done: S contains every edge of $\Delta^+(R)$. Otherwise, Proposition 8.35 implies that there is an important (X, Y)-cut $S' = \Delta^+(R')$ such that $R \subseteq R'$ and $|S'| \leq |S_R|$. We claim that $S^* = (S \setminus S_R) \cup S'$, which has size at most $|S|$, is also a solution, proving the statement of the lemma.

The reason why replacing S_R with the important (s_ℓ, T)-cut S' does not break the solution is because every path that we need to cut in Skew Edge Multicut ends in $T = \{t_1, \dots, t_\ell\}$.

Clearly, there is no path between s_ℓ and any terminal in T in $G \setminus S^*$, as $S' \subseteq S^*$ is an (s_ℓ, T)-cut. Suppose therefore that there is an $s_i \to t_j$ path P for some $i \geq j$ in $G \setminus S^*$. If P goes through a vertex of $R \subseteq R'$, then it goes through at least one edge of the (s_ℓ, T)-cut $S' = \Delta^+(R')$, which is a subset of S^*, a contradiction. Therefore, we may assume that P is disjoint from R. Then P does not go through any edge of $S_R = \Delta(R)$ and therefore the fact that it is disjoint from S^* implies that it is disjoint from S, contradicting the assumption that S is a solution. □

Equipped with Lemma 8.37 and Lemma 8.40, a branching algorithm very similar to the proof of Theorem 8.19 shows the fixed-parameter tractability of Skew Edge Multicut.

Theorem 8.41. Skew Edge Multicut *on a graph with n vertices and m edges can be solved in time $\mathcal{O}(4^k \cdot k^3 \cdot (n + m))$.*

8.6 Directed Feedback Vertex Set

For undirected graphs, the Feedback Vertex Set problem asks for a set of at most k vertices whose deletion makes the graph acyclic, that is, a forest.

We have already seen a branching algorithm for this problem in Section 3.3, an algorithm based on iterative compression in Section 4.3, and a simple randomized algorithm in Section 5.1. It is easy to see that the edge version of Feedback Vertex Set is polynomial-time solvable.

For directed graphs, both the edge and the vertex versions are interesting. A *feedback vertex set* of a directed graph G is set $S \subseteq V(G)$ of vertices such that $G \setminus S$ has no directed cycle, while a *feedback arc set*[2] is a set $S \subseteq E(G)$ of edges such that $G \setminus S$ has no directed cycle. Given a graph G and an integer k, Directed Feedback Vertex Set asks for a feedback vertex set of size at most k. The Directed Feedback Arc Set is defined analogously; recall that in Section 2.2.2 we have designed a polynomial kernel for a special case of Directed Feedback Arc Set, namely Feedback Arc Set in Tournaments.

There is a simple reduction from the vertex version to the edge version (Exercise 8.16 asks for a reduction in the reverse direction).

Proposition 8.42. Directed Feedback Vertex Set *can be reduced to* Directed Feedback Arc Set *in linear time without increasing the parameter.*

Proof. Let (G, k) be an instance of Directed Feedback Vertex Set. We construct a graph G', where two vertices v_{in} and v_{out} correspond to each vertex $v \in V(G)$. For every edge $(a, b) \in E(G)$, we introduce the edge $(a_{\text{out}}, b_{\text{in}})$ into G'. Additionally, for every $v \in V(G)$, we introduce the edge $(v_{\text{in}}, v_{\text{out}})$ into G'. Observe that there is a one-to-one correspondence between the directed cycles of length exactly ℓ in G and the directed cycles of length exactly 2ℓ in G'.

We claim that G has a feedback vertex set S of size at most k if and only if G' has a feedback arc set S' of size at most k. For the forward direction, given a feedback vertex set $S \subseteq V(G)$ of G, let S' contain the corresponding edges of G', that is, $S' = \{(v_{\text{in}}, v_{\text{out}}) : v \in S\}$. If there is a directed cycle C' in $G' \setminus S'$, then the corresponding cycle C in G contains at least one vertex $v \in S$. But then the corresponding edge $(v_{\text{in}}, v_{\text{out}})$ of C' is in S', a contradiction.

For the reverse direction, let $S' \subseteq E(G)$ be a feedback arc set of G'. We may assume that every edge of S' is of the form $(v_{\text{in}}, v_{\text{out}})$: if $(a_{\text{out}}, b_{\text{in}}) \in S'$, then we may replace it with $(b_{\text{in}}, b_{\text{out}})$, as every directed cycle going through the former edge goes through the latter edge as well. Let $S = \{v : (v_{\text{in}}, v_{\text{out}}) \in S'\}$. Now, if $G \setminus S$ has a directed cycle C, then the corresponding directed cycle C' of G goes through an edge $(v_{\text{in}}, v_{\text{out}}) \in S'$, implying that vertex v of C is in S, a contradiction. □

[2] Some authors prefer to use the term "arc" for the edges of directed graphs. While we are using the term "directed edge" in this book, the term "arc" is very commonly used in the context of the Directed Feedback Arc Set problem, hence we also use it in this section.

In the rest of the section, we show the fixed-parameter tractability of Directed Feedback Arc Set using the technique of iterative compression, introduced in Chapter 4.

We apply a trick that is often useful when solving edge-deletion problems using iterative compression: the initial solution that we need to compress is a vertex set, not an edge set.

That is, the input of the Directed Feedback Arc Set Compression problem is a directed graph G, an integer k, and a feedback *vertex* set W of G; the task is to find a feedback *arc* set S of size at most k. We show, by a reduction to Skew Edge Multicut, that this problem is FPT parameterized by $|W|$ and k.

Lemma 8.43. Directed Feedback Arc Set Compression *can be solved in time $\mathcal{O}(|W|! \cdot 4^k \cdot k^3(n+m))$ on a graph with n vertices and m edges.*

Proof. The task is to find a set S of edges where $G \backslash S$ has no directed cycle, or, equivalently, has a topological ordering. Every topological ordering induces an ordering on the set W of vertices. Our algorithm starts by guessing an ordering $W = \{w_1, \ldots, w_{|W|}\}$ and the rest of the algorithm works under the assumption that there is a solution compatible with this ordering. There are $|W|!$ possibilities for the ordering of W, thus the running time of the rest of the algorithm has to be multiplied by this factor.

Every directed cycle contains a $w_i \to w_j$ path for some $i \geq j$: the cycle contains at least one vertex of W and has to go "backwards" at some point. Therefore, cutting all such paths (which is a problem similar to Skew Edge Multicut) gives a feedback arc set.

We build a graph G' the following way. We replace every vertex w_i with two vertices s_i, t_i, and an edge (t_i, s_i) between them. We define $E_W = \{(t_i, s_i) : 1 \leq i \leq |W|\}$. Then we replace every edge (a, w_i) with (a, t_i), every edge (w_i, a) with (s_i, a), and every edge (w_i, w_j) with (s_i, t_j) (see Fig. 8.9) Let us define the Skew Edge Multicut instance $(G', (s_1, t_1), \ldots, (s_{|W|}, t_{|W|}), k)$. The following two claims establish a connection between the solutions of this instance and our problem.

Claim 8.44. *If there is a set $S \subseteq E(G)$ of size at most k such that $G \setminus S$ has a topological ordering inducing the order $w_1, \ldots, w_{|W|}$ on W, then the* Skew Edge Multicut *instance has a solution.*

Proof. For every edge $e \in S$, there is a corresponding edge e' of G'; let S' be the set corresponding to S. Suppose that $G' \setminus S$ has a directed $s_i \to t_j$ path P for some $i \geq j$. Path P may go through several edges (t_r, s_r), but it

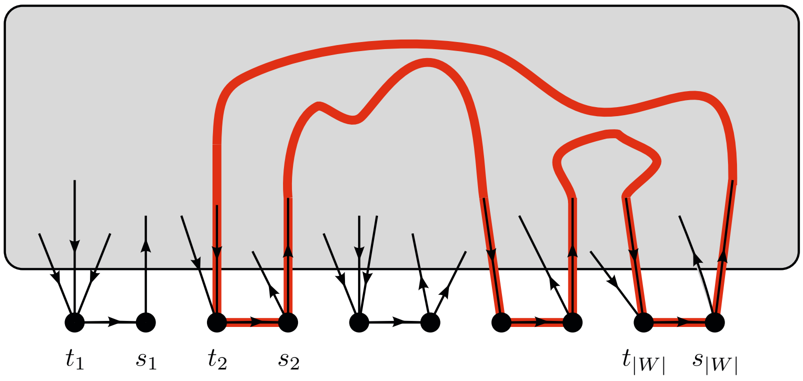


Fig. 8.9: Solving Directed Feedback Arc Set Compression in the proof of Lemma 8.43. Every directed cycle in G' has an $s_i \to t_j$ subpath for some $i \geq j$

has an $s_{i'} \to t_{j'}$ subpath P' such that $i' \geq j'$ and P' has no internal vertex of the form t_r, s_r. If $i' = j'$, then the edges of G corresponding to P' give directed cycle C in G disjoint from S, a contradiction. If $i' > j'$, then there is a directed $w_{i'} \to w_{j'}$ path P of G corresponding to P' that is disjoint from S, contradicting the assumption that $w_{i'}$ is later than $w_{j'}$ in the topological ordering of $G \setminus S$. ⌟

Claim 8.45. *Given a solution S' of the* Skew Edge Multicut *instance, one can find a feedback arc set S of size at most k for G.*

Proof. Suppose that the Skew Edge Multicut instance has a solution S' of size at most k. For every edge of $E(G') \setminus E_W$, there is a corresponding edge of G; let S be the set of at most k edges corresponding to $S' \setminus E_W$. Suppose that $G \setminus S$ has a directed cycle C. This cycle C has to go through at least one vertex of W. If C goes through exactly one vertex w_i, then the edges of G' corresponding to C give a directed $t_i \to s_i$ path disjoint from S', a contradiction (note that this path is certainly disjoint from E_W). If C goes through more than one w_i, then C has a directed $w_i \to w_j$ subpath for some $i > j$ such that the internal vertices of P are not in W. Then the edges of G' corresponding to P give a directed $s_i \to t_j$ path in G' disjoint from S' (and from E_W), again a contradiction. ⌟

Suppose that there is a feedback arc set $S \subseteq E(G)$ of size at most k. A topological ordering of $G \setminus S$ induces an ordering on W and, as the algorithm tries every ordering of W, eventually this ordering is reached. At this point, Claim 8.44 implies that the constructed Skew Edge Multicut instance has a solution S' and we can find such a solution using the algorithm of Theorem 8.41. Then Claim 8.45 allows us to find a solution of Directed

Feedback Arc Set Compression. The algorithm tries $|W|!$ orderings of W and uses the $\mathcal{O}(4^k \cdot k^3 \cdot (n+m))$-time algorithm of Theorem 8.41, every other step can be done in linear time. $\square$

Finally, let us see how iterative compression can be used to solve Directed Feedback Arc Set using Directed Feedback Arc Set Compression.

Theorem 8.46. *There is an $\mathcal{O}((k+1)! \cdot 4^k \cdot k^3 \cdot n(n+m))$-time algorithm for* Directed Feedback Arc Set *on graphs with n vertices and m edges.*

Proof. Let $v_1, \ldots, v_n$ be an arbitrary ordering of the vertices of G and let $G_i = G[\{v_1, \ldots, v_i\}]$. For $i = 1, 2, \ldots, n$, we compute a feedback arc set S_i of size at most k for G_i. For $i = 1$, $S_i = \emptyset$ is a trivial solution. Suppose now that we have already computed S_i; then S_{i+1} can be computed as follows. Let $W_i \subseteq V(G_i)$ be the head of each edge in S_i. Clearly, $|W_i| \le k$ and $G \setminus W_i$ is acyclic (since $G \setminus S_i$ is acyclic). Moreover, $W_i \cup \{v_{i+1}\}$ has size at most $k+1$ and $G_{i+1} \setminus (W_i \cup \{v_{i+1}\}) = G_i \setminus W_i$, hence it is also acyclic. Therefore, we may invoke the algorithm of Lemma 8.43 to find a feedback arc set S_{i+1} of size at most k for G_{i+1}. If the algorithm answers that there is no feedback arc set of size at most k for G_{i+1}, then there is no such feedback arc set for the supergraph G of G_{i+1} and hence we can return a negative answer for Directed Feedback Arc Set on (G, k). Otherwise, if the algorithm returns a feedback arc set S_{i+1} for G_{i+1}, then we can continue the iteration with $i + 1$.

As $|W_i| \le k + 1$ in each call of the Directed Feedback Arc Set Compression algorithm of Lemma 8.43, its running time is $\mathcal{O}((k+1)! \cdot 4^k \cdot k^3 \cdot (n+m))$. Taking into account that we call this algorithm at most n times, the claimed running time follows. $\square$

Using the simple reduction from the vertex version to the edge version (Proposition 8.42), we obtain the fixed-parameter tractability of Directed Feedback Vertex Set.

Corollary 8.47. *There is an $\mathcal{O}((k+1)! \cdot 4^k \cdot k^3 \cdot n(n+m))$-time algorithm for* Directed Feedback Vertex Set *on graphs with n vertices and m edges.*

8.7 Vertex-deletion problems

All the problems treated so far in the chapter were defined in terms of removing edges of a graph (with the exception of Directed Feedback Vertex Set, which was quickly reduced to Directed Feedback Arc Set). In this section, we introduce the definitions needed for vertex-removal problems and state (without proofs) the analogues of the edge-results we have seen in the previous sections.

A technical issue specific to vertex-removal problems is that we have to define whether a vertex set separating X and Y is allowed to contain a vertex of $X \cup Y$. If so, then X and Y need not be disjoint (but then the separator has to contain $X \cap Y$). In a similar way, we have to define whether, say, Vertex Multiway Cut allows the deletion of the terminals. Note that in an edge-deletion problem we can typically make an edge "undeletable" by replacing it with $k+1$ parallel edges. Similarly, one may try to make a vertex undeletable by replacing it with a clique of size $k+1$ (with the same neighborhood), but this can be notationally inconvenient, especially when terminals are present in the graph. Therefore, we state the definitions and the results in a way that we assume that the (directed or undirected) graph G is equipped with a set $V^\infty(G) \subseteq V(G)$ of *undeletable vertices.* Given two (not necessarily disjoint) sets $X, Y \subseteq V(G)$ of vertices, an *(X,Y)-separator* is a set $S \subseteq V(G) \setminus V^\infty(G)$ of vertices such that $G \setminus S$ has no $(X \setminus S) - (Y \setminus S)$ path. We define minimal and minimum (X,Y)-separators the usual way. We can characterize minimal (X,Y)-separators as the neighborhood of a set of vertices.

Proposition 8.48. *If S is a minimal (X,Y)-separator in G, then $S = N_G(R)$ (or $S = N_G^+(R)$ in directed graphs), where R is the set of vertices reachable from $X \setminus S$ in $G \setminus S$.*

The results of Section 8.1 can be adapted to (X,Y)-separators; we omit the details. Important (X,Y)-separators are defined analogously to important (X,Y)-cuts.

Definition 8.49 (Important separator). Let G be a directed or undirected graph and let $X, Y \subseteq V(G)$ be two sets of vertices. Let $S \subseteq V(G) \setminus V^\infty(G)$ be an (X,Y)-separator and let R be the set of vertices reachable from $X \setminus S$ in $G \setminus S$. We say that S is an *important (X,Y)-separator* if it is inclusion-wise minimal and there is no (X,Y)-separator $S' \subseteq V(G) \setminus V^\infty(G)$ with $|S'| \le |S|$ such that $R \subset R'$, where R' is the set of vertices reachable from $X \setminus S'$ in $G \setminus S'$.

The analogue of Proposition 8.7 holds for (X,Y)-separators.

Proposition 8.50. *Let G be a directed or undirected graph and $X, Y \subseteq V(G)$ be two sets of vertices. Let $S \subseteq V(G) \setminus V^\infty(G)$ be an (X,Y)-separator and let R be the set of vertices reachable from $X \setminus S$ in $G \setminus S$. Then there is an important (X,Y)-separator $S' = N_G(R')$ (or $S' = N_G^+(R')$ if G is directed) such that $|S'| \le |S|$ and $R \subseteq R'$.*

The algorithm for enumerating important (X,Y)-cuts and the combinatorial bounds on their number can be adapted to important (X,Y)-separators.

Theorem 8.51. *Let $X, Y \subseteq V(G)$ be two sets of vertices in a (directed or undirected) graph G with n vertices and m edges, let $k \ge 0$ be an integer, and let $\mathcal{S}_k$ be the set of all (X,Y)-important separators of size at most k. Then $|\mathcal{S}_k| \le 4^k$ and $\mathcal{S}_k$ can be constructed in time $\mathcal{O}(|\mathcal{S}_k| \cdot k^2 \cdot (n+m))$.*

Lemma 8.52. *Let G be a (directed or undirected) graph and let $X, Y \subseteq V(G)$ be two sets of vertices. If $\mathcal{S}$ is the set of all important (X,Y)-separators, then $\sum_{S \in \mathcal{S}} 4^{-|S|} \leq 1$.*

Let G be a graph and $T \subseteq V(G)$ be a set of terminals. A *vertex multiway cut* is a set $S \subseteq V(G) \setminus V^{\infty}(G)$ of vertices such that every component of $G \setminus S$ contains at most one vertex of $T \setminus S$. Given an undirected graph G, an integer k, and a set $T \subseteq V(G)$ of terminals, the VERTEX MULTIWAY CUT problem asks for vertex multiway cut of size at most k. The analogue of the pushing lemma for EDGE MULTIWAY CUT (Lemma 8.18) can be adapted to the vertex-deletion case.

Lemma 8.53 (Pushing lemma for VERTEX MULTIWAY CUT**).** *Let $t \in T$ be a terminal in an undirected graph G. If G has a vertex multiway cut $S \subseteq V(G) \setminus V^{\infty}(G)$, then it also has a vertex multiway cut $S^* \subseteq V(G) \setminus V^{\infty}(G)$ with $|S^*| \leq |S|$ such that S^* contains an important $(t, T \setminus t)$-separator.*

Theorem 8.51, Lemma 8.52, and Lemma 8.53 allow us to solve VERTEX MULTIWAY CUT using a branching strategy identical to the one used in Theorem 8.19.

Theorem 8.54. VERTEX MULTIWAY CUT *on a graph with n vertices and m edges can be solved in time $\mathcal{O}(4^k \cdot k^3 \cdot (n+m))$.*

The result of Theorem 8.22 was actually proved in the literature for the (more general) vertex version, hence VERTEX MULTICUT is also fixed-parameter tractable parameterized by k.

Theorem 8.55 ([59, 357]). VERTEX MULTICUT *on an n-vertex graph can be solved in time $2^{\mathcal{O}(k^3)} \cdot n^{\mathcal{O}(1)}$.*

Exercises

8.1. Given an undirected graph G and $S \subseteq V(G)$, let $i_G(S)$ be number of edges induced by S (that is, $i_G(S) = |E(G[S])|$). Is i_G submodular?

8.2. Prove that a set function $f : 2^{|V(G)|} \to \mathbb{R}$ is submodular if and only if

$$f(A \cup \{v\}) - f(A) \geq f(B \cup \{v\}) - f(B) \tag{8.3}$$

holds for every $A \subseteq B \subseteq V(G)$ and $v \in V(G)$. Informally, inequality (8.3) says that the marginal value of v with respect to the superset B (that is, the increase of value if we extend B with v) cannot be larger than with respect to a subset A.

8.3. Let $f : 2^{|V(G)|} \to \mathbb{R}$ be a submodular function that is symmetric: $f(X) = f(V(G) \setminus X)$ for every $X \subseteq V(G)$. Show that f is posimodular:

$$f(A) + f(B) \geq f(A \setminus B) + f(B \setminus A) \tag{8.4}$$

holds for every $A, B \subseteq V(G)$.

8.4. Give an example of a graph G and sets $A, B \subseteq V(G)$ for which the submodularity inequality is sharp.

8.5. Give a polynomial-time algorithm for finding the sets $R_{\min}$ and $R_{\max}$ defined in Theorem 8.4 using only the facts that such sets exists and that a minimum (X,Y)-cut can be found in polynomial time.

8.6. Let G be an undirected graph and let $A, B \subseteq V(G)$ be two disjoint sets of vertices. Let $\Delta(R^{AB}_{\min})$ and $\Delta(R^{AB}_{\max})$ be the minimum (A,B)-cuts defined by Theorem 8.4, and let $\Delta(R^{BA}_{\min})$ and $\Delta(R^{BA}_{\max})$ be the minimum (B,A)-cuts defined by Theorem 8.4 (reversing the role of A and B). Show that $R^{AB}_{\min} = V(G) \setminus R^{BA}_{\max}$ and $R^{AB}_{\max} = V(G) \setminus R^{BA}_{\min}$.

8.7. Prove Proposition 8.9.

8.8. Give an example where the algorithm of Theorem 8.11 finds an (X,Y)-cut that is not an important (X,Y)-cut (and hence throws it away in the filtering phase). Can such a cut come from the first recursive branch of the algorithm? Can it come from the second?

8.9. Let G be an undirected graph, let $X, Y \subseteq V(G)$ be two disjoint sets of vertices, and let λ be the minimum (X,Y)-cut size. Let $\mathcal{S}$ be the set of all important (X,Y)-cuts. Prove that $\sum_{S \in \mathcal{S}} 4^{-|S|} \leq 2^{-\lambda}$ holds.

8.10. Prove Theorem 8.16.

8.11. Show that Edge Multiway Cut is polynomial-time solvable on trees.

8.12 (☠). Show that Edge Multicut is NP-hard on trees.

8.13. Give a $2^k \cdot n^{\mathcal{O}(1)}$-time algorithm for Edge Multicut on trees.

8.14. Reduce Directed Edge Multicut with $\ell = 2$ to Directed Edge Multiway Cut with $|T| = 2$.

8.15. Reduce Edge Multiway Cut with $|T| = 3$ to Directed Edge Multiway Cut with $|T| = 2$.

8.16. Reduce Directed Feedback Arc Set to Directed Feedback Vertex Set in polynomial time.

8.17. Show that Vertex Multicut is polynomial-time solvable on trees.

8.18 (☠). In the Digraph Pair Cut, the input consists of a directed graph G, a designated vertex $s \in V(G)$, a family of pairs of vertices $\mathcal{F} \subseteq \binom{V(G)}{2}$, and an integer k; the goal is to find a set X of at most k edges of G, such that for each pair $\{u,v\} \in \mathcal{F}$, either u or v is not reachable from s in the graph $G - X$. Show an algorithm solving Digraph Pair Cut in time $2^k n^{\mathcal{O}(1)}$ for an n-vertex graph G.

8.19. Consider the following vertex-deletion variant of Digraph Pair Cut with a sink: given a directed graph G, designated vertices $s, t \in V(G)$, a family of pairs of vertices $\mathcal{F} \subseteq \binom{V(G)}{2}$, and an integer k, check if there exists an (s,t)-separator $X \subseteq V(G) \setminus \{s,t\}$ of size at most k such that every pair in $\mathcal{F}$ either contains a vertex of X or contains a vertex that is unreachable from s in $G - X$.

Show how to solve this vertex-deletion variant in time $2^k n^{\mathcal{O}(1)}$ for an n-vertex graph G, using the algorithm of the previous exercise.

Hints

8.1 The function i_G is not submodular, but *supermodular*: the left-hand side of (8.1) is always at most the right-hand side.

8.2 To see that (8.3) is a necessary condition for f being submodular, observe that it is the rearrangement of (8.1) applied for $A \cup \{v\}$ and B. To see that it is a necessary condition, let us build $X \cup Y$ from $X \cap Y$ by first adding the elements of $X \setminus Y$ one by one and then adding the elements of $Y \setminus X$ one by one. Let us compare the total marginal increase of adding these elements to the total marginal increase of building X from $X \cap Y$, plus the total marginal increase of building Y from $X \cap Y$. The submodularity follows by observing that the marginal increase of adding elements of $Y \setminus X$ cannot be larger if $X \setminus Y$ is already present in the set.

8.3 By submodularity and symmetry, we have

$$\begin{aligned} f(A) + f(B) &= f(A) + f(V(G) \setminus B) \\ &\geq f(A \cap (V(G) \setminus B)) + f(A \cup (V(G) \setminus B)) \\ &= f(A \setminus B) + f(V(G) \setminus (B \setminus A)) \\ &= f(A \setminus B) + f(B \setminus A). \end{aligned}$$

8.4 Analyze the proof of (8.1): which edges are counted different number of times in the sides of this inequality? Any example with an edge between $A \setminus B$ and $B \setminus A$ will do the job.

8.6 If $\Delta(R)$ is a minimum (A, B)-cut, then $\Delta(R) = \Delta(V(G) \setminus R)$ and it is also a minimum (B, A)-cut. The equalities then follow simply from the fact that minimizing R is the same as maximizing $V(G) \setminus R$.

8.8 It is possible that the first branch gives an important (X, Y)-cut S in $G \setminus xy$ such that $S \cup \{xy\}$ is not an important (X, Y)-cut in G. For example, let $X = \{x\}$, $Y = \{y\}$, and suppose that graph G has the edges xa, xb, ab, bc, ay, by, cy. Then $R_{\max} = \{x\}$. In graph $G \setminus xb$, the set $\Delta_{G \setminus xb}(\{x, a\}) = \{ab, ay\}$ is an important (X, Y)-cut. However, $\Delta_G(\{x, a\}) = \{ay, ab, xb\}$ is not an important (X, Y)-cut in G, as $\Delta_G(\{x, a, b, c\}) = \{ay, by, cy\}$ has the same size.

One can show that every (X, Y)-cut returned by the second branch is an important (X, Y)-cut in G.

8.9 We need to show that in each of the two branches of the algorithm in Theorem 8.11, the total contributions of the enumerated separators to the sum is at most $2^{-\lambda}/2$. In the first branch, this is true because we augment each cut with a new edge. In the second branch, this is true because λ strictly increases.

8.10 The proof has the following main arguments:

1. The enumeration algorithm should be of the form Enumerate(G, X, Y, Z, k), which, assuming $Z \subseteq \Delta_G(X)$, returns every important (X, Y)-cut of size at most k containing Z.
2. As in the proof of Theorem 8.11, we compute $R_{\max}$ in $G \setminus Z$ and branch on an edge $xy \in \Delta_G(X)$.
3. In the first branch, we include xy into Z. We recurse only if no edge of Z has both endpoints in $R_{\max}$ and the algorithm of Lemma 8.15 predicts that Enumerate$(G, R_{\max}, Y, Z \cup \{xy\}, k)$ returns at least one important (X, Y)-cut.
4. In the second branch, we check if no edge of Z has both endpoints in $R_{\max} \cup \{y\}$ and recurse if Lemma 8.15 predicts that Enumerate$(G, R_{\max} \cup \{y\}, Y, Z, k)$ returns at least one important (X, Y)-cut.

5. Observe that now every recursive call returns at least one important (X, Y)-cut, hence the number of leaves of the recursion tree can be bounded from above by the number of important (X, Y)-cuts returned.

8.11 Solution 1: Dynamic programming. Assume that the tree is rooted and let T_v be the subtree rooted at v. Let A_v be the minimum cost of separating the terminals in T_v, and let B_v be the minimum cost with the additional constraint that v is separated from every terminal in T_v. Give a recurrence relation for computing A_v and B_v if these values are known for every child of v.

Solution 2: We may assume that every leaf of the tree contains a terminal, otherwise the leaf can be removed without changing the problem. Select any leaf and argue that there is a solution containing the edge incident to this leaf.

8.12 Reduction from VERTEX COVER: the graph is a star, edges play the role of vertices, and the terminal pairs play the role of edges.

8.13 Assume that the tree is rooted and select a pair (s_i, t_i) such that the topmost vertex v of the unique $s_i - t_i$ path P has maximum distance from the root. Argue that there is a solution containing an edge of P incident to v.

8.14 Introduce two terminals v_1, v_2, and add the edges (v_1, s_1), (t_1, v_2), (v_2, s_2), (t_2, v_1).

8.16 Subdivide each edge and replace each original vertex with an independent set of size $k + 1$.

8.17 Assume that the tree is rooted and select a pair (s_i, t_i) such that the topmost vertex v of the unique $s_i - t_i$ path P has maximum distance from the root. Argue that there is a solution containing vertex v.

8.18 The main observation is the following: if we choose a set of vertices $F \subseteq \bigcup \mathcal{F}$ such that no vertex of v is reachable from s in the graph $G - X$ (for a solution X we are looking for), then we may greedily choose an (s, F)-cut of minimum possible size that minimizes the set of vertices reachable from s.

In other words, we consider the following algorithm. Start with $F = \emptyset$, and repeatedly perform the following procedure. Compute a minimum (s, F)-cut $X = \Delta^+(R)$ that minimizes the set R. If X is a solution, then report X. If $|X| > k$, then report that there is no solution in this branch. Otherwise, pick a pair $\{u, v\} \in \mathcal{F}$ such that both u and v are reachable from s in $G - X$, and branch: include either u or v into F. Prove that the size of X strictly increases at each step, yielding the promised bound on the size of the search tree.

8.19 Perform the standard reduction from directed edge-deletion problems to vertex-deletion ones, by going to the (appropriately adjusted) line graph of the input graph. To handle the sink t, replace t with its $k + 2$ copies $t_1, t_2, \ldots, t_{k+2}$, and add a pair $\{t_i, t_j\}$ to $\mathcal{F}$ for every $1 \leq i < j \leq k + 2$.

Bibliographic notes

There is a large body of literature on polynomial-time algorithms for minimum cut problems, maximum flow problems, and their relation to submodularity. The monumental work of Schrijver [412] and the recent monograph of Frank [214] give detailed overviews of the subject. The Ford-Fulkerson algorithm for finding maximum flows (which is the only maximum flow algorithm we needed in this chapter) was published in 1956 [212]. For other, more efficient, algorithms for finding maximum flows, see any standard algorithm textbook such as Cormen, Leiserson, Rivest, Stein [97].

Marx [347] defined the notion of important separators and proved the fixed-parameter tractability of Vertex Multiway Cut using the branching approach of Theorem 8.19. The 4^k bound and the proof of Theorem 8.11 is implicit in the work of Chen, Liu, and Lu [84]. Using different approaches, one can obtain better than $4^k \cdot n^{\mathcal{O}(1)}$ time algorithms: Xiao [439] gave a $2^k \cdot n^{\mathcal{O}(1)}$-time algorithm for Edge Multiway Cut; Cao, Chen, and Fan gave a $1.84^k \cdot n^{\mathcal{O}(1)}$-time algorithm for Edge Multiway Cut; and Cygan, Pilipczuk, Pilipczuk, and Wojtaszczyk [122] gave a $2^k \cdot n^{\mathcal{O}(1)}$ time algorithm for Vertex Multiway Cut.

The NP-hardness of Edge Multicut on trees was observed by Garg, Vazirani, and Yannakakis [225]. The $2^k \cdot n^{\mathcal{O}(1)}$ algorithm of Exercise 8.13 is by Guo and Niedermeier [242]. The polynomial-time algorithm for solving Vertex Multicut on trees (Exercise 8.17) was described first by Calinescu, Fernandes, and Reed [69].

The NP-hardness of Edge Multiway Cut with $|T| = 3$ was proved by Dalhaus, Johnson, Papadimitriou, Seymour, and Yannakakis [124]. Interestingly, for a fixed number ℓ of terminals, they also show that the problem can be solved in time $n^{\mathcal{O}(\ell)}$ time on planar graphs. The running time for planar graphs was improved to $2^{\mathcal{O}(\ell)} \cdot n^{\mathcal{O}(\sqrt{\ell})}$ by Klein and Marx [295].

The fixed-parameter tractability of Edge Multicut and Vertex Multicut parameterized by the solution size k was proved independently by Bousquet, Daligault, and Thomassé [59] and Marx and Razgon [357]. The paper of Marx and Razgon introduced the random sampling of important separators technique. This technique was used then by Lokshtanov and Marx [324] to prove the fixed-parameter tractability of (p, q)-Cluster parameterized by q. Lokshtanov and Marx proved the problem is FPT also with parameter p, and studied other variants of the problem, such as when the requirement $|C| \leq p$ is replaced by requiring that the graph induced by C have at most p nonedges. This variant was recently used to analyze modular structure of large networks in nature [60].

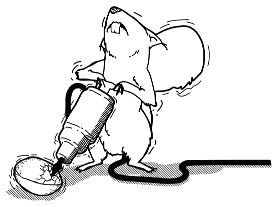
Uncrossing (variants of) submodular functions in different ways is a standard technique of combinatorial optimization. The variant described in Section 8.4, using posimodularity to uncross sets by replacing them with subsets, was used by, for example, Chekuri and Ene for submodular partition and multiway cut problems [75, 74].

The random sampling of important separators technique was used by Lokshtanov and Ramanujan [330] to solve Parity Multiway Cut and by Chitnis, Egri, and Marx [90] to solve a certain list homomorphism problem with vertex removals. The technique was generalized to directed graphs by Chitnis, Hajiaghayi, and Marx [91] to show the fixed-parameter tractability of Directed Edge Multiway Cut and Directed Vertex Multiway Cut parameterized by the size of the solution. The directed version was used by Kratsch, Pilipczuk, Pilipczuk, and Wahlström [309] to prove that Edge Multicut is FPT parameterized by k and ℓ on directed acyclic graphs and by Chitnis, Cygan, Hajiaghayi, and Marx [89] to prove that Directed Subset Feedback Vertex Set is FPT.

The fixed-parameter tractability of Skew Edge Multicut, Directed Feedback Vertex Set, and Directed Feedback Arc Set were shown by Chen, Liu, Lu, O'Sullivan, and Razgon [85].

Chapter 9
Advanced kernelization algorithms

The systematic study of the kernelization framework, whose foretaste we had in Chapter 2, revealed an intrinsic mathematical richness of this notion. In particular, many classic techniques turn out to be very useful in this context; examples include tools of combinatorial optimization, linear algebra, probabilistic arguments, or results of the graph minors theory. In this chapter, we provide an overview of some of the most interesting examples of more advanced kernelization algorithms. In particular, we provide a quadratic kernel for the FEEDBACK VERTEX SET *problem. We also discuss the topics of* above guarantee parameterizations *in the context of kernelization, of kernelization on planar graphs, and of so-called* Turing kernelization.



Recall that a kernelization algorithm, given an instance (x, k) of some parameterized problem, runs in polynomial time and outputs an equivalent instance (x', k') of the same problem such that $|x'|, k' \leq f(k)$, for some function f. This function f is called *the size of the kernel.* The smaller the kernel is, the better; that is, we would like to have $f(k)$ as slowly-growing as possible. Usually, the goal is to obtain f that is polynomial or even linear.

As discussed in Chapter 2, from the practical point of view, parameterized complexity gives us a framework to rigorously analyze various preprocessing algorithms, present in the literature for decades. From the theoretical point of view, studying kernelization has two main objectives. First, the existence of a kernel for a decidable parameterized problem is equivalent to its fixed-parameter tractability, and kernelization can be seen as one of the ways to prove that a studied problem is FPT. For instance, for the EDGE CLIQUE COVER problem of Chapter 2 (Section 2.2) the simple kernelization algorithm is in fact the only known approach from the parameterized perspective. Second, the pursuit of as small a kernel size as possible — in particular, the study of which problems admit polynomial kernels, and which (most likely) do not — leads to a better understanding and deeper insight into the class of FPT problems. Whereas Chapter 2 and this chapter are devoted to proving positive results, we give the methodology for proving lower bounds for kernelization in Chapter 15.

We start this chapter with a beautiful example of how classic tools of advanced matching theory can be used to obtain a kernel for the Feedback Vertex Set problem. We perform a deep analysis of the role of high-degree vertices in a Feedback Vertex Set instance, obtaining a reduction that bounds the maximum degree of the input graph linearly in the parameter. This implies an $\mathcal{O}(k^2)$ bound on the number of vertices and edges in a yes-instance, yielding the kernel bound. As we will see in Chapter 15, under plausible complexity assumptions, this kernel is essentially tight with respect to the bitsize, i.e., the number of bits in the binary representation.

In Section *9.2, we discuss the topic of *above guarantee parameterizations* in the context of kernelization, on the example of Max-Er-SAT. In Chapter 3, we have seen an example of above guarantee parameterizations of Vertex Cover, namely Vertex Cover Above LP and Vertex Cover Above Matching. In the Max-Er-SAT problem, we are given an r-CNF formula φ, where every clause contains exactly r literals with pairwise different variables, together with an integer k, and we ask for an assignment satisfying at least k clauses. It is easy to see that a random assignment satisfies $m(1-2^{-r})$ clauses on average, where m is the number of clauses in φ. Hence, we expect k to be rather large in interesting instances, and parameterization by k does not make much sense. Instead, we study the above guarantee parameterization of $k - m(1-2^{-r})$.

For the Max-Er-SAT problem, we show how a combination of an algebraic approach and probabilistic anti-concentration inequalities leads to a polynomial kernel. The main idea can be summarized as follows: we express the input formula as a multivariate polynomial, and argue that if the polynomial has too many monomials, then some moment inequalities imply that a random assignment satisfies the required number of clauses with positive probability. Consequently, in this case we can infer that we are dealing with a yes-instance. On the other hand, if the polynomial has a bounded number of monomials, the polynomial itself is already a "concise" description of the instance at hand.

In Section 9.3, we briefly touch the tip of the iceberg: the field of kernelization in sparse graph classes. In Section 7.7, we have seen that some topological sparsity assumption on the input graph, such as planarity, greatly helps in designing FPT algorithms: for example, Dominating Set, which is unlikely to be FPT in general graphs (see Chapter 13), admits even a subexponential algorithm on planar graphs. In this chapter, we show how planarity constraints can be exploited to give a linear kernel for the Connected Vertex Cover problem in planar graphs. This should be contrasted with the fact that the same problem most probably does not admit a polynomial kernel in general graphs, as we will learn in Chapter 15 (see Exercise 15.4, point 3). A similar behavior can be observed for many other problems. Note also that in Section 7.7, we have shown a subexponential FPT algorithm for CONNECTED VERTEX COVER in planar graphs, whereas for general graphs only single-exponential algorithms are known (see Exercise 6.6).

Finally, in Section 9.4, we discuss a relaxed variant of kernelization, called *Turing kernelization*. Recall that a preprocessing algorithm aims at reducing the time-consuming computation to as small an instance as possible. In the classic kernelization framework we produce a single small equivalent instance; here, we relax the condition of having a *single* instance by allowing the algorithm to solve for free (i.e., in an oracle fashion) instances of small size. In particular, we can return a set of small instances together with a recipe on how to obtain a solution for the original instance based on solutions for the generated instances; note that such an algorithm allows us, for example, to solve the output instances in parallel on multiple machines. However, in principle we do not need to limit ourselves to such an approach: adaptive algorithms, whose next steps depend on an oracle answer to some previous query, are also allowed. The working example for this approach will be the MAX LEAF SUBTREE problem, where the task is to determine whether there exists a subtree of a given graph that has at least k leaves. As we shall see, this problem separates classic kernelization from Turing kernelization: while it admits a polynomial Turing kernel, no classic polynomial kernel can be expected under plausible complexity assumptions (see Exercise 15.4, point 2).

9.1 A quadratic kernel for FEEDBACK VERTEX SET

Recall that a *feedback vertex set* of a graph G is a set of vertices $X \subseteq V(G)$ that hits all the cycles of G; equivalently, $G \setminus X$ is a forest. In the FEEDBACK VERTEX SET problem we look for a feedback vertex set of a given graph G whose size does not exceed the given budget k, where k is the parameter.

As in Section 3.3, it is convenient to consider FEEDBACK VERTEX SET on multigraphs, where the input graph G may contain multiple edges and loops. We treat double edges and loops as cycles. We also use the convention that a loop at a vertex v contributes 2 to the degree of v.

Let us remind the reader that in Section 3.3, we introduced five reduction rules for FEEDBACK VERTEX SET (Reductions FVS.1–FVS.5) such that, after exhaustively applying these rules, either we immediately conclude that we are dealing with a no-instance, or the reduced graph G contains no loops, only single and double edges, and has minimum degree at least 3. The behavior of these rules was as follows:

- **Reduction FVS.1**: remove a vertex with a loop and decrease the budget;

- **Reduction FVS.2**: reduce multiplicity of an edge with multiplicity more than 2 to exactly 2;
- **Reduction FVS.3**: remove a vertex of degree 0 or 1;
- **Reduction FVS.4**: contract a vertex of degree 2 onto one of its neighbors;
- **Reduction FVS.5**: if $k < 0$, then conclude that we are dealing with a no-instance.

All these rules are trivially applicable in polynomial time. Thus, in the rest of this section we assume that the input graph G contains no loops, only single and double edges, and has minimum degree at least 3. Our goal is to develop further, more involved reduction rules than the one described above, aiming at a polynomial kernel for FEEDBACK VERTEX SET.

We now focus on a vertex of *large* degree in G. It seems reasonable to suspect that it should be possible to reduce such vertices: they tend to participate in many cycles in G, and hence they are likely to be included in a solution X. If this is not the case, then there is some hope that only a small part of the neighborhood of such a vertex actually plays some "significant role" in the FEEDBACK VERTEX SET problem, and consequently this neighborhood can be simplified.

To strengthen the motivation for this approach, let us observe that an upper bound on the maximum degree actually yields immediately the desired kernel.

Lemma 9.1. *If a graph G has minimum degree at least 3, maximum degree at most d, and a feedback vertex set of size at most k, then it has less than $(d+1)k$ vertices and less than $2dk$ edges.*

Proof. Let X be a feedback vertex set of G of size at most k, and let $Y = V(G) \setminus X$. Denote by F the set of edges of G with one endpoint in X and the other in Y. We estimate the size of F from two perspectives — first using the set X, and second using the set Y.

First, by the bound on the maximum degree we have that $|F| \le d|X|$. Second, since $G \setminus X = G[Y]$ is a forest, $G[Y]$ contains less than $|Y|$ edges; in other words, $G[Y]$ has average degree lower than 2. Therefore,

$$\sum_{v \in Y} d_G(v) = \sum_{v \in Y} |N(v) \cap X| + \sum_{v \in Y} |N(v) \cap Y| < |F| + 2|Y|.$$

On the other hand, the minimum degree of G is at least 3. Thus,

$$\sum_{v \in Y} d_G(v) \ge 3|Y|.$$

Consequently, we obtain $|Y| < |F| \le d|X|$ and hence

$$|V(G)| = |X| + |Y| < (d+1)|X| \le (d+1)k.$$

To estimate the number of edges of G, we again use the bound on the maximum degree: there are at most $d|X|$ edges incident to X, and all other edges of G are contained in $G[Y]$. As $G[Y]$ is a forest, it has less than $|Y|$ edges. Consequently,

$$|E(G)| < d|X| + |Y| < 2d|X| \leq 2dk.$$

□

Formally, Lemma 9.1 supports the following reduction rule.

Reduction FVS.6. If $|V(G)| \geq (d+1)k$ or $|E(G)| \geq 2dk$, where d is the maximum degree of G, then terminate the algorithm and return that (G,k) is a no-instance.

In particular, if the maximum degree of G is less than $7k$, then we have $|V(G)| < 7k^2$ and $|E(G)| < 14k^2$, or Reduction FVS.6 triggers. Therefore, in the rest of this section, we pick a vertex $v \in V(G)$ of degree $d \geq 7k$ and we aim either (a) to conclude that v needs to be included in any solution to the instance (G,k), or (b) to simplify the neighborhood of v.

What structure in G will certify that v needs to be included in any solution to (G,k)? If we think of FEEDBACK VERTEX SET as a *hitting problem* the following answer is natural: a family of $k+1$ cycles $C_1, C_2, \ldots, C_{k+1}$ in G, all passing through v, such that no two cycles C_i share any vertex except for v. We will call such a structure a *flower*, with cycles C_i being *petals* and vertex v being *the core*. (Observe that these terms are almost consistent with the notion of sunflower defined in Section 2.6: the sets $V(C_1), \ldots, V(C_{k+1})$ form a sunflower with k petals and core $\{v\}$.)

Clearly, a flower with $k+1$ petals implies that v is in every solution of size at most k, as otherwise the solution would need to contain at least one vertex of $C_i \setminus \{v\}$ for every $1 \leq i \leq k+1$. Therefore, if there is such a flower, then we can simplify the instance by deleting vertex v. If there is no such flower, then we would like to use this fact to obtain some structural information about the graph that allows further reduction. But how can we find flowers with a certain number of petals and what can we deduce from the nonexistence of flowers? It turns out that finding flowers can be formulated as a disjoint paths problem between neighbors of v, which can be reduced to maximum matching (in a not necessarily bipartite graph). Then we can use classic matching algorithms to find the maximum number of petals and, if we do not find $k+1$ petals, we can use characterization theorems for maximum matching to deduce structural information about the graph.

The following theorem of Gallai (see also Fig. 9.1 for an illustration) formulates the tool that we need; the next section is devoted to its proof. For brevity, given a set $T \subseteq V(G)$, by *a T-path* we mean a path of positive length with both endpoints in T.

Theorem 9.2 (Gallai). *Given a simple graph G, a set $T \subseteq V(G)$, and an integer s, one can in polynomial time either*

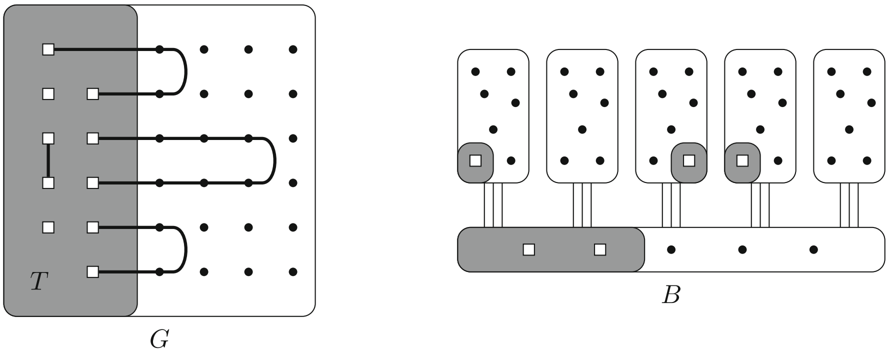


Fig. 9.1: Two possible outcomes of Gallai's theorem: either a large family of vertex-disjoint T-paths (on the left) or a small hitting set B for all T-paths (on the right). Vertices of T are depicted as squares, whereas all other vertices of G are circles

1. *find a family of $s+1$ pairwise vertex-disjoint T-paths, or*
2. *conclude that no such family exists and, moreover, find a set B of at most $2s$ vertices, such that in $G \setminus B$ no connected component contains more than one vertex of T.*

9.1.1 Proof of Gallai's theorem

The main step in the proof of Theorem 9.2 is to translate the question into the language of matching theory. Consider a graph H constructed from G by duplicating each vertex of $V(G) \setminus T$ and connecting the copies by an edge (see Fig. 9.2). Formally,

$$\begin{aligned}
V(H) &= T \cup \{x^1, x^2 \ :\ x \in V(G) \setminus T\},\\
E(H) &= E(G[T]) \cup \{ux^1, ux^2 \ :\ u \in T, x \in V(G) \setminus T, ux \in E(G)\}\\
&\qquad \cup \{x^1y^1, x^1y^2, x^2y^1, x^2y^2 \ :\ x, y \in V(G) \setminus T, xy \in E(G)\}\\
&\qquad \cup \{x^1x^2 \ :\ x \in V(G) \setminus T\}.
\end{aligned}$$

The crucial observation is the following.

Lemma 9.3. *For any nonnegative integer s, there exist s pairwise vertex-disjoint T-paths in G if and only if there exists a matching in H of size $s + |V(G) \setminus T|$.*

Proof. Let $\mathcal{P}$ be a family of pairwise vertex-disjoint T-paths in G. Without loss of generality, we may assume that the internal vertices of the

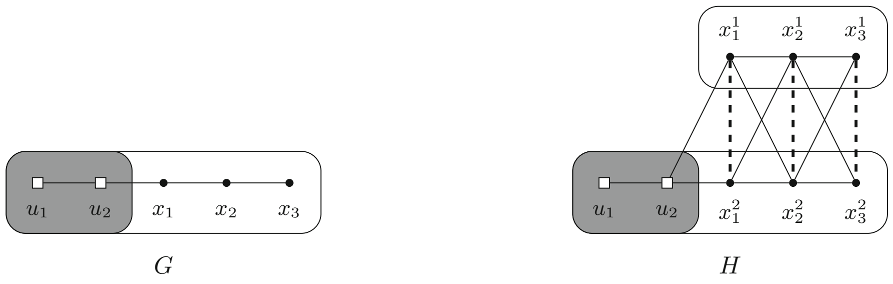


Fig. 9.2: Construction of the graph H. The matching M_0 is depicted with dashed lines

paths in $\mathcal{P}$ are disjoint from T, as otherwise such a path may be shortened. For any $P \in \mathcal{P}$, define a path P^H in H as follows: if the sequence of consecutive vertices of P equals $u, x_1, x_2, \ldots, x_{|P|-1}, w$ where $u, w \in T$, $|P|$ denotes the number of edges of P, and $x_i \notin T$ for $1 \leq i < |P|$, then let $P^H = u, x_1^1, x_1^2, x_2^1, x_2^2, \ldots, x_{|P|-1}^1, x_{|P|-1}^2, w$. Moreover, let $M_0 = \{x^1x^2 \ : \ x \in V(G) \setminus T\}$ be a matching of size $|V(G) \setminus T|$ in H. Observe that every path P^H has every second edge in the matching M_0. Let us denote $E_0(P^H) := E(P^H) \cap M_0$ to be the set of edges of M_0 on P^H, and $E_1(P^H) := E(P^H) \setminus M_0$ to be the set of remaining edges of P^H. Note that the path P^H starts and ends with an edge of $E_1(P^H)$ and, consequently, $|E_0(P^H)| + 1 = |E_1(P^H)|$. Construct a set of edges M as follows: start with $M := M_0$ and then, for every $P \in \mathcal{P}$, replace $E_0(P^H) \subseteq M_0$ with $E_1(P^H)$. (In other words, treat P^H as an augmenting path for the matching M_0.) Since the paths of $\mathcal{P}$ are pairwise vertex-disjoint, the sets $E_0(P^H)$ for $P \in \mathcal{P}$ are pairwise disjoint and, moreover, M is a matching in H. Finally, note that $|M| = |M_0| + |\mathcal{P}| = |V(G) \setminus T| + |\mathcal{P}|$.

In the other direction, let M be a matching in H of size $s + |V(G) \setminus T| = s + |M_0|$. Consider the symmetric difference $M_0 \triangle M$. Note that the graph $(V(G), M_0 \triangle M)$ has maximum degree 2, so it consists of paths and cycles only, where edges of M_0 and M appear on these paths and cycles alternately. Observe also that the only connected components of $(V(G), M_0 \triangle M)$ that contain more edges of M than of M_0 are paths that begin and end with an edge of M. Since $|M| - |M_0| = s$, we have at least s such paths, and they are pairwise vertex-disjoint. Consider one such path Q. By the definition of M_0, path Q starts and ends in a vertex of T and no internal vertex of Q lies in T. Consider a path Q^G being a "collapsed" version of Q in the graph G, defined as follows: the endpoints of Q^G are the same as those of Q, while whenever the path Q traverses an edge $x^1x^2 \in M_0$, we add the vertex $x \in V(G) \setminus T$ to the path Q^G. It is easy to see that Q^G is a T-path in G and, furthermore, if we proceed with this construction for at least s vertex-disjoint paths Q in

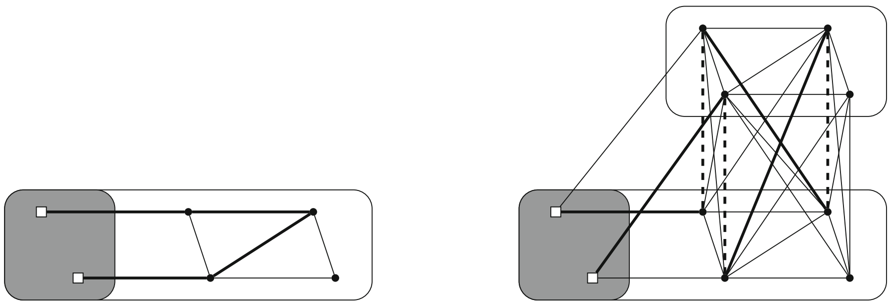


Fig. 9.3: Equivalence of a T-path in G and an augmenting path to the matching M_0 in H

H, we obtain at least s vertex-disjoint T-paths in G. This finishes the proof of the lemma (see also Fig. 9.3). $\square$

Lemma 9.3 gives us the desired connection between T-paths and matching theory. Observe that to compute the maximum number of vertex-disjoint T-paths in G, we simply need to compute a maximum matching in H; in fact, the proof of Lemma 9.3 gives us a way to efficiently transform a maximum matching in H into a family of vertex-disjoint T-paths in G of maximum possible cardinality. Consequently, Lemma 9.3 allows us to check in Theorem 9.2 whether the requested number of T-paths can be found. However, to obtain the last conclusion of Theorem 9.2 — a hitting set B for all T-paths — we need to study more deeply possible obstructions for the existence of a large number of vertex-disjoint T-paths. By Lemma 9.3, such an obstruction should, at the same time, be an obstruction to the existence of a large matching in H. Luckily, these obstructions are captured by the well-known Tutte-Berge formula.

Theorem 9.4 (Tutte-Berge formula, see [338]). *For any graph H, the following holds:*

$$|V(H)| - 2\nu(H) = \max_{S \subseteq V(H)} \mathrm{odd}(H - S) - |S|,$$

where $\nu(H)$ denotes the size of a maximum matching in H and $\mathrm{odd}(H - S)$ *denotes the number of connected components of $H - S$ with an odd number of vertices.*

Furthermore, a set S for which the maximum on the right-hand side is attained can be found in polynomial time.

We now use Theorem 9.4 to obtain the set B of Theorem 9.2, while the algorithmic statement of Theorem 9.4 will ensure that the set B is efficiently computable.

For a set $X \subseteq V(G)$, by $\mathcal{C}(X)$ we denote the family of connected components of $G - X$. In our proof we study the following quantity, defined for every $X \subseteq V(G)$:

$$\xi(X) := |X| + \sum_{C \in \mathcal{C}(X)} \left\lfloor \frac{|C \cap T|}{2} \right\rfloor .$$

Fix a set $X \subseteq V(G)$ and observe that every T-path in G either contains a vertex of X, or is completely contained in one connected component of $\mathcal{C}(X)$. In a family of pairwise vertex-disjoint T-paths there can be at most $|X|$ paths of the first type, while a component $C \in \mathcal{C}(X)$ can accommodate at most $\lfloor |C \cap T|/2 \rfloor$ pairwise vertex-disjoint T-paths since every such path has both endpoints in T. Consequently, G cannot accommodate more than $\xi(X)$ pairwise vertex-disjoint T-paths.

As an intermediate step to prove Theorem 9.2, we prove that in fact the upper bound given by $\xi(X)$ is tight:

Lemma 9.5. *The maximum possible cardinality of a family of pairwise vertex-disjoint T-paths in G equals $\min_{X \subseteq V(G)} \xi(X)$. Moreover, one can in polynomial time compute a set $W \subseteq V(G)$ satisfying $\xi(W) = \min_{X \subseteq V(G)} \xi(X)$.*

Proof. Recall that, by Lemma 9.3, the maximum possible number of pairwise vertex-disjoint T-paths in G equals $\nu(H) - |V(G) \setminus T|$, where we unravel the quantity $\nu(H)$ with Theorem 9.4.

For brevity, for every $S \subseteq V(H)$ we define the *deficiency* of S as $\mathrm{df}(S) = \mathrm{odd}(H - S) - |S|$. We start with invoking the algorithmic statement of Theorem 9.4 to compute the set S_0 of maximum deficiency in H, that is, with $\mathrm{df}(S_0) = |V(H)| - 2\nu(H)$.

Assume that, for some $x \in V(G) \setminus T$, the set S_0 contains exactly one vertex among the copies x^1, x^2 of x in H; w.l.o.g., let $x^1 \in S_0$ and $x^2 \notin S_0$. Consider a set $S_0' = S_0 \setminus \{x^1\}$. The crucial observation is that, as x^1 and x^2 are true twins in H (that is, they have exactly the same closed neighborhoods), all neighbors of x^1 in H are contained either in S_0 or in the connected component of $H - S_0$ that contains x^2. Consequently, the sets of connected components of $H - S_0$ and $H - S_0'$ differ only in one element: the connected component that contains x^2. We infer that $\mathrm{odd}(H - S_0') \geq \mathrm{odd}(H - S_0) - 1$. As $|S_0'| = |S_0| - 1$, we have $\mathrm{df}(S_0') \geq \mathrm{df}(S_0)$. By the maximality of S_0, the set S_0' is of maximum deficiency as well. Consequently, we may consider S_0' instead of S_0. By repeating this process exhaustively, we may assume that for any $x \in V(G) \setminus T$, either both x^1 and x^2 belong to S_0 or neither of them does.

Let $W^{\overline{T}} = \{x \in V(G) \setminus T \ : \ x^1, x^2 \in S_0\}$ be the projection of S_0 onto $V(G) \setminus T$, let $W^T = S_0 \cap T$, and let $W = W^{\overline{T}} \cup W^T$. Note that it suffices to prove that there exists a family of $\xi(W)$ pairwise vertex-disjoint T-paths in G; to show this claim we will strongly rely on the fact that W has been constructed using a set S_0 of maximum deficiency in H.

Recall that for any $x \in V(G) \setminus T$, either both x^1 and x^2 belong to S_0 or neither of them does. Consequently, every component of $H - S_0$ contains either both x^1 and x^2 or neither of them, which implies that a component of $H - S_0$ has odd size if and only if it contains an odd number of vertices of T. Since the connected components of $H - S_0$ are in one-to-one correspondance with the components of $\mathcal{C}(W)$, we infer that

$$\sum_{C \in \mathcal{C}(W)} \frac{|C \cap T|}{2} - \left\lfloor \frac{|C \cap T|}{2} \right\rfloor = \frac{\mathrm{odd}(H - S_0)}{2}.$$

Since $\sum_{C \in \mathcal{C}(W)} |C \cap T| = |T \setminus W|$ and the Tutte-Berge formula implies that $\mathrm{df}(S_0) = |V(H)| - 2\nu(H)$, we obtain the following:

$$\begin{aligned}
\xi(W) &= |W| + \sum_{C \in \mathcal{C}(W)} \left\lfloor \frac{|C \cap T|}{2} \right\rfloor \\
&= |W| + \frac{1}{2}\left(|T \setminus W| - \mathrm{odd}(H - S_0)\right) \\
&= |W| + \frac{1}{2}\big(|T| - |W^T| - (|S_0| + \mathrm{df}(S_0))\big) \\
&= |W| + \frac{1}{2}\Big(|T| - |W^T| - (|W^T| + 2|W^{\overline{T}}|) - (|V(H)| - 2\nu(H))\Big) \\
&= |W| + \frac{1}{2}\left(|T| - 2|W| - |V(H)| + 2\nu(H)\right) \\
&= \nu(H) - \frac{1}{2}|V(H) \setminus T| = \nu(H) - |V(G) \setminus T|.
\end{aligned}$$

This finishes the proof of the lemma. □

Recall that our goal is to construct a small set B that intersects every T-path in G. The set W of Lemma 9.5 almost meets our needs, but some connected components of $G - W$ may contain more than one vertex of T. We fix it in the simplest possible way: we start with $B := W$ computed using Lemma 9.5 and then, for each connected component C of $G - W$ that contains at least one vertex of T, we insert into B all vertices of $C \cap T$ except for one. Clearly, by construction, each connected component of $G - B$ contains at most one vertex of T. It remains to show the required upper bound on the size of B. Observe that:

$$|B| = |W| + \sum_{C \in \mathcal{C}(W)} \max(0, |C \cap T| - 1) \leq |W| + 2 \sum_{C \in \mathcal{C}(W)} \left\lfloor \frac{|C \cap T|}{2} \right\rfloor \leq 2\xi(W).$$

Consequently, by Lemma 9.5 the cardinality of B is not larger than twice the maximum cardinality of a family of pairwise vertex-disjoint T-paths in G. This concludes the proof of Theorem 9.2.

9.1.2 Detecting flowers with Gallai's theorem

We would like to apply Gallai's theorem to detect flowers with the core at some $v \in V(G)$, in order to efficiently apply the following reduction rule:

Reduction FVS.7. If there exists a vertex $v \in V(G)$ and a flower with core v and more than k petals, then delete v and decrease k by 1.

The safeness of Reduction FVS.7 is immediate. In what follows we argue how to apply Reduction FVS.7 in polynomial time; a small difficulty is that we need to be careful with multiple edges that may be present in G. Fix one vertex $v \in V(G)$. Observe that multiple edges not incident to v behave as single edges from the point of view of flowers with core v: for any such multiple edge xy, only one petal of the flower may use one of the copies of xy and it does not matter which copy is used. Moreover, any double edge incident to v can be greedily taken as a petal to the constructed flower: if vx is a double edge in G, then in any flower, at most one petal uses the vertex x and it can be replaced with the cycle consisting of both copies of vx. Consequently, define D to be the set of vertices x such that vx is a double edge in G, and let $\widehat{G}$ be the graph $G \setminus D$ with all double edges replaced with single ones. If $|D| > k$, then we can immediately trigger Reduction FVS.7 on v. Otherwise, as we have argued, the existence of a flower with $s \geq |D|$ petals and core v in G is equivalent to the existence of a flower with $s - |D|$ petals and core v in $\widehat{G}$.

Since the graph $\widehat{G}$ is simple, we can apply Theorem 9.2 to the graph $\widehat{G} \setminus \{v\}$, the set $T = N_{\widehat{G}}(v)$, and the threshold $s = k - |D|$. Clearly, T-paths in $\widehat{G} \setminus \{v\}$ are in one-to-one correspondence with cycles in $\widehat{G}$ passing through v. Hence, by Theorem 9.2, we either obtain a flower with at least $(k - |D| + 1)$ petals and core v in $\widehat{G}$, or we learn that no such flower exists and we get a set B of size at most $2k - 2|D|$ that intersects all cycles in $\widehat{G}$ passing through v. In the first case, together with $|D|$ double edges incident to v we obtain a flower with $(k + 1)$ petals, and we may trigger Reduction FVS.7 on v.

Otherwise, we not only learn that there is no flower with core v and more than k petals, but we also obtain the set B that intersects all cycles through v in $\widehat{G}$. Define $Z = B \cup D$. The following lemma summarizes the properties of Z.

Lemma 9.6. *If $v \in V(G)$ and there does not exist a flower in G with core v and more than k petals, then we can in polynomial time compute a set $Z \subseteq V(G) \setminus \{v\}$ with the following properties: Z intersects every cycle that passes through v in G, $|Z| \leq 2k$ and there are at most $2k$ edges incident to v and with second endpoint in Z.*

Proof. The bound on the number of edges between v and Z follows from the bound $|B| \leq 2k - 2|D|$ and the fact that for every vertex $u \in D$ there are exactly two edges uv in G, whereas for every $w \in B$ there is at most one edge uw. The remaining claims are immediate. □

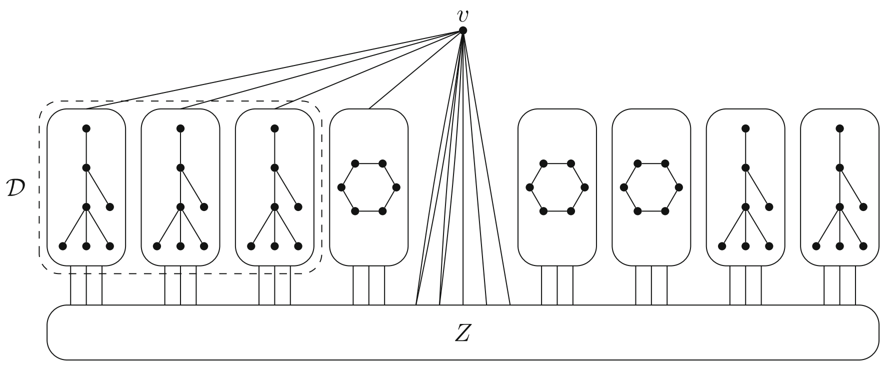


Fig. 9.4: The structure of the graph G with the set Z

9.1.3 Exploiting the blocker

In the rest of this section, we are going to use the structure given by the set Z to derive further reduction rules.

Let us first give some intuition. Assume that vertex v has degree at least ck, for some large constant c (in the proof we use $c = 7$) and assume further Reduction FVS.7 is not applicable. First, note that double edges incident to v contribute at most $2k$ to its degree. Obtain a set Z from Lemma 9.6 for v and look at the connected components of $G \setminus (\{v\} \cup Z)$; each of them is connected with at most one edge to v, and hence there are at least $(c-2)k$ of them. Moreover, if more than k of them are not trees, then we obtain $k+1$ pairwise vertex-disjoint cycles in G and we can conclude that we are dealing with a no-instance. Hence, in the remaining case at most k of the components are not trees. Consequently, we obtain at least $(c-3)k$ components that have a very simple structure: They are trees, they are connected to v by exactly one edge, and they can have some neighborhood in the set Z. On the other hand, Z is relatively small when compared to the total number of these components. The intuition is that most of these components are actually meaningless from the point of view of hitting cycles passing through v, and hence can be reduced.

For the rest of this section, we assume that we work with a vertex v with degree $d \geq 7k$. Let $\mathcal{C}$ be the family of connected components of $G \setminus (\{v\} \cup Z)$. We first observe that most of the components of $\mathcal{C}$ have to be trees, as otherwise we obtain more than k vertex-disjoint cycles in G; see also Fig. 9.4 for an illustration.

Reduction FVS.8. If more than k components of $\mathcal{C}$ contain cycles (i.e., are not trees), terminate the algorithm and return that (G, k) is a no-instance.

As Z intersects all cycles passing through v, for any $C \in \mathcal{C}$ at most one edge connects v with C. The next important observation is that many components of $\mathcal{C}$ are connected to v with *exactly* one edge.

Lemma 9.7. *There are at least $4k$ components of $\mathcal{C}$ that are trees and are connected to v with exactly one edge.*

Proof. Recall that the degree of v equals $d \geq 7k$ and that, by Lemma 9.6, there are at most $2k$ edges incident to z and a vertex of Z. Thus, there are at least $5k$ components of $\mathcal{C}$ that are connected with v with exactly one edge. If Reduction FVS.8 is inapplicable, at most k of them contain cycles. Hence, at least $4k$ of them satisfy the desired properties. $\square$

Let $\mathcal{D}$ be the family of components of $\mathcal{C}$ that satisfy the requirements of Lemma 9.7; that is, for each $A \in \mathcal{D}$ it holds that $G[A]$ is a tree and there exists exactly one edge incident to both v and a vertex of C.

We now construct an auxiliary bipartite graph H_v as follows: one bipartition class is Z, the second one is $\mathcal{D}$, and an edge connects $z \in Z$ with $A \in \mathcal{D}$ if and only if there exists an edge in G between z and some vertex in A. Since Reduction FVS.3 is inapplicable, and $G[A]$ is a tree connected to v with only one edge, for any $A \in \mathcal{D}$, some vertices of A need to be adjacent to Z in G. Consequently, no vertex of H_v in the second bipartition class (i.e., $\mathcal{D}$) is isolated. Moreover, $|Z| \leq 2k - |D|$ and $|\mathcal{D}| \geq 4k$. Hence, we may use the q-expansion lemma for $q = 2$ (Lemma 2.18) to the graph H_v, obtaining a 2-expansion in H_v. More precisely, in polynomial time we can compute a nonempty set $\widehat{Z} \subseteq Z$ and a family $\widehat{\mathcal{D}} \subseteq \mathcal{D}$ such that:

1. $N_{H_v}(\widehat{\mathcal{D}}) = \widehat{Z}$, that is, $N_G(\bigcup \widehat{\mathcal{D}}) = \widehat{Z} \cup \{v\}$;
2. each $z \in \widehat{Z}$ has two private components $A_z^1, A_z^2 \in \widehat{\mathcal{D}}$ such that $z \in N_G(A_z^1)$ and $z \in N_G(A_z^2)$. Here, by private we mean that components A_z^1, A_z^2 are all different for different $z \in \widehat{Z}$.

We claim the following.

Lemma 9.8. *For any feedback vertex set X in G that does not contain v, there exists a feedback vertex set X' in G such that $|X'| \leq |X|$ and $\widehat{Z} \subseteq X'$.*

Proof. Consider a feedback vertex set X in G with $v \notin X$. Define

$$X' = \left(X \setminus \bigcup \widehat{\mathcal{D}}\right) \cup \widehat{Z}.$$

First, we show that X' is a feedback vertex set of G. Let C be any cycle in G. If C does not intersect any component of $\widehat{\mathcal{D}}$, then the vertex of X that hits C belongs also to X', by the definition of X'. Suppose then that C intersects some $A \in \widehat{\mathcal{D}}$. Recall that $G[A]$ is a tree, $N_G(A) \subseteq \widehat{Z} \cup \{v\}$ and there is exactly one edge connecting A and v in G. Hence, C has to pass through at least one vertex of $\widehat{Z}$, which is included in X'. We conclude that in both cases C is hit by X', and so X' is a feedback vertex set of G.

We now bound the size of X'. For any $z \in \widehat{Z}$, consider a cycle C_z that starts from v, passes through A_z^1 to z and goes back to v through A_z^2. The family of cycles $\{C_z : z \in \widehat{Z}\}$ forms a flower with core v and a petal for each $z \in \widehat{Z}$. Since $v \notin X$, we infer that X needs to contain at least $|\widehat{Z}|$ vertices from $\widehat{Z} \cup \bigcup \widehat{\mathcal{D}}$. Consequently, we have $|X'| \leq |X|$, and the lemma is proved. □

Lemma 9.8 allows us to greedily choose either v or $\widehat{Z}$ into the solution feedback vertex set. If we were to design an FPT algorithm we would simply branch; however, in the case of kernelization — which is a polynomial-time algorithm — we need to encode such a behavior using some sort of gadgets. Note that we need to be careful: we cannot simply erase the components of $\widehat{\mathcal{D}}$ as, in the case of deletion of v, they may play an important role with respect to cycles passing through some vertices of $\widehat{Z}$ (but not v), and we do not properly understand this role. However, what we can do is simplify the *adjacency* between v and $\bigcup \widehat{\mathcal{D}} \cup \widehat{Z}$. Recall that, due to Lemma 9.1, it suffices if our reduction decreases the degree of a high-degree vertex v.

Reduction FVS.9. Delete all edges between v and $\bigcup \widehat{\mathcal{D}}$, and make v adjacent to each vertex of $\widehat{Z}$ with a double edge.

Let us now formally verify that Reduction FVS.9 is safe. Denote by G' the graph constructed from G by its application.

Lemma 9.9. *The minimum possible sizes of a feedback vertex set in G and in G' are equal.*

Proof. Let X be a feedback vertex set in G of minimum possible size. By Lemma 9.8, we may assume that $v \in X$ or $\widehat{Z} \subseteq X$. However, we have $G \setminus \{v\} = G' \setminus \{v\}$, whereas $G' \setminus \widehat{Z}$ is a subgraph of $G \setminus \widehat{Z}$. Hence, in both cases, $G' \setminus X$ is a subgraph of $G \setminus X$, and X is a feedback vertex set of G' as well.

In the second direction, let X be a feedback vertex set in G'. As v is connected with every vertex of $\widehat{Z}$ with double edge in G', we have $v \in X$ or $\widehat{Z} \subseteq X$. In the first case, as before, $G \setminus \{v\} = G' \setminus \{v\}$ and hence X is a feedback vertex set in G as well. Suppose then that $\widehat{Z} \subseteq X$. Then any cycle in $G \setminus X$ would need to pass through some edge of $E(G) \setminus E(G')$, that is, through an edge connecting v with some $A \in \widehat{\mathcal{D}}$. However, since $G[A]$ is a tree connected to v with only one edge, such a cycle would need to pass through some vertex of $\widehat{Z}$. This is a contradiction with $\widehat{Z} \subseteq X$, and so the lemma is proved. □

Note that a priori it is not obvious that Reduction FVS.9 actually makes some simplification of the graph, since it substitutes some set of edges with some other set of double edges. Therefore, we need to formally prove that the reduction rules cannot be applied infinitely, or superpolynomially many times. We shall do this using a *potential method*: we define a measure of the instance at hand, which (a) is never negative, (b) is initially polynomially bounded by the size of the instance, and (c) strictly decreases whenever any of the reductions is applied. For an instance (G, k), let $E_{\neg 2}(G)$ be the set of all the loops and edges of G except for edges of multiplicity 2. We define

$$\phi(G) = 2|V(G)| + |E_{\neg 2}(G)|.$$

We now claim that potential ϕ strictly decreases whenever applying some reduction rule (of course, providing that the rule did not terminate the algorithm). This holds trivially for all the reductions except for Reduction FVS.9 and Reduction FVS.4. For Reduction FVS.9, we remove a nonempty set of single edges from the graph, thus decreasing $|E_{\neg 2}(G)|$, while the introduced double edges are not counted in this summand. For Reduction FVS.4, the nontrivial situation is when the reduction contracts one copy of an edge of multiplicity 2, and thus we obtain a new loop that is counted in summand $|E_{\neg 2}(G)|$. However, the reduction also removes one vertex of the graph, and so the summand $2|V(G)|$ decreases by 2. Hence the whole potential ϕ decreases by at least 1 in this case.

As each reduction is applicable in polynomial time, the kernelization algorithm in polynomial time either terminates and concludes that we are dealing with a no-instance, or concludes that no rule is applicable. In this case the graph G has maximum degree less than $7k$, and by the inapplicability of Reduction FVS.6, we have $|V(G)| \leq 7k^2$ and $|E(G)| \leq 14k^2$. This concludes the description of the quadratic kernel for FEEDBACK VERTEX SET. We can summarize the result obtained in this section with the following theorem.

Theorem 9.10. FEEDBACK VERTEX SET *admits a kernel with at most* $7k^2$ *vertices and* $14k^2$ *edges.*

*9.2 Moments and MAX-Er-SAT

The solution size or the value of an objective function is often a natural parameterization. However, some problems are trivially FPT with such a measure, as in nontrivial instances the value we are looking for is proportional to the input size.

This is exactly the case for the MAXIMUM SATISFIABILITY problem, where given a CNF formula φ with n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$, we are to find an assignment with the maximum number of satisfied clauses. Formally, in the decision version, the instance is additionally equipped with an integer k, and we ask for an assignment ψ that satisfies at least k clauses. Observe the following: if we take any assignment ψ and its negation $\overline{\psi}$, then any clause C_i is satisfied by at least one of these assignments. Consequently, either ψ or $\overline{\psi}$ satisfies at least $m/2$ clauses of φ, and all Maximum Satisfiability instances (φ, k) with $k \leq m/2$ are yes-instances.

If you recall the main principle of parameterized complexity — do something smart when the parameter is small — then, clearly, parameterization by k does not make much sense for Maximum Satisfiability: in all interesting instances, the number of clauses is bounded by $2k$. However, one can consider also a more refined way to parameterize Maximum Satisfiability, namely the so-called above guarantee parameterization. Define $k' = k - m/2$ in a Maximum Satisfiability instance (φ, k), and consider Maximum Satisfiability parameterized by k', that is, the *excess* above the "guaranteed" solution value of at least $m/2$ satisfied clauses. Recall that in Chapter 3 we have seen an example of above guarantee parameterizations of Vertex Cover.

The study of the above guarantee parameterization of Maximum Satisfiability is deferred to Exercise 9.3, where you are asked to show its fixed-parameter tractability. In this section, we consider, as an example, a more specific variant of the Maximum Satisfiability problem, namely Max-Er-SAT. Here we assume that the input formula φ is an r-CNF formula for some positive integer r, that is, each clause of φ consists of exactly r literals. Moreover, we assume that literals in one clause pairwise involve different variables, i.e., there are no two identical literals and no literal is the negation of another one. Therefore, requiring that each clause consists of *exactly* r literals (as opposed to the more usual requirement of *at most* r literals) is a nontrivial condition: for example, we cannot just repeat one of the literals of a clause with less than r literals to increase its size to exactly r. The more "standard" version, where every clause contains *at most* r literals, is called Max-r-SAT.

For an r-CNF formula one can show a stronger lower bound on the maximum number of satisfied clauses than for a general CNF formula.

Lemma 9.11. *For an r-CNF formula φ with m clauses, there exists an assignment satisfying at least $m(1 - 2^{-r})$ clauses.*

Proof. Consider a random assignment ψ, where each variable x_i is independently at random assigned true or false value with probability $\frac{1}{2}$ each. A clause C_i is falsified by ψ if and only if all its literals evaluate to false with respect to ψ, which happens with probability exactly 2^{-r}. By the linearity of expectation, the expected number of satisfied clauses of φ is $m(1 - 2^{-r})$, as

$$\mathrm{E}(\text{number of satisfied clauses}) = \sum_{1 \leq i \leq m} \Pr(C_i \text{ is satisfied}) = m(1 - 2^{-r})\,.$$

Therefore, there exists an assignment satisfying at least $m(1 - 2^{-r})$ clauses. □

Because of the Lemma 9.11, we know that it is always possible to satisfy at least fraction $(1 - 2^{-r})$ of all the clauses. Following the reasoning for

Maximum Satisfiability from the beginning of this section, we consider the above guarantee parameterization of the Max-Er-SAT problem, where $k' = k - m(1 - 2^{-r})$ is our new parameter. We shall show that Max-Er-SAT is FPT when parameterized by k' using the following approach.

1. We change the setting and reduce the problem to checking whether some low-degree polynomial X constructed from the input formula φ can take value at least $2^r k'$ when each variable is assigned either 1 or -1.
2. When we choose the values of variables uniformly and independently at random from $\{-1, 1\}$, then the mean value of X is 0. The crucial step is to prove that if X contains many monomials (in terms of k'), then the distribution of its values must be very "spread" around the mean. More precisely, we prove an anti-concentration inequality showing that X attains value larger or equal to $2^r k'$ with positive probability, providing that it has sufficiently many monomials.
3. Hence, we can either conclude that we are definitely working with a yes-instance, or the polynomial X has only few monomials (in terms of k'). Thus, in polynomial time we have compressed the given instance to size polynomial in k', which leads to a polynomial kernel and an FPT algorithm.

9.2.1 Algebraic representation

We are going to encode the formula φ with clauses $\mathtt{Cls}(\varphi) = \{C_1, C_2, \ldots, C_m\}$ and variables $\mathtt{Vars}(\varphi) = \{x_1, x_2, \ldots, x_n\}$ as a polynomial. For a clause C_i, we denote by $\mathtt{Vars}(C_i)$ the set of variables appearing in literals of C_i. Let us identify the Boolean value *true* with the real number 1 and the value *false* with -1. Consequently, an assignment is a function $\psi\colon \{x_1, x_2, \ldots, x_n\} \to \{-1, 1\}$. The crucial algebraic object in our considerations will be the following polynomial:

$$X(x_1, x_2, \ldots, x_n) = \sum_{1 \le i \le m} \left(1 - \prod_{x_j \in \mathtt{Vars}(C_i)} (1 + \varepsilon_{x_j, C_i} x_i) \right), \tag{9.1}$$

where $\varepsilon_{x_j, C_i} = -1$ if x_j appears in C_i positively, while $\varepsilon_{x_j, C_i} = 1$ if x_j appears in C_i negatively. Observe that if an assignment satisfies the clause C_i, then the corresponding product is 0. If we consider r to be a constant, then polynomial X can be written explicitly as a sum of monomials in linear time: expanding each of the inner products results in at most 2^r monomials, and with the additional 1 in each summand we have at most $(2^r + 1) \cdot m$ monomials in total. As we shall later see, the input instance is trivially a yes-instance, unless this number of monomials is much smaller (bounded in terms of k'); that is, if most of the summands in the expansion of formula (9.1) actually cancel out.

We now establish a relation between polynomial X and the maximum number of clauses that may be satisfied in formula φ. In what follows, for an assignment ψ, we use a shorthand $X(\psi) := X(\psi(x_1), \psi(x_2), \ldots, \psi(x_n))$.

Lemma 9.12. *For an assignment* $\psi\colon \{x_1, ..., x_n\} \to \{-1, 1\}$, *we have*

$$X(\psi) = 2^r \cdot \left(\mathrm{sat}\,(\varphi, \psi) - \left(1 - 2^{-r}\right) m\right),$$

where $\mathrm{sat}\,(\varphi, \psi)$ *denotes the number of clauses of* φ *that* ψ *satisfies.*

Proof. Observe that $\prod_{x_j \in C_i} \left(1 + \varepsilon_{x_j, C_i} x_i\right)$ equals 2^r if C is falsified and 0 otherwise. Hence,

$$\begin{aligned} X(\psi) &= m - \sum_{C_i \in \mathtt{Cls}(\varphi)} [C_i \text{ not satisfied in } \psi] \cdot 2^r \\ &= m - (m - \mathrm{sat}\,(\varphi, \psi)) \cdot 2^r \\ &= 2^r \cdot \left(\mathrm{sat}\,(\varphi, \psi) - \left(1 - 2^{-r}\right) m\right). \end{aligned}$$

□

Lemma 9.12 allows us to reformulate the problem using polynomial X: The question whether there is an assignment satisfying at least $k = m(1 - 2^{-r}) + k'$ clauses of φ is equivalent to asking whether there is an assignment $\psi : \{x_1, ..., x_n\} \to \{-1, 1\}$ for which $X(\psi) \geq 2^r k'$.

Our strategy is to show that if X has sufficiently many monomials, then for some ψ we always have $X(\psi) \geq 2^r k'$. To achieve this goal, we consider ψ as a random vector that picks the value of each variable independently and uniformly at random, and we study the first few moments of the random variable $X(\psi)$. The desired anti-concentration property will follow from standard probabilistic inequalities relating the distribution of a random variable with its moments.

9.2.2 Tools from probability theory

We are going to use the following well-known inequality.

Lemma 9.13 (Hölder's inequality, probabilistic version). *For any reals* $p, q > 1$ *satisfying* $\frac{1}{p} + \frac{1}{q} = 1$ *and any real random variables* Y *and* Z *the* *following holds:*

$$\mathrm{E}(|YZ|) \le (\mathrm{E}(|Y|^p))^{\frac{1}{p}} \cdot (\mathrm{E}(|Z|^q))^{\frac{1}{q}}\,.$$

We remark that the special case $p = q = 2$ is often called the *Cauchy-Schwarz inequality.*

Lemma 9.14 (Cauchy-Schwarz inequality, probabilistic version). *For any real random variables Y and Z the following holds:*

$$\mathrm{E}(|YZ|) \le \sqrt{\mathrm{E}(|Y|^2) \cdot \mathrm{E}(|Z|^2)}.$$

We use Hölder's inequality to relate the second and the fourth moment of a random variable.

Lemma 9.15. *Assume that X is a real random variable with finite and positive fourth moment (i.e., $0 < \mathrm{E}\big(X^4\big) < \infty$). Then*

$$\mathrm{E}(|X|) \ge \frac{\big(\mathrm{E}\big(X^2\big)\big)^{\frac{3}{2}}}{(\mathrm{E}(X^4))^{\frac{1}{2}}}.$$

Proof. Take $Y = |X|^{\frac{2}{3}}$, $Z = |X|^{\frac{4}{3}}$, $p = \frac{3}{2}$, $q = 3$. Using Hölder's inequality we obtain

$$\mathrm{E}\big(X^2\big) \le (\mathrm{E}(|X|))^{\frac{2}{3}} \cdot \big(\mathrm{E}\big(X^4\big)\big)^{\frac{1}{3}}\,.$$

The lemma follows by standard algebraic manipulations. □

The following corollary shows that the only way a zero-mean random variable X may achieve a high second moment in comparison to its fourth moment is to attain high values with positive probability.

Corollary 9.16. *Let X be a random variable and suppose that $\mathrm{E}(X) = 0$, $\mathrm{E}\big(X^2\big) = \sigma^2$, and $0 < \mathrm{E}\big(X^4\big) \le b \cdot \sigma^4$. Then*

$$\Pr\left(X \ge \frac{\sigma}{2\sqrt{b}}\right) > 0.$$

Proof. By Lemma 9.15, we have $\mathrm{E}(|X|) \ge \frac{\sigma}{\sqrt{b}}$. Since $\mathrm{E}(X) = 0$, by the law of total expectation, we have

$$\Pr(X > 0) \cdot \mathrm{E}(X|X > 0) + \Pr(X < 0) \cdot \mathrm{E}(X|X < 0) = 0 \qquad (9.2)$$

(note that we do not have to consider the contribution of the term $\Pr(X = 0) \cdot \mathrm{E}(X|X = 0)$, as it is obviously zero). Since $\mathrm{E}\big(X^4\big) > 0$, at least one of $\Pr(X > 0)$ and $\Pr(X < 0)$ has to be nonzero and if one of them is nonzero, then $\mathrm{E}(X) = 0$ implies that both of them are nonzero. By (9.2) we infer

$$\Pr(X > 0) \cdot \mathrm{E}(X|X > 0) = \Pr(X < 0) \cdot \mathrm{E}(-X|X < 0). \qquad (9.3)$$

On the other hand, from the law of total expectation applied to $|X|$ we have

$$\mathrm{E}(|X|) = \Pr(X > 0) \cdot \mathrm{E}(X|X > 0) + \Pr(X < 0) \cdot \mathrm{E}(-X|X < 0). \qquad (9.4)$$

Combining (9.3) and (9.4) we obtain

$$\Pr(X > 0) \cdot \mathrm{E}(X|X > 0) = \frac{\mathrm{E}(|X|)}{2} \geq \frac{\sigma}{2\sqrt{b}}.$$

Since $\mathrm{E}(X|X > 0) \geq \frac{\sigma}{2\sqrt{b}}$, we infer that X takes value at least $\frac{\sigma}{2\sqrt{b}}$ with positive probability. □

9.2.3 Analyzing moments of $X(\psi)$

Let $X = X(x_1, .., x_n)$ be the polynomial defined in (9.1). As already mentioned, we can represent X as a sum of monomials. More precisely, we apply exhaustively the distributivity of addition and multiplication on reals to the formula (9.1), and cancel out summands whenever possible. In this way, we obtain a representation of X as a sum of monomials $X = \sum_{I \in \mathbf{S}} X_I$, where $\mathbf{S} \subseteq \binom{[n]}{\leq r}$ and for each $I \in \mathbf{S}$ we have $X_I = c_I \cdot \prod_{i \in I} x_i$ for some $c_I \neq 0$. Observe that $X(0, 0, \ldots, 0) = 0$, so there is no constant term in X; in other words, $\emptyset \notin \mathbf{S}$. Moreover, X is a *multilinear polynomial*, that is, it is linear with respect to every variable (in the representation as a sum of monomials, in every monomial each variable appears with exponent 0 or 1). We remark here that, due to cancellation, $|\mathbf{S}|$ may be much smaller than the original number of summands in the expanded form of (9.1).

Now we apply the derived inequalities to our polynomial, but in order to treat the value of X as a random variable, we first need to formalize the probability distribution we have discussed previously. Henceforth, we treat X as a random variable defined on the domain $\{-1, 1\}^n$, where every variable x_i takes value -1 or 1 uniformly at random (i.e., with probability $\frac{1}{2}$ each), and values of all the variables are independent. As all the monomials in X have at least one variable, by the linearity of expectations and independence of variables, we have $\mathrm{E}(X) = 0$. Let $\sigma^2 = \mathrm{E}(X^2)$ be the second moment of X. Now, we give a connection between σ^2 and $|\mathbf{S}|$, that is, the number of monomials of X.

Lemma 9.17. $\sigma^2 \geq |\mathbf{S}|$.

Proof. Observe that for any $I, J \in \mathbf{S}$, $I \neq J$, we have that $\mathrm{E}(X_I X_J) = c_I c_J \cdot \mathrm{E}\Big(\prod_{i \in I \triangle J} x_i\Big) = 0$, where $I \triangle J$ is the symmetric difference of I and J. Using this, we obtain

$$\begin{aligned}\mathrm{E}(X^2) &= \mathrm{E}\left(\left(\sum_{I\in\mathbf{S}} X_I\right)\left(\sum_{I\in\mathbf{S}} X_I\right)\right) = \sum_{I\in\mathbf{S}} \mathrm{E}(X_I^2) + \sum_{I,J\in\mathbf{S},I\neq J} \mathrm{E}(X_I X_J)\\ &= \sum_{I\in\mathbf{S}} \mathrm{E}(X_I^2) = \sum_{I\in\mathbf{S}} c_I^2 \geq |\mathbf{S}|.\end{aligned}$$

The last inequality holds because every c_I is a nonzero integer. □

The remaining prerequisite of Corollary 9.16 is an upper bound on the fourth moment of X, which we prove by induction on n. For this reason, we state the following lemma in a general way.

Lemma 9.18 (Hypercontractive inequality). *Let X be a multilinear polynomial of degree at most r. Then,* $\mathrm{E}(X^4) \leq 9^r \left(\mathrm{E}(X^2)\right)^2$.

Proof. We prove the lemma by induction on n, the number of variables. In the base case assume $n = 0$, which means that X consists only of a constant term c (potentially $c = 0$). Then $\mathrm{E}(X^4) = c^4 = \left(\mathrm{E}(X^2)\right)^2$ and the inequality holds.

In the inductive step we assume that $n \geq 1$, and let us express the polynomial as $X = P(x_1, \ldots, x_{n-1}) \cdot x_n + Q(x_1 \ldots, x_{n-1})$. Observe that in this expression the values of P and Q depend only on the first $n-1$ variables, and are independent of the value of x_n. Consequently, since $\mathrm{E}(YZ) = \mathrm{E}(Y)\,\mathrm{E}(Z)$ for independent random variables Y and Z, we obtain the following:

$$\begin{aligned}\mathrm{E}(X^4) &= \mathrm{E}\left((Px_n + Q)^4\right)\\ &= \mathrm{E}\left(P^4x_n^4 + 4P^3x_n^3Q + 6P^2x_n^2Q^2 + 4Px_nQ^3 + Q^4\right)\\ &= \mathrm{E}(P^4)\underbrace{\mathrm{E}(x_n^4)}_{=1} + 4\,\mathrm{E}(P^3Q)\underbrace{\mathrm{E}(x_n^3)}_{=0} + 6\,\mathrm{E}(P^2Q^2)\underbrace{\mathrm{E}(x_n^2)}_{=1}\\ &\qquad + 4\,\mathrm{E}(PQ^3)\underbrace{\mathrm{E}(x_n)}_{=0} + \mathrm{E}(Q^4)\\ &\leq 9^{r-1}\left(\mathrm{E}(P^2)\right)^2 + 6\,\mathrm{E}(P^2Q^2) + 9^r\left(\mathrm{E}(Q^2)\right)^2 .\end{aligned}$$

The last inequality comes from the induction hypothesis applied to P and Q. Notice that P consists of monomials whose number of variables is at most $r-1$, while monomials in Q can still have exactly r variables. Using the Cauchy-Schwarz inequality, we obtain

$$6\,\mathrm{E}(P^2Q^2) \leq 6 \cdot \left(\mathrm{E}(P^4)\right)^{\frac{1}{2}}\left(\mathrm{E}(Q^4)\right)^{\frac{1}{2}} .$$

Then, by applying induction to P^4 and Q^4 on the right-hand side we get:

$$\begin{aligned}
\mathrm{E}\big(X^4\big) &\le 9^{r-1}\left(\mathrm{E}\big(P^2\big)\right)^2 + 6\cdot\left(9^{r-1}\left(\mathrm{E}\big(P^2\big)\right)^2\right)^{\frac{1}{2}}\left(9^{r}\left(\mathrm{E}\big(Q^2\big)\right)^2\right)^{\frac{1}{2}} \\
&\qquad + 9^r\left(\mathrm{E}\big(Q^2\big)\right)^2 \\
&= \underbrace{9^{r-1}}_{\le 9^r}\left(\mathrm{E}\big(P^2\big)\right)^2 + \underbrace{6\cdot 3^{2r-1}}_{=2\cdot 9^r}\mathrm{E}\big(P^2\big)\cdot\mathrm{E}\big(Q^2\big) + 9^r\left(\mathrm{E}\big(Q^2\big)\right)^2 \\
&\le 9^r\left(\mathrm{E}\big(P^2\big)+\mathrm{E}\big(Q^2\big)\right)^2 .
\end{aligned}$$

To analyze the right-hand side of the desired inequality, we use again the fact that P and Q are independent of x_n.

$$\begin{aligned}
9^r\left(\mathrm{E}\big(X^2\big)\right)^2 &= 9^r\left(\mathrm{E}\big((Px_n+Q)^2\big)\right)^2 \\
&= 9^r\left(\mathrm{E}\big(P^2x_n^2\big) + 2\,\mathrm{E}(2PQx_n) + \mathrm{E}\big(Q^2\big)\right)^2 \\
&= 9^r\left(\mathrm{E}\big(P^2\big)\underbrace{\mathrm{E}\big(x_n^2\big)}_{=1} + 2\,\mathrm{E}(PQ)\underbrace{\mathrm{E}(x_n)}_{=0} + \mathrm{E}\big(Q^2\big)\right)^2 \\
&= 9^r\left(\mathrm{E}\big(P^2\big)+\mathrm{E}\big(Q^2\big)\right)^2 .
\end{aligned}$$

This concludes the proof of the lemma. □

Having all the tools prepared, we are ready to show the main lemma of this section.

Lemma 9.19. *If $|\mathbf{S}| \ge 4\cdot 9^r\cdot 4^r\cdot k'^2$, then there exists an assignment ψ : $\{x_1,\dots,x_n\}\to\{-1,1\}$ such that $X(\phi)\ge 2^r k'$.*

Proof. Since $k'>0$, we have $\mathbf{S}\neq\emptyset$ and X is not a zero polynomial. It follows that $\mathrm{E}\big(X^4\big)>0$. Having Lemma 9.18, we can use Corollary 9.16 with $b=9^r$, obtaining

$$\Pr\Big(X\ge\frac{\sigma}{2\cdot 3^r}\Big)=\Pr\Big(X\ge\frac{\sigma}{2\sqrt{b}}\Big)>0\,.$$

However, Lemma 9.17 implies that $\sigma^2\ge|\mathbf{S}|$, whereas $|\mathbf{S}|\ge 4\cdot 9^r\cdot 4^r\cdot k'^2$. Therefore,

$$\Pr\Big(X\ge\frac{2\cdot 3^r\cdot 2^r\cdot k'}{2\cdot 3^r}\Big)>0\,.$$

We conclude that there always exists an assignment ψ such that $X(\psi)\ge 2^r\cdot k'$. □

Theorem 9.20. *The problem* Max-Er-SAT *is fixed parameter tractable when parameterized by $k'=k-m(1-2^{-r})$, and can be solved in time $\mathcal{O}(m+2^{\mathcal{O}(k'^2)})$.*

Proof. Recall that we treat r as a constant. As mentioned before, in $\mathcal{O}(m)$ time we compute the monomial representation of the polynomial X. If X

has at least $4 \cdot 9^r \cdot 4^r \cdot k'^2$ monomials, then, by Lemmas 9.19 and 9.12, we are working with a yes-instance, and we may provide a positive answer. If, however, X does not have that many monomials, then only $\mathcal{O}(k'^2)$ variables appear in the simplified form of X and we can exhaustively check all the $2^{\mathcal{O}(k'^2)}$ assignments. $\square$

Lemma 9.19 implies that either X has many monomials and we can safely provide a positive answer to the problem (by Lemma 9.12), or the number of monomials in X is bounded by $\mathcal{O}(k'^2)$. Even though we have only used polynomial time, we have not yet obtained a kernel for the following two reasons.

First, we have to bound the total instance size, not only the number of monomials. The encoding of the instance needs to include also the coefficients standing in front of these monomials, and they can be a priori as large as m, the number of clauses. However, it is easy to verify that they will never be larger than m, which means that their binary representations are of length $\mathcal{O}(\log m)$. To compress the total size of the encoding of the instance to polynomial in k', we can apply the following standard trick. Providing that $\log m \leq k'^2$, our instance is already of total size $\mathcal{O}(k'^4)$ and we are done. However, if $\log m > k'^2$, then $m > 2^{k'^2}$ and we can solve our instance in polynomial time using Theorem 9.20. In this manner, we can always compress the total bitsize, i.e., the number of bits in the binary representation, of the instance to $\mathcal{O}(k'^4)$.

Second, we have not obtained a kernel, because the polynomial X is not an instance of the Max-Er-SAT problem we were working with initially. Such a shrinking algorithm where the target problem may be different from the source problem is called a *polynomial compression*; this notion will be formally stated in Definition 15.8. Nevertheless, having a polynomial compression algorithm to a language which belongs to NP, like our problem with polynomial X, we can obtain also a standard polynomial kernel for the source problem: the only thing we need to do is apply an NP-hardness reduction that transforms the output of the compression back to an instance of Max-Er-SAT.

Theorem 9.21. *The problem* Max-Er-SAT *admits a polynomial kernel when parameterized by* $k' = k - m(1 - 2^{-r})$.

9.3 Connected Vertex Cover in planar graphs

In Section 7.7 we have seen a simple example of how the technique of bidimensionality leads to subexponential parameterized algorithms in planar graphs. This powerful framework gives not only subexponential algorithms, but also efficient kernelization procedures and approximation schemes for many problems. Moreover, the approach applies not only to planar graphs, but also to more general graph classes, such as graphs with a fixed excluded minor. In this section, we present only the tip of the iceberg in the context of kernelization: We show a kernel with very few (at most $4k$) vertices for the Connected Vertex Cover problem on planar graphs.

Recall that a *vertex cover* of a graph G is a set of vertices X that hits all edges of G, that is, such that $G \setminus X$ is an edgeless graph. A vertex cover X is a *connected vertex cover* if additionally $G[X]$ is connected. In the Connected Vertex Cover problem, we look for a connected vertex cover of a given graph G whose size does not exceed the given budget k. We consider the parameterization by k.

9.3.1 Plane graphs and Euler's formula

Recall that a *planar graph* is a graph that can be embedded into the plane or, equivalently, into the sphere, whereas a *plane graph* is a planar graph together with its one fixed embedding into the plane (or sphere). We refer the reader to Section 6.3 and to the appendix for precise definitions.

Our starting point is Euler's well-known formula.

Theorem 9.22 (Euler's formula). *In any connected nonempty plane graph G, the number of faces is exactly*

$$|E(G)| - |V(G)| + 2.$$

We remark here that Euler's formula remains true if we allow G to be a multigraph.

A simple well-known corollary of Euler's formula is the following lemma.

Lemma 9.23. *If G is a nonempty simple planar bipartite graph whose every connected component contains at least three vertices, then $|E(G)| \leq 2|V(G)| - 4$.*

Proof. First, observe that we may consider each connected component of G separately. Hence, for the rest of the proof we additionally assume that G is connected and, by the assumptions of the lemma, $|V(G)| \geq 3$.

Second, fix a planar embedding of G, and let f be the number of faces in this embedding. Since G is connected, every face in this embedding is homeomorphic to an open disc. For a face F, by the length of F we mean the length of the closed walk encircling F; note that this walk is not necessarily a simple cycle, as some cutvertices of G or bridges in G may be traversed more than once.

Since $|V(G)| \geq 3$ and G is simple, each face of G is of length at least 3. On the other hand, since G is bipartite, each face is of even length. Hence, each face is of length at least 4. Observe that if we sum up the lengths of all faces we obtain twice the number of edges of the graph G. We infer

$$4f \leq 2|E(G)|. \tag{9.5}$$

By plugging (9.5) into Euler's formula we obtain

$$|E(G)| - |V(G)| + 2 \leq |E(G)|/2\,,$$

which is equivalent to the claim of the lemma. □

9.3.2 A lemma on planar bipartite graphs

Our main tool in this section is the following simple lemma.

Lemma 9.24 (Planar neighborhood lemma). *Let G be a simple planar bipartite graph with bipartition classes A and B. Moreover, assume that each vertex in B is of degree at least 3. Then $|B| \leq 2|A|$.*

Proof. First, observe that we can assume G does not contain isolated vertices: due to the degree assumption on the side B, such vertices may appear only on the side A and only contribute to the right-hand side of the inequality in question. Second, note that the lemma is trivial if G is empty.

If G is nonempty and does not contain isolated vertices, then the degree assumption implies that every connected component of G has at least four vertices. Hence, the graph G satisfies the assumptions of Lemma 9.23, and we obtain that $|E(G)| \leq 2|V(G)| - 4$. On the other hand, note that, due to the degree bound, it holds that $|E(G)| \geq 3|B|$. Hence, $3|B| \leq 2|V(G)| - 4 = 2|A| + 2|B| - 4$, and the lemma follows. □

Lemma 9.24, although a simple corollary of Euler's formula, turns out to be extremely useful in the context of kernelization in planar graphs, through the following application.

Corollary 9.25. *Let G be a planar graph and $A \subseteq V(G)$ be an arbitrary set of vertices. Then there are at most $2|A|$ connected components of $G \setminus A$ that are adjacent to more than two vertices of A.*

Proof. Define $\mathcal{B}$ to be the family of connected components of $G \setminus A$, and create an auxiliary bipartite graph H with bipartite classes A and $\mathcal{B}$ where a vertex $a \in A$ is adjacent to a component $C \in \mathcal{B}$ if and only if there exists an edge between a and some vertex of C in G. In other words, to obtain H, we start with the graph G, contract all connected components of $G \setminus A$ into single vertices, suppressing multiple edges, and delete all edges of $G[A]$. In particular, H is a planar simple bipartite graph. Lemma 9.24 implies that at most $2|A|$ elements of $\mathcal{B}$ are adjacent to more than two vertices of A. In the language of the graph G, this is exactly the desired conclusion of the lemma. □

A typical usage of Corollary 9.25 in the context of kernelization is as follows: In the studied graph G, we have recognized some set A of small size (for example, the hypothetical solution or its approximation). Corollary 9.25 immediately provides a bound on the number of connected components of $G \setminus A$ that have more than two neighbors in A; think of such components as the ones with a *complicated* neighborhood in A. All the other components have at most two neighbors in A; such components have *simple* neighborhood in A. There is hope that some problem-specific reduction rules may be able to exploit the simplicity of this structure, and reduce the (a priori unbounded) number of components of the latter type.

9.3.3 The case of Connected Vertex Cover

We shall now see how Corollary 9.25 works in the simple example of Connected Vertex Cover. Let (G, k) be an input and assume for a moment that (G, k) is a yes-instance. Let X be a solution: a vertex cover of G such that $G[X]$ is connected and $|X| \leq k$.

Apply Corollary 9.25 to the set X in the graph G. The assumption that X is a vertex cover greatly simplifies the picture: each connected component of $G \setminus X$ is actually a single vertex. Thus, by Corollary 9.25 there are at most $2|X| \leq 2k$ vertices of $G \setminus X$ of degree at least 3. In other words, we have obtained, almost for free, a bound on the number of vertices of degree at least 3 in the graph: at most k of them reside in X, and at most $2k$ outside X. The vertices of smaller degree will turn out to be simple enough to allow us to reduce them.

We first deal with isolated vertices. Clearly, they are irrelevant to the problem.

Reduction CVC.1. Delete all isolated vertices.

Moreover, note that the requirement that the solution needs to be connected gives us the following rule.

Reduction CVC.2. If G is not connected, then conclude that we are dealing with a no-instance.

For the sake of formal correctness, we will need also the following trivial reduction rule; it will allow us to avoid some border cases in the argumentation.

Reduction CVC.3. If $k \leq 0$ or $|V(G)| \leq 3$, then terminate the algorithm and output an answer.

We now focus on vertices of degree 1. Let v be a vertex of degree 1 in G, and let u be its sole neighbor. As $|V(G)| > 3$ and G is connected, any connected vertex cover of G needs to include u: it needs to include u or v to hit the edge uv, but the only way to include v and remain connected is to include u as well. It is now useful to think of the vertex v as an "annotation": the only role of v in the instance is to force the solution to take vertex u. From this perspective, it clearly does not make sense to have multiple annotations at one vertex, i.e., multiple degree-one vertices attached to the same neighbor, since they play exactly the same role in the instance. This intuitively justifies the safeness of the following reduction; the formal proof is left to the reader as Exercise 9.4.

Reduction CVC.4. If a vertex u has multiple neighbors of degree 1, then delete all but one of them.

We now move to vertices of degree 2. Let v be such a vertex, and let u_1 and u_2 be its neighbors. There are two cases to consider, depending on whether v is a cutvertex in G or not.

Let us first focus on the case when v is a cutvertex. We claim that any connected vertex cover in G needs to contain v. Indeed, if a vertex cover X of G does not contain v, it needs to contain both u_1 and u_2. However, as v is a cutvertex, the only way to make $G[X]$ connected is to include v into X as well.

Consider now a graph G', where a vertex v with its incident edges is replaced with an edge u_1u_2; in other words, G' is constructed from G by contracting the edge vu_1 and keeping the new vertex under the name u_1.

Take any connected vertex cover X of G. By the previous argumentation, we have that $v \in X$. Moreover, since $|V(G)| > 3$ and G is connected, X cannot consist of only v and hence either u_1 or u_2 belongs to G. It now follows that $X \setminus \{v\}$ is a connected vertex cover of G' of size $|X| - 1$.

Let now X' be any connected vertex cover of G'. Then clearly $X' \cup \{v\}$ is a vertex cover of G of size $|X'| + 1$. Moreover, as $u_1u_2 \in E(G')$, we have $u_1 \in X'$ or $u_2 \in X'$, and hence $G[X]$ is connected.

Concluding, we have proved that connected vertex covers of G and of G' are in one-to-one correspondence relying on exclusion/inclusion of vertex v. This claim allows us to formulate the following reduction rule.

Reduction CVC.5. If there exists a cutvertex v of degree 2 in G, then contract one of its incident edges and decrease k by 1.

We are left with vertices of degree 2 that are not cutvertices. The following observation is now crucial.

Lemma 9.26. *Let v be a vertex of degree 2 in G that is not a cutvertex. Then there exists a connected vertex cover of G of minimum possible size that does not contain v, but contains both neighbors of v.*

Proof. Let X be a connected vertex cover of G of minimum possible size. If $v \notin X$, we have $N_G(v) \subseteq X$ and we are done, so assume otherwise. Let u_1 and u_2 be the two neighbors of v in G. As v is not a cutvertex, there exists a cycle C in G that passes through u_1, v and u_2. In particular, u_1 and u_2 are of degree at least 2.

As C contains at least three vertices, $|X| \geq 2$ and, by the connectivity of X, one of the neighbors of v needs to be contained in X. Without loss of generality assume that $u_1 \in X$.

If $u_2 \notin X$, then consider $X' = (X \setminus \{v\}) \cup \{u_2\}$. Clearly, X' is a vertex cover of G as well. Moreover, as $u_2 \notin X$, X needs to contain *both* neighbors of u_2 on the cycle C, and, consequently, $G[X']$ is connected.

Assume then that $u_2 \in X$. Clearly, $X \setminus \{v\}$ is a vertex cover of G, but may not be connected: $G[X \setminus \{v\}]$ may consist of two connected components, X_1 and X_2, where $u_1 \in X_1$ and $u_2 \in X_2$. We now focus on the cycle C. As X is a vertex cover of G, there exist three consecutive vertices x_1, y, x_2 on C such that $x_1 \in X_1$, $x_2 \in X_2$ and $y \notin X$. However, in this case $(X \setminus \{v\}) \cup \{y\}$ is a connected vertex cover of G of size $|X|$, and we are done.

Lemma 9.26 allows us again to treat a vertex of degree 2 that is not a cutvertex as an "annotation" to its neighbors: the only role it plays is to annotate that these two neighbors can be safely included into the solution. Recall that vertices of degree 1 annotate their neighbors in a similar manner, so we should be able to replace the type of annotation (see also Fig. 9.5).

Reduction CVC.6. If there exists a vertex v of degree 2 that is not a cutvertex, then delete v and add a neighbor of degree 1 to each former neighbor of v.

Again, the formal verification of safeness of Reduction CVC.6 is left to the reader as Exercise 9.5.

We emphasize that Lemma 9.26 proves safeness of Reduction CVC.6 only in the case when it is applied to only one vertex at a time. That is, it is possible that there does not exist a single connected vertex cover of minimum possible size in G that contains at once neighbors of *all* vertices of degree 2. In particular, consider a case when one of the neighbors of v, say u_1, is of degree 2 as well. Then an application of Reduction CVC.6 to v turns u_1 into a cutvertex (separating the newly created neighbor of degree 1 from the rest of G) and Reduction CVC.5 triggers on u_1.

Observe that an exhaustive application of Reductions CVC.1, CVC.5 and CVC.6 completely removes vertices of degree 0 and 2, while Reduction CVC.4 limits the number of vertices of degree 1. We may now conclude with the following.

Lemma 9.27. *If (G, k) is a yes-instance and no reduction rule is applicable to G, then $|V(G)| \leq 4k$.*

Proof. Let X be a connected vertex cover of G of size at most k. Consider the vertices of $Y = V(G) \setminus X$ and recall that Y is an independent set in G.

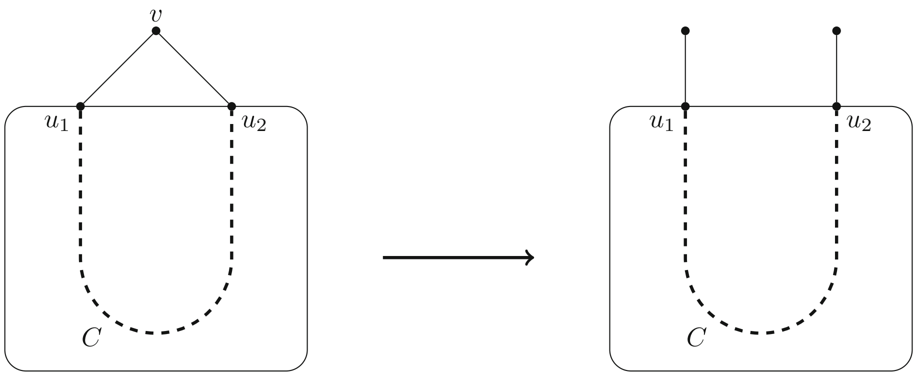


Fig. 9.5: A situation where Reduction CVC.6 is applicable and the outcome of its application

As discussed before, Corollary 9.25 implies that at most $2|X|$ vertices of Y have degree greater than 2 in G. Moreover, no vertex of G may have degree 0 or 2, while each vertex in X may have at most one neighbor in Y of degree 1. Consequently, $|Y| \leq 2|X| + |X| \leq 3k$ and $|V(G)| \leq 4k$. $\square$

Lemma 9.27 allows us to claim the final reduction rule that explicitly bounds the size of the kernel. We remark that it always triggers when k becomes negative in the process of reducing the graph.

Reduction CVC.7. If G has more than $4k$ vertices, then conclude that we are dealing with a no-instance.

Finally, we should also remark that all reduction rules are trivially applicable in polynomial time and all somehow simplify the graph: each reduction rule either terminates the algorithm, or decreases the number of edges of the graph, or maintains the number of edges of the graph while decreasing the number of vertices of degree different than 1.

9.4 Turing kernelization

As we will see in Chapter 15, for some problems it is impossible to construct polynomial kernels (under widely believed complexity assumptions). However, imagine a setting where instead of creating a single small instance, our algorithm would be allowed to create a set of small instances together with a recipe on how to compute the answer for the original instance, given the answers for all the generated small instances. One can go even further, and define an adaptive version of such a routine, called Turing kernelization.

Definition 9.28 (Turing kernelization). Let Q be a parameterized problem and let $f : \mathbb{N} \to \mathbb{N}$ be a computable function. A *Turing kernelization* for Q of size f is an algorithm that decides whether a given instance $(x, k) \in \Sigma^* \times \mathbb{N}$ is contained in Q in time polynomial in $|x| + k$, when given access to an oracle that decides membership in Q for any instance (x', k') with $|x'|, k' \leq f(k)$ in a single step.

As for the classic definition of a kernel, a Turing kernel is *polynomial* if f is a polynomial function. Obviously, regular kernels are Turing kernels as well, as it is enough to preprocess an instance in polynomial time and then use a single oracle call. Note, however, that Turing kernelization is much less restrictive than regular kernelization, as the total size of instances passed to the oracle may be even significantly larger than the input size. Furthermore, a Turing kernel can be adaptive, that is, its next steps and further queries can depend on the answer obtained from the oracle in the past.

The motivation of Turing kernelization stems from the practical viewpoint on the concept of data preprocessing. From a practitioner's viewpoint, the notion of kernelization is supposed to provide a formal framework for designing preprocessing procedures that lead to instances that are "crackable" by a brute-force algorithm. At the end of the day, the practitioner's goal is usually to solve the problem, and it is not that important whether the brute-force procedure will be run on a single instance obtained at the end of the preprocessing phase, or rather that the whole process of solving the task will require tackling several small instances along the way, reflecting different subcases. Thus, Turing kernelization expresses the practical concept of preprocessing in a better way than the classic definition of kernelization, introduced in Chapter 2. Unfortunately, this comes at a cost of having much weaker theoretical tools for analyzing this relaxed notion of a kernel.

In this section, we present an example of a problem that separates the classes of parameterized problems admitting polynomial kernels and admitting polynomial Turing kernels. In the Max Leaf Subtree problem, we are given a graph G together with an integer k and the question is whether there is a subtree T of G with at least k leaves. Recall that Exercise 6.15 asked you to show that Max Leaf Subtree admits a nonuniform FPT algorithm.

Observe that if G has several connected components, then T has to reside in a single connected component of G, as it is connected. The Max Leaf Subtree problem does not admit a polynomial kernel under widely believed complexity assumptions (see Exercise 15.4, point 2). However, intuitively, its kernelization hardness is caused by the fact that given a graph with several connected components, it is hard to identify those components which are unlikely to contain a solution. As a matter of fact, the situation changes drastically if we assume the given graph is connected.

9.4.1 *A polynomial Turing kernel for* Max Leaf Subtree

Our first step is to show that, with an additional assumption that the given graph is connected, Max Leaf Subtree admits a polynomial kernel in the classic sense. Observe that if a solution T for an instance (G, k) exists in a connected graph G, then there also exists a solution which is a spanning tree of G. This is because extending a subtree of G to a spanning tree cannot reduce the number of its leaves.

One can show that subdividing an edge of a graph that connects two vertices of degree 2 does not change the maximum number of leaves over all spanning trees of a graph. The following reduction rule is the reversal of such an operation, and the proof of its safeness is left to the reader as Exercise 9.8.

Reduction MLS.1. If there is a vertex v of degree 2 where the neighbors u_1, u_2 of v are nonadjacent and also have degrees equal to 2, then delete v and add an edge u_1u_2.

In the main lemma of this section, we want to show that if G does not contain long paths with all inner vertices of degree 2 in G (i.e., when Rule MLS.1 is not applicable) and G has appropriately many vertices, then (G, k) is definitely a yes-instance.

Lemma 9.29. *If Reduction MLS.1 is inapplicable and* $|V(G)| \geq 6k^2$*, then* (G, k) *is a yes-instance.*

Proof. For the sake of contradiction, assume that $|V(G)| \geq 6k^2$ and (G, k) is a no-instance. Let us first select greedily a set S of vertices at distance at least 3 from each other (i.e., have disjoint closed neighborhoods), in the following way. Recall that for a positive integer d and vertex set $S \subseteq V(G)$, we denote by $N^d[S]$ the set of vertices of G within distance at most d from S. Consider the following procedure. Initially, set $S = \emptyset$ and, as long as $V(G) \setminus N^2[S]$ contains a vertex of degree at least 3 (in G), add to S a vertex of $V(G) \setminus N^2[S]$ that has the largest degree (in G). Let $r = |S|$ and $v_1, \ldots, v_r$ be the sequence of vertices added to S in this order. Denote $d_i = \deg_G(v_i)$.

Claim 9.30. *If* $\sum_{1 \leq i \leq r}(d_i - 2) \geq k$*, then* (G, k) *is a yes-instance.*

Proof. Note that, by construction, the vertices in the set S have pairwise disjoint neighborhoods. Consequently, we can construct a forest F consisting of r stars centered at vertices of S having exactly $\sum_{1 \leq i \leq r} d_i$ leaves. As G is connected, we can transform F into a tree in a step-by-step manner, each time adding to F a path between two of its connected components, losing at most two leaves at a time. We need to add at most $r-1$ paths to connect the r stars, hence we lose at most $2(r-1) < 2r$ leaves. Therefore, $\sum_{1 \leq i \leq r}(d_i - 2)$ is a lower bound on the number of leaves of the resulting tree. ⌟

In particular, if $r \geq k$ then we are done, because all the vertices in S are of degree at least 3 and by Claim 9.30 (G, k) is a yes-instance. In the following, by $\mathrm{dist}_G(u, v)$ we denote the distance between u and v in G.

Claim 9.31. *If, for some $v \in V(G)$ and $d \geq 1$, we have $|\{u \in V(G) : \mathrm{dist}_G(u, v) = d\}| \geq k$, then (G, k) is a yes-instance.*

Proof. It is enough to construct any tree spanning all the vertices at distance at most $d - 1$ from v and attach vertices at distance d as leaves. ⌟

Since G is connected, we can conclude that each connected component of $G \setminus N^2[S]$ contains at least one vertex at distance exactly 3 from some vertex of S. In the next claim, we give an upper bound on the number of such vertices.

Claim 9.32. *For any positive integer d, if there are more than $r(k - 1)$ vertices $u \in V(G)$ that are at distance exactly d from S (i.e., $\mathrm{dist}_G(u, S) = d$), then (G, k) is a yes-instance.*

Proof. As $|S| = r$, by the pigeonhole principle there exists $v \in S$ such that at least k vertices of G are at distance d from v. By Claim 9.31 for this particular vertex v, (G, k) is a yes-instance.

By Claim 9.32 for $d = 3$, there are at most $r(k - 1) < k^2$ connected components of $G \setminus N^2[S]$. Hence, to bound the number of vertices of $V(G) \setminus N^2[S]$, it is enough to bound the number of vertices of a single connected component of $G \setminus N^2[S]$.

Claim 9.33. *Each connected component of $G \setminus N^2[S]$ consists of at most four vertices.*

Proof. Denote $H = G \setminus N^2[S]$. As all the vertices of H are of degree at most 2, the graph H consists of paths and cycles only. If H contained a connected component of size at least 5, then there would be a vertex v of degree 2 in H, such that both its neighbors would be of degree 2 in H as well. However, no vertex of $V(H) = V(G) \setminus N^2[S]$ is of degree more than 2 in G, hence Rule MLS.1 would be applicable, a contradiction. ⌟

Consequently, $V(G) \setminus N^2[S]$ contains at most $4k^2$ vertices. By Claim 9.32 for $d = 1$ and $d = 2$, there are at most $2r(k - 1)$ vertices that are at distance 1 or at distance 2 from S. This implies $|N^2[S]| \leq r + r(k-1) + r(k-1) < 2k^2$. Therefore, G contains less than $6k^2$ vertices in total, a contradiction. This finishes the proof of Lemma 9.29. □

By Lemma 9.29 we infer the following reduction rule.

Reduction MLS.2. If $|V(G)| \geq 6k^2$, then conclude that (G, k) is a yes-instance.

Consequently we have the following theorem and a Turing kernelization as a corollary.

Theorem 9.34. *The* Max Leaf Subtree *problem in connected graphs admits a kernel with less than* $6k^2$ *vertices.*

Corollary 9.35. *The* Max Leaf Subtree *problem admits a polynomial Turing kernelization.*

Proof. Consider an instance (G, k) of Max Leaf Subtree and let $C_1, \ldots, C_r$ be the connected components of G. We invoke the kernelization algorithm from Theorem 9.34 on r instances (C_i, k), where the graph is obviously connected, producing an equivalent instance (C'_i, k'), where both $V(C'_i)$ and k' are bounded by $\mathcal{O}(k^2)$. Finally the Turing kernelization routine calls an oracle that determines the answer for each instance (C'_i, k'). If any of these answers is positive, then we can return yes, and otherwise it is safe to return no. □

Let us repeat that Corollary 9.35 stands in a counterposition to Exercise 15.4.2: While Max Leaf Subtree admits a polynomial Turing kernel, the existence of a classic polynomial kernel can be refuted under standard complexity assumptions.

Exercises

9.1 (✎). Consider the following weighted version of the task of finding the maximum number of vertex-disjoint T-paths in a graph G: each edge of G is additionally equipped with a nonnegative weight, and we look for a family of pairwise vertex-disjoint T-paths that has the maximum possible cardinality and, among those of the maximum possible cardinality, has the minimum possible total weight. Prove that this problem is polynomial-time solvable.

9.2 (☠). Consider the following weighted variant of Feedback Vertex Set: each vertex of the graph is additionally equipped with a nonnegative weight, and we look for a feedback vertex set that has at the same time at most k vertices and total weight at most W, where both k and W are given as an input. Adjust the argumentation of Section 9.1 to give a kernel for this variant with the number of vertices bounded polynomially in k.

9.3. In the Maximum Satisfiability problem, we are given a formula φ with n variables and m clauses, together with an integer k, and we are to decide whether there exists an assignment satisfying at least k clauses. Show that this problem is FPT when parameterized by $k' = k - m/2$.

9.4 (✎). Prove formally that Reduction CVC.4 is safe. In particular, show that the minimum possible size of a connected vertex cover before and after its application are equal.

9.5 (✎). Using Lemma 9.26, prove formally that Reduction CVC.4 is safe.

9.6 (☠). In the Edge Dominating Set problem, we look for a set of at most k edges in the given graph G whose endpoints form a vertex cover of G. Show that Edge Dominating Set in planar graphs admits a kernel with $\mathcal{O}(k)$ vertices.

9.7 (☠). Consider the following problem: given a planar graph G and integers k and d, decide if there exists a set X of at most k vertices of G such that each connected component of $G \setminus X$ has at most d vertices. Prove that this problem admits a kernel with $\mathcal{O}(kd)$ vertices.

9.8. Prove that Reduction MLS.1 is safe. In particular, show that there is a subtree with at least k leaves in the graph before the reduction if and only if there is a subtree with at least k leaves in the graph after the reduction.

9.9. Prove that one can strengthen Claim 9.33 and show that each connected component of $G \setminus N^2[S]$ consists of at most three vertices, leading to a slightly improved $5k^2$ upper bound on the number of vertices in the kernel.

9.10 (☠). Improve the reasoning of Lemma 9.29 to obtain an $\mathcal{O}(k)$ bound on the number of vertices of G.

Hints

9.1 Recall that finding a matching that has minimum possible weight among matchings of maximum possible cardinality is polynomial-time solvable.

9.2 First, observe that the argumentation of Section 9.1 breaks in the weighted setting at Lemma 9.8, as the vertices of $\bigcup \widehat{\mathcal{D}}$ may have much smaller weights than those of $\widehat{Z}$. You need to strengthen this lemma to force any feedback vertex set of size at most k to contain either v or $\widehat{Z}$. The main trick to obtain it is to replace the use of q-expansion lemma for $q = 2$ with a use for $q = k + 2$. The cost of this change is a cubic instead of quadratic bound on the number of vertices of the final graph.

9.3 Show that there exists a constant $c > 1/2$, such for that each CNF formula φ which does not contain at the same time a single literal clause x and its negation $\neg x$, there exists an assignment satisfying at least cm clauses.

9.6 Proceed as with Connected Vertex Cover in Section 9.3. Use Corollary 9.25 with the set A being the endpoints of the hypothetical solution to bound the number of vertices of degree more than 2. Vertices of degree 0 and 1 can be treated very similarly to the approach in Section 9.3. A bit more work is required with vertices of degree 2. First, show how to reduce many such vertices with common neighborhoods. Second, show that planarity implies that there are only $\mathcal{O}(k)$ possible pairs of vertices of a solution that share a common neighbor of degree 2.

9.7 Again, proceed as in Section 9.3 and apply Corollary 9.25 to the hypothetical solution X. Although the connected components of $G \setminus X$ are not isolated vertices, they have size bounded by d and you need to care only about the components that neighbor one or two vertices in X.

9.9 Note that a connected component of H which is a path of four vertices needs to have at least one of its endpoints of degree two in G.

9.10 Improve Claim 9.32 to get an $\mathcal{O}(r)$ bound on the number of vertices at distance d from G. To achieve this, combine the reasonings of Claims 9.30 and 9.31.

Bibliographic notes

The presented quadratic kernel for FEEDBACK VERTEX SET is due to Thomassé [420]. We remark that it is possible to bound the number of vertices of G by $4k^2$. Earlier polynomial kernels for this problem have $\mathcal{O}(k^{11})$ vertices [61] and $\mathcal{O}(k^3)$ vertices [49]. The presented proof of Gallai's theorem [223] is based on a simple proof of its extension, namely Mader's $\mathcal{S}$-paths theorem, given by Schrijver [411]. It should be mentioned that Gallai's theorem [223] often appears in the literature in a slightly different form that asserts the main equality of Lemma 9.5. Note that in this formulation Gallai's theorem provides a *tight* min-max relation between the maximum number of pairwise vertex-disjoint T-paths and an obstacle of the form of the set W, whereas our variant of Gallai's theorem introduces a multiplicative factor 2. However, we decided to present our variant as we find it easier to work with in the context of kernelization algorithms.

Parameterizations above and below guarantee were introduced by Mahajan and Raman [342]. The presented algorithm and polynomial compression for MAX-r-SAT is due to Alon, Gutin, Kim, Szeider, and Yeo [11], whereas the probabilistic approach was introduced in [250]. The number of variables in the compressed instance, and, consequently, the running time bound, has been further improved by Kim and Williams [290]. Exercise 9.3 is from [105]. Other interesting examples of algorithms and hardness results for above and below guarantee parameterizations include [103, 106, 107, 108, 248, 249, 343]. Gutin and Yeo [252] provide a comprehensive survey of various constraint satisfaction problems parameterized above and below tight bounds.

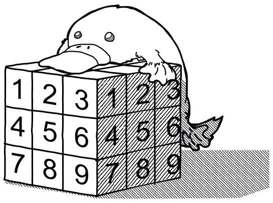
The presented kernel for CONNECTED VERTEX COVER in planar graphs is due to Wang, Yang, Guo, and Chen [432], where also the kernel for EDGE DOMINATING SET from Exercise 9.6 is shown. An earlier linear kernel for CONNECTED VERTEX COVER in planar graphs with at most $14k$ vertices is due to Guo and Niedermeier [243]. We remark that the kernel of Section 9.3 can be improved to yield a bound of $\frac{11}{3}k$ on the number of vertices [307].

The first example of Turing kernels of polynomial size were obtained for the DIRECTED MAX LEAF problem by Binkele-Raible, Fernau, Fomin, Lokshtanov, Saurabh, and Villanger [35]. Currently, the smallest known kernel for MAX LEAF SUBTREE on connected graphs has at most $3.75k$ vertices [169]. We remark that Reduction MLS.1 suffices for obtaining a kernel with $\mathcal{O}(k)$ vertices; see, for example, Appendix A in [274] as well as Exercise 9.10.

The formal foundation for Turing kernelization is the concept of *oracle Turing machines* (see [189]), where we constrain the power of the oracle to being able to answer only queries that are short in terms of the parameter. The definition of Turing kernelization is from [35]. Recent advances, like the work of Jansen [275] on planar LONGEST PATH, suggest that Turing kernelization has stronger computational power than previously suspected, due to possible usage of adaptiveness in the algorithm. Contrary to normal kernelization, we do not know any framework to refute the existence of Turing kernelizations under standard complexity assumptions. However, there has been work on creating a complexity hierarchy for kernelization [258], and some problems are conjectured to be hard for Turing kernelization.

Chapter 10
Algebraic techniques: sieves, convolutions, and polynomials

In this chapter we discuss selected techniques that have some "algebraic flavor," i.e., they use manipulation of algebraic expression, rather than analyzing the combinatorial properties of objects. Another common theme in these techniques is "sieving": we sieve out some unwanted objects by means of algebraic cancellation.



Essentially all the techniques we cover in this chapter are older than parameterized complexity and have been developed in different contexts. We begin in Section 10.1 with the classical inclusion–exclusion principle and present its simple (but non-obvious!) applications to problems like Hamiltonian Cycle or Chromatic Number, where the only parameter involved is the number of vertices, and a slightly more technical application to Steiner Tree parameterized by the number of terminals.

Next, in Section 10.2 we show that the algorithms based on the inclusion–exclusion principle can be reinterpreted using two transforms of functions on the subset lattice, i.e., functions of the form $f\colon 2^V \to \mathbb{Z}$. These transforms, called the zeta transform and the Möbius transform, play also a similar role as the Fourier transform and the inverse Fourier transform in the harmonic analysis of functions, e.g., they can be used for fast computation of convolutions of functions on the subset lattice. This is shown in Section 10.3, with an application to Chromatic Number. In Chapter 11 we will see more applications of this technique in accelerating dynamic-programming algorithms for bounded treewidth graphs.

Finally, in Section 10.4 we present algorithms based on evaluating multivariate polynomials. To use this approach for a parameterized problem, one designs a polynomial p that

- can be evaluated in FPT time, and

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_10

- encodes all the solutions, in particular p is nonzero if and only if there is at least one solution.

Then it turns out that there is a simple randomized algorithm for testing whether p is nonzero based on the Schwartz-Zippel lemma. We illustrate this scheme by algorithms for Longest Path. In the resulting algorithms, one can observe an algebraic reformulation of the color-coding approach from Section 5.2.

Throughout this chapter, unless otherwise specified, n and m denote the number of vertices and edges of the input graph. We will often use two different bracket notations in this chapter. The first is the Iverson bracket notation, which is 1 if the condition in square brackets is satisfied, and 0 otherwise. For example, for integers a, b, c, d, the value of the expression $[a < b, c < d]$ is 1 if $a < b$ and $c < d$ both hold, and 0 otherwise. The second usage of brackets is to denote for an integer k the set $[k] = \{1, \ldots, k\}$.

We will often use the binomial theorem, which states that $\sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i} = (a+b)^n$. Note that if $a = -b$, then this expression is 0 for every $n > 0$, but it is 1 for $n = 0$.

10.1 Inclusion–exclusion principle

In this section, we present applications of the inclusion–exclusion principle, a well-known tool in combinatorics. Usually, the inclusion–exclusion principle is described as a formula for computing $|\bigcup_{i=1}^{n} A_i|$ for a collection of sets $A_1, \ldots, A_n$. For two sets A_1 and A_2, the size $|A_1 \cup A_2|$ of the union can be less than the sum $|A_1| + |A_2|$ of the sizes: the expression $|A_1| + |A_2|$ counts the size of the intersection $A_1 \cap A_2$ twice, hence we have to subtract it to avoid double counting:

$$|A_1 \cup A_2| = |A_1| + |A_2| - |A_1 \cap A_2|.$$

For three sets A_1, A_2, A_3, we have the formula

$$|A_1 \cup A_2 \cup A_3| = |A_1| + |A_2| + |A_3| - |A_1 \cap A_2| - |A_1 \cap A_3| - |A_2 \cap A_3| + |A_1 \cap A_2 \cap A_3|.$$

That is, after subtracting the size of the pairwise intersections, the common intersection $A_1 \cap A_2 \cap A_3$ was considered three times positively and three times negatively, thus we have to add it once more. Below we state the formula for the general case.

Theorem 10.1 (Inclusion–exclusion principle, union version). *Let $A_1, \ldots, A_n$ be finite sets. Then*

$$\Big|\bigcup_{i \in [n]} A_i\Big| = \sum_{\emptyset \neq X \subseteq [n]} (-1)^{|X|+1} \cdot \Big|\bigcap_{i \in X} A_i\Big|. \tag{10.1}$$

Proof. Consider any element $e \in \bigcup_{i\in[n]} A_i$. Let $k \geq 1$ be the number of sets A_i containing e and let $A_{i_1}, \ldots, A_{i_k}$ be these sets. Note that for $X \neq \emptyset$ we have $e \in \bigcap_{i\in X} A_i$ if and only if $X \subseteq \{i_1, \ldots, i_k\}$. Hence the amount e contributes to the right-hand side of (10.1) equals

$$\sum_{\emptyset\neq X\subseteq\{i_1,\ldots,i_k\}} (-1)^{|X|+1} = (-1)\sum_{j=1}^{k}\binom{k}{j}(-1)^j = (-1)((1-1)^k - 1) = 1,$$

as required. □

Intuitively, Theorem 10.1 allows us to compute the size of the union if for some reason the sizes of all the intersections are easy to compute. However, for our applications, we need the dual form of this statement that allows an easy computation of the size of the intersection if the sizes of all the unions are easy to compute. We state the dual form below. It can be easily derived from Theorem 10.1 using De Morgan's laws so we skip the proof here; however, in Section 10.4 we prove a generalized, weighted inclusion–exclusion principle.

Theorem 10.2 (Inclusion–exclusion principle, intersection version). *Let $A_1, \ldots, A_n \subseteq U$, where U is a finite set. Denote $\bigcap_{i\in\emptyset}(U \setminus A_i) = U$. Then*

$$\Big|\bigcap_{i\in[n]} A_i\Big| = \sum_{X\subseteq[n]} (-1)^{|X|}\Big|\bigcap_{i\in X}(U \setminus A_i)\Big|.$$

In a typical algorithmic application of the inclusion–exclusion formula we need to count some objects that belong to a universe U, and it is in some sense *hard.* More precisely, we are interested in objects that satisfy *n requirements* $A_1, \ldots, A_n$ and each requirement is defined as the set of objects that satisfy it. Thus, the inclusion–exclusion formula translates the problem of computing $|\bigcap_{i\in[n]} A_i|$ into computing 2^n terms of the form $|\bigcap_{i\in X}(U \setminus A_i)|$. In our applications, computing these terms is in some sense *easy.* If, for example, each of them is computable in polynomial time, then the inclusion–exclusion formula gives an algorithm which performs $2^n n^{\mathcal{O}(1)}$ arithmetic operations.

10.1.1 Hamiltonian Cycle

The idea above applies smoothly to the well-known Hamiltonian Cycle problem: given a directed graph G, we want to decide whether G contains a cycle which contains all the vertices of G, called the Hamiltonian cycle. In fact, we will solve a more general problem — we will *count* the number of Hamiltonian cycles. Recall that Exercise 6.2 asked you for a $2^n n^{\mathcal{O}(1)}$-time algorithm using dynamic programming. In this section we show that using the inclusion–exclusion formula we can get an algorithm running in roughly the same time, but using only *polynomial space.*

A walk $v_0, \ldots, v_k$ is called *closed* when $v_0 = v_k$. We also say that the *length* of this walk is k, i.e., the number of its edges.

Let v_0 be an arbitrary vertex from $V(G)$ and let the universe U be the set of all closed walks of length n of the form $v_0, v_1, \ldots, v_n = v_0$. For every vertex $v \in V(G)$, define a requirement $A_v \subseteq U$ as the set of all closed walks of length n that start at v_0 and visit v at least once. Clearly, our goal is to compute $|\bigcap_{v \in V(G)} A_v|$: a closed walk of length n can visit all vertices at least once only if it is a cycle of length exactly n. By the inclusion–exclusion principle (Theorem 10.2) this reduces to computing, for every $X \subseteq V(G)$, the value of $|\bigcap_{v \in X}(U \setminus A_v)|$, i.e., the number of closed walks of length n from v_0 in graph $G' := G - X$. This is easily computable in polynomial time: if M is the adjacency matrix of G', then we compute the n-th power M^n and return the entry from the diagonal of M^n that corresponds to v_0 (see Exercise 10.2). Therefore, we can compute in polynomial time each of the 2^n terms of the inclusion–exclusion formula and hence we can compute $|\bigcap_{v \in V(G)} A_v|$.

There is one more thing to check. So far we just showed that the number of Hamiltonian cycles can be found using $2^n n^{\mathcal{O}(1)}$ arithmetic operations. The question is how costly is an arithmetic operation? Every number that appears in the algorithm is bounded by the number of walks of length at most n from some vertex, which is $\mathcal{O}(n^n)$. This means it takes $\mathcal{O}(n \log n)$ bits and each arithmetic operation takes polynomial time. An important property of the algorithm sketched above is that it takes only $\mathcal{O}(n^2)$, i.e., polynomial, space.

Theorem 10.3. *The number of Hamiltonian cycles in a directed n-vertex graph can be computed in time $2^n n^{\mathcal{O}(1)}$ and polynomial space.*

10.1.2 Steiner Tree

In the unweighted Steiner Tree problem, we are given an undirected graph G, a set of vertices $K \subseteq V(G)$, called *terminals*, and a number $\ell \in \mathbb{N}$. The goal is to determine whether there is a tree $H \subseteq G$ (called the Steiner tree) with at most ℓ edges that connects all the terminals. Recall that in Section 6.1.2 we presented a $3^{|K|} n^{\mathcal{O}(1)}$-time algorithm for Steiner Tree using dynamic programming (in fact, that algorithm works also for the weighted version, where we have arbitrary weights on the edges). Here we show a $2^{|K|} n^{\mathcal{O}(1)}$-time algorithm for (the unweighted version of) this problem parameterized by $|K|$. Without loss of generality, we assume $|K| \geq 2$.

The idea is to apply the inclusion–exclusion-based algorithm similar to the one we use for Hamiltonian Cycle. There, we observed that although

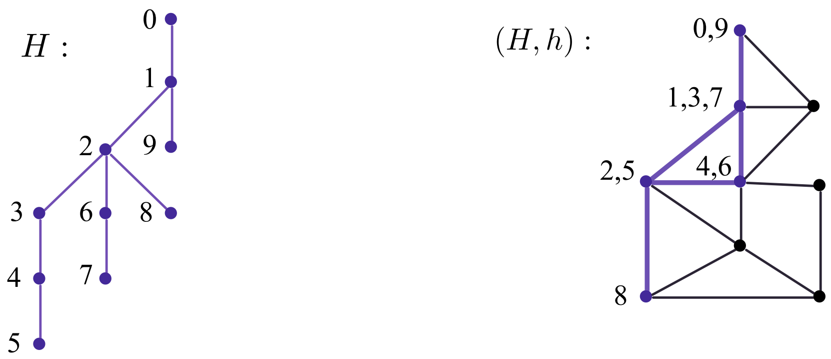


Fig. 10.1: An ordered rooted tree H and a branching walk (H, h) in a graph

counting paths or cycles is hard, counting walks is easy. Here, instead of counting trees, we will count *branching walks*. Recall that in an *ordered* rooted tree the children of every node have a fixed order. This assumption is important because the algorithm we present here *counts* certain branching walks and their number depends on whether the corresponding tree is ordered or not.

A branching walk is a relaxed notion of a tree, or more precisely a homomorphic image of a tree. Formally, a *branching walk* in graph G is a pair $B = (H, h)$ where H is an ordered rooted tree and $h\colon V(H) \to V(G)$ is a homomorphism, i.e., if $xy \in E(H)$ then $h(x)h(y) \in E(G)$. We say that branching walk B is from s when $h(r) = s$, where r is the root of H. We denote by $V(B) = h(V(H))$ the set of vertices in G where h maps the vertices of H. Observe that $h(V(H))$ induces a connected graph in G. The length of the branching walk B is defined as $|E(H)|$. Of course, every walk is a branching walk (with tree H being a path). Also, every tree H in G corresponds to a branching walk (H, h) with h being the identity function. One can see that the image $h(H) = \{h(u)h(v) \ : \ uv \in E(H)\}$ is a tree if and only if h is injective. Figure 10.1 shows an example of a branching walk with non-injective h. The following observation provides a connection between the STEINER TREE problem and branching walks (the easy proof is left for the reader).

Observation 10.4. Let $s \in K$ be any terminal. Graph G contains a tree H such that $K \subseteq V(H)$ and $|E(H)| \leq \ell$ if and only if G contains a branching walk $B = (H_B, h)$ from s in G such that $K \subseteq V(B)$ and $|E(H_B)| = \ell$.

Following Observation 10.4, we will count branching walks of length ℓ from s that visit all the terminals and return true if and only if the resulting number is nonzero. Our universe U is now the set of all branching walks of length ℓ from s. For every $v \in K$, we have the requirement $A_v = \{B \in U \ : \ v \in V(B)\}$. Hence, our goal is to compute $|\bigcap_{v \in K} A_v|$. By the inclusion–exclusion principle, this reduces to computing, for every $X \subseteq K$, the value of $|\bigcap_{v\in X}(U\setminus A_v)|$. Note here that $\bigcap_{v\in X}(U\setminus A_v)$ is the set of exactly those branching walks from U that avoid the vertices of X.

Lemma 10.5. *For every $X\subseteq K$, the value of $|\bigcap_{v\in X}(U\setminus A_v)|$ can be computed using $\mathcal{O}(m\ell^2)$ arithmetic operations.*

Proof. Let $G'=G-X$. For $a\in V(G')$ and $j\in\{0,\ldots,\ell\}$, let $b_j(a)$ be the number of length j branching walks from a in G'. Notice that if $s\notin V(G')$ we should return 0, and otherwise our goal is to compute $b_\ell(s)$. We compute $b_j(a)$ for all $a\in V(G')$ and $j\in\{0,\ldots,\ell\}$ using dynamic programming with the following recursive formula:

$$b_j(a)=\begin{cases}1 & \text{if } j=0,\\ \displaystyle\sum_{t\in N_{G'}(a)}\ \sum_{j_1+j_2=j-1} b_{j_1}(a)b_{j_2}(t) & \text{otherwise.}\end{cases}$$

In the formula above, for every neighbor t of a, we count the number of branching walks where t is the first child of a; for a fixed t we consider all possibilities for the size j_2 of the subtree rooted at t (and j_1 is the number of remaining edges of the tree excluding the edge at as well). Note that this is the point where considering branching walks instead of trees pays off: multiplying $b_{j_1}(a)$ and $b_{j_2}(t)$ is justified, as we do not require the branching walks starting at a and t to be vertex disjoint; hence we may combine any two of them. We leave the formal proof of the correctness of the formula to the reader. The time bound follows, since for every $j\in\{0,\ldots,\ell\}$ and every edge of G', the inner sum is computed twice. □

Since the numbers in our computations have $\mathcal{O}(\ell\log n)$ bits (Exercise 10.4), arithmetic operations take polynomial time. We have just proved the following theorem.

Theorem 10.6. *The unweighted* Steiner Tree *problem can be solved in time $2^{|K|}\cdot n^{\mathcal{O}(1)}$ and polynomial space.*

In the previous section we showed an algorithm for *counting* Hamiltonian cycles, which obviously also solves the corresponding decision problem. Here we solved unweighted Steiner Tree by counting branching walks. It is natural to ask whether one can count Steiner trees within similar running time. The answer is no, unless FPT = W[1] (see Exercise 10.5).

10.1.3 Chromatic Number

A k-coloring of a graph G is a function $c: V(G)\to[k]$, such that $c(u)\neq c(v)$ whenever u and v are adjacent. In the Chromatic Number problem we are given a graph G with an integer k and we want to know whether G has a

k-coloring. Recall that Exercise 6.1 asked you for a $3^n n^{\mathcal{O}(1)}$-time algorithm, using dynamic programming. In this section we provide a faster algorithm based on the inclusion–exclusion principle. Clearly, each color class in k-coloring is an independent set and every subset of an independent set is also an independent set. This leads us to the following easy observation.

Observation 10.7. *G has a k-coloring if and only if there is a* cover *of $V(G)$ by k independent sets, i.e., there are k independent sets $I_1, \ldots, I_k$ such that $\bigcup_{j=1}^{k} I_j = V(G)$.*

By Observation 10.7 it suffices to compute the number of covers of $V(G)$ by independent sets. Our universe U is now the set of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets (not necessarily disjoint nor even different!). Clearly, we need to cover every vertex, i.e., for every $v \in V(G)$ we have the requirement

$$A_v = \left\{ (I_1, \ldots, I_k) \in U \ : \ v \in \bigcup_{j=1}^{k} I_j \right\}.$$

Then $|\bigcap_{v \in V(G)} A_v|$ is the required number of covers of $V(G)$ by independent sets. For $Y \subseteq V(G)$, let $s(Y)$ be the number of independent sets in $G[Y]$, the subgraph of G induced by Y. By the inclusion–exclusion principle, finding $|\bigcap_{v \in V(G)} A_v|$ reduces to computing, for every $X \subseteq V(G)$, the value of

$$\left| \left\{ (I_1, \ldots, I_k) \in U \ : \ I_1, \ldots, I_k \subseteq V(G) \setminus X \right\} \right| = s(V(G) \setminus X)^k.$$

Although we do not know how to compute $s(V(G) \setminus X)^k$ in polynomial time, $s(Y)$ can be computed at the beginning *simultaneously for all subsets* $Y \subseteq V(G)$ using a simple dynamic programming routine that employs the formula

$$s(Y) = s(Y \setminus \{y\}) + s(Y \setminus N[y]), \tag{10.2}$$

where y is an arbitrary vertex in Y. This requires $\mathcal{O}(2^n)$ arithmetical operations on $\mathcal{O}(n)$-bit numbers, and takes space $2^n n^{\mathcal{O}(1)}$. Once we have $s(Y)$ computed for every $Y \subseteq V(G)$, the value of $s(V(G) \setminus X)^k$ can be found in polynomial time by $\mathcal{O}(\log k)$ multiplications of $\mathcal{O}(nk)$-bit numbers. We have just proved the following theorem.

Theorem 10.8. *The* Chromatic Number *problem can be solved in time and space $2^n n^{\mathcal{O}(1)}$.*

It is natural here to ask for a *polynomial space* algorithm. A $2^n n^{\mathcal{O}(1)}$ time polynomial-space algorithm is currently not known. Let us modify the algorithm from Theorem 10.8 at the cost of worse running time. When we aim at polynomial space we cannot afford to tabulate the number of independent sets $s(Y)$ for every $Y \subseteq V(G)$. If we just use the formula 10.2 in a recursive procedure, without storing the computed values, computing $s(Y)$ takes

$2^{|Y|}n^{\mathcal{O}(1)}$ time for *each* subset $Y \subseteq V(G)$ *separately*. By the binomial theorem the total time needed to compute $|\bigcap_{v\in V(G)} A_v|$ using the inclusion–exclusion principle is then

$$\left(\sum_{X\subseteq V(G)} 2^{|V(G)\setminus X|}\right) n^{\mathcal{O}(1)} = \left(\sum_{k=0}^{n} \binom{n}{k} 1^k 2^{n-k}\right) n^{\mathcal{O}(1)} = 3^n n^{\mathcal{O}(1)}.$$

However, there are better branching algorithms for finding the number of independent sets in a given n-vertex graph. In Exercise 3.17 the reader is asked to show such an algorithm running in time $1.381^n n^{\mathcal{O}(1)}$. The currently best known polynomial-space algorithm is due to Wahlström [429] and runs in $1.238^n n^{\mathcal{O}(1)}$ time. By a similar application of the binomial theorem as above we get the following result.

Theorem 10.9. *The* Chromatic Number *problem can be solved in time $2.238^n n^{\mathcal{O}(1)}$ and polynomial space.*

Finally, let us note that there is a $2^n n^{\mathcal{O}(1)}$-time algorithm for *counting* k-colorings and we present one in Section 10.3.1.

10.2 Fast zeta and Möbius transforms

In this section, we consider functions on the subset lattice, i.e., functions of the form $f\colon 2^V \to \mathbb{Z}$, where V is some n-element set (in fact, $\mathbb{Z}$ can be replaced by any ring). It turns out that the inclusion–exclusion algorithms from Section 10.1 can be phrased in terms of transforms between functions on the subset lattice, i.e., functions that take as an argument a function $f\colon 2^V \to \mathbb{Z}$ and return another function $g\colon 2^V \to \mathbb{Z}$. We use simplified notation, i.e., for a transform α and function f the function $\alpha(f)$ is denoted as αf. The first is the *zeta transform*, defined as

$$(\zeta f)(X) = \sum_{Y\subseteq X} f(Y).$$

Let us also define the *Möbius transform*,

$$(\mu f)(X) = \sum_{Y\subseteq X} (-1)^{|X\setminus Y|} f(Y).$$

It will be convenient to have the following *odd-negation transform*,

$$(\sigma f)(X) = (-1)^{|X|} f(X).$$

We will frequently use compositions of transforms. The composition $\alpha \circ \beta$ of two transforms α and β is denoted shortly by $\alpha\beta$. Recall that in the composition notation α is applied after β, i.e., for every function $f\colon 2^V \to \mathbb{Z}$ we have $\alpha\beta f = \alpha(\beta(f))$. Clearly, $\sigma\sigma$ is the identity transform id. It is also easy to see (Exercise 10.10) that the three transforms above are related as follows.

Proposition 10.10. $\zeta = \sigma\mu\sigma$ *and* $\mu = \sigma\zeta\sigma$.

As we will see, the following formula is a counterpart of the inclusion–exclusion principle.

Theorem 10.11 (Inversion formula). *We have* $\mu\zeta = \zeta\mu = \mathrm{id}$, *that is,*

$$f(X) = (\mu\zeta f)(X) = (\zeta\mu f)(X)$$

for every $X \subseteq V$.

Proof. We observe that:

$$\begin{aligned}
(\mu\zeta f)(X) &= (\sigma\zeta\sigma\zeta f)(X) && \text{(Proposition 10.10)}\\
&= (-1)^{|X|}\sum_{Y\subseteq X}(\sigma\zeta f)(Y) && \text{(expanding σ and ζ)}\\
&= (-1)^{|X|}\sum_{Y\subseteq X}(-1)^{|Y|}\sum_{Z\subseteq Y} f(Z) && \text{(expanding σ and ζ)}\\
&= (-1)^{|X|}\sum_{Z\subseteq X} f(Z)\cdot \sum_{Z\subseteq Y\subseteq X}(-1)^{|Y|} && \text{(reordering terms)}\\
&= f(X) + (-1)^{|X|}\sum_{Z\subsetneq X} f(Z)\cdot \underbrace{\sum_{Z\subseteq Y\subseteq X}(-1)^{|Y|}}_{\text{sums to 0}} && \text{(separating $Z = X$)}\\
&= f(X).
\end{aligned}$$

The inner sum in the penultimate line is equal to 0 because every nonempty finite set A has the same number of subsets of odd and even cardinality (here we consider $A = X \setminus Z$). To see this, pick any $a \in A$. We partition all subsets of A into pairs: every subset $S \subseteq A\setminus\{a\}$ can be paired up with the set $S\cup\{a\}$. Each such pair has one set of even size and one set of odd size, which proves the claim.

The second part of the theorem follows by $\zeta\mu = \zeta\sigma\zeta\sigma = \sigma\mu\sigma\sigma\zeta\sigma = \sigma\mu\zeta\sigma = \sigma\sigma = \mathrm{id}$. □

Let us revisit the Chromatic Number problem, now using the transforms. For $X \subseteq V(G)$, let $f(X)$ be the number of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets in G and $\bigcup_{j=1}^{k} I_j = X$. Then, for every $X \subseteq V(G)$, the value $f(X)$ is the number of covers of X by k independent sets. Observe that $(\zeta f)(X) = \sum_{Y\subseteq X} f(Y)$ is the number of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets in G and $\bigcup_{j=1}^{k} I_j \subseteq X$. Using the notation from Section 10.1.3, we have $(\zeta f)(X) = s(X)^k$, and hence the values of ζf can be computed in polynomial time after a $2^n n^{\mathcal{O}(1)}$-time preprocessing, as in Section 10.1.3. By the inversion formula (Theorem 10.11), we have $f(V(G)) = (\mu\zeta f)(V(G))$ and hence we can compute the values of f from ζf directly using the definition of μ in time $2^n n^{\mathcal{O}(1)}$. By Observation 10.7, graph G is k-colorable if and only if $f(V(G)) \neq 0$. This re-proves Theorem 10.8.

As we will see, sometimes it is useful to compute the zeta or Möbius transform for *all* subsets X of V. The algorithm that computes these values separately for each subset using the definition takes $\mathcal{O}(\sum_{X\subseteq V} 2^{|X|}) = \mathcal{O}(\sum_{i=0}^{|X|}\binom{|X|}{i}2^i) = \mathcal{O}((2+1)^n) = \mathcal{O}(3^n)$ arithmetic operations and evaluations of the transformed function f. It turns out that this can be done in time roughly the same as the size of the output 2^n.

Theorem 10.12 (Fast zeta/Möbius transform). *Given all the 2^n values of f in the input, all the 2^n values of ζf and μf can be computed using $\mathcal{O}(2^n \cdot n)$ arithmetic operations.*

Proof. Consider the zeta transform. Let $V = \{1, \dots, n\}$. Instead of considering f to be a function on subsets of V, it will be convenient to interpret f as an n-variable function on $\{0,1\}$. We let $f(x_1, \dots, x_n)$ be $f(X)$, where $X \subseteq V$ is the set with characteristic vector $(x_1, \dots, x_n)$, that is, $i \in X$ if and only if $x_i = 1$. With this interpretation, the zeta transform is defined as

$$(\zeta f)(x_1, \dots, x_n) = \sum_{y_1,\dots,y_n \in \{0,1\}} [y_1 \le x_1, \dots, y_n \le x_n] \cdot f(y_1, \dots, y_n)$$

(recall that the value of the bracket is 1 if the expression in the bracket is true and 0 otherwise).

Consider fixing the last $n-j$ bits, that is, we define

$$\zeta_j(x_1, \dots, x_n) := \sum_{y_1,\dots,y_j \in \{0,1\}} [y_1 \le x_1, \dots, y_j \le x_j] \cdot f(y_1, \dots, y_j, \underbrace{x_{j+1}, \dots, x_n}_{\text{fixed}}).$$

Consistently, denote $\zeta_0(x_1, \dots, x_n) := f(x_1, \dots, x_n)$. Observe that $\zeta_n(X) = (\zeta f)(X)$. The values of $\zeta_j(x_1, \dots, x_n)$ for all $j \in \{1, \dots, n\}$ and $x_1, \dots, x_n \in \{0,1\}$ can be found by dynamic programming using the following formula.

$$\zeta_j(x_1, \dots, x_n) = \begin{cases} \zeta_{j-1}(x_1, \dots, x_n) & \text{when } x_j = 0, \\ \zeta_{j-1}(x_1, \dots, x_{j-1}, 1, x_{j+1}, \dots, x_n) + \\ \zeta_{j-1}(x_1, \dots, x_{j-1}, 0, x_{j+1}, \dots, x_n) & \text{when } x_j = 1. \end{cases}$$

Indeed, if $x_j = 0$, then every nonzero term has $y_j = 0$ (and hence $y_j = x_j$), thus we may consider x_j fixed as well. On the other hand, if $x_j = 1$, then we need to consider two types of terms: those with $y_j = 1$ and those with $y_j = 0$. Therefore, by fixing the value of the j-th coordinate to 1 and 0, respectively, we get the contribution of the two types of terms. We see that the computation takes $\mathcal{O}(2^n \cdot n)$ arithmetic operations. The claim for the Möbius transform follows now by Proposition 10.10. $\square$

In Theorem 10.12, we state the time bound only by means of the number of arithmetic operations. The precise asymptotic bound for the time depends on the model of computations used, e.g., answers to questions like "how long does it take to address an n-dimensional array?" or "what is the cost of an arithmetic operation?" (note that the algorithm computes sums of $\Omega(2^n)$ input numbers). However, if an arithmetic operation on two input numbers (values of function f) can be carried out in time polynomial in n, then we can safely claim a $2^n n^{\mathcal{O}(1)}$ upper bound on the running time.

Using the fast Möbius transform, in the last step of the coloring algorithm described in this section, instead of computing just $(\mu\zeta f)(V(G))$ we can compute $f(X) = (\mu\zeta f)(X)$ for all $X \subseteq V(G)$. By Observation 10.7, for every $X \subseteq V(G)$ the graph $G[X]$ is k-colorable if and only if $f(X) \neq 0$. It follows that in time $2^n n^{\mathcal{O}(1)}$ we have found *all* k-colorable induced subgraphs of G.

10.3 Fast subset convolution and cover product

In this section, we consider binary operations on functions on the subset lattice. The *subset convolution* of two functions $f, g\colon 2^V \to \mathbb{Z}$ is a function $(f * g)\colon 2^V \to \mathbb{Z}$ such that for every $Y \subseteq V$,

$$(f * g)(Y) = \sum_{X \subseteq Y} f(X) g(Y \setminus X).$$

Note that equivalently,

$$(f * g)(Y) = \sum_{\substack{A \cup B = Y \\ A \cap B = \emptyset}} f(A) g(B).$$

The *cover product* of two functions $f, g\colon 2^V \to \mathbb{Z}$ is a function $(f *_c g)\colon 2^V \to \mathbb{Z}$ such that for every $Y \subseteq V$,

$$(f *_c g)(Y) = \sum_{A \cup B = Y} f(A) g(B).$$

Subset convolution and cover product are computed in many algorithms. Perhaps the most natural application in parameterized complexity is accelerating the dynamic programming over tree decompositions, and we will see such examples in Chapter 11. At the end of this section, we motivate subset convolution by presenting an algorithm that counts proper colorings for all induced subgraphs of a graph.

Now let us focus on algorithms for computing (all the $2^{|V|}$ values of) the subset convolution and the cover product. The naive algorithms following from the definition require $\mathcal{O}(3^n)$ and $\mathcal{O}(4^n)$ arithmetic operations, respectively. Indeed, for the subset convolution the number of additions and multiplications is proportional to the total number of pairs of sets X and Y such that $Y \subseteq X \subseteq V$, which equals $\sum_{k=0}^{n} \binom{n}{k} 2^k = 3^n$ by the binomial theorem. For the cover product, we consider all the triples (A, B, Y) such that $A, B \subseteq Y \subseteq V$. There are $2^n \cdot 2^n = 4^n$ such triples, since every choice of subsets $A, B \subseteq V$ determines the triple.

In what follows, we show that the subset convolution and cover product can be computed much faster, i.e., in time $2^n n^{\mathcal{O}(1)}$. (Note that this is the size of the input, up to a polynomial factor.)

The *pointwise product* of two functions $f, g\colon 2^V \to \mathbb{Z}$ is a function $(f \cdot g)\colon 2^V \to \mathbb{Z}$ such that for every $Y \subseteq V$,

$$(f \cdot g)(Y) = f(Y) \cdot g(Y).$$

Clearly, the pointwise product can be computed in $\mathcal{O}(2^n)$ arithmetic operations. This motivates the following lemma.

Lemma 10.13. *The zeta transform of the cover product is the pointwise product of the zeta-transformed arguments, i.e., $\zeta(f *_c g) = (\zeta f) \cdot (\zeta g)$.*

Proof. For every $Y \subseteq V$ we have,

$$\begin{aligned} \zeta(f *_c g)(Y) &= \sum_{X \subseteq Y} \sum_{A \cup B = X} f(A) g(B) = \sum_{A \cup B \subseteq Y} f(A) g(B) \\ &= \left(\sum_{A \subseteq Y} f(A) \right) \left(\sum_{B \subseteq Y} g(B) \right) = (\zeta f(Y))(\zeta g(Y)). \end{aligned}$$

□

A reader familiar with the classic Fourier transform considered in calculus will certainly recognize in Lemma 10.13 an analogue of one of the most fundamental properties of the Fourier transform: the zeta transform translates convolution into point-wise multiplication. By the inversion formula we get a backward translation by means of the Möbius transform. Formally, we have

$$f *_c g = \mu((\zeta f) \cdot (\zeta g)).$$

Using Lemma 10.13, we now give an algorithm for computing the cover product.

Theorem 10.14. *For two functions $f, g\colon 2^V \to \mathbb{Z}$, given all the 2^n values of f and g in the input, all the 2^n values of the cover product $f *_c g$ can be computed in $\mathcal{O}(2^n \cdot n)$ arithmetic operations.*

Proof. Similarly as in the example with coloring using ζ and μ-transforms, by Lemma 10.13, the ζ-transform of the desired result is easy to compute. More precisely, all the 2^n values of $\zeta(f *_c g)$ can be found by computing all the 2^n values of ζf and all the 2^n values of ζg, and finally multiplying the results for every $Y \subseteq V$. By Theorem 10.12, this takes $\mathcal{O}(2^n \cdot n)$ arithmetic operations. By the Inversion Formula (Theorem 10.11), we can compute $f *_c g$ by applying the Möbius transform to $\zeta(f *_c g)$. This is done again in time $\mathcal{O}(2^n \cdot n)$ using Theorem 10.12. □

We now show how to use the algorithm for computing the cover product to compute the convolution. The key idea is to consider separately, for every $0 \le i \le |X|$, the contribution of those terms of $f * g$ where $|A| = i$ and $|B| = |X| - i$.

Theorem 10.15 (Fast subset convolution). *For two functions $f, g\colon 2^V \to \mathbb{Z}$, given all the 2^n values of f and g in the input, all the 2^n values of the subset convolution $f * g$ can be computed in $\mathcal{O}(2^n \cdot n^3)$ arithmetic operations.*

Proof. Let us introduce the following notation which truncates a function to sets of a fixed cardinality:

$$f_k(S) = f(S) \cdot [|S| = k].$$

The key to the fast subset convolution computation is the observation that disjointness can be controlled using cardinalities, as follows.

$$\begin{aligned}
(f * g)(X) &= \sum_{\substack{A \cup B = X \\ A \cap B = \emptyset}} f(A)g(B) \\
&= \sum_{i=0}^{|X|} \sum_{\substack{A \cup B = X \\ |A|=i, |B|=|X|-i}} f(A)g(B) \\
&= \sum_{i=0}^{|X|} \sum_{A \cup B = X} f_i(A) g_{|X|-i}(B) \\
&= \sum_{i=0}^{|X|} (f_i *_c g_{|X|-i})(X).
\end{aligned} \tag{10.3}$$

Note that the second equality holds, since if A and B form a partition of X, then clearly $|A| = i$ and $|B| = |X| - i$ for some $0 \le i \le |X|$. Our algorithm is as follows. First, for all $i, j \in \{0, \dots, n\}$ and $X \subseteq 2^V$ we compute and store $(f_i *_c g_j)(X)$. By Theorem 10.14, it takes $\mathcal{O}(2^n \cdot n^3)$ operations. Then we can compute $(f * g)(X)$ for all $X \subseteq 2^V$ using the identity (10.3) in $\mathcal{O}(2^n \cdot n)$ operations. $\square$

10.3.1 Counting colorings via fast subset convolution

In this section, we motivate the notion of subset convolution by an algorithm that *counts* the number of k-colorings in every induced subgraph of the input graph G. Note that we interpret a k-coloring as a function $2^{V(G)} \to [k]$ and we count the number of distinct functions that are valid k-colorings. This means that we consider two colorings to be distinct (and count both of them) if they partition the vertices into color classes the same way, but the colors of the classes are different.

Define a function $s\colon 2^{V(G)} \to \{0, 1\}$ such that for every $X \subseteq V(G)$ we have $s(X) = 1$ if and only if X is an independent set. Then

$$(\underbrace{s * s * \cdots * s}_{k \text{ times}})(X)$$

is the number of k-colorings of $G[X]$ (note that a k-coloring needs not use all the k colors). It can be found in time and space $2^n n^{\mathcal{O}(1)}$ by applying the fast subset convolution algorithm (Theorem 10.15) $k - 1$ times (or even $\mathcal{O}(\log k)$ times, if we apply a fast exponentiation algorithm, that is, we iteratively compute the 2^{2i}-th power by multiplying the 2^i-th power with itself). We have proved the following theorem.

Theorem 10.16. *The number of k-colorings in every induced subgraph of the input graph G can be computed in time and space $2^n n^{\mathcal{O}(1)}$.*

10.3.2 Convolutions and cover products in min-sum semirings

Recall that a semiring is an algebraic structure similar to a ring, but without the requirement that each element must have an additive inverse. More precisely, a semiring is a triple $(R, +, \cdot)$, where R is a set and $+$ and $\cdot$ are binary operations called addition and multiplication. Both operations are associative, addition is commutative, multiplication distributes over addition, i.e., $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(b + c)va = (b \cdot a) + (c \cdot a)$. Finally, there are two elements $0, 1 \in R$ such that 0 is the additive identity and annihilates R, i.e., $a \cdot 0 = 0$, while 1 is the multiplicative identity. In a ring, additionally, for every element a there is an element b (called an inverse of a) such that $a + b = 0$.

In applications we frequently need to compute convolution-like expressions of the form

$$(f * g)(Y) = \min_{\substack{A \cup B = Y \\ A \cap B = \emptyset}} (f(A) + g(B)),$$

or

$$(f *_c g)(Y) = \min_{A \cup B = Y} (f(A) + g(B)).$$

At first glance reusing the subset convolution and cover product notation here seems confusing. However, one can notice that in fact these *are* the subset convolution and the cover product for functions f and g with values in the *integer min-sum semiring*, i.e., the semiring $(\mathbb{Z} \cup \{+\infty\}, \min, +)$. (Note that $+\infty$ is the additive identity in this semiring.) Similar expressions with max replacing min might also be of interest and then they correspond to the subset convolution and the cover product in *integer max-sum semiring*, i.e., the semiring $(\mathbb{Z} \cup \{-\infty\}, \max, +)$.

Note that in the previous sections we were working in the $(\mathbb{Z}, +, \cdot)$ ring,[1] and in fact the fast cover product algorithm from Theorem 10.14 and the fast subset convolution algorithm from Theorem 10.15 are correct for any ring. These algorithms rely on the fast zeta and Möbius transforms from Theorem 10.12. We encourage the reader to verify that the fast zeta transform algorithm works perfectly fine in a semiring; however in the definition of the Möbius transform the additive inverse is used, hence the Möbius transform cannot even be defined in a semiring. Hence we cannot apply the fast cover product and the fast subset convolution algorithms in the $(\mathbb{Z} \cup \{+\infty\}, \min, +)$ semiring, since integers do not have inverses under the min operation. It is unclear whether we can extend these algorithms to semirings. However, if f and g have small integer values we can use a simple embedding trick to get the following result.

Theorem 10.17. *For two functions $f, g \colon 2^V \to \{-M, \dots, M\}$, given all the 2^n values of f and g in the input, all the 2^n values of the subset convolution of f and g over the integer min-sum (or max-sum) semiring can be computed in time $2^n n^{\mathcal{O}(1)} \cdot \mathcal{O}(M \log M \log \log M)$. The same holds for the cover product.*

Proof. Let $\beta = 2^n + 1$. We define two functions $f', g' \colon 2^V \to \{0, \dots, \beta^{2M}\}$ by putting $f'(X) = \beta^{M+f(X)}$ and $g'(X) = \beta^{M+g(X)}$. Then,

[1] We would like to emphasize that in this section the role of the + operation changes: in the min-sum semiring, + plays the role of the multiplicative operation.

$$\begin{aligned}(f' * g')(Y) &= \sum_{X \subseteq Y} f'(X) g'(Y \setminus X) \\ &= \sum_{X \subseteq Y} \beta^{2M + f(X) + g(Y \setminus X)} = \sum_{i=-2M}^{2M} \alpha_i \beta^{2M+i},\end{aligned}$$

where α_i is the number of sets X such that $f(X) + g(Y \setminus X) = i$. Note that $\alpha_i \in \{0, \ldots, 2^{|Y|}\}$ and $2^{|Y|} \leq 2^n < \beta$, hence the values of α_i can be extracted from the value of $(f' * g')(Y)$, as we can interpret the α_i's as digits in base-β notation. Then, $(f * g)(Y) = \min\{i \ : \ \alpha_i > 0\}$.

It follows that $f * g$ over the min-sum semiring can be obtained from $f' * g'$ computed over the standard integer ring. By Theorem 10.15 this takes $\mathcal{O}(2^n \cdot n^2)$ arithmetic operations on $\mathcal{O}(Mn)$-bit integers, and each of them takes time $\mathcal{O}(Mn \log(Mn) \log\log(Mn))$ if the Schönhage-Strassen multiplication algorithm [410] is used. The proof for the cover product is analogous, and it is left as Exercise 10.12. $\square$

Let us conclude this section with a simple application (see Section 11.1.2 for another one). Consider a variant of the Chromatic Number problem, where we are given an undirected graph G, an integer k and a cost function $c\colon V(G) \times [k] \to \{-M, \ldots, M\}$. The cost of a coloring $\chi\colon V(G) \to [k]$ is defined as $\sum_{v \in V(G)} c(v, \chi(v))$. In other words, coloring a vertex $v \in V(G)$ with a color $i \in [k]$ has cost $c(v, i)$. We want to determine the minimum cost of a k-coloring of G (provided that such colorings exist). Similarly as in Section 10.3.1, for every color $i \in [k]$ let us define a function $s_i\colon 2^{V(G)} \to \mathbb{Z} \cup \{+\infty\}$ such that for every $X \subseteq V(G)$ we have

$$s_i(X) = \begin{cases} \sum_{x \in X} c(x, i) & \text{when } X \text{ is an independent set,} \\ +\infty & \text{otherwise.} \end{cases}$$

Then,

$$(s_1 * s_2 * \cdots * s_k)(X)$$

evaluated in the min-sum semiring is the minimum cost of a k-coloring of $G[X]$. All these costs for all subsets $X \subseteq V(G)$ can be found in time $2^n n^{\mathcal{O}(1)} M \log M \log\log M$ by applying Theorem 10.17 $k - 1$ times (or even $\mathcal{O}(\log k)$ times, as noted in Section 10.3.1). We have proved the following theorem.

Theorem 10.18. *Given a graph G, an integer k and a cost function $c\colon V(G) \times [k] \to \{-M, \ldots, M\}$ one can compute in time $2^n n^{\mathcal{O}(1)} M \log M \log\log M$ the minimum cost of a k-coloring of every induced subgraph of G.*

10.4 Multivariate polynomials

In this section, we are going to revisit the Longest Path problem introduced in Section 5.2. Using randomization in a different way and with the help of algebra we will get a significantly faster FPT algorithm. It is convenient here to abuse the standard notation and by k-path we mean a path on k vertices, while k-walk is a sequence of vertices $v_1, \ldots, v_k$ such that neighboring vertices in the sequence are adjacent. Let us recall that in the Longest Path problem, given a graph G and an integer k, we ask whether there is a k-path in G. We begin with recalling two tools that play a crucial role in this approach.

Our first tool is the use of finite fields. Let us recall that a field is a triple $(F, +, \cdot)$, where F is a set and $+$ and $\cdot$ are binary operations called addition and multiplication. Both operations are associative and commutative and multiplication distributes over addition. There are two elements $0, 1 \in F$ such that 0 is the additive identity and 1 is the multiplicative identity. Finally, every element of F has an additive inverse, i.e., for every $a \in F$ there is an element $b \in F$ such that $a + b = 0$; and every element of $F \setminus \{0\}$ has a multiplicative inverse, i.e., for every $a \in F \setminus \{0\}$ there is an element $b \in F$ such that $a \cdot b = 1$. The most common examples are the fields of rational, real and complex numbers. However, here we focus on finite fields, in particular, finite fields of size 2^s for some integer s.

It is known that for every prime p and integer $s \geq 1$, there is a unique finite field (up to isomorphism) of size p^s. Finite fields of size p for a prime p are easy to construct: we can simply take the set $\{0, 1, \ldots, p-1\}$, with addition and multiplication modulo p. Constructing fields of size p^s for $s \geq 2$ is more involved and describing these fields in detail is beyond the scope of this book. However, here we need to know just two basic facts about fields of size 2^s:

- We can represent elements of a field of size 2^s as bit vectors of length s and perform field operations (addition, subtraction, multiplication, division) on these bit vectors fast, i.e., in time $\mathcal{O}(s \log s \log \log s)$.
- These fields are of *characteristic* 2, i.e., $1 + 1 = 0$. In particular, for any element a, we have $a + a = a \cdot (1 + 1) = a \cdot 0 = 0$.

The simplest field of characteristic 2 is the familiar GF(2), which corresponds to the set $\{0, 1\}$ with addition and multiplication modulo 2. In general, if there is a field of size $q \in \mathbb{N}$, then there is exactly one such field (up to isomorphism) and it is denoted by GF(q).

We will be working with *multivariate polynomials* such as

$$c_1 x_1^2 x_3 + c_2 x_2^8 x_3^3 x_4 x_5 + c_3,$$

interpreted over a finite field F; here c_1, c_2, c_3 are constants from the field F. Formally, recall that a (multivariate) *monomial* is an expression of the form $m = ax_1^{c_1}x_2^{c_2}\cdots x_n^{c_n}$, where a is an element of a ring F (in our case it will be a finite field), $x_1, \ldots, x_n$ are variables and $c_1, \ldots, c_n$ are nonnegative integers. Note that the monomial m is a formal object represented by the sequence $(a, c_1, \ldots, c_n)$. The *degree* of m is defined as $\sum_{i=1}^n c_n$. Monomials with the same power sequences $(c_1, \ldots, c_n)$ can be added using the distributive law, i.e., $ax_1^{c_1}x_2^{c_2}\cdots x_n^{c_n} + bx_1^{c_1}x_2^{c_2}\cdots x_n^{c_n} = (a+b)x_1^{c_1}x_2^{c_2}\cdots x_n^{c_n}$.

A *polynomial* is a finite sum of monomials, i.e., an expression of the form

$$p = \sum_{(c_1,\ldots,c_n)\in(\mathbb{N}\cup\{0\})^n} a_{c_1,\ldots,c_n} x_1^{c_1}x_2^{c_2}\cdots x_n^{c_n}, \tag{10.4}$$

where the coefficients $a_{c_1,\ldots,c_n}$ are nonzero only for a finite number of tuples $(c_1, \ldots, c_n)$. We can then define addition and multiplication on these formal objects in the usual way. In such a way we get a ring, denoted by $F[x_1, \ldots, x_n]$. This ring has the (additive) zero element, namely the element where *all* the coefficients $a_{c_1,\ldots,c_n}$ are equal to 0. The corresponding polynomial is called the *zero polynomial*, or the *identically zero* polynomial. The degree of the polynomial p is defined as the maximum degree of its monomials in the expansion (10.4), i.e., $\max\{\sum_{i=1}^n c_i \ : \ a_{c_1,\ldots,c_n} \neq 0\}$; the zero polynomial has no degree.

As described above, a polynomial, which is a formal object, can be represented as a set of tuples of the form $(a, c_1, \ldots, c_n)$. We can also represent a polynomial as an arithmetic circuit, with addition and multiplication gates, where the input gates correspond to variables and coefficients. The circuit corresponds to an expression which, by applying the associativity of addition and distributivity of multiplication over addition, can be reduced to an equivalent expanded expression which is a sum of monomials, like (10.4). However, note that the size of the expansion can be exponential in the size of the original expression/circuit, e.g., the expansion of the expression $\prod_{i=1}^n (x_i + 1)$ has 2^n monomials. This is one of the crucial properties we exploit here: in the polynomials we are going to use, the set of monomials corresponds to a huge search space (e.g., the set of all k-vertex paths), but the whole polynomial can be represented by a much smaller circuit.

A polynomial p on variables $x_1, \ldots, x_n$ can be evaluated in a given tuple $(\tilde{x}_1, \ldots, \tilde{x}_n)$ of elements from the field F. By $p(\tilde{x}_1, \ldots, \tilde{x}_n)$ we denote the value of an expression representing p where for every $i \in \{1, \ldots, n\}$ the value $\tilde{x}_i$ substitutes the variable x_i. Note that the arithmetic circuit that represents the polynomial corresponds to an algorithm that computes the evaluation (and the running time of the algorithm is linear in the number of gates of the circuit). A polynomial p defines a function $f_p\colon F^n \to F$ in a natural way, i.e., $f_p(\tilde{x}_1, \ldots, \tilde{x}_n) = p(\tilde{x}_1, \ldots, \tilde{x}_n)$. It is however important to note that if a function f corresponding to a polynomial in $F[x_1, \ldots, x_n]$ evaluates to 0 for every argument (i.e., is the zero function), it does not imply that the polynomial is the identically zero polynomial (while the converse obviously holds). Indeed, consider for example $F = \mathrm{GF}(2)$ (with addition and multiplication modulo 2) and the polynomial $x + x^2$ (see also Exercise 10.14). Nevertheless, we note that for every function $f\colon F^n \to F$ there is at most one polynomial p of degree smaller than the size of the field F such that for every argument $(\tilde{x}_1, \ldots, \tilde{x}_n) \in F^n$ we have $f(\tilde{x}_1, \ldots, \tilde{x}_n) = p(\tilde{x}_1, \ldots, \tilde{x}_n)$ (we ask the reader to prove a slightly more general version of this statement in Exercise 10.14).

Our second tool is the following claim, called the Schwartz-Zippel lemma. It is well known that a nonzero univariate polynomial of degree d over the reals has at most d roots, thus the roots are indeed very sparse. For multivariate polynomials, we may have arbitrarily many roots; consider, for example, the polynomial x_1x_2. The following lemma shows (in a discrete setting) that roots of multivariate polynomials are sparse: when we randomly sample the values of the variables from a finite subset of the field, the probability of finding a root is small.

Lemma 10.19. *Let $p(x_1, x_2, \ldots, x_n) \in F[x_1, \ldots, x_n]$ be a polynomial of degree at most d over a field F, and assume p is not identically zero. Let S be a finite subset of F. Sample values $a_1, a_2, \ldots, a_n$ from S uniformly at random. Then,*

$$\Pr(p(a_1, a_2, \ldots, a_n) = 0) \le d/|S|.$$

Proof. We use induction on n. For $n = 1$, the claim follows from the well-known fact that a nonzero univariate polynomial of degree at most d over a field F has at most d roots. This can be proved by the polynomial remainder theorem, which states that a polynomial p in a single variable x has a root r if and only if polynomial $x - r$ divides p in the ring $F[x]$. This implies that a polynomial of degree at most d can have at most d different roots.

Now consider $n > 1$. Let us rewrite the polynomial p as

$$p(x_1, \ldots, x_n) = \sum_{i=0}^{d} x_1^i \cdot p_i(x_2, \ldots, x_n).$$

Let k be the largest index i such that the polynomial p_i is not identically zero; such a value k exists, since p is not identically zero. Note that p_k is of degree at most $d - k$. For sampled values of $a_2, \ldots, a_n$, define a univariate polynomial in x_1:

$$q(x_1) = \sum_{i=0}^{k} x_1^i \cdot p_i(a_2, \ldots, a_n).$$

Let $\mathcal{E}_1$ be the event "$q(a_1) = 0$" and let $\mathcal{E}_2$ be the event "$p_k(a_2, \ldots, a_n) = 0$". By $\overline{\mathcal{E}_2}$ we denote the complement of $\mathcal{E}_2$. Then,

$$\begin{aligned}\Pr(p(a_1, a_2, \ldots, a_n) = 0) &= \Pr(\mathcal{E}_1)\\ &= \Pr(\mathcal{E}_1 \cap \mathcal{E}_2) + \Pr\big(\mathcal{E}_1 \cap \overline{\mathcal{E}_2}\big).\end{aligned}$$

By the induction hypothesis, $\Pr(\mathcal{E}_1 \cap \mathcal{E}_2) \leq \Pr(\mathcal{E}_2) \leq \frac{d-k}{|S|}$. Hence it suffices to show that $\Pr\big(\mathcal{E}_1 \cap \overline{\mathcal{E}_2}\big) \leq \frac{k}{|S|}$. When $\Pr\big(\overline{\mathcal{E}_2}\big) = 0$ the claim clearly holds. Otherwise, $\Pr\big(\mathcal{E}_1 \cap \overline{\mathcal{E}_2}\big) = \Pr\big(\mathcal{E}_1 \mid \overline{\mathcal{E}_2}\big) \Pr\big(\overline{\mathcal{E}_2}\big)$. Hence it suffices to show that $\Pr\big(\mathcal{E}_1 \mid \overline{\mathcal{E}_2}\big) \leq \frac{k}{|S|}$. Indeed, the event $\mathcal{E}_2$ depends only on the values of $a_2, \ldots,$ a_n and for any fixed $a_2, \ldots, a_n$ such that $\mathcal{E}_2$ is not true, $q(x_1)$ is a univariate polynomial that is not identically zero (as x_1^k has a nonzero coefficient). This means that $q(x_1)$ is of degree exactly k. Therefore, fixing $a_2, \ldots, a_n$ such that $\mathcal{E}_2$ is not true, there are at most k values of a_1 for which $q(a_1) = 0$, hence we can bound the conditional probability $\Pr\big(\mathcal{E}_1 \mid \overline{\mathcal{E}_2}\big)$ by $\frac{k}{|S|}$. $\square$

A typical algorithmic application of Lemma 10.19 has the following scheme. We can efficiently *evaluate* a polynomial p of degree d, and we want to test whether p is a nonzero polynomial. Then, we pick S so that $|S| \geq 2d$ and we evaluate p on a random vector $\mathbf{s} \in S^n$. We answer YES if we got $p(\mathbf{s}) \neq 0$, and otherwise we answer NO. If p is the zero polynomial, then we always answer NO, and otherwise we answer YES with probability at least $\frac{1}{2}$. Hence we get a Monte Carlo algorithm with one-sided error.

For example, assume we are given two functions $f, g \colon F \to F$, for a field F. The functions are given as (arbitrarily complicated) arithmetic circuits with only addition, subtraction and multiplication gates, and we want to test efficiently whether f and g are equal. Note that f and g correspond to polynomials represented by the circuits. Hence, the function $f-g$ corresponds to a polynomial p. We can easily bound the degree of p by a maximum number d of multiplication gates on a leaf-to-root path in the circuits representing f and g. We evaluate $f - g$ in a random element of a finite subset of the field F larger than $2d$, and we report that $f = g$ if and only if we get 0. If $f = g$, then we always get 0 and the answer is correct. Otherwise, p is not the identically zero polynomial, so we answer NO with probability at least $\frac{1}{2}$. Note also that by repeating the algorithm, say $\log n$ times, we can lower the error probability to $\mathcal{O}(\frac{1}{n})$; see Chapter 5.

The problem described above is usually called POLYNOMIAL IDENTITY TESTING, and the Schwartz-Zippel lemma provides an efficient randomized algorithm solving it. Actually, no deterministic analogue of this algorithm is known, and providing such is considered a major open problem.

10.4.1 LONGEST PATH *in time* $2^k n^{\mathcal{O}(1)}$

We proceed to show the following theorem.

Theorem 10.20. *There is a one-sided error Monte Carlo algorithm with false negatives that solves the* Longest Path *problem for directed graphs in time $2^k n^{\mathcal{O}(1)}$ and polynomial space.*

The general idea is clear: we want to find a polynomial P such that P is nonzero if and only if the input graph G contains a k-path. What should P look like? Assume that for every edge e of graph G we have a variable x_e, for every vertex v of G we have a variable y_v, and P is a function of all variables in $\{x_e \ : \ e \in E(G)\} \cup \{y_v \ : \ v \in V(G)\}$. The first idea could be to have a monomial for every k-path in the graph. However, since counting k-paths is known to be $W[1]$-hard, at the first glance evaluating such a polynomial fast seems infeasible: we could evaluate this polynomial over a large field, say, the field of rationals, substituting 1 for every variable. As we will see, it is possible to define such polynomial, so that we can evaluate it fast over finite field of characteristic 2. However, to get to this point, let us try something different. Counting walks is easy, so assume that for every k-walk $v_1, e_1, v_2 \ldots, e_{k-1}, v_k$ in the graph we have the monomial $\prod_{i=1}^{k-1} x_{e_i} \prod_{i=1}^{k} y_{v_i}$. Then we can expect efficient evaluation; however, instead of just paths, our polynomial will also contain monomials for many unwanted walks. In other words, instead of checking whether the polynomial is nonzero, we need to verify whether it contains a *multilinear* monomial (i.e., a monomial where every variable has degree at most 1) — such monomials are exactly the ones that correspond to simple paths.

To overcome this difficulty, we extend our polynomial by considering *labelled walks*, i.e., we introduce a monomial for every pair (W, ℓ), where $W = v_1, \ldots, v_k$ is a k-walk and $\ell\colon [k] \to [k]$ is a bijection. Thus, every vertex v_i gets a unique label (more precisely, if W visits a vertex v several times, then every *visit* of v gets a unique label). It turns out that the monomials corresponding to the non-path walks cancel out over fields of characteristic 2. We now proceed to formal argumentation.

Our final polynomial P has variables of two kinds. For every edge $(u, v) \in E(G)$ there is a variable $x_{u,v}$ and for every vertex $v \in V(G)$ and a number $\ell \in [k]$ there is a variable $y_{v,\ell}$. The vectors of all x- and y- variables are denoted by $\mathbf{x}$ and $\mathbf{y}$, respectively. We define

$$P(\mathbf{x}, \mathbf{y}) = \sum_{\text{walk } W = v_1, \ldots, v_k} \ \sum_{\substack{\ell\colon [k] \to [k] \\ \ell \text{ is bijective}}} \prod_{i=1}^{k-1} x_{v_i, v_{i+1}} \prod_{i=1}^{k} y_{v_i, \ell(i)}. \tag{10.5}$$

We treat the polynomial P as a multivariate polynomial over the field $\mathrm{GF}(2^{\lceil \log(4k) \rceil})$. For labelled walk (W, ℓ), we define the monomial

$$\mathrm{mon}_{W,\ell}(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{k-1} x_{v_i, v_{i+1}} \prod_{i=1}^{k} y_{v_i, \ell(i)}.$$

Let us show the cancelling property of P.

Lemma 10.21.

$$P(\mathbf{x},\mathbf{y}) = \sum_{\text{path } W = v_1,\ldots,v_k} \; \sum_{\substack{\ell\colon [k]\to[k] \\ \ell \text{ is bijective}}} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y})$$

Proof. Let $W = v_1, \ldots, v_k$ be a walk, and let $\ell\colon [k] \to [k]$ be a bijection. Assume $v_a = v_b$ for some $a < b$, and if there are many such pairs, then take the lexicographically first one. We define $\ell'\colon [k] \to [k]$ as follows:

$$\ell'(x) = \begin{cases} \ell(b) & \text{if } x = a, \\ \ell(a) & \text{if } x = b, \\ \ell(x) & \text{otherwise.} \end{cases}$$

By the definition of ℓ', we have that $\ell(a) = \ell'(b)$ and $\ell(b) = \ell'(a)$. Since $v_a = v_b$, this means that $y_{v_a,\ell(a)} = y_{v_b,\ell'(b)}$ and $y_{v_b,\ell(b)} = y_{v_a,\ell'(a)}$. Note that $(W,\ell) \neq (W,\ell')$ since ℓ is injective. However, the monomials corresponding to (W,ℓ) and (W,ℓ') are the same:

$$\begin{aligned}
\mathrm{mon}_{W,\ell} &= \prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k} y_{v_i,\ell(i)} \\
&= \prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i\in[k]\setminus\{a,b\}} y_{v_i,\ell(i)} \cdot y_{v_a,\ell(a)} y_{v_b,\ell(b)} \\
&= \prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i\in[k]\setminus\{a,b\}} y_{v_i,\ell'(i)} \cdot y_{v_b,\ell'(b)} y_{v_a \ell'(a)} = \mathrm{mon}_{W,\ell'}.
\end{aligned}$$

By defining ℓ', we have defined a mapping that maps a pair (W,ℓ) to a pair (W,ℓ') such that the monomials $\mathrm{mon}_{W,\ell}$ and $\mathrm{mon}_{W,\ell'}$ are identical. Applying this mapping to (W,ℓ'), we get (W,ℓ) back: we again swap the values of $\ell'(a)$ and $\ell'(b)$. This means that we have paired up all the labelled non-path walks and for every such pair the two corresponding monomials are identical, so they cancel out because the field is of characteristic 2. $\square$

Now, let W be a path and let $\ell\colon [k] \to [k]$ be a bijection. Consider the monomial $\mathrm{mon}_{W,\ell} = \prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k} y_{v_i,\ell(i)}$. We see that since W is a path, from $\mathrm{mon}_{W,\ell}$ we can uniquely recover (W,ℓ) and hence there are no more labelled walks that correspond to $\mathrm{mon}_{W,\ell}$. Hence the monomial corresponding to (W,ℓ) does *not* cancel out, implying that P is a polynomial that is not identically zero. Together with Lemma 10.21, this observation implies the following.

Corollary 10.22. *P is not the identically zero polynomial if and only if the input graph G contains a k-path.*

The only missing element now is how to evaluate P at given vector $(\mathbf{x},\mathbf{y})$ efficiently. At this point we need a generalized, weighted version of the inclusion–exclusion principle. We state it below with a proof, which is almost the same as in the unweighted version (Theorem 10.1). If R is a ring and $\mathbf{w}\colon U \to R$ is a weight function then the weight of a set $X \subseteq U$ is defined as $\mathbf{w}(X) = \sum_{x\in X} \mathbf{w}(x)$.

Theorem 10.23 (Weighted inclusion–exclusion principle). *Let $A_1,\ldots,$ A_n be a family of sets of a finite set U. Denote $\bigcap_{i\in\emptyset}(U\setminus A_i) = U$. Then*

$$\mathbf{w}\left(\bigcap_{i\in[n]} A_i\right) = \sum_{X\subseteq[n]} (-1)^{|X|}\cdot \mathbf{w}\left(\bigcap_{i\in X}(U\setminus A_i)\right). \tag{10.6}$$

Proof. Consider any element $e \in U$. Let k be the number of sets A_i that contain e and let $A_{i_1},\ldots,A_{i_k}$ be these sets. Note that $e \in \bigcap_{i\in X}(U\setminus A_i)$ if and only if $X \cap \{i_1,\ldots,i_k\} = \emptyset$. Hence $\mathbf{w}(e)$ appears in the right-hand side of (10.6) multiplied by

$$\sum_{X\subseteq[n]\setminus\{i_1,\ldots,i_k\}} (-1)^{|X|} = \sum_{j=0}^{n-k}\binom{n-k}{j}(-1)^j = (1-1)^{n-k} = [k=n].$$

This coincides with the contribution of $\mathbf{w}(e)$ to the left-hand side of (10.6). □

Let us fix a walk W. Define the universe U as the set of all functions $\ell\colon [k]\to[k]$ and for every $\ell \in U$ define the weight $\mathbf{w}(\ell) = \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y})$. Finally, for $i \in \{1,\ldots,k\}$ let $A_i = \{\ell \in U \ : \ \ell^{-1}(i) \neq \emptyset\}$, that is, the set of those mappings that contain i in their images. Then $\bigcap_{i\in[k]} A_i$ is the set of all surjective mappings. A mapping $\ell\colon [k]\to[k]$ is surjective if and only if it is bijective. Therefore, we have

$$\sum_{\substack{\ell\colon [k]\to[k]\\ \ell \text{ is bijective}}} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y}) = \sum_{\substack{\ell\colon [k]\to[k]\\ \ell \text{ is surjective}}} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y}) = \mathbf{w}\left(\bigcap_{i=1}^{k} A_i\right). \tag{10.7}$$

Furthermore, by (10.6),

$$\begin{aligned}
\mathbf{w}\left(\bigcap_{i=1}^{k} A_i\right) &= \sum_{X\subseteq[k]} (-1)^{|X|}\cdot \mathbf{w}\left(\bigcap_{i\in X}(U\setminus A_i)\right) && \text{(by (10.6))}\\
&= \sum_{X\subseteq[k]} \mathbf{w}\left(\bigcap_{i\in X}(U\setminus A_i)\right) && \text{(characteristic 2)}\\
&= \sum_{X\subseteq[k]}\ \sum_{\ell\colon [k]\to[k]\setminus X} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y})\\
&= \sum_{X\subseteq[k]}\ \sum_{\ell\colon [k]\to X} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y}). && \text{(reordering terms)}
\end{aligned} \tag{10.8}$$

In the second equality, the $(-1)^{|X|}$ term disappears because $-1 = 1$ if we use a field of characteristic 2 (this is not crucial; it just simplifies the formulas a bit). In the third equality, we observe that $\bigcap_{i\in X}(U\setminus A_i)$ contains those labellings ℓ whose images are disjoint from X. Combining Lemma 10.21 with (10.7) and (10.8), we get

$$\begin{aligned}
P(\mathbf{x},\mathbf{y}) &= \sum_{\text{path } W}\ \sum_{\substack{\ell\colon [k]\to[k]\\ \ell \text{ is bijective}}} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y}) && \text{(Lemma 10.21)}\\
&= \sum_{\text{walk } W}\ \sum_{X\subseteq[k]}\ \sum_{\ell\colon [k]\to X} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y}) && \text{(by (10.7) and (10.8))}\\
&= \sum_{X\subseteq[k]}\ \sum_{\text{walk } W}\ \sum_{\ell\colon [k]\to X} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y}). && \text{(reordering terms)}
\end{aligned} \tag{10.9}$$

For a fixed $X \subseteq [k]$, let us denote $P_X(\mathbf{x},\mathbf{y}) = \sum_{\text{walk } W}\sum_{\ell\colon [k]\to X} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y})$. From (10.9), it now follows that evaluating P boils down to 2^k evaluations of polynomials P_X. As the following lemma shows, this is a standard dynamic-programming exercise. Note that the dynamic programming works in polynomial time unlike many other dynamic-programming algorithms in this chapter.

Lemma 10.24. *Let $X \subseteq [k]$. The polynomial P_X can be evaluated using $\mathcal{O}(km)$ field operations.*

Proof. Using dynamic programming, we fill a two-dimensional table T, defined as follows

$$T[v,d] = \sum_{\substack{\text{walk } W = v_1,\ldots,v_d\\ v_1 = v}}\ \sum_{\ell\colon [d]\to X}\ \prod_{i=1}^{d-1} x_{v_i,v_{i+1}} \prod_{i=1}^{d} y_{v_i,\ell(i)}.$$

Then,

$$T[v,d] = \begin{cases} \sum_{\ell \in X} y_{v,\ell} & \text{when } d = 1, \\ \sum_{\ell \in X} y_{v,\ell} \sum_{(v,w)\in E(G)} x_{v,w} \cdot T[w, d-1] & \text{otherwise.} \end{cases}$$

In the above recurrence we check all the possibilities for the label of v and for the second vertex w on the walk. Hence, $P_X(\mathbf{x},\mathbf{y}) = \sum_{s\in V(G)} T[s,k]$ can be evaluated using $\mathcal{O}(km)$ field operations. □

It follows that the value of P for a given assignment of its variables can be evaluated using $\mathcal{O}(2^k km)$ field operations. The algorithm now proceeds as follows. We evaluate P over a vector of random elements from the field of size $2^{\lceil \log(4k) \rceil} \geq 4k$, and we return NO if we get 0; otherwise we return YES. Since P is of degree at most $2k-1$, the Schwartz-Zippel lemma tells us that if there is a k-path, i.e., P is nonzero, then the evaluation returns zero (and we report a wrong answer) with probability at most $1/2$. This proves Theorem 10.20.

Let us stop for a while and think about the connections between the $2^k n^{\mathcal{O}(1)}$ algorithm above and the $(2e)^k n^{\mathcal{O}(1)}$ color coding-based algorithm from Section 5.2. In both approaches, we are fighting against collisions: we want to consider only those walks where every vertex appears at most once. In the color coding approach, we guess a coloring of vertices in k colors, hoping that vertices of a k-path get k different colors. Testing whether such a colorful path exists was done using dynamic programming over all subsets of the set of colors. However, it can be also done in polynomial space using the inclusion–exclusion principle — this is the topic of Exercise 10.17. If the reader solves this exercise, then it will turn out that for every set of colors X, it requires counting the k-walks that have vertices colored with colors of $[k] \setminus X$ only. This in turn can be done by dynamic programming very similar to the one that was used to evaluate polynomial P_X, where labels correspond to colors. It seems that, thanks to the algebraic tools, we were able to get rid of the need for sampling the required coloring (or, in a way, we iterated over the whole probabilistic space almost for free). Indeed, there are more examples of problems that were first solved by color coding, and then faster algorithms with the techniques from this chapter were developed (see Exercises 10.19 and 10.20).

Finally, we remark that the use of the inclusion–exclusion formula for getting rid of the bijectivity requirement is not the only way to obtain the running time of $2^k n^{\mathcal{O}(1)}$. Perhaps a simpler solution is to use dynamic programming, and compute the entries of the following table, for every subset $X \subseteq [k]$ and vertex $v \in V(G)$.

$$T[X,v] = \sum_{\substack{\text{walk } W = v_1,\ldots,v_{|X|} \\ v_1 = v}} \sum_{\substack{\ell\colon [|X|]\to[k] \\ \ell([k])=X}} \prod_{i=1}^{|X|-1} x_{v_i,v_{i+1}} \prod_{i=1}^{|X|} y_{v_i,\ell(i)}.$$

(The reader can notice yet another connection with color coding here; indeed, this dynamic programming resembles determining the existence of a colorful path in Lemma 5.5.) Notice that after computing table T the evaluation of polynomial P is obtained easily using the formula $P(\mathbf{x},\mathbf{y}) = \sum_{v\in V(G)} T[[k],v]$. However, such an algorithm uses exponential space.

10.4.2 Longest Path *in time* $2^{k/2}n^{\mathcal{O}(1)}$ *for undirected bipartite graphs*

In this section we deal with an *undirected* bipartite graph G with bipartition $V(G) = V_1 \cup V_2$. Our goal is to accelerate the algorithm from the previous section to time $2^{k/2}n^{\mathcal{O}(1)} = 1.42^k n^{\mathcal{O}(1)}$. For simplicity, assume that k is even. Recall that the exponential factor in the running time of that algorithm depends on the number of *labels* used only. Can we use just $k/2$ labels? An obvious attempt in this direction is to label only those vertices of the walks that are in V_1. This simple idea *almost* works. Our new polynomial would be the following:

$$Q_0(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W = v_1,\ldots,v_k \\ v_1\in V_1}} \sum_{\substack{\ell\colon [k/2]\to[k/2] \\ \ell \text{ is bijective}}} \prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k/2} y_{v_{2i-1},\ell(i)}. \quad (10.10)$$

Again, let us denote $\mathrm{mon}_{W,\ell} = \prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k/2} y_{v_{2i-1},\ell(i)}$.

There are *three* differences between the above polynomial and the polynomial of (10.5). First, a minor one, is that we sum over walks that begin in V_1. Since G is bipartite and k is even, every path on k vertices has one endpoint in V_1 and the other in V_2, so we do not lose any path. Second, the labelling bijection has domain/co-domain of size $k/2$. The meaning of this is that for a pair (W,ℓ), for every $i \in \{1,\ldots,k/2\}$, the vertex v_{2i-1} gets label $\ell(i)$. Note that all these vertices lie in V_1, while the other vertices of the walk are in V_2. Third, since now we are working with undirected graphs, the edge variables are indexed with *unordered* pairs of vertices (so $x_{uv} = x_{vu}$).

Observe that, as before, for a *path* W and bijection $\ell\colon [k/2] \to [k/2]$, we can uniquely recover (W,ℓ) from $\mathrm{mon}_{W,\ell}$, hence the monomial corresponding to (W,ℓ) does *not* cancel out. It remains to show that non-path walks do cancel out. Consider a non-path labelled walk (W,ℓ), $W = v_1,\ldots,v_k$. Since

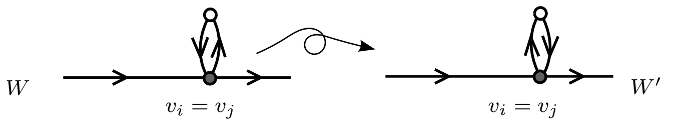


Fig. 10.2: The problematic case. The white vertex belongs to V_1 and the grey vertex belongs to V_2

W is not a path, there exist some pairs (i, j) such that $i < j$ and $v_i = v_j$. Let us call such pairs *bad*. There are two cases to verify.

Case 1. There exists a bad pair (i, j) such that $v_i = v_j \in V_1$. If there are several such bad pairs (i, j), we pick the lexicographically first one. Note that both v_i and v_j have labels, so we can get a different labelling ℓ' by swapping their labels, i.e., swapping the values of $\ell(\frac{i+1}{2})$ and $\ell(\frac{j+1}{2})$. The monomials $\text{mon}(W, \ell)$ and $\text{mon}(W, \ell')$ are exactly in the same so we can proceed as in Lemma 10.21.

Case 2. For all the bad pairs (i, j), we have $v_i = v_j \in V_2$. Again we pick the lexicographically first bad pair (i, j). Now vertices v_i and v_j do not have labels, so we cannot use the swapping trick. The new idea is to use the fact that G is undirected and reverse the closed walk $v_i, v_{i+1}, \ldots, v_j$. Let W' be the new k-walk. We also reverse the order of labels on the closed walk, i.e., in the new labelling ℓ', for every $t \in \{1, \ldots, \frac{j-i}{2}\}$, we put $\ell'(\frac{i}{2} + t) = \ell(\frac{j}{2} + 1 - t)$ (so that every vertex receives the same label as before), and otherwise ℓ' is equal to ℓ. It is easy to see that $\text{mon}(W, \ell) = \text{mon}(W', \ell')$ as desired. Moreover, if we start from (W', ℓ') and follow the same way of assignment, then we get (W, ℓ) again. The last thing to check is whether we really create *pairs* in this way, i.e., whether $(W', \ell') \neq (W, \ell)$. This is true when the closed walk $v_i, v_{i+1}, \ldots, v_j$ is of length greater than 2, because then it visits V_1 at least twice and $\ell' \neq \ell$ (because ℓ is injective). However, in the only remaining case, when the closed walk is of length 2, $\ell = \ell'$. Moreover, if we reverse a closed walk of length 2 we get exactly the same walk, so $W = W'$ (see Fig. 10.2). So the case with a closed walk of length 2 is problematic.

We see that the polynomial Q_0 defined in 10.10 does not work, but it is quite close to what we need. Can we fix it? The most naive thing to do is to modify the polynomial to exclude the problematic case. Surprisingly, this works. We say that a walk $v_1, \ldots, v_k$ is *admissible*[2] when for every $i \in \{1, \ldots, k-2\}$, if $v_i \in V_2$ and $v_{i+1} \in V_1$ then $v_{i+2} \neq v_i$. Define

[2] The condition $v_{i+1} \in V_1$ seems redundant here, but we will use the same definition for general graphs in Section *10.4.3.

$$Q(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W = v_1,\ldots,v_k\\ v_1 \in V_1\\ W \text{ is admissible}}} \sum_{\substack{\ell\colon [k/2]\to[k/2]\\ \ell \text{ is bijective}}} \underbrace{\prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k/2} y_{v_{2i-1},\ell(i)}}_{\mathrm{mon}(W,\ell)} . \tag{10.11}$$

Note that for a *path* W and bijection $\ell\colon [k/2] \to [k/2]$, the monomial $\mathrm{mon}(W,\ell)$ does not cancel out as before. Also, the proof for Case 1 is valid again (we do not change the walk so it stays admissible). In Case 2, we also get $\mathrm{mon}(W,\ell) = \mathrm{mon}(W',\ell')$ and now, since W is admissible, we avoid the problematic case and $(W',\ell') \neq (W,\ell)$. However, we need also to verify whether W' is admissible. Let (i,j) be defined as in Case 2. Assume $W' = w_1,\ldots,w_k$ is not admissible, i.e., for some $t \in \{2,\ldots,k-1\}$ we have $w_{t-1} \in V_2$, $w_t \in V_1$ and $w_{t+1} = w_{t-1}$. If $t \le i-1$ or $t \ge j+1$ then the same sequence appears in W, so W is not admissible, a contradiction. Similarly, if $t \in \{i+1,\ldots,j-1\}$, then the same sequence appears in W in reversed order, contradiction again. Hence $t \in \{i,j\}$. However, since $v_t \in V_1$ then Case 1 would apply, a final contradiction.

Corollary 10.25. *Q is not the identically zero polynomial if and only if the input graph G contains a k-path.*

Now it remains to show that Q can be evaluated in time $2^{k/2}n^{\mathcal{O}(1)}$. Exactly in the same way as in Section 10.4.1, from the inclusion–exclusion principle we derive the following

$$Q(\mathbf{x},\mathbf{y}) = \sum_{X\subseteq[k/2]} \sum_{\text{admissible walk } W} \sum_{\ell\colon [k/2]\to X} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y}).$$

For a fixed $X \subseteq [k/2]$, we define polynomial

$$Q_X(\mathbf{x},\mathbf{y}) = \sum_{\text{admissible walk } W} \sum_{\ell\colon [k/2]\to X} \mathrm{mon}_{W,\ell}(\mathbf{x},\mathbf{y}).$$

For every $X \subseteq [k/2]$, the polynomial Q_X can be evaluated in polynomial time, by a standard extension of the dynamic programming described in the proof of Lemma 10.24. We leave the details for the reader as Exercise 10.21 (the crux is to keep track of not only the first vertex of the walk, but also of the second one). This finishes the proof of the following theorem.

Theorem 10.26. *There is a one-sided error Monte Carlo algorithm with false negatives which solves the* Longest Path *problem for undirected bipartite graphs in time $2^{k/2}n^{\mathcal{O}(1)} = 1.42^k n^{\mathcal{O}(1)}$ and polynomial space.*

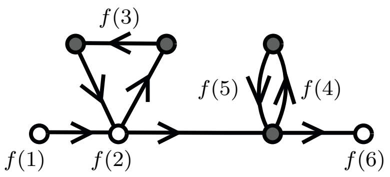


Fig. 10.3: Enumerating labelled objects in a walk by function f. The white vertices belong to V_1 and the grey vertices belong to V_2

*10.4.3 Longest Path *in time* $2^{3k/4}n^{\mathcal{O}(1)}$ *for undirected graphs*

In this section, we extend the algorithm for bipartite graphs to arbitrary (but still undirected) graphs, at the cost of worse running time of $2^{3k/4}n^{\mathcal{O}(1)} = 1.69^k n^{\mathcal{O}(1)}$. Recall that to improve over the $2^k n^{\mathcal{O}(1)}$-time algorithm it suffices to reduce the number of labels. We will mimic the bipartite case. We begin with choosing a random bipartition $V = V_1 \cup V_2$, i.e., every vertex is assigned to V_1 or V_2 with equal probability $1/2$. We use labelled walks again, but this time we label not only vertices from V_1 but also V_2V_2-edges, i.e., every edge that has both endpoints in V_2. How many labels should we use? To label a walk $v_1, \ldots, v_k$ we need:

- a different label for each $i \in \{1, \ldots, k\}$ s.t. $v_i \in V_1$, and
- a different label for each $i \in \{1, \ldots, k\}$ s.t. $v_i, v_{i+1} \in V_2$.

For every $i \in \{1, \ldots, k/2\}$, either one of $\{v_{2i-1}, v_{2i}\}$ is in V_1 or both of them are in V_2, so we need at least $k/2$ labels. However, if the walk W lies entirely in V_1 or V_2, then we need k or $k-1$ labels, and then our approach will bring us only to $2^k n^{\mathcal{O}(1)}$ running time. In order to use less labels, we will not consider all the walks. For an integer K, a walk $W = v_1, \ldots, v_k$ is *L-admissible* when it is admissible (in the same sense as in Section 10.4.2) and $|\{i : v_i \in V_1\}| + |\{i : v_iv_{i+1} \in V_2\}| = L$. Note that in an L-admissible walk, we can label all its visits of V_1-vertices and V_2V_2-edges using L different labels.

The polynomial we are about to define contains standard variables $x_{v,w}$ and $y_{e,\ell}$ that we used in the previous sections, however now in the variables $y_{e,\ell}$ the index e can be either a vertex or a V_2V_2-edge. There are also new variables z_u for each $u \in V(G)$. Likewise, the vector of variables z_u shall be denoted by $\mathbf{z}$. The reason we add these variables shall be explained later. Now, for every L, let us define a polynomial

$$R_L(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \sum_{\substack{\text{walk } W = v_1, \ldots, v_k \\ W \text{ is } L\text{-admissible}}} \sum_{\substack{\ell\colon [L] \to [L] \\ \ell \text{ is bijective}}} z_{v_1} \prod_{i=1}^{k-1} x_{v_i v_{i+1}} \prod_{i=1}^{L} y_{f_W(i), \ell(i)},$$

where $f_W(i)$ is the i-th labelled object (V_1-vertex or V_2V_2-edge) in walk W, in the order of visiting vertices/edges by the walk (see Fig. 10.3). Again, the term $z_{v_1}\prod_{i=1}^{k-1} x_{v_i v_{i+1}} \prod_{i=1}^{L} y_{f_W(i),\ell(i)}$ corresponding to one labelled walk will be denoted by $\mathrm{mon}(W,\ell)$.

The final polynomial for the non-bipartite case is defined as follows:

$$R = \sum_{L=k/2}^{\lceil \frac{3}{4}k \rceil} R_L.$$

Note that we use $\lceil \frac{3}{4}k \rceil$ labels only. However, can we really ignore all the walks that are not ($\leq \lceil \frac{3}{4}k \rceil$)-admissible? Note that if there is a k-path in G, but all the k-paths are not ($\leq \lceil \frac{3}{4}k \rceil$)-admissible, the polynomial R can be the zero polynomial, so the analogue of Corollary 10.22 *does not hold!* Nevertheless, recall that whether a path is L-admissible or not depends on the chosen random bipartition of $V(G)$, and if there is a k-path we are allowed to make a mistake with a small probability. Fix a k-path $P = v_1, \ldots, v_k$. For the random choice of bipartition (V_1, V_2), consider a random variable

$$L(P) = |\{i \; : \; v_i \in V_1\}| + |\{i \; : \; v_i v_{i+1} \in V_2\}|.$$

By the linearity of expectation we infer

$$\mathrm{E}(L(P)) = \frac{k}{2} + \frac{k-1}{4} = \frac{3k-1}{4}.$$

It follows that *sometimes* P should be ($\leq \lceil \frac{3}{4}k \rceil$)-admissible. We can bound it more precisely using Markov's inequality, which states that $\Pr(X \geq a) \leq \mathrm{E}(X)/a$ holds for every nonnegative random variable X and positive a.

$$\Pr\big(P \text{ is not } L\text{-admissible for all } L \leq \lceil \tfrac{3}{4}k \rceil\big) \leq \frac{(3k-1)/4}{\lceil \frac{3}{4}k \rceil + 1} = 1 - \Omega(k^{-1}).$$

Moreover, it is easy to check that for any bijection $\ell\colon [L(P)] \to [L(P)]$, from $\mathrm{mon}(P,\ell)$ we can uniquely recover P and ℓ. Note that here we need the new z_{v_1} factor, because without it the monomial $\mathrm{mon}(P,\ell)$ corresponds to *two* labelled walks: one which traverses P from v_1 to v_k and another which goes in the other direction. Hence the analogue of the '$\Leftarrow$' implication of Corollary 10.22 is as follows.

Corollary 10.27. *If the input graph G contains a k-path, then the polynomial R is not identically zero with probability $\Omega(k^{-1})$.*

Although the success probability may seem to be small, observe that, as our algorithm has one-sided error, it is enough to repeat it $k \log n$ times and the failure probability drops to $(1 - \Omega(k^{-1}))^{k \log n} = (e^{-\Omega(k^{-1})})^{k \log n} = e^{-\Omega(\log n)} = 1/n^{\Omega(1)}$.

Now we proceed to show that if the polynomial R is nonzero, then G contains a k-path. In other words, we show that non-path walks cancel out. Consider a non-path labelled L-admissible walk (W, ℓ), where $W = v_1, \ldots, v_k$ and $L \leq \lceil \frac{3}{4}k \rceil$. The analysis of Case 1 from the bipartite case is still valid. In Case 2 (i.e., whenever $v_i = v_j$ then $i = j$ or $v_i \in V_2$), we choose the lexicographically first pair (i, j) such that $v_i = v_j$ (hence $v_i \in V_2$). We pair up the labeled walk (W, ℓ) with a new labelled walk (W', ℓ'), which is obtained from (W, ℓ) by reversing the closed walk $v_i, v_{i+1}, \ldots, v_j$, and also reversing the order of labels on this closed walk. The same argumentation as in the bipartite case shows that W' is L-admissible, and that (W', ℓ') is assigned back to (W, ℓ).

By the definition of (W', ℓ'), we have that $\{(f_W(i), \ell(i)) \ : \ i = 1, \ldots, L\} = \{(f_{W'}(i), \ell'(i)) \ : \ i = 1, \ldots, L\}$. It follows that $\text{mon}(W, \ell) = \text{mon}(W', \ell')$. Let us show that $(W', \ell') \neq (W, \ell)$. Assume $W = W'$. This means that $v_i, v_{i+1}, \ldots, v_j = v_i, v_{j-1}, \ldots, v_i$. In other words, if we put $t = (i+j)/2$, then our closed walk follows a walk from v_i to v_t and back:

$$v_i, v_{i+1}, \ldots, v_j = v_i, v_{i+1}, \ldots, v_{t-1}, v_t, v_{t-1}, \ldots, v_{i+1}, v_i.$$

Note that $v_{i+1}, \ldots, v_{t-1} \in V_2$ for otherwise Case 1 applies. Moreover, $v_t \in V_2$, for otherwise the closed walk $v_{t-1}v_tv_{t+1}$ makes W not admissible, a contradiction. Hence, the closed walk being reversed contains at least two V_2V_2-edges. We reverse the order of labels assigned to these edges, so $\ell' \neq \ell$. Thus we have shown that $(W', \ell') \neq (W, \ell)$. Moreover, if we follow the same way of assignment we get (W, ℓ) again, so just as in the previous sections we partitioned all the non-path labelled walks into pairs with the same monomials. This means that the monomials cancel out as required.

Now it remains to show that for every $i = k/2, \ldots, \lceil 3k/4 \rceil$, the polynomial R_i can be evaluated in time $2^i n^{\mathcal{O}(1)}$. As in the bipartite case, using inclusion–exclusion this reduces to a standard exercise in dynamic programming. To enumerate the walks W with $L(W) = i$ we additionally keep track of the number of V_1-vertices and V_2V_2 edges. The details of this simple adjustment are left to the reader. This proves the following.

Theorem 10.28. *There is a one-sided error Monte Carlo algorithm with false negatives which solves the* Longest Path *problem for undirected graphs in time* $2^{3k/4}n^{\mathcal{O}(1)} = 1.69^k n^{\mathcal{O}(1)}$ *and polynomial space.*

Let us finally make a note on the derandomization of the algorithms for Longest Path discussed in this chapter. In Section 5.6, we explain how randomized algorithms based on color coding can be derandomized by a small additive cost $\mathcal{O}(\log^2 k)$ in the exponent. It is not clear at all if a similar result can be claimed for algorithms from this chapter based on monomial testing. We will explain in Chapter 12 (Theorem 12.37) how techniques based on representative sets of matroids can be used to obtain a deterministic $2.619^k n^{\mathcal{O}(1)}$-time algorithm for Longest Path.

Exercises

10.1 (🍃). Derive Theorem 10.2 from Theorem 10.1 using De Morgan's laws.

10.2 (🍃). Let A be an adjacency matrix of a directed graph G over the vertex set $[n]$, i.e., $a_{ij} = [(i,j) \in E(G)]$ for $i,j = 1,\ldots,n$. For a positive integer k, let $M = A^k$. Show that for every $i,j = 1,\ldots,n$, the number of length k walks from i to j is equal to m_{ij}.

10.3 (🍃). Prove Observation 10.4.

10.4 (🍃). Show that the number of length ℓ branching walks in an n-vertex graph can be represented using $\mathcal{O}(\ell \log n)$ bits.

10.5. Knowing that counting k-vertex paths in FPT time implies that FPT = W[1] by a result of Flum and Grohe [188], show that it is impossible to count solutions for the Steiner Tree problem, unless FPT = W[1].

10.6. The algorithms for Hamiltonian Cycle and Steiner Tree tree presented in this chapter suffer a multiplicative $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$ overhead in the running time because of performing arithmetic operations on $\mathcal{O}(n \log n)$-bit numbers. Show that one can shave this overhead down to $\mathcal{O}(\log n)$ at the cost of getting a one-sided error Monte Carlo algorithm with false negatives occurring with probability $\mathcal{O}(n^{-1})$.

10.7. In the TSP problem, we are given a complete graph with a weight function $w : V^2 \to \{0,\ldots,W\}$ and the goal is to find a Hamiltonian cycle H of the smallest weight (i.e., $\sum_{uv \in E(H)} w(u,v)$). Describe a $2^n n^{\mathcal{O}(1)} \cdot W$-time, $Wn^{\mathcal{O}(1)}$-space algorithm for the TSP problem.

10.8 (🍃). In the List Coloring problem, we are given a graph G and for each vertex $v \in V(G)$ there is a set (also called a list) of admissible colors $L(v) \subseteq \mathbb{N}$. The goal is to verify whether it is possible to find a proper vertex coloring $c : V(G) \to \mathbb{N}$ of G such that for every vertex v we have $c(v) \in L(v)$. In other words, $L(v)$ is the set of colors allowed for v.

Show a $2^n n^{\mathcal{O}(1)}$-time algorithm for List Coloring.

10.9. Show an algorithm which computes the number of perfect matchings in a given n-vertex *bipartite* graph in $2^{n/2} n^{\mathcal{O}(1)}$ time and polynomial space. (The solution is called Ryser's Formula.)

10.10 (🍃). Prove Proposition 10.10.

10.11 (🍃). Show that $(\mathbb{Z} \cup \{+\infty\}, \min, +)$ is a semiring.

10.12 (🍃). Modify the proof of Theorem 10.17 to get an algorithm for cover product in min-sum semirings.

10.13. The *packing product* of two functions $f,g : 2^V \to \mathbb{Z}$ is a function $(f *_p g) : 2^V \to \mathbb{Z}$ such that for every $Y \subseteq V$,

$$(f *_p g)(Y) = \sum_{\substack{A,B \subseteq Y \\ A \cap B = \emptyset}} f(A)g(B).$$

Show that all the $2^{|V|}$ values of $f *_p g$ can be computed in $2^n n^{\mathcal{O}(1)}$ time, where $n = |V|$.

10.14. Let p be a polynomial in n variables over the finite field $\mathrm{GF}(q)$, such that the degree of p in each variable (i.e., the maximum exponent of a variable) is strictly less than q. Show that if p evaluates to zero for every argument (i.e., p corresponds to the zero function), then all the coefficients of p are zero (p is the identically zero polynomial).

10.15. Let f be a function $f : \mathrm{GF}(q)^n \to \mathrm{GF}(q)$ that corresponds to a polynomial (possibly of large degree). Show that there is exactly one polynomial p that evaluates to $f(x_1, \dots, x_n)$ for every tuple $(x_1, \dots, x_n) \in \mathrm{GF}(q)^n$ such that the degree of p in each variable (i.e., the maximum exponent of a variable) is strictly less than q.

10.16. In the $2^k n^{\mathcal{O}(1)}$-time algorithm for Longest Path, we introduced labels so that the non-path walks pair up and hence cancel out. This resulted in $\mathcal{O}(2^k)$ overhead in the running time. In the $2^{k/2} n^{\mathcal{O}(1)}$-time algorithm for bipartite graphs we used a different argument for pairing-up non-path walks, by reversing closed subwalks. Does it mean that we can skip the labels then? It seems we cannot, because then we solve Longest Path in polynomial time! Explain what goes wrong.

10.17. Given a graph G and a coloring $c : V(G) \to [k]$, verify in $\mathcal{O}(2^k \operatorname{poly}(|V|))$ time *and polynomial space* whether G contains a k-path with vertices colored with all k colors.

10.18. In the Weighted Longest Path problem, we are given a directed graph G with a weight function $w : E(G) \to \{0, \dots, W\}$ and the goal is to find a k-path P of the smallest weight (i.e., $\sum_{uv \in E(P)} w(u, v)$). Describe a $2^k \cdot W \cdot n^{\mathcal{O}(1)}$-time polynomial space Monte Carlo algorithm for this problem.

10.19. Describe a Monte Carlo $2^{3k} n^{\mathcal{O}(1)}$-time polynomial-space algorithm for the Triangle Packing problem: given a graph G and a number $k \in \mathbb{N}$, decide whether G contains k disjoint triangles (subgraphs isomorphic to K_3).

10.20. In the Colorful Graph Motif problem we are given a graph G, a coloring $c : V(G) \to \mathbb{N}$, and a finite set of colors $M \subseteq \mathbb{N}$. Denote $k = |M|$. The goal is to decide whether there is a subset $S \subseteq V(G)$ of size k such that $G[S]$ is connected, and $c(S) = M$.

(a) Describe a $c^k n^{\mathcal{O}(1)}$-time polynomial-space algorithm for a constant c, ideally a $2^k n^{\mathcal{O}(1)}$-time Monte Carlo algorithm.
(b) The same, but in the case when M is a multiset (this version is called Graph Motif)? (☠)

10.21. Modify the dynamic programming from Lemma 10.24 to make it evaluate the polynomials Q_X from Section 10.4.2.

Hints

10.5 Show that by counting appropriate Steiner trees one can compute the number of k-vertex paths connecting a fixed pair of vertices.

10.8 Modify slightly the $2^n n^{\mathcal{O}(1)}$-time algorithm for the regular vertex coloring.

10.9 Inclusion–exclusion.

10.14 Use induction on the number of variables n, as in the proof of the Schwartz-Zippel lemma.

10.15 To show that there is at least one such polynomial, consider any polynomial p corresponding to f and note that if p has a monomial that contains x_i^r for some indeterminate

x_i, then by replacing x_i^r by x_i^{r-q} we get another polynomial corresponding to f. To show that there is at most one such polynomial, assume the contrary and use Exercise 10.14.

10.17 Modify the $2^k n^{\mathcal{O}(1)}$-time algorithm for Longest Path.

10.18 There are at least two approaches. Perhaps the simplest approach is to consider every possible weight $i = 0, \dots, kW$ and check if there is a k-path of weight i by focusing on walks of weight i only. Another approach could be to introduce a new variable z which will not be evaluated and the degree of z in a monomial corresponds to the weight of the relevant walk.

10.19 Use a similar approach as in the $2^k n^{\mathcal{O}(1)}$-time algorithm for k-path from this chapter.

10.20 Point (a) can be done using two approaches: reduce to Steiner Tree or proceed as in Exercise 10.17 using branching walks. To get $2^k n^{\mathcal{O}(1)}$ running time in the latter approach observe that one can use M as the set of labels. The extension to the multiset variant is just a matter of technical details in the latter approach.

10.21 Keep track not only of the first vertex of the walk, but also of the second one.

Bibliographic notes

The principle of inclusion–exclusion is one of the most basic tools in combinatorics. The books of Aigner [7], Ryser [408], and Stanley [418] contain detailed discussions of the method. The application of the inclusion–exclusion principle for Hamiltonicity was rediscovered several times. The algorithm running in time $2^n n^{\mathcal{O}(1)}$ and using polynomial space for the problem was given by Kohn, Gottlieb, and Kohn in 1969 [302]. It was rediscovered in 1982 by Karp [282], and in 1993 by Bax [27, 28]. For an overview of the area of exact exponential algorithms we refer the reader to surveys of Woeginger [436, 437] and the book of Fomin and Kratsch [197].

The notion on branching walk and the poly-space $\mathcal{O}(2^{|K|} n^{\mathcal{O}(1)})$-time algorithm for the unweighted Steiner Tree problem is due to Nederlof [374]. This algorithm can be extended to the weighted case; as in Exercise 10.7 for integer edge weights bounded by C, this comes at the cost of an $\mathcal{O}(C)$ factor in the time and space complexity. Nederlof and Lokshtanov [329] show that the space can be improved to $n^{\mathcal{O}(1)} \log C$, hence polynomial in the input size, as opposed to $n^{\mathcal{O}(1)} C$, called pseudo-polynomial (the paper contains similar consequences for other problems, like Subset Sum or Knapsack). However the running times of polynomial-space algorithms in [329] is $\mathcal{O}(2^{|K|} n^{\mathcal{O}(1)} C)$, which is not FPT. The first algorithms for graph coloring with running time $2^n n^{\mathcal{O}(1)}$ were given independently by Björklund and Husfeldt and Koivisto (see the joint journal article [40]). The Möbius inversion formula presented in this chapter can be seen as a generalization of the well-known Möbius inversion formula from number theory. The version for the subset lattice is usually attributed to Weisner [433] and Hall [255]. The fast zeta transform algorithm dates back to Yates [441] but it was first used in the context of FPT algorithms by Björklund, Husfeldt, Kaski and Koivisto [37], who presented also the algorithms for subset convolution and cover product included in this chapter and also for packing product (Exercise 10.13). The odd-negation transform is from lecture notes of Kaski [284]. The fast subset convolution algorithm from [37] is faster by a factor of $\mathcal{O}(n)$ than the one presented in this chapter. The Ryser formula (Exercise 10.9) is due to Ryser [408].

Koutis [305] was the first to observe that using algebraic methods one can enumerate all k-walks and get the non-path walks cancelled. His approach was to directly check whether a polynomial contains a multilinear monomial using group algebras. Koutis obtained a

$2^{3k/2}n^{\mathcal{O}(1)}$-time algorithm which was next improved to $2^k n^{\mathcal{O}(1)}$ by Williams [435] with similar methods. The idea of using bijective labellings to get the non-path walks cancelled was discovered by Björklund [36] as a tool for his $1.66^n n^{\mathcal{O}(1)}$-time algorithm for the Hamiltonian cycle problem in undirected graphs. It was next extended by Björklund, Husfeldt, Kaski and Koivisto [39] to a $1.66^k n^{\mathcal{O}(1)}$-time algorithm for k-path in undirected graphs and the algorithms from Section 10.4 are based on this work. The slight speed-up from $2^{3k/4}n^{\mathcal{O}(1)} = 1.682^k n^{\mathcal{O}(1)}$ to $1.66^k n^{\mathcal{O}(1)}$ is obtained by using $\lceil(\frac{3}{4} - \epsilon)k\rceil$ labels for some $\epsilon > 0$. This comes at the price of getting extremely low probability that the solution k-path is not filtered out, so the whole algorithm needs to be repeated $\alpha(\epsilon)^n$ number of times, for some function α. For the optimal choice of ϵ the overall time complexity is $1.66^k n^{\mathcal{O}(1)}$. Björklund [36] attributes this speed-up to Williams. Koutis' paper [305] contains also the first $2^{3k}n^{\mathcal{O}(1)}$-time algorithm for triangle packing (Exercise 10.19) and the currently best result is the $1.493^{3k}n^{\mathcal{O}(1)}$-time algorithm of Björklund, Husfeldt, Kaski and Koivisto [39]. The $2^k n^{\mathcal{O}(1)}$-time algorithm for the multiset version of the Graph Motif problem (Exercise 10.20) is due to Björklund, Kaski and Kowalik [41]. They also show that the existence of a $(2-\epsilon)^k n^{\mathcal{O}(1)}$-time algorithm for the Graph Motif problem implies a $(2-\epsilon')^k n^{\mathcal{O}(1)}$-time algorithm for Set Cover.

For the construction and properties of finite fields, see the textbook of Lidl and Niederreiter [321]. For a fixed prime p, multiplication in a finite field $\mathrm{GF}(p^n)$ can be performed in time $\mathcal{O}(n \log n \log\log n)$ by applying the Schönhage-Strassen algorithm [410] for integer multiplication (see [226]).

The Schwartz-Zippel lemma was found independently by DeMillo and Lipton [136], Zippel [443] and Schwartz [413] (their formulations were not completely equivalent but each of them would suffice for our needs in this chapter). We use the name Schwartz-Zippel lemma because it is well established in the literature.

Kabanets and Impagliazzo [279] show that the existence of a polynomial-time algorithm for Polynomial Identity Testing would imply arithmetic circuit lower bounds for NEXP, which would be a major breakthrough in computational complexity.

In all algorithms from Section 10.4 we could have used the isolation lemma (see Chapter 11) instead of the Schwartz-Zippel lemma, at a price of significantly higher polynomial factors in the running time. The essence of the algorithms (cancelling of objects corresponding to non-simple walks) stays the same. However, it is very enlightening to see the other approach and be able to translate the algorithms from one approach to another. We refer the reader to the paper of Fomin and Kaski [196] with an accessible exposition of a $2^k n^{\mathcal{O}(1)}$-time k-path algorithm described in the framework of the isolation lemma.

Chapter 11
Improving dynamic programming on tree decompositions

In this chapter we consider more sophisticated dynamic-programming routines for graphs of bounded treewidth, compared with the classic ones described in Chapter 7. We give examples of algorithms based on subset convolution, Cut & Count and rank-based approaches.

In Chapter 7 we have seen a few examples of straightforward dynamic programming for graphs of bounded treewidth. The idea behind the design of those algorithms was to use small vertex separators, namely the bags of the tree decomposition, to extract succinct information about partial solutions that is enough to carry on the construction of a final solution. In the case of Dominating Set (see Section 7.3.2) for each vertex it was enough to store one of its three possible states, which led to $\mathcal{O}(3^k)$ states per node of the tree decomposition. However the running time needed to perform the computation in a straightforward manner in the join nodes was $\mathcal{O}(4^k)$. In Section 11.1 we consider the #Perfect Matchings problem and again the Dominating Set problem, where the standard approach gives respectively $\mathcal{O}(3^k)$ and $\mathcal{O}(2^k)$ states per node of the tree decomposition of width k, but it is not clear how to obtain such a running time in join nodes. We show how to obtain such running times by invoking the fast subset convolution from Chapter 10.

Next, in Section 11.2 we deal with a class of problems with some kind of global requirement in the problem definition, informally called *connectivity problems.* For all of those problems the standard dynamic-programming approach gives $k^{\mathcal{O}(k)}$ states per node, whereas we would like to have only single exponential dependence on treewidth.

First, we show a randomized approach called Cut & Count, which by reduction to counting modulo 2, together with cut-based cancellation arguments, gives us Monte Carlo algorithms with $2^{\mathcal{O}(k)}$ dependence on treewidth (for the definition of Monte Carlo algorithms see Chapter 5).

In Section *11.2.2 we show how, by using tools from linear algebra, one can obtain deterministic algorithms with similar running times (though with a worse base of the exponential function). The main idea is to treat states in the dynamic programming computation as vectors in a suitably defined linear space. Even though seemingly this translation makes things more complicated, it shows that linear dependence of vectors enables us to discard some states and bound the number of considered states by $2^{\mathcal{O}(k)}$.

11.1 Applying fast subset convolution

In this section we apply fast algorithms for subset convolution developed in Chapter 10 to obtain faster dynamic programs on graphs of bounded treewidth.

11.1.1 Counting perfect matchings

Let us recall that a matching M of a graph G is perfect if all vertices of G are matched by M, i.e., every vertex of G is an endpoint of some edge from M. Consider the following #Perfect Matchings problem: given a graph G, the task is to find the number of perfect matchings in G. This problem is $\#P$-complete and thus it is unlikely to be solvable in polynomial time. We will show an efficient FPT algorithm (parameterized by treewidth) which applies the fast subset convolution described in Section 10.3.

We assume that we are given a nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of graph G of width k. For each node $t \in V(T)$ and set $S \subseteq X_t$, we compute $c[t, S]$, which is the number of matchings in the graph $G[V_t]$, the graph induced by the vertices contained in X_t and the bags below X_t, such that:

- all vertices in $V_t \setminus X_t$ are matched,
- all vertices in S are matched to vertices in $V_t \setminus X_t$, and
- all vertices in $X_t \setminus S$ are unmatched.

Note that if r is a root of T then $c[r, \emptyset]$ is the number of perfect matchings in G. Let us describe how the nodes of the tree are processed.

Leaf node. Since in a nice tree decomposition each leaf corresponds to an empty bag, the number of matchings that use no vertex equals 1, i.e., it is the empty matching. Therefore we set $c[t, \emptyset] = 1$ for every leaf t.

Introduce node. Let t' be the child of an introduce node t and $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$. Observe that the vertex v cannot be matched yet, as it is not adjacent to vertices of $V_t \setminus X_t$. Hence, for every $S \subseteq X_t$, we have $c[t, S] = c[t', S]$ if $v \notin S$ and $c[t, S] = 0$ if $v \in S$.

Forget node. Let t' be the child of a forget node t, that is, for some $w \in X_{t'}$ we have $X_t = X_{t'} \setminus \{w\}$. Then,

$$c[t, S] = c[t', S \cup \{w\}] + \sum_{v \in S \cap N_G(w)} c[t', S \setminus \{v\}],$$

where the first summand corresponds to the choice of matching w to a vertex of $V_{t'} \setminus X_{t'}$, while the second summand (i.e., the sum over $v \in S \cap N_G(w)$) corresponds to matching w to a vertex of S.

Note that processing the leaf nodes, introduce nodes and forget nodes takes $2^k \cdot k^{\mathcal{O}(1)}$ time only. The join nodes are slightly more involved.

Join node. Let t_1 and t_2 be the children of a join node t. Thus $X_t = X_{t_1} = X_{t_2}$. Note that if a vertex of X_t is matched to $V_t \setminus X_t$ then it is matched either to $V_{t_1} \setminus X_{t_1}$, or to $V_{t_2} \setminus X_{t_2}$, but not to both since these sets are disjoint. Thus,

$$c[t, S] = \sum_{A \subseteq S} c[t_1, A] \cdot c[t_2, S \setminus A].$$

The naive computation of the values of $c[t, S]$ for all $S \subseteq X_t$ using the above formula takes time proportional to $\sum_{S \subseteq X_t} 2^{|S|} = 3^{|X_t|} \leq 3^{k+1}$. However, we can treat the entries $c[t, S]$ for a fixed node t as a function $f : 2^{X_t} \to \mathbb{N}$ such that for every $S \subseteq X_t$ we have $f(S) = c[t, S]$. Similarly, define $g, h : 2^{X_t} \to \mathbb{N}$ such that for every $S \subseteq X_t$ we have $g(S) = c[t_1, S]$ and $h(S) = c[t_2, S]$. Then,

$$c[t, S] = f(S) = \sum_{A \subseteq S} g(A) \cdot h(S \setminus A).$$

In other words, computing $c[t, S]$ for all subsets S is just computing the subset convolution of g and h, which can be done in time $2^{|X_t|} \cdot |X_t|^{\mathcal{O}(1)} = 2^k \cdot k^{\mathcal{O}(1)}$ by Theorem 10.15. Let us conclude with the following theorem.

Theorem 11.1. *Let G be an n-vertex graph given together with its tree decomposition of width k. Then one can find the number of perfect matchings in G in time $2^k \cdot k^{\mathcal{O}(1)} \cdot n$.*

11.1.2 Dominating Set

Recall the $4^k \cdot k^{\mathcal{O}(1)} \cdot n$-time algorithm for the Dominating Set problem parameterized by treewidth k from Section 7.3.2. It turns out that it can be accelerated to time $3^k \cdot k^{\mathcal{O}(1)} \cdot n$ using the fast subset convolution from Section 10.3. Note that processing introduce nodes and forget nodes is already fast enough. We will show here that for a join node t the values of $c[t, f]$ for all the 3^k functions $f : X_t \to \{0, \hat{0}, 1\}$ can be computed in time $3^k k^{\mathcal{O}(1)}$. Recall that

$$c[t, f] = \min_{f_1, f_2} \{c[t_1, f_1] + c[t_2, f_2] - |f^{-1}(1)|\},$$

where t_1 and t_2 are the children of t (recall that $X_{t_1} = X_{t_2} = X_t$ in the nice tree decomposition) and the minimum is taken over all colorings $f_1, f_2 : X_t \to \{0, \hat{0}, 1\}$ consistent with f, that is,

- $f(v) = 1$ if and only if $f_1(v) = f_2(v) = 1$,
- $f(v) = 0$ if and only if $(f_1(v), f_2(v)) \in \{(\hat{0}, 0), (0, \hat{0})\}$,
- $f(v) = \hat{0}$ if and only if $f_1(v) = f_2(v) = \hat{0}$.

Note that the above three conditions are equivalent to

- $f^{-1}(1) = f_1^{-1}(1) = f_2^{-1}(1)$,
- $f^{-1}(0) = f_1^{-1}(0) \cup f_2^{-1}(0)$,
- $f_1^{-1}(0) \cap f_2^{-1}(0) = \emptyset$.

One could think that the condition $f^{-1}(\hat{0}) = f_1^{-1}(\hat{0}) \cap f_2^{-1}(\hat{0})$ is missing here; however, note that it is implied by the first two.

The above considerations suggest that if we fix $f^{-1}(1)$, then what we want to compute resembles the subset convolution. Let us follow this idea. Fix a set $R \subseteq X_t$. Let $\mathcal{F}_R$ denote the set of all functions $f : X_t \to \{0, \hat{0}, 1\}$ such that $f^{-1}(1) = R$. We want to compute the values of $c[t, f]$ for all $f \in \mathcal{F}_R$.

Note that every function $f \in \mathcal{F}_R$ can be represented by a set $S \subseteq X_t \setminus R$, namely the preimage of 0. Hence we can define the coloring represented by S as

$$g_S(x) = \begin{cases} 1 & \text{if } x \in R, \\ 0 & \text{if } x \in S, \\ \hat{0} & \text{if } x \in X_t \setminus (R \cup S). \end{cases}$$

Then for every $f \in \mathcal{F}_R$ we have

$$c[t, f] = \min_{\substack{A \cup B = f^{-1}(0) \\ A \cap B = \emptyset}} (c[t_1, g_A] + c[t_2, g_B]) - |f^{-1}(1)|.$$

For $v \in \{t_1, t_2\}$, we define functions $c_v : 2^{X_{t_i} \setminus R} \to \mathbb{N}$ such that for every $S \subseteq X_t \setminus R$ we have $c_v(S) = c[v, g_S]$. We see that for every $S \subseteq X_t \setminus R$,

$$c[t, g_S] = (c_{t_1} * c_{t_2})(S) - |R|,$$

where the subset convolution is over the min-sum semiring (see Section 10.3.2). Note that the table c stores integers bounded by the number of vertices in the graph. Hence, by Theorem 10.17, we can compute $c[t, f]$ for every $f \in \mathcal{F}_R$

in $2^{|X_t \setminus R|} \cdot n^{\mathcal{O}(1)}$ time. Since $\sum_{R \subseteq X_t} 2^{|X_t \setminus R|} = 3^{|X_t|} \leq 3^{k+1}$, the total time spent for all subsets $R \subseteq X_t$ is $3^k \cdot n^{\mathcal{O}(1)}$.

Theorem 11.2. *Let G be an n-vertex graph given together with its tree decomposition of width k. Then* Dominating Set *is solvable in time $3^k \cdot n^{\mathcal{O}(1)}$.*

In Exercise 11.3 we ask the reader to improve the dependence on n in Theorem 11.2 from polynomial to linear.

11.2 Connectivity problems

In Section 7.3.3, we gave a standard dynamic-programming algorithm solving Steiner Tree in time $k^{\mathcal{O}(k)}n$ on graphs of treewidth k. The main reason this approach suffers from the $k^{\mathcal{O}(k)}$ dependence on treewidth is that we have to keep track of partitions into connected components. In this section we show two methods that allow us to bypass the need to store all the possible partitions. In Section 11.2.1 we describe a randomized approach which reduces the decision variant of the problem to counting modulo 2, which as in Chapter 10 appears to be a fruitful approach. Next, in Section *11.2.2, we show a greedy method based on tools from linear algebra that allow us to store only a single exponential number of partitions.

11.2.1 Cut & Count

Assume that we are given a graph G, together with a set of terminals $K \subseteq V(G)$, a tree decomposition $\mathcal{T}$ of G, and an integer ℓ. We are to decide whether there exists a Steiner tree of size at most ℓ. In other words we are to check whether there is a connected subgraph of G with at most ℓ edges containing all the terminals K. Our strategy is first to solve the counting modulo 2 variant of the problem, i.e., find the parity of the number of connected subgraphs of G with at most ℓ edges spanning all the vertices of K. A priori this does not seem to be easier than solving the decision variant, but as we are about to see working modulo 2 allows cancellation tricks as in the case of finite fields of characteristic 2 in Section 10.4.

Let $\mathcal{R}$ be the set of all subgraphs of G with at most ℓ edges containing K, and let $\mathcal{S}$ be the set all solutions, i.e., all subgraphs from $\mathcal{R}$ which are connected. A formal definition follows.

$$\mathcal{R} = \{H \subseteq G \ : \ |E(H)| \leq \ell, K \subseteq V(H)\} \tag{11.1}$$
$$\mathcal{S} = \{H \in \mathcal{R} \ : \ H \text{ is connected}\} \tag{11.2}$$

By straightforward methods we can not only find the parity of $|\mathcal{R}|$ but also the value of $|\mathcal{R}|$ itself in time $2^k n^{\mathcal{O}(1)}$ (see Exercise 11.4). However this is not exactly what we are after. We would like to use some kind of overcounting argument, so that each element of $\mathcal{R} \setminus \mathcal{S}$ would be counted an even number of times, while each element of $\mathcal{S}$ would be counted an odd number of times. For this reason we define a family of cuts, which are just partitions of the set of vertices $V(H)$ into two sets V^1, V^2. We say that a subgraph H of G and a cut (V^1, V^2) are *consistent* if no edge of H has its two endpoints on two different sides of the cut, i.e., when $E(H) \subseteq \binom{V^1}{2} \cup \binom{V^2}{2}$, where $\binom{X}{2}$ denotes all size 2 subsets of X (edges with both endpoints in X).

For a subgraph H of G let us ask ourselves a question, how many cuts is the graph H consistent with? Since no edge of H can go across the cut, each of its connected components has to be either entirely in V^1, or entirely in V^2. Other than that we are free to choose the side of the cut for every connected component of H independently. This means that a subgraph H with c connected components is consistent with 2^c cuts. This is almost perfect, as all subgraphs with more than one connected component are consistent with an even number of cuts, but unfortunately a connected subgraph H is consistent with two subgraphs, instead of one subgraph as we would like it to be. For this reason we select some arbitrary terminal vertex $v_1 \in K$, and fix it permanently into the V^1 side of the cut. For a subset $W \subseteq V(G)$ and $v_1 \in K$ define

$$\texttt{cuts}(W, v_1) := \{(V^1, V^2) \ : \ V^1 \cup V^2 = W \wedge V^1 \cap V^2 = \emptyset \wedge v_1 \in V^1\}.$$

Now we show that with the updated definition of cuts instead of calculating the parity of $|\mathcal{S}|$ directly, we can count pairs consisting of a subgraph from $\mathcal{R}$ and a cut consistent with it.

$$\mathcal{C} = \{(H, (V^1, V^2)) \in \mathcal{R} \times \texttt{cuts}(V(H), v_1) \ : \ H \text{ is consistent with } (V^1, V^2)\}\,. \tag{11.3}$$

Lemma 11.3. *The parity of $|\mathcal{C}|$ is the same as the parity of $|\mathcal{S}|$.*

Proof. Consider a graph H from $\mathcal{R}$ with c connected components. Each connected component has to be contained either in V^1 or in V^2, as otherwise there would be an edge crossing the cut. However, the connected component of H containing v_1 has to be contained in the V^1-side of the cut. Consequently H is consistent with 2^{c-1} cuts, which is an odd number for $H \in \mathcal{S}$ and an even number for $H \in \mathcal{R} \setminus \mathcal{S}$. □

Therefore it remains to show how to compute the parity of $|\mathcal{C}|$ by standard methods with only $2^{\mathcal{O}(k)}$ dependence on treewidth. The key observation is that we can do that by storing local information only, that is, unlike in Section 7.3.3, we do not need partitions in our dynamic programming state anymore. For each node $t \in V(T)$, an integer $i \leq \ell$, and a function $f : X_t \to \{0,1,2\}$ we compute $c[t,f,i]$, which is the number of pairs $(H,(V^1,V^2))$, such that:

- H is a subgraph of (V_t, E_t) with exactly i edges, and H contains all the vertices of $K \cap V_t$,
- (V^1, V^2) is a cut consistent with H, i.e., $V^1 \cup V^2 = V(H)$, $V^1 \cap V^2 = \emptyset$, $v_1 \notin V^2$,
- the set of vertices of H intersecting the bag is exactly $f^{-1}(1) \cup f^{-1}(2)$, i.e., $V(H) \cap X_t = f^{-1}(1) \cup f^{-1}(2)$,
- the function f describes how vertices of $X_t \setminus f^{-1}(0)$ are assigned to V^1 and V^2, that is, $V^j \cap V(H) \cap X_t = f^{-1}(j)$ for $j \in \{1,2\}$.

We stop the description of the dynamic-programming routine here and leave the details to the reader (see Exercise 11.5), as no new ideas are needed from this point. The number of states we have defined is $3^k n^{\mathcal{O}(1)}$, and this is also the running time of our algorithm computing $|\mathcal{C}|$, which by Lemma 11.3 is sufficient to find the parity of $|\mathcal{S}|$. Consequently we established an algorithm for the counting modulo 2 variant of Steiner Tree with $2^{\mathcal{O}(k)}$ dependence on treewidth, which was our first goal.

It remains to show why solving the parity version of the problem is helpful when facing the decision version. In Chapter 10 we have already seen an example of this, when we were working with polynomials in fields of characteristic 2. Here, we use an alternative to polynomial identity testing, which is called the isolation lemma.

Consider the following experiment: we choose a large integer N and assign to every edge $e \in E(G)$ a weight $\mathbf{w}(e) \in \{1,2,\ldots,N\}$ uniformly and independently at random. For a subgraph $H \in \mathcal{R}$, we define its weight as $\mathbf{w}(H) = \sum_{e \in E(H)} \mathbf{w}(e)$. For an integer w, let $\mathcal{R}_w = \{H \in \mathcal{R} \ : \ \mathbf{w}(H) = w\}$ and similarly define $\mathcal{S}_w$ and $\mathcal{C}_w$. It is a standard task to adjust the aforementioned dynamic-programming algorithm to compute $|\mathcal{C}_w|$ for all reasonable weights w, at the cost of an additional polynomial factor in n and N in the running time bound. Intuitively, large N should ensure that in a yes-instance the random choice of weights sufficiently shuffles the solutions among the sets $\mathcal{S}_w$ such that for at least one weight w the set $\mathcal{S}_w$ is of odd size, and we can detect it by computing $|\mathcal{C}_w|$. The isolation lemma is an even stronger statement: it asserts that it suffices to take $N = \Omega(|E(G)|)$ to obtain, with constant probability, at least one set $\mathcal{S}_w$ of size *exactly* 1.

Let us now proceed with a formal statement and proof of the isolation lemma.

Definition 11.4. A function $\mathbf{w} : U \to \mathbb{Z}$ *isolates* a set family $\mathcal{F} \subseteq 2^U$ if there is a unique $S' \in \mathcal{F}$ with $\mathbf{w}(S') = \min_{S \in \mathcal{F}} \mathbf{w}(S)$.

In the above definition $\mathbf{w}(X)$ denotes $\sum_{u \in X} \mathbf{w}(u)$.

Lemma 11.5 (Isolation lemma). *Let $\mathcal{F} \subseteq 2^U$ be a set family over a universe U with $|\mathcal{F}| > 0$. For each $u \in U$, choose a weight $\mathbf{w}(u) \in \{1,2,\ldots,N\}$*

uniformly and independently at random. Then

$$\Pr(\mathbf{w} \text{ isolates } \mathcal{F}) \geq 1 - \frac{|U|}{N} .$$

Proof. Let $u \in U$ be an arbitrary element of the universe U, which appears in at least one set of $\mathcal{F}$ and does not appear in at least one set of $\mathcal{F}$ (note that such an element does not need to exist). For such u we define

$$\alpha(u) = \min_{S \in \mathcal{F}, u \notin S} \mathbf{w}(S) - \min_{S \in \mathcal{F}, u \in S} \mathbf{w}(S \setminus \{u\}) . \tag{11.4}$$

Note that $\alpha(u)$ is independent of the weight of $\mathbf{w}(u)$, as $\mathbf{w}(u)$ it not used at all at the right-hand side of (11.4). Consequently

$$\Pr(\alpha(u) = \mathbf{w}(u)) \leq 1/N, \tag{11.5}$$

since we can imagine that the weight $\mathbf{w}(u)$ is sampled in the end, when all the other weights, and $\alpha(u)$ in particular, are already known.

Assume that there are two sets $A, B \in \mathcal{F}$ of minimum weight with respect to $\mathbf{w}$ and let x be an arbitrary element of $A \setminus B$ (note that $A \setminus B \neq \emptyset$ as all the weights are positive). We have

$$\begin{aligned} \alpha(x) &= \min_{S \in \mathcal{F}, x \notin S} \mathbf{w}(S) - \min_{S \in \mathcal{F}, x \in S} \mathbf{w}(S \setminus \{x\}) \\ &= \mathbf{w}(B) - (\mathbf{w}(A) - \mathbf{w}(x)) \\ &= \mathbf{w}(x) . \end{aligned}$$

Hence if $\mathbf{w}$ does not isolate $\mathcal{F}$ then there is an element x and sets $A, B \in \mathcal{F}$ such that $x \in A \setminus B$ and $\alpha(x) = \mathbf{w}(x)$. However, by (11.5) and the union bound, this happens with probability at most $|U|/N$. □

The reason Lemma 11.5 is useful for us is that it gives a reduction from the decision version of the problem to the parity version as follows. Our universe is $U = E(G)$, and we put $\mathcal{F} = \{E(H) \ : \ H \in \mathcal{S}\}$. We sample an integral weight $\mathbf{w}(e)$ from $\{1, \ldots, 2|U|\}$ for each element of the universe $e \in U$, i.e., for each edge. For each w between 1 and $2|E|\ell$ we compute the parity of the number of pairs in $\mathcal{C}$, where the subgraph H is of weight exactly w with respect to $\mathbf{w}$, which can be easily handled by the dynamic-programming routine by adding a polynomial overhead in the running time. If for some w the parity is odd, we answer YES, otherwise we answer NO.

Even though there might be exponentially many solutions, that is elements of $\mathcal{S}$, with probability at least $1/2$ for some weight w between 1 and $2|E|\ell$ there is exactly one solution from $\mathcal{S}$ of weight w with respect to $\mathbf{w}$. As a corollary we obtain a Monte Carlo algorithm where the probability of error is at most $1/2$.

Theorem 11.6. *There is a one-sided error Monte Carlo algorithm with false negatives which solves the* Steiner Tree *problem in time $3^k n^{\mathcal{O}(1)}$ given a tree decomposition of width k.*

The described technique may be applied to several problems, with a global connectivity constraint, which we call connectivity problems. A list of such problems will be given in Theorem 11.13. The general scheme of the approach is as follows.

The Cut & Count technique may be applied to problems involving a connectivity requirement. Let $\mathcal{S} \subseteq 2^U$ be a set of solutions (usually the universe U is the set of vertices or edges/arcs of the input graph); we aim to decide whether it is empty. Conceptually, Cut & Count can naturally be split into two parts:

- **The Cut part**: Relax the connectivity requirement by considering the set $\mathcal{R} \supseteq \mathcal{S}$ of possibly disconnected candidate solutions. Furthermore, consider the set $\mathcal{C}$ of pairs (X, C) where $X \in \mathcal{R}$ and C is a consistent cut of X.
- **The Count part**: Isolate a single solution by sampling weights of all elements in U. Compute $|\mathcal{C}| \bmod 2$ using a sub-procedure. Non-connected candidate solutions $X \in \mathcal{R} \setminus \mathcal{S}$ cancel since they are consistent with an even number of cuts. The only connected candidate $x \in \mathcal{S}$ remains (if it exists).

Finally we note that not only is the general scheme of the presented technique rather simple, but it also gives small constants in the base of the exponent (e.g., see Exercises 11.6 and 11.7). As a matter of fact, often the constants in the base of the exponent given by the Cut & Count approach are optimal under the Strong Exponential Time Hypothesis (see Theorem 14.41).

*11.2.2 Deterministic algorithms by Gaussian elimination

Even though the Cut & Count technique allowed us to break the $k^{\mathcal{O}(k)}$ barrier, it is natural to ask whether one can achieve single exponential dependence on treewidth by a deterministic algorithm. Additionally the approach presented in this section allows for smaller polynomial factors in the running time, and at the same time is able to handle arbitrary real weights in optimization problems, which the Cut & Count technique inherently cannot capture. However, the drawback of the new approach is a worse base of the exponential function in the dependence on treewidth.

Recall the standard dynamic-programming approach for the Steiner Tree problem from Section 7.3.3. In this section we slightly change the definition of a state as follows. For a bag X_t, a set $X \subseteq X_t$ (a set of vertices touched by a Steiner tree), and an integer i, we define $c[t, X, i]$ as the family of all partitions $P = \{P_1, P_2, \ldots, P_q\}$ of X for which there exists a forest F in (V_t, E_t) such that

- $|E(F)| = i$, i.e., F has exactly i edges,
- F has exactly q connected components $C_1, \ldots, C_q$ and for each $s \in \{1, \ldots, q\}$, $P_s \subseteq V(C_s)$. Thus the partition P corresponds to connected components of F;
- $X_t \cap V(F) = X$, i.e., vertices in $X_t \setminus X$ are untouched by F;
- every terminal vertex from $K \cap V_t$ is in $V(F)$.

Note that there are two significant differences between the dynamic programming of this section and the one from Section 7.3.3. First, the size of the Steiner tree under construction is explicitly stored in the state under the variable i; earlier we just stored the minimum size as a table entry. Second, instead of storing a partition as a part of the state description, we store it in $c[t, X, i]$, as $c[t, X, i]$ is now a set family, not a number. It should be clear that the dynamic-programming routine from Section 7.3.3 can be adjusted to handle the new definition of a state in our table.[1]

The total space used in this algorithm is $k^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$, and the reason for this is the number of possible partitions of the set X, i.e., the set of vertices of the current bag touched by the Steiner tree under construction. Note that the number of subsets $X \subseteq X_t$ is at most 2^{k+1}, which is single exponential, hence it is enough if we could somehow consider not all the partitions but only $2^{\mathcal{O}(k)}$ of them.

Fix a subset $X \subseteq X_t$ and define $U = X$. Let $\Pi(U)$ be the set of all partitions of U. A single set within a partition is called a block. For two partitions $P, Q \in \Pi(U)$ denote by $P \sqcup Q$ the most refined partition R of U, such that every block of both P and Q is contained in a single block of R (in the partition lattice language $P \sqcup Q$ is the join of P and Q). Less formally, if we imagine two arbitrary graphs G_P, G_Q on the same vertex set U, where the family of connected components of G_P is P and the family of connected components of G_Q is Q, then $P \sqcup Q$ corresponds to the set of connected components in the union of G_P and G_Q (in particular if the union of G_P and G_Q is connected, then $P \sqcup Q = \{U\}$). For example in a universe of four elements $\{a, b, c, d\}$, and partitions $P = \{\{a\}, \{b\}, \{c, d\}\}$ and $Q = \{\{a, b\}, \{c\}, \{d\}\}$, we have $P \sqcup Q = \{\{a, b\}, \{c, d\}\}$.

Consider a very large matrix $\mathcal{M}$. Each row and column of $\mathcal{M}$ corresponds to a partition of U. An entry in row P and column Q is equal to 1 if $P \sqcup Q$ is the single block partition $\{U\}$, and zero otherwise (see Fig. 11.1).

[1] An observant reader probably noticed that with this new definition the total number of states is quadratic in n. We ignore this loss at the moment as we will oust this additional $\mathcal{O}(n)$ factor later.

Intuitively each row of the matrix $\mathcal{M}$ is a partition corresponding to some partial solution in the graph (V_t, E_t), that is, the part of the graph which we have seen so far in the decomposition. At the same time each column of $\mathcal{M}$ corresponds to a potential partition of X into connected components by the part of the solution contained in $E(G) \setminus E_t$. A single value in the matrix tells us whether two such parts of a solution, the one which we have already constructed and the one we are still to construct, are compatible, i.e., if when joined they give a connected subgraph.

Definition 11.7. Define matrix $\mathcal{M} \in \{0,1\}^{\Pi(U)\times\Pi(U)}$ as

$$\mathcal{M}[P,Q] = \begin{cases} 1 & \text{if } P \sqcup Q = \{U\}, \\ 0 & \text{otherwise.} \end{cases}$$

It seems that the information encoded by $\mathcal{M}$ is excessive for keeping connected sets in dynamic programming. For example, consider rows 2, 3, and 4 of the matrix $\mathcal{M}$ in Fig. 11.1 and let P, Q, and R be the partitions of U corresponding to these rows. Observe that in each column of $\mathcal{M}$ the total number of 1s in rows $\{P,Q,R\}$, is either zero, or at least two. Consequently, for every partition $X \in \Pi(U)$, if there is a partition $S \in \{P,Q,R\}$, such that $X \sqcup S = \{U\}$ (i.e., $\mathcal{M}[X,S] = 1$), then there is also $S' \in \{P,Q,R\}$, such that $S' \neq S$, and $X \sqcup S' = \{U\}$. This means that in our example, if we have all three partitions P,Q,R in some family $c[t,X,i]$ in the dynamic-programming routine, then we can discard one of them! Unfortunately, we do not know how to efficiently check whether some partition is irrelevant with respect to the mentioned criterion, as the number of columns of $\mathcal{M}$ is $|U|^{\mathcal{O}(|U|)}$ and we cannot afford iterating over all of them.

For this reason we pick a different route. Now consider the first four rows of the matrix and observe that their sum is a zero vector when we work in GF(2) (see Section 10.4 for properties of finite fields). But when a sum of vectors is a zero vector in GF(2), then for any column there is an even number of 1s, and in particular there is no column with a single 1. Our strategy is to use computation in GF(2) to show an upper bound on the rank of $\mathcal{M}$, as when we have a subset of rows of $\mathcal{M}$ of carnality greater than rank($\mathcal{M}$), there always exists a subset summing to zero.

In order to prove a $c^{|U|}$ upper bound on the rank of $\mathcal{M}$ we again consider a class of simple partitions, namely cuts into two sets, just as we did in Section 11.2.1. The following lemma shows that when operating in GF(2), the matrix $\mathcal{M}$ can be decomposed into a product of two cutmatrices $\mathcal{C}$, defined as follows.

In Definition 11.8, we use relation $\sqsubseteq$ on partitions, where $P \sqsubseteq Q$ when P is coarser than Q, meaning that each block of Q is contained in a single block of P, or in other words that Q is a refinement of P.

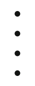
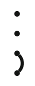
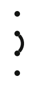
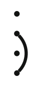
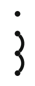
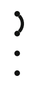
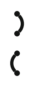
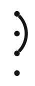
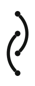
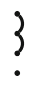
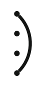
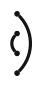
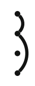
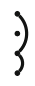
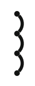
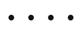
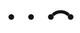
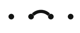
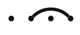
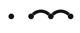
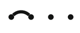
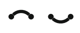
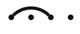
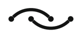
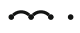
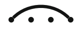
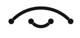
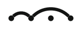
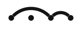
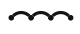
															
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1
	0	0	0	0	0	0	1	0	1	0	0	0	1	1	1
	0	0	0	0	0	0	1	0	0	1	0	1	0	1	1
	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	1	0	0	0	1	0	0	1	0	1	1
	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1
	0	0	0	0	1	0	1	0	0	0	0	1	1	0	1
	0	1	1	0	1	1	1	0	0	1	1	1	1	1	1
	0	1	0	1	1	0	1	0	1	0	1	1	1	1	1
	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1
	0	1	0	1	1	1	1	1	1	1	0	0	1	1	1
	0	1	1	0	1	0	1	1	1	1	0	1	0	1	1
	0	0	1	1	1	1	1	0	1	1	0	1	1	0	1
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Fig. 11.1: Each row and column corresponds to a partition of a 4-element universe, where the topmost vertex of a vertical graph corresponds to the leftmost vertex of a horizontal graph. Each entry of this matrix is equal to 1 iff the union of the two graphs is connected, i.e., when the join of the two partitions is the partition with a single block

Definition 11.8. Define

$$\mathtt{cuts}(U) := \{(V^1, V^2) \ : \ V^1 \cup V^2 = U \wedge V^1 \cap V^2 = \emptyset \wedge v_1 \in V^1\},$$

where v_1 stands for an arbitrary but fixed element of U. Define a matrix $\mathcal{C} \in \{0,1\}^{\Pi(U) \times \mathtt{cuts}(U)}$ by $\mathcal{C}[Q,(V^1,V^2)] = [(V^1,V^2) \sqsubseteq Q]$. In other words, element $(Q,(V^1,V^2))$ of $\mathcal{C}$ is 1 if $(V^1,V^2) \sqsubseteq Q$ and 0 otherwise.

Matrix $\mathcal{C}$ represents which pairs of partitions and cuts are consistent, i.e., the partition needs to be a refinement of the cut. It is crucial that one side of the cut is fixed to contain a particular vertex since that allows the following argument via counting modulo 1: For each pair of partitions, the product $\mathcal{C}\mathcal{C}^T$ can be seen to count the number of partitions that are consistent with both P and Q. If $P \sqcup Q = \{U\}$ then there is only one such partition, namely $(U, \emptyset)$. Otherwise, if $P \sqcup Q$ is a partition with at least two sets, then there is an even number of cuts that are consistent, as for each block of $P \sqcup Q$ which does not contain $v_1 \in V^1$, we are free to put the block either entirely to V^1 or into V^2, independently of other blocks.

Lemma 11.9. *It holds that $\mathcal{M} \equiv_2 \mathcal{C}\mathcal{C}^T$, where $\equiv_2$ denotes equality in* GF(2).

Proof. For every $P, Q \in \Pi(U)$ we have that

$$\begin{aligned}(\mathcal{C}\mathcal{C}^T)[P,Q] &= \sum_{(V^1,V^2)\in\mathtt{cuts}(U)} [(V^1,V^2) \sqsubseteq P]\cdot[(V^1,V^2)\sqsubseteq Q] \\ &= \sum_{(V^1,V^2)\in\mathtt{cuts}(U)} [(V^1,V^2) \sqsubseteq P \wedge (V^1,V^2)\sqsubseteq Q] \\ &= \sum_{(V^1,V^2)\in\mathtt{cuts}(U)} [(V^1,V^2) \sqsubseteq P \sqcup Q] \\ &= 2^{\#\mathtt{blocks}(P\,\sqcup\,Q)-1} \\ &\equiv_2 [P \sqcup Q = \{U\}].\end{aligned}$$

The crux here is that we count the number of cuts which are coarser than both P and Q, i.e., cuts $(V^1, V^2) \in \mathtt{cuts}(U)$ such that each set of P or Q is completely contained in V^1 or V^2. This is equivalent to a situation where each block of $P \sqcup Q$ is contained in V^1 or V^2. Considering the blocks of $P \sqcup Q$ it is clear that the block containing v_1 must be in V^1, by the definition of $\mathtt{cuts}(U)$. Any further block can be either in V^1 or V^2, which gives a total of $2^{\#\mathtt{blocks}(P\,\sqcup\,Q)-1}$ coarser cuts. Clearly, that number is odd if and only if there is a single block in $P \sqcup Q$ (the one containing 1), i.e., if $P \sqcup Q = \{U\}$. □

Based on Lemma 11.9 we prove that instead of looking for linear dependence between rows in $\mathcal{M}$ we can in fact look for linear dependence in $\mathcal{C}$, which is much more convenient as $\mathcal{C}$ has a single exponential (with respect to $|U|$) number of columns, while $\mathcal{M}$ does not. The following lemma proves that given a set of partitions linearly dependent in $\mathcal{C}$ we can discard any one of them. (Here, $\mathcal{C}[Q,\cdot]$ denotes the row of $\mathcal{C}$ corresponding to the partition Q.)

Lemma 11.10. *Let $\mathcal{X} \subseteq \Pi(U)$ and $P, R \in \Pi(U)$ such that $\mathcal{C}[P,\cdot] \equiv_2 \sum_{Q\in\mathcal{X}} \mathcal{C}[Q,\cdot]$ and $P \sqcup R = \{U\}$. Then, there exists $Q \in \mathcal{X}$ such that $Q \sqcup R = \{U\}$.*

Proof. We know that by Lemma 11.9

$$\begin{aligned}\mathcal{M}[P,\cdot] &\equiv_2 \mathcal{C}[P,\cdot]\mathcal{C}^T\\ &\quad\{\text{by assumption}\}\\ &\equiv_2 (\sum_{Q\in\mathcal{X}} \mathcal{C}[Q,\cdot])\mathcal{C}^T\\ &\quad\{\text{by linearity of matrix multiplication}\}\\ &\equiv_2 \sum_{Q\in\mathcal{X}} \mathcal{M}[Q,\cdot].\end{aligned}$$

In particular, $\mathcal{M}[P,R] \equiv_2 \sum_{Q\in\mathcal{X}} \mathcal{M}[Q,R]$. Thus, if $\mathcal{M}[P,R] = 1$ then $\mathcal{M}[Q,R] = 1$ for some $Q \in \mathcal{X}$ and the lemma follows. □

Lemma 11.10 implies that finding a basis of the row space of the matrix $\mathcal{C}$ by standard Gaussian elimination gives the following theorem.

Theorem 11.11. *There is an algorithm that given a set of partitions $\mathcal{A} \subseteq \Pi(U)$ in time $|\mathcal{A}|^{\mathcal{O}(1)}2^{\mathcal{O}(|U|)}$ finds a set $\mathcal{A}' \subseteq \mathcal{A}$ of size at most $2^{|U|-1}$, such that for any $P \in \mathcal{A}$ and $R \in \Pi(U)$ with $P \sqcup R = \{U\}$, there exists a partition $Q \in \mathcal{A}'$, such that $Q \sqcup R = \{U\}$.*

Proof. Let $\mathcal{A}'$ be a basis of the row space of the matrix $\mathcal{C}[\mathcal{A},\cdot]$ (in particular $\mathcal{A}'$ is a subset of rows of $\mathcal{C}[\mathcal{A},\cdot]$ of maximum rank, which is why the approach is called rank-based). We can find $\mathcal{A}'$ by standard Gaussian elimination in time $|\mathcal{A}|^{\mathcal{O}(1)}2^{\mathcal{O}(|U|)}$. As the matrix $\mathcal{C}$ has $2^{|U|-1}$ columns, the rank of $\mathcal{C}[\mathcal{A},\cdot]$, and hence the size of $\mathcal{A}'$, is upper bounded by $2^{|U|-1}$.

Consider any $P \in \mathcal{A}$ and $R \in \Pi(U)$ with $P \sqcup R = \{U\}$. If $P \in \mathcal{A}'$ then it is enough to take $Q = P$. Otherwise P was not taken to the basis of the row space of $\mathcal{C}$, which means that there is a subset $\mathcal{X} \subseteq \mathcal{A}'$ such that $\mathcal{C}[P,\cdot] \equiv_2 \sum_{Q\in\mathcal{X}} \mathcal{C}[Q,\cdot]$. Then, by Lemma 11.10 there is $Q \in \mathcal{A}'$ such that $Q \sqcup R = \{U\}$ and the lemma follows. □

Let us take a step back and recall what was our goal for this section. In our dynamic-programming approach for the Steiner Tree problem, for a fixed node X_t of the given tree decomposition, a fixed subset $X \subseteq X_t$, and a fixed integer i we wanted to limit the number of considered partitions of X in $c[t,X,i]$. In order to do that for every triple (t,X,i), instead of computing the entire family $c[t,X,i]$ we only compute a subset $c'[t,X,i] \subseteq c[t,X,i]$ as follows.

Fix a triple (t,X,i). We want to compute a subset $c'[t,X,i] \subseteq c[t,X,i]$, given the values $c'[t',\cdot,\cdot]$ where t' iterates over the children of t. In the straightforward dynamic-programming algorithm, similar to the one of Section 7.3.3, the recursive formula for $c[t,X,i]$ depends on the values $c[t',\cdot,\cdot]$ for the children of t. In our algorithm, we first apply the aforementioned recursive formula for $c[t,X,i]$, but taking instead the values $c'[t',\cdot,\cdot]$ as an input. Second, we apply Theorem 11.11 to the obtained set to shrink its size to at most $2^{|X|-1}$, obtaining the final value $c'[t,X,i]$. In particular we are keeping only a single exponential number of partitions, deciding which to keep in a greedy manner based on the Gaussian elimination.

Note that in order to prove the correctness of the approach, one needs the following lemma, which can be proved by inspecting the dynamic programming recursive formulas in each node of the tree decomposition. On a high level the proof of the following lemma is not complicated, however formalizing all the details is a nontrivial and rather tedious task, hence we omit the proof.

Lemma 11.12. *For any t, X, i, partition $P \in c[t, X, i]$, and partition R of X such that $P \sqcup R = \{U\}$, there exists $Q \in c'[t, X, i]$ such that $Q \sqcup R = \{U\}$.*

In particular, Lemma 11.12 implies that if $c[t, X, i]$ is nonempty then $c'[t, X, i]$ is also nonempty, which guarantees that when deciding what the final answer is (after the dynamic programming computation is done), c' and c lead to the same result.

It is not hard to extend Theorem 11.11 to a weighted setting (see Exercise 11.12) which allows us to solve the weighted variant of the Steiner Tree problem with single exponential dependence on treewidth. By considering weighted partitions we can omit the accumulator i from our state definition, and retrieve the linear dependence on n (see Exercise 11.13).

The rank-based approach presented above also works for other connectivity problems.

Theorem 11.13. *Let G be an n-vertex graph given together with its tree decomposition of width k. Then*

- Steiner Tree,
- Feedback Vertex Set,
- Connected Dominating Set,
- Hamiltonian Path *and* Longest Path,
- Connected Vertex Cover, *and*
- Connected Feedback Vertex Set.

are solvable in time $2^{\mathcal{O}(k)} \cdot n$.

As in Section 7.7.2 by combining Lemma 7.28 with Theorem 11.13, we obtain the following corollary, which improves the exponent of the running time from Corollary 7.31 by a $\log k$ factor.

Corollary 11.14. *On planar graphs*

- Steiner Tree *parameterized by the size of the tree,*
- Feedback Vertex Set,
- Connected Dominating Set,
- Longest Path,
- Connected Vertex Cover, *and*
- Connected Feedback Vertex Set.

are solvable in time $2^{\mathcal{O}(\sqrt{k})}n^{\mathcal{O}(1)}$.

We would like to note that the CYCLE PACKING problem was listed in Theorem 7.10 but does not appear in Theorem 11.13. The intuitive reason for this is that in the CYCLE PACKING problem we want to maximize the number of connected components (i.e., cycles) in the solution while the approaches given in this section work only for problems where we are to minimize the number of connected components. In fact one can show that under the Exponential Time Hypothesis (see Chapter 14) it is impossible to break the $k^{\mathcal{O}(k)}$ dependence for CYCLE PACKING, but this is beyond the scope of this book.

Exercises

Unless otherwise noted, in the exercises assume a tree decomposition of width k is given.

11.1. Recall that in the INDUCED MATCHING problem we are asked if there is a subset of 2ℓ vertices in G inducing a matching on ℓ edges. Show that INDUCED MATCHING is solvable in time $3^k n^{\mathcal{O}(1)}$.

11.2. In the TOTAL DOMINATING SET problem we want to decide whether there is a set $X \subseteq V(G)$ of size at most ℓ such that each vertex in G (in particular each vertex in X) has a neighbor in X. Show that TOTAL DOMINATING SET is solvable in time $4^k n^{\mathcal{O}(1)}$.

11.3. Show how to obtain $3^k k^{\mathcal{O}(1)} n$ running time for the DOMINATING SET problem (note the linear dependency on n).

11.4 (✎). Give a $2^k n^{\mathcal{O}(1)}$-time algorithm computing the cardinality of the set $\mathcal{R}$ defined by (11.1).

11.5. Give a $3^k n^{\mathcal{O}(1)}$-time algorithm computing the cardinality of the set $\mathcal{C}$ defined by (11.3).

11.6. Show a one-sided error Monte Carlo $3^k n^{\mathcal{O}(1)}$-time algorithm with false negatives solving the CONNECTED VERTEX COVER problem.

11.7. Show a one-sided error Monte Carlo $4^k n^{\mathcal{O}(1)}$-time algorithm with false negatives solving the CONNECTED DOMINATING SET problem.

11.8 (☠). Show a one-sided error Monte Carlo $3^k n^{\mathcal{O}(1)}$-time algorithm with false negatives solving the FEEDBACK VERTEX SET problem.

11.9 (☠). Show a one-sided error Monte Carlo $3^k n^{\mathcal{O}(1)}$-time algorithm with false negatives solving the FEEDBACK VERTEX SET problem parameterized by the solution size (i.e., when k is the number of vertices we are allowed to remove).

11.10 (☠). Give a $3^k n^{\mathcal{O}(1)}$ time Monte Carlo algorithm for STEINER TREE that applies the Schwartz-Zippel lemma instead of the isolation lemma.

11.11 (☠). Give a $2^k n^{\mathcal{O}(1)}$-time algorithm for the LONGEST PATH problem parameterized by the length of the path (i.e., here k denotes the length of the path, and no tree decomposition is given) that uses the isolation lemma instead of the Schwartz-Zippel lemma.

11.12. Show that there is an algorithm that given a set of partitions $\mathcal{A} \subseteq \Pi(U)$ and a weight function $\mathbf{w} : \mathcal{A} \to \mathbb{N}$ in time $|\mathcal{A}|^{\mathcal{O}(1)} 2^{\mathcal{O}(|U|)}$ finds a set $\mathcal{A}' \subseteq \mathcal{A}$ of size at most $2^{|U|-1}$, such that for any $P \in \mathcal{A}$ and $R \in \Pi(U)$ with $P \sqcup R = \{U\}$, there exists a partition $Q \in \mathcal{A}'$, such that $Q \sqcup R = \{U\}$ and $\mathbf{w}(Q) \leq \mathbf{w}(P)$.

11.13. In the weighted variant of the Steiner Tree problem each instance is additionally equipped with a weight function $\mathbf{w} : E(G) \to \mathbb{N}$ and we are to find a minimum cost Steiner tree.

Show an algorithm solving this problem in time $2^{\mathcal{O}(k)} n$ (note linear dependence on n).

Hints

11.1 Each vertex can be in one of three possible states:

- already matched,
- decided to be matched, but not yet matched,
- decided not to be matched.

11.2 Each vertex can be in one of four possible states, depending on whether it is chosen to a solution or not, and whether it already has a neighbor in a solution or not.

Consider a join node t with children t_1 and t_2. For a fixed state at t, the interesting pairs of states at t_1 and t_2 keep the same choice whether a vertex in a bag is in the solution or not, whereas the sets of already dominated vertices need to satisfy a certain subset convolution formula.

11.3 Show a way to narrow the maximum possible difference between values $c[t, f]$ for a node t over all functions f to $\mathcal{O}(k)$. Then argue that we can use the min-sum semiring with values bounded by $\mathcal{O}(k)$.

11.4 In a state of dynamic programming keep the subset of vertices selected to H as well as the number of edges added so far to H.

11.8 Let X be a feedback vertex set we are looking for, and assume $|X| = \ell$. The first step is to guess c, the number of connected components of $G - X$. Then, $G - X$ has $n - \ell$ vertices and $n - \ell - c$ edges. Our goal is to use Cut & Count and cuts of the graph $G - X$ to ensure that $G - X$ has at most c connected components, while, in separate accumulators of the dynamic programming, keeping track of the number of vertices and edges of $G - X$.

However, there is a significant challenge to generalize the approach from the Steiner Tree example, where we want the solution graph to have one connected component, to our case of at most c connected components. To cope with this issue, we introduce a set M of c *markers*, which will serve the role of the vertex v_1 in the Steiner Tree example. More precisely, we define $\mathcal{C}$ to be a family of triples (X, M, C), where:

1. $|X| = \ell$;
2. $G - X$ has exactly $n - \ell - c$ edges;
3. $M \subseteq V(G) \setminus X$ and $|M| = c$;
4. $C = (V^1, V^2)$ is a cut of $G - X$, consistent with $G - X$, such that $M \subseteq V^1$.

It is a relatively standard task to design a dynamic-programming algorithm counting $|\mathcal{C}|$ in time $3^k n^{\mathcal{O}(1)}$, if a decomposition of width k is provided. An argument similar to the one for Steiner Tree shows that, for every sets $M, X \subseteq V(G)$ there exists exactly one cut C such that $(X, M, C) \in \mathcal{C}$ if and only if:

1. $|X| = \ell$;
2. $G - X$ is a forest with exactly c connected components; and

3. $|M| = c$ and every connected component of $G - X$ contains exactly one vertex of M.

Moreover, for other pairs $M, X \subseteq V(G)$, there is an even number of cuts C such that $(X, M, C) \in \mathcal{C}$.

To conclude, we apply the isolation lemma to the universe containing two copies of every vertex of $V(G)$; one copy of a vertex v corresponds to taking v into X, and the second copy corresponds to taking v into M.

11.9 Use iterative compression together with the solution to Exercise 11.8.

11.10 For every edge $e \in E(G)$ create a variable x_e and for every vertex $v \in V(G)$ create a variable y_v. With every subgraph $H \in \mathcal{R}$ associate a monomial $p(H) = \prod_{e \in E(H)} x_e \cdot \prod_{v \in V(H)} y_v$. Design a dynamic-programming algorithm to compute, within the promised time bound, the value of the polynomial $\sum_{(H,C) \in \mathcal{C}} p(H)$, given fixed values of the input variables. Argue, as in the case for Steiner Tree, that in a field of characteristic 2 the aforementioned polynomial is nonzero if and only if the input instance is a yes-instance. Finally, use the Schwartz-Zippel lemma to get a Monte Carlo algorithm.

11.11 Proceed as in Section 10.4.1. Without loss of generality, assume the input graph is directed. Count pairs (W, ℓ), where W is a walk with k vertices and $\ell : [k] \to [k]$ is a bijection. Randomly choose weights $\mathbf{w}(e)$ for every edge $e \in E(G)$ and $\mathbf{w}(v, i)$ for every vertex $v \in V(G)$ and $i \in [k]$. To a pair (W, ℓ) assign a weight $\sum_{e \in W} \mathbf{w}(e) + \sum_{i=1}^{k} \mathbf{w}(v_i, \ell(i))$, where v_i is the i-th vertex on the walk W, and count the parity of the number of pairs (W, ℓ) of every fixed weight.

11.12 Instead of finding an arbitrary basis of $\mathcal{C}$ it is enough to find a minimum weight basis.

11.13 Use the modified Theorem 11.11 from Exercise 11.12, remove the weight accumulator i from the definition of a state of the dynamic programming and consider a family of weighted partitions for a fixed pair (t, X).

Bibliographic notes

The application of the fast subset convolution to the problems parameterized by treewidth described in Section 11.1 was first described by van Rooij, Bodlaender and Rossmanith [405]. Dorn [142] shows how to speed up dynamic programming on graphs of bounded branchwidth by making use of matrix multiplication; see also Bodlaender, van Leeuwen, van Rooij and Vatshelle [55].

It was believed for a long time that $k^{\mathcal{O}(k)}$ dependence in treewidth for connectivity problems is optimal. This motivated research on faster algorithms on planar, and more generally, on H-minor-free graphs [147, 146, 145, 407]. These algorithms were exploiting specific properties of sparse graphs like Catalan structures. The Cut & Count technique from Section 11.2.1 was obtained by Cygan, Nederlof, Pilipczuk, Pilipczuk, van Rooij, and Wojtaszczyk [118], while the deterministic rank-based approach from Section *11.2.2 comes from Bodlaender, Cygan, Kratsch and Nederlof [48] (see [170] for experimental evaluation of the rank-based approach). The idea behind Theorem 11.11—maintaining only a "representative" subset of the so far constructed partial solutions—is also used in Chapter 12. Interestingly, the factorization lemma (Lemma 11.9) can be derived from a much more general setting studied in communication complexity [335, 339] (more specifically, Lemma 5.7 from [335]). In [206], fast computations of representative families of independent sets in different families of matroids are used to obtain single-exponential algorithms on graphs of bounded treewidth. The work of Freedman, Lovász and Schrijver [217, 336] is relevant to the rank-based approach. Specifically, Freedman, Lovász and Schrijver [217, 336] introduced connection matrices of graph parameters and showed that finite rank of such matrices implies efficient algorithms for determining the graph parameter on graphs of small treewidth; see the book of Lovász [337, Theorem 6.4]. Godlin, Kotek and Makowsky [229, 304, 344] gave general conditions on $\mathbf{MSO_2}$ definable graph properties with a finite connection rank.

For the Hamiltonian Cycle problem, instead of using all the partitions in the matrix $\mathcal{M}$ it is enough to use partitions with all the blocks of size exactly 2, i.e., perfect matchings. This fact was investigated by Cygan, Kratsch and Nederlof [113], leading to improved running times for Hamiltonian Cycle.

For Cycle Packing on planar graphs a single exponential (in treewidth) algorithm is given by Baste and Sau [25].

The isolation lemma in the form presented in this chapter comes from Mulmuley, Vazirani and Vazirani [370], but the first discovery of the isolation lemma is credited to Valiant and Vazirani [424].

The currently fastest randomized algorithms (in terms of the dependence on treewidth) for Steiner Tree and Feedback Vertex Set run in time $3^k n^{\mathcal{O}(1)}$ [118], and the constant 3 cannot be improved under the Strong Exponential Time Hypothesis. For Hamiltonian Cycle parameterized by pathwidth the currently fastest algorithm runs in time $(2+\sqrt{2})^k n^{\mathcal{O}(1)}$ [113] (surprisingly, this bound is also tight under SETH), while for treewidth the currently best running time is $4^k n^{\mathcal{O}(1)}$ [118].

When restricted to deterministic algorithms parameterized by treewidth, the best running time for Steiner Tree and Feedback Vertex Set is $(1 + 2^{\omega-1} \cdot 3)^k n^{\mathcal{O}(1)}$ [201].

Chapter 12
Matroids

Tools and techniques from matroid theory have proven very useful in the design of parameterized algorithms and kernels. In this chapter, we showcase how a polynomial-time algorithm for Matroid Parity *can be used as a subroutine in an* FPT *algorithm for* Feedback Vertex Set*. We then define the notion of representative sets, and use it to give polynomial kernels for d-*Hitting Set *and d-*Set Packing*, and fast (deterministic)* FPT *algorithms for* Longest Path *and* Long Directed Cycle.

Matroid theory unifies and generalizes common phenomena in two seemingly unrelated areas: graph theory and linear algebra. For example, in a certain technical sense, the notion of linear independence in a vector space and acyclicity in graphs behave very similarly. As we shall see, the connection between the two areas can be exploited algorithmically by translating graph-theoretic problems into the language of linear algebra and then we can use efficient algebraic methods to solve them.

Many of the problems considered in this book are problems on graphs, or on set systems, where input is a finite universe U and a family $\mathcal{F}$ of sets over U. A family $\mathcal{F}$ of sets over a finite universe U is a *matroid* if it satisfies the following three *matroid axioms*:

- $\emptyset \in \mathcal{F}$,
- if $A \in \mathcal{F}$ and $B \subseteq A$ then $B \in \mathcal{F}$,
- if $A \in \mathcal{F}$ and $B \in \mathcal{F}$ and $|A| < |B|$ then there is an element $b \in B \setminus A$ such that $(A \cup \{b\}) \in \mathcal{F}$.

The second property is called the *hereditary* property of matroids, while the third is called the *exchange* property.

DOI 10.1007/978-3-319-21275-3_12

Just like graphs, matroids have been studied extensively, and there are many connections between matroid theory and graph theory. A thorough introduction to matroid theory is beyond the scope of this book, the aim of this chapter is to showcase how some tools and techniques from matroid theory can be used to design parameterized algorithms.

The first example is about the Feedback Vertex Set problem. Here, we are given a graph G and integer k, and asked whether there exists a set S on at most k vertices such that $G - S$ is acyclic. The problem is NP-complete. However, quite surprisingly, it turns out to be polynomial-time solvable on graphs of maximum degree 3 by a reduction to the so-called Matroid Parity problem. This claim, proved in Section 12.2.2, is the missing piece of the $3.619^k n^{\mathcal{O}(1)}$-time algorithm for Feedback Vertex Set described in Section *4.3.2.

Perhaps the most dramatic examples of the use of matroid theory in parameterized algorithms are the randomized polynomial kernels for Odd Cycle Transversal and Almost 2-SAT by Kratsch and Wahlström [312, 310]. The main ingredient from matroid theory in the kernelization algorithm for Almost 2-SAT is the concept of *representative sets*. Unfortunately the kernelization algorithms for Odd Cycle Transversal and Almost 2-SAT are quite complex, so we do not cover them in this book. Instead, we show how representative sets can be used to give polynomial kernels for d-Hitting Set and d-Set Packing.

It turns out that representative sets are useful not only for designing kernelization algorithms, but also for designing fast FPT algorithms. We give two examples of fast parameterized algorithms that use representative sets; the first is a $6.75^{k+o(k)} n^{\mathcal{O}(1)}$-time algorithm for the Long Directed Cycle, the second a $2.619^k n^{\mathcal{O}(1)}$-time algorithm for Longest Path in directed graphs.

This chapter assumes that the reader has some basic knowledge of linear algebra. We will freely make use of such concepts as linear independence, the span of a set of vectors, basis vectors, dimension of a space and determinants. We will also use finite fields, but will restrict ourselves to the finite fields of prime size, that is the integers modulo a prime p. We refer you to the textbook [316] for an introduction to linear algebra.

Before we start, we define some of the basic notions from matroid theory. Note that, as matroid theory generalizes concepts from both graph theory and linear algebra, it borrows terms from both of these areas. If $(U, \mathcal{F})$ is a matroid, then the elements of U are called the *edges* of the matroid, the sets $S \in \mathcal{F}$ are called *independent sets*.[1] An inclusion-maximal independent set S is a *basis* of the matroid, while the inclusion-minimal non-independent sets are called *circuits*. From the exchange property of matroids, it follows

[1] It should be emphasized that the term "independent" here alludes to linear independence of vectors and has nothing to do with independent sets in graphs. In particular, the family of independent sets of a graph is *not* necessarily a matroid (Exercise 12.1).

directly that all the bases of a matroid $(U, \mathcal{F})$ have the same size; this size is referred to as the *rank* of the matroid.

12.1 Classes of matroids

Often the matroids that show up in applications have a particular structure. We will now look at some of the most common ones. The most basic class of matroids are the *uniform* matroids. A uniform matroid is completely defined by the universe U and its rank r, as every subset S of the universe of size at most r is independent. It is easy to verify that uniform matroids indeed satisfy the matroid axioms.

12.1.1 Linear matroids and matroid representation

One of the main purposes of matroids is to serve as a "combinatorial" generalization of the notion of linear independence of vectors. Linear matroids are the matroids that can be defined using linear independence. Let U be a set, and assign to every $e \in U$ a vector v_e over some field. The vectors for the different elements of the universe should all be over the same field F and have the same dimension. We now define a family $\mathcal{F}$ of independent sets as follows. A set $S \subseteq U$ is in $\mathcal{F}$ if and only if the set $\{v_e \ : \ e \in S\}$ forms a linearly independent set of vectors.

Lemma 12.1. *$(U, \mathcal{F})$ is a matroid.*

Proof. We need to verify that $(U, \mathcal{F})$ satisfies the matroid axioms. Clearly, $\emptyset \in \mathcal{F}$. Furthermore, a subset of linearly independent vectors is also linearly independent, therefore $(U, \mathcal{F})$ satisfies the hereditary property. It remains to prove that $(U, \mathcal{F})$ satisfies the exchange property.

Let A and B be sets in $\mathcal{F}$ such that $|A| < |B|$. Then the set $\{v_e \ : \ e \in A\}$ spans a space of dimension $|A|$ while $\{v_e \ : \ e \in B\}$ spans a space of dimension $|B|$. It follows that there is an element $b \in B$ such that v_b is not in the span of $\{v_e \ : \ e \in A\}$, and so $(A \cup \{b\}) \in \mathcal{F}$. □

We define a matrix M with one column for each edge $e \in U$ such that the column vector of e is v_e. The matrix M is said to *represent* the matroid $(U, \mathcal{F})$ over F. A matroid $\mathcal{M} = (U, \mathcal{F})$ is called *representable over F* if there is a matrix over F that represents $\mathcal{M}$. A matroid that is representable over the field GF(2), that is, the integers modulo 2, is called a *binary* matroid. A matroid is called *representable* or *linear* if it is representable over some field. Even though not all matroids are representable, all the matroids that we will be working with in this book are.

Row operations, i.e., multiplying a row by a nonzero scalar, or adding one row to another do not affect the linear independence of columns. Therefore, we can perform row operations on a matrix M representing a matroid $\mathcal{M}$, and the resulting matrix will still represent the same matroid. Thus, if the rank of M is r, we can, by means of row operations, modify M such that the top r rows are linearly independent and all the remaining rows have zeroes everywhere. We can then simply delete the rows that are all 0, since they do not affect the linear independence of columns anyway. Hence, for a representable matroid of rank r, we can always ensure that the representation matrix has exactly r rows. If for some reason we are given a representation matrix with more rows, we can find a representation matrix with r rows in polynomial time by means of Gaussian elimination.

In most situations we may assume without loss of generality that the representation matrix of a matroid of rank r has exactly r rows.

12.1.2 Representation of uniform matroids

We will now show that a uniform matroid with a universe U of size n and rank k can be represented over any field GF(p) for $p > n$. Let the elements of the universe be numbered as $U = e_1, e_2, \ldots, e_n$. Assign to each element e_i a unique nonzero field element α_i. We set the vector v_i of the element e_i to be the k-dimensional vector $(1, \alpha_i^1, \alpha_i^2, \ldots, \alpha_e^{k-1})$.

Since the vectors are k-dimensional, it follows that for any set $A \subseteq U$ of size more than k, the set $\{v_i \ : \ e_i \in A\}$ is not linearly independent. We now show that for any set $A \subseteq U$ of size k, the set $\{v_i \ : \ e_i \in A\}$ is linearly independent. It suffices to show that the matrix M_A with column vectors $\{v_i \ : \ e_i \in A\}$ has a nonzero determinant. The matrix M_A is a Vandermonde matrix [270], and hence the determinant of M_A is equal to

$$\prod_{\substack{0 \leq i < j \leq n \\ e_i \in A, e_j \in A}} \alpha_j - \alpha_i.$$

Since this is a product of nonzero field elements, it follows that the determinant of M_A is nonzero, and hence $\{v_i \ : \ e_i \in A\}$ is a set of linearly independent vectors.

12.1.3 Graphic matroids

The graphic matroids are matroids that arise from graphs in the following way. The graphic matroid $\mathcal{M}_G$ of an undirected graph G has universe $E(G)$ and a set $S \subseteq E(G)$ is independent if the subgraph $(V(G), S)$ is acyclic. Many of the terms used with general matroids have been coined with graphic matroids in mind. For example, the edges of $\mathcal{M}_G$ are just the edges of G. The circuits of $\mathcal{M}_G$ are exactly the cycles of G.

It is not immediately obvious that for every graph G, $\mathcal{M}_G$ is actually a matroid. It is easy to see that the empty set is independent and that the hereditary property always holds, but the exchange property is a little trickier. If $A \subseteq E(G)$ is independent and $B \subseteq E(G)$ is independent, and $|A| < |B|$, which element $b \in B$ can we pick so that $A \cup \{b\}$ is independent? Consider a connected component C of $(V(G), A)$. As $(V(G), A)$ is acyclic, if B has more edges inside $V(C)$ than A, then there is a cycle in B, contradicting that B is independent. Therefore, in each component of $(V(G), A)$, the graph $(V(G), B)$ contains at most as many edges as A does. Since $|A| < |B|$ it follows that there is an edge $b \in B$ whose endpoints are in different connected components of $(V(G), A)$. But then $A \cup \{b\}$ is independent, and hence $\mathcal{M}_G$ is a matroid.

To represent the graphic matroid of a graph G make a matrix with a column for each edge and a row for each vertex. The entry corresponding to a vertex v and an edge e is 0 if v is not incident to e, and either 1 or -1 if it is incident. Each column should contain exactly one 1 and one -1.

Theorem 12.2. *Graphic matroids are representable over any field.*

Proof. We give a constructive proof, i.e., an algorithm that given an undirected graph G computes a representation of the graphic matroid $\mathcal{M}_G$. We start by turning G into a directed graph D by arbitrarily orienting each edge. The representation of $\mathcal{M}_G$ is a matrix M with one column v_e for each edge $e \in E(G)$ and a row for each vertex. We set all the entries of v_e to 0 except in the two entries that are in the rows corresponding to the endpoints u and v of e. If the edge e is oriented from u to v in D, we set the entry of v_e corresponding to u to -1 and the entry corresponding to v to 1. If the edge is oriented from v to u, we set the entry of v_e corresponding to u to 1 and the entry corresponding to v to -1.

It remains to show that a set $A \subseteq E(G)$ is acyclic if and only if the set of vectors $\{v_e \ : \ e \in A\}$ is linearly independent. We first show the forward direction. Suppose for contradiction that $A \subseteq E(G)$ is acyclic but $\{v_e \ : \ e \in A\}$ is not linearly independent. Then there exists a set of values $\{\alpha_e \ : \ e \in A\}$ such that at least one of the values α_e is nonzero, and

$\sum_{e \in A} \alpha_e v_e = 0$. Let $A' \subseteq A$ be the set of edges e in A such that α_e is nonzero. Since A is acyclic, A' is acyclic as well, and hence there exists a vertex u that is incident to exactly one edge $\hat{e}$ in A'. But then the entry of $\sum_{e \in A} \alpha_e v_e$ corresponding to u must be nonzero, as the only nonzero contribution to the sum comes from $\alpha_{\hat{e}} \cdot v_{\hat{e}}$. This contradicts $\sum_{e \in A} \alpha_e v_e = 0$.

For the reverse direction, suppose that $A \subseteq E(G)$ contains a cycle C. We will construct a set of values $\{\alpha_e \ : \ e \in A\}$ such that at least one of the values α_e is nonzero, and $\sum_{e \in A} \alpha_e v_e = 0$. This shows that the set $\{v_e \ : \ e \in A\}$ is linearly dependent. Let A' be the set of edges of the cycle C. We set $\alpha_e = 0$ for every edge $e \notin C$. Consider a walk around the cycle C. For each edge $e \in A'$ we set the variable α_e to 1 if the arc in D corresponding to e is oriented in the direction of the walk, and to -1 otherwise. Consider now the sum $\sum_{e \in A} \alpha_e v_e$. Any entry corresponding to a vertex not on the cycle C is 0. For the entry corresponding to a vertex u on the cycle C, the sum $\sum_{e \in A} \alpha_e v_e$ has two nonzero terms, one for each edge incident to u on C. Let e_1 be the edge by which the walk enters u, and e_2 be the edge by which the walk leaves u. It is easily verified that the term in the sum corresponding to e_1 is 1 and the term corresponding to e_2 is -1. Thus $\sum_{e \in A} \alpha_e v_e = 0$, completing the proof. $\square$

In particular, Theorem 12.2 implies that graphic matroids are binary matroids. Observe that if we are working with GF(2) then $-1 = 1$. Thus the representation matrix for the graphic matroid of G in GF(2) is just a matrix with a row for each vertex, a column for each edge, and the entry corresponding to a vertex u and edge e is 1 if the edge e is incident to v. This matrix is called the vertex-edge incidence matrix.

12.1.4 Transversal matroids

Transversal matroids are another class of matroids that arise from graphs. For a bipartite graph $G = (U \cup B, E)$ with all edges between U and B, we can define a matroid M with universe U, and a set $S \subseteq U$ is independent if there exists a matching in G such that every vertex in S is an endpoint of a matching edge. It is easy to see that $\mathcal{M}$ satisfies $\emptyset \in \mathcal{F}$ and the hereditary property; however it is not at all obvious that $\mathcal{M}$ satisfies the exchange property.

In Exercises 12.2–12.3, we will consider two different approaches for proving that $\mathcal{M}$ is a matroid. One approach is to show that $\mathcal{M}$ satisfies the exchange property using a flow argument. Another approach is to give a representation for $\mathcal{M}$.

Theorem 12.3. *There is a polynomial-time randomized algorithm that, given as input a bipartite graph $G = (U \cup B, E)$ and a positive integer x, outputs a matrix A of integers between 0 and $x \cdot n \cdot 2^n$. With success probability at least $1 - 1/x$, the matrix A is a representation of the transversal matroid $\mathcal{M}$ of*

G over the rationals. If the algorithm fails, then A represents a matroid $\mathcal{M}'$ with universe U such that every independent set of $\mathcal{M}'$ is independent in $\mathcal{M}$.

Theorem 12.3 proves that transversal matroids are in fact matroids: given any bipartite graph $G = (U \cup B, E)$ the algorithm will output a representation of $\mathcal{M}$ with positive probability. Note that the integers in the representation matrix A can be exponential in n; however they still only require $\mathcal{O}(n)$ bits to be stored. Since the basic operations on integers, such as addition and multiplication, can be done in time polynomial in the size of the bit representation, the overhead caused from working with the integers in A is only polynomial in n. Even though the proof of Theorem 12.3 is quite easy (see Exercise 12.3), it remains a challenging open problem to design a deterministic polynomial-time algorithm that, given as input $G = (U \cup B, E)$, outputs a representation of the corresponding transversal matroid $\mathcal{M}$.

12.1.5 Direct sum and partition matroids

The *direct sum* of two matroids is an operation that takes two matroids $\mathcal{M}_1$ and $\mathcal{M}_2$ over disjoint universes U_1 and U_2 respectively, and produces a new matroid $\mathcal{M}$ over $U_1 \cup U_2$. A subset S of $U_1 \cup U_2$ is independent in $\mathcal{M}$ if $S \cap U_1$ is independent in $\mathcal{M}_1$ and $S \cap U_2$ is independent in $\mathcal{M}_2$. It is easy to verify that $\mathcal{M}$ satisfies the matroid axioms.

If $\mathcal{M}_1$ and $\mathcal{M}_2$ can be represented over the same field $\mathrm{GF}(p)$, then their direct sum $\mathcal{M}$ can also be represented over $\mathrm{GF}(p)$. In particular if the matrix A_1 represents $\mathcal{M}_1$ and A_2 represents $\mathcal{M}_2$ then the matrix

$$\begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}$$

represents $\mathcal{M}$.

Partition matroids are the matroids that can be written as disjoint sums of uniform matroids. That is, in the universe U is partitioned into classes $U_1, \ldots, U_k$ with an integer a_i associated to each class U_i, and a set $S \subseteq U$ is independent in the partition matroid if $|S \cap U_i| \le a_i$ for every $i \in [k]$. The representation of uniform matroids described in Section 12.1.2, together with the above paragraph, gives a way to represent any partition matroid with edge set U over any field of size at least $\max_{i \le k}(|U_i| + 1)$.

12.2 Algorithms for matroid problems

What are the most common computational problems about matroids and how efficiently can we solve these problems? One kind of question we have already seen a few examples of is, given as input a graph or a different structure that implicitly defines a matroid $\mathcal{M}$, compute a representation of $\mathcal{M}$. Sometimes we need to compute the representation of $\mathcal{M}$ in a particular field GF(p) or determine that such a representation does not exist. We will not be dwelling on these kinds of questions beyond what has already been discussed in the previous sections.

Another type of problem is when a matroid $\mathcal{M}$ is given as input and we have to answer some question about it. When discussing computational problems on matroids, we have to define first how the matroid is described in the input. For example, we may assume that the matroid is given by listing all independent sets/bases, given implicitly (e.g., as the transversal matroid of a bipartite graph), or we can consider a setting where the matroid is given by an oracle capable of answering whether a set is independent. In this chapter, we consider computational problems exclusively on linear matroids and we assume that the representation matrix A of the matroid $\mathcal{M}$ is given as input.

Possibly the most fundamental question is where we are given a set S and asked whether the set is independent in the matroid $\mathcal{M}$. This can easily be done in polynomial time using Gaussian elimination. Using Gaussian elimination we can also find in polynomial time a column basis for the matrix A. This corresponds to a maximum size independent set (or basis) in the matroid $\mathcal{M}$ and therefore we can determine the rank of $\mathcal{M}$. Sometimes we want to find a largest possible, or a smallest possible circuit. Sometimes we are given a weight function $\mathbf{w} : U \to \mathbb{Z}$ on the universe, and the task is to find the basis (or circuit) S that maximizes (or minimizes) the total weight of S. We will denote the weight of an element $u \in U$ by $\mathbf{w}(u)$ and the weight of a set $S \subseteq U$ by $\mathbf{w}(S) = \sum_{u \in S} \mathbf{w}(u)$. The easiest of these optimization problems is to find a basis of minimum weight. Here, a simple greedy algorithm does the trick.

To find a basis of minimum weight, start with $S = \emptyset$ and repeatedly extend S by the smallest weight element whose addition to S keeps S independent.

Formally, we start with $S_0 = \emptyset$. Then, for each $i \geq 1$, let $S_i = S_{i-1} \cup \{u\}$ where $u \notin S$ is the minimum weight element such that $S_{i-1} \cup \{u\}$ is independent. Let r be the rank of $\mathcal{M}$. Since all maximal independent sets of $\mathcal{M}$ have size r, it follows that the algorithm will be able to find an independent set S_r, but unable to find any u such that $S_r \cup \{u\}$ is independent. The algorithm outputs the set S_r. We claim that this set is actually the basis of minimum weight.

Theorem 12.4. *The greedy algorithm computes a basis S_r of minimum weight.*

Proof. Let X be a basis of $\mathcal{M}$ of minimum weight. Order the elements of X into $x_1, x_2, \ldots, x_r$ by nondecreasing weight. For each $i \leq r$, define $X_i = \{x_1, \ldots, x_i\}$. We prove by induction that the weight of S_i is at most the weight of X_i. For the base case, the weight of both S_0 and X_0 is 0. For the inductive step, we prove that the weight of S_i is at most that of X_i, assuming that the statement holds for all $j < i$.

Let u be the element the algorithm added to S_{i-1} to obtain S_i. By the exchange property, there is an element $v \in X_i$ such that $S_{i-1} \cup \{v\}$ is independent. Since the algorithm picked u over v, it follows that the weight of u is at most the weight of v. Furthermore, since the elements x_i are ordered by nondecreasing weight and $v \in X_i$, we have $\mathbf{w}(v) \leq \mathbf{w}(x_i)$. Thus

$$\mathbf{w}(S_i) = \mathbf{w}(S_{i-1}) + \mathbf{w}(u) \leq \mathbf{w}(S_{i-1}) + \mathbf{w}(v) \leq \mathbf{w}(X_{i-1}) + \mathbf{w}(x_i) \leq \mathbf{w}(X_i)$$

(in the second inequality, we used the induction hypothesis). This concludes the proof. $\square$

We leave it as an exercise (see Exercise 12.4) to prove that an analogous greedy algorithm can be used to find a basis of maximum weight. While the optimization questions about bases are computationally tractable, the ones about circuits are not. For example, circuits in the graphic matroid $\mathcal{M}_G$ of a graph G are exactly the cycles of G. Thus, determining whether $\mathcal{M}_G$ has a circuit of length at least n is equivalent to the NP-complete HAMILTONIAN CYCLE problem [224]. Since graphic matroids are binary matroids, it follows that finding the largest circuit is NP-complete for binary matroids. In graphic matroids finding the *smallest* circuit corresponds to finding the shortest cycle in a graph G. This can easily be done in polynomial time. On the other hand, finding the smallest circuit of a binary matroid can easily be seen to be equivalent to the following NP-complete problem, called EVEN SET. Here, input is a family $\mathcal{F}$ of sets over a universe U and an integer k. The task is to find a nonempty subset S of U of size at most k such that for every set $X \in \mathcal{F}$, $|X \cap S|$ is even. To see that EVEN SET is equivalent to finding the smallest circuit of a binary matroid, consider the set-element incidence matrix A of U and $\mathcal{F}$. Here, every column corresponds to an element $u \in U$, every row to a set $X \in \mathcal{F}$ and the entry in the column of u and row of X is 1 if $u \in X$. The matrix A represents a matroid where the circuits are exactly the minimal subsets S of U such that for every set $X \in \mathcal{F}$, $|X \cap S|$ is even.

Since the smallest/largest circuit problems are NP-hard, it is interesting to investigate their parameterized complexity. Determining whether a given transversal matroid has a circuit of size at most k is equivalent to asking whether a given bipartite graph G with bipartitions U and B contains a set $S \subseteq U$ of size at most k such that $|N(S)| < |S|$. In Exercise 13.28 the reader is asked to show that this problem is in fact W[1]-hard, and therefore unlikely to admit an FPT algorithm. For *binary* matroids finding a circuit of size at most k is equivalent to the EVEN SET problem, and determining whether EVEN SET admits an FPT algorithm remains a challenging open problem. For the parameterized complexity of problems related to Even Set, see Chapter 13. On the other hand, one may use matroid minor theory to show that the problem of finding a circuit of size at least k is fixed parameter tractable for all matroids representable over a fixed finite field [227, 262]. The parameterized complexity of this problem on matroids representable over the integers remains open.

12.2.1 Matroid intersection and matroid parity

Since it is easy to find the minimum or maximum weight independent set in a matroid in polynomial time, it is natural to ask whether polynomial-time algorithms could be achieved for generalizations of this problem. A possible generalization is to have multiple matroids over the same universe, and look for a minimum (or maximum) weight set which is simultaneously independent in all of the matroids.

For a fixed integer ℓ, we define the ℓ-Matroid Intersection problem as follows. Input consists of a universe U and ℓ matrices representing matroids $\mathcal{M}_1, \mathcal{M}_2, \dots \mathcal{M}_\ell$ over U, and an integer k. The task is to determine whether there exists a set $S \subseteq U$ of size at least k that is independent in all of the matroids $\mathcal{M}_i$, $i \leq \ell$.

For two set systems $\mathcal{S}_1 = (U, \mathcal{F}_1)$ and $\mathcal{S}_2 = (U, \mathcal{F}_2)$, we define their *intersection* as

$$\mathcal{S}_1 \cap \mathcal{S}_2 = (U, \mathcal{F}_1 \cap \mathcal{F}_2).$$

Consider now the intersection $\mathcal{M}_1 \cap \mathcal{M}_2$ of two matroids $\mathcal{M}_1 = (U, \mathcal{F}_1)$ and $\mathcal{M}_2 = (U, \mathcal{F}_2)$. We have that $\emptyset \in \mathcal{F}_1 \cap \mathcal{F}_2$ and that $S \in \mathcal{F}_1 \cap \mathcal{F}_2$ implies that all subsets of S also are in $\mathcal{F}_1 \cap \mathcal{F}_2$. Thus, if $\mathcal{M}_1 \cap \mathcal{M}_2$ would also satisfy the exchange axiom, then $\mathcal{M}_1 \cap \mathcal{M}_2$ would be a matroid and we could find a maximum weight set which is simultaneously independent in both by using the greedy algorithm of Theorem 12.4. In fact, then we could use the greedy algorithm to solve ℓ-Matroid Intersection for any ℓ. However the intersection of two matroids is not necessarily a matroid, as the reader is asked to prove in Exercise 12.6. On the other hand, the intersection of a matroid with a *uniform* matroid is always a matroid; this operation is called the *truncation* of the matroid. In particular, in Exercise 12.7, the reader is asked to prove the following lemma.

Lemma 12.5. *If $\mathcal{M} = (U, \mathcal{F})$ is a matroid of rank at least r and $\mathcal{U}_r$ is the uniform matroid with edge set U and rank r, then $\mathcal{M} \cap \mathcal{U}_r$ is a matroid of rank r. Furthermore there is a polynomial-time randomized algorithm that, given integers $x \geq 2$ and r, and a representation of $\mathcal{M}$ over* GF(p) *with $p > x \cdot r \cdot |U|^r$, outputs an $r \times |U|$ matrix A. With probability at least $1 - 1/x$ the matrix A is a representation of $\mathcal{M} \cap \mathcal{U}$ over* GF(p). *Otherwise A represents a matroid $\mathcal{M}'$ such that all independent sets in $\mathcal{M}'$ are independent in $\mathcal{M} \cap \mathcal{U}$.*

Thus we are still left with the question of the computational complexity of ℓ-Matroid Intersection. The answer turns out to be rather surprising; the problem is polynomial time solvable for $\ell = 2$, but NP-complete for larger values of ℓ. We start with showing the hardness proof.

Theorem 12.6. 3-Matroid Intersection *is* NP*-complete.*

Proof. Membership in NP is trivial, and so we prove hardness. We show hardness by reduction from the Hamiltonian Path problem in bipartite graphs. Here, input is a bipartite graph G with bipartitions V_L and V_R. The task is to determine whether there exists a path that visits every vertex of G exactly once. It is well known that Hamiltonian Path remains NP-complete even when the input graph is required to be bipartite [224].

We make three matroids, $\mathcal{M}_L$, $\mathcal{M}_R$ and $\mathcal{M}_G$, all with the edge set $E(G)$ as universe. $\mathcal{M}_L$ is a partition matroid where a subset $A \subseteq E(G)$ is independent if every vertex in V_L is incident to at most two edges in A. Similarly, $\mathcal{M}_R$ is a partition matroid where a subset $A \subseteq E(G)$ is independent if every vertex in V_R is incident to at most two edges in A. Finally $\mathcal{M}_G$ is the graphic matroid of G.

We claim that an edge set $A \subseteq E(G)$ of size at least $n - 1$ is independent in $\mathcal{M}_L$, $\mathcal{M}_R$ and $\mathcal{M}_G$ if and only if A is the edge set of a path in G that visits every vertex exactly once. Indeed, since A is independent in $\mathcal{M}_G$, the set A is acyclic. Since A has size at least $n - 1$ it follows that $(V(G), A)$ is a spanning tree T of G. Since A is independent in both $\mathcal{M}_L$, $\mathcal{M}_R$ it follows that the maximum degree of T is 2. However a spanning tree of maximum degree 2 is a path visiting each vertex exactly once. This concludes the proof. $\square$

A polynomial-time algorithm for the 2-Matroid Intersection problem follows from an algorithm for a slightly more general problem called Matroid Parity. Here, we are given as input a matrix A representing a matroid $\mathcal{M} = (U, \mathcal{F})$, a collection $\mathcal{P} \subseteq \binom{U}{2}$ of pairs of elements of U, and an integer k. The objective is to find a subset $\mathcal{S} \subseteq \mathcal{P}$ of size exactly k such that the set $\bigcup_{X \in \mathcal{S}} X$ has size $2k$ and is independent in $\mathcal{M}$. Note that the size constraint is equivalent to demanding that all pairs in $\mathcal{S}$ be pairwise disjoint. Thus it is sufficient to find a subset $\mathcal{S} \subseteq \mathcal{P}$ of size *at least* k such that all pairs in $\mathcal{S}$ are pairwise disjoint and the set $\bigcup_{X \in \mathcal{S}} X$ is independent in $\mathcal{M}$. A polynomial-time algorithm for Matroid Parity is out of scope of this book; we will simply use the existence of such an algorithm as a black box.

Theorem 12.7. Matroid Parity *admits a polynomial-time algorithm.*

In Exercise 12.7, the reader is asked to show how Theorem 12.7 implies a polynomial-time algorithm for 2-Matroid Intersection.

It is worth noting that Matroid Parity is a more complex problem than 2-Matroid Intersection. While a polynomial time deterministic algorithm for 2-Matroid Intersection can be found in textbooks [412] and is often covered in courses on combinatorial optimization, the deterministic algorithms for Matroid Parity are quite complex. A possible reason is that 2-Matroid Intersection generalizes Bipartite Matching, while Matroid Parity can be seen as a generalization of Maximum Matching.

In the Maximum Matching problem we are given as input a graph G and asked to find a largest possible *matching* in G, i.e. a set $S \subseteq E(G)$ such that no two edges in S share a common endpoint. In the Bipartite Matching problem the input graph G is required to be bipartite. Bipartite Matching can easily be solved in polynomial time by a reduction to Maximum Flow (if you have not yet seen this reduction, try making one yourself). Maximum Matching is also known to be solvable in polynomial time [412], but the algorithms for this problem are more advanced.

Bipartite Matching reduces to 2-Matroid Intersection in the following way. Let G be the input bipartite graph with bipartitions A and B. We make two matroids, $\mathcal{M}_A$ and $\mathcal{M}_B$, both with edge set $E(G)$. An edge subset S of $E(G)$ is independent in $\mathcal{M}_A$ if no vertex of A is incident to more than one edge of S. Similarly, an edge subset S of $E(G)$ is independent in $\mathcal{M}_B$ if no vertex of B is incident to more than one edge of S. Note that both $\mathcal{M}_A$ and $\mathcal{M}_B$ are, by construction, partition matroids. Further, a set S of edges is simultaneously independent in $\mathcal{M}_A$ and in $\mathcal{M}_B$ if and only if S is a matching in G.

We now reduce Maximum Matching to Matroid Parity. Given a graph G we make a matroid $\mathcal{M}$ whose universe is $V(G)$, and *every* vertex subset is independent. From the edge set $E(G)$ we make the collection $\mathcal{P}$ of pairs in the Matroid Parity instance in the most natural way; each edge uv corresponds to the pair $\{u, v\} \in \mathcal{P}$. Now, a subset S of edges is a matching if and only if the corresponding subset $\mathcal{S} \subseteq \mathcal{P}$ satisfies that all pairs in $\mathcal{S}$ are pairwise disjoint, and that $\bigcup_{X \in \mathcal{S}} X$ is independent in $\mathcal{M}$. Here, the second part of the condition on $\mathcal{S}$ always holds.

2-Matroid Intersection generalizes Bipartite Matching, while Matroid Parity generalizes Maximum Matching and 2-Matroid Intersection. Matroid Parity is solvable in polynomial time, and hence the other three problems are as well.

Another important difference between 2-Matroid Intersection and Matroid Parity is that the two problems behave very differently in the oracle model. In the oracle model the input matroids are not required to be linear, and the algorithm is not given any representation of the matroids as input. The only way an algorithm may receive information about the input matroids is by making queries on the form “is the edge set A independent?”. 2-Matroid Intersection remains polynomial time solvable in the oracle model [109, 161], while Matroid Parity requires an exponential number of oracle calls [278, 332].

12.2.2 Feedback Vertex Set *in subcubic graphs*

Recall the Feedback Vertex Set problem, where we are given as input a graph G and an integer k, and the task is to find a subset S of $V(G)$ such that $G-S$ is acyclic. In Chapter 4, we gave a $3.619^k n^{\mathcal{O}(1)}$-time algorithm for Feedback Vertex Set, under the assumption that a variant of this problem is solvable in polynomial time. Specifically, in the Special Disjoint FVS problem, we are given as input a graph G and a vertex set $W \subseteq V(G)$ such that $G - W$ is independent and every vertex not in W has degree at most 3. The objective is to find a smallest possible set $S \subseteq V(G) \setminus W$ such that $G - S$ is a forest. In Chapter 4, we stated without proof a lemma claiming that Special Disjoint FVS can be solved in polynomial time. We now substantiate this claim.

Lemma 4.10 (rephrased). Special Disjoint FVS *has a polynomial-time algorithm.*

We prove the lemma here under the assumption that the input graph G is simple. Observe that when performing the reductions in Chapter 4, one may create double edges, making the considered graph a multigraph. However, when Lemma 4.10 is invoked, any double edge of G would have one endpoint in W and one outside. Then the endpoint outside of W would have been deleted by Reduction Rule FVS*.2. Thus it is sufficient to prove the lemma for simple graphs.

We start by applying some (but not all) of the reduction rules described in Section 3.3, in order to get a slightly more structured instance. We will use the following reduction rules from Section 3.3.

- Reduction FVS*.1: Delete all the vertices of degree at most 1 of G.
- Reduction FVS*.2: If there exists a vertex v in $V(G) \setminus W$ such that $G[W \cup \{v\}]$ contains a cycle, then include v in the solution and solve the problem on $G - v$.

We introduce a few more reduction rules.

Reduction FVS*.5. If $G[W]$ contains a cycle, then return no.

Reduction FVS*.6. Suppose there is an edge uv with both endpoints in W. Obtain G' from G by contracting the edge uv. In graph G', we put the vertex resulting from the contraction into the set W. Solve the problem on G'.

Safeness of Rule FVS*.5 is trivial, so we only discuss Rule FVS*.6. We only apply Rule FVS*.6 if Rules FVS*.2 and FVS*.5 cannot be applied. Thus the contraction of uv does not create any self-loops or double edges, keeping the graph simple. Next we prove that Rule FVS*.6 is safe.

Lemma 12.8. *Rule FVS**.6 is safe.*

Proof. We claim that for any set $S \subseteq V(G) \setminus W$, graph $G - S$ is acyclic if and only if $G' - S$ is. If $G - S$ is acyclic, then observe that $G' - S$ is a minor of $G - S$. Since a minor of a forest is a forest, $G' - S$ is acyclic as well. Now assume that $G - S$ contains a cycle C; pick a shortest such cycle. If C does not contain both u and v, then C is a cycle in $G' - S$ as well. If C contains both u and v, then u and v appear consecutively on C, since C is a shortest cycle. In this case, contracting the edge uv changes C into a new cycle C' of length 1 less than C. Since the contraction of uv does not make any double edges it follows that the length of C is at least 4, and hence C' is a simple cycle in $G' - S$. $\square$

If we start with an instance of Special Disjoint FVS and exhaustively apply the reduction rules above, we arrive at an instance of (G, W) of the problem where $G[W]$ and $G - W$ are both independent sets, no vertex has degree less than 2, and all vertices of $V(G) \setminus W$ have degree at most 3. Next we get rid of the vertices in $V(G) \setminus W$ of degree 2. Note that the simple argument of Reduction FVS*.3 in Section 4.3 no longer works. We cannot just remove the vertex and make its two neighbors adjacent: as the two neighbors are in W and hence undeletable, this transformation could turn a yes-instance into a no-instance.

Reduction FVS*.7. If G has a degree 2 vertex $v \notin W$, then add v to W.

Lemma 12.9. *Rule FVS*.7 is safe.*

Proof. It is sufficient to prove that there exists an optimal solution S disjoint from the degree 2 vertex v. Let x and y be the neighbors of v. Consider a minimum size feedback vertex set $S \subseteq V(G) \setminus W$. If $v \notin S$, then we are done. Suppose now $v \in S$. Since $S \setminus \{v\}$ is not a feedback vertex set of G, there is a path between x and y in $G - S$. Since $G - S$ is acyclic, this path is unique, and since W is independent, the path contains a vertex $v' \notin W$. We claim that $S' = (S \cup \{v'\}) \setminus \{v\}$ is also a feedback vertex set. Indeed, any cycle in $G - S'$ must contain v and a path between x and y disjoint from v. But the path between x and y in $G - S$ is unique and contains v', contradicting that the cycle was disjoint from S'. $\square$

Let us note that we only apply Rule FVS*.7 when all the preceding rules may not be applied. Furthermore, note that after an application of Rule FVS*.7, $G[W]$ is no longer independent and thus some of the other rules will apply. It is important that we apply Rule FVS*.7 one vertex at a time, rather than inserting all degree 2 vertices into W simultaneously. Indeed, if G was a cycle on four vertices and W contained the first and third vertex on the cycle, simultaneously inserting the remaining two vertices into W would make a no-instance, even though the original instance has a solution of size 1. We are now ready to prove Lemma 4.10.

Proof (of Lemma 4.10). After an exhaustive application of Reductions FVS*.1, FVS*.2, FVS*.5, FVS*.6, and FVS*.7, we arrive at an instance of Special

Disjoint FVS where W is an independent set and every vertex not in W has degree exactly 3. We show how to solve such instances in polynomial time by giving a reduction to the Matroid Parity problem. Let H be a clique with vertex set W and $\mathcal{M}_H$ be the graphic matroid of H. Let A be a matrix representing $\mathcal{M}_H$; such a matrix can easily be computed in polynomial time; see the proof of Theorem 12.2.

We will make an instance of Matroid Parity where the matroid is $\mathcal{M}_H$. We now make a collection $\mathcal{P} \subseteq \binom{W}{2}$ of pairs of elements of W as follows. For each vertex $u \in V(G) \setminus W$, let x, y and z be the neighbors of u. We add the pair $P_u = \{xy, yz\}$ to $\mathcal{P}$. Note that, for two vertices u and v, the pairs P_u and P_v could be identical; we still add both pairs to $\mathcal{P}$. Thus $\mathcal{P}$ could be a multiset (but of course adding more copies of the same pair to $\mathcal{P}$ does not change the solvability of the Matroid Parity instance). If we are looking for a solution S of size t to Special Disjoint FVS we set $k = n - |W| - t$ and use this value in the instance of Matroid Parity. To conclude the proof, we need to show that G has a feedback vertex set $S \subseteq V(G) \setminus W$ of size at most t if and only if there exists a subset $\mathcal{Z} \subseteq \mathcal{P}$ of size at least k such that all pairs in $\mathcal{Z}$ are disjoint and the set $\bigcup_{X \in \mathcal{Z}} X$ is independent in $\mathcal{M}_H$.

There is a natural bijection between subsets of $V(G) \setminus W$ and subsets of $\mathcal{P}$. For a set $X \subseteq V(G) \setminus W$ we define $\mathcal{Z}_X = \{P_u \ : \ u \in X\}$. Clearly, the function that takes X to $\mathcal{Z}_X$ is a bijection. We claim that for any $X \subseteq V(G) \setminus W$, the graph $G[W \cup X]$ is acyclic if and only if $\bigcup_{P_u \in \mathcal{Z}_X} P_u$ is independent in the matroid $\mathcal{M}_H$ and all pairs in $\mathcal{Z}_X$ are disjoint.

Suppose first the pairs in $\mathcal{Z}_X$ are not pairwise disjoint. Then there are two pairs P_u and P_v in $\mathcal{Z}_X$ that both contain the edge xy of H. In this case, u, x, v, y forms a cycle on four vertices in $G[W \cup X]$. Suppose now that the pairs in $\mathcal{Z}_X$ are pairwise disjoint, but that $\bigcup_{P_u \in \mathcal{Z}_X} P_u$ is not independent in $\mathcal{M}_H$. Then this set contains a circuit $C = c_1 c_2 \ldots c_\ell$. Here, the c_i's are vertices in W. For each $0 \leq i < \ell$, let u_i be the vertex in X such that the pair corresponding to u_i contains the edge $c_i c_{i+1}$. Since the pairs are disjoint, u_i is uniquely defined, and adjacent to both c_i and to c_{i+1}. Similarly, let u_ℓ be the vertex in X whose pair contains $c_\ell c_1$. The vertices $u_1, \ldots, u_\ell$ are not necessarily distinct. On the other hand they cannot all be equal, since C is a simple cycle on at least three edges, while each vertex u_i corresponds to a pair that contains at most two edges of C. Therefore there exists an i such that $u_i \neq u_{i+1}$. But then $u_i c_{i+1} u_{i+1}$ is a simple path on three vertices, and

$$u_{i+1} c_{i+2} u_{i+2} c_{i+3} u_{i+3} \ldots c_\ell u_\ell c_1 u_1 c_2 u_2 \ldots c_i u_i$$

is a walk from u_{i+1} to u_i in $G[W \cup X]$ avoiding c_i. Consequently, $G[W \cup X]$ contains the walk $u_i c_{i+1} u_{i+1}$ and a walk back to u_i avoiding c_i, so it contains a cycle.

For the reverse direction, suppose now that $G[W \cup X]$ contains a cycle C. Since $G[W \cup X]$ is bipartite, we have that $C = c_1 u_1 c_2 u_2 \ldots c_\ell u_\ell$, where each c_i is in W while each u_i is in X. Let $\mathcal{Z}_C \subseteq \mathcal{Z}_X$ be the set of pairs corresponding to $\{u_1, u_2, \ldots, u_\ell\}$, and define $E_C = \bigcup_{P_u \in \mathcal{Z}_C} P_u$. The set E_C is a set of edges in H. If two of the pairs in $\mathcal{Z}_C$ have nonempty intersections, then we are done, so assume that all the pairs in $\mathcal{Z}_C$ are disjoint. Then $|E_C| = 2\ell$. We now count the vertices in W that are incident to at least one edge in E_C. This is all of the vertices $c_1, c_2, \ldots, c_\ell$, plus at most one more vertex for each u_i, since the pair corresponding to u_i is incident to c_i, c_{i+1} and one more vertex. Hence there are at most 2ℓ vertices incident to edges in E_C. Any graph on 2ℓ edges and at most 2ℓ vertices must contain a cycle, and so we conclude that the set E_C is not independent in $\mathcal{M}_G$.

Finally, observe that if G has a feedback vertex set $S \subseteq V(G) \setminus W$ of size at most t, then $X = V(G) \setminus (W \cup S)$ is a set of size at least $n - |W| - t = k$ such that $G[W \cup X]$ is acyclic. Similarly, if $X \subseteq V(G) \setminus S$ is a set of size at least k such that $G[W \cup X]$ is acyclic, then $S = V(G) \setminus (W \cup X)$ is a feedback vertex set of size at most t. This concludes the proof. □

A nice consequence of Lemma 4.10 is that Feedback Vertex Set is polynomial time solvable on graphs of maximum degree 3.

Theorem 12.10. Feedback Vertex Set *admits a polynomial-time algorithm on graphs of maximum degree* 3.

Proof. Given as input a graph G of maximum degree 3, we make a new graph G' from G by replacing every edge by a degree 2 vertex whose neighbors are the endpoints of the edge. We set W to be the set of newly introduced vertices. A subset S of $V(G)$ is a feedback vertex set of G if and only if it is a feedback vertex set of G'. Thus we can run the polynomial-time algorithm of Lemma 4.10 on the Special Disjoint FVS instance (G', W). □

12.3 Representative sets

Consider the following game, played by two players, Alice and Bob. Alice has a family $\mathcal{A}$ of sets; all of Alice's sets have the same size p. Bob, on the other hand, has only one set B of size q. Alice does not know Bob's set B, but she does know q. The game is very simple; they sit around for a while in silence, and then Bob reveals B to Alice. At this point Alice wins if she has a set $A \in \mathcal{A}$ such that A is disjoint from B. If Alice does not have such a set, then Bob wins.

Of course this is not much of a game; the outcome is predetermined by the family dealt to Alice and the set dealt to Bob. However, there is a catch. Alice is blessed with poor memory, so she really does not want to remember all of $\mathcal{A}$. In particular she will choose a subset $\mathcal{A}'$ of $\mathcal{A}$ to remember and forget the rest. When Bob's set B is revealed she can only look in $\mathcal{A}'$ for a set that is disjoint from B. Even though Alice has poor memory she really hates losing to Bob, and so she does not want to forget a set $A \in \mathcal{A}$ if that could make the difference between winning and losing the game. Thus she wants to remember a subset $\mathcal{A}' \subseteq \mathcal{A}$ such that for any set B of size q, if there is an $A \in \mathcal{A}$ that is disjoint from B, then there is an A' in $\mathcal{A}'$ that also is disjoint from B. The family $\mathcal{A}'$ "represents" $\mathcal{A}$ in the sense that $\mathcal{A}$ and $\mathcal{A}'$ win against exactly the same sets B. This brings us to the definition of representative sets.

Definition 12.11. Let $\mathcal{A}$ be a family of sets of size p. A subfamily $\mathcal{A}' \subseteq \mathcal{A}$ is said to q-represent $\mathcal{A}$ if for every set B of size q such that there is an $A \in \mathcal{A}$ that is disjoint from B, there is an $A' \in \mathcal{A}'$ that also is disjoint from B. If $\mathcal{A}'$ q-represents $\mathcal{A}$, we write $\mathcal{A}' \subseteq^q_{\mathrm{rep}} \mathcal{A}$.

Is there a nontrivial upper bound on the number of sets that Alice has to remember? And if so, does there exist an efficient algorithm that given the family $\mathcal{A}$ and q decides which sets to remember and which to forget? The answer to both questions turns out to be yes, as we soon shall see.

Let's ask a different question. Suppose $\mathcal{A} = \{A_1, A_2, \ldots A_m\}$. Is there a necessary condition for Alice *not* to be able to forget A_1? Well, forgetting A_1 should make the difference between winning and losing for at least one possible set B_1. In particular there has to exist a set B_1 of size q such that A_1 is disjoint from B_1, and B_1 has nonempty intersection with every other set $A_i \in \mathcal{A}$, $i \neq 1$. Suppose now that Alice cannot forget any set $A_i \in \mathcal{A}$. By the reasoning above, for each $i \leq m$ there exists a set B_i such that $B_i \cap A_i = \emptyset$ and $B_j \cap A_i \neq \emptyset$ for all $j \neq i$. Observe that if $i \neq j$ the sets B_i and B_j must be distinct, since $B_i \cap A_i = \emptyset$, while $B_j \cap A_i \neq \emptyset$. The next lemma from extremal combinatorics, called *Bollobás' lemma*, upper bounds m in terms of p and q, thus giving an upper bound on the number of sets Alice has to remember.

Lemma 12.12. *Let $\mathcal{A} = \{A_1, A_2 \ldots A_m\}$ be a family of sets of size p, and $\mathcal{B} = \{B_1, B_2 \ldots B_m\}$ be a family of sets of size q, such that for every i, $A_i \cap B_i = \emptyset$, and for every i,j such that $i \neq j$, $A_i \cap B_j \neq \emptyset$. Then $m \leq \binom{p+q}{p}$.*

Before we give a proof of Lemma 12.12 let us note the consequences it has for representative sets. By the reasoning above, if Alice has the family $\mathcal{A} = \{A_1, A_2, \ldots A_m\}$ and cannot forget any single set A_i, then there exists a family $\mathcal{B} = \{B_1, B_2 \ldots B_m\}$ of sets of size q that satisfies the conditions of Lemma 12.12. But then, the lemma implies that $|\mathcal{A}| \leq \binom{p+q}{p}$. If Alice has more than $\binom{p+q}{p}$ sets then there exists one set that she may safely forget. Formally, for every family $\mathcal{A}$ of sets of size p there is a subfamily $\mathcal{A}' \subseteq^q_{rep} \mathcal{A}$ of size at most $\binom{p+q}{p}$. We are now ready to give the proof of Lemma 12.12. This proof is a surprising application of the probabilistic method.

Lemma 12.12 implies that Alice never needs to remember more than $\binom{p+q}{p}$ sets.

Proof (of Lemma 12.12). Consider a random permutation σ of the universe. For each $i \leq m$, let "event i" be the event that all of A_i appears before all of B_i according to σ. The probability that event i occurs is exactly

$$\frac{1}{\binom{p+q}{p}}.$$

Suppose now that all of A_i appears before all of B_i. Then all of A_j may not appear before all of B_j for $j \neq i$. This is because $B_j \cap A_i \neq \emptyset$ and $A_j \cap B_i \neq \emptyset$, and hence the elements in $B_j \cap A_i$ appear before the elements in $A_j \cap B_i$. Thus, for $i \neq j$, event i and event j are disjoint. In a probability distribution, the sum of the probabilities of all the different outcomes is 1. Thus the sum of the probabilities of disjoint events is at most 1, and hence

$$m \cdot \frac{1}{\binom{p+q}{p}} \leq 1.$$

Multiplying both sides by $\binom{p+q}{p}$ proves the lemma. $\square$

At this point we arrive at an intriguing computational problem. We are given as input a family $\mathcal{A}$ of sets of size p, and an integer q. We know from Lemma 12.12 that there exists a family $\mathcal{A}' \subseteq \mathcal{A}$ of size at most $\binom{p+q}{p}$ such that $\mathcal{A}'$ q-represents $\mathcal{A}$, but how do we find such an $\mathcal{A}'$? The derivation of $\mathcal{A}'$ from $\mathcal{A}$ seems "nonconstructive" in nature because the family $\mathcal{B}$ is not given to us as input. It is in fact possible to compute for each A_i whether there exists a set B_i of size q disjoint from A_i, but intersecting all other sets in $\mathcal{A}$. This amounts to solving a d-Hitting Set instance with $d = p$. This gives an algorithm for computing $\mathcal{A}'$ from $\mathcal{A}$. This algorithm is explored in Exercise 12.9. However, we will soon give a faster algorithm for a more general variant of the problem using completely different techniques.

12.3.1 Playing on a matroid

We now slightly change the game played by Alice and Bob. The board is now a matroid $\mathcal{M}$ with universe U and a family $\mathcal{F}$ of independent sets. Alice has a family $\mathcal{A}$ of subsets of U; all of her sets have the same size p. Bob has a set $B \subseteq U$. In the previous game, all Alice had to do was to come up with a set which was disjoint from B. We make Alice's task a bit harder.

Definition 12.13. We will say that A *fits* with B if A is disjoint from B and $A \cup B$ is independent.

When Bob reveals his set B, Alice wins if she has a set A which fits B.

Just as before, Alice is to remember a subfamily $\mathcal{A}' \subseteq \mathcal{A}$, and when B is revealed she is only allowed to look for A in $\mathcal{A}'$. Just as before, she does not want to forget any sets in $\mathcal{A}$ if that could make the difference between winning and losing the game. This brings us to a generalized notion of representative sets, tuned to this new setting.

Definition 12.14. Let $\mathcal{M}$ be a matroid and $\mathcal{A}$ be a family of sets of size p in $\mathcal{M}$. A subfamily $\mathcal{A}' \subseteq \mathcal{A}$ is said to q-represent $\mathcal{A}$ if for every set B of size q such that there is an $A \in \mathcal{A}$ that fits B, there is an $A' \in \mathcal{A}'$ that also fits B. If $\mathcal{A}'$ q-represents $\mathcal{A}$, we write $\mathcal{A}' \subseteq^q_{\mathrm{rep}} \mathcal{A}$.

First, observe that because of the hereditary property of matroids, Alice never needs to remember any sets in $\mathcal{A}$ that are not independent in the matroid. Similarly she may assume that Bob's set B is independent, since otherwise surely no set in $\mathcal{A}$ fits B.

Compare Definition 12.14 with Definition 12.11. Definition 12.14 makes Alice's job harder, because Alice now has to remember sets that not only are disjoint from B, but fit B as well. At the same time Definition 12.14 makes Alice's job *easier*, because she only needs to consider those sets B that fit some set in $\mathcal{A}$, instead of all the sets that are disjoint from some set in $\mathcal{A}$. Observe also that if the matroid $\mathcal{M}$ is the *uniform* matroid of rank at least $p+q$ then the two definitions coincide. Thus Definition 12.14 is truly a generalization of Definition 12.11.

If we use representative sets without defining which matroid we are working with, then the matroid in question is the uniform matroid of rank $p+q$, and Definition 12.14 and Definition 12.11 coincide.

For the first game we showed that Alice only needs to remember $\binom{p+q}{p}$ sets. What about this new game? Next we will prove that even in this new setting, remembering $\binom{p+q}{p}$ sets is sufficient, and that we can efficiently compute which sets to remember and which to forget.

In the following we will refer to a family of sets of size p by a *p-family*. The constant ω is the *matrix multiplication constant*, that is, the infinimum of the set of reals c such that multiplication of two $n \times n$ matrices can be done in $\mathcal{O}(n^c)$ time. It was shown by Vassilevska Williams [425] that $\omega < 2.373$.

Theorem 12.15. *There is an algorithm that, given a matrix M over a field* $\mathrm{GF}(s)$, *representing a matroid $\mathcal{M} = (U, \mathcal{F})$ of rank k, a p-family $\mathcal{A}$ of independent sets in $\mathcal{M}$, and an integer q such that $p + q = k$, computes a q-representative family $\mathcal{A}' \subseteq^q_{\mathrm{rep}} \mathcal{A}$ of size at most $\binom{p+q}{p}$ using at most $\mathcal{O}(|\mathcal{A}|(\binom{p+q}{p}p^{\omega} + \binom{p+q}{p}^{\omega-1}))$ operations over* $\mathrm{GF}(s)$.

Theorem 12.15 implies that Alice never needs to remember more than $\binom{p+q}{p}$ sets, and that she can compute which sets to remember and which to forget in time polynomial in $|\mathcal{A}|$.

In the proof of Theorem 12.15, we assume that the matrix M has full row rank, i.e., that M has exactly k rows. This assumption is justified by the discussion at the end of Section 12.1.1. Before commencing with the proof of Theorem 12.15, we review a useful formula for the determinant of a matrix.

For a matrix M with k rows and a subset I of $\{1, \ldots, k\}$, we define $M[I]$ to be the submatrix of M containing all the rows indexed by I. We will denote $\{1, \ldots, k\} \setminus I$ by $\bar{I}$, and use a shorthand $\sum I$ for $\sum_{i \in I} i$.

Proposition 12.16 (Generalized Laplace expansion). *Let M_A be a $(p+q) \times p$ matrix, and M_B be a $(p+q) \times q$ matrix. Then,*

$$\det([M_A|M_B]) = (-1)^{\lceil p/2 \rceil} \sum_{I \subseteq [p+q], |I|=p} (-1)^{\sum I} \det(M_A[I]) \det(M_B[\bar{I}]).$$

By $[M_A|M_B]$ we mean the matrix obtained from M_A and M_B by appending for each $i \leq p+q$ the i-th row of M_B to the i-th row of M_A.

The proof of the Generalized Laplace expansion can be found in most textbooks on linear algebra (see, for example, [371]) and we omit it here.

How does the Generalized Laplace expansion of determinants relate to representative sets? In fact, the relationship is quite direct. For an edge set A, define the matrix M_A to be the submatrix of the representation matrix M containing the columns of M corresponding to elements of A. We have the following observation.

Observation 12.17. An edge set A of size p fits an edge set B of size q if and only if the determinant of $[M_A|M_B]$ is nonzero.

Proof. Suppose A fits B. Then the $p+q$ columns of M corresponding to $A \cup B$ are linearly independent, and hence $\det[M_A|M_B] \neq 0$. If $A \cap B \neq \emptyset$ then there is a column that appears both in M_A and in M_B. Hence the columns of $[M_A|M_B]$ are not linearly independent and so $\det[M_A|M_B] = 0$. Furthermore, by the definition of representation matrices, if A and B are disjoint but $A \cup B$ is not independent, then the columns of $[M_A|M_B]$ are linearly dependent and $\det[M_A|M_B] = 0$. □

Suppose we have in our possession an edge set A of size p, and know that in the future an edge set B of size q will arrive. When B arrives, we are going to check whether A fits B by making use of Observation 12.17. In particular we will compute $\det([M_A|M_B])$ using the Generalized Laplace Expansion. Even though we do not know B yet, we can perform quite a lot of this computation just with the knowledge of the matrix M, the set A, p and q.

Order the $\binom{p+q}{p}$ subsets I of $\{1, \ldots, k\}$ of size exactly p into $I_1, I_2, \ldots I_\ell$, where $\ell = \binom{p+q}{p}$. For an edge set A of size p, define a $\binom{p+q}{p}$ dimensional vector v^A, where the i-th entry of v^A is

$$v_i^A = \det(M_A[I_i]).$$

For each $i \leq \ell$, we have that $\bar{I}_i = \{1, \ldots, k\} \setminus I_i$ is a set of size q. For each edge set B of size q, we define a $\binom{p+q}{q} = \binom{p+q}{p}$ dimensional vector $\bar{v}^B$, where the i-th entry of $\bar{v}^B$ is

$$\bar{v}_i^B = \det(M_B[\bar{I}_i]).$$

Finally, for ease of notation, we define

$$\sigma_i = (-1)^{\sum I_i}.$$

The following observation follows directly from Observation 12.17 and the Generalized Laplace expansion of $\det([M_A|M_B])$.

Observation 12.18. An edge set A of size p fits an edge set B of size q if and only if $\sum_{i=1}^{\ell} \sigma_i v_i^A \bar{v}_i^B \neq 0$.

Knowing A, we may precompute v^A. When B is revealed we compute $\bar{v}^B$ from B, and then test whether $\sum_{i=1}^{\ell} \sigma_i v_i^A \bar{v}_i^B$ is nonzero. If it is nonzero then A fits B, otherwise A does not fit B.

We are now ready to prove Theorem 12.15.

Proof (of Theorem 12.15). Given M, $\mathcal{A}$, p, and q, the algorithm computes for each $A \in \mathcal{A}$ the vector v^A. This takes time $\mathcal{O}(|\mathcal{A}|\binom{p+q}{p}p^\omega)$, since we need to compute $|\mathcal{A}|\binom{p+q}{p}$ determinants of $p \times p$ matrices, and computing the determinant of an $n \times n$ matrix can be done in $\mathcal{O}(n^\omega)$ time.

The algorithm arranges the vectors $\{v^A : A \in \mathcal{A}\}$ in a $\binom{p+q}{p} \times |\mathcal{A}|$ matrix X, and finds a column basis Z of X. Since X has $\binom{p+q}{p}$ rows, it follows that the size of the column basis Z is at most $\binom{p+q}{p}$. Finding a column basis of an $n \times m$ matrix can be done in time $\mathcal{O}(mn^{\omega-1})$ (see [48]), thus it takes $|\mathcal{A}|\binom{p+q}{p}^{\omega-1}$ time to find the basis Z. The algorithm outputs the set

$$\mathcal{A}' = \{A : v^A \in Z\}.$$

We remark that the given time bounds are in terms of the number of field operations (additions and multiplications) that we need to perform. When working with integers modulo a prime s, this incurs an overhead of $\mathcal{O}(\log s \log\log s)$ in the running time.

Since $|Z| \leq \binom{p+q}{p}$, it follows that $|\mathcal{A}'| \leq \binom{p+q}{p}$. It remains to show that $\mathcal{A}'$ actually q-represents $\mathcal{A}$. Consider any set B of size q that fits some set

$A \in \mathcal{A}$. We need to show that there is a set $A' \in \mathcal{A}'$ that also fits B. By Observation 12.18, we have that $\sum_{i=1}^{\ell} \sigma_i v_i^A \bar{v}_i^B \neq 0$.

Since Z is a column basis for the matrix X, and v^A is a column of X, it follows that we can assign a field element $\lambda_{A'}$ for each $A' \in \mathcal{A}'$ such that

$$v_A = \sum_{C \in \mathcal{A}'} \lambda_{A'} v^{A'}.$$

We then have that

$$0 \neq \sum_{i=1}^{\ell} \sigma_i v_i^A \bar{v}_i^B = \sum_{i=1}^{\ell} \sum_{A' \in \mathcal{A}'} \sigma_i \lambda_{A'} v_i^{A'} \bar{v}_i^B = \sum_{A' \in \mathcal{A}'} \lambda_{A'} \sum_{i=1}^{\ell} \sigma_i v_i^{A'} \bar{v}_i^B.$$

This means that there has to be at least one $A' \in \mathcal{A}'$ such that $\sum_{i=1}^{\ell} \sigma_i v_i^{A'} \bar{v}_i^B \neq 0$. But then Observation 12.18 implies that A' fits B, completing the proof. $\square$

Note that Theorem 12.15 requires the matroid $\mathcal{M}$ to have rank exactly $p+q$. If $\mathcal{M}$ has larger rank, one can first truncate it to rank $p+q$ by taking the intersection of $\mathcal{M}$ with a uniform matroid of rank $p+q$, and compute representative sets in this new matroid, which has rank $p+q$. To compute a representation matrix of the intersection of $\mathcal{M}$ and the uniform matroid, we may use Lemma 12.5. Using Lemma 12.5 has the consequence that the algorithm for computing representative sets becomes randomized. Furthermore, Lemma 12.5 only works if $\mathcal{M}$ has a representation over a sufficiently large field. These issues will not show up when dealing with the problems considered in this chapter, but they do show up in more complex applications [310] of representative sets.

12.3.2 Kernel for d-Hitting Set

In Chapter 2, we gave a kernel for the d-Hitting Set problem of size $\mathcal{O}(k^d \cdot d! \cdot d)$ based on the sunflower lemma. We now show how representative sets can be used to get a kernel of size $\mathcal{O}(\binom{k+d}{d} \cdot d)$. This is upper bounded by $\mathcal{O}(k^d \cdot \frac{2^d d}{d!}) = \mathcal{O}(k^d)$. Hence the size bound of this kernel has the same exponent as the sunflower lemma-based kernel, but with a better dependence on d in the constant factor.

In the d-Hitting Set problem we are given as input a family $\mathcal{A}$ over a universe U, together with an integer k, and all the sets in $\mathcal{A}$ have size *at most d*. Using Exercise 2.27, we may alternatively focus on the Ed-Hitting Set problem, where every set in $\mathcal{A}$ is of size *exactly d*. The objective is to determine whether there is a set $B \subseteq U$ of size at most k such that B has nonempty intersection with all sets in $\mathcal{A}$.

We model the Ed-Hitting Set problem as a game between Alice and Bob, just as described in the beginning of Section 12.3. The board is the uniform matroid with edge set U and rank $k+d$. Hence a set A of size d fits a set B of size k if and only if they are disjoint.

Alice is trying to prove that the Ed-Hitting Set instance does not have a solution, while Bob is trying to demonstrate a solution to Alice. In the beginning, Alice gets the family $\mathcal{A}$, while Bob secretly thinks of a tentative solution set $B \subseteq U$ of size k. Then Bob reveals B to Alice, and Alice either finds a set $A \in \mathcal{A}$ that fits B, proving that B is not a hitting set, or if no such A exists, concedes that B is indeed a solution. Just as in Section 12.3, Alice does not have to remember all of $\mathcal{A}$; a k-representative subfamily $\mathcal{A}' \subseteq^k_{\mathrm{rep}} \mathcal{A}$ suffices.

Theorem 12.19. *Both* Ed-Hitting Set *and* d-Hitting Set *admit kernels with at most* $\binom{k+d}{d}$ *sets and at most* $\binom{k+d}{d} \cdot d$ *elements.*

Proof. We first focus on Ed-Hitting Set. Given the instance $(\mathcal{A}, U, k)$, we compute a k-representative subfamily $\mathcal{A}'$ of $\mathcal{A}$ of size at most $\binom{k+d}{d}$ using Theorem 12.15. To invoke Theorem 12.15, we need a representation matrix of the uniform matroid with edge set U and rank $k+d$; such a representation matrix may be computed in polynomial time (see Section 12.1.2). Finally we remove from U all the elements that do not appear in any set of $\mathcal{A}'$. Let U' be the resulting universe; clearly $|U'| \leq \binom{k+d}{d} \cdot d$. The kernelization algorithm outputs the instance $(\mathcal{A}', U', k)$. It remains to prove that $\mathcal{A}$ has a hitting set of size at most k if and only if $\mathcal{A}'$ does.

First, any set of size at most k that has nonempty intersection with all sets in $\mathcal{A}$ also has nonempty intersection with all sets in $\mathcal{A}'$. For the reverse direction, suppose that there is a hitting set B of size k for $\mathcal{A}'$, but that B is not a hitting set of $\mathcal{A}$. If B is not a hitting set of $\mathcal{A}$, then there is a set $A \in \mathcal{A}$ that fits B. But $\mathcal{A}'$ k-represents $\mathcal{A}$ and hence there is an $A' \in \mathcal{A}'$ that fits B. This contradicts that B is a hitting set of $\mathcal{A}'$.

For d-Hitting Set, observe that the reduction of Exercise 2.27 does not change the number of sets in an instance. □

12.3.3 Kernel for d-Set Packing

In the d-Set Packing problem we are given as input a family $\mathcal{A}$ over a universe U, together with an integer k. All the sets in $\mathcal{A}$ have size at most d. The objective is to determine whether there is a subfamily $\mathcal{S} \subseteq \mathcal{A}$ of size k such that the sets in $\mathcal{S}$ are pairwise disjoint. In Ed-Set Packing, every set of $\mathcal{A}$ is required to have size *exactly* d.

In Chapter 2 (Exercise 2.29) you are asked to make a kernel for d-Set Packing using the sunflower lemma. This kernel has size $\mathcal{O}(d!d^dk^d)$. Using representative sets, we give a kernel for Ed-Set Packing with $\binom{kd}{d} \leq e^dk^d$

sets and at most $\binom{kd}{d} \cdot d \leq e^d dk^d$ elements; by using Exercise 2.30, the same kernel bounds hold for the d-Set Packing problem. Thus the kernel given here has the same exponent as the kernel from Chapter 2, but the kernel given here has better dependence on d in the constant.

In the kernel for Ed-Hitting Set in Section 12.3.2, the input family $\mathcal{A}$ was a family of *constraints*, and we were trying to find irrelevant constraints that could be dropped without turning a no-instance into a yes-instance. For Ed-Set Packing the family $\mathcal{A}$ represents *possibilities*, since $\mathcal{A}$ contains the sets we can select into the solution. We are now interested in forgetting some of these possibilities while making sure we do not turn a yes-instance into a no-instance.

Theorem 12.20. *Both* Ed-Set Packing *and* d-Set Packing *admit kernels with at most* $\binom{kd}{d}$ *sets and at most* $\binom{kd}{d} \cdot d$ *elements.*

Proof. As discussed earlier, it suffices to focus on Ed-Set Packing. Given the instance $(\mathcal{A}, U, k)$, we compute a $(k-1)d$-representative subfamily $\mathcal{A}'$ of $\mathcal{A}$ of size at most $\binom{kd}{d}$ using Theorem 12.15. When invoking Theorem 12.15, the matroid is the uniform matroid with edge set U and rank kd. Then we remove from U all elements that do not appear in any set of $\mathcal{A}'$. Let U' be the resulting universe; clearly $|U'| \leq \binom{kd}{d} \cdot d$. The kernelization algorithm outputs the instance $(\mathcal{A}', U', k)$.

Clearly, any subfamily $\mathcal{S} \subseteq \mathcal{A}'$ of size k such that the sets in $\mathcal{S}$ are pairwise disjoint is also such a subfamily of $\mathcal{A}$. It remains to prove that if there is a subfamily $\mathcal{S} \subseteq \mathcal{A}$ of size k such that the sets in $\mathcal{S}$ are pairwise disjoint, then there is such a subfamily $\mathcal{S}' \subseteq \mathcal{A}'$. Suppose there is a subfamily $\mathcal{S} \subseteq \mathcal{A}$ of size k such that the sets in $\mathcal{S}$ are pairwise disjoint. Of all such families, pick the one with most sets in $\mathcal{A}'$. Suppose for contradiction that $\mathcal{S}$ contains a set $A \in \mathcal{A} \setminus \mathcal{A}'$. Define

$$B = \bigcup_{S \in \mathcal{S} \setminus \{A\}} S.$$

Since all sets in $\mathcal{S}$ are pairwise disjoint, it follows that A fits B. But then there exists a set $A' \in \mathcal{A}'$ that also fits B. Then A' is disjoint from all sets in $\mathcal{S} \setminus \{A\}$. Consequently, $\mathcal{S}' = (\mathcal{S} \cup \{A'\}) \setminus \{A\}$ is a subfamily of $\mathcal{A}$ of size k such that the sets in $\mathcal{S}'$ are pairwise disjoint. Furthermore, $\mathcal{S}'$ has more sets from $\mathcal{A}'$ than $\mathcal{S}$ does, contradicting the choice of $\mathcal{S}$. We conclude that $\mathcal{S} \subseteq \mathcal{A}'$, completing the proof. $\square$

12.3.4 ℓ-Matroid Intersection

In Section 12.2.1 we showed that ℓ-Matroid Intersection is NP-complete for $\ell = 3$. We now give a randomized FPT algorithm for ℓ-Matroid Intersection, parameterized by ℓ and the size k of the independent set searched for. We first give an algorithm for an intermediate problem, Matroid Ed-Set Packing. Here, we are given integers d and k, a matrix M over GF(s) representing a matroid $\mathcal{M}$ of rank kd with universe U, and a family $\mathcal{A}$ of subsets of U, where all sets in $\mathcal{A}$ have the same size d. The objective is to determine whether there is a subfamily $\mathcal{S} \subseteq \mathcal{A}$ of size k such that the sets in $\mathcal{S}$ are pairwise disjoint, and such that $\bigcup_{S \in \mathcal{S}} S$ is independent in the matroid $\mathcal{M}$. Note that the only difference between Matroid Ed-Set Packing and Ed-Set Packing is the last matroid independence constraint. It should therefore come as no surprise that the same techniques that work for Ed-Set Packing also work for Matroid Ed-Set Packing.

Theorem 12.21. *There is an algorithm for* Matroid Ed-Set Packing *that uses* $|\mathcal{A}|^{\mathcal{O}(1)} + k^{\mathcal{O}(dk)}$ *operations over* GF(s) *and at most* $\mathcal{O}((|\mathcal{A}|^{\mathcal{O}(1)} + k^{\mathcal{O}(dk)}) \log s \log\log s)$ *time.*

Proof. Using Theorem 12.15 we compute a $d(k-1)$-representative subfamily $\mathcal{A}' \subseteq \mathcal{A}$ of size at most $\binom{dk}{d} \leq (ek)^d$. This requires at most $|\mathcal{A}|^{\mathcal{O}(1)}$ operations over GF(s). We claim that if there is a solution $\mathcal{S} \subseteq \mathcal{A}$ of size k such that all sets in $\mathcal{S}$ are pairwise disjoint and $\bigcup_{S \in \mathcal{S}} S$ is independent $\mathcal{M}$, then there is such a family $\mathcal{S}' \subseteq \mathcal{A}'$ as well. The proof of this claim is almost identical to the proof of Theorem 12.20 and therefore omitted.

Now we can by brute force iterate over all subfamilies $\mathcal{S}' \subseteq \mathcal{A}'$ of size k, and check whether all the sets in $\mathcal{S}'$ are pairwise disjoint and $\bigcup_{S \in \mathcal{S}} S$ is independent in $\mathcal{M}$. There are at most $\binom{(ek)^d}{d} \leq k^{\mathcal{O}(dk)}$ possible choices for $\mathcal{S}'$. For each choice of $\mathcal{S}'$ we can verify whether $\mathcal{S}'$ is a solution using $(k + d)^{\mathcal{O}(1)}$ operations over GF(s). Thus the second part of the algorithm uses $k^{\mathcal{O}(dk)}$ operations over GF(s). The operations over GF(s) dominate the running time. Each such operation is implemented in $\mathcal{O}(\log s \log\log s)$ time, yielding the claimed running time bound. $\square$

Even though the proof of Theorem 12.21 is basically identical to the proof of Theorem 12.20, we give a kernel for Ed-Set Packing in Theorem 12.20 while Theorem 12.21 only gives an FPT algorithm for Matroid Ed-Set Packing. The reason for the difference is that, in the proof of Theorem 12.21, even after we have reduced the family $\mathcal{A}$ to size $(ek)^d$, we still need to keep a representation of the matroid $\mathcal{M}$. Of course it is sufficient to only keep the part of the representation of $\mathcal{M}$ which corresponds to elements appearing in some set in $\mathcal{A}$. However it is possible that s is so much bigger than k and d, that storing a single field element requires a number of bits which is superpolynomial in k^d. This is not a problem for the FPT algorithm, but prevents us from stating the result as a polynomial kernel.

We are now ready to give an FPT algorithm for ℓ-Matroid Intersection. Here, we are given ℓ representation matrices $M_1, \ldots M_\ell$ (over GF(s)) of matroids $\mathcal{M}_1, \ldots \mathcal{M}_\ell$ over the same universe U, and an integer k. The objective is to determine whether there exists a set $S \subseteq U$ of size k which is simultaneously independent in all the matroids $M_1, \ldots M_\ell$. We give an algorithm for this problem in the case that all the input matrices $M_1, \ldots M_\ell$ have rank exactly k.

Lemma 12.22. *There is an algorithm for ℓ-Matroid Intersection in the case that all input matrices $M_1, \ldots M_\ell$ have the same rank k. The algorithm uses at most $|U|^{\mathcal{O}(1)} + k^{\mathcal{O}(\ell k)}$ operations over* GF(s) *and at most $\mathcal{O}((|U|^{\mathcal{O}(1)} + k^{\mathcal{O}(\ell k)}) \log s \log\log s)$ time.*

Proof. Given as input an instance of ℓ-Matroid Intersection we construct an instance of Matroid Ed-Set Packing with $d = \ell$. First we make ℓ disjoint copies of the universe U, call these copies $U_1, \ldots U_\ell$. Then, for each i, let $\mathcal{M}_i^\star$ be a copy of the matroid $\mathcal{M}_i$, but over the universe U_i rather than U. We may of course use the matrix M_i as a representation matrix of $\mathcal{M}_i^\star$. Let $\mathcal{M}^\star$ be the direct sum of $\mathcal{M}_1^\star, \mathcal{M}_2^\star, \ldots, \mathcal{M}_\ell^\star$. Using the method described in Section 12.1.5, we can obtain a representation matrix $M^\star$ of $\mathcal{M}^\star$ in polynomial time. $\mathcal{M}^\star$ has rank $k\ell$, and will be the matroid in the instance of Matroid Ed-Set Packing.

We now describe the family $\mathcal{A}$. For each element $u \in U$, let $S_u = \{u_1, u_2, \ldots, u_\ell\}$ where u_i is the copy of u in U_i. We set $\mathcal{A} = \{S_u \ : \ u \in U\}$. Clearly all sets in $\mathcal{A}$ are pairwise disjoint. Furthermore, a subset $S \subseteq U$ is simultaneously independent in $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_\ell$ if and only if $\bigcup_{u \in S} S_u$ is independent in $\mathcal{M}^\star$. Therefore the input instance to ℓ-Matroid Intersection is a "yes" instance if and only if there is a subfamily $\mathcal{S} \subseteq \mathcal{A}$ of size k so that all sets in $\mathcal{S}$ are pairwise disjoint and $\bigcup_{S_u \in \mathcal{S}} S_u$ is independent in $\mathcal{M}^\star$. Thus we can run the algorithm of Theorem 12.21 on the produced Matroid Ed-Set Packing instance and return the same answer as this algorithm. Theorem 12.21 yields the claimed running time bound. □

At this point we would like to get rid of the assumption of Lemma 12.22 that the matrices $M_1, \ldots M_\ell$ have the same rank k. One can use Lemma 12.5 to turn all the matrices into matrices of rank at most k, at the cost of making the algorithm randomized. Unfortunately Lemma 12.5 requires the field GF(s) to be big enough in order to work. If s is too small, it is possible to transform the representation matrices $M_1 \ldots M_\ell$ into representation matrices over a larger field GF(s'), where s' is sufficiently large to apply Lemma 12.5 and guarantee small constant error probability. However, this step requires using fields that are not just integers modulo a prime. We omit the details.

Theorem 12.23. *There is a randomized algorithm for ℓ-Matroid Intersection. The algorithm uses at most $\mathcal{O}((|U|^{\mathcal{O}(1)} + k^{\mathcal{O}(\ell k)})(\log s)^{\mathcal{O}(1)})$ time, and produces the correct answer with probability at least $\frac{9}{10}$.*

12.3.5 Long Directed Cycle

We now give an example of how representative sets can be a useful tool in the design of graph algorithms. In particular, we will consider the Long Directed Cycle problem. Here, the input is an n-vertex and m-edge directed graph G and a positive integer k. The question is whether G contains a directed cycle of length *at least* k. In this section, we will give an algorithm for Long Directed Cycle with running time $26.9^k n^{\mathcal{O}(1)}$.

The closely related problem where we are interested in finding a cycle of length *exactly* k can be solved in $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ time by color coding. However, color coding does not seem to be able to handle the Long Directed Cycle problem, because here the *shortest* cycle out of the ones of length at least k could have length much bigger than k. With a cute trick, it is possible to solve Long Directed Cycle in time $2^{\mathcal{O}(k \log k)} n^{\mathcal{O}(1)}$ (see Exercise 5.9), but getting the running time down to $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ is more challenging.

The starting observation is that G has a cycle C of length at least k if and only if G contains a simple path P on k vertices such that one can get back from the endpoint of P to the starting point without having to intersect with P on the way back.

Lemma 12.24. *G has a cycle C on at least k vertices if and only if there exist vertices u, v, a simple path P_1 from u to v on exactly k vertices, and a simple path P_2 from v to u such that $V(P_1) \cap V(P_2) = \{u, v\}$.*

Proof. Given C, let P_1 be the first k vertices of C and let u and v be the first and last vertices of P_1, respectively. Let P_2 be the path back from v to u along the cycle C. For the reverse direction, $P_1 \cup P_2$ forms a cycle on at least k vertices. $\square$

We can use Lemma 12.24 to make an $n^{k+\mathcal{O}(1)}$-time algorithm for Long Directed Cycle. For every choice of u and v and every path P_1 from u to v on exactly k vertices we check in polynomial time whether there is a path P_2 from v to u in $G - (V(P_1) \setminus \{u, v\})$. The algorithm runs in time $n^{k+\mathcal{O}(1)}$ since there are at most n^k choices for P_1.

Is it really necessary to try *all* possible choices for P_1? As we now shall prove, it is in fact sufficient to restrict the choice of P_1 to a relatively small representative family. To that end, we need to define some notation. For every pair of distinct vertices u, v and integer p, we define

$$\mathcal{P}^p_{uv} = \Big\{ X \;:\; X \subseteq V(G), |X| = p, \text{ and there is a directed } uv\text{-path in } G[X] \text{ visiting all vertices of } X \Big\}.$$

For each distinct pair u, v of vertices, each integer $p \leq k$, and integer $q \leq 2k - p$, we will compute a q-representative family

$$\widehat{\mathcal{P}}^{p,q}_{uv} \subseteq^q_{\text{rep}} \mathcal{P}^p_{uv}.$$

The matroid with respect to which we define the q-representative family $\widehat{\mathcal{P}}^{p,q}_{uv}$ is the uniform matroid with edge set $V(G)$ and rank $p+q$. Observe that Theorem 12.15 implies that, for each choice of u, v, p, and q, there *exists* a q-representative subfamily $\widehat{\mathcal{P}}^{p,q}_{uv}$ of $\mathcal{P}^p_{uv}$ of size at most $\binom{p+q}{p} \leq 4^k$. However, directly *computing* $\widehat{\mathcal{P}}^{p,q}_{uv}$ from $\mathcal{P}^p_{uv}$ by using Theorem 12.15 would take time $n^{\mathcal{O}(k)}$, since it would take this much time just to list the families $\mathcal{P}^p_{uv}$. For now, let us not worry about how exactly the representative families $\widehat{\mathcal{P}}^{p,q}_{uv}$ will be computed, and instead focus on how we can use them once we have them.

Lemma 12.25. *If there exists a cycle C of length at least k, then there exist vertices u, v, and a path P_1 such that $V(P_1) \in \widehat{\mathcal{P}}^{k,q}_{uv}$, $q \leq k$, and there exists a path P_2 from v to u with $V(P_1) \cap V(P_2) = \{u, v\}$.*

Proof. Let $C = v_1, v_2, v_3, \ldots, v_r$ be a shortest cycle out of all cycles in G of length at least k. Let P_1' be the path $v_1, v_2, v_3, \ldots, v_k$ and P_2 be the path $v_k, v_{k+1}, \ldots, v_r, v_1$ (see Fig. 12.1). We now define a vertex set B as follows.

$$B = \begin{cases} \{v_{k+1}, v_{k+2}, \ldots, v_{2k}\} & \text{if } r \geq 2k, \\ \{v_{k+1}, v_{k+2}, \ldots, v_r\} & \text{otherwise.} \end{cases}$$

We set $u = v_1$, $v = v_k$ and $q = |B| \leq k$, and observe that $V(P_1') \in \mathcal{P}^k_{uv}$ and that $V(P_1')$ fits B, as they are disjoint. Since $\widehat{\mathcal{P}}^{k,q}_{uv}$ q-represents $\mathcal{P}^k_{uv}$, it follows that there exists a path P_1 such that $V(P_1) \in \widehat{\mathcal{P}}^{k,q}_{uv}$ and $V(P_1)$ fits B.

We claim that $V(P_1) \cap V(P_2) = \{u, v\}$. If $r < 2k$, then this follows immediately, since $V(P_1)$ is disjoint from B and B contains all internal vertices of P_2. Thus, suppose for contradiction that $r \geq 2k$, but that $V(P_1) \cap V(P_2)$ contains a vertex different from u and v.

Let v_i be the first vertex on P_2 such that $v_i \in V(P_1) \cap V(P_2) \setminus \{u, v\}$, and let $P_2^\star$ be the subpath of P_2 from v to v_i. Since $V(P_1)$ fits B, it follows that $i > 2k$ and hence $P_2^\star$ is a path on at least k, but strictly less than $|V(P_2)|$, vertices. Furthermore, the choice of v_i implies that $V(P_1) \cap V(P_2^\star) = \{v, v_i\}$. Let $P_1^\star$ be the subpath of P_1 from v_i to v. Appending $P_1^\star$ to $P_2^\star$ yields a simple cycle $C^\star$ on at least k vertices. However, the number of vertices on $C^\star$ is

$$|V(P_1^\star)| + |V(P_2^\star)| - 2 < |V(P_1)| + |V(P_2)| - 2 = |V(C)|.$$

This contradicts the choice of C. □

Our algorithm for Long Directed Cycle will compute representative families $\widehat{\mathcal{P}}^{p,q}_{uv}$ for each choice of u, v, $p \leq k$, and $q \leq 2k - p$, and then use

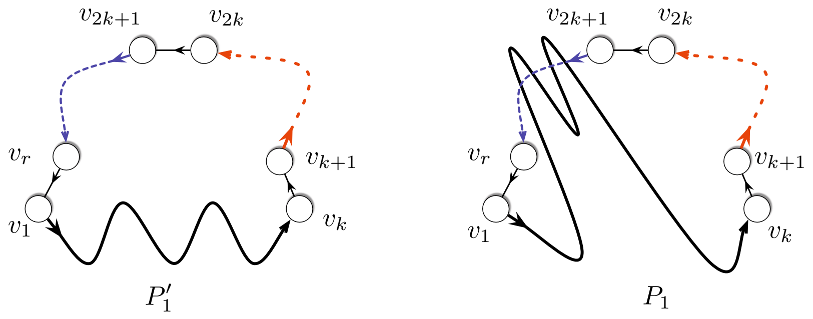


Fig. 12.1: Illustration for the proof of Lemma 12.25.

Lemma 12.25 together with Lemma 12.24 to check whether there is a cycle of length at least k in G. All that remains is to find an efficient algorithm to compute the representative families $\widehat{\mathcal{P}}^{p,q}_{uv}$.

In order to compute the representative families $\widehat{\mathcal{P}}^{p,q}_{uv}$ we will exploit the fact that a simple recurrence holds for the sets of paths $\mathcal{P}^{p}_{uv}$. This recurrence uses the *subset convolution* operation on set families; we now introduce this operation. Given two families $\mathcal{A}$ and $\mathcal{C}$ the *convolution* of $\mathcal{A}$ and $\mathcal{C}$ is a new family $\mathcal{A} * \mathcal{C}$ defined as follows.

$$\mathcal{A} * \mathcal{C} = \{A \cup C \ : \ A \in \mathcal{A}, C \in \mathcal{C}, \text{ and } A \cap C = \emptyset\}.$$

In Section 10.3, we defined the subset convolution operation for functions that take as input a subset of a universe and output an integer. The relationship between subset convolution for set families defined here and subset convolution of functions is very close, and explored in Exercise 12.12.

The crux of the algorithm for computing the representative families $\widehat{\mathcal{P}}^{p,q}_{uv}$ is that the following recurrence holds for the set (of vertex sets) of paths of length exactly p. In particular, for all u, v and $p \geq 3$ it holds that

$$\mathcal{P}^{p}_{uv} = \bigcup_{wv \in E(G)} \mathcal{P}^{p-1}_{uw} * \{\{v\}\}. \tag{12.1}$$

To see that the equation holds, observe that a path from u to w of length $p-1$ can be extended into a path of length p from u to v if and only if wv is an edge and the path from u to w does not already visit v. We now describe how (12.1) can be used to compute the representative families $\widehat{\mathcal{P}}^{p,q}_{uv}$.

12.3.5.1 Combining representative families

An important feature of (12.1) is that it only uses $\cup$ and $*$ operators. In this section we will show that whenever a family $\mathcal{A}$ can be built up from simpler families by only using these operations, a representative family for $\mathcal{A}$ can be efficiently computed by repeatedly combining simpler representative families by the same operations.

It is a quite common setting that there is a large family for which we know a small representative family exists, but are unable to compute it directly because the original family is just too big. In such a setting the tools we develop here often come in handy. We first turn to the union operation.

Lemma 12.26. *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are both p-families, $\mathcal{A}'_1$ q-represents $\mathcal{A}_1$ and $\mathcal{A}'_2$ q-represents $\mathcal{A}_2$, then $\mathcal{A}'_1 \cup \mathcal{A}'_2$ q-represents $\mathcal{A}_1 \cup \mathcal{A}_2$.*

Proof. Consider a set B of size q such that there is a set $A \in \mathcal{A}_1 \cup \mathcal{A}_2$ that fits B. If $A \in \mathcal{A}_1$, then there is an $A' \in \mathcal{A}'_1$ that fits B. If $A \in \mathcal{A}_2$, then there is an $A' \in \mathcal{A}'_2$ that fits B. $\square$

Thus, if we want to compute a q-representative family of $\mathcal{A}_1 \cup \mathcal{A}_2$, it is sufficient to first compute a q-representative family $\mathcal{A}'_1$ of $\mathcal{A}_1$, then another q-representative family $\mathcal{A}'_2$ of $\mathcal{A}_2$, and return $\mathcal{A}'_1 \cup \mathcal{A}'_2$. At this point there is a snag, Theorem 12.15 tells us that there is a q-representative family of $\mathcal{A}_1 \cup \mathcal{A}_2$ of size $\binom{p+q}{p}$. If $\mathcal{A}'_1$ and $\mathcal{A}'_2$ were the result of applying Theorem 12.15 to $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively, their size could also be as much as $\binom{p+q}{p}$. But then $\mathcal{A}'_1 \cup \mathcal{A}'_2$ would have size as much as $2\binom{p+q}{p}$. How can we get rid of this annoying factor 2? Well, we could just compute a q-representative family of $\mathcal{A}^\star$ of $\mathcal{A}'_1 \cup \mathcal{A}'_2$ by invoking Theorem 12.15 again. Note that invoking Theorem 12.15 on $\mathcal{A}'_1 \cup \mathcal{A}'_2$ is much less costly than invoking it on $\mathcal{A}_1 \cup \mathcal{A}_2$, since $\mathcal{A}'_1 \cup \mathcal{A}'_2$ is already a pretty small family of size just a little bit more than $\binom{p+q}{p}$. All that remains is a sanity check that $A^\star$ actually q-represents $\mathcal{A}_1 \cup \mathcal{A}_2$.

Lemma 12.27. *If $\mathcal{A}^\star$ q-represents $\mathcal{A}'$ and $\mathcal{A}'$ q-represents $\mathcal{A}$, then $\mathcal{A}^\star$ q-represents $\mathcal{A}$.*

Proof. Consider a set B of size q such that some $A \in \mathcal{A}$ fits B. Since $\mathcal{A}'$ q-represents $\mathcal{A}$, there is an $A' \in \mathcal{A}'$ that fits B. Now, since $\mathcal{A}^\star$ q-represents $\mathcal{A}'$, there is an $A^\star \in \mathcal{A}^\star$ that fits B. $\square$

We now give an analogue of Lemma 12.26 for subset convolution. Subset convolution has a slightly more complicated behavior with respect to representative sets than union, because the sets in the family $\mathcal{A}_1 * \mathcal{A}_2$ are bigger than the sets in $\mathcal{A}_1$ and $\mathcal{A}_2$.

Lemma 12.28. *Let $\mathcal{A}_1$ be a p_1-family and $\mathcal{A}_2$ be a p_2-family. Suppose $\mathcal{A}'_1$ $(k-p_1)$-represents $\mathcal{A}_1$ and $\mathcal{A}'_2$ $(k-p_2)$-represents $\mathcal{A}_2$. Then $\mathcal{A}'_1 * \mathcal{A}'_2$ $(k-p_1-p_2)$-represents $\mathcal{A}_1 * \mathcal{A}_2$.*

Proof. Suppose there is a set B of size $(k-p_1-p_2)$ that fits a set $(A_1 \cup A_2) \in \mathcal{A}_1 * \mathcal{A}_2$, where $A_1 \in \mathcal{A}_1$, $A_2 \in \mathcal{A}_2$, and $A_1 \cap A_2 = \emptyset$. Then $|B \cup A_2| = k - p_1$ and so there is a set $A_1' \in \mathcal{A}_1'$ that fits $B \cup A_2$, that is, $|A_1' \cup A_2 \cup B| = k$ and $A_1' \cup A_2 \cup B$ is independent. This means that $A_2 \in \mathcal{A}_2$ fits with $A_1' \cup B$. Now, there is a set $A_2' \in \mathcal{A}_2'$ that fits $A_1' \cup B$ as well, and so $|A_1' \cup A_2' \cup B| = k$ and $A_1' \cup A_2' \cup B$ is independent. But then $A_1' \cup A_2'$ fits B, and $(A_1' \cup A_2') \in \mathcal{A}_1' * \mathcal{A}_2'$. □

Consider a p_1-family $\mathcal{A}_1$ and a p_2-family $\mathcal{A}_2$ with $p = p_1 + p_2$. In order to compute a q-representative family of $\mathcal{A}_1 * \mathcal{A}_2$, we can first compute a $(q+p_2)$-representative family $\mathcal{A}_1'$ of $\mathcal{A}$, then a $(q+p_1)$-representative family $\mathcal{A}_2'$ of $\mathcal{A}_2$, and output $\mathcal{A}_1' * \mathcal{A}_2'$. However $\mathcal{A}_1' * \mathcal{A}_2'$ could be quite large compared to $\binom{p+q}{p}$, and so we might want to run Theorem 12.15 on $\mathcal{A}_1' * \mathcal{A}_2'$ to get a representative family $\mathcal{A}^\star$ of size $\binom{p+q}{p}$. Note that computing $\mathcal{A}_1' * \mathcal{A}_2'$ from $\mathcal{A}_1'$ and $\mathcal{A}_2'$ takes roughly $|\mathcal{A}_1'| \cdot |\mathcal{A}_2'|$ time. On the other hand, computing $\mathcal{A}^\star$ from $\mathcal{A}_1' * \mathcal{A}_2'$ takes around $|\mathcal{A}_1'| \cdot |\mathcal{A}_2'| \cdot \binom{p+q}{p}^{\omega-1}$ time, so this step is the more costly one. Nevertheless, if $\mathcal{A}_1'$ and $\mathcal{A}_2'$ both have size $2^{\mathcal{O}(p+q)}$, then $\mathcal{A}_1' * \mathcal{A}_2'$ can be computed from $\mathcal{A}_1'$ and $\mathcal{A}_2'$ in time $2^{\mathcal{O}(p+q)}$ times a polynomial overhead.

12.3.5.2 Long directed cycle, wrapping up

We are now ready to return to the Long Directed Cycle problem. All that was left was to compute, for every u, v, $p \leq k$, and $q \leq k$, a q-representative family $\widehat{\mathcal{P}}_{uv}^{p,q}$ of $\mathcal{P}_{uv}^{p}$. The size of $\widehat{\mathcal{P}}_{uv}^{p,q}$ will be upper bounded by $\binom{p+q}{p}$. For reasons that will become apparent soon, we compute a few extra representative families, in particular, we will compute representative families for all $q \leq 2k - p$ rather than only for $q \leq k$. To that end we will combine (12.1) with the tricks we developed in Section 12.3.5.1.

Lemma 12.29. *There is an algorithm that, given G and k, computes, for every pair u, v of vertices and every integers $2 \leq p \leq k$ and $q \leq 2k - p$, a family $\widehat{\mathcal{P}}_{uv}^{p,q}$ that q-represents $\mathcal{P}_{uv}^{p}$. The algorithm takes time $4^{\omega k} n^{\mathcal{O}(1)} \leq 26.9^k n^{\mathcal{O}(1)}$.*

Proof. For $p = 2$, it is very easy to compute representative families of $\mathcal{P}_{uv}^2$, because the families $\mathcal{P}_{uv}^2$ themselves are so simple. Specifically, we have that $\mathcal{P}_{uv}^2 = \{\{u, v\}\}$ if $uv \in E(G)$ and $\mathcal{P}_{uv}^2 = \emptyset$ otherwise. So we just set

$$\widehat{\mathcal{P}}_{uv}^{2,q} = \mathcal{P}_{uv}^2$$

for every u, v and $q \leq 2k-2$. Clearly, $\widehat{\mathcal{P}}_{uv}^{2,q}$ q-represents $\mathcal{P}_{uv}^2$ and $|\widehat{\mathcal{P}}_{uv}^{2,q}| \leq 1 \leq \binom{2+q}{2}$. Computing these families takes polynomial time.

At this point the algorithm enters a loop in which p iterates over the set $\{3, 4, \ldots k\}$ in increasing order. The loop invariant is that at the start of each iteration, for every u, v, $1 \leq p' < p$ and $q \leq 2k - p'$, a family $\widehat{\mathcal{P}}^{p',q}_{uv}$ of size at most $\binom{p'+q}{p'}$ that represents $\mathcal{P}^{p'}_{uv}$ has already been computed.

In each iteration the algorithm first computes a new family $\widetilde{\mathcal{P}}^{p,q}_{uv}$ for each u, v and $q \leq 2k - p$ as follows

$$\widetilde{\mathcal{P}}^{p,q}_{uv} = \bigcup_{wv \in E(G)} \widehat{\mathcal{P}}^{p-1,q+1}_{uw} * \{\{v\}\}. \tag{12.2}$$

Lemmas 12.26 and 12.28 immediately imply that $\widetilde{\mathcal{P}}^{p,q}_{uv}$ q-represents $\mathcal{P}^{p}_{uv}$. Furthermore,

$$|\widetilde{\mathcal{P}}^{p,q}_{uv}| \leq |\widehat{\mathcal{P}}^{p-1,q+1}_{uv}| \cdot n, \tag{12.3}$$

and therefore

$$|\widetilde{\mathcal{P}}^{p,q}_{uv}| \leq \binom{p+q}{p} n.$$

Furthermore, computing $\widetilde{\mathcal{P}}^{p,q}_{uv}$ using Equation 12.2 takes time $\mathcal{O}(\binom{p+q}{p} n^{\mathcal{O}(1)})$. Now, the algorithm computes a q-representative family

$$\widehat{\mathcal{P}}^{p,q}_{uv} \subseteq^{q}_{\text{rep}} \widetilde{\mathcal{P}}^{p,q}_{uv}$$

by applying Theorem 12.15 on $\widetilde{\mathcal{P}}^{p,q}_{uv}$. This takes time

$$|\widetilde{\mathcal{P}}^{p,q}_{uv}| \binom{p+q}{p}^{\omega-1} n^{\mathcal{O}(1)} \leq \binom{p+q}{p}^{\omega} n^{\mathcal{O}(1)}.$$

Since q-representation is transitive (Lemma 12.27), we have that $\widehat{\mathcal{P}}^{p,q}_{uv}$ also q-represents $\mathcal{P}^{p}_{uv}$. Furthermore

$$|\widehat{\mathcal{P}}^{p,q}_{uv}| \leq \binom{p+q}{p}$$

and so we have managed to compute the families $\widehat{\mathcal{P}}^{p,q}_{uv}$ as desired. Thus, assuming that the loop invariant holds at the beginning of the iteration, it holds at the beginning of the next iteration as well. Hence, by the end of the last iteration, the algorithm has computed the families $\widehat{\mathcal{P}}^{p,q}_{uv}$ with the desired properties for every pair u, v of vertices and all integers $2 \leq p \leq k$ and $q \leq 2k - p$. This concludes the proof. □

With Lemma 12.29 at hand, it is easy to get an algorithm for Long Directed Cycle.

Theorem 12.30. *There is a* $26.9^k n^{\mathcal{O}(1)}$*-time algorithm for* Long Directed Cycle

Proof. The algorithm starts by applying Lemma 12.29 and computing, for every pair u, v of vertices and integers $2 \leq p \leq k$, $q \leq 2k - p$, a family

$$\widehat{\mathcal{P}}^{p,q}_{uv} \subseteq^q_{\mathrm{rep}} \mathcal{P}^p_{uv}$$

of size at most $\binom{p+q}{p}$. This takes $4^{\omega k} n^{\mathcal{O}(1)} \leq 26.9^k n^{\mathcal{O}(1)}$ time. Then the algorithm checks, for each u, v, $q \leq k$, and set $A \in \widehat{\mathcal{P}}^{k,q}_{uv}$, whether there exists a path P_2 from v to u with $A \cap V(P_2) = \{u, v\}$. This takes time

$$\sum_{u,v,q \leq k} \binom{k+q}{k} n^{\mathcal{O}(1)} \leq 4^k n^{\mathcal{O}(1)}.$$

By Lemmas 12.24 and 12.25, such a set A and path P_2 exists if and only if G has a cycle of length at least k. □

12.4 Representative families for uniform matroids

A common theme for the problems considered so far is that whenever we applied Theorem 12.15 in order to compute a representative family, we used a uniform matroid as the underlying matroid. Is it possible to get an improved variant of Theorem 12.15 that only works for uniform matroids? There are two kinds of improvement possible: on the size of the output representative family and on the time it takes to compute it. It is possible to show that the size bound of Theorem 12.15 cannot be improved, even for uniform matroids (see Exercise 12.11). However it is possible to improve the running time.

Theorem 12.31. *Let $\mathcal{M}$ be a uniform matroid and $\mathcal{A}$ be a p-family of independent sets of $\mathcal{M}$. There is an algorithm that, given $\mathcal{A}$, a rational number $0 < x < 1$, and an integer q, computes a q-representative family $\mathcal{A}' \subseteq^q_{\mathrm{rep}} \mathcal{A}$ of size at most $x^{-p}(1-x)^{-q}2^{o(p+q)}$ in time $|\mathcal{A}|(1-x)^{-q}2^{o(p+q)}$.*

The proof of Theorem 12.31 is out of the scope of the book; however we will see a proof of a weaker variant of Theorem 12.31 in Exercise 12.13. To get a feel of Theorem 12.15, it is useful to consider what happens when $x = \frac{1}{2}$. In this case the size of the representative family is upper bounded by $2^{p+q+o(p+q)}$, and the running time is upper bounded by $|\mathcal{A}|2^{q+o(p+q)}$. For $p = q$ the size of the output family is essentially the same as the $\binom{p+q}{p}$ bound of Theorem 12.15, while the running time is much better. However, if $\frac{p}{p+q}$ is sufficiently far from $1/2$, then setting $x = 1/2$ will make Theorem 12.15 output a representative family which is significantly larger than the optimal $\binom{p+q}{p}$ bound.

In order to compare Theorems 12.15 and 12.31 it is useful to set $x = \frac{p}{p+q}$. Since

$$\binom{p+q}{p} \leq \left(\frac{p}{p+q}\right)^{-p} \left(\frac{q}{p+q}\right)^{-q} \leq \binom{p+q}{p}(p+q)$$

we obtain the following corollary by setting $x = \frac{p}{p+q}$ and applying Theorem 12.31.

Corollary 12.32. *Let $\mathcal{M}$ be a uniform matroid and $\mathcal{A}$ be a p-family of independent sets of $\mathcal{M}$. There is an algorithm that given $\mathcal{A}$ and an integer q computes a q-representative family $\mathcal{A}' \subseteq^q_{\text{rep}} \mathcal{A}$ of size at most $\binom{p+q}{p} 2^{o(p+q)}$ in time $|\mathcal{A}|(\frac{q}{p+q})^{-q} 2^{o(p+q)}$.*

For uniform matroids, Corollary 12.32 will compute representative families much faster than Theorem 12.15, at the cost that the computed representative family is a factor $2^{o(p+q)}$ larger than $\binom{p+q}{p}$. When p and q are of the same order of magnitude, the $2^{o(p+q)}$ factor becomes negligible, but when p or q is very small, such as in the kernelization applications of Sections 12.3.2 and 12.3.3, one may not use Corollary 12.32 in place of Theorem 12.15, because the subexponential factor in the size bound of Corollary 12.32 would make the upper bound on the kernel size super-polynomial.

The value $x = \frac{p}{p+q}$ is the choice for x that minimizes the size of the representative family $\mathcal{A}'$ computed by Theorem 12.31. The reason one might sometimes want to invoke Theorem 12.31 with a different value of x is that different values of x yield a faster running time for the computation of the representative family $\mathcal{A}'$, at the cost of making $\mathcal{A}'$ bigger.

12.5 Faster Long Directed Cycle

We now have at our disposal a new hammer, Theorem 12.31. The purpose of Theorem 12.31 is that when we want to compute q-representative sets for uniform matroids, Theorem 12.31 does it faster than Theorem 12.15. In Section 12.3.5.2, the application of Theorem 12.15 was the main bottleneck for the algorithm for Long Directed Cycle. What happens if we just replace all the applications of Theorem 12.15 by calls to Theorem 12.31? More precisely, we will prove a more efficient version of Lemma 12.29.

Lemma 12.33. *There is an algorithm that, given G and k, computes for every pair u, v of vertices and integers $2 \leq p \leq k$ and $q \leq 2k - p$, a family $\widehat{\mathcal{P}}^{p,q}_{uv}$ that q-represents $\mathcal{P}^p_{uv}$. The size of $\widehat{\mathcal{P}}^{p,q}_{uv}$ is at most*

$$\left(\frac{p+2q}{p}\right)^p \left(\frac{p+2q}{2q}\right)^q \cdot 2^{o(p+q)},$$

and the algorithm takes time $6.75^{k+o(k)} n^{\mathcal{O}(1)}$.

Proof. Consider the algorithm in the proof of Lemma 12.29. For $p = 2$ computing the families $\widehat{\mathcal{P}}^{p,q}_{uv}$ is trivial. For each choice of u, v, $3 \leq p \leq k$, and q, the algorithm of Lemma 12.29 first computes a family $\widetilde{\mathcal{P}}^{p,q}_{uv}$ and then applies Theorem 12.15 to compute a q-representative family $\widehat{\mathcal{P}}^{p,q}_{uv}$ of $\widetilde{\mathcal{P}}^{p,q}_{uv}$.

The only property from Theorem 12.15 for which we needed to argue correctness is that $\widehat{\mathcal{P}}^{p,q}_{uv}$ in fact q-represents $\widetilde{\mathcal{P}}^{p,q}_{uv}$. Thus, if we replace the call to Theorem 12.15 by a call to Theorem 12.31, we still get a correct algorithm. All that we need to do is to select the value for x that we will use for each of the calls to Theorem 12.31, and analyze the running time and the size of the output families. The running time of the algorithm is dominated by a polynomial number of calls to Theorem 12.31. Hence, to bound the running time of the algorithm it is sufficient to bound the running time of the call to Theorem 12.31 that takes the longest time.

When invoking Theorem 12.31 in order to compute the q-representative family $\widehat{\mathcal{P}}^{p,q}_{uv}$ of $\widetilde{\mathcal{P}}^{p,q}_{uv}$, we will use the value $x_{p,q}$ for x, where

$$x_{p,q} = \frac{p}{p+2q}.$$

This choice of $x_{p,q}$ is not taken out of thin air, it is in fact the choice that ends up minimizing the running time of the algorithm. The choice of $x_{p,q}$ is discussed in Exercise 12.14.

When we apply Theorem 12.31 with $x = x_{p,q}$ in order to obtain $\widehat{\mathcal{P}}^{p,q}_{uv} \subseteq^q_{\text{rep}} \widetilde{\mathcal{P}}^{p,q}_{uv}$, the size of $\widehat{\mathcal{P}}^{p,q}_{uv}$ is bounded as follows.

$$|\widehat{\mathcal{P}}^{p,q}_{uv}| \le s_{p,q} \cdot 2^{o(p+q)}.$$

Here, $s_{p,q}$ is defined as

$$s_{p,q} = (x_{p,q})^{-p} \cdot (1 - x_{p,q})^{-q}.$$

This proves the claimed size bound on $\widehat{\mathcal{P}}^{p,q}_{uv}$.

From (12.3), we have that $|\widetilde{\mathcal{P}}^{p,q}_{uv}| \le s_{p-1,q+1} \cdot n$. Thus, when we apply Theorem 12.31 to compute $\widehat{\mathcal{P}}^{p,q}_{uv}$ from $\widetilde{\mathcal{P}}^{p,q}_{uv}$, this takes time

$$s_{p-1,q+1} \cdot (1 - x_{p,q})^{-q} \cdot 2^{o(p+q)} \cdot n^{\mathcal{O}(1)}. \tag{12.4}$$

To analyze the running time further we need the following claim.

Claim 12.34. *For any $p \ge 3$ and $q \ge 1$, $s_{p-1,q+1} \le e^2 \cdot p \cdot s_{p,q}$.*

Proof. Inserting the definition of $x_{p,q}$ into the definition of $s_{p,q}$ yields

$$s_{p,q} = p^{-p}(2q)^{-q}(p+2q)^{p+q}.$$

This yields the following inequality

$$\begin{aligned}\frac{s_{p-1,q+1}}{s_{p,q}} &= \frac{(p-1)^{-p+1}(2q+2)^{-q-1}(p+2q+1)^{p+q}}{p^{-p}(2q)^{-q}(p+2q)^{p+q}}\\ &= \frac{p^p}{(p-1)^{p-1}}\cdot\frac{(2q)^q}{(2q+2)^{q+1}}\cdot\frac{(p+2q+1)^{p+q}}{(p+2q)^{p+q}}\\ &\le p\cdot\left(1+\frac{1}{p-1}\right)^{p-1}\cdot 1\cdot\left(1+\frac{1}{p+2q}\right)^{p+q}\\ &\le p\cdot e\cdot e = e^2\cdot p.\end{aligned}$$

In the last inequality, we used that $(1+1/x)^x < e$ for every $x \ge 0$. ⌟

From (12.4) and Claim 12.34, we have that the running time for computing $\widehat{\mathcal{P}}^{p,q}_{uv}$ from $\widetilde{\mathcal{P}}^{p,q}_{uv}$ is bounded by

$$s_{p,q}\cdot(1-x_{p,q})^{-q}\cdot 2^{o(p+q)}\cdot n^{\mathcal{O}(1)} \le \left(\frac{p+2q}{p}\right)^p\left(\frac{p+2q}{2q}\right)^{2q}\cdot 2^{o(p+q)}\cdot n^{\mathcal{O}(1)}.$$

The total running time of the algorithm is bounded by the maximum of the function

$$f(p,q) = \left(\frac{p+2q}{p}\right)^p\left(\frac{p+2q}{2q}\right)^{2q}$$

times a polynomial in n and a subexponential $2^{o(p+q)}$ factor. We are interested in the maximum of f on the domain $3 \le p \le k$ and $1 \le q \le 2k-p$. Simple calculus shows that the maximum is attained for $p = q = k$. Hence the maximum of f on the domain $3 \le p \le k$ and $1 \le q \le 2k-p$ is

$$\left(\frac{3k}{k}\right)^k\cdot\left(\frac{3k}{2k}\right)^{2k} = \left(3\cdot\left(\frac{3}{2}\right)^2\right)^k = 6.75^k.$$

Then the running time of the algorithm is upper bounded by $6.75^{k+o(k)}n^{\mathcal{O}(1)}$, completing the proof. □

We now have a faster algorithm to compute the representative families $\widehat{\mathcal{P}}^{p,q}_{uv}$. This yields an improved bound for the running time of our algorithm for Long Directed Cycle.

We recall the $26.9^k n^{\mathcal{O}(1)}$-time algorithm for Long Directed Cycle given in Theorem 12.30. First it computed, for every pair of vertices u,v and integers $2 \le p \le k$ and $1 \le q \le 2k-p$, a q-representative family $\widehat{P}^{p,q}_{uv}$ of $\widehat{P}^{p}_{uv}$. These families were computed using Lemma 12.29, and therefore have size at most 4^k. Computing these families took $26.9^k n^{\mathcal{O}(1)}$ time. After this, the algorithm loops over every set in these families and runs a polynomial-time check (namely a breadth first search) for each set. This takes $4^k n^{\mathcal{O}(1)}$ time.

The bottleneck in the algorithm of Theorem 12.30 is the call to Lemma 12.29. If, instead of using Lemma 12.29, we use Lemma 12.33, then computing the representative families only takes $6.75^{k+o(k)}n^{\mathcal{O}(1)}$ time. However, the families may now be bigger, but their total size is not more than $6.75^{k+o(k)}n^{\mathcal{O}(1)}$, since it took that much time to compute them. Thus, if we now loop over every set in the representative families and run a breadth first search, just as in the proof of Theorem 12.30, this will take at most $6.75^{k+o(k)}n^{\mathcal{O}(1)}$ time. Hence we arrive at the following theorem.

Theorem 12.35. *There is a* $6.75^{k+o(k)}n^{\mathcal{O}(1)}$*-time algorithm for* Long Directed Cycle.

12.6 Longest Path

We can use the insights of the algorithms for Long Directed Cycle to make a fast deterministic algorithm for Longest Path. Here, we are given a directed graph G and an integer k, and the task is to determine whether G contains a simple path on at least k vertices.

In Section 12.3.5, we defined for every pair of vertices u and v and integer $p \geq 1$ the family

$$\mathcal{P}^p_{uv} = \Big\{X \;:\; X \subseteq V(G), |X| = p, \text{ and there is a directed } uv\text{-path in } G[X] \text{ visiting all vertices of } X\Big\}.$$

The Longest Path problem can be reformulated to asking whether there exists a u and a v such that $\mathcal{P}^k_{uv}$ is nonempty. Our algorithm will check whether $\mathcal{P}^k_{uv}$ is nonempty by computing, for every u, v, $2 \leq p \leq k$, and $0 \leq q \leq k-p$, a q-representative family

$$\widehat{\mathcal{P}}^{p,q}_{uv} \subseteq^q_{\mathrm{rep}} \mathcal{P}^p_{uv}.$$

Here, just as for Long Directed Cycle, the underlying matroid is the uniform matroid of rank $p+q$.

The crucial observation is that $\widehat{\mathcal{P}}^{k,0}_{uv}$ 0-represents $\mathcal{P}^p_{uv}$. Thus, if $\mathcal{P}^p_{uv}$ is nonempty, then $\mathcal{P}^k_{uv}$ contains some set A that fits with the empty set $\emptyset$. But then $\widehat{\mathcal{P}}^{k,0}_{uv}$ must also contain a set which fits with $\emptyset$, and hence $\widehat{\mathcal{P}}^{k,0}_{uv}$ must be nonempty as well. Thus, having computed the representative families $\widehat{\mathcal{P}}^{p,q}_{uv}$, all we need to do is to check whether there is a pair u, v of vertices such that $\widehat{\mathcal{P}}^{k,0}_{uv}$ is nonempty. All that remains is an algorithm that computes the representative families $\widehat{\mathcal{P}}^{p,q}_{uv}$.

Lemma 12.36. *There is an algorithm that given G and k computes for every pair u, v of vertices and integers $2 \leq p \leq k$ and $q \leq k-p$, a family $\widehat{\mathcal{P}}^{p,q}_{uv}$ that q-represents $\mathcal{P}^p_{uv}$. The size of $\widehat{\mathcal{P}}^{p,q}_{uv}$ is at most*

$$\left(\frac{p+2q}{p}\right)^p \left(\frac{p+2q}{2q}\right)^q \cdot 2^{o(p+q)},$$

and the algorithm takes time $\phi^{2k+o(k)}n^{\mathcal{O}(1)} = 2.619^k n^{\mathcal{O}(1)}$. *Here,* ϕ *is the golden ratio* $\frac{1+\sqrt{5}}{2}$.

Proof. The proof of Lemma 12.36 is identical to the proof of Lemma 12.33, with the only exception being that the q-representative family $\widehat{\mathcal{P}}_{uv}^{p,q}$ of $\mathcal{P}_{uv}^{p}$ is computed for all $q \leq k-p$, rather than for all $q \leq 2k-p$. In the running time analysis, this has the effect that the running time of the algorithm is equal to the maximum of the function

$$f(p,q) = \left(\frac{p+2q}{p}\right)^p \left(\frac{p+2q}{2q}\right)^{2q}$$

times a polynomial in n and a subexponential $2^{o(p+q)}$ factor. Here, the domain of f is $2 \leq p \leq k$ and $0 \leq q \leq k-p$ (as opposed to $0 \leq q \leq 2k-p$, in the proof of Lemma 12.33). Simple calculus shows that this function is maximized for $p = (1-\frac{1}{\sqrt{5}})k$ and $q = k-p$. For these values of p and q, $f(p,q) = \phi^{2k}$ where ϕ is the golden ratio $\frac{1+\sqrt{5}}{2}$. Thus the running time of the algorithm is upper bounded by $\phi^{2k+o(k)}n^{\mathcal{O}(1)} = 2.619^k n^{\mathcal{O}(1)}$. □

Armed with Lemma 12.36, all we need to do in order to solve the Longest Path problem is to invoke Lemma 12.36 and then check whether any of the families $\widehat{\mathcal{P}}_{uv}^{k,0}$ is nonempty. This yields the following theorem.

Theorem 12.37. *There is a* $\phi^{2k+o(k)}n^{\mathcal{O}(1)} = 2.619^k n^{\mathcal{O}(1)}$*-time algorithm for* Longest Path. *Here,* ϕ *is the golden ratio* $\frac{1+\sqrt{5}}{2}$.

This algorithm is faster than the $4^{k+o(k)}n^{\mathcal{O}(1)}$-time algorithm of Section *5.4, which is based on divide and color. It is not quite as fast as the algorithms presented in Chapter 10, but these algorithms are randomized, whereas the algorithm presented here is deterministic.

Exercises

12.1 (✎). Let G be a graph and let family $\mathcal{F}$ contain those subsets of $V(G)$ that are independent in G. Show that $\mathcal{F}$ is not necessarily a matroid.

12.2. Let G be a bipartite graph with bipartition U and B. Define a family $\mathcal{F}$ of subsets of U, where $S \subseteq U$ is in F if and only if there is a matching in G such that every vertex in S is an endpoint of a matching edge. Prove that $\mathcal{M} = (U, \mathcal{F})$ is a matroid. In other words, prove that transversal matroids are matroids.

12.3. Prove Theorem 12.3.

12.4 (✎). Give a greedy algorithm that, given a matroid $\mathcal{M} = (U, \mathcal{F})$ and a weight function $\mathbf{w} : U \to \mathbb{Z}$, computes a basis of maximum weight. How can we find an independent set of maximum weight? (Note that the weights can be negative.)

12.5 (✎). Let G be a bipartite graph. Find two matroids $\mathcal{M}_1$ and $\mathcal{M}_2$ such that an edge set $S \subseteq E(G)$ is a matching in G if and only if it is independent both in $\mathcal{M}_1$ and $\mathcal{M}_2$.

12.6 (✎). Prove that the intersection of two matroids is not necessarily a matroid.

12.7. Prove Lemma 12.5.

12.8 (✎). Give a polynomial time reduction from the 2-Matroid Intersection problem to Matroid Parity.

12.9. Give an algorithm that, given a family $\mathcal{A}$ of sets of size p and integer q, computes a subfamily $\mathcal{A}' \subseteq^q_{rep} \mathcal{A}$ in time $\mathcal{O}(|\mathcal{A}|^2 \cdot p^{q+O(1)})$. Here the underlying matroid is the uniform matroid of rank $p+q$. Use an algorithm for d-Hitting Set to determine for a given $A_i \in \mathcal{A}$ whether there is some set B_i of size q, disjoint from A_i, but intersecting all other sets in $\mathcal{A}$.

12.10 (✎). Use Theorem 12.15 to prove the following generalization of Bollobás' lemma. Let $\mathcal{M}$ be a matroid and $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$ and $\mathcal{B} = \{B_1, B_2, \ldots, B_m\}$ be families of independent sets in $\mathcal{M}$ such that (a) all sets in $\mathcal{A}$ have size p, (b) all sets in $\mathcal{B}$ have size q, (c) A_i fits B_j if and only if $i = j$. Then $m \leq \binom{p+q}{p}$.

12.11 (✎). Construct for each p and q a p-family $\mathcal{A}$ of size exactly $\binom{p+q}{p}$ such that no proper subfamily $\mathcal{A}'$ of $\mathcal{A}$ q-represents $\mathcal{A}$.

12.12 (✎). Let $\mathcal{A}$ and $\mathcal{B}$ be two families of subsets of the same universe U. Prove that $(\mathbf{1}_{\mathcal{A}} * \mathbf{1}_{\mathcal{B}})(S) \neq 0$ if and only if $S \in \mathcal{A} * \mathcal{B}$. Here $\mathbf{1}_{\mathcal{A}}$ is the *characteristic function* of the family $\mathcal{A}$. More precisely $\mathbf{1}_{\mathcal{A}}$ is a function that takes as input a subset S of U and returns 1 if $S \in \mathcal{A}$ and 0 otherwise. Recall that the convolution of two functions is defined in Section 10.3.

12.13. Give an algorithm that, given as input a family $\mathcal{A}$ of sets of size p over a universe of size n, computes a q-representative family $\mathcal{A}'$ of $\mathcal{A}$ when the underlying matroid is the uniform matroid of rank $p+q$. The algorithm should have running time at most $|\mathcal{A}| \cdot 2^{p+q+o(p+q)} n^{\mathcal{O}(1)}$ and the size of $\mathcal{A}'$ should be at most $2^{p+q+o(p+q)}$. Your algorithm should use the construction of *universal sets* given in Chapter 5.

12.14. The running time of the Long Directed Cycle algorithm of Section 12.5 is dominated by the maximum of Equation 12.4, when maximized over p and q. It is reasonable to assume that the choice of $x_{p,q}$ is *continuous*, i.e., that $x_{p-1,q+1} \approx x_{p,q}$, and that therefore $s_{p-1,q+1} \approx s_{p,q}$. Then Equation 12.4 reduces to

$$s_{p,q} \cdot (1 - x_{p,q})^{-q} \cdot 2^{o(p+q)} \cdot n^{\mathcal{O}(1)}.$$

For a *fixed* value of p and q, find the value of $0 < x_{p,q} < 1$ that minimizes the expression above.

12.15. Improve Lemma 12.22 so that the algorithm for ℓ-Matroid Intersection only uses $|U|^{\mathcal{O}(1)} + 2^{\mathcal{O}(\ell k)}$ operations over GF(s).

12.16 (☠). Consider the graphic matroid $\mathcal{M}$ of a clique on n vertices. Show that if $p+q = n-1$, A is an edge set of size p and B is an edge set of size q, then A fits B if and only if $A \cup B$ is a spanning tree of the clique. Use this fact, together with Theorem 12.15, to give a $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ time deterministic algorithm for Hamiltonian Cycle on graphs of treewidth at most k. You may assume that a tree-decomposition of G of width at most k is given as input.

Hints

12.1 The problem is with the exchange property. Consider, for example, a path on an odd number of vertices, and two independent sets: one consisting of every odd vertex on the path, and one consisting of every even one.

12.2 Let S_1 and S_2 be two independent sets in $\mathcal{M}$ with $|S_1| < |S_2|$. Let M_i be a matching that witnesses that $S_i \in \mathcal{F}$ for $i = 1, 2$. Since $|M_2| > |M_1|$, the symmetric difference $M_1 \triangle M_2$ contains a path P in G of odd length whose every odd edge belongs to M_2 and every even edge belongs to M_1 (i.e., an augmenting path for the matching M_1). Observe that P contains exactly one vertex, say v, of $S_2 \setminus S_1$ and no other vertex of $U \setminus S_1$. Consequently, $M_1 \triangle E(P)$ is a matching in G witnessing that $S_1 \cup \{v\} \in \mathcal{F}$.

12.3 For every $e \in E(G)$, create a variable x_e. Consider a $|B| \times |U|$ matrix A where the entry at position (b, u), $u \in U$, $b \in B$, equals 0 if $ub \notin E(G)$, and x_{ub} if $ub \in E(G)$. For a set $S \subseteq U$, let A_S be the submatrix of A consisting of the columns corresponding to the vertices of S. The crucial observation is that S is independent in $\mathcal{M}$ if and only if there exists a set $T \subseteq B$ of size $|S|$ such that the determinant of submatrix $A_{S,T}$ (consisting of rows and columns corresponding to vertices of T and S), treated as a multivariate polynomial, is nonzero. This follows from the permutation definition of the determinant: a nonzero monomial in $\det A_{S,T}$ corresponds to a perfect matching in $G[S \cup T]$.

To finish the proof, it suffices to apply the Schwartz-Zippel lemma (Lemma 10.19) to observe that evaluating every variable x_e to a random integer between 1 and $x \cdot n \cdot 2^n$ with good probability does not turn any crucial nonzero polynomial into a zero.

12.5 Let A_1 and A_2 be the bipartition classes of G. For $i = 1, 2$, define $\mathcal{M}_i$ as a direct sum of matroids $\mathcal{M}_v$ for $v \in A_i$, where $\mathcal{M}_v$ is the uniform matroid of rank 1 over the universe consisting of the edges incident to v.

12.7 Let $p \geq r$ be the rank of $\mathcal{M}$ and, by applying Gaussian elimination, assume that the matrix M representing $\mathcal{M}$ has p rows. Construct an $r \times p$ matrix B where at every cell (i, j) a new variable $x_{i,j}$ is created. Consider the $r \times |U|$ matrix BM. Argue that a set S of columns is independent in BM (over the ring of multivariate polynomials) if and only if the corresponding set of elements of U is independent in $\mathcal{M} \cap \mathcal{U}_r$. Use the Schwartz-Zippel lemma (Lemma 10.19) together with determinants to show that, if you randomly pick for every $x_{i,j}$ a value in GF(p), then such an evaluated matrix BM represents $\mathcal{M} \cap \mathcal{U}_r$ with good probability.

12.8 Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be the two input matroids over the same universe U. Make two copies of U, U_1 and U_2, and modify each $\mathcal{M}_i$ to operate over the universe U_i. Consider the Matroid Parity instance consisting of the direct sum of $\mathcal{M}_1$ and $\mathcal{M}_2$, and a set of pairs $\{\{u_1, u_2\} : u \in U\}$, where u_i is the copy of element $u \in U$ in the universe U_i.

12.9 The mentioned algorithm for d-Hitting Set is the standard branching one: as long as there exists a set $A' \in \mathcal{A} \setminus \{A_i\}$ that is disjoint with the currently constructed set B_i, consider every possibility of adding an element of $A' \setminus A_i$ into B_i.

12.11 The family of all subsets of $[p + q]$ of size exactly p does the job.

12.13 Consider a construction $\mathcal{U}$ of $(n, p+q)$-universal sets. Observe that a set A of size p and a set B of size q are disjoint if and only if there is some $S \in \mathcal{U}$ such that $A \subseteq S$ and $B \cap S = \emptyset$. What happens if we keep one set $A \in \mathcal{A}$, $A \subseteq S$, for every set $S \in \mathcal{U}$?

To get rid of the $\log n$ factor, coming from the construction of the $(n, p+q)$-universal set, repeat the above procedure as long as it decreases the size of $\mathcal{A}$ by a constant factor.

12.15 Consider the direct sum of the matroids $M_1, \ldots M_\ell$. Each subset of U of size k naturally corresponds to an independent set of size $k\ell$ in this new matroid. Use Theorem 12.15 on this matroid to get the algorithm.

12.16 The straightforward dynamic-programming algorithm for Hamiltonian Cycle may work as follows. For a bag X_t, for every partition of X_t into $A_0 \cup A_1 \cup A_2$ and every perfect matching M in a clique on the vertex set A_1, we would like to know if there exists a subset $F \subseteq E_t$ such that

1. every connected component of (V_t, F) is a path with both endpoints in A_1, and $V(F) = V_t \setminus A_0$;
2. two vertices $u, v \in A_1$ are the endpoints of the same path in (V_t, F) if and only if $uv \in M$.

The $2^{\mathcal{O}(k \log k)} n^{\mathcal{O}(1)}$ time complexity comes purely from the number of choices of the matching M.

Using representative sets, argue that it suffices, at every moment, for fixed bag X_t and sets A_0, A_1, and A_2, to keep only a representative family of matchings M of size $2^{\mathcal{O}(k)}$ for which the corresponding set F exists. To this end, observe that, if M represents part of the solution Hamiltonian cycle present in G_t, and M' represents the part of the cycle in $G - E_t$, then we care only about whether $M \cup M'$ is a Hamiltonian cycle in a clique on vertex set A_1.

Bibliographic notes

Matroid theory is a deep and well-studied field with a wealth of results. We refer to the textbook of Welsh [434] or the book of Oxley [379] for an introduction to the field. The randomized representation of transversal matroids of Theorem 12.3 can be found in the report of Edmonds and Fulkerson [162]. In this book we have chosen not to cover the deep and interesting theory of matroid minors. Highlights of matroid minor theory with consequences for fixed-parameter tractable algorithms include an “excluded grid theorem” for matroids [227] by Geelen, Gerards, and Whittle, and an analogue of Courcelle's theorem (Theorem 7.11) for matroids [262] by Hliněný.

The first polynomial-time algorithm for Matroid Parity is due to Lovász [334] with running time $\mathcal{O}(n^{17})$. The fastest known deterministic algorithm for Matroid Parity is by Gabow and Stallmann [221], running in $\mathcal{O}(nr^{\omega})$ time where r is the rank of the input matroid and $\omega < 2.37$ is the matrix multiplication exponent (see [425]). The fastest randomized algorithm, building on work by Lovász [333], is due to Cheung, Lau, and Leung [87] and has running time $\mathcal{O}(nr^{\omega-1})$. Even though the algorithm we give in this book for 2-Matroid Intersection is by reduction to Matroid Parity, 2-Matroid Intersection is considered an easier problem. A polynomial-time algorithm for 2-Matroid Intersection can be found in the textbook by Schrijver [412]. The FPT algorithm for ℓ-Matroid Intersection of Exercise 12.15 is due to Marx [351], but using a slower algorithm for the computation of representative sets than the one presented in Theorem 12.15, resulting in a somewhat slower algorithm than what is stated in Exercise 12.15.

The first use of matroids for the Feedback Vertex Set was by Ueno, Kajitani, and Gotoh [423], who gave a polynomial-time algorithm for Feedback Vertex Set on cubic graphs. Ideas from this algorithm were later used by Cao, Chen, and Liu [70], who gave a $3.83^k n^{\mathcal{O}(1)}$-time algorithm for Feedback Vertex Set. The $3.619^k n^{\mathcal{O}(1)}$-time algorithm for Feedback Vertex Set presented in this book is by Kociumaka and Pilipczuk [301]. Kociumaka and Pilipczuk [301] also give the currently fastest known deterministic parameterized algorithm for Feedback Vertex Set, with running time $3.592^k n^{\mathcal{O}(1)}$. The fastest known *randomized* algorithm for Feedback Vertex Set is by Cygan, Nederlof, Pilipczuk, Pilipczuk, van Rooij, and Wojtaszczyk [118], runs in $3^k n^{\mathcal{O}(1)}$ time, and uses Cut & Count (see Section 11.2.1) together with iterative compression.

The notion of representative sets can be thought of as an algorithmic variant of the classical Bollobás' lemma [57] and its generalization to linear matroids by Lovász [331] (see Exercise 12.10). The notion of representative sets was defined by Monien [368], where he gave an algorithm for Longest Path running in $k^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ time. Monien [368] gave an algorithm computing a q-representative family for uniform matroids with running time $\mathcal{O}(|\mathcal{A}| q p^{q+1})$, where $\mathcal{A}$ is the input p-family. The algorithm outputs a q-representative family of size $\mathcal{O}(p^q)$. Marx [346] gave an algorithm computing representative families for uniform matroids with $\mathcal{O}(|\mathcal{A}|^2 p^q)$ running time, outputting a q-representative family of size $\binom{p+q}{p}$. Subsequently Marx [351] gave an algorithm computing representative families for linear matroids. The algorithm outputs a q-representative family of size at most $\binom{p+q}{p}$ in time $\binom{p+q}{p}(p+q)^{\mathcal{O}(q)}|\mathcal{A}|^2$. Fomin, Lokshtanov, and Saurabh [206] proved Theorem 12.15; this is the fastest currently known algorithm for computing representative sets of linear matroids. In the same paper Fomin, Lokshtanov and Saurabh [206] gave a faster algorithm for computing representative sets for uniform matroids, in particular they proved Corollary 12.32. Subsequently, Theorem 12.31 was independently proved by Fomin, Lokshtanov, Panolan, and Saurabh [201] and Shachnai and Zehavi [416]. This is the currently fastest known algorithm for computing representative sets of uniform matroids.

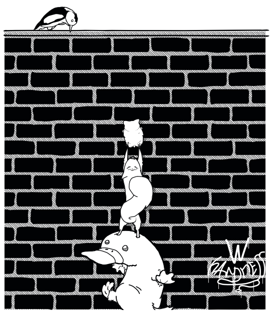
Monien [368] was the first to use representative sets to design parameterized algorithms. Monien [368] used representative sets to give a $k^{\mathcal{O}(k)} n^{O(1)}$-time algorithm for Longest Path. Plehn and Voigt [386] used representative sets and gave a $k^{\mathcal{O}(k)} n^{O(t)}$-time algorithm for Subgraph Isomorphism where the pattern graph H has k vertices and treewidth t. Some more algorithms using representative sets, in particular an algorithm for ℓ-Matroid Intersection, were given by Marx [351]. With respect to kernelization, Kratsch and Wahlström [312] gave a randomized polynomial kernel for Odd Cycle Transversal by encoding the problem as a problem on a certain kind of linear matroid, called gammoids. Kratsch and Wahlström [310] used representative sets of linear matroids to also give a randomized polynomial kernel for Almost 2-SAT, as well as for several other interesting problems. The kernel for d-Hitting Set presented in Section 12.3.2 is due to Kratsch and Wahlström [311]. The kernel for d-Set Packing in Section 12.3.3 is due to Zehavi [442].

The Long Directed Cycle problem was shown fixed-parameter tractable by Gabow and Nie [220], who gave a $k^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$-time algorithm for the problem. Fomin, Lokshtanov, and Saurabh [206] gave the first $c^k n^{\mathcal{O}(1)}$ algorithm for Long Directed Cycle by using representative sets. The algorithm of Fomin, Lokshtanov, and Saurabh [206] is slightly slower than the one stated in Theorem 12.35, because they use Corollary 12.32 to compute representative sets, rather than Theorem 12.31. Theorem 12.35, i.e., a $6.75^k n^{\mathcal{O}(1)}$-time algorithm for Long Directed Cycle, was proved independently by Fomin, Lokshtanov, Panolan, and Saurabh [201] and Shachnai and Zehavi [416].

Part III
Lower bounds

Chapter 13
Fixed-parameter intractability

In this chapter, we use parameterized reductions and W[1]*-hardness to provide evidence that certain parameterized problems are unlikely to be fixed-parameter tractable.*



The goal of the previous chapters was to present the most fundamental algorithmic techniques for showing the fixed-parameter tractability of parameterized problems. In this chapter, we study the complementary questions on lower bounds: how it is possible to show (provide evidence) that a parameterized problem is not fixed-parameter tractable.

In the earliest days of computer science, when the first algorithms were developed to solve combinatorial problems of practical interest such as the Traveling Salesperson Problem, it was observed with frustration that every proposed algorithm had an inconveniently large worst-case upper bound on the running time, typically exponential in the size of the input. Several decades of research have not been able to explain and prove why exponential time is needed for these problems. The theory of NP-completeness at least showed that there is one common underlying reason for the lack of polynomial-time algorithms, and therefore conditional lower bounds based on NP-completeness are the best we can have at this point of time. The fact that computation cannot be cheap seems to be a fundamental principle and understanding the exact reason for this apparent hardness is a deep question that is well beyond our current knowledge.

Our goal in this chapter is to develop a lower bound theory for parameterized problems, similar to the NP-completeness theory of polynomial-time computation. Instead of taking this as an opportunity to delve deeper into the mysteries of efficient computation, we adopt the pragmatic viewpoint of the algorithm designer: our focus is on presenting evidence for as many problems

DOI 10.1007/978-3-319-21275-3_13

as possible that algorithms with certain specifications do not exist. Being aware of and being able to produce such negative results is essential for the algorithm designer: without them, countless hours can be wasted on trying to prove results that contradict commonly accepted assumptions. Knowing that certain problems are (probably) not fixed-parameter tractable prevents us from attacking the same problem over and over again with little hope of progress. Furthermore, the lower-bound theory helps the algorithm designer in a more subtle way as well. In many cases, it is very helpful to look at a problem from both the algorithmic and the complexity viewpoints at the same time. A failed attempt at finding an algorithm can highlight certain difficult situations, giving insight into the structure of hard instances, which can be the starting point of a hardness proof. Conversely, if one can pinpoint a specific reason why a proposed hardness proof cannot be made to work, then it may suggest an algorithmic idea that can renew the attack on the algorithmic side of the question. The authors of this textbook have experienced many times that an answer was found only after repeatedly jumping back and forth between attacks from the algorithmic side and the complexity side of the problem. Moreover, the hardness proofs might tell us what special cases or problem variants merit further exploration. Hence the lower bound theory helps not only in finding the answers, but can even steer the algorithm designer towards the right formulation of the questions.

As we have no proof of P $\neq$ NP, we cannot rule out the possibility that problems such as Clique and Dominating Set are polynomial-time solvable and hence FPT. Therefore, our lower bound theory has to be conditional: we are proving statements of the form "if problem A has a certain type of algorithm, then problem B has a certain type of algorithm as well." If we have accepted as a working hypothesis that B has no such algorithms (or we have already proved that such an algorithm for B would contradict our working hypothesis), then this provides evidence that problem A does not have this kind of algorithm either. To prove such statements in the context of fixed-parameter tractability, we need a notion of reduction that transfers (negative evidence for) fixed-parameter tractability from one problem to the other. As we shall see, the standard notion of polynomial-time reduction used in NP-completeness theory is not sufficient for our purposes. Section 13.1 introduces the notion of parameterized reductions, which have a slightly different flavor than NP-hardness proofs and therefore require a different toolbox of techniques. Then Section 13.2 presents a selection of basic reductions from Clique to various problems. If we accept as a working hypothesis that Clique is not fixed-parameter tractable, then these reductions from Clique are practical evidence that certain other problems are not fixed-parameter tractable either. The assumption that Clique is not fixed-parameter tractable is a stronger assumption than P $\neq$ NP, but it seems that we need such a strong assumption, as we currently do not know how to base fixed-parameter intractability results on assuming P $\neq$ NP only.

A remarkable aspect of NP-completeness is that there are literally thousands of natural hard problems that are equally hard in the sense that they are reducible to each other. The situation is different in the case of parameterized problems: there seem to be different levels of hardness and even basic problems such as Clique and Dominating Set seem to occupy different levels. Downey and Fellows introduced the W-hierarchy in an attempt to classify parameterized problems according to their hardness. We give a brief overview of this hierarchy in Section 13.3. The Clique problem is W[1]-complete, that is, complete for the first level of the W-hierarchy. Therefore, Clique not being fixed-parameter tractable is equivalent to FPT $\neq$ W[1]. This is the basic assumption of parameterized complexity; we interpret W[1]-hardness as evidence that a problem is not fixed-parameter tractable. In Section *13.4, we connect the class W[1] to a fundamental computational problem about Turing machines. This connection provides further evidence supporting the assumption FPT $\neq$ W[1]. Moreover, it allows a very convenient way of proving membership in W[1]. In Section *13.5, we use this and other arguments to conclude that all the problems studied in Section 13.2 are actually complete either for W[1] or for W[2]. In Section 13.6, we finish the chapter with a selection of W[1]-hardness results, demonstrating some of the challenges one faces in typical hardness proofs and ways of overcoming these difficulties. We remark that Section 14.4.1 contains further important parameterized reductions.

Following our pragmatic viewpoint, we omit here the discussion of issues that belong to structural complexity theory and hence are not essential to the algorithm designer. What complexity classes can be defined and which problems are complete for them? Which of these classes are equal? Are there intermediate problems between two classes? What notions of reductions can we define? Can we increase the strength of our negative evidence by basing it on weaker conjectures? What role does nonuniformity play in the complexity of the problems? These are fine theoretical questions, but are not of immediate importance to the algorithm designer who only wants some evidence that the answer to a particular algorithmic question is negative.

Finally, let us comment on the fact that all the lower bounds of this and the following chapters are conditional on various conjectures. The reader might be skeptical of the validity of these lower bounds, as they are based on unproven assumptions. In particular, the strongest of these conjectures, SETH, is still somewhat controversial and not fully accepted by the computational complexity community. Nevertheless, these lower bounds still send an important and useful message. The reader might dismiss these lower bounds and spend time on working towards an algorithmic result violating these lower bounds. But then the reader should be aware that he or she is actually working towards disproving one of the basic conjectures and the difficulty of proving the algorithmic result is not specific to the problem at hand, but it requires answering a basic well-studied question first. Then one could argue that perhaps it is more efficient to spend time on concentrating on this basic question directly, rather than answering it indirectly via some more problem-specific algorithmic question.

13.1 Parameterized reductions

Let us recall the standard notion of polynomial-time reduction used in NP-hardness proofs. A *polynomial-time many-one*[1] *reduction* from problem A to problem B is a polynomial-time algorithm that, given an instance x of problem A, outputs an *equivalent* instance x' of problem B, that is, x is a yes-instance of problem A if and only if x' is a yes-instance of problem B. If there is such a reduction from A to B and B is polynomial-time solvable, then A is also polynomial-time solvable: given an instance x of problem A, we can run the reduction to get an instance x' of problem B, which we can solve using the assumed polynomial-time algorithm for B.

We need an analogous notion of reduction for parameterized problems that transfers fixed-parameter tractability.

Definition 13.1 (Parameterized reduction). Let $A, B \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems. A *parameterized reduction* from A to B is an algorithm that, given an instance (x, k) of A, outputs an instance (x', k') of B such that

1. (x, k) is a yes-instance of A if and only if (x', k') is a yes-instance of B,
2. $k' \leq g(k)$ for some computable function g, and
3. the running time is $f(k) \cdot |x|^{\mathcal{O}(1)}$ for some computable function f.

As a minor technical remark, let us observe that it can be assumed that the functions f and g are nondecreasing: for example, we may replace the computable function $g(k)$ with $\hat{g}(k) = \max_{i=1}^{k} g(k) \geq g(k)$, which is a computable nondecreasing function.

Before comparing this notion of reduction to classic polynomial-time reductions, let us formally state and prove that parameterized reductions work as intended.

Theorem 13.2. *If there is a parameterized reduction from A to B and B is* FPT*, then A is* FPT *as well.*

Proof. Let (x, k) be an instance of problem A. The parameterized reduction gives an equivalent instance (x', k') in time $f(k)|x|^{c_1}$ with $k' \leq g(k)$ and $|x'| \leq f(k)|x|^{c_1}$ (clearly, the running time of the reduction is an upper bound on the size of the instance created). Suppose that B has an algorithm with running time $h(k)n^{c_2}$; by the earlier discussion, we may assume that h is

[1] The terminology "many-one" refers to the fact that many instances of problem A might be mapped to the same instance of problem B. Some authors use the term *Karp-reduction* for such a reduction.

nondecreasing. Then running this algorithm on (x', k') determines whether (x', k') is a yes-instance of B (equivalently, whether (x, k) is a yes-instance of A) in time at most

$$h(k')|x'|^{c_2} \leq h(g(k))(f(k)|x|^{c_1})^{c_2}$$

(the inequality uses the fact that h is nondecreasing). Together with the time required for the parameterized reduction, the total running time is at most $f'(k)|x|^{c_1 c_2}$, where $f'(k) = f(k) + h(g(k))f(k)^{c_2}$ is a computable function. That is, the problem B is fixed-parameter tractable. □

The second item in Definition 13.1 requires something that is not required for classic polynomial-time reductions: the new parameter should be bounded by a function of the parameter of the original instance. Therefore, not every NP-hardness proof gives a parameterized reduction.

There is a very simple reduction from Independent Set to Clique: the size of the maximum independent set in graph G is the same as the size of the maximum clique in the complement $\overline{G}$. Therefore, given an instance (G, k), creating an instance $(\overline{G}, k)$ is a polynomial-time reduction from Independent Set to Clique. In fact, this is a parameterized reduction: the new parameter is the same in the original instance, hence $g(k) = k$ is a trivial bound on the new parameter. The same reduction can be used in the other direction as well, that is, to reduce Clique to Independent Set. Therefore, these two problems are equally hard in the sense that one is FPT if and only if the other is.

It is a well-known fact that an n-vertex graph G has an independent set of size k if and only if it has a vertex cover of size $n - k$. Therefore, (G, k) is a yes-instance of Independent Set if and only if $(G, n-k)$ is a yes-instance of Vertex Cover. This immediately gives a polynomial-time reduction from Independent Set to Vertex Cover. However, this is *not* a parameterized reduction: as n can be arbitrarily large compared to k, there is no function $g(k)$ such that $n - k \leq g(k)$. We do not expect that there is a parameterized reduction from Independent Set to Vertex Cover: as the latter problem is FPT, the existence of such a reduction would imply by Theorem 13.2 that Independent Set is FPT as well, which we do not believe.

The third item in Definition 13.1 allows the running time of the reduction to be more than polynomial: it can have an additional factor depending only on the parameter k. This means that not every parameterized reduction is a polynomial-time reduction and if there is a parameterized reduction from A to B such that the unparameterized version of A is NP-hard, then this *does not* necessarily imply that B is NP-hard. As an artificial example, consider the problem Clique$_{\log}$, where (G, k) is a yes-instance if $k \leq \log_2 |V(G)|$ and G has a clique of size k. The following simple transformation is a parameterized reduction from Clique to Clique$_{\log}$: given an instance (G, k), we output the instance (G', k), where G' is obtained from G by adding 2^k isolated vertices. Then $k \leq \log_2 |V(G')|$ always holds, and hence (G', k) is a yes-instance of Clique$_{\log}$ if and only if G has a clique of size k. However, Clique$_{\log}$ can be solved in time $|V(G)|^{\mathcal{O}(\log_2 |V(G)|)}$ by brute force, as the answer is trivial if $k > \log_2 |V(G)|$. That is, the problem can be solved in quasi-polynomial time $n^{\mathcal{O}(\log n)}$, which we do not expect to be possible for NP-hard problems.

In general, parameterized reductions and polynomial-time reductions are incomparable. However, the vast majority of parameterized reductions (including most reductions appearing in this chapter) are actually polynomial-time reductions.

Later in this section, a hardness proof for Dominating Set on Tournaments will be a more natural example of a parameterized reduction with running time exponentially depending on the parameter k.

13.2 Problems at least as hard as Clique

In this section, we present a series of parameterized reductions showing that Clique can be reduced to various basic problems. This means that these problems are at least as hard as Clique in the sense that if any of them is FPT, then it follows that Clique is FPT as well. This is strong evidence that none of the problems considered in this section is FPT.

To be more precise, the reductions given in this section are not all from Clique. We use the fact, stated in the following theorem, that parameterized reductions compose: that is, a reduction from A to B and reduction from B to C implies that there is a reduction from A to C. Therefore, to show that there is a parameterized reduction from Clique to a problem C, it is sufficient to reduce a problem B to C, where B is a problem to which we already have a reduction from Clique.

Theorem 13.3. *If there are parameterized reductions from A to B and from B to C, then there is a parameterized reduction from A to C.*

Proof. Let $\mathcal{R}_1$ be a parameterized reduction from A to B (with running time $f_1(k) \cdot |x|^{c_1}$ and parameter bound $g_1(k)$) and let $\mathcal{R}_2$ be a parameterized reduction from B to C (with running time $f_2(k) \cdot |x|^{c_2}$ and parameter bound $g_2(k)$). As discussed after Definition 13.1, we may assume that the functions f_1, f_2, g_1, g_2 are nondecreasing. Given an instance (x, k) of A, reduction $\mathcal{R}$ first uses $\mathcal{R}_1$ to create an equivalent instance (x', k') of B (with $|x'| \leq f_1(k) \cdot |x|^{c_1}$ and $k' \leq g_1(k)$), and then uses reduction $\mathcal{R}_2$ on (x', k') to create an equivalent instance of C (with $k'' \leq g_2(k')$). It is clear that (x, k) is a yes-instance of A if and only if (x'', k'') is a yes-instance of C. We have $k'' \leq g_2(k') \leq g_2(g_1(k))$ (using the fact that g_2 is nondecreasing), thus k'' is bounded by a computable function of k. The running time of the reduction is

$$\begin{aligned} f_1(k) \cdot |x|^{c_1} + f_2(k') \cdot |x'|^{c_2} &\leq f_1(k) \cdot |x|^{c_1} + f_2(g_1(k)) \cdot (f_1(k) \cdot |x|^{c_1})^{c_2} \\ &\leq (f_1(k) + f_2(g_1(k))f_1(k)) \cdot |x|^{c_1 c_2} \\ &= f^*(k) \cdot |x|^{c_1 c_2}, \end{aligned}$$

where $f^*(k) = f_1(k) + f_2(g_1(k))f_1(k)$ is a computable function (we use in the first inequality the fact that f_2 is nondecreasing). Thus reduction $\mathcal{R}$ satisfies all requirements of Definition 13.1. □

It is common practice to prove NP-hardness for the most restricted version of a problem, as this can be convenient when the problem becomes the source of a reduction later: it can be easier to devise a reduction if input instances of only some restricted form need to be considered. The same principle holds for parameterized problems and in particular for Clique, which is the source of most parameterized reductions. We show first that Clique remains hard even if the graph is regular, that is, every vertex has the same degree.

Theorem 13.4. *There is a parameterized reduction from* Clique *to* Clique *on regular graphs.*

Proof. Given an instance (G, k) of Clique, we create a graph G' as described below and output the instance (G', k). If $k \leq 2$, then the Clique problem is trivial, hence we can output a trivial yes- or no-instance. Let d be the maximum degree of G.

(i) Take d distinct copies $G_1, \ldots, G_d$ of G and let v_i be the copy of $v \in V(G)$ in graph G_i.
(ii) For every vertex $v \in V(G)$, let us introduce a set V_v of $d - d_G(v)$ vertices and add edges between every vertex of V_v and every v_i for $1 \leq i \leq d$.

Observe that every vertex of G' has degree exactly d. To prove the correctness of the reduction, we claim that G has a k-clique if and only if G' has. The left to right implication is clear: copies of G appear as subgraphs in G', thus any clique in G gives a corresponding clique in G'. For the reverse direction, observe that the vertices introduced in step (ii) do not appear in any triangles. Therefore, assuming $k \geq 3$, these vertices cannot be part of a k-clique. Removing these vertices gives d disjoint copies of G, thus any k-clique appearing there implies the existence of a k-clique in G. □

As the complement of a regular graph is also regular, we get an analogous result for Independent Set.

Proposition 13.5. *There is a parameterized reduction from* Clique *to* Independent Set *on regular graphs.*

Note that we have restricted the graph to be regular, but we did not specify that the graph is, say, 3-regular or has any other fixed degree bound. Indeed, Clique and Independent Set are both FPT on r-regular graphs for every fixed integer r, and moreover, FPT with combined parameters $k + r$ (see Exercise 3.2). This has to be contrasted with the fact that Independent Set is NP-hard on 3-regular graphs.

The following reduction shows an example where we can exploit the fact that the graph appearing in the Clique instance is regular. The input to the Partial Vertex Cover problem is a graph G with two integers k and s, and (G, k, s) is a yes-instance if G has a set S of k vertices that covers at least s edges (we say that a set S covers an edge xy if at least one of x and y is in S). Vertex Cover is the special case $s = |E(G)|$; the following reduction shows that the general Partial Vertex Cover problem is probably harder than Vertex Cover.

Theorem 13.6. *There is a parameterized reduction from* Independent Set *on regular graphs to* Partial Vertex Cover *parameterized by k.*

Proof. Let (G, k) be an instance of Independent Set, where G is an r-regular graph. We claim that G has an independent set of size k if and only if (G, k, rk) is a yes-instance of Partial Vertex Cover. If G has an independent set S of size k, then every edge of G is covered by at most one vertex of S, hence together they cover exactly rk edges. Conversely, suppose that G has a set S of k vertices covering rk edges. As each vertex of G covers exactly r edges, this is only possible if no edge of G is covered twice by these k vertices, that is, if they form an independent set. □

Note that Partial Vertex Cover is fixed-parameter tractable parameterized by s, the number of edges to cover (Exercise 5.11).

Next we consider a version of the Clique problem that is a useful starting point for hardness proofs. The input of Multicolored Clique[2] (also called Partitioned Clique) consists of a graph G, an integer k, and a partition $(V_1, \ldots, V_k)$ of the vertices of G; the task is to decide if there is a k-clique containing exactly one vertex from each set V_i.

Theorem 13.7. *There is a parameterized reduction from* Clique *on regular graphs to* Multicolored Clique *on regular graphs.*

Proof. Let (G, k) be an instance of Clique, where G is an r-regular graph on n vertices. We construct an instance $(G', k, (V_1, \ldots, V_k))$ as follows. For every vertex $v \in V(G)$, we introduce k vertices $v_1, \ldots, v_k$ into G'. The set V_i contains v_i for every $v \in V(G)$. We define the edges of G' as follows:

(i) Every V_i is a set of n independent vertices.

[2] The name "multicolored" comes from an alternate way of defining the problem: we can say that the vertices of G are colored by k colors and we are looking for a clique where every vertex has a distinct color.

(ii) If u and v are different and adjacent in G, then we make u_i and v_j adjacent for every $1 \leq i, j \leq k$, $i \neq j$.

Observe that the degree of every vertex in G' is $(k-1)r$: vertex v_i is adjacent to r vertices from each V_j with $i \neq j$.

We claim that G has a k-clique if and only if G' has a k-clique with exactly one vertex from each V_i. Suppose that $v^1, \dots, v^k$ are the vertices of a k-clique in G (ordered arbitrarily). Then $v_1^1 \in V_1$, ..., $v_k^k \in V_k$ is a k-clique in G'. Conversely, suppose that $v_1^1 \in V_1$, ..., $v_k^k \in V_k$ is a k-clique in G'. It follows from the definition of G' that v_i^i and v_j^j are only adjacent in G' if v^i and v^j are different and adjacent in G. Therefore, $\{v^1, \dots, v^k\}$ induces a k-clique in G. □

One may analogously define the multicolored version of Independent Set, where the k vertices of the independent set have to include one vertex from each class of the given partition $(V_1, \dots, V_k)$. Taking the complement of the graph is a parameterized reduction between Clique and Independent Set. Observe that this also holds for the multicolored version.

Corollary 13.8. *There are parameterized reductions between* Multicolored Clique *on regular graphs and* Multicolored Independent Set *on regular graphs.*

Our next reduction gives evidence for the fixed-parameter intractability of Dominating Set.

Theorem 13.9. *There is a parameterized reduction from* Multicolored Independent Set *to* Dominating Set.

Proof. Let $(G, k, (V_1, \dots, V_k))$ be an instance of Multicolored Independent Set. We construct a graph G' the following way (see Fig. 13.1).

(i) For every vertex $v \in V(G)$, we introduce v into G'.
(ii) For every $1 \leq i \leq k$, we make the set V_i a clique in G'.
(iii) For every $1 \leq i \leq k$, we introduce two new vertices x_i, y_i into G' and make them adjacent to every vertex of V_i.
(iv) For every edge $e \in E(G)$ with endpoints $u \in V_i$ and $v \in V_j$, we introduce a vertex w_e into G' and make it adjacent to every vertex of $(V_i \cup V_j) \setminus \{u, v\}$.

The intuitive idea of the reduction is the following. The vertices x_i and y_i ensure that a dominating set of size k has to select exactly one vertex from each V_i. The vertices w_e ensure that the selected vertices form an independent set in the original graph: if both endpoints of an edge e are selected, then the vertex w_e is not dominated.

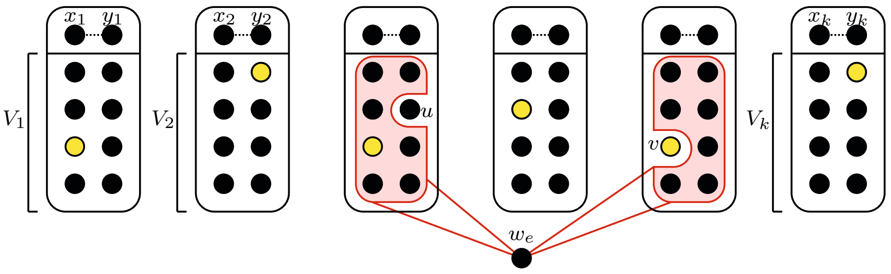


Fig. 13.1: The reduction from Multicolored Independent Set to Dominating Set in Theorem 13.9. The set $V_i \cup \{x_i, y_i\}$ induces a clique minus the edge x_iy_i. If edge e connectes $u \in V_i$ and $v \in V_j$, then we introduce a vertex w_e adjacent to $(V_i \cup V_j) \setminus \{u, v\}$

Formally, we claim that G has an independent set with exactly one vertex from each V_i if and only if G' has a dominating set of size k. Suppose that there is an independent set I in G with one vertex from each V_i; we show that it is a dominating set in G'. First, as $I \cap V_i \neq \emptyset$, the set I dominates the set $V_i \cup \{x_i, y_i\}$. Consider now a vertex w_e, where $u \in V_i$ and $v \in V_j$ are the endpoints of edge $e \in E(G)$. As u and v are adjacent, at least one of them is not in I. In other words, I contains a vertex of either $V_i \setminus \{u\}$ or $V_j \setminus \{v\}$, that is, I contains a vertex of $(V_i \cup V_j) \setminus \{u, v\}$, which dominates w_e.

To prove the reverse direction of the equivalence, suppose now that D is a dominating set of size k in G'. To dominate vertices x_i and y_i, the set D has to contain at least one vertex from $V_i \cup \{x_i, y_i\}$. As these sets are disjoint for different values of i, we can assume that D contains exactly one vertex from each of $V_i \cup \{x_i, y_i\}$. As x_i and y_i are not adjacent, this vertex of D cannot be x_i or y_i. Therefore, let us assume that the dominating set D consists of the vertices $v_1 \in V_1, \ldots, v_k \in V_k$. We claim that D is an independent set in G. Suppose that v_i and v_j are the endpoints of an edge e. Then vertex w_e of G' is adjacent only to $(V_i \cup V_j) \setminus \{v_i, v_j\}$, and hence D does not dominate w_e, a contradiction. □

As there is a very simple reduction between (multicolored versions of) Clique and Independent Set, a reduction from one can be reinterpreted to be a reduction from the other with minimal changes. However, sometimes choosing one of the two problems is more convenient notationally or conceptually; for example, in the proof of Theorem 13.9, it is convenient notationally that (iv) adds a new vertex w_e for every edge e (rather than for every nonedge).

Given a set system $\mathcal{F}$ over a universe U and an integer k, the Set Cover problem asks if there are k sets in $\mathcal{F}$ such that their union is U. In this chapter, we always parameterize Set Cover by the number k of selected sets in the solution. The following easy reduction shows that Set Cover is as hard as Dominating Set (Exercise 13.2 asks for a reduction in the reverse direction).

Theorem 13.10. *There is a parameterized reduction from* Dominating Set *to* Set Cover.

Proof. Let (G, k) be an instance of Dominating Set. We create an instance $(\mathcal{F}, U, k)$ of Set Cover as follows. We let $U := V(G)$ and for every $v \in V(G)$, we introduce the set $N_G[v]$ (the closed neighborhood of v) into $\mathcal{F}$. Suppose that D is a dominating of size k in G. Then the union of the corresponding k sets of $\mathcal{F}$ covers U: an uncovered element would correspond to a vertex of G not dominated by D. Conversely, if the union of k sets in $\mathcal{F}$ is U, then the corresponding k vertices of G dominate every vertex: a vertex not dominated in G would correspond to an element of U not covered by the k sets. □

On directed graphs, we define dominating set in the following way: a set $D \subseteq V(G)$ is a dominating set of G if for every vertex $v \in V(G) \setminus D$, there is a vertex $u \in D$ such that edge (u, v) is in G. Equivalently, we can say that we want the union of the closed outneighborhoods to cover every vertex. It is easy to reduce Dominating Set on undirected graphs to Dominating Set on directed graphs: we can replace each undirected edge xy with two directed edges (x, y) and (y, x) (in other words, we replace xy with a bidirected edge between x and y). Furthermore, the reduction from Dominating Set to Set Cover appearing in Theorem 13.10 can be adapted to directed graphs. Therefore, Dominating Set on undirected graphs, Dominating Set on directed graphs, and Set Cover are reducible to each other.

A very interesting special case of Dominating Set on directed graphs is when the input graph is a *tournament,* that is, for every pair u, v of distinct vertices, exactly one of (u, v) and (v, u) is in the graph. It is easy to see that every tournament on n vertices has a dominating set of size $\mathcal{O}(\log n)$.

Lemma 13.11. *Every tournament on n vertices has a dominating set of size at most* $1 + \log n$.

Proof. We prove the claim by induction on n. For $n = 1$ it is trivial, so let us assume $n \geq 2$.

Since T has $\binom{n}{2}$ edges, the sum of outdegrees of all the vertices is equal to $\binom{n}{2} = \frac{n(n-1)}{2}$. As there are n vertices in T, there exists some vertex u with outdegree at least $\frac{n-1}{2}$. Let U^+ be the set of outneighbors of u, and let U^- be the set of inneighbors of u. Then U^+ and U^- form a partition of $V(T) \setminus \{u\}$, and $|U^-| \leq \frac{n-1}{2} \leq \lfloor \frac{n}{2} \rfloor$. Since $\lfloor \frac{n}{2} \rfloor < n$, by induction hypothesis we may find a set $D' \subseteq U^-$ that is a dominating set in $T[U^-]$ and has at most $1 + \log\lfloor \frac{n}{2} \rfloor \leq \log n$ vertices. Then $D := D' \cup \{u\}$ is a dominating set in T, and has size at most $1 + \log n$. □

Therefore, Dominating Set on Tournaments can be solved by brute force in time $n^{\mathcal{O}(\log n)}$, making it very unlikely that it is NP-hard. Is there such a spectacular collapse of complexity compared to general directed graphs also for the parameterized version of the problem? Surprisingly, we are able to show that this is not true: there is a parameterized reduction from Set Cover to Dominating Set on Tournaments.

The crucial building block in our reduction will be a construction of sufficiently small tournaments that do not admit a dominating set of size k. Such tournaments are traditionally called *k-paradoxical*, and were studied intensively in combinatorics. Note that by Lemma 13.11, a k-paradoxical tournament must have at least 2^k vertices. It turns out that once we increase the size of the vertex set slightly over this threshold, the k-paradoxical tournaments start to appear.

Theorem 13.12. *For every $n \geq r_k = 2 \cdot 2^k \cdot k^2$ there exists a tournament on n vertices that does not admit a dominating set of size k.*

The proof of Theorem 13.12, due to Erdős, is a classic application of the probabilistic method: a randomly sampled tournament on at least r_k vertices does not admit a dominating set of size k with positive probability. The reader is asked to verify this fact in Exercise 13.4. We remark that the multiplicative constant 2 in the function r_k can be in fact substituted with $\ln 2 + o_k(1)$.

In our reduction we need to *construct* a k-paradoxical tournament, and hence it seems necessary to have a constructive, deterministic version of Theorem 13.12. Apparently, there exist deterministic constructions that achieve similar asymptotic size of the vertex set as Theorem 13.12; in Exercise 13.5, the reader is asked to work out details of a simple deterministic construction with slightly worse bounds. We can, however, circumvent the need of a deterministic construction at the cost of allowing the construction to run in doubly exponential time in terms of k. The running time will be still fixed-parameter tractable, which is sufficient for our purposes.

Lemma 13.13. *There exists an algorithm that, given an integer k, works in time $2^{\binom{r_k}{2}} \cdot r_k^{k+1} \cdot k^{\mathcal{O}(1)}$ and outputs a tournament T_k that has r_k vertices and does not admit a dominating set of size k.*

Proof. We construct T_k by iterating through all the $2^{\binom{r_k}{2}}$ tournaments on r_k vertices, and for each of them verifying in time $r_k^{k+1} \cdot k^{\mathcal{O}(1)}$ whether it is k-paradoxical by checking all the k-tuples of vertices. Theorem 13.12 guarantees that one of the verified tournaments will be in fact k-paradoxical. □

We are finally ready to provide the reduction.

Theorem 13.14. *There is a parameterized reduction from* Set Cover *to* Dominating Set on Tournaments.

Proof. The reduction starts with an instance $(\mathcal{F}, U, k)$ of Set Cover, and outputs an equivalent instance $(T, k+1)$ of Dominating Set on Tournaments. The first step is a construction of a $(k+1)$-paradoxical tournament

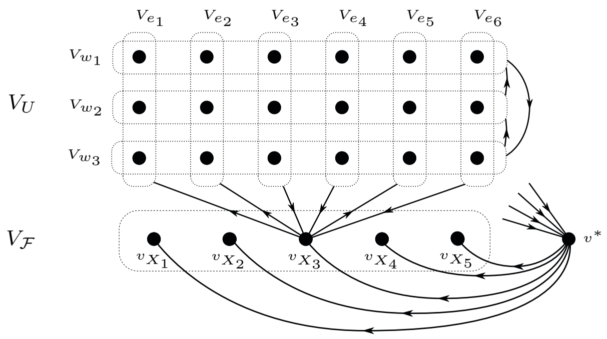


Fig. 13.2: A schematic view of the construction of Theorem 13.14 for a universe of size 6, family of size 5, and S being a cyclic tournament on three vertices. Edges adjacent to the sets V_{e_i} and V_{w_j} represent appropriately oriented complete bipartite graphs, and edges created in (i) between $V_{\mathcal{F}}$ and V_U have been depicted only for the set $X_3 = \{e_1, e_2, e_5\}$

$S = T_{k+1}$ on r_{k+1} vertices using Lemma 13.13; this contributes a factor $2^{\binom{r(k+1)}{2}} \cdot r(k+1)^{k+2} \cdot (k+1)^{\mathcal{O}(1)}$ to the running time, which is just a function of k.

The vertex set of the constructed tournament T is defined as follows:

(i) For every $e \in U$, create a set of r_{k+1} vertices $V_e = \{v_{e,w} \ : \ w \in V(S)\}$, one for each vertex of S. Let $V_w = \{v_{e,w} \ : \ e \in U\}$, and let $V_U = \bigcup_{e \in U} V_e = \bigcup_{w \in V(S)} V_w$.
(ii) For every $X \in \mathcal{F}$, create one vertex v_X. Let $V_{\mathcal{F}} = \{v_X \ : \ X \in \mathcal{F}\}$.
(iii) Moreover, create one vertex v^*.

We now create the edge set of T.

(i) For every set $X \in \mathcal{F}$ and every element $e \in U$, if $e \in X$ then introduce an edge from v_X to every vertex of V_e, and if $e \notin X$ then introduce an edge from every vertex of V_e to v_X.
(ii) For every set $X \in \mathcal{F}$, introduce an edge (v^*, v_X).
(iii) For every element $e \in X$ and $w \in V(S)$, introduce an edge $(v_{e,w}, v^*)$.
(iv) For every $w_1, w_2 \in V(S)$ with $w_1 \neq w_2$, introduce an edge from every vertex of V_{w_1} to every vertex of V_{w_2} if $(w_1, w_2) \in E(S)$, and introduce the reverse edges if $(w_2, w_1) \in E(S)$.
(v) For every $w \in V(S)$, put edges between vertices of V_w arbitrarily.
(vi) Finally, put the edges between vertices of $V_{\mathcal{F}}$ arbitrarily.

It is easy to see that the constructed digraph T is indeed a tournament, and that the construction can be performed in time polynomial in $|U|$, $|\mathcal{F}|$, and r_{k+1} (provided S is already constructed). We now claim that T admits a dominating set of size $k + 1$ if and only if the input instance $(\mathcal{F}, U, k)$ is a yes-instance.

Assume first that $\mathcal{G} \subseteq \mathcal{F}$ is a subfamily of size at most k such that $\bigcup \mathcal{G} = U$. Consider $D = \{v^*\} \cup \{v_X : X \in \mathcal{G}\}$. Clearly $|D| \leq k+1$, and observe that D is a dominating set of T: each vertex of $V_{\mathcal{F}}$ is dominated by v^*, while each vertex $v_{e,w} \in V_U$ is dominated by a vertex $v_X \in D$ for $X \in \mathcal{G}$ such that $e \in X$.

Conversely, suppose that T admits a dominating set D such that $|D| \leq k+1$. Since D has to dominate v^*, either D contains v^* or at least one vertex of V_U. Consequently, $|D \cap V_{\mathcal{F}}| \leq k$. Let $\mathcal{G} = \{X \in \mathcal{F} : v_X \in D\}$. Clearly $|\mathcal{G}| \leq k$, so it suffices to prove that $\bigcup \mathcal{G} = U$.

For the sake of contradiction assume that there exists some $e_0 \in U$ that does not belong to any set of $\mathcal{G}$. Let $Z \subseteq V(S)$ be the set of all vertices $z \in V(S)$ such that $D \cap V_z \neq \emptyset$. Since $|D| \leq k+1$, we also have $|Z| \leq k+1$. Since S is $(k+1)$-paradoxical, we have that there exists some vertex $w_0 \in V(S)$ that is not dominated by Z in S. Consider now the vertex v_{e_0,w_0} in T. By the definition of the edges between $V_{\mathcal{F}}$ and V_U, introduced in (i), we have that v_{e_0,w_0} is not dominated by $D \cap V_{\mathcal{F}}$, since then e_0 would belong to $\bigcup \mathcal{G}$ by the definition of $\mathcal{G}$. Since Z does not dominate w_0 in S, by the definition of the edges introduced in (iv) it also follows that $D \cap V_U$ does not dominate v_{e_0,w_0}. Note that in particular $w_0 \notin Z$, so $D \cap V_{w_0} = \emptyset$ and v_{e_0,w_0} cannot be dominated from within V_{w_0}. Finally, v_{e_0,w_0} cannot be dominated by $D \cap \{v^*\}$, since v^* does not have any outneighbor in V_U. We infer that v_{e_0,w_0} is not dominated by D at all, which contradicts the assumption that D is a dominating set in T. □

Note that, unlike most parameterized reductions in this book and also in the literature, the reduction in Theorem 13.14 is *not* a polynomial-time reduction: the size of the constructed instance has exponential dependence on k and the running time depends double exponentially on k. Therefore, this reduction crucially uses the fact that property 3 in Definition 13.1 allows $f(k) \cdot |x|^{\mathcal{O}(1)}$ time instead of just $|x|^{\mathcal{O}(1)}$.

CONNECTED DOMINATING SET is the variant of DOMINATING SET where we additionally require that the dominating set induce a connected graph. Not surprisingly, this version of the problem is also hard.

Theorem 13.15. *There is a parameterized reduction from* DOMINATING SET *to* CONNECTED DOMINATING SET.

Proof. Let (G,k) be an instance of DOMINATING SET. We construct a graph G' the following way.

(i) For every vertex $v \in V(G)$, we introduce two adjacent vertices v_1, v_2.
(ii) We make the set $\{v_1 : v \in V(G)\}$ a clique K of size $|V(G)|$.
(iii) We make v_1 and u_2 adjacent if v and u are adjacent in G.

We claim that (G,k) is a yes-instance of DOMINATING SET if and only if (G',k) is a yes-instance of CONNECTED DOMINATING SET. Suppose first that $S = \{v^1, \ldots, v^k\}$ is a dominating set of size k in G. Then we claim that

$S' = \{v_1^1, \dots, v_1^k\}$ is a connected dominating set of size k in G'. Clearly, $G'[S']$ is a clique and hence it is connected. To see that S' is a dominating set in G', observe that v_1^1 dominates K, and if u is dominated by v^i in G, then u_2 is dominated by v_1^i in G'.

For the proof of the reverse direction of the equivalence, let S' be a connected dominating set of size k in G'. Let v be in S if at least one of v_1 and v_2 is in S'; clearly, $|S| \leq |S'| = k$. We claim that S is a dominating set of G. Consider any vertex $u \in V(G)$. Vertex u_2 of G' is dominated by some vertex v_1 or v_2 that belongs to S'. Then v is in S and, by the construction of G', it dominates u in G, as required. $\square$

13.3 The W-hierarchy

As we have discussed above, most of the natural NP-hard problems are equivalent to each other with respect to polynomial-time reductions (this holds for problems that are NP-complete), but this does not seem to be true for hard parameterized problems. For example, we have shown that (MULTICOLORED) INDEPENDENT SET can be reduced to DOMINATING SET (Theorem 13.9), but it is not known if there is a parameterized reduction in the other direction. This suggests that, unlike in the case of NP-complete problems, there is a hierarchy of hard parameterized problems, with, for example, INDEPENDENT SET and DOMINATING SET occupying different levels of this hierarchy. Downey and Fellows introduced the W-hierarchy in an attempt to capture the exact complexity of various hard parameterized problems. In this section, we briefly overview the fundamental definitions and results related to this hierarchy. It has to be noted, however, that if our only concern is to provide some evidence that a specific problem is not fixed-parameter tractable (which is the usual viewpoint of an algorithm designer), then the exact structure of this hierarchy is irrelevant: the reductions from CLIQUE, as presented in Section 13.2 and appearing later in this chapter, already provide practical evidence that a problem is unlikely to be FPT. Nevertheless, the reader should be aware of the basic terminology of the W-hierarchy, as the parameterized algorithms and complexity literature usually assumes its knowledge.

A *Boolean circuit* is a directed acyclic graph where the nodes are labeled in the following way:

- every node of indegree 0 is an *input node,*
- every node of indegree 1 is a *negation node,*
- every node of indegree ≥ 2 is either an *and-node* or an *or-node.*

Additionally, exactly one of the nodes with outdegree 0 is labeled as the *output node* (in addition to being, for example, an and-node). The *depth*

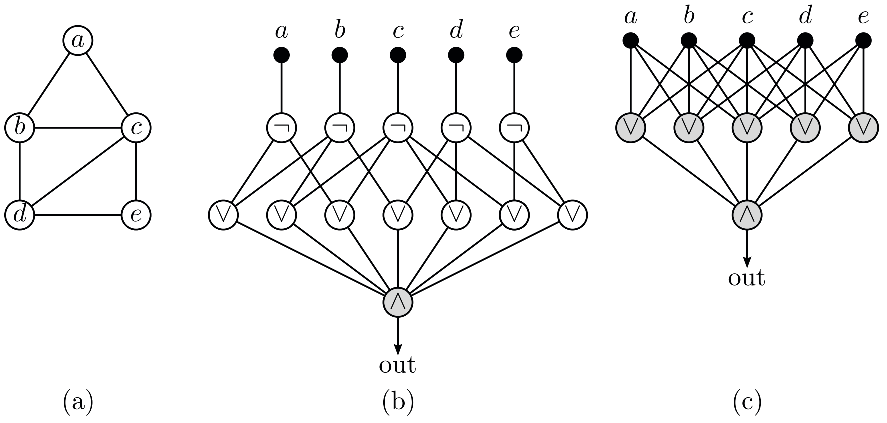


Fig. 13.3: (a) A graph G with five vertices and seven edges. (b) A depth-3, weft-1 circuit satisfied by the independent sets of G. Each or-node corresponds to an edge of G and has indegree 2. (c) A depth-2, weft-2 circuit satisfied by the dominating sets of G. Each or-node corresponds to a vertex of G and its inneighbors correspond to the closed neighborhood to the vertex

of the circuit is the maximum length of a path from an input node to the output node.

Assigning 0-1 values to the input nodes determines the value of every node in the obvious way. In particular, if the value of the output gate is 1 in an assignment to the input nodes, then we say that the assignment *satisfies* the circuit. It can be easily checked in polynomial time if a given assignment satisfies the circuit.

Deciding if a circuit has a satisfying assignment is clearly an NP-complete problem: for example, 3-SAT is its special case. We can define a parameterized version of finding a satisfying assignment in the following way. The *weight* of an assignment is the number of input gates receiving value 1. In the WEIGHTED CIRCUIT SATISFIABILITY (WCS) problem, we are given a circuit C and an integer k, the task is to decide if C has a satisfying assignment of weight *exactly* k. By trying all the $\mathcal{O}(n^k)$ assignments of weight exactly k and then checking whether it satisfies C, we can solve the problem in polynomial time for every fixed k. However, the problem does not seem to be fixed-parameter tractable: it is quite easy to reduce hard parameterized problems such as CLIQUE or DOMINATING SET to WEIGHTED CIRCUIT SATISFIABILITY (see Fig. 13.3).

The levels of the W-hierarchy are defined by restricting WEIGHTED CIRCUIT SATISFIABILITY to various classes of circuits. Formally, if $\mathcal{C}$ is a class of circuits, then we define WCS[$\mathcal{C}$] to be the restriction of the problem where the input circuit C belongs to $\mathcal{C}$. It seems to be a natural idea to restrict the depth of the circuits in the class $\mathcal{C}$; there is a large body of literature on the expressive power of bounded-depth circuits. However, the restriction we need here is a bit more subtle than that. First, we distinguish between *small nodes*, which have indegree at most 2, and *large nodes*, which have indegree > 2 (as we shall see later, the choice of the constant 2 is not essential here). The *weft*[3] of a circuit is the maximum number of large nodes on a path from an input node to the output node. We denote by $\mathcal{C}_{t,d}$ the class of circuits with weft at most t and depth at most d.

Definition 13.16 (W-hierarchy). For $t \geq 1$, a parameterized problem P belongs to the class W[t] if there is a parameterized reduction from P to WCS[$\mathcal{C}_{t,d}$] for some $d \geq 1$.

Exercise 13.6 shows that the constant 2 in the definition of small/large nodes is not essential: any other constant ≥ 2 would result in the same definition for W[t].

As indicated in Fig. 13.3, it is possible to reduce Independent Set to Weighted Circuit Satisfiability for circuits of depth 3 and weft 1. By Definition 13.16, membership in W[1] follows.

Proposition 13.17. Independent Set *is in* W[1].

One can also show the converse statement: every problem in W[1] can be reduced to Independent Set, that is, the problem is W[1]-hard. The proof of this statement is nontrivial and beyond the scope of this brief introduction. Together with Proposition 13.17, it follows that Independent Set is W[1]-complete.

Theorem 13.18 ([151]). Independent Set *is* W[1]*-complete.*

The circuit in Fig. 13.3 expressing Independent Set has a very specific form: it consists of one layer of negation nodes, one layer of small or-nodes with indegree 2, and a final layer consisting of a single large and-node. Theorem 13.18 shows that this form is in some sense canonical for weft-1 circuits: the satisfiability problem for (bounded-depth) weft-1 circuits can be reduced to weft-1 circuits of this form.

What would be the corresponding canonical forms for weft-t circuits for $t \geq 2$? We need some definitions first. We say that a Boolean formula is *t-normalized* if it is of the following form: it is the conjunction of disjunctions of conjunctions of disjunctions, ..., alternating for t levels. For example, the formula

$$(x_1 \vee \bar{x}_3 \vee \bar{x}_5) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_4 \vee x_5) \wedge (x_1 \vee \bar{x}_2)$$

is 2-normalized, as it is the conjunction of disjunctions of literals. More formally, we first define both Δ_0 and Γ_0 to be the set of formulas consisting of

[3] Weft is a term related to weaving cloth: it is the thread that runs from side to side in the fabric.

only a single literal. Then for $t \geq 1$, we define Δ_t to contain formulas that are the disjunction of an arbitrary number of Γ_{t-1} formulas and Γ_t to contain formulas that are the conjunction of an arbitrary number of Δ_{t-1} formulas. With these definitions, Γ_t is exactly the set of t-normalized formulas. We say that a t-normalized formula is *monotone* (resp., *antimonotone*) if every literal is positive (resp., negated).

The WEIGHTED t-NORMALIZED SATISFIABILITY problem is the weighted satisfiability problem corresponding to circuits representing t-normalized formulas; the monotone/antimonotone versions of the problem are defined in the obvious way. The following theorem, whose proof again goes beyond the scope of this introduction, characterizes the classes W[t] for $t \geq 2$ in terms of these problems.

Theorem 13.19 ([149, 150, 151]).

1. *For every even $t \geq 2$, the following problems are* W[t]*-complete:*
   - WEIGHTED t-NORMALIZED SATISFIABILITY
   - WEIGHTED MONOTONE t-NORMALIZED SATISFIABILITY
   - WEIGHTED MONOTONE $(t + 1)$-NORMALIZED SATISFIABILITY
2. *For every odd $t \geq 3$, the following problems are* W[t]*-complete:*
   - WEIGHTED t-NORMALIZED SATISFIABILITY
   - WEIGHTED ANTIMONOTONE t-NORMALIZED SATISFIABILITY
   - WEIGHTED ANTIMONOTONE $(t + 1)$-NORMALIZED SATISFIABILITY

Theorem 13.19 has three rather surprising aspects. First, it shows that t-normalized formulas are sufficient to express the full power of (bounded-depth) weft-t circuits. These formulas have very specific structures: besides the negation nodes at the inputs, they consist of t layers of large nodes, alternating between and-nodes and or-nodes. One can interpret Theorem 13.19 as saying that additional layers of small nodes do not increase the expressive power, as t-normalized formulas already give rise to W[t]-complete problems. This justifies why we defined W[t] based on weft instead of depth. Second, Theorem 13.19 states that already monotone t-normalized formulas (for even t) and antimonotone t-normalized formulas (for odd t) are sufficient to capture the full power of W[t]. Finally, Theorem 13.19 also states the surprising fact that the power of monotone/antimonotone t-normalized and $(t + 1)$-normalized formulas coincide for certain values of t.

Let us focus now on W[2], the second level of the hierarchy. Figure 13.3 shows that it is possible to reduce DOMINATING SET to WEIGHTED CIRCUIT SATISFIABILITY for circuits of depth 2 and weft 2; hence membership in W[2] follows. We have seen in Section 13.2 that DOMINATING SET and SET COVER (and hence HITTING SET) are equally hard with respect to parameterized reductions (Theorem 13.10 and Exercise 13.2).

Proposition 13.20. DOMINATING SET*,* SET COVER*, and* HITTING SET *are in* W[2]*.*

We observe now that these problems are complete for W[2]. All we have to note is that HITTING SET is equivalent to WEIGHTED MONOTONE 2-NORMALIZED SATISFIABILITY. A 2-monotone normalized formula is of the form

$$(x_1 \vee x_3 \vee x_6) \wedge (x_2 \vee x_3) \wedge (x_1 \vee x_5 \vee x_6).$$

To satisfy such a formula, we have to set at least one variable x_i to 1 in each clause. This is exactly the HITTING SET problem, where the universe is the set of variables and the sets are the clauses. Thus Theorem 13.19 shows that HITTING SET is W[2]-complete.

Theorem 13.21. DOMINATING SET*,* SET COVER*, and* HITTING SET *are* W[2]*-complete.*

The usual expectation is that the W-hierarchy is a proper hierarchy: $W[t] \neq W[t+1]$ for every $t \geq 1$. In Theorem 13.9, we have seen a reduction from INDEPENDENT SET to DOMINATING SET. In light of Theorem 13.21, we do not expect a reduction in the reverse direction: this would imply that INDEPENDENT SET is both W[1]- and W[2]-complete and hence W[1] = W[2] would follow.

*13.4 Turing machines

CLIQUE and INDEPENDENT SET are among the most studied optimization problems in computer science; hence the fact that no algorithm with running time $n^{o(k)}$ was found despite massive efforts is good practical evidence suggesting that these problems are not fixed-parameter tractable. The reductions in Section 13.2 (and those appearing later in Section 13.6) transfer this apparent intractability to several other problems. In this section, we give a more theoretical supporting evidence for the intractability of these problems.

Turing machines formalize the intuitive notion of computation in a precise mathematical way. Computational problems where the task is to predict what a Turing machine will do are typically very hard: as Turing machines can express arbitrarily complex computations, it seems that one needs to simulate the computation of the Turing machine to answer these questions. In the case of the HALTING problem, there is even a formal proof of hardness: a classic diagonalization argument shows that there is no algorithm deciding whether a Turing machine halts in a finite number of steps. The P $\neq$ NP problem is a time-bounded version of this question: one has to decide in deterministic polynomial time whether a nondeterministic polynomial-time Turing machine has an accepting path. While there is no proof that this is not possible, it seems that a nondeterministic Turing machine is such an opaque and generic object that it is unlikely that one can avoid simulating a superpolynomial number of computation paths.

We define a parameterized problem analogously to the acceptance problem of nondeterministic polynomial-time Turing machines, and use it to give further evidence that Clique is not fixed-parameter tractable. In the Short Turing Machine Acceptance problem, the input contains the description of a nondeterministic Turing machine M, a string x, and an integer k; the task is to decide if M with input x has a computation path reaching an accepting state in at most k steps. If the reader sees this problem for the first time, it might appear that simulating a Turing machine for a small number k of steps is an easy task, but let us point out that a nondeterministic Turing machine may branch into an unbounded number of states (bounded only by the length n of the input), hence naive simulation of k steps would lead to an $n^{\mathcal{O}(k)}$ time algorithm. Therefore, the same intuitive argument suggesting P $\neq$ NP also supports the conjecture that Short Turing Machine Acceptance is not fixed-parameter tractable: it is unlikely that a problem that requires understanding a Turing machine with $n^{\mathcal{O}(k)}$ computation paths is FPT.

We present a parameterized reduction from Short Turing Machine Acceptance to Independent Set in this section. This can be considered as evidence for the fact that Independent Set (as well as all the other problems considered in Section 13.2) are not fixed-parameter tractable. The reduction is fairly tedious, but the principle is simple. An accepting computation path of k steps can be fully described by $\mathcal{O}(k^2)$ pieces of information: a sequence of k states, a sequence of k transitions, the content of the first k cells of the tape of a Turing machine in each of the first k steps. We need to construct a graph where the choice of vertices for an independent set of size $\mathcal{O}(k^2)$ represents all this information, and then make sure that this information is consistent, that is, correctly describes a sequence of configurations forming an accepting computation path.

We assume that the reader is familiar with the concept of Turing machines (see standard textbooks such as [269, 417, 380] for more background). Here we review only the basic notation required for the discussion of Turing machines. Since we are using Turing machines as evidence for hardness, and not as a basis for building complexity theory, we avoid treating them in greater detail. We remind the reader that certain details and conventions in the definition (such as whether the tape is infinite in both directions, whether the machine can leave an accepting state, etc.) usually do not matter for the computational power of Turing machines and proofs can be changed accordingly.

A *single-tape nondeterministic Turing machine* is described by a tuple $M = (Q, \Sigma, \Delta, s_0, F)$, where

- Q is a finite set of states,
- Σ is the alphabet,
- $s_0 \in Q$ is the initial state,

- $F \subseteq Q$ is the set of accepting states, and
- $\Delta \subseteq Q \times (\Sigma \cup \{\$, \Box\}) \times Q \times (\Sigma \cup \{\$\}) \times \{\leftarrow, -, \rightarrow\}$ is the transition relation.

The meaning of the transition relation is the following: if, for example, $(q, a, q', b, \leftarrow) \in \Delta$, then this means that whenever the Turing machine is in state q and symbol a appears under the head, then one possible legal step is to move into state q', write b on the tape, and move the head one step to the left. The special symbol $\Box$ represents empty cells of the tape and special symbol \$ appears at the left end (position 0) of the tape. Initially, the tape contains an input word x followed by an infinite sequence of $\Box$ symbols; the head is at position 1. We assume that whenever the Turing machine reads the symbol \$, it does not change it (that is, writes \$ on it) and does not move further to the left.

Theorem 13.22. *There is a parameterized reduction from* Short Turing Machine Acceptance *to* Independent Set.

Proof. Let (M, x, k) be an instance of Short Turing Machine Acceptance. We construct an equivalent instance (G, k') of Independent Set in the following way. By minimal modifications of the Turing machine M, we may assume that whenever it reaches an accepting state, it stays there. Therefore, we may assume that the computation path is infinite and the question is whether M happens to be in an accepting state after step k. Note that during the first k steps, the Turing machine can visit only positions 0, 1, ..., $k+1$ of the tape; hence it is sufficient to "simulate" what happens on these positions. The vertices of G are

- $a_{i,j,\sigma}$ for every $0 \le i \le k$, $0 \le j \le k+1$, $\sigma \in \Sigma \cup \{\$, \Box\}$ and
- $b_{i,j,\delta}$ for every $0 \le i \le k$, $0 \le j \le k+1$, $\delta \in \Delta$.

The intuitive meaning of $a_{i,j,\sigma}$ being in the independent set is that after step i, the symbol at position j of the tape is σ and the head is *not* on this position. The intuitive meaning of $b_{i,j,\delta}$ being in the independent set is that after step i, the head is at position j, and the next transition is δ; in particular, if, say, $\delta = (q, \sigma, q', \sigma', \leftarrow) \in \Delta$, then this means that after step i, the symbol at position j of the tape is σ and the internal state is q. Let $A_{i,j} = \{a_{i,j,\sigma} \ : \ \sigma \in \Sigma \cup \{\$, \Box\}\}$, let $B_{i,j,q,\sigma}$ contain every $b_{i,j,\delta}$ where $\delta \in \Delta$ is a transition such that the first coordinate of δ is q and the second coordinate of δ is σ. Let $B_{i,j,q,\_} = \bigcup_{\sigma \in \Sigma \cup \{\$, \Box\}} B_{i,j,q,\sigma}$, let $B_{i,j,\_,\sigma} = \bigcup_{q \in Q} B_{i,j,q,\sigma}$, and let $B_{i,j} = \bigcup_{q \in Q} \bigcup_{\sigma \in \Sigma \cup \{\$, \Box\}} B_{i,j,q,\sigma,}$. Let $k' = (k+1)(k+2)$.

We remove some of the vertices of the graph to enforce certain boundary conditions:

(i) Let us remove $A_{0,1} \cup (B_{0,1} \setminus B_{0,1,s_0,\sigma})$, where σ is the first symbol under the head in the initial position. This ensures that a vertex $b_{0,1,\delta} \in$

$B_{0,1,s_0,\sigma}$ of the independent set correctly describes the initial state of the Turing machine.

(ii) For every $2 \le j \le k+1$, let us remove $(A_{0,j} \setminus \{a_{0,j,\sigma}\}) \cup B_{0,j}$, where σ is the j-th symbol of the tape in the initial configuration. This ensures that the initial state of the input tape is correctly represented by the independent set.

(iii) For every $0 \le i \le k$, let us remove $(A_{i,0} \setminus \{a_{i,0,\$}\}) \cup (B_{i,0} \setminus B_{i,0,\_,\$})$. This ensures that the symbol at position 0 of the tape is always \$.

(iv) For every $0 \le j \le k+1$, let us remove $B_{k,j,q,\_}$ for every $q \notin F$. This ensures that the vertex $b_{k,j,\delta}$ representing the state of the Turing machine after step k represents an accepting state.

To enforce the interpretation of the vertices described above, we add edges in the following way.

(i) For every $0 \le i \le k$, $0 \le j \le k+1$, we make $A_{i,j} \cup B_{i,j}$ a clique. This ensures that the only way to have an independent set of size $k' = (k+1)(k+2)$ is to select exactly one vertex from each $A_{i,j} \cup B_{i,j}$.

(ii) For every $0 \le i \le k$, we make $\bigcup_{j=0}^{k+1} B_{i,j}$ a clique. This ensures that, for every i, at most one vertex of the independent set can be from some $B_{i,j}$; the rest has to be from the $A_{i,j}$'s, corresponding to the fact that the head of the Turing machine is at one position after step i.

(iii) For every $0 \le i < k$, $0 \le j \le k+1$, and $\sigma \in \Sigma \cup \{\$, \Box\}$, we make $a_{i,j,\sigma}$ adjacent to every vertex of $(A_{i+1,j} \setminus \{a_{i+1,j,\sigma}\}) \cup (B_{i+1,j} \setminus B_{i+1,j,\_,\sigma})$. This ensures that if the head of the Turing machine is *not* at position j after step i, then the same symbol σ appears at position j also after step $i+1$. That is, the vertex selected from $A_{i+1,j} \cup B_{i+1,j}$ should be either $a_{i+1,j,\sigma}$ or a vertex from $B_{i+1,j,\_,\sigma}$.

(iv) For every $0 \le i < k$, $0 \le j \le k+1$, and $\delta \in \Delta$, we add edges to $b_{i,j,\delta}$ in the following way:

- if $\delta = (q, \sigma, q', \sigma', \leftarrow)$, then $b_{i,j,\delta}$ is adjacent to every vertex of $A_{i+1,j-1} \cup (B_{i+1,j-1} \setminus B_{i+1,j-1,q',\_}) \cup (A_{i+1,j} \setminus \{a_{i+1,j,\sigma'}\})$ (note that such a left move is not possible for $j = 0$, since then $b_{i,j,\delta} \in B_{i,j,\_,\$}$ and no left move is defined on symbol \$. This ensures that if $b_{i,j,\delta}$ is in the independent set, then the independent set represents that after step $i+1$, the head is at position $j-1$, the state is q', and the symbol at position j is σ'.
- if $\delta = (q, \sigma, q', \sigma', -)$, then $b_{i,j,\delta}$ is adjacent to every vertex of $A_{i+1,j} \cup (B_{i+1,j} \setminus B_{i+1,j,q',\sigma'})$,
- if $\delta = (q, \sigma, q', \sigma', \rightarrow)$, then $b_{i,j,\delta}$ is adjacent to every vertex of $(A_{i+1,j} \setminus \{a_{i+1,j,\sigma'}\}) \cup A_{i+1,j+1} \cup (B_{i+1,j+1} \setminus B_{i+1,j+1,q',\_})$.

As every $b_{i,j,\delta}$ is fully connected to one of $A_{i+1,j-1}$, $A_{i+1,j}$, and $A_{i+1,j+1}$, these edges ensure that if some $b_{i,j,\delta}$ is in the independent set, then a vertex of $B_{i+1,j-1} \cup B_{i+1,j} \cup B_{i+1,j+1}$ is also in the independent set, that is, a position of the head and a state is defined after step $i+1$

as well. Moreover, this new state and position are the result of a valid transition $\delta \in \Delta$.

We claim that M can be in an accepting state on input x after k steps if and only if G has an independent set of size k'. For the first direction, if M has an accepting path, then we can select k' vertices according to the interpretation of the vertices $a_{i,j}$, $b_{i,j,\delta}$ described above. Then it can be verified that this set remains independent after adding the edges introduced in (i)-(iv).

For the other direction, suppose that G has an independent set I of size k'. This is only possible if it contains exactly one vertex from $A_{i,j} \cup B_{i,j}$ for every $0 \leq i \leq k$, $0 \leq j \leq k+1$. We claim that, for every $0 \leq i \leq k$, the set I_i of $k+2$ vertices selected from $A_{i,0} \cup B_{i,0}, \dots, A_{i,k+1} \cup B_{i,k+1}$ describe a configuration C_i of the Turing machine, and these $k+1$ configurations together describe an accepting computation path. The way we removed vertices ensures the statement for $i = 0$, and in fact the configuration C_0 corresponding to I_0 is the initial state of the Turing machine and the tape. Let us prove the statement by induction on i: suppose that the configurations $C_0, \dots, C_i$ describe the first i steps of a computation path. Then the way we have added edges ensure that I_{i+1} describes a configuration C_{i+1} that can follow C_i. In particular, as $B_{i+1,0} \cup \dots \cup B_{i+1,k+1}$ is a clique, at most one vertex of I_{i+1} can describe the position of the head. One vertex of I_i describes the position of the head after step i, that is, I_i contains a vertex of $B_{i,j}$ for some $0 \leq j \leq k+1$. Then the edges introduced in (iv) ensure that I_{i+1} also contains at least one vertex describing the position of the head: every vertex of $B_{i,j}$ is fully connected to either $A_{i+1,j-1}$, $A_{i+1,j}$, or $A_{i+1,j+1}$, which forces the selection of a vertex from $B_{i+1,j-1}$, $B_{i+1,j}$, or $B_{i+1,j+1}$, respectively. Therefore, exactly one vertex of I_{i+1} describes the position of the head. This means that we can define the configuration C_{i+1} based on I_{i+1}, and it can be verified (by analysing the edges added in (iii)-(iv)) that it is indeed a configuration that can follow C_i. □

It is not difficult to give a reduction in the reverse direction: from Independent Set to Short Turing Machine Acceptance (Exercise 13.8), showing the W[1]-completeness of the latter problem.

Theorem 13.23. Short Turing Machine Acceptance *is* W[1]*-complete.*

*13.5 Problems complete for W[1] and W[2]

In Section 13.2, we have seen a set of problems that are at least as hard as Clique or Dominating Set, and therefore W[1]-hard or W[2]-hard, respectively. Now we complete the picture by showing that those problems are actually W[1]-complete or W[2]-complete.

Clique and Independent Set are both W[1]-complete (Theorem 13.18). We have reduced Independent Set to Partial Vertex Cover in Theorem 13.6, showing that PARTIAL VERTEX COVER is W[1]-hard. To show membership in W[1], we can reduce PARTIAL VERTEX COVER to SHORT TURING MACHINE ACCEPTANCE.

Theorem 13.24. *There is a parameterized reduction from* PARTIAL VERTEX COVER *to* SHORT TURING MACHINE ACCEPTANCE.

Proof (sketch). The idea is the following. We encode the input instance to PARTIAL VERTEX COVER using the states and transitions of the output Turing machine. Moreover, we take a large alphabet that includes a distinct symbol for each vertex of the input graph. In the first k steps of the Turing machine, a set S of k vertices is guessed and written on the tape. The remaining steps are deterministic. The Turing machine checks whether the k vertices are distinct and counts the total degree of the k vertices. The total degree can be more than the number of edges covered by S, as edges with both endpoints in S are counted twice. Therefore, the Turing machine counts the number of edges induced by S: it reads every pair of vertices from the tape, keeps it in the internal state and increases the counter if they are adjacent. In particular, the input graph is encoded by the transition function: for every state representing a pair of vertices and the value of the counter, the transition function defines a new state where the counter is increased by 1 (if the two vertices are adjacent) or has the same value (if the two vertices are nonadjacent). Finally, the Turing machine accepts if the total degree minus the number of induced edges is at least s. It is not difficult to implement this procedure with a Turing machine that takes $f(k)$ steps; the details are left to the reader (Exercise 13.10). □

An instance $(G, k, (V_1, \ldots, V_k))$ of MULTICOLORED CLIQUE can be easily reduced to CLIQUE: all we need to do is remove those edges whose both endpoints are in the same V_i. Then any k-clique has exactly one vertex from each V_i. Therefore, MULTICOLORED CLIQUE and MULTICOLORED INDEPENDENT SET are both W[1]-complete.

Theorem 13.25. *The following parameterized problems are* W[1]*-complete:*

- CLIQUE
- MULTICOLORED CLIQUE
- INDEPENDENT SET
- MULTICOLORED INDEPENDENT SET
- PARTIAL VERTEX COVER
- SHORT TURING MACHINE ACCEPTANCE

We have seen in Theorem 13.21 that DOMINATING SET, SET COVER, and HITTING SET are W[2]-complete. Theorem 13.15 presented a reduction from DOMINATING SET to CONNECTED DOMINATING SET, hence CONNECTED DOMINATING SET is W[2]-hard. Let us try to show a reduction in the reverse direction to prove that CONNECTED DOMINATING SET is W[2]-complete. For this purpose, we introduce an auxiliary problem first. The DOMINATING SET with Pattern is a version of Dominating Set where we prescribe what subgraph the dominating set should induce. Formally, the input is a graph G, an integer k, and a graph H on the vertex set $[k]$. The task is to find a k-tuple $(v_1, \ldots, v_k)$ of distinct vertices such that (1) they form a dominating set and (2) vertices v_i and v_j are adjacent if and only if ij is an edge of H. It is easy to observe that Dominating Set with Pattern is W[2]-hard: the reduction of Theorem 13.15 from Dominating Set to Connected Dominating Set has the property that if the created instance has a connected dominating set of size k, then it has a dominating set of size k inducing a clique (Exercise 13.12). We can show that Dominating Set with Pattern is in W[2] by reducing it to Dominating Set.

Theorem 13.26. *There is a parameterized reduction from* Dominating Set with Pattern *to* Dominating Set.

Proof. Let (G, k, H) be an instance of Dominating Set with Pattern. We construct a graph G' as follows.

(i) For every vertex $v \in V(G)$, we introduce k vertices $v_1, \ldots, v_k$.
(ii) For every $1 \leq i \leq k$, the set $V_i = \{v_i \;:\; v \in V(G)\}$ forms a clique.
(iii) For every $1 \leq i \leq k$, we introduce two vertices x_i, y_i, and make them adjacent to every vertex of V_i.
(iv) For every $v \in V(G)$, we introduce a vertex v^* and make it adjacent to every vertex of $\{u_i \;:\; u \in N_G[v], 1 \leq i \leq k\}$.
(v) For every $1 \leq i < j \leq k$ such that ij is an edge of H and for every pair (a, b) of nonadjacent vertices in G (including the possibility $a = b$), we introduce a vertex $w_{i,j,a,b}$ and make it adjacent to every vertex of $(V_i \cup V_j) \setminus \{a_i, b_j\}$.
(vi) For every $1 \leq i < j \leq k$ such that ij is *not* an edge of H and for every pair (a, b) such that either a and b are adjacent in G or $a = b$, we introduce a vertex $w_{i,j,a,b}$ and make it adjacent to every vertex of $(V_i \cup V_j) \setminus \{a_i, b_j\}$.

Let $(v^1, \ldots, v^k)$ be a k-tuple of distinct vertices of G forming a dominating set such that v_i and v_j are adjacent if and only if ij is an edge of H. We claim that $S' = \{v^1_1, \ldots, v^k_k\}$ is a dominating set of G'. Vertex v^i_i dominates every vertex in $V_i \cup \{x_i, y_i\}$. Consider a vertex v^* introduced in (iv). There is a vertex v^i dominating v in G, hence v^i_i dominates v^* in G'. Consider now a vertex $w_{i,j,a,b}$ introduced in (v). As ij is an edge of H, vertices v^i and v^j are adjacent. As either $a = b$ or a and b are not adjacent, assertions $v^i = a$ and $v^j = b$ cannot be both true, and hence assertions $v^i_i = a_i$ and $v^j_j = b_j$ cannot be both true. This implies that $(V_i \cup V_j) \setminus \{a_i, b_j\}$ contains at least one of v^i_i and v^j_j, and we can conclude that $w_{i,j,a,b}$ is dominated by S'. We argue similarly for a vertex $w_{i,j,a,b}$ introduced in (vi): now, as ij is not an edge of H, vertices v^i and v^j are not adjacent. As a and b are adjacent or $a = b$, we have again that $(V_i \cup V_j) \setminus \{a_i, b_j\}$ contains at least one of v^i_i and v^j_j, dominating $w_{i,j,a,b}$.

For the reverse direction, let S' be a dominating set of G'. It has to contain at least one vertex from the set $V_i \cup \{x_i, y_i\}$ to dominate the vertices x_i and y_i. As these sets are disjoint for distinct values of i, we can conclude that S' contains exactly one vertex from $V_i \cup \{x_i, y_i\}$; in fact, exactly one vertex from V_i, as x_i and y_i do not dominate each other. Let $S' = \{v_1^1, \ldots, v_k^k\}$, where $v_i^i \in V_i$. We claim first that $S = \{v^1, \ldots, v^k\}$ is a dominating set of G: if a vertex $v \in V(G)$ is not dominated by S, then vertex v^* is not dominated by S' in G'. Next we claim that if ij is an edge of H, then v^i and v^j are different and adjacent in G. Suppose that ij is an edge of H for some $1 \le i < j \le k$. If v^i and v^j are equal or not adjacent, then we have introduced a vertex w_{i,j,v^i,v^j} that is adjacent only to $(V_i \cup V_j) \setminus \{v_i^i, v_j^j\}$, hence it is not dominated by S' in G', a contradiction. Similarly, we claim that if ij is not an edge of H, then v^i and v^j are two distinct nonadjacent vertices. Indeed, if v^i and v^j are adjacent or equal, then we have introduced a vertex w_{i,j,v^i,v^j} that is adjacent only to $(V_i \cup V_j) \setminus \{v_i^i, v_j^j\}$, hence not dominated by S'. In particular, the two claims imply that $(v^1, \ldots, v^k)$ are distinct vertices, as any two of them are either connected by an edge or nonadjacent distinct vertices. Therefore, we get that $(v^1, \ldots, v^k)$ is a k-tuple of distinct vertices forming a dominating set in G and inducing the required pattern. □

In the light of Theorem 13.26, we can show that Connected Dominating Set is in W[2] by reducing it to Dominating Set with Pattern. A solution of Connected Dominating Set induces a connected graph on k vertices and there are $f(k) = 2^{\mathcal{O}(k^2)}$ connected graphs on k labelled vertices. Therefore, it is tempting to say that Connected Dominating Set can be reduced to Dominating Set with Pattern by trying every possible such pattern and applying an algorithm for Dominating Set with Pattern. However, this is not a many-one parameterized reduction, as defined in Definition 13.1: given an instance I of Connected Dominating Set, instead of creating a single equivalent instance I' of Dominating Set with Pattern, we create a set I'_1, …, I'_t of instances of Dominating Set with Pattern such that I is a yes-instance if and only if at least one I'_i is a yes-instance. This is an example of a so-called *parameterized Turing reduction.* Without formalizing the details, a parameterized Turing reduction from A to B is an algorithm that solves an instance of A by having access to an "oracle" that can solve instances of B in constant time. The reduction has the property that, given an instance (x, k) of A, the running time is $f(k)|x|^{\mathcal{O}(1)}$ and the parameter of every created instance of B can be bounded by $f(k)$.

Turing reductions transfer fixed-parameter tractability the same way as many-one reductions: one could state an analogue of Theorem 13.2 saying that parameterized Turing reduction from A to B implies that if B is FPT, then A is FPT as well. Therefore, from the point of view of giving negative evidence for fixed-parameter tractability, Turing reductions are just as good as many-one reductions. However, we defined the W-hierarchy by closure under many-one reductions, thus to complete the picture, we need a many-one reduction to prove the membership of Connected Dominating Set in W[2]. It is possible to modify the reduction of Theorem 13.26 to output a single instance of Dominating Set for a collection of patterns (Exercise 13.13). Alternatively, we can use the definition of W[2] directly and reduce Connected Dominating Set to the circuit satisfiability problem.

Theorem 13.27. Connected Dominating Set *is in* W[2].

Proof. Given an instance (G, k) of Connected Dominating Set, we construct an equivalent instance (C, k) of Weighted Circuit Satisfiability. Let $v_1, \ldots, v_n$ be the vertices of G. The input nodes of the constructed circuit are $x_{i,j}$ for $1 \leq i \leq k$, $1 \leq j \leq n$. The interpretation of $x_{i,j} = 1$ is that the i-th vertex of the solution is v_j.

(i) For $1 \leq i \leq k$, node z_i is the disjunction of $\{x_{i,j} \ : \ 1 \leq j \leq n\}$ and O_1 is the conjunction of all the z_i nodes. Then $O_1 = 1$ expresses the requirement that there are integers $s_1, \ldots, s_k$ such that $x_{1,s_1}, \ldots, x_{k,s_k}$ are exactly the input nodes with value 1, describing a tuple $(v_{s_1}, \ldots, v_{s_k})$ of k vertices.
(ii) For $1 \leq j \leq i$, node w_j expresses that vertex v_j is dominated in the solution: it is the disjunction of the input nodes $\{x_{i,j'} \ : \ 1 \leq i \leq k, v_{j'} \in N[v_j]\}$.
(iii) Node O_2 expresses that the input represents a dominating set: it is the conjunction of w_j for every $1 \leq j \leq n$.
(iv) For every P such that $\emptyset \subsetneq P \subsetneq [k]$, node c_P expresses that there is an edge between $\{v_{s_i} \ : \ i \in P\}$ and $\{v_{s_i} \ : \ i \notin P\}$. For every $i_1 \in P$, $i_2 \in [k] \setminus P$, and pair (v_{j_1}, v_{j_2}) of adjacent or equal vertices in G, we introduce a node e_{P,i_1,i_2,j_1,j_2} that is the conjunction of x_{i_1,j_1} and x_{i_2,j_2}; the node c_P is the disjunction of all these nodes.
(v) Node O_3 expresses that $(v_{s_1}, \ldots, v_{s_k})$ induces a connected graph: it is the conjunction of all the nodes c_P.
(vi) The output node is the conjunction of O_1, O_2, and O_3 (more precisely, there is a small node O' that is the conjunction of O_1 and O_2, and the output node is a small node that is the conjunction of O' and O_3).

Observe that the constructed circuit C has constant depth (independent of k) and weft 2: a path from an input to the output contains either

- a large node z_i and the large node O_1,
- or a large node w_j and the large node O_2,
- or a large node c_P and the large node O_3.

It is easy to see that if $(v_{s_1}, \ldots, v_{s_k})$ is a connected dominating set, then setting $x_{1,s_1}, \ldots, x_{k,s_k}$ to 1 satisfies the circuit. In particular, as $(v_{s_1}, \ldots, v_{s_k})$ induces a connected graph, for every $\emptyset \subsetneq P \subsetneq [k]$ there has to be an $i_1 \in P$ and $i_2 \in [k] \setminus P$ such that $v_{s_{i_1}}$ and $v_{s_{i_2}}$ are adjacent. This means that the node $e_{i_1,i_2,s_{i_1},s_{i_2}}$ has value 1, implying that c_P is 1 as well.

The reverse direction is also easy to show: it is clear that any satisfying assignment describes a k-tuple $(v_{s_1}, \ldots, v_{s_k})$ of vertices (possibly with repeated vertices) that forms a dominating set of the graph. If these vertices did not induce a connected graph, then they can be partitioned into two disconnected parts, that is, there is a $\emptyset \subsetneq P \subsetneq [k]$ such that there is no edge between v_{i_1} and v_{i_2} for any $i_1 \in P$ and $i_2 \in [k] \setminus P$. Then c_P has value 0, implying that O_3 has value 0, and hence the output has value 0 as well. □

We have shown in Section 13.2 that Dominating Set on undirected graphs, general directed graphs, tournaments, and Set Cover (or Hitting Set) are equally hard (see also Exercises 13.2). We summarize these results in the following theorem.

Theorem 13.28. *The following problems are* W[2]*-complete:*

- Dominating Set
- Dominating Set *for directed graphs*
- Dominating Set on Tournaments
- Connected Dominating Set
- Set Cover
- Hitting Set
- Weighted 2-normalized Satisfiability
- Weighted Monotone 2-normalized Satisfiability
- Weighted Monotone 3-normalized Satisfiability

13.6 Parameterized reductions: further examples

In Section 13.2 we have presented reductions showing that basic problems such as Dominating Set are unlikely to be fixed-parameter tractable. The goal of this section is to explain, via further examples, the different types of parameterized reductions one can encounter in the parameterized complexity literature.

13.6.1 Reductions keeping the structure of the graph

The simplest type of reduction is when there is relatively simple correspondence between the input graph and the output graph. This is the case, for example, in the trivial reduction from Clique to Independent Set, or when reducing Dominating Set to Connected Dominating Set (Theorem 13.15). In other examples, the reduction may involve more complicated operations than just taking the complement of the graph (e.g., subdividing edges, replacing vertices/edges with certain fixed subgraphs, etc.), but the

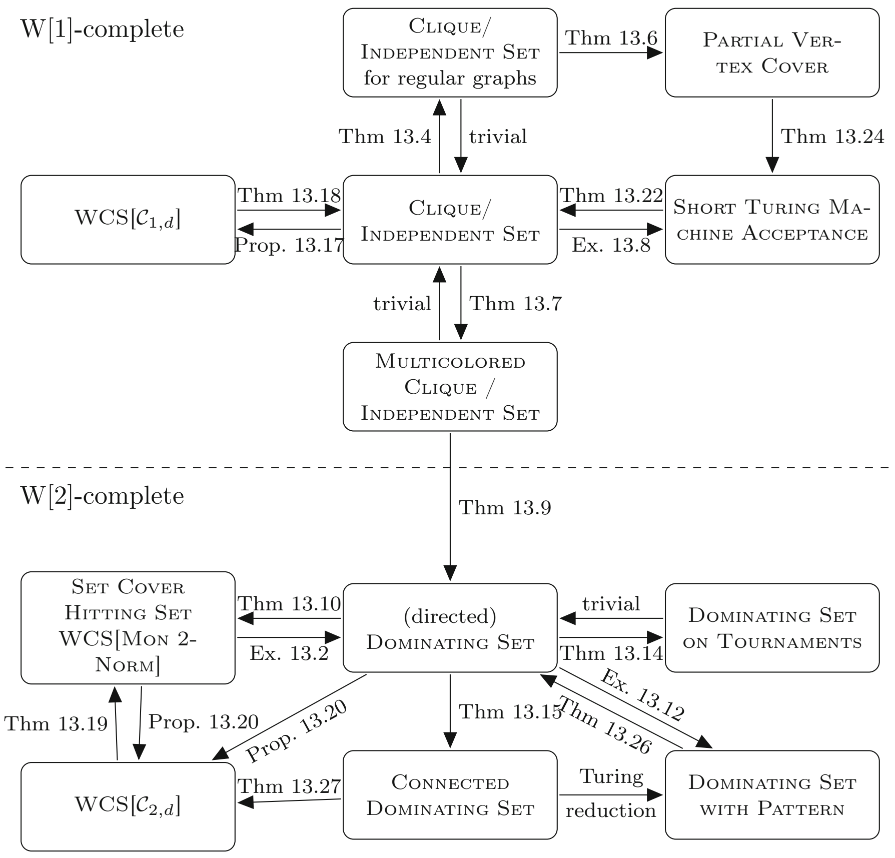


Fig. 13.4: W[1]-complete and W[2]-complete problems in Sections 13.2–*13.5 and the reductions between them. WCS[Mon 2-Norm] is short for Weighted Monotone 2-normalized Satisfiability

correspondence between the vertices/edges of the two graphs is fairly straightforward. The following reduction is an example where the correspondence is still relatively clear, although not completely trivial. In the Balanced Vertex Separator problem, we are given a graph G and integer k; the task is to find a set S of at most k vertices such that every component of $G - S$ has at most $|V(G)|/2$ vertices.

Theorem 13.29. Balanced Vertex Separator *is* W[1]*-hard.*

Proof. We give a parameterized reduction from Clique. Let (G, k) be an instance of Clique. Let $n = |V(G)|$ and $m = |E(G)|$. We may assume that n is sufficiently large compared to k, for example, $n \geq 4k^2$: adding $4k^2$

isolated vertices to the graph G does not change the problem. Let us set $\ell = n + m - 2(k + \binom{k}{2}) \geq 0$. We construct a graph G' as follows.

(i) The vertex set of G' contains $V(G)$, which forms a clique in G'.
(ii) For every edge $e = uv$ of G, we introduce a vertex w_e in G' and make it adjacent to u and v.
(iii) We introduce a clique K on ℓ new vertices as another component of G'.

The output instance is (G', k).

The intuition behind the construction is that if we remove a set S of vertices from the clique formed by $V(G)$ in G', then it creates some number of components of size 1, corresponding to the edges induced by S in G. Therefore, if we want to maximize the number of these components appearing (to reduce the size of the other components), then we need a set S that maximizes the number of edges induced in G, that is, forms a k-clique.

Note that G' has $n+m+\ell = 2(n+m-k-\binom{k}{2}) \geq 2\ell$ vertices. Suppose that S is a k-clique of G. Then $G' \setminus S$ has the following components: $\binom{k}{2}$ components of size 1, the clique K of size $\ell \leq |V(G')|/2$, and one component containing the remaining $(n + m + \ell) - |S| - \binom{k}{2} - \ell = n + m - k - \binom{k}{2} = |V(G')|/2$ vertices, that is, every component of $G' - S$ has size at most $|V(G')|/2$.

For the reverse direction, suppose that S is a set of at most k vertices in G' such that every component of $G' - S$ has size at most $|V(G')|/2 = n + m - k - \binom{k}{2}$. First we show that it can be assumed that $S \subseteq V(G)$. Clearly, it makes no sense to remove any vertex of the clique K of size ℓ, as the size of this component is already at most $|V(G')|/2$. Suppose that $w_e \in S$ for some edge $e = uv$. If $u, v \in S$, then we many omit w_e from S: now w_e becomes an isolated vertex in $G - (S \setminus \{w_e\})$. If, say, $u \notin S$, then $(S \setminus \{w_e\}) \cup \{u\}$ is also a solution: omitting w_e from S increases the size of the component of u by 1 (note that u and v are adjacent, hence they cannot be in two different components of $G' - S$), and introducing u into S decreases the size of this component by 1. Therefore, we may assume $S \subseteq V(G)$.

Consider the component C of $G' - S$ that contains the clique $V(G) \setminus S$ (as we have assumed that $n = |V(G)| > k$, the component C is nonempty). If an edge $e \in E(G)$ has an endpoint not in S, then vertex $w_e \notin S$ of G' is also in the component C. Thus if we denote by p the number of edges induced by S in G, then component C has size $(n - |S|) + m - p$. Then the bound on the size of C implies $(n - |S|) + m - p \leq |V(G')|/2 = n + m - k - \binom{k}{2}$, that is, we get $|S| + p \geq k + \binom{k}{2}$. As $|S| \leq k$, this is only possible if $|S| = k$ and S induces a clique in G. $\square$

13.6.2 Reductions with vertex representation

The most common technique for NP-hardness proofs is to represent parts of the original instance by gadgets. When reducing a graph problem to another graph problem, this is done by replacing each vertex and each edge of the original graph by a "gadget": a small graph satisfying certain properties. Then it is argued that these gadgets have properties ensuring that the solutions of the constructed instance correspond to the solutions of the original instance. A typical example of a reduction of this type is reducing Vertex Cover to Hamiltonian Cycle. Similarly, when reducing from 3-SAT, one may use gadgets representing the variables and clauses of the original formula. A typical example is reducing 3-SAT to Clique.

Representing vertices and edges of the original graph (or variables and clauses of the original formula) by gadgets is usually not a suitable technique for parameterized reductions. The problem is that, in most cases, by introducing a new gadget, one has to increase the parameter of the constructed instance. Therefore, the parameter of the new instance would be proportional to the number of vertices/edges/variables/clauses of the original instance, which can be much larger than the parameter of the original instance, violating the second requirement of Definition 13.1.

A typical parameterized reduction from Clique uses gadgets in a very different way than NP-hardness proofs. Instead of creating a gadget for each vertex/edge of the input graph, we create k gadgets, one for each vertex of the *solution.* If we can ensure that the new parameter is proportional to the number of gadgets, then we satisfy the requirement that the new parameter is bounded by a function of the original parameter.

A consequence of using only k gadgets is that we need very different, much more powerful gadgets than in typical NP-hardness proofs. For example, in a polynomial-time reduction from, say, Vertex Cover, the gadget needs to be able to represent only a constant number of different states: each vertex is either selected or not, and each edge can be covered in three possible ways (first endpoint, second endpoint, both endpoints). On the other hand, if we want a gadget to represent one of the vertices of the solution, then we need a gadget that is able to represent n different states, where n is the number of vertices of the original graph.

Theorem 13.9, the reduction from Multicolored Independent Set to Dominating Set, can be considered as an example of representing the vertices of the solution by gadgets. Selecting one vertex of the clique V_i of G' corresponds to selecting a vertex of V_i into the independent set. Thus we may view the clique V_i as a gadget with $|V_i|$ possible states. Then the vertices w_e "verify" that the states of the gadgets $V_1, \ldots, V_k$ correspond to an independent set of G.

The following reduction demonstrates the same principle: we want to represent the k vertices of the solution by k gadgets and, for any two of these gadgets, we want to ensure that they represent nonadjacent vertices. This reduction also serves as an example for a reduction to a problem where the parameter is not related to the solution size, but it is a structural parameter measuring some property of the instance (in our case, the treewidth). In such a reduction, what we need to ensure is that the structural parameter is proportional to the number of gadgets.

List Coloring is a generalization of Vertex Coloring: given a graph G, a set of colors C, and a list function $L : V(G) \to 2^C$ (that is, a subset of colors $L(v)$ for each vertex v), the task is to assign a color $c(v) \in L(v)$ to each vertex $v \in V(G)$ such that adjacent vertices receive different colors. Vertex Coloring on a graph with treewidth w can be solved in time $2^{\mathcal{O}(w \log w)} \cdot n^{\mathcal{O}(1)}$ by standard dynamic-programming techniques (Theorem 7.10). However, the straightforward application of dynamic programming yields only a $n^{\mathcal{O}(w)}$-time algorithm for the more general List Coloring problem (Exercise 7.20). The following theorem shows that there is a reason for that: the problem is W[1]-hard parameterized by treewidth.

Theorem 13.30. List Coloring *is* W[1]*-hard when parameterized by the treewidth of the graph.*

Proof. We present a parameterized reduction from Multicolored Independent Set. Let $(G, k, (V_1, \ldots, V_k))$ be an instance of Multicolored Independent Set. The set C of colors corresponds to $V(G)$. We construct a graph G' in the following way (see Fig. 13.5).

(i) For every $1 \le i \le k$, we introduce a vertex u_i with $L(u_i) = V_i$.
(ii) For every $1 \le i < j \le k$ and for every edge ab with $a \in V_i$, $b \in V_j$, we introduce a vertex $w_{i,j,a,b}$ with $L(w_{i,j,a,b}) = \{a, b\}$ and make it adjacent to u_i and u_j.

Observe that $\{u_1, \ldots, u_k\}$ is a vertex cover of G', hence G' has treewidth at most k. Therefore, the reduction satisfies the requirement that the parameter of the constructed instance is bounded by a function of the parameter of the original instance. Suppose that G has an independent set of size k, consisting of vertices $v_1 \in V_1, \ldots, v_k \in V_k$. We can construct the following list coloring of G'. First, we set the color of u_i to $v_i \in V_i$. Then for every vertex $w_{i,j,a,b}$, it is true that either u_i has a color different from a, or u_j has a color different from b (as a and b are adjacent, while v_i and v_j are not). Therefore, one of the two colors appearing in the list of $w_{i,j,a,b}$ is not used on its neighbors; hence we can extend the coloring to $w_{i,j,a,b}$.

For the reverse direction, suppose that $c\colon V(G') \to V(G)$ is a list coloring of G'. We claim that $c(u_1) \in L(u_1) = V_1, \ldots, c(u_k) \in L(u_k) = V_k$ is an independent set in G. For a contradiction, suppose that $a = c(u_i)$ and $b =$

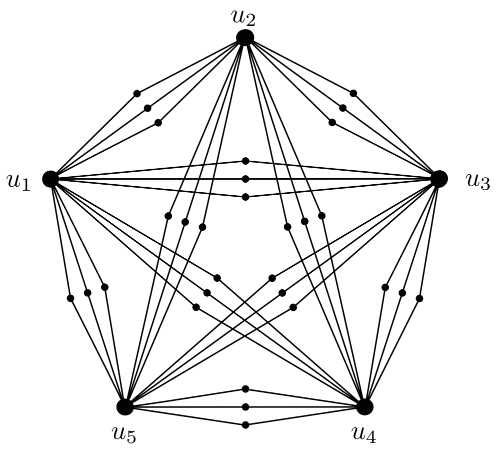


Fig. 13.5: The graph G' of the List Coloring instance constructed in the reduction of Theorem 13.30 for $k = 5$

$c(u_j)$ are adjacent for some $1 \leq i < j \leq k$. Then the vertex $w_{i,j,a,b}$ exists in G with list $\{a, b\}$. However, this vertex is adjacent to both u_i and u_j, hence $c(w_{i,j,a,b})$ can be neither a nor b, a contradiction. $\square$

13.6.3 Reductions with vertex and edge representation

In reductions from Clique/Independent Set based on vertex representation, we need to enforce that gadgets represent adjacent/nonadjacent vertices. In Theorem 13.9 we enforced this condition by introducing the vertices w_e. Introducing these vertices had no effect on the parameter, as these vertices were supposed to be dominated by the k vertices selected from $V_1, \ldots, V_k$. Similarly, in Theorem 13.30, we introduced the vertices $w_{i,j,a,b}$ to enforce that the coloring on $w_1, \ldots, w_k$ represents an independent set. The introduction of these vertices did not increase the treewidth of the constructed instance. However, there are reductions where it seems difficult to enforce the adjacency/nonadjacency condition on two gadgets. We present two examples of such problems and show how to prove W[1]-hardness in this case by representing both the edges and the vertices of the clique by gadgets.

Example 1: Odd Set and variants

In the Odd Set problem, where given a set system $\mathcal{F}$ over a universe U and an integer k, the task is to find a subset $S \subseteq U$ of size at most k such that the intersection of S with every set in $\mathcal{F}$ has odd size. A natural idea for reducing an instance $(G,k,(V_1,\ldots,V_k))$ of MULTICOLORED INDEPENDENT SET to an instance of ODD SET would be to let U be $V(G)$, keep the same parameter k, and introduce the sets $V_1,\ldots,V_k$. The only way a set S of size at most k can have odd intersection with these k disjoint sets is to have exactly one vertex from each of V_1, $\ldots$, V_k. Now we would like to introduce sets into $\mathcal{F}$ to enforce that the vertex of the solution in V_i is not adjacent to the vertex of the solution in V_j. If $a \in V_i$ and $b \in V_j$ are adjacent, then introducing the set $\{a,b\}$ into $\mathcal{F}$ certainly prevents the possibility that both a and b is in a solution S. However, it also prevents the possibility that neither of a and b is in S, which may very well be the case for a solution. The reader may try other simple tricks, but it seems that the parity requirement is not strong enough to represent the complicated condition that two vertices are not adjacent.

We get around this difficulty by a reduction from MULTICOLORED CLIQUE where we represent both the vertices and the edges of the clique, that is, we have k vertex gadgets and $\binom{k}{2}$ edge gadgets. Then what we need to ensure is that if an edge gadget represents the edge uv, then the corresponding vertex gadgets represent u and v, respectively. As we shall see, such conditions are much easier to express and enforce.

Theorem 13.31. ODD SET *is* W[1]*-hard.*

Proof. We present a parameterized reduction from MULTICOLORED CLIQUE. Let $(G,k,(V_1,\ldots,V_k))$ be an instance of MULTICOLORED CLIQUE. Let $E_{i,j}$ be the set of edges between V_i and V_j, and let $E_{i,j,v}$ be the subset of $E_{i,j}$ consisting of the edges incident to v. We construct an instance $(\mathcal{F},U,k')$ of ODD SET as follows.

(i) The universe is $U := \bigcup_{i=1}^{k} V_i \cup \bigcup_{1\le i<j\le k} E_{i,j}$.
(ii) Let $k' := k + \binom{k}{2}$.
(iii) For every $1 \le i \le k$, let $\mathcal{F}$ contain V_i, and for every $1 \le i < j \le k$, let $\mathcal{F}$ contain $E_{i,j}$.
(iv) For every $1 \le i < j \le k$, let $\mathcal{F}$ contain $E'_{i,j,v} := E_{i,j,v} \cup (V_i \setminus \{v\})$ for every $v \in V_i$ and $E'_{i,j,w} := E_{i,j,w} \cup (V_j \setminus \{w\})$ for every $w \in V_j$.

The intuitive idea of the reduction is the following. The solution needs to select one vertex from each of V_1, $\ldots$, V_k (describing k vertices of G) and one vertex from each $E_{i,j}$ (describing $\binom{k}{2}$ edges of G). Then the sets introduced in (iv) ensure that the vertices selected from V_i and V_j are the endpoints of the edge represented by the vertex selected from $E_{i,j}$. More precisely, the set $E'_{i,j,v}$ ensures that exactly one of $V_i \setminus \{v\}$ and $E_{i,j,v}$ contains a vertex of the solution, that is, $v \in V_i$ is in the solution if and only if the solution contains a vertex of $E_{i,j,v}$.

Formally, let $v_1 \in V_1, \ldots, v_k \in V_k$ be a k-clique in G and let $e_{i,j}$ be the edge between v_i and v_j. We claim that the set $S := \bigcup_{i=1}^{k}\{v_i\} \cup \bigcup_{1\le i<j\le k}\{e_{i,j}\}$ has odd intersection with every set in $\mathcal{F}$. This is clearly true for the sets V_i and $E_{i,j}$. Consider now a set $E'_{i,j,v}$ introduced in (iv). Suppose that $v \in V_i$ (the argument is similar for the case $v \in V_j$). If the endpoint of $e_{i,j}$ in V_i is v, then $v = v_i$ and $e_{i,j} \in E_{i,j,v}$ hold and hence $S \cap E'_{i,j,v} = \{e_{i,j}\}$, which has odd size. If the endpoint of $e_{i,j}$ in V_i is not v, then $v \neq v_i$ and $e_{i,j} \notin E_{i,j,v}$, hence $S \cap E'_{i,j,v} = \{v_i\}$, which is again odd.

For the reverse direction, let $S \subseteq U$ be a set of size at most k'. As the $k' = k + \binom{k}{2}$ sets introduced in (iii) are disjoint, each of them contains exactly one vertex of S. Let $S \cap V_i = \{v_i\}$ and $S \cap E_{i,j} = \{e_{i,j}\}$. We claim that $v_1 \in V_1$, $\ldots, v_k \in V_k$ is a k-clique in G and $e_{i,j}$ is the edge between v_i and v_j. Suppose that v_i is not an endpoint of $e_{i,j}$. Then $v_i \notin E'_{i,j,v_i}$ and $e_{i,j} \notin E'_{i,j,v_i}$ hold and hence $S \cap E'_{i,j,v_i} = \emptyset$, a contradiction. The argument is similar if v_j is not an endpoint of $e_{i,j}$. Therefore, $e_{i,j}$ is the edge between v_i and v_j; hence $\{v_1, \ldots, v_k\}$ indeed induces a clique in G. □

In general, the requirement that $a \in V_i$ and $b \in V_j$ are adjacent can be an arbitrarily complicated binary relation: the bipartite graph between V_i and V_j can be arbitrarily complicated. The key idea in the proof of Theorem 13.31 is that the requirement that $v \in V_i$ is an endpoint of an edge $e \in E_{i,j}$ is a much simpler binary relation: it is a projection, that is, for every $e \in E_{i,j}$, there is only a single $v \in V_i$ that satisfies the requirement. The general principle that we can learn from this example is that, after setting up the gadgets representing the k vertices and the $\binom{k}{2}$ edges of the clique, all we need to ensure is that each state of an edge gadget *forces* a particular state for the the two vertex gadgets representing the endpoints of the edge. That is, we do not need to implement arbitrarily complicated relations between the gadgets, only a very specific simple relation requiring that a state of an edge gadget always implies a particular state on a vertex gadget.

The proof of Theorem 13.31 allows us to prove W[1]-hardness for some variants of Odd Set. First, Exact Odd Set is the variant that asks for a solution of size *exactly* k. Observe that, given a yes-instance of Multicolored Clique, the reduction of Theorem 13.31 produces an instance of Odd Set that has a solution of size exactly k'. Therefore, this is also a correct parameterized reduction from Multicolored Clique to Exact Odd Set, proving the W[1]-hardness of the latter problem.

One can define the Even Set and Exact Even Set problems analogously: now each set has to contain an even number of vertices of the solution. Note, however, that the empty set is always a correct solution for Even Set with this definition. Therefore, in the Even Set problem, we require the solution to be a *nonempty* set of at most k elements such that every set in the instance contains an even number of elements of the solution. It is not difficult to reduce Exact Odd Set to Exact Even Set (Exercise 13.19), showing that Exact Even Set is also W[1]-hard. On the other hand, the fixed-parameterized tractability of Even Set is a notorious open question. It is equivalent to other very interesting and deep questions: it can be also formulated as finding the minimum distance of a binary linear code or the minimum length cycle in a binary matroid. Therefore, settling this question may require a deeper combinatorial understanding and could be more challenging than the simple reduction of Theorem 13.31.

Unique Hitting Set is the variant of Odd Set where we require that each set contains *exactly one* element of the solution; Exact Unique Hitting Set is defined analogously. Looking at the reduction in Theorem 13.31, we can observe that if the Multicolored Clique instance is a yes-instance, then the constructed instance of Odd Set has the property that every set contains in fact only a single element of the solution. Therefore, the proof of Theorem 13.31 actually provides a reduction from Multicolored Clique to Unique Hitting Set or Exact Unique Hitting Set.

Theorem 13.32. *The following problems are* W[1]*-hard:*

- Odd Set
- Exact Odd Set
- Exact Even Set
- Unique Hitting Set
- Exact Unique Hitting Set

Interestingly, one can prove that Unique Hitting Set and Exact Unique Hitting Set are actually W[1]-complete by a reduction to Short Turing Machine Acceptance (Exercise 13.20), but this is open for the other three problems.

It is natural to define analogues of Odd Set (and its variants) on graphs with the closed neighborhoods of the vertices playing the role of the set system, the same way as Dominating Set can be considered as a graph-theoretic analogue of Hitting Set. For example, the graph version of Odd Set asks for a set of at most k vertices such that $N[v]$ contains an odd number of vertices of the solution for every vertex v. The graph version of Exact Unique Hitting Set is also known under the name Perfect Code. If the vertex set of a graph G is the set of all potential codewords and two vertices are adjacent if the corresponding codewords are "close," then Perfect Code asks for a code containing exactly k words such that every word is close to exactly one word in the code. One can show that the graph versions of all five problems considered in Theorem 13.32 are also W[1]-hard (Exercise 13.21).

Example 2: Steiner problems in directed graphs

Given an undirected graph G, a set $K \subseteq V(G)$ of terminals, and an integer ℓ, the Steiner Tree problem asks for a tree H with at most ℓ edges connecting all the terminals. In Sections 6.1.2 and 10.1.2 we gave algorithms for Steiner Tree running in time $3^{|K|}n^{\mathcal{O}(1)}$ and $2^{|K|}n^{\mathcal{O}(1)}$, respectively. One can define similar problems on directed graphs. Given a set of terminals $K \subseteq V(G)$ and a root $r \in V(G)$, DIRECTED STEINER TREE asks for a directed tree rooted at r such that every terminal in K is reachable from r on the tree. This problem generalizes STEINER TREE to directed graphs in the sense that we are looking for a subgraph that provides reachability from one distinguished vertex to every other terminal. See Exercise 6.7 for a $3^{|K|}n^{\mathcal{O}(1)}$-time algorithm for DIRECTED STEINER TREE. Another, equally natural, generalization is the STRONGLY CONNECTED STEINER SUBGRAPH problem, where the input is a directed graph G, a set $K \subseteq V(G)$ of terminals, and an integer ℓ; the task is to find a strongly connected subgraph of G with at most ℓ vertices that contains every vertex of K. This problem generalizes STEINER TREE in the sense that we have to provide a path from any terminal to every other terminal. Note that one can define STRONGLY CONNECTED STEINER SUBGRAPH by requiring either at most ℓ edges or at most ℓ vertices in the solution. The two versions are different, but here we discuss only the vertex version for simplicity. The parameter we consider is the number ℓ of vertices in the solution. Unlike DIRECTED STEINER TREE, this problem is W[1]-hard and the proof is another example of the edge representation strategy.

Theorem 13.33. STRONGLY CONNECTED STEINER SUBGRAPH *is* W[1]-*hard.*

Proof. We present a parameterized reduction from MULTICOLORED CLIQUE. Let $(G, k, (V_1, \ldots, V_k))$ be an instance of MULTICOLORED CLIQUE. Let $E_{i,j}$ be the set of edges between V_i and V_j. We construct an instance (G', K, ℓ) of STRONGLY CONNECTED STEINER SUBGRAPH as follows (see Fig. 13.6).

(i) G' contains all the vertices of G, a vertex x, vertices $y_{i,j}$ $(1 \leq i < j \leq k)$, and a vertex w_e for every $e \in E(G)$. We define $E'_{i,j} = \{w_e \;:\; e \in E_{i,j}\}$ for $1 \leq i < j \leq k$.
(ii) Let $K := \{x\} \cup \{y_{i,j} \;:\; 1 \leq i < j \leq k\}$ and let $\ell := k + 1 + 2\binom{k}{2} = |K| + k + \binom{k}{2}$.
(iii) For every $1 \leq i \leq k$ and $v \in V_i$, we introduce the edges (x, v) and (v, x).
(iv) For every $1 \leq i < j \leq k$ and $e \in E_{i,j}$, we introduce the edges $(y_{i,j}, w_e)$ and $(w_e, y_{i,j})$.
(v) For every $1 \leq i < j \leq k$ and $e \in E_{i,j}$, if $a \in V_i$ and $b \in V_j$ are the endpoints of e, then we introduce the edges (a, w_e) and (w_e, b).

The intuitive idea of the reduction is the following. The solution can afford to select only one vertex from each of $V_1, \ldots, V_k$ ("vertex gadgets" describing k vertices of G) and one vertex from each $E'_{i,j}$ ("edge gadgets" describing $\binom{k}{2}$ edges of G). Each vertex w_e in $E_{i,j}$ has only one inneighbor and one outneighbor different from $y_{i,j}$; thus if w_e is part of the solution, then those two vertices have to be selected as well.

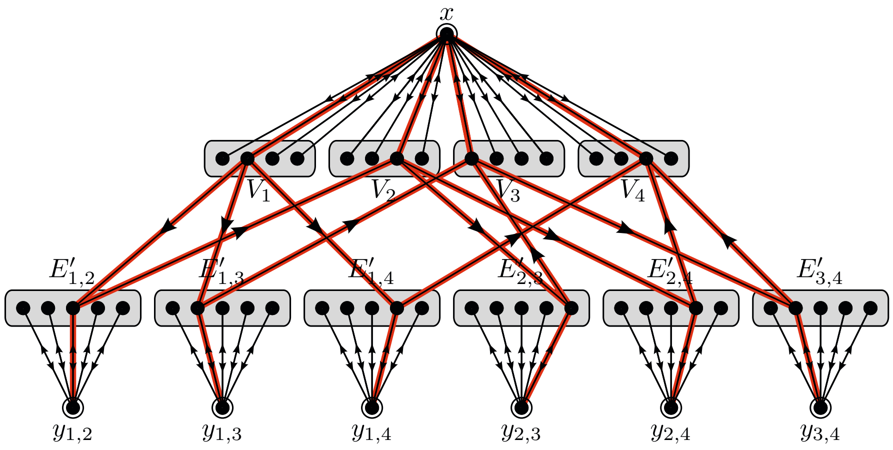


Fig. 13.6: An overview of the reduction in the proof of Theorem 13.33. The circled vertices are the terminals and the red edges show a possible solution. For clarity, we show only those edges of the graph between the V_i's and $E'_{i,j}$ that are in the solution

Therefore, the state of each edge gadget forces a state on two vertex gadgets, implying that the k-vertex gadgets describe a k-clique in G.

Formally, let $v_1 \in V_1, \ldots, v_k \in V_k$ be a k-clique in G and let $e_{i,j}$ be the edge between v_i and v_j. We claim that the set $S := K \cup \{v_i \;:\; 1 \le i \le k\} \cup \{e_{i,j} \;:\; 1 \le i < j \le k\}$ induces a strongly connected subgraph $G'[S]$; note that $|S| = \ell$. For every $1 \le i < j \le k$, the directed closed walk $xv_iw_{e_{i,j}}y_{i,j}w_{e_{i,j}}v_jx$ uses only vertices of S. Moreover, these $\binom{k}{2}$ directed closed walks cover every vertex of S, hence $G'[S]$ is strongly connected.

For the reverse direction, let S be a set of size at most ℓ such that $K \subseteq S \subseteq V(G')$ and $G'[S]$ is strongly connected. Clearly, such a set has to include at least one outneighbor of each vertex in K, which means that S contains at least one vertex of $E'_{i,j}$ for every $1 \le i < j \le k$. All the inneighbors of $E'_{i,j} \cup \{y_{i,j}\}$ are in V_i, while all the outneighbors are in V_j, hence each of V_i and V_j has to contain at least one vertex of the solution. Taking into account the quota of $\ell = |K| + k + \binom{k}{2}$ on the size of the solution, this is only possible if S contains exactly one vertex $v_i \in V_i$ for every $1 \le i \le k$ and exactly one vertex $w_{e_{i,j}} \in E'_{i,j}$ for every $1 \le i < j \le k$. We claim that $v_1 \in V_1, \ldots, v_k \in V_k$ is a k-clique in G and $e_{i,j}$ is the edge between v_i and v_j. Suppose that v_i is not an endpoint of $e_{i,j}$. Then v_i is not an inneighbor of $w_{e_{i,j}}$. The only other inneighbor of $w_{e_{i,j}}$ is $y_{i,j}$ and the only inneighbor of $y_{i,j}$ in S is $w_{e_{i,j}}$ itself. Therefore, the strongly connected component of $G'[S]$ containing

$w_{e_{i,j}}$ consists of only $y_{i,j}$ and $w_{e_{i,j}}$, a contradiction. The argument is similar if v_j is not an endpoint of $e_{i,j}$. Therefore, $e_{i,j}$ is the edge between v_i and v_j, hence $\{v_1, \ldots, v_k\}$ indeed induces a clique in G. □

Note that the reduction in Theorem 13.33 creates a directed graph containing bidirected edges, that is, there are pairs of vertices that are connected by directed edges in both directions. Exercise 13.26 asks for a reduction that does not create bidirected edges.

Exercises

13.1 (✐). Is there a parameterized reduction from VERTEX COVER to INDEPENDENT SET?

13.2 (✐). Give a parameterized reduction from SET COVER to DOMINATING SET.

13.3 (✐). In the MULTICOLORED BICLIQUE problem the input consists of a bipartite graph G with bipartition classes A, B, an integer k, a partition of A into k sets $A_1, A_2, \ldots, A_k$, and a partition of B into k sets $B_1, B_2, \ldots, B_k$; the question is whether there exists a subgraph of G isomorphic to the biclique $K_{k,k}$, with one vertex in each of the sets A_i and B_i. Prove that MULTICOLORED BICLIQUE is W[1]-hard.

13.4. Prove Theorem 13.12.

13.5. In this exercise we will work out an alternative, purely combinatorial construction of k-paradoxical tournaments of size $2^{\mathrm{poly}(k)}$.

- Find a 2-paradoxical tournament T^* on seven vertices.
- Assume we are given some tournament T. Construct a tournament T' as follows. The vertices of T' are triples of vertices of T, i.e., $V(T') = V(T) \times V(T) \times V(T)$. Let us consider a pair of triples $u_1 = (a_1, b_1, c_1)$ and $u_2 = (a_2, b_2, c_2)$, where $a_1 \neq a_2$, $b_1 \neq b_2$, and $c_1 \neq c_2$. Consider now pairs $\{a_1, a_2\}$, $\{b_1, b_2\}$, $\{c_1, c_2\}$, and count for how many of them the edge in T was directed from the vertex with subscript 1 to the vertex of subscript 2 (e.g., from a_1 to a_2). If for at least two pairs this was the case, we put $(u_1, u_2) \in E(T')$, and otherwise we put $(u_2, u_1) \in E(T')$. For all the pairs of triples where at least one of the coordinates is the same, we put the edge between the triples arbitrarily. Prove that if T was k-paradoxical, then T' is $\lfloor \frac{3k}{2} \rfloor$-paradoxical.
- Define the sequence $T_0, T_1, T_2, \ldots$ as follows: $T_0 = T^*$, and for $m \geq 1$ the tournament T_m is constructed from T_{m-1} using the construction from the previous point. Prove that $|V(T_m)| = 7^{3^m}$ and that T_m is $g(m)$-paradoxical for a function $g(m) \in \Omega((3/2)^m)$.
- Using the previous point, provide an algorithm that, given an integer k, constructs a k-paradoxical tournament of size $2^{\mathcal{O}\left(k^{\log_{3/2} 3}\right)} \leq 2^{\mathcal{O}(k^{2.71})}$. The construction should work in time polynomial with respect to the size of the constructed tournament.

13.6. Let $\mathcal{C}_{s,t,d}$ be the class of circuits having depth at most d where every path from an input node to the output node contains at most t nodes with indegree larger than s. (Thus the class $\mathcal{C}_{t,d}$ defined in Section 13.3 is $\mathcal{C}_{2,t,d}$.) Show that, for every $t, d \geq 1$, $s \geq 2$, there is a parameterized reduction from WCS[$\mathcal{C}_{s,t,d}$] to WCS[$\mathcal{C}_{t,ds}$]. Conclude that replacing $\mathcal{C}_{t,d}$ in Definition 13.16 with $\mathcal{C}_{s,t,d}$ for some fixed $s \geq 2$ would not change the definition of W[t].

13.7. Consider the WEIGHTED CIRCUIT SATISFIABILITY problem, restricted to *monotone* (i.e., without negations) circuits from the class $\mathcal{C}_{1,d}$ for some fixed constant d. Prove that this problem is FPT when parameterized by k, the weight of the assignment in question.

13.8. Give a parameterized reduction from Independent Set to Short Turing Machine Acceptance.

13.9. Consider a restricted variant of the Short Turing Machine Acceptance problem, where we consider only instances (M, x, k) with x being an empty string. Reduce the general Short Turing Machine Acceptance problem to the restricted one.

13.10. Work out the details of the parameterized reduction from Partial Vertex Cover to Short Turing Machine Acceptance (Theorem 13.24).

13.11. Prove that Dominating Set remains W[2]-hard even if restricted to graphs excluding the star $K_{1,4}$ as an induced subgraph.

13.12. Give a parameterized reduction from Dominating Set to Dominating Set with Pattern.

13.13 (☠). Modify the reduction in the proof of Theorem 13.26 to obtain a parameterized many-one reduction from Connected Dominating Set to Dominating Set.

13.14. Given a graph G and an integer k, the Induced Matching problem asks for an induced matching of size k, that is, k edges $x_1y_1, \ldots, x_ky_k$ such that the $2k$ endpoints are all distinct and there is no edge between $\{x_i, y_i\}$ and $\{x_j, y_j\}$ for any $i \neq j$. Prove that Induced Matching is W[1]-complete.

13.15. Given a graph G and an integer k, the Independent Dominating Set problem asks for a set of exactly k vertices that is both an independent set and a dominating set. Prove that Independent Dominating Set is W[2]-complete.

13.16. In the Long Induced Path problem the input consists of a graph G and an integer k, and the question is whether G contains the path on k vertices as an induced subgraph. Prove that this problem is W[1]-hard when parameterized by k.

13.17. Consider the following variant of the Steiner Tree problem: We are given a graph G, a set of terminals $K \subseteq V(G)$, and an integer parameter k. The question is whether one can find a set $X \subseteq V(G) \setminus K$ of cardinality at most k such that $G[K \cup X]$ is connected. In other words, we parameterize the Steiner Tree by the number of nonterminals that can be added to the tree, while the number of terminals can be unbounded. Prove that this parameterization of Steiner Tree is W[2]-hard.

13.18 (✎). Is List Coloring W[1]-hard parameterized by the vertex cover number?

13.19. Give a parameterized reduction from Exact Odd Set to Exact Even Set.

13.20 (☠). Show that Unique Hitting Set and Exact Unique Hitting Set are in W[1].

13.21. Show that the graph versions of all five problems in Theorem 13.32 are W[1]-hard. (The graph version of Exact Unique Hitting Set is called Perfect Code.)

13.22. In the Subset Sum problem the input consists of a sequence of n integers $a_1, a_2, \ldots, a_n$ and two integers s, k, and the question is whether one can find a set $I \subseteq \{1, 2, \ldots, n\}$ of exactly k indices such that $\sum_{i \in I} a_i = s$. Prove that Subset Sum is W[1]-hard when parameterized by k.

13.23. In the Set Packing problem the input consists of family $\mathcal{F}$ of subsets of a finite universe U and an integer k, and the question is whether one can find k pairwise disjoint sets in $\mathcal{F}$. Prove that Set Packing is W[1]-hard when parameterized by k.

13.24. In the Exact CNF-SAT problem the input consists of formula φ in the conjunctive normal form and an integer k, and the question is whether one can find an assignment ψ of weight exactly k such that in each clause of φ exactly one literal is satisfied. Prove that Exact CNF-SAT is W[1]-hard when parameterized by k.

13.25. In the Permutation Composition problem the input consists of a family $\mathcal{P}$ of permutations of a finite universe U, additional permutation π of U, and an integer k, and the question is whether one can find a sequence $\pi_1, \pi_2, \ldots, \pi_k \in \mathcal{P}$ such that $\pi = \pi_1 \circ \pi_2 \circ \ldots \circ \pi_k$. Prove that Permutation Composition is W[1]-hard when parameterized by k.

13.26 (✎). Modify the reduction of Theorem 13.33 in such a way that the constructed graph has no bidirected edges.

13.27. In the Vertex k-Way Cut problem the input consists of a graph G and two integers k, s, and the question is whether one can find a set X of at most k vertices such that the graph $G - X$ has at least s connected components. Prove that Vertex k-Way Cut is W[1]-hard when parameterized by $k + s$.

13.28. Given a bipartite graph G with bipartite classes $A, B \subseteq V(G)$ and an integer k, the Hall Set problem asks for a *Hall set* of size at most k, that is, a set $S \subseteq A$ of size at most k such that $|N(S)| < |S|$. Show that Hall Set is W[1]-hard.

13.29 (☠). Recall that a graph is 2-degenerate if its every subgraph contains a vertex of degree at most 2. In the 2-degenerate Vertex Deletion problem one is given a graph G and an integer k, and the question is whether there exists a set X of at most k vertices such that the graph $G - X$ is 2-degenerate. Prove that 2-degenerate Vertex Deletion is W[2]-hard when parameterized by k (if you find it more convenient, you may prove just W[1]-hardness).

13.30 (☠). In the Longest Common Subsequence problem the input consists of a finite alphabet Σ, a set of strings $s_1, s_2, \ldots, s_\ell \in \Sigma^*$ and an integer k, and the question is whether there exists a string $s \in \Sigma^*$ of length k such that s is a subsequence of s_i for every $1 \leq i \leq s$. Prove that Longest Common Subsequence is W[1]-hard when parameterized by k.

Hints

13.1 The following algorithm satisfies the formal definition of a parameterized reduction: solve the Vertex Cover instance in FPT time and output a trivial yes-instance or no-instance of Independent Set. More generally, if A is FPT, then A has a parameterized reduction to any parameterized problem B that is nondegenerate in the sense that it has at least one yes-instance and at least one no-instance.

13.4 Sample the tournament by choosing the orientation of each edge independently and uniformly at random. Prove that for $n \geq r(k)$ the sampled tournament is k-paradoxical with positive probability.

13.6 Every OR-node or AND-node with indegree at most s can be simulated with a chain of at most $s - 1$ nodes with indegree 2. This way, we can construct an equivalent circuit such that depth increases only by a factor of at most s. In fact, by replacing a node with indegree s with an (almost) complete binary tree of indegree-2 nodes, we can restrict the increase of depth to $\mathcal{O}(\log s)$.

13.7 Observe that there is only a constant number of large gates. Guess the output of all large gates. For each large OR gate, guess which in-edge of this gate is satisfied, and what the assignment is of the input gates that make this particular in-edge satisfied (it depends only on a constant number of input gates). Finally, to resolve large AND-gates, design a simple branching algorithm.

13.8 In the first k steps, the Turing machine guesses k vertices and writes them on the tape. In the next $\binom{k}{2}$ phases, the Turing machine checks whether every pair of the guessed vertices is distinct and nonadjacent. Each phase can be implemented in $\mathcal{O}(k)$ steps.

13.9 Modify the Turing machine in such a way that it writes x on the tape in the first $|x|$ steps (and then the head returns to the starting position).

13.11 Try to adjust the reduction for Connected Dominating Set of Theorem 13.15 so that the output graph has no $K_{1,4}$ as an induced subgraph. You would probably want to turn as many big parts of the graph into cliques as possible.

13.12 The reduction of Theorem 13.15 from Dominating Set to Connected Dominating Set has the property that if the created instance has a connected dominating set of size k, then it has a dominating set of size k inducing a clique.

13.13 Express Connected Dominating Set as $2^{\mathcal{O}(k^2)}$ instances of Dominating Set with Pattern and reduce them to $2^{\mathcal{O}(k^2)}$ instances of Dominating Set (or Set Cover if you will). Then use the composition algorithm of Section 15.2.3 to create a single instance of Set Cover.

13.16 Reduce from Multicolored Independent Set. Given a Multicolored Independent Set instance $(G, k, (V_1, V_2, \ldots, V_k))$, aim at an instance (G', k') of Long Induced Path with $k' = 3k$. In the intended solution, the $(3i-1)$-th vertex of the path belongs to V_i, whereas the remaining vertices belong to some gadgets you introduce.

13.17 Reduce from Set Cover.

13.18 Note that treewidth of a graph is at most its vertex cover number. Therefore, the statement of Theorem 13.30 (W[1]-hardness parmeterized by treewidth) *does not* imply W[1]-hardness parameterized by vertex cover number, as vertex cover can be a much larger parameter. However, by looking at the proof of Theorem 13.30, we can observe that the reduction creates instances of List Coloring with vertex cover number at most k, which means that the same proof proves also the (stronger) result that the problem is W[1]-hard parameterized by the vertex cover number.

13.19 By introducing a new element x and a new set $\{x\}$ on it, we may assume that the parameter k of the Exact Odd Set instance is even. To create an instance of Exact Even Set, introduce a new element y, add it to every set, let us introduce a new set $U \setminus \{y\}$, and set $k' := k + 1$ (which is an odd number). Now the set $U \setminus \{y\}$ ensures that every solution of size exactly k' contains y. Consequently, if A is a set of the Exact Odd Set instance, then the fact that the solution of the Exact Even Set instance has even intersection with $A \cup \{y\}$ implies that it has odd intersection with A.

13.20 We can prove membership by a reduction to Short Turing Machine Acceptance. The Turing machine first guesses the (at most k) elements of the solution. Then by testing all the $\binom{k}{2}$ pairs, it verifies that there is no set in the instance that contains two elements of the solution. Finally, to check that all the sets are hit by the solution, the Turing machine sums the total degree of the elements (that is, the number of sets they are contained in) of the solution. As no set is hit by more than one element of the solution, this sum is exactly the number of sets hit by the solution. To implement this algorithm in a Turing machine, we have to hard-code which pairs of elements are contained in some set together and the degree of each element.

13.22 Reduce from Perfect Code or Exact Unique Hitting Set.

13.23 Reduce from Independent Set: for each vertex v of the input instance (G, k), create a set F_v consisting of all edges of G incident to v.

13.24 Reduce from Perfect Code or Exact Unique Hitting Set.

13.25 Reduce from Perfect Code or Exact Unique Hitting Set.

13.26 Split each nonterminal vertex v into two vertices v_{in} and v_{out} connected by an edge $(v_{\text{in}}, v_{\text{out}})$. Modify k' appropriately.

13.27 Try a reduction similar to the one given in the proof of Theorem 13.29.

13.28 Reduction from Clique. Given a graph G, we construct a bipartite graph where class A corresponds to the edges of G and class B corresponds to the vertices of B; the vertex of A corresponding to edge uv of G is adjacent to the two vertices $u, v \in B$. Additionally, we introduce a set of $\binom{k}{2} - k - 1$ vertices into B and make them adjacent to every vertex of A. Show that every Hall set of size at most $\binom{k}{2}$ has size exactly $\binom{k}{2}$ and corresponds to the edges of a k-clique in G.

13.29 You may find the following characterization of d-degenerate graphs useful: a graph is d-degenerate if one can delete all the vertices of the graph one by one, each time removing a vertex that at the current moment has degree at most d.

Start the reduction from Set Cover. Create k set-choice gadgets that emulate choices of k sets of the solution. Each gadget should be 3-regular, and should become 2-degenerate after removing any of its vertices. Then create a gadget for each element of the universe, and wire these gadgets with the set-choice gadgets to encode the input instance. Ensure the following properties: (i) If an element is not covered, then the closed neighborhood of the corresponding gadget induces a 3-regular graph untouched by the solution. (ii) If an element is covered, then the closed neighborhood of the corresponding gadget is affected by the solution and it is possible to "rip" the whole gadget in the process described in the previous paragraph.

13.30 Reduce from Multicolored Clique. Let $(G, k, (V_1, V_2, \ldots, V_k))$ be a Multicolored Clique instance. The string s should encode the choice of vertices and edges of the clique, that is, $V(G) \cup E(G) \subseteq \Sigma$ (you may need some additional special symbols in Σ that will serve as separators or markers). To this end, fix an arbitrary total order $\preceq$ on each set V_i. Let $V_i^{\uparrow}$ be the string consisting of all vertices of V_i in the increasing order of $\preceq$, and $V_i^{\downarrow}$ be the string $V_i^{\uparrow}$ reversed. The orders on V_i and V_j impose a lexicographical order on each set $E_{i,j}$ of edges connecting V_i and V_j in the graph G, and hence the strings $E_{i,j}^{\uparrow}$ and $E_{i,j}^{\downarrow}$ are defined in a natural manner. Consider two strings

$$s_1 = V_1^{\uparrow} V_2^{\uparrow} \ldots V_k^{\uparrow} E_{1,2}^{\uparrow} E_{1,3}^{\uparrow} \ldots E_{k-1,k}^{\uparrow} \text{ and}$$
$$s_2 = V_1^{\downarrow} V_2^{\downarrow} \ldots V_k^{\downarrow} E_{1,2}^{\downarrow} E_{1,3}^{\downarrow} \ldots E_{k-1,k}^{\downarrow}.$$

Observe that the longest common subsequence of s_1 and s_2 is of length $k + \binom{k}{2}$ and contains exactly one vertex v_i from each set V_i and exactly one edge $e_{i,j}$ from each set $E_{i,j}$.

Now, design a "gadget" consisting of two strings that ensures that v_i is an endpoint of $e_{i,j}$, for each $1 \leq i < j \leq n$. Similarly, verify the second endpoint of $e_{i,j}$.

Bibliographic notes

Downey and Fellows [149, 150, 148] defined the notion of parameterized reductions and recognized that they can be used to transfer fixed-parameter intractability results. The W[1]-hardness of Clique and Independent Set on regular graphs, as well as the fact that this can be used to show that Partial Vertex Cover is W[1]-hard, was observed by Cai [64] and Marx [350]. The introduction of Multicolored Clique and its W[1]-hardness is usually attributed to Fellows, Hermelin, Rosamond, and Vialette [179], but earlier Pietrzak [384] studied the same problem under the name Partitioned Clique.

The W[2]-completeness of Dominating Set on Tournaments was proved by Downey and Fellows [152]; however, the issue of how to construct efficiently a k-paradoxical tournament was glossed over. The probabilistic proof of the existence of small k-paradoxical tournaments, i.e., Theorem 13.12, was given by Erdős [166]. The best known deterministic construction, given by Graham and Spencer [233], achieves size $\mathcal{O}(4^k \cdot k^2)$ for a k-paradoxical tournament and is based on number-theoretical concepts. The combinatorial construction of Exercise 13.5 has been proposed by Tyszkiewicz [422]. The vast majority of parameterized reductions in the literature are actually polynomial-time reductions. The reduction to Dominating Set on Tournaments presented in Theorem 13.14 is one of the rare exceptions: the running time depends superpolynomially on k. Other examples of such reductions can be found in the W[1]-hardness proofs of the Vapnik-Chervonenkis (VC) dimension [152] and certain parameterizations of Closest Substring [349] (see Theorem 14.27 in Section 14.4).

The W-hierarchy was defined by Downey and Fellows [149, 148, 149, 152, 150, 151]. These series of papers prove, among other results, the W[1]-completeness of Independent Set (Theorem 13.18) and the characterization of W[t] by normalized circuit satisfiability (Theorem 13.19). The connection of W[1] and Turing machines was investigated by Cai, Chen, Downey, and Fellows [67]. They show W[1]-hardness of Short Turing Machine Acceptance by a reduction from Clique and show membership in W[1] by reducing Short Turing Machine Acceptance to Weighted Circuit Satisfiability with weft-1 circuits. Together with the W[1]-hardness of Independent Set (Theorem 13.18), this gives a reduction from Short Turing Machine Acceptance to Independent Set. In Section 13.22, we worked out a direct reduction from Short Turing Machine Acceptance to Independent Set to provide self-contained evidence (without the need for the W-hierarchy and Theorem 13.18) why Independent Set is unlikely to be fixed-parameter tractable.

Cesati [72, 71] observed that reduction to Short Turing Machine Acceptance can be a convenient way of proving membership in W[1]. We used this technique for Partial Vertex Cover (Theorem 13.24) and for Unique Hitting Set (Exercise 13.20).

The W[1]-hardness proof of Balanced Vertex Separator is by Marx [347]. The W[1]-hardness proof of List Coloring parameterized by treewidth is by Fellows, Fomin, Lokshtanov, Rosamond, Saurabh, Szeider, and Thomassen [177]. In Section 14.29, we will see a different way of proving W[1]-hardness for List Coloring parameterized by treewidth, which also works for planar graphs.

Downey and Fellows [151] proved the W[1]-hardness of Perfect Code. Downey, Fellows, Vardy, and Whittle [155] used the W[1]-hardness of Perfect Code to show (via complicated reductions) the W[1]-hardness of variants of Odd Set (and of further problems that we do not discuss here). In Section 13.6, we presented a very simple and self-contained W[1]-hardness proof for Odd Set that can be adapted for other variants.

The W[2]-hardness of Dominating Set in graphs excluding $K_{1,4}$ as an induced subgraph has been proved independently by Cygan, Philip, Pilipczuk, Pilipczuk, and Wojtaszczyk [119] and Hermelin, Mnich, van Leeuwen, and Woeginger [259]. It is worth mentioning that, as proved in both of these papers, Dominating Set becomes FPT if restricted to graphs excluding $K_{1,3}$ as an induced subgraph (i.e., in claw-free graphs).

The W[1]-hardness of various edge and vertex versions of Strongly Connected Steiner Subgraph was shown by Guo, Niedermeier, and Suchý [244] and by Chitnis, Hajiaghayi, and Marx [92]. In Section 13.6, we have presented a somewhat simpler proof that works only for the vertex variant.

Vertex deletion problems to d-degenerate graphs (Exercise 13.29) were studied by Mathieson [360]. It appears that they are even harder than as stated in Exercise 13.29.

Given a graph G and an integer k, the Biclique problem asks for a $K_{k,k}$ subgraph, that is, two disjoint sets $X, Y \subseteq V(G)$ of vertices of size exactly k such that every vertex of X is adjacent to every vertex of Y. The fixed-parameter tractability of Biclique was a longstanding open question. Very recently, Lin [322] proved that Biclique is W[1]-hard, resolving this question. (Exercise 13.3 considers the multicolored version of the problem, which can be shown to be W[1]-hard in a much easier way.)

Chapter 14
Lower bounds based on the Exponential-Time Hypothesis

The Exponential Time Hypothesis (ETH) is a conjecture stating that, roughly speaking, n-variable 3-SAT *cannot be solved in time $2^{o(n)}$. In this chapter, we prove lower bounds based on ETH for the time needed to solve various problems. In many cases, these lower bounds match (up to small factors) the running time of the best known algorithms for the problem.*

In the previous chapter, we have learned tools for distinguishing parameterized problems that do admit fixed-parameter tractable algorithms from those that probably do not. However, as we have seen before, FPT algorithms come with a full variety of running times. On one hand, we have very efficient subexponential parameterized algorithms following from bidimensionality (see Section 7.7), with a typical running time of the form $\mathcal{O}^*(2^{\mathcal{O}(\sqrt{k}\cdot\mathrm{polylog}(k))})$[1]. On the other hand we have algorithms derived from Courcelle's theorem, where the dependency on the treewidth of the graph is not even elementary. Somewhere in between lie "standard" fixed-parameter tractable problems amenable to other techniques, like branching, color coding, iterative compression, kernelization, algebraic tools, representative sets, etc. For these problems the typical running time of an FPT algorithm is $\mathcal{O}^*(2^{\mathcal{O}(k)})$, $\mathcal{O}^*(2^{\mathcal{O}(k \log k)})$, $\mathcal{O}^*(2^{\mathrm{poly}(k)})$, or doubly exponential in k. Therefore, it seems that the class FPT has to contain an inner hierarchy of classes, corresponding to respective forms of parameter dependency of the running times of FPT algorithms. In order to uncover and describe this inner hier-

[1] In this chapter we focus on lower bounds on the parametric dependence of the running times of parameterized algorithms. For this reason, we extensively use the $\mathcal{O}^*$-notation that suppresses factors polynomial in the input size; recall that we defined it in Section 1.1.

© Springer International Publishing Switzerland 2015
M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_14

archy, we need to develop a methodology for proving lower bounds: Given a particular problem, we need to establish an asymptotic lower bound on function f that can appear in the running time of any FPT algorithm solving the problem. The assumption that FPT $\neq$ W[1] seems too weak to achieve this goal, and therefore we will introduce stronger complexity assumptions.

The Exponential Time Hypothesis (ETH) is a conjecture stating that, roughly speaking, 3-SAT has no algorithm subexponential in the number of variables. As we will see this conjecture implies that FPT $\neq$ W[1], hence it can be also used to give conditional evidence that certain problems are not fixed-parameter tractable. More importantly, ETH allows us to prove quantitative results of various forms. For example, we can prove results saying that (assuming ETH) a problem cannot be solved in time $2^{o(n)}$, or a parameterized problem cannot be solved in time $f(k)n^{o(k)}$, or a fixed-parameter tractable problem does not admit a $2^{o(k)}n^{\mathcal{O}(1)}$-time algorithm.[2] In many cases, the lower bounds obtained this way match (up to small factors) the best known algorithm, giving a tight understanding of the complexity of the problem. In recent years, the field of parameterized complexity has been completely revolutionized by the fact that such tight lower bounds can be proved. Obtaining tight lower bounds seems to be a reachable goal for many problems and working towards this goal opens up new algorithmic questions to explore. A variant of ETH, called the Strong Exponential Time Hypothesis (SETH) can be used to give even more refined lower bounds saying, e.g., that a parameterized problem cannot be solved in time $(2-\varepsilon)^k n^{\mathcal{O}(1)}$ for any $\varepsilon > 1$.

In Section 14.1 we introduce ETH and SETH, and derive classic consequences of the former. In Section 14.3 we show how ETH can be used to establish lower bounds on the running times of FPT algorithms. Section 14.4 is devoted to the study of problems that are W[1]-hard and hence probably not FPT; it appears that for such problems ETH can be used to provide sharp estimates on the asymptotic behavior of the function of the parameter appearing in the exponent of n. Finally, in Section *14.5 we present selected lower bounds based on SETH.

14.1 The Exponential-Time Hypothesis: motivation and basic results

The starting point of our considerations will be the hardness of the CNF-SAT problem. Recall that in the CNF-SAT problem we are given a propositional formula φ on n Boolean variables $x_1, x_2, \ldots, x_n$ that is in *conjunctive normal form* (*CNF*). This means that $\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, where the C_is, called *clauses*, are of the form $C_i = \ell_1^i \vee \ell_2^i \vee \ldots \vee \ell_{r_i}^i$. Here, the ℓ_j^is are *literals*, that is,

[2] Recall that we say that $f(n) \in o(n)$ if $f(n) \leq \frac{n}{s(n)}$ for some unbounded and nondecreasing function s.

appearances of some variable in a negated or non-negated form. The question is whether there exists an assignment of true/false values to the variables so that φ becomes true. By restricting the number of variables appearing in each clause to some constant q we arrive at the q-SAT problem.

Since for $q \geq 3$ the q-SAT is NP-complete, we do not expect it to be solvable in polynomial time. Our current knowledge, however, is very far from this unattainable goal. Obviously, the general CNF-SAT problem can be solved in time $\mathcal{O}^*(2^n)$ by trying all possible true/false assignments. Apparently, we do not know any algorithm that is substantially faster than this brute-force solution. Some improvement is possible, however, when the length of clauses is restricted: for every $q \geq 3$, there exists $\gamma_q = 1 - \Theta(\frac{1}{q})$ and an algorithm resolving q-SAT in time $\mathcal{O}^*(2^{\gamma_q n})$. For the most studied case of $q = 3$, the current champion achieves $\gamma_3 \leq 0.387$.

The current status of research on satisfiability problems suggests that the following two natural barriers are hard to break:

1. Obtaining a *subexponential* algorithm for 3-SAT, i.e., one with running time $2^{o(n)}$.
2. Finding an algorithm for the general CNF-SAT problem with running time $\mathcal{O}^*((2-\varepsilon)^n)$ for some constant $\varepsilon > 0$.

These two barriers motivate the complexity assumptions that we will introduce in a moment. For $q \geq 3$, let δ_q be the infinimum of the set of constants c for which there exists an algorithm solving q-SAT in time $\mathcal{O}^*(2^{cn})$. In this definition, we allow only deterministic algorithms; we will discuss randomized analogues in what follows. The *Exponential-Time Hypothesis* and *Strong Exponential-Time Hypothesis* are then defined as follows.

Conjecture 14.1 (Exponential-Time Hypothesis, ETH).

$$\delta_3 > 0$$

Conjecture 14.2 (Strong Exponential-Time Hypothesis, SETH).

$$\lim_{q\to\infty} \delta_q = 1$$

Intuitively, ETH states that any algorithm for 3-SAT needs to search through an exponential number of alternatives, while SETH states that as q grows to infinity, brute-force check of all possible assignments becomes more and more inevitable. As we will see later, ETH actually implies that FPT $\neq$ W[1], which supports the intuition that we are introducing stronger assumptions in order to obtain sharper estimates of the complexity of parameterized problems.

Looking back at the motivating barriers, note that ETH implies that 3-SAT cannot be solved in time $2^{o(n)}$, while SETH implies that CNF-SAT cannot be solved in time $\mathcal{O}^*((2-\varepsilon)^n)$ for any $\varepsilon > 0$. However, the converse implications are unclear. It appears that the (possibly slightly stronger) statements of ETH and SETH, as stated in Conjectures 14.1 and 14.2, are much more robust when it comes to inferring their corollaries via reductions. As we will see later, SETH actually implies ETH; see Theorem 14.5.

Let us briefly discuss the possibility of defining the randomized versions of ETH and SETH. In Chapter 5 we have already seen a notion of one-sided error in the case of Monte Carlo algorithms. One can also consider randomized algorithms with two-sided error. Such an algorithm both for any yes-instance and for any no-instance returns an incorrect answer with probability bounded by a constant that is strictly smaller than 1/2. Here, we modify the definition of δ_q by allowing also randomized two-sided error algorithms. Let $\delta_q^\star \leq \delta_q$ be the modified constant for q-SAT. Then randomized ETH and randomized SETH state that $\delta_3^\star > 0$ and $\lim_{q\to\infty} \delta_q^\star = 1$, respectively. Clearly, randomized ETH and SETH imply their deterministic analogues. In principle, whenever the existence of a deterministic algorithm with some running time can be excluded under deterministic ETH, the same chain of reductions usually shows that a randomized algorithm with the same running time is excluded under randomized ETH.

Allowing randomization in the definitions of ETH and SETH seems reasonable from the point of view of the research on satisfiability: the vast majority of fast algorithms for q-SAT are inherently randomized, including the currently fastest $\mathcal{O}(2^{0.386n})$ algorithm for 3-SAT. On the other hand, the assumption of allowing randomization is inconvenient from a pragmatic point of view: dealing with the error probability in multiple reductions that we present in this chapter would lead to more technical and obfuscated proofs. Therefore, we decided to use the deterministic versions of ETH and SETH as our main assumptions, and the reader is asked to verify the steps where randomization creates technical difficulties in Exercises 14.1 and 14.4. Henceforth, whenever we are referring to an algorithm, we mean a deterministic algorithm.

Before we proceed, let us remark that while ETH is generally considered a plausible complexity assumption, SETH is regarded by many as a quite doubtful working hypothesis that can be refuted any time. For this reason, lower bounds proved under the assumption of SETH should not be regarded as supported by very strong arguments, but rather that the existence of better algorithms would constitute a major breakthrough in the complexity of satisfiability.

Most reductions from, say, 3-SAT, output instances of the target problem whose sizes depend on the actual size of the input formula, rather than on the cardinality of its variable set. However, an arbitrary instance of 3-SAT can have up to $\Theta(n^3)$ distinct clauses; as we will see in the next chapter, this size cannot be nontrivially shrunk in polynomial time under plausible complexity assumptions (see Theorem 15.29). Therefore, a reduction that starts from an arbitrary instance of 3-SAT and proceeds directly to constructing the output instance, would inevitably construct an instance of size at least cubic in the number of input variables.

Suppose now that we are trying to prove a lower bound on the running time of algorithms solving the target problem. Then given such a reduction with a cubic blow-up, the best we could hope for is excluding an algorithm for the target problem that on any input instance x achieves running time $2^{o(|x|^{1/3})}$; here, $|x|$ is the size of instance x. Indeed, composing the reduction with such an algorithm would yield a $2^{o(n)}$ algorithm for 3-SAT, contradicting ETH. However, if we could somehow assume that the number of clauses of the input instance was linear in terms of n, then the reduction could give even an $2^{o(|x|)}$ lower bound, providing that the output size is linear in the size of the input formula.

Therefore, it seems natural to try to *sparsify* the input instance of 3-SAT (or more generally q-SAT), in order to be able to assume that the number of clauses is comparable to the number of variables. This is exactly the idea behind the milestone result of this theory, called the sparsification lemma.

Theorem 14.3 (Sparsification lemma, [273]). *For all $\varepsilon > 0$ and positive q, there is a constant $K = K(\varepsilon, q)$ such that any q-CNF formula φ with n variables can be expressed as $\varphi = \bigvee_{i=1}^{t} \psi_i$, where $t \leq 2^{\varepsilon n}$ and each ψ_i is a q-CNF formula with the same variable set as φ and at most Kn clauses. Moreover, this disjunction can be computed by an algorithm running in time $\mathcal{O}^*(2^{\varepsilon n})$.*

Since the proof of the sparsification lemma is not necessary to understand the remainder of this chapter, we omit it in this book and refer the reader to original work [273] for an exposition.

Before we proceed, let us elaborate on what this result actually says. The sparsification lemma provides a reduction that sparsifies the given instance of q-SAT so that its total size is comparable to the size of its variable set. In order to circumvent the aforementioned incompressibility issues for q-SAT, we allow the reduction to use exponential time and output a logical OR of exponentially many instances (i.e., it is a Turing reduction, see Section *13.5), yet these exponential functions can be as small as we like. Thus, whenever reducing from q-SAT for a constant q, we can essentially assume that the number of clauses is bounded linearly in the number of variables.

The following immediate consequence of the sparsification lemma will be our main tool for obtaining further corollaries.

Theorem 14.4. *Unless ETH fails, there exists a constant $c > 0$ such that no algorithm for 3-SAT can achieve running time $\mathcal{O}^*(2^{c(n+m)})$. In particular, 3-SAT cannot be solved in time $2^{o(n+m)}$.*

Proof. Assume that for every $c > 0$ there exists an algorithm $\mathcal{A}_c$ that solves 3-SAT in time $\mathcal{O}^*(2^{c(n+m)})$. We are going to show that for every $d > 0$ there exists an algorithm $\mathcal{B}_d$ that solves 3-SAT in time $\mathcal{O}^*(2^{dn})$, contradicting ETH. Fix any $d > 0$, and let K be the constant given by Theorem 14.3 for $\varepsilon = d/2$ and $q = 3$. Consider an algorithm $\mathcal{B}_d$ for the 3-SAT problem that works as follows:

1. Apply the algorithm of Theorem 14.3 to the input formula φ and parameter $\varepsilon = d/2$.
2. Apply algorithm $\mathcal{A}_{c'}$ for $c' = \frac{d}{2(K+1)}$ to each output formula ψ_i, and return that φ is satisfiable if at least one of the ψ_is is.

The correctness of the algorithm follows directly from Theorem 14.3. By Theorem 14.3, the first step of the algorithm takes time $\mathcal{O}^*(2^{\frac{dn}{2}})$, while the second takes time $\mathcal{O}^*(2^{\frac{dn}{2}} \cdot 2^{\frac{d}{2(K+1)} \cdot (K+1)n}) = \mathcal{O}^*(2^{dn})$, as each formula ψ_i has at most Kn clauses and n variables. Hence, the whole algorithm runs in time $\mathcal{O}^*(2^{dn})$. □

To see another application of the sparsification lemma, we will now show that SETH implies ETH.

Theorem 14.5. *If SETH holds, then so does ETH.*

Proof. For the sake of contradiction, assume that $\delta_3 = 0$. Hence, for every $c > 0$ there exists an algorithm $\mathcal{A}_c$ that solves 3-SAT in time $\mathcal{O}^*(2^{cn})$. We will show that this implies $\delta_q = 0$ for every $q \geq 3$, which contradicts SETH.

Consider the following algorithm for q-SAT. Given the input formula φ, first apply the algorithm of Theorem 14.3 for some $\varepsilon > 0$, to be determined later. This algorithm runs in time $\mathcal{O}^*(2^{\varepsilon n})$ and outputs at most $2^{\varepsilon n}$ formulas ψ_i such that φ is satisfiable if and only if any of the ψ_is is, and each ψ_i has at most $K(\varepsilon, q) \cdot n$ clauses. We now focus on resolving satisfiability of one $\psi = \psi_i$.

Consider the following standard reduction from q-SAT to 3-SAT: as long as there exists some clause $C = \ell_1 \vee \ell_2 \vee \ldots \vee \ell_p$ for some $p \geq 4$, add a fresh variable y and replace C with clauses $C_1 = \ell_1 \vee \ell_2 \vee y$ and $C_2 = \neg y \vee \ell_3 \vee \ldots \vee \ell_p$. Observe that this transformation preserves satisfiability of the formula, and every original clause of length $p > 3$ gives rise to $p - 3$ new variables and $p - 2$ new clauses in the output formula. Hence, if we apply this reduction to the formula ψ, we obtain an equivalent formula ψ' that is in 3-CNF and that has at most $(1 + q \cdot K(\varepsilon, q)) \cdot n$ variables. If we now apply algorithm $\mathcal{A}_\delta$ to ψ', for some $0 < \delta < 1$, then we resolve satisfiability of ψ' in time $\mathcal{O}^*(2^{\delta' n})$ for $\delta' = \delta \cdot (1 + q \cdot K(\varepsilon, q))$. By applying this procedure to each of the listed formulas ψ_i, we resolve satisfiability of φ in time $\mathcal{O}^*(2^{\delta'' n})$ for $\delta'' = \varepsilon + \delta' = \varepsilon + \delta \cdot (1 + q \cdot K(\varepsilon, q))$.

Recall that the constant ε in Theorem 14.3 can be chosen to be arbitrarily close to 0. Since $\delta_3 = 0$, then having chosen $\varepsilon > 0$ (and thus $K(\varepsilon, q)$) we can choose δ so that δ' is arbitrarily close to 0. Therefore, ε and δ can be chosen in such a manner that δ'' is arbitrarily close to 0, which implies that $\delta_q = 0$. □

14.2 ETH and classical complexity

The CNF-SAT and 3-SAT problems lie at the very foundations of the theory of NP-completeness. Problems around satisfiability of propositional formulas were the first problems whose NP-completeness has been settled, and the standard approach to prove NP-hardness of a given problem is to try to find a polynomial-time reduction from 3-SAT, or from some problem whose NP-hardness is already known. Thus, 3-SAT is in some sense the "mother of all NP-hard problems", and the current knowledge about NP-completeness can be viewed as a net of reductions originating precisely at 3-SAT. Therefore, it should not be surprising that by making a stronger assumption about the complexity of 3-SAT, we can infer stronger corollaries about all the problems that can be reached via polynomial time reductions from 3-SAT.

Consider, for instance, a problem A that admits a *linear* reduction from 3-SAT, i.e., a polynomial-time algorithm that takes an instance of 3-SAT on n variables and m clauses, and outputs an equivalent instance of A whose size is bounded by $\mathcal{O}(n+m)$. Then, if A admitted an algorithm with running time $2^{o(|x|)}$, where $|x|$ is the size of the input instance, then composing the reduction with such an algorithm would yield an algorithm for 3-SAT running in time $2^{o(n+m)}$, which contradicts ETH by Theorem 14.4. Actually, many known reductions for classic problems are in fact linear. Examples include Vertex Cover, Dominating Set, Feedback Vertex Set, 3-Coloring, and Hamiltonian Cycle; the reader is asked to verify this fact in Exercise 14.2. As a corollary we infer that none of these problems admits an algorithm running in subexponential time in terms of the instance size. Hence, while all of them can be solved in time $\mathcal{O}^*(2^N)$ for graphs on N vertices and M edges, the existence of algorithms with running time $2^{o(N)}$ or even $2^{o(M)}$ is unlikely.[3]

Theorem 14.6. *Each of the following problems has a linear reduction from* 3-SAT*:* Vertex Cover*,* Dominating Set*,* Feedback Vertex Set*,* 3-Coloring*,* Hamiltonian Cycle*. Therefore, unless ETH fails, none of them admits an algorithm working in time $2^{o(N+M)}$, where N and M are the cardinalities of the vertex and edge sets of the input graph, respectively.*

Of course, if the polynomial-time reduction from 3-SAT to the target problem has a different output size guarantee than linear, then the transferred lower bound on the complexity of the target problem is also different. The following basic observation can be used to transfer lower bounds between problems.

[3] In this chapter, we will consistently use n and m to denote the number of variables and clauses of a formula, respectively, whereas N and M will refer to the number of vertices and edges of a graph.

Observation 14.7. Suppose that there is a polynomial-time reduction from problem A to problem B that, given an instance x of A, constructs an equivalent instance of B having length at most $g(|x|)$ for some nondecreasing function g. Then an $\mathcal{O}^*(2^{o(f(|x|))})$-time algorithm for B for some nondecreasing function f implies an $\mathcal{O}^*(2^{o(f(g(|x|)))})$-time algorithm for A.

Therefore, in order to exclude an algorithm for a problem B with running time $\mathcal{O}^*(2^{o(f(|x|))})$, we need to provide a reduction from 3-SAT to B that outputs instances of size $\mathcal{O}(g(n+m))$, where g is the inverse of f.

Let us now make an example of an application of this framework where f and g are not linear. To this end, we will consider problems on planar graphs such as Planar Vertex Cover, Planar Dominating Set, Planar Feedback Vertex Set, Planar 3-Coloring and Planar Hamiltonian Cycle. The NP-hardness of these problems can be established by the means of a pivot problem called Planar 3-SAT. For an instance φ of 3-SAT, we can consider its *incidence graph* G_φ defined as follows: the vertex set of G_φ contains one vertex per each variable of φ and one vertex per clause of φ, and we put an edge between a variable x and a clause C if and only if x appears in C (either positively or negatively). In the Planar 3-SAT problem we require that the input formula of 3-SAT has a planar incidence graph. The following theorem, which we leave without a proof due to its technicality, provides a lower bound for Planar 3-SAT.

Theorem 14.8 ([320]). *There exists a polynomial-time reduction from* 3-SAT *to* Planar 3-SAT *that, for a given instance φ with n variables and m clauses, outputs an equivalent instance φ' of* Planar 3-SAT *with $\mathcal{O}((n+m)^2)$ variables and clauses. Consequently,* Planar 3-SAT *is* NP*-hard and does not admit an algorithm working in time $2^{o(\sqrt{n+m})}$, unless ETH fails.*

We remark that the reduction of Theorem 14.8 actually outputs a formula φ' and an ordering $(x_1, x_2, \ldots, x_{n'})$ of the variables of φ' such that $G_{\varphi'}$ remains planar even after adding the edges $x_1x_2, x_2x_3, \ldots, x_{n'-1}x_{n'}, x_{n'}x_1$. This property, called in what follows *the existence of a variable circuit*, turns out to be very useful in more complicated reductions.

Roughly speaking, in the proof of Theorem 14.8, one embeds the input formula arbitrarily on the plane, and then replaces each crossing of edges of G_φ by a constant-size *crossover gadget* that transfers the information via the crossing without disturbing planarity. Since the number of edges of G_φ is at most $3m$, the number of crossings is at most $9m^2$ and thus the bound on the size of the output formula follows.

The gist of standard NP-hardness reductions for problems like Vertex Cover, 3-Coloring or Dominating Set is to replace each variable x with a variable gadget H_x, replace each clause C with a clause gadget H_C, and wire the gadgets to simulate the behavior of the input instance of 3-SAT using the properties of the target problem. Due to Theorem 14.8, we can use the same approach for planar problems as well: with a little bit of carefulness, replacing every variable and clause in a planar embedding of G_φ with a respective gadget will not disturb planarity. The reader is asked to verify the following theorem in Exercise 14.3.

Theorem 14.9. *Let A be any of the following problems:* Planar Vertex Cover*,* Planar Dominating Set*,* Planar Feedback Vertex Set*. There exists a polynomial-time reduction that takes an instance φ of* Planar 3-SAT *with n variables and m clauses, and outputs an equivalent instance of A whose graph has $\mathcal{O}(n+m)$ vertices and edges. The same is true if A is* Planar 3-Coloring *or* Planar Hamiltonian Cycle*, and φ has a variable circuit. Consequently, unless ETH fails none of the aforementioned problems admits an algorithm with running time $2^{o(\sqrt{N})}$, where N is the number of vertices of the input graph.*

Note, however, that all the problems considered in Theorem 14.9 can in fact be solved in time $2^{\mathcal{O}(\sqrt{N})}$: this can be easily seen by pipelining the fact that a planar graph on N vertices has treewidth $\mathcal{O}(\sqrt{N})$ (see Corollary 7.24[4]) with a single-exponential dynamic-programming routine on a tree decomposition (see Theorem 7.9 and Theorem 11.13). Thus, we have obtained essentially matching upper and lower bounds on the complexity: achieving $2^{\mathcal{O}(\sqrt{N})}$ is possible, and it is hard to break this barrier.

14.3 ETH and fixed-parameter tractable problems

So far we have seen how ETH can be used to prove lower bounds on the classic complexity of NP-hard problems, i.e., complexity measured only in terms of the size of the input instance. We now proceed to the multivariate world: we shall present applications of ETH to parameterized problems, and in particular to problems that are fixed-parameter tractable. It turns out that ETH is a very convenient tool for proving lower bounds on the parameter dependence of the running time of FPT algorithms. In many cases these lower bounds (almost) match known algorithmic upper bounds.

[4] Note that Corollary 7.24 not only proves an upper bound on treewidth, but also gives a polynomial-time algorithm which provides a desired decomposition.

14.3.1 Immediate consequences for parameterized complexity

Observe that ETH and SETH can be seen as assumptions about the complexity of the q-SAT problems equipped with parameterization by the number of variables n. The sparsification lemma serves then as a reduction from parameterization by n to parameterization by $n+m$, and Theorem 14.4 establishes a lower bound for the latter. Therefore, if we consider 3-SAT as a source of parameterized reductions rather than classic ones, then we can infer lower bounds on the parameterized complexity of problems, instead of the standard instance-size complexity.

Suppose, for instance, that a parameterized problem A admits a *linear parameterized* reduction from 3-SAT: a polynomial-time algorithm that takes an instance of 3-SAT on n variables and m clauses, and outputs an equivalent instance of A with *parameter* k bounded by $\mathcal{O}(n+m)$. If A admitted an FPT algorithm with running time $\mathcal{O}^*(2^{o(k)})$, then we could compose the reduction with this algorithm and thus solve 3-SAT in time $2^{o(n+m)}$, contradicting ETH by Theorem 14.4. Note here that we bound *only* the output parameter, while the whole output instance may have superlinear size.

Again, for many classic FPT problems on general graphs, like Vertex Cover or Feedback Vertex Set, the known NP-hardness reductions in fact yield linear parameterized reductions from 3-SAT. In the case of these two particular problems this conclusion is trivial: we already know that Vertex Cover and Feedback Vertex Set admit classical linear reductions from 3-SAT, and parameters in meaningful instances are bounded by the cardinality of the vertex set. This shows that both these problems, and any others that admit linear parameterized reductions from 3-SAT, probably do not admit *subexponential parameterized algorithms*, that is, FPT algorithms with running time $\mathcal{O}^*(2^{o(k)})$.

Similarly as before, we may prove lower bounds for different forms of FPT running time by having different guarantees on the output parameter. This is encapsulated in the following analogue of Observation 14.7.

Observation 14.10. Suppose that there is a polynomial-time parameterized reduction from problem A to problem B such that if the parameter of an instance of A is k, then the parameter of the constructed instance of B is at most $g(k)$ for some nondecreasing function g. Then an $\mathcal{O}^*(2^{o(f(k))})$-time algorithm for B for some nondecreasing function f implies an $\mathcal{O}^*(2^{o(f(g(k)))})$ algorithm for A.

Therefore, in order to exclude an algorithm for a parameterized problem B with running time $\mathcal{O}^*(2^{o(f(k))})$, we need to provide a reduction from 3-SAT to B that outputs instances with the parameter k bounded by $\mathcal{O}(g(n+m))$, where g is the inverse of f.

Observe that formally a nondecreasing function f might not have an inverse. For that reason throughout the chapter we slightly overuse the definition of an inverse function, and for a nondecreasing function f its inverse is any function g satisfying $g(f(x)) = \Theta(x)$.

Again, to see how this framework works in practice, consider the examples of Planar Vertex Cover, Planar Dominating Set, and Planar Feedback Vertex Set problems. By composing Theorems 14.8 and 14.9 we obtain NP-hardness reductions for these problems that start with a 3-SAT instance with n variables and m clauses, and output instances of the target problem with at most $\mathcal{O}((n+m)^2)$ vertices. Obviously, we can assume that the requested size of vertex cover/dominating set/feedback vertex set in the output instance is also bounded by $\mathcal{O}((n+m)^2)$. Thus, following the notation of Observation 14.10, these reductions can serve as parameterized reductions from 3-SAT for a quadratic function g. The following corollary is immediate.

Theorem 14.11. *Unless ETH fails, none of the following problems admits an FPT algorithm running in time $\mathcal{O}^*(2^{o(\sqrt{k})})$:* Planar Vertex Cover*,* Planar Dominating Set*,* Planar Feedback Vertex Set*.*

Recall that in Chapter 7 we have given algorithms for all the problems mentioned in Theorem 14.11 that run in time $\mathcal{O}^*(2^{\mathcal{O}(\sqrt{k})})$ (in the cases of Planar Vertex Cover and Planar Dominating Set) or in time $\mathcal{O}^*(2^{\mathcal{O}(\sqrt{k}\log k)})$ (in the case of Planar Feedback Vertex Set), using the technique of bidimensionality (see Corollaries 7.30 and 7.31). In fact, by using the faster dynamic-programming routine for Feedback Vertex Set from Theorem 11.13, we can solve Planar Feedback Vertex Set also in time $\mathcal{O}^*(2^{\mathcal{O}(\sqrt{k})})$. Therefore, the lower bounds of Theorem 14.11 essentially match the upper bounds for all the considered problems. This shows that appearance of the square root in the exponent is not just an artifact of the technique of bidimensionality, but rather an intrinsic property of the problems' complexity.

**14.3.2 Slightly super-exponential parameterized complexity*

One class of problems, for which we have some understanding of the methodology of proving sharp lower bounds, is the problems solvable in *slightly super-exponential* parameterized time, i.e., in time $\mathcal{O}^*(2^{\mathcal{O}(k\log k)})$ for the parameter k. The running time of such a form appears naturally in the following situations:

(i) A seemingly optimal FPT algorithm is a branching procedure that performs $\mathcal{O}(k)$ guesses, each time choosing from a set of possibilities of cardinality $k^{\mathcal{O}(1)}$. An example of such algorithm is the one given for Closest String in Section 3.5.
(ii) The parameter is the treewidth t of a given graph, and the natural dynamic-programming routine has $2^{\mathcal{O}(t \log t)}$ states per bag of the tree decomposition, usually formed by partitions of the bag. An example of such an algorithm is the one given by Theorem 7.8 for the Steiner Tree problem.

By excluding the existence of an $\mathcal{O}^*(2^{o(k \log k)})$ algorithm for the problem under ETH, we can support the claim that the obtained slightly super-exponential upper bound is not just a result of the approach taken, but such running time is necessary because of the nature of the problem itself.

We now aim at providing a base set of auxiliary problems that will be helpful for further reductions. At the end of this section we will see some examples of how they can be utilized for proving lower bounds for 'real-life' parameterized problems, say of type (i) or (ii) from the list above. The following problem will be our 'canonical' example where slightly super-exponential running time is both achievable and optimal under ETH. Intuitively, it reflects a generic situation with which we are faced when approaching a parameterized problem of type (i).

In the $k \times k$ Clique problem we are given a graph H on a vertex set $[k] \times [k]$. In other words, the vertex set is formed by a $k \times k$ table, and vertices are of form (i, j) for $1 \leq i, j \leq k$. The question is whether there exists a clique X in H that contains exactly one vertex from each row of the table, i.e., for each $i \in [k]$ there is exactly one element of X that has i on the first coordinate. Note that without loss of generality we may assume that each row of the table forms an independent set.

Obviously, $k \times k$ Clique can be solved in time $\mathcal{O}^*(k^k)$ by trying all possible choices of vertices from the consecutive rows. It appears that this running time is essentially optimal assuming ETH, which is proved in the following theorem. In the proof, we will use the fact that the 3-Coloring problem admits a linear reduction from 3-SAT, and hence does not admit a subexponential algorithm in terms of $N + M$, unless ETH fails; see Theorem 14.6 and Exercise 14.2.

Theorem 14.12. *Unless ETH fails, $k \times k$ Clique cannot be solved in time $\mathcal{O}^*(2^{o(k \log k)})$.*

Proof. We shall present a polynomial-time reduction that takes a graph G on N vertices, and outputs an instance (H, k) of $k \times k$ Clique such that (i) (H, k) is a yes-instance if and only if G is 3-colorable, and (ii) $k = \mathcal{O}(N/\log N)$. Composing this reduction with an $\mathcal{O}^*(2^{o(k \log k)})$ algorithm for $k \times k$ Clique would yield an algorithm for 3-Coloring working in time $2^{o(N)}$, which would contradict ETH by Theorem 14.6.

Let $k = \lceil \frac{2N}{\log_3 N} \rceil$. Arbitrarily partition the vertex set of the input graph into k parts $V_1, V_2, \ldots, V_k$, so that each part V_i has cardinality at most $\lceil \frac{\log_3 N}{2} \rceil$. For every part V_i, list all the possible 3-colorings of $G[V_i]$; note that there are at most $3^{|V_i|} \leq 3^{\lceil \frac{\log_3 N}{2} \rceil} \leq 3\sqrt{N}$ such colorings. If no coloring has been found for some part, we conclude that we are dealing with a no-instance. Observe also that without loss of generality we may assume that $3\sqrt{N} \leq k$, since otherwise N is bounded by a universal constant and we may solve the input instance by brute-force.

Duplicate some of the listed colorings if necessary, so that for each part V_i we have a list of exactly k 3-colorings. Let $c_i^1, c_i^2, \ldots, c_i^k$ be the 3-colorings listed for V_i. For each $i \in [k]$ and $j \in [k]$, create a vertex (i, j). For every pair of vertices (i_1, j_1) and (i_2, j_2) make them adjacent if and only if $i_1 \neq i_2$ and $c_{i_1}^{j_1} \cup c_{i_2}^{j_2}$ is a valid 3-coloring on $G[V_{i_1} \cup V_{i_2}]$. This concludes the construction of the graph H, and of the output instance (H, k) of $k \times k$ Clique.

We now argue equivalence of the instances. If c is a valid 3-coloring of G, then $c|_{V_i}$ is a valid 3-coloring of $G[V_i]$. Hence, for each $i \in [k]$, there exists an index $j_i \in [k]$ such that $c_{j_i} = c|_{V_i}$. By the construction, it follows that vertices (i, j_i) for $i \in [k]$ form a k-clique in H: for each $i_1, i_2 \in [k]$, $i_1 \neq i_2$, we have that $c_{i_1}^{j_{i_1}} \cup c_{i_2}^{j_{i_2}}$ is equal to $c|_{V_{i_1} \cup V_{i_2}}$, and hence is a valid 3-coloring of $G[V_{i_1} \cup V_{i_2}]$.

Conversely, assume that a set $\{(i, j_i) \ : \ i \in [k]\}$ induces a k-clique in H. We claim that $c = \bigcup_{i \in [k]} c_i^{j_i}$ is a valid 3-coloring of G. Take any $uv \in E(G)$; we prove that $c(u) \neq c(v)$. If $u, v \in V_i$ for some $i \in [k]$, then we have $c(u) \neq c(v)$ since $c_i^{j_i} = c|_{V_i}$ is a valid 3-coloring of $G[V_i]$. Suppose then that $u \in V_{i_1}$ and $v \in V_{i_2}$ for $i_1 \neq i_2$. Since (i_1, j_{i_1}) and (i_2, j_{i_2}) are adjacent in H, we have that $c_{i_1}^{j_{i_1}} \cup c_{i_2}^{j_{i_2}}$ is a valid 3-coloring of $G[V_{i_1} \cup V_{i_2}]$, and hence $c(u) = c_{i_1}^{j_{i_1}}(u) \neq c_{i_2}^{j_{i_2}}(v) = c(v)$. □

In the $k \times k$ Clique problem the constraint is that each row of the table has to contain one vertex of the clique. It is natural to ask about the more robust version of this problem, where we additionally require that each column of the table also needs to contain one vertex of the clique, or, equivalently, that the vertices of the clique must induce a permutation of $[k]$. We call this problem $k \times k$ Permutation Clique. The following theorem provides a randomized lower bound for this version.

Theorem 14.13. *Assume there exists an algorithm solving $k \times k$ Permutation Clique in time $\mathcal{O}^*(2^{o(k \log k)})$. Then there exists a randomized algorithm solving $k \times k$ Clique in time $\mathcal{O}^*(2^{o(k \log k)})$. The algorithm can have only false negatives, that is, on a no-instance it always gives a negative answer, whereas on a yes-instance it gives a positive answer with probability at least $\frac{1}{2}$.*

Proof. Suppose that we are given an algorithm for $k \times k$ PERMUTATION CLIQUE that works in time $\mathcal{O}^*(2^{o(k \log k)})$. We are going to present a randomized algorithm for $k \times k$ CLIQUE with the stated specification.

Let (H, k) be the input instance of $k \times k$ CLIQUE. Consider the following algorithm: for each row i of the table, uniformly and independently at random pick a permutation π_i of $[k]$, and permute the vertices of this row according to π_i keeping the original adjacencies. More precisely, if we denote by H' the obtained graph, then $V(H') = [k] \times [k]$ and vertices (i, j) and (i', j') are adjacent in H' if and only if vertices $(i, \pi_i^{-1}(j))$ and $(i', \pi_{i'}^{-1}(j'))$ are adjacent in H. Apply the supposed algorithm for $k \times k$ PERMUTATION CLIQUE to the instance (H', k); if this algorithm has found a solution to (H', k), then return a positive answer, and otherwise provide a negative answer.

Let π be the permutation of $[k] \times [k]$ that defines the graph H', that is, $\pi(i, j) = (i, \pi_i(j))$. Obviously, if a solution X' in the instance (H', k) of $k \times k$ PERMUTATION CLIQUE has been found, then in particular $\pi^{-1}(X')$ is a solution to the input instance (H, k) of $k \times k$ CLIQUE. Therefore, if the algorithm returns a positive answer, then it is always correct. Assume now that the input instance (H, k) admits some solution X; we would like to show that with high probability the shuffled instance (H', k) admits a clique that induces a permutation of $[k]$. Let $X' = \pi(X)$. Obviously, X' is still a clique in H' and it contains one vertex from each row; we claim that with high enough probability it also contains one vertex from each column.

Let f_X and $f_{X'}$ be functions from $[k]$ to $[k]$ such that $(i, f_X(i)) \in X$ and $(i, f_{X'}(i)) \in X'$ for each $i \in [k]$. We have that $f_{X'}(i) = \pi_i(f_X(i))$ for each $i \in [k]$, and we need to provide a lower bound on the probability that $f_{X'}$ is a permutation. However, by the choice of π, for every index $i \in [k]$ the value of $\pi_i(f_X(i))$ is chosen uniformly and independently at random among the elements of $[k]$. Hence, every function from $[k]^{[k]}$ is equally probable as $f_{X'}$. Since there are $k!$ permutations of $[k]$, and k^k functions in $[k]^{[k]}$ in total, we infer that

$$\Pr(f_{X'} \text{ is a permutation}) = \frac{k!}{k^k} > e^{-k}.$$

The last inequality follows from the known fact that $k! > (k/e)^k$; recall that we used this inequality in the same manner in the proof of Lemma 5.4. Hence, with probability at least e^{-k} the instance (H', k) will be a yes-instance of $k \times k$ PERMUTATION CLIQUE, and the algorithm will provide a positive answer.

Using the standard technique of independent runs (see Chapter 5), we repeat the algorithm e^k times in order to reduce the error probability from at most $1 - e^{-k}$ to at most $(1 - e^{-k})^{e^k} \leq e^{-e^{-k} \cdot e^k} = 1/e < 1/2$. Note that thus the whole procedure runs in time $e^k \cdot \mathcal{O}^*(2^{o(k \log k)}) = \mathcal{O}^*(2^{o(k \log k)})$. □

The presented proof of Theorem 14.13 gives a randomized Turing reduction from $k \times k$ CLIQUE to $k \times k$ PERMUTATION CLIQUE. It is therefore easy to see that the existence of an $\mathcal{O}^*(2^{o(k \log k)})$ algorithm for $k \times k$ PERMUTATION CLIQUE can be refuted under the stronger assumption of randomized

ETH; the reader is asked to verify this fact in Exercise 14.4. However, since the reduction of Theorem 14.13 is randomized, we cannot a priori state the same lower bound under deterministic ETH. Fortunately, the construction can be derandomized using perfect hash families, and thus we may obtain a deterministic Turing reduction. Hence, we have in fact the following lower bound; we omit its proof due to technicality.

Theorem 14.14 ([326]). *Unless ETH fails,* $k \times k$ PERMUTATION CLIQUE *cannot be solved in time* $\mathcal{O}^*(2^{o(k \log k)})$.

Let us now define the $k \times k$ HITTING SET problem: we are given a family $\mathcal{F}$ of subsets of $[k] \times [k]$, and we would like to find a set X, consisting of one vertex from each row, such that $X \cap F \neq \emptyset$ for each $F \in \mathcal{F}$. Again, in the permutation variant we ask for a set X that induces a permutation of $[k]$. Hardness for $k \times k$ (PERMUTATION) HITTING SET follows from an easy reduction from $k \times k$ (PERMUTATION) CLIQUE.

Theorem 14.15. *Unless ETH fails,* $k \times k$ (PERMUTATION) HITTING SET *cannot be solved in time* $\mathcal{O}^*(2^{o(k \log k)})$.

Proof. Let (H, k) be an instance of $k \times k$ (PERMUTATION) CLIQUE. Construct an instance $(\mathcal{F}, k)$ of $k \times k$ (PERMUTATION) HITTING SET as follows: for each pair of nonadjacent vertices (i_1, j_1) and (i_2, j_2) where $i_1 \neq i_2$, introduce a set $F \in \mathcal{F}$ that consists of all the vertices in rows i_1 and i_2 apart from vertices (i_1, j_1) and (i_2, j_2). Thus, any set X that contains one vertex from each row has a nonempty intersection with F if and only if both pairs (i_1, j_1) and (i_2, j_2) are not included in X simultaneously. It follows that any such X is a solution to the instance $(\mathcal{F}, k)$ of $k \times k$ (PERMUTATION) HITTING SET if and only if it is a solution to the instance (H, k) of $k \times k$ (PERMUTATION) CLIQUE, and so the instances (H, k) and $(\mathcal{F}, k)$ are equivalent. The theorem follows from an application of Theorems 14.12 and 14.14. □

It turns out that for further reductions it is most convenient to have one more assumption about the $k \times k$ (PERMUTATION) HITTING SET problem. Namely, we say that an instance $(\mathcal{F}, k)$ of one of these problems uses only *thin sets* if each $F \in \mathcal{F}$ contains at most one vertex from each row. Note that the instance obtained in the reduction of Theorem 14.15 does not have this property; however, hardness can be also obtained under this assumption.

Theorem 14.16 ([326]). *Unless ETH fails,* $k \times k$ (PERMUTATION) HITTING SET WITH THIN SETS *cannot be solved in time* $\mathcal{O}^*(2^{o(k \log k)})$.

The proof of Theorem 14.16 is a chain of three technical reductions, and we give it as Exercises 14.5, 14.6 and 14.7.

As already mentioned in the beginning of this section, the presented problems form a convenient base for further reductions. We now provide one such example. The problem of our interest will be the CLOSEST STRING problem,

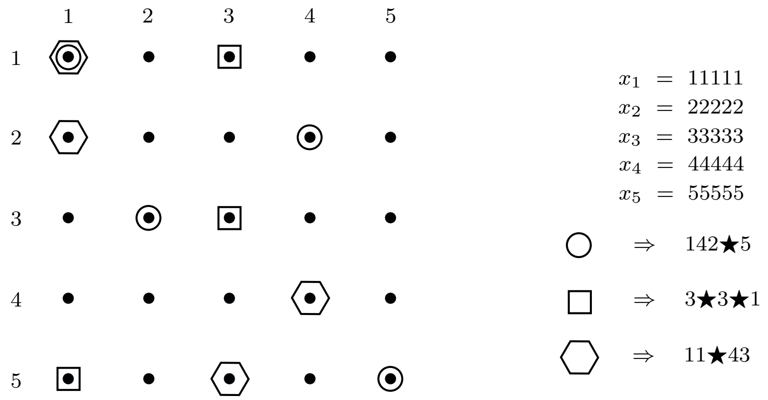


Fig. 14.1: Example of the reduction of Theorem 14.17 applied to an instance of 5 × 5 Permutation Hitting Set with thin sets that has three sets, with each set containing at most one element from each row. The elements of these sets are denoted by circles, squares, and hexagons, respectively

studied in Section 3.5. In this problem, we are given a finite alphabet Σ, a set of strings $\{x_1, x_2, \ldots, x_n\}$ over Σ, each of length L, and an integer d. The question is whether there exists a string $y \in \Sigma^L$ such that x_i and y differ on at most d positions, for each i. In Section 3.5 we have shown how to solve Closest String in time $\mathcal{O}^*((d+1)^d)$ whereas Exercise 3.25 asked for an analysis of an $\mathcal{O}^*(c^d|\Sigma|^d)$-time algorithm for some constant c. We can now show that both these running times are essentially optimal.

Theorem 14.17. *Unless ETH fails,* Closest String *cannot be solved in* $\mathcal{O}^*(2^{o(d \log d)})$ *nor* $\mathcal{O}^*(2^{o(d \log |\Sigma|)})$ *time.*

Proof. We provide a polynomial-time algorithm that takes an instance $(\mathcal{F}, k)$ of $k \times k$ Permutation Hitting Set with thin sets, and outputs an equivalent instance $(\Sigma, d, \{x_1, x_2, \ldots, x_n\})$ of Closest String with $L = k$, $d = k-1$ and $|\Sigma| = k+1$. If there existed an algorithm for Closest String with running time $\mathcal{O}^*(2^{o(d \log d)})$ or $\mathcal{O}^*(2^{o(d \log |\Sigma|)})$, then pipelining the reduction with this algorithm would give an algorithm for $k \times k$ Permutation Hitting Set with thin sets with running time $\mathcal{O}^*(2^{o(k \log k)})$, contradicting ETH by Theorem 14.16.

Let $\Sigma = [k] \cup \{\bigstar\}$ for some additional symbol $\bigstar$ that the reader may view as 'mismatch'. First, introduce k strings $x_1, x_2, \ldots, x_k$ such that $x_i = i^k$, i.e., symbol i repeated k times. Then, for every $F \in \mathcal{F}$ introduce a string z_F constructed as follows. For each $i \in [k]$, if F contains some pair (i, j_i) in row i, then put j_i on the i-th position of z_F. Otherwise, if no such pair exists, put $\bigstar$ on the i-th position of z_F. Note that this definition is valid due to all the sets being thin. Finally, we set $d = k - 1$ and conclude the construction; see Fig. 14.1 for a small example.

We now formally argue that instances $(\mathcal{F}, k)$ and $(\Sigma, k-1, \{x_i : i \in [k]\} \cup \{z_F : F \in \mathcal{F}\})$ are equivalent. Assume first that X is a solution to the instance $(\mathcal{F}, k)$. For each $i \in [k]$, let $j_i \in [k]$ be the unique index such that $(i, j_i) \in X$. Construct a string y by putting j_i on the i-th position of y for each $i \in [k]$. Note that since X induces a permutation of $[k]$, then for every $j \in [k]$ there exists $i \in [k]$ such that $j = j_i$, and so y and x_j differ on at most $k-1$ positions. Moreover, since $X \cap F$ is nonempty for each set $F \in \mathcal{F}$, we have that y coincides with z_F on at least one position, so it differs on at most $k-1$ positions. Hence y is a solution to the constructed instance of Closest String.

Conversely, take any solution y of the constructed Closest String instance. For each j, we have that x_j and y differ on at most $k-1$ positions, so they coincide on at least one. Therefore, y needs to contain every $j \in [k]$ at least once. Since $|y| = k$, then we infer that y contains every $j \in [k]$ exactly once, and it does not contain any $\bigstar$. Let $X = \{(i, y[i]) : i \in [k]\}$. We already know that $X \subseteq [k] \times [k]$ and that X induces a permutation of $[k]$. We are left with proving that X has a nonempty intersection with every $F \in \mathcal{F}$. We know that y and z_F coincide on at least one position, say i. As y does not contain any $\bigstar$, we have that $z_F[i] = y[i] = j_i$ for some $j_i \in [k]$. Consequently $(i, j_i) \in X$ and $(i, j_i) \in F$ by the definitions of X and z_F, so $X \cap F \neq \emptyset$. $\square$

Another important problem that roughly fits into the same category as Closest String is Distortion, defined as follows. We are given a graph G and an integer d. The task is to find an embedding η of the vertices of G into $\mathbb{Z}$ such that $\mathrm{dist}(u, v) \leq |\eta(u) - \eta(v)| \leq d \cdot \mathrm{dist}(u, v)$ for all $u, v \in V(G)$, where $\mathrm{dist}(u, v)$ is the distance between u and v in G. In other words, one is asked to embed the graph metric into the linear structure of $\mathbb{Z}$, so that the distances are stretched at most d times. The problem of embedding various (graph) metrics into low-dimensional vector spaces is the subject of intensive studies both from a purely mathematical, and from a computational point of view; Distortion is thus the most basic and fundamental problem in this theory. Surprisingly, Distortion is fixed-parameter tractable when parameterized by the stretch d, and it admits an $\mathcal{O}^*(d^d)$ algorithm. Using the methodology introduced in this section one can prove that this is optimal under ETH; we omit the proof.

Theorem 14.18 ([176, 326]). Distortion *can be solved in time* $\mathcal{O}^*(d^d)$. *Moreover, unless ETH fails there is no* $\mathcal{O}^*(2^{o(d \log d)})$ *algorithm for this problem.*

Recall that another class of problems where the complexity of the form $\mathcal{O}^*(2^{\mathcal{O}(t \log t)})$ appears naturally, are treewidth t parameterizations where the natural space of states of the dynamic-programming routine is formed by partitions of the bag. In Chapter 7, we have presented one such dynamic program running in time $\mathcal{O}^*(2^{\mathcal{O}(t \log t)})$, for the Steiner Tree problem. While in Chapter 11 we have seen that Steiner Tree can be solved faster, in time

$\mathcal{O}^*(2^{\mathcal{O}(t)})$, for some other problems the presented framework can be used to show that the running time of $\mathcal{O}^*(2^{\mathcal{O}(t \log t)})$ is essentially optimal. The next theorem, which we leave without a proof, provides some examples.

Theorem 14.19 ([118, 326]). *Unless ETH fails, the following problems cannot be solved in time $\mathcal{O}^*(2^{o(t \log t)})$, where t is the width of a given path decomposition of the graph:* Cycle Packing, Vertex Disjoint Paths.

Recall that in the Cycle Packing problem we want to decide whether a given graph has k vertex-disjoint cycles, while in Vertex Disjoint Paths given k pairs of vertices the question is whether one can connect those pairs by vertex-disjoint paths. Note that Theorem 14.19 is stated in terms of a path decomposition and not a tree decomposition, which makes the lower bound stronger.

The idea behind the proof of Theorem 14.19 is to reduce from $k \times k$ (Permutation) Hitting Set with thin sets and try to embed the search space of this problem, which has size roughly k^k, into the states of the standard dynamic program working on a path decomposition. We create a graph which has a long path decomposition of width $\mathcal{O}(k)$ and using problem-dependent combinatorics we make sure that the choice the solution makes on the first bag (for instance, a choice of a matching for the Cycle Packing problem) is propagated along the decomposition to all the other bags. This propagated choice has to reflect the intended solution of the input $k \times k$ (Permutation) Hitting Set with thin sets instance. Then we attach multiple gadgets along the path decomposition, one for every set to be hit. Each gadget should have $\mathcal{O}(k)$ pathwidth, and its role is to verify that the propagated solution from the first bag indeed hits the corresponding set. Thus, all the gadgets are satisfied if and only if the chosen and propagated solution hits all the sets. We will look more closely on reductions of this type in Section 14.5.2, where we discuss optimality of bounded-treewidth dynamic programming under SETH.

In essence, the outcoming message is that the nature of problems exemplified by those mentioned in Theorem 14.19 makes it *necessary* to have $2^{\Omega(t \log t)}$ states per bag of the decomposition. The space of states cannot be pruned substantially, since each state is, roughly speaking, meaningful on its own.

**14.3.3 Double exponential parameterized complexity*

Recall that in Section 2.2.3 we have considered the Edge Clique Cover problem: given a graph G and an integer k, verify whether G contains k subgraphs $C_1, C_2, \ldots, C_k$ such that each C_i is complete and $\bigcup_{i=1}^{k} E(C_i) =$

$E(G)$. Already very simple reduction rules yield a kernel for Edge Clique Cover with at most 2^k vertices, and the problem can be solved in time $\mathcal{O}^*(2^{2^{\mathcal{O}(k)}})$ by performing Set Cover-like dynamic programming on the edge set of the kernel, similar to the one of Section 6.1. In the light of our lower bound methodology, it is natural to ask whether such running time is optimal under ETH. This appears to be the case.

Theorem 14.20 ([120]). *There exists a polynomial-time algorithm that, given a* 3-SAT *instance φ on n variables and m clauses, constructs an equivalent* Edge Clique Cover *instance (G, k) with $k = \mathcal{O}(\log n)$ and $|V(G)| = \mathcal{O}(n + m)$. Consequently,* Edge Clique Cover *cannot be solved in time $\mathcal{O}^*(2^{2^{o(k)}})$ unless ETH fails.*

The proof of Theorem 14.20 is a technical reduction that we omit in this book. Again, Theorem 14.20 shows that, despite the simplicity of the kernelization approach, nothing substantially better can be achieved. The reason for this lies not in the naivety of the technique, but in the problem itself.

14.4 ETH and W[1]-hard problems

In Chapter 13, we have seen techniques for giving evidence that a problem is not fixed-parameter tractable, that is, the parameter k has to appear in the exponent of the running time. But we have not explored the question of how exactly the exponent has to depend on k for a W[1]-hard problem: for all we know, it is possible that a W[1]-hard problem has a $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(\log\log\log k)}$-time algorithm, which would be "morally equivalent" to the problem being FPT. In Section 14.3, we have seen how ETH can be used to classify FPT problems by giving (often tight) lower bounds on the function $f(k)$ appearing in the running time. It turns out that ETH also allows us to prove (often tight) lower bounds on the asymptotic behavior of the function g in an $f(k)n^{g(k)}$ time algorithm. The first such lower bound result we present is for Clique (or, equivalently, for Independent Set).

Theorem 14.21. *Assuming ETH, there is no $f(k)n^{o(k)}$-time algorithm for* Clique *or* Independent Set *for any computable function f.*

Proof. We will show that if there is an $f(k)n^{o(k)}$-time algorithm for Clique for some computable function f, then ETH fails. Suppose that Clique on every graph H can be solved in time $f(k)|V(H)|^{k/s(k)}$, where $s(k)$ is a nondecreasing unbounded function. We use this algorithm to solve 3-Coloring on an n-vertex graph G in time $2^{o(n)}$; by Theorem 14.6, this contradicts ETH. The reduction is similar to the proof of Theorem 14.12, but instead of splitting the vertices into $k = \mathcal{O}(n/\log n)$ groups, the number k of groups now depends on the function f.

We split the vertices in the 3-Coloring instance into roughly $k = f^{-1}(n)$ groups, where f^{-1} is the inverse of f (e.g., if $f(k) = 2^{\Theta(k)}$, then $f^{-1}(n) = \Theta(\log n)$). We create a Clique instance where each vertex represents a 3-coloring of one of the groups. The created graph H has size at most $k \cdot 3^{n/k} = k \cdot 3^{n/f^{-1}(n)} = 2^{o(n)}$. We can observe that the running time of the assumed $f(k)|V(H)|^{o(k)}$-time algorithm for Clique on H is $2^{o(n)}$: we have $f(k) = f(f^{-1}(n)) \leq n$ and $|V(H)|^{o(k)}$ is roughly $(3^{n/k})^{o(k)} = 2^{o(n)}$.

We need to be very careful when defining the number k of groups we need; one reason is that we have no assumption on the time needed to compute $f(k)$, besides the fact that it is computable. We may assume that the computable function f satisfies $f(k) \geq \max\{k, k^{k/s(1)}\}$ for every $k \geq 1$ (e.g., by replacing it with the computable function $f'(k) = \max\{f(k), k, k^{k/s(1)}\}$ if necessary). Let us start computing the values $f(k)$ for $k = 1, 2, \ldots$; we stop this computation when either a k with $f(k) > n$ is reached, or when the total number of steps used so far exceeds n. Let k be the last value for which we computed $f(k)$ and we have $f(k) \leq n$ (we may assume that $f(1) \leq n$ and we can compute $f(1)$ in n steps, otherwise n is bounded by a universal constant). The value k selected this way is a function of n only, that is, we have computed the value $k = g(n)$ for some computable function $g(n)$. Moreover, computing $k = g(n)$ took only time $\mathcal{O}(n)$. Note that from the way function g is defined it follows that it is nondecreasing, unbounded, and $f(g(n)) \leq n$. Moreover, the assumption $f(k) \geq k$ and the inequality $f(g(n)) \leq n$ imply that $g(n) \leq n$.

Split the vertices of G into k groups $V_1, \ldots, V_k$ of size at most $\lceil n/k \rceil$ each. Let us build a graph H where each vertex corresponds to a proper 3-coloring of one of the groups. The number of vertices in H is at most

$$|V(H)| \leq k \cdot 3^{\lceil n/k \rceil} \leq k \cdot 3^{2(n/k)} = k \cdot 3^{2(n/g(n))} = 2^{o(n)}$$

(we used that $k = g(n) \leq n$ in the second inequality and the fact that $g(n)$ is nondecreasing and unbounded in the last equality). We connect two vertices of H if they are not conflicting. That is, if u represents a coloring of $G[V_i]$ and v represents a coloring of $G[V_j]$ for some $i \neq j$, then u and v are adjacent only if the union of the colorings corresponding to u and v is a valid coloring of $G[V_i \cup V_j]$. It is easy to verify that a k-clique of H corresponds to a proper 3-coloring of G and one can construct this coloring in time polynomial in the size of H, that is, in time $2^{o(n)}$. Therefore, using the assumed algorithm for Clique, a 3-coloring of G can be found in time

$$\begin{aligned}
f(k)|V(H)|^{k/s(k)} &\le n(k \cdot 3^{\lceil n/k \rceil})^{k/s(k)} && \text{(using } f(k) = f(g(n)) \le n\text{)}\\
&\le n(k \cdot 3^{2n/k})^{k/s(k)} && \text{(using } k = g(n) \le n\text{)}\\
&\le n \cdot k^{k/s(1)} \cdot 3^{2n/s(g(n))} && \text{(}s(k) \text{ is nondecreasing)}\\
&\le n^2 \cdot 3^{2n/s(g(n))} && \text{(using } k^{k/s(1)} \le f(k) \le n\text{)}\\
&= 2^{o(n)}. && \text{(function } s(g(n)) \text{ is}\\
& && \text{nondecreasing and unbounded)}
\end{aligned}$$

Therefore, we have shown that the assumed algorithm would contradict ETH. □

In particular, Theorem 14.21 shows that, assuming ETH, Clique is not FPT. Therefore, Theorem 14.21 shows that ETH implies that FPT $\neq$ W[1].

Having established a tight lower bound for Clique, we can easily transfer it to other problems with parameterized reductions. As in Observation 14.10, the strength of the lower bound on the target problem depends on the bound on the new parameter in the reduction.

Observation 14.22. Suppose that there is a polynomial-time parameterized reduction from problem A to problem B such that if the parameter of an instance of A is k, then the parameter of the constructed instance of B is at most $g(k)$ for some nondecreasing function g. Then an $f(k) \cdot n^{o(h(k))}$-time algorithm for B for some computable function f and nondecreasing function h implies an $f(g(k)) \cdot n^{o(h(g(k)))}$ algorithm for A.

Therefore, in order to exclude an algorithm for a parameterized problem B with running time $f(k) \cdot n^{o(h(k))}$ for any computable function f, we need to provide a reduction from Clique to B that outputs instances with the parameter k bounded by $\mathcal{O}(g(k))$, where g is the inverse of h.

Observing that most of the parameterized reductions in Chapter 13 have linear parameter dependence, we can obtain tight lower bounds for a number of problems.

Corollary 14.23. *Assuming ETH, there is no $f(k)n^{o(k)}$-time algorithm for the following problems for any computable function f:*

- Clique *on regular graphs,*
- Independent Set *on regular graphs,*
- Multicolored Clique *on regular graphs,*
- Multicolored Independent Set *on regular graphs,*
- Dominating Set*,*
- Dominating Set on Tournaments*,*
- Connected Dominating Set*,*

- Balanced Vertex Separator *parameterized by the size k of the solution,*
- List Coloring *parameterized by the treewidth k.*

Let us point out the surprising fact that we have the same lower bound here for Dominating Set and Dominating Set on Tournaments (the exponent of n cannot be $o(k)$), despite the significant difference in complexity for the unparameterized versions: only $2^{\Theta(n)}$-time algorithms are known for Dominating Set and they are optimal under ETH (see Theorem 14.6), while Dominating Set on Tournaments can be solved in time $n^{\mathcal{O}(\log n)}$. From the viewpoint of Observation 14.22, all that matters is that the reduction of Theorem 13.14 has the property that the new parameter is linear in the original parameter. The double-exponential dependence on k in the running time of the reduction does not affect the quality of the lower bound obtained in terms of the exponent of n.

There are W[1]-hard problems for which it seems hard to obtain tight lower bounds using a parameterized reduction from Clique and Observation 14.22; we need new technology to make the lower bounds tighter for these problems. Consider, for example, the Odd Set problem. In Theorem 13.31, we have presented a parameterized reduction from Multicolored Clique to Odd Set. A crucial idea of the reduction was to represent both the vertices and the edges of the k-clique in the Odd Set instance, and this was the reason why the parameter of the constructed instance was $k + \binom{k}{2} = \mathcal{O}(k^2)$. Therefore, Observation 14.22 with $g(k) = \mathcal{O}(k^2)$ and $h(k) = \sqrt{k}$ implies that, assuming ETH, there is no $f(k)n^{o(\sqrt{k})}$-time algorithm for Odd Set for any computable function f. The naive algorithm solves the problem in time $n^{\mathcal{O}(k)}$ by brute force, giving a significant gap between the upper and the lower bound. As we have discussed in Section 13.6.3, it seems difficult to design a reduction from Clique to Odd Set in which we represent only the vertices of the k-clique, and representing the $\binom{k}{2}$ edges of the k-clique inevitably leads to a quadratic blow-up of the parameter.

A key idea for making these reductions from Clique tighter is to interpret them as reductions from Subgraph Isomorphism instead. In the Subgraph Isomorphism problem, we are given two graphs H and G; the task is to decide if H is isomorphic to a (not necessarily induced) subgraph of G. Parameterized reductions from Clique where vertices and edges are represented by gadgets (e.g, Theorem 13.31) can be usually generalized to reductions from Subgraph Isomorphism with minimal modifications. As finding a k-clique is a special case of Subgraph Isomorphism, Theorem 14.21 implies that, assuming ETH, there is no $f(k)n^{o(\sqrt{k})}$-time algorithm for Subgraph Isomorphism for any computable function f, where k is the number of edges of the smaller graph H. The following result, whose proof is beyond the scope of this chapter, gives a lower bound parameterized by the number of *edges* of H:

Theorem 14.24 ([352]). *Assuming ETH, there is no $f(k)n^{o(k/\log k)}$-time algorithm for* Subgraph Isomorphism*, where k is the number of edges of the smaller graph H and f is a computable function.*

We remark that it is an interesting open question whether the factor $\log k$ in the exponent can be removed, making this result tight. We can modify the reduction of Theorem 13.31 to be a reduction from Subgraph Isomorphism to Odd Set, where the new parameter is linear in the number of edges of the smaller graph H (Exercise 14.11). Then Theorem 14.24 and Observation 14.22 give a tighter lower bound, almost matching the trivial $n^{\mathcal{O}(k)}$-time algorithm.

Theorem 14.25. *Assuming ETH, there is no $f(k)n^{o(k/\log k)}$-time algorithm for* Odd Set*.*

Similarly, one can modify the reduction in the proof of Theorem 13.33 to get tighter lower bounds for Strongly Connected Steiner Subgraph (Exercise 14.12).

Theorem 14.26. *Assuming ETH, there is no $f(k)n^{o(k/\log k)}$-time algorithm for* Strongly Connected Steiner Subgraph*.*

Closest Substring (a generalization of Closest String) is an extreme example where reductions increase the parameter exponentially or even double exponentially, and therefore we obtain very weak lower bounds. In this problem, the input consists of strings $s_1, \ldots, s_t$ over an alphabet Σ, and integers L and d. The task is to find a string s of length L such that, for every $1 \le i \le t$, s_i has a substring s'_i of length L with $d_H(s, s_i) \le d$. Here $d_H(x, y)$ is the Hamming-distance of two strings, that is, the number of positions where they differ.

Let us restrict our attention to the case where the alphabet is of constant size, say binary. There is a (very involved) reduction from Clique (with k being the size of the clique we are looking for) to Closest Substring where $d = 2^{\mathcal{O}(k)}$ and $t = 2^{2^{\mathcal{O}(k)}}$ in the constructed instance. Therefore, we get weak lower bounds with only $o(\log d)$ and $o(\log \log t)$ in the exponent. Interestingly, these lower bounds are actually tight, as there are algorithms matching them.

Theorem 14.27 ([349]). Closest Substring *over an alphabet of constant size can be solved in time $f(d)n^{\mathcal{O}(\log d)}$ or in time $f(t, d)n^{\mathcal{O}(\log \log t)}$. Furthermore, assuming ETH, there are no algorithms for the problem with running time $f(t, d)n^{o(\log d)}$ or $f(t, d)n^{o(\log \log t)}$ for any computable function f.*

14.4.1 Planar and geometric problems

We have seen in earlier chapters that many parameterized problems are fixed-parameter tractable when restricted to planar graphs. There are powerful

$S_{1,1}$: (1,1) (3,1) (2,4)	$S_{1,2}$: (5,1) (1,4) (5,3)	$S_{1,3}$: (1,1) (2,4) (3,3)
$S_{2,1}$: (2,2) (1,4)	$S_{2,2}$: (3,1) (1,2)	$S_{2,3}$: (2,2) (2,3)
$S_{3,1}$: (1,3) (2,3) (3,3)	$S_{3,2}$: (1,1) (1,3)	$S_{3,3}$: (2,3) (5,3)

Fig. 14.2: An instance of Grid Tiling with $k = 3$ and $n = 5$. The red pairs form a solution

techniques available for attacking planar problems, for example, the shifting strategy (Section 7.7.3) or bidimensionality (Section 7.7.2). Nevertheless, there exist natural parameterized problems that are W[1]-hard on planar graphs. In this section, we develop a convenient technology for proving the W[1]-hardness of these problems. Moreover, assuming ETH, this technology allows us to prove that there is no $f(k) \cdot n^{o(\sqrt{k})}$ algorithm for the problem for any computable function f. This is very often matched by a corresponding algorithm, giving us tight understanding of the complexity of a planar problem.

The key tool of the section is the following (somewhat artificial) problem, which will be a convenient starting point for reductions to planar problems. The input of Grid Tiling consists of an integer k, an integer n, and a collection $\mathcal{S}$ of k^2 nonempty sets $S_{i,j} \subseteq [n] \times [n]$ $(1 \leq i, j \leq k)$. The task is to find, for each $1 \leq i, j \leq k$, a pair $s_{i,j} \in S_{i,j}$ such that

- If $s_{i,j} = (a, b)$ and $s_{i+1,j} = (a', b')$, then $a = a'$.
- If $s_{i,j} = (a, b)$ and $s_{i,j+1} = (a', b')$, then $b = b'$.

In other words, if (i, j) and (i', j') are adjacent in the first or second coordinate, then $s_{i,j}$ and $s_{i',j'}$ agree in the first or second coordinate, respectively. An instance of Grid Tiling with a solution is shown in Fig. 14.2: we imagine $S_{i,j}$ to be in row i and column j of a matrix. Observe that the constraints ensure that the first coordinate of the solution is the same in each column and the second coordinate is the same in each row. We can prove the hardness of Grid Tiling by a simple reduction from Clique.

Theorem 14.28. Grid Tiling *is* W[1]*-hard and, unless ETH fails, it has no $f(k)n^{o(k)}$-time algorithm for any computable function f.*

Proof. The proof is by reduction from Clique. Let (G, k) be an instance of Clique. Let $n = |V(G)|$; we assume that the vertices of G are the integers in $[n]$. We construct an instance of Grid Tiling as follows. For every $1 \leq i, j \leq k$, we define

$$S_{i,j} = \begin{cases} \{(a, a) \ : \ 1 \leq a \leq n\} & \text{if } i = j, \\ \{(a, b) \ : \ a \neq b \text{ and vertices } a \text{ and } b \text{ are adjacent in } G\} & \text{if } i \neq j. \end{cases}$$

We claim that the constructed instance of Grid Tiling has a solution if and only if G contains a k-clique. First, suppose that vertices $v_1, \ldots, v_k$ are distinct and form a clique in G. Then it is easy to verify that $s_{i,j} = (v_j, v_i)$ (note that the order of i and j is swapped in the indices[5]) is a solution of Grid Tiling. Conversely, suppose that the pairs $s_{i,j}$ $(1 \leq i, j \leq k)$ form a solution. As $s_{i,j}$ and $s_{i+1,j}$ agree in the first coordinate and $s_{i,j}$ and $s_{i,j+1}$ agree in the second coordinate, it follows that there are values $v_1, \ldots, v_k, v'_1, \ldots, v'_k$ such that $s_{i,j} = (v_j, v'_i)$ for every $1 \leq i, j \leq k$. As $S_{i,i} = \{(a, a) \ : \ 1 \leq a \leq n\}$, this is only possible if $v_i = v'_i$ for every $1 \leq i \leq k$. It follows that v_i and v_j are two adjacent (distinct) vertices: as $(v_j, v_i) = (v_j, v'_i) \in S_{i,j}$, the definition of $S_{i,j}$ implies that $v_i v_j$ is an edge of G. Therefore, $v_1, \ldots, v_k$ is indeed a k-clique in G. □

There are two aspects of Grid Tiling that make it particularly suited for reduction to planar problems:

1. If the sets are represented by gadgets, then each gadget has to interact only with the gadgets representing adjacent sets, making the high-level structure of the constructed instance planar.
2. The constraint that needs to be enforced on adjacent gadgets is fairly simple: equality of the first/second coordinate of the value.

The following reduction demonstrates the first aspect of Grid Tiling. In Section 13.6, we have shown, by a reduction from Multicolored Clique, that List Coloring is W[1]-hard parameterized by treewidth. Furthermore, in Section 14.4, we have observed that the reduction actually shows that it has no $f(t)n^{o(t)}$-time algorithm for any computable function f, unless ETH fails. As the reduction in Theorem 13.30 was from Multicolored Clique, the created instance is highly non-planar (see Fig. 13.5), being essentially a subdivided clique. On the other hand, when we reduce from Grid Tiling, the resulting graph is grid-like, and hence planar.

[5] We could have defined Grid Tiling by swapping the meaning of the coordinates of $s_{i,j}$, that is, $s_{i,j}$ and $s_{i,j+1}$ have to agree in the first coordinate. Then we would have avoided the unnatural swap of indices in this proof. However, as we shall see, in many proofs (e.g., Theorems 14.32 and 14.34), the current definition is much more convenient.

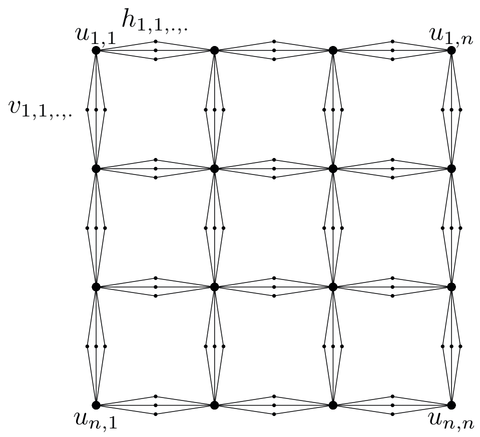


Fig. 14.3: The graph G' of the List Coloring instance constructed in the reduction of Theorem 14.29 for $k = 4$

Theorem 14.29. List Coloring *on planar graphs parameterized by the treewidth t is* W[1]*-hard. Moreover, it has no $f(t)n^{o(t)}$-time algorithm for any computable function f, unless ETH fails.*

Proof. We present a parameterized reduction from Grid Tiling. Let $(n, k, \mathcal{S})$ be an instance of Grid Tiling. The set C of colors will correspond to $[n]\times[n]$; then every set $S_{i,j}$ can be interpreted as a set of colors. We construct a graph G' the following way (see Fig. 14.3).

(i) For every $1 \leq i, j \leq k$, we introduce a vertex $u_{i,j}$ with $L(u_{i,j}) = S_{i,j}$.
(ii) For every $1 \leq i < k$, $1 \leq j \leq k$ and for every pair $(a, b), (a', b') \in [n] \times [n]$ with $a \neq a'$, we introduce a vertex $v_{i,j,(a,b),(a',b')}$ whose list is $\{(a, b), (a', b')\}$ and make it adjacent to $u_{i,j}$ and $u_{i+1,j}$.
(iii) For every $1 \leq i \leq k$, $1 \leq j < k$ and for every pair $(a, b), (a', b') \in [n] \times [n]$ with $b \neq b'$, we introduce a vertex $h_{i,j,(a,b),(a',b')}$ whose list is $\{(a, b), (a', b')\}$ and make it adjacent to $u_{i,j}$ and $u_{i,j+1}$.

To bound the treewidth of G', observe that every vertex other than the vertices $u_{i,j}$ has degree 2. Suppressing degree-2 vertices of G' does not change its treewidth (see Exercise 7.13), assuming its treewidth is at least 3, and gives a $k \times k$-grid with parallel edges, which has treewidth k.

Suppose that $\{s_{i,j} \in S_{i,j} \; : \; 1 \leq i, j \leq k\}$ is a solution of the Grid Tiling instance; let $s_{i,j} = (a_{i,j}, b_{i,j})$. We can construct the following list coloring of G'. First, we set the color of $u_{i,j}$ to $s_{i,j} \in S_{i,j} = L(u_{i,j})$. Then for every vertex $v_{i,j,(a,b),(a',b')}$, it is true that either $u_{i,j}$ has a color different from (a, b), or $u_{i+1,j}$ has a color different from (a', b') (as $a_{i,j} = a_{i+1,j}$, while $a \neq a'$). Therefore, one of the two colors appearing in the list of $v_{i,j,(a,b),(a',b')}$ is not used on its neighbors, hence we can extend the coloring to this vertex. The argument is similar for the vertices $h_{i,j,(a,b),(a',b')}$.

For the reverse direction, suppose that $c : V(G') \to [n] \times [n]$ is a list coloring of G'. We claim that the pairs $s_{i,j} = c(u_{i,j}) \in L(u_{i,j}) = S_{i,j}$ form a solution for the Grid Tiling instance. For a contradiction, suppose that $s_{i,j} = (a, b)$ and $s_{i+1,j} = (a', b')$ do not agree in the first coordinate, that is, $a \neq a'$. Then the vertex $v_{i,j,(a,b),(a',b')}$ exists in G and has list $\{(a, b), (a', b')\}$. However, this vertex is adjacent to both $u_{i,j}$ and $u_{i+1,j}$, which have colors (a, b) and (a', b') in the coloring; hence $v_{i,j,(a,b),(a',b')}$ cannot have either color on its list, a contradiction. The argument is similar in the case when $s_{i,j}$ and $s_{i,j+1}$ do not agree in the second coordinate; in this case, we arrive to a contradiction by observing that both colors in the list vertex $h_{i,j,(a,b),(a',b')}$ are already used by its neighbors. □

The second reason why Grid Tiling is a useful starting point for reductions to planar problems is that the constraints between adjacent gadgets are very simple: it requires that the first/second coordinates of the values of the two gadgets are equal. This can be much simpler than testing adjacency of two vertices in a reduction from Multicolored Clique and is similar to the projection constraint between edge gadgets and vertex gadgets (see Theorem 13.31). In Theorem 14.29, the List Coloring problem was very powerful and it was easy to express arbitrary binary relations, hence this aspect of Grid Tiling was not exploited.

There are reductions, however, where it does matter that the constraints between the gadgets are simple. We introduce a variant of Grid Tiling where the constraints are even more suited for planar problems. The input of Grid Tiling with $\leq$ is the same as the input of Grid Tiling, but we have less than or equal constraints between the first/second coordinates instead of equality (see Fig. 14.4). That is, the task is to find, for each $1 \leq i, j \leq k$, a value $s_{i,j} \in S_{i,j}$ such that

- If $s_{i,j} = (a, b)$ and $s_{i+1,j} = (a', b')$, then $a \leq a'$.
- If $s_{i,j} = (a, b)$ and $s_{i,j+1} = (a', b')$, then $b \leq b'$.

This variant of Grid Tiling is very convenient when we want to prove lower bounds for a problem involving planar/geometric objects or distances between points: the $\leq$ constraints have natural interpretations in such problems.

We show first that the lower bounds of Theorem 14.28 hold for Grid Tiling with $\leq$ as well.

Theorem 14.30. Grid Tiling with $\leq$ *is* W[1]*-hard and, unless ETH fails, it has no $f(k)n^{o(k)}$-time algorithm for any computable function f.*

Proof. Given an instance $(n, k, \mathcal{S})$ of Grid Tiling, we construct an equivalent instance $(n', k', \mathcal{S}')$ of Grid Tiling with $\leq$ with $n' = 3n^2(k+1)+n^2+n$ and $k' = 4k$. For every set $S_{i,j}$ of the input instance, there are 16 corresponding sets $S'_{i',j'}$ ($4i - 3 \leq i' \leq 4i$, $4j - 3 \leq j' \leq 4j$) in the constructed instance (see Fig. 14.5). We call these sets the *gadget representing* $S_{i,j}$. The four inner

$S_{1,1}$: (1,1) (3,1) (2,4)	$S_{1,2}$: (5,1) (1,4) (5,3)	$S_{1,3}$: (1,1) (2,5) (3,3)
$S_{2,1}$: (2,2) (1,4)	$S_{2,2}$: (3,1) (2,2)	$S_{2,3}$: (3,2) (2,3)
$S_{3,1}$: (1,3) (2,3) (3,3)	$S_{3,2}$: (1,1) (2,3)	$S_{3,3}$: (5,4) (3,4)

Fig. 14.4: An instance of GRID TILING WITH $\leq$ with $k = 3$ and $n = 5$. The red pairs form a solution

sets $S_{4i-2,4j-2}$, $S_{4i-2,4j-1}$, $S_{4i-1,4j-2}$, $S_{4i-1,4j-1}$ are dummy sets containing only the single pair shown in the figure. The 12 outer sets are defined as follows. We define the mapping $\iota(a, b) = na + b$ and let $N = 3n^2$. For every $(a, b) \in S_{i,j}$, we let $z = \iota(a, b)$ and introduce the pairs shown in Fig. 14.5 into the 12 outer sets. It can be easily verified that both coordinates of each pair are positive and at most n'. This concludes the description of the constructed instance.

The intuition behind the construction is the following. Selecting a pair from one of the 12 outer sets in a gadget selects a pair from $S_{i,j}$. We show that in every solution, a "vortex" on the 12 outer sets ensures that each of them select the same pair $s_{i,j} \in S_{i,j}$. Then we show that the interaction of the outer sets of the gadgets enforce exactly the constraints of the GRID TILING problem. For example, the second set of the last row of the gadget representing $S_{i,j}$ and the second set of the first row of the gadget representing $S_{i+1,j}$ interact in a way that ensures that the first coordinate of $s_{i,j}$ is at most the first coordinate of $s_{i+1,j}$, while the third sets of these rows ensure the reverse inequality, implying that the two first coordinates are equal.

Let $s_{i,j} \in S_{i,j}$ for $1 \leq i, j \leq k$ be a solution of the GRID TILING instance. For every $s_{i,j} = (a, b)$, we select the corresponding pairs from the 16 sets in the gadget of $S_{i,j}$ for $z = \iota(a, b)$, as shown in Fig. 14.5. First, it is easy to verify that the constraints are satisfied between the sets of the same gadget. Consider now the last row of the gadget of $S_{i,j}$ and the first row of the gadget of $S_{i+1,j}$. For the first sets in these rows, the constraints are satisfied:

$S'_{4i-3,4j-3}$: $(iN-z, jN+z)$	$S'_{4i-3,4j-2}$: $(iN+a, jN+z)$	$S'_{4i-3,4j-1}$: $(iN-a, jN+z)$	$S'_{4i-3,4j}$: $(iN+z, jN+z)$
$S'_{4i-2,4j-3}$: $(iN-z, jN+b)$	$S'_{4i-2,4j-2}$: $((i+1)N, (j+1)N)$	$S'_{4i-2,4j-1}$: $(iN, (j+1)N)$	$S'_{4i-2,4j}$: $(iN+z, (j+1)N+b)$
$S'_{4i-1,4j-3}$: $(iN-z, jN-b)$	$S'_{4i-1,4j-2}$: $((i+1)N, jN)$	$S'_{4i-1,4j-1}$: (iN, jN)	$S'_{4i-1,4j}$: $(iN+z, (j+1)N-b)$
$S'_{4i,4j-3}$: $(iN-z, jN-z)$	$S'_{4i,4j-2}$: $((i+1)N+a, jN-z)$	$S'_{4i,4j-1}$: $((i+1)N-a, jN-z)$	$S'_{4i,4j}$: $(iN+z, jN-z)$

Fig. 14.5: The 16 sets of the constructed Grid Tiling with $\le$ instance representing a set $S_{i,j}$ of the Grid Tiling in the reduction in the proof of Theorem 14.30, together with the pairs corresponding to a pair $(a,b) \in S'_{i,j}$ (with $z = \iota(a,b)$)

the pair selected from $S'_{4i,4j-3}$ has first coordinate less than iN, while the pair selected from $S'_{4(i+1)-3,4j-3} = S'_{4i+1,4j-3}$ has the first coordinate at least $(i+1)N - (n^2+n) > iN$. Similarly, there is no conflict between the last sets of these rows. If $a_{i,j} = a_{i+1,j}$ are the first coordinates of $s_{i,j}$ and $s_{i+1,j}$, then the first coordinates of the sets selected from the second sets of the rows, $S'_{4i,4j-2}$ and $S'_{4i+4j-2}$, are $(i+1)N + a_{i,j}$ and $(i+1)N + a_{i+1,j}$, respectively, and the former is equal to the latter. One can show in a similar way that there is no conflict between the third sets of the rows, and that there is no conflict between the last column of the gadget representing $S_{i,j}$ and the first column of the gadget representing $S_{i,j+1}$.

For the reverse direction, suppose that $s'_{i,j}$ for $1 \le i,j \le k'$ is a solution of the constructed Grid Tiling with $\le$ instance. Consider the 12 outer sets in the gadget representing $S_{i,j}$. The 12 pairs selected in the solution from these sets define 12 values z; let $z_{4i-3,4j-3}$, etc., be these 12 values. We claim that all these 12 values are the same. The second coordinate of the set selected from $S'_{4i-3,4j-3}$ is $jN + z_{4i-3,4j-3}$, the second coordinate of the set selected from $S'_{4i-3,4j-2}$ is $jN + z_{4i-3,4j-2}$; hence the definition of Grid Tiling with $\le$ implies $z_{4i-3,4j-3} \le z_{4i-3,4j-2}$. With similar reasoning, by going around the outer sets, we get the following inequalities.

$$\begin{array}{rllll}
z_{4i-3,4j-3} \le z_{4i-3,4j-2} & \le z_{4i-3,4j-1} & \le z_{4i-3,4j} & \text{(first row)} \\
z_{4i-3,4j} \le z_{4i-2,4j} & \le z_{4i-1,4j} & \le z_{4i,4j} & \text{(last column)} \\
-z_{4i,4j-3} \le -z_{4i,4j-2} & \le -z_{4i,4j-1} & \le -z_{4i,4j} & \text{(last row)} \\
-z_{4i-3,4j-3} \le -z_{4i-2,4j-3} & \le -z_{4i-1,4j-3} & \le -z_{4i,4j-3} & \text{(first column)}
\end{array}$$

Putting everything together, we get a cycle of inequalities showing that these 12 values are all equal; let $z^{i,j}$ be this common value and let $s_{i,j} = (a_{i,j}, b_{i,j})$ be the corresponding pair, that is, $\iota(a_{i,j}, b_{i,j}) = z^{i,j}$. The fact that $z^{i,j}$ was defined using the pairs appearing in the gadget of $S_{i,j}$ implies that $s_{i,j} \in S_{i,j}$. Let us show now that $a_{i,j} = a_{i+1,j}$. The pair selected from $S'_{4i,4j-2}$ has first coordinate $(i+1)N + a_{i,j}$, while the pair selected from $S'_{4(i+1)-3,4j-2} = S'_{4i+1,4j-2}$ has first coordinate $(i+1)N + a_{i+1,j}$. The definition of Grid Tiling with $\le$ implies now that $a_{i,j} \le a_{i+1,j}$ holds. Similarly, comparing the first coordinates of the pairs selected from $S'_{4i,4j-1}$ and $S'_{4i+1,4j-1}$ shows $-a_{i,j} \le -a_{i+1,j}$; hence $a_{i,j} = a_{i+1,j}$ follows. A similar argument, by looking at the last column of the gadget representing $S_{i,j}$ and the first column of the gadget representing $S_{i,j+1}$ shows $b_{i,j} = b_{i,j+1}$. Therefore, we have proved that the constructed $s_{i,j}$'s indeed form a solution of the Grid Tiling instance.

□

Scattered Set is a generalization of Independent Set: given a graph G and integers k and d, the task is to find a set S of k vertices in G such that the distance between any two distinct $u, v \in S$ is at least d. Clearly, Independent Set is the special case $d = 2$, hence there is no $f(k)n^{o(k)}$-time algorithm for Scattered Set on general graphs, unless ETH fails (Corollary 14.23). On planar graphs, we have seen that the special case Independent Set is FPT (parameterized by k) and has subexponential parameterized algorithms. Moreover, we can show that Scattered Set remains FPT on planar graphs for any fixed d, or with combined parameters k and d (Exercise 7.41). If d is part of the input, we can do somewhat better than finding a solution by an $n^{\mathcal{O}(k)}$ time brute-force algorithm.

Theorem 14.31 ([355]). Scattered Set *on planar graphs can be solved in time* $n^{\mathcal{O}(\sqrt{k})}$.

We show now that the algorithm of Theorem 14.31 is optimal with respect to the dependence of the exponent on k.

Theorem 14.32. Scattered Set *on planar graphs is* W[1]*-hard when parameterized by k only. Moreover, it has no $f(k)n^{o(\sqrt{k})}$-time algorithm for any computable function f, unless ETH fails.*

Proof. We present a parameterized reduction from Grid Tiling with $\le$. Let $(n, k, \mathcal{S})$ be an instance of Grid Tiling with $\le$; without loss of generality assume that $k \ge 2$. We represent each $S_{i,j}$ with an $n \times n$ grid $R_{i,j}$; let $v^{i,j}_{a,b}$ be the vertex of this grid in row a and column b. (Note that the distance between the first and last columns or rows of an $n \times n$ grid is $n-1$.) Let $L = 100n$ and $d = 4L + n - 1$. We construct a graph G' in the following way (see Fig. 14.6).

(i) For every $1 \le i, j \le k$ and $(a, b) \in S_{i,j}$, we introduce a vertex $w^{i,j}_{a,b}$ and connect it to the vertex $v^{i,j}_{a,b}$ with a path of length L.
(ii) For every $1 \le i \le k$, $1 \le j < k$, we introduce a new vertex $\alpha_{i,j}$ and connect $\alpha_{i,j}$ to $v^{i,j}_{a,n}$ and $v^{i,j+1}_{a,1}$ for every $a \in [n]$ with a path of length L.
(iii) For every $1 \le i < k$, $1 \le j \le k$, we introduce a new vertex $\gamma_{i,j}$ and connect $\gamma_{i,j}$ to $v^{i,j}_{n,b}$ and $v^{i+1,j}_{1,b}$ for every $b \in [n]$ with a path of length L.

Note that the resulting graph is planar, see Fig. 14.6.

We may assume that the solution selects exactly one vertex $w^{i,j}_{a,b}$ from each grid $R_{i,j}$. The locations of these vertices correspond to a solution of the Grid Tiling with $\le$ instance. The distance constraint of Scattered Set ensures that the row number of the selected vertex in $R_{i,j}$ is at most the row number of the selected vertex in $R_{i+1,j}$, and the column number of the selected vertex in $R_{i,j}$ is at most the column number of the selected vertex in $R_{i,j+1}$.

Suppose that $s_{i,j} \in S_{i,j}$ is a solution for the Grid Tiling with $\le$ instance; let $s_{i,j} = (a_{i,j}, b_{i,j})$. For every $1 \le i, j \le k$, let Z contain $z_{i,j} = w^{i,j}_{a_{i,j},b_{i,j}}$. We claim that vertices in Z are at distance at least d from each other. Notice first that two adjacent grids have distance at least $2L$ from each other: we have to go through at least two paths of length L to go from $R_{i,j}$ to $R_{i+1,j}$ for example. Therefore, if $|i-i'|+|j-j'| \ge 2$, then every path between $z_{i,j}$ and $z_{i',j'}$ has to go through at least $2 \cdot 2 + 2 = 6$ paths of length L (the extra 2 comes from the 2 paths with endpoints $z_{i,j}$ and $z_{i',j'}$), hence their distance is at least $6L > d$. Consider now the vertices $z_{i,j}$ and $z_{i+1,j}$. If a path P connecting these two vertices has length less than $6L$, then P has to go through $\gamma_{i,j}$. The distance between $z_{i,j}$ and $\gamma_{i,j}$ is $2L + n - a_{i,j} < 3L$, which is realized by a path using two paths of length L and a vertical subpath

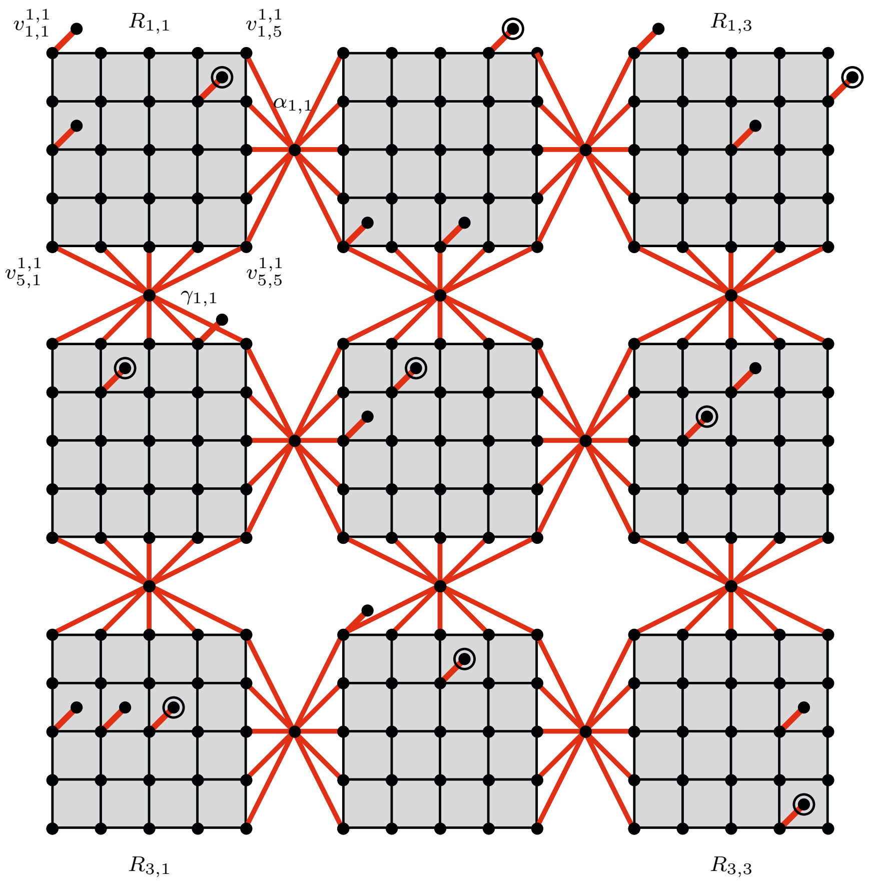


Fig. 14.6: Theorem 14.32: reducing the instance of Grid Tiling with $\leq$ in Fig. 14.4 with $k = 3$ and $n = 5$ to an instance of Scattered Set. The red edges represent paths of length L, the circled vertices correspond to the solution shown in red in Fig. 14.4

of the grid. Note that the shortest path cannot leave the grid and return to it, as then it would use at least four paths of length L. Similarly, we can observe that the distance between $z_{i+1,j}$ and $\gamma_{i,j}$ is $2L + a_{i+1,j} - 1$. Thus the distance between $z_{i,j}$ and $z_{i+1,j}$ is $4L + n - 1 + (a_{i+1,j} - a_{i,j}) \geq 4L + n - 1 = d$, since $s_{i,j}$ and $s_{i+1,j}$ satisfy the constraint that the first coordinate of $s_{i,j}$ is at most the first coordinate of $s_{i+1,j}$. A similar argument (using the fact that the second coordinate of $s_{i,j}$ is at most the second coordinate of $s_{i,j+1}$) shows that the distance between $z_{i,j}$ and $z_{i,j+1}$ is at least d. Therefore, the distance between any two vertices in Z is at least d.

For the reverse direction, suppose that Z is a set of k^2 vertices with pairwise distance at least d. Let $X_{i,j}$ be the set of all vertices that are at distance at most $L < d/2$ from some vertex of $R_{i,j}$. Observe that the distance between any two vertices in $X_{i,j}$ is less than d, hence $X_{i,j}$ can contain at most one vertex of Z. Furthermore, observe that the union of the $X_{i,j}$'s cover the entire graph. Therefore, each of the k^2 vertices of Z appear in at least one of these sets, which is only possible if each $X_{i,j}$ contains exactly one vertex of Z and each vertex of Z appears in only one $X_{i,j}$. In particular, this means that the vertices $\alpha_{i,j}$, $\gamma_{i,j}$ of G, which are contained in two of these sets, cannot be part of the solution Z. We claim that no vertex of the grid $R_{i,j}$ is selected. As $k \geq 2$, grid $R_{i,j}$ has a neighbor $R_{i',j'}$, that is, $|i-i'|+|j-j'|=1$. Now every vertex of $X_{i',j'}$ is at distance at most $n-1+2L+n-1+L < d$ from every vertex of $R_{i,j}$. This means that if a vertex of $R_{i,j}$ is in Z, then $X_{i',j'}$ is disjoint from Z, contradicting our observation above. A similar argument shows that a vertex of $X_{i,j} \cap Z$ cannot lie on any path connecting a side of $R_{i,j}$ with one of the neighboring vertices $\alpha_{i,j}$, $\alpha_{i,j-1}$, $\gamma_{i,j}$, $\gamma_{i-1,j}$. Hence, the single vertex of $X_{i,j} \cap Z$ has to be on some $v^{i,j}_{a_{i,j},b_{i,j}} - w^{i,j}_{a_{i,j},b_{i,j}}$ path for some $(a_{i,j}, b_{i,j}) \in S_{i,j}$— otherwise it would be too close to an adjacent grid. We may actually assume that this vertex is the degree-1 vertex $w^{i,j}_{a_{i,j},b_{i,j}}$ at the end of the path: moving the vertex closer to the end of the path does not make it any closer to the other vertices of Z. Let $z_{i,j} = w^{i,j}_{a_{i,j},b_{i,j}}$ be the vertex of Z contained in $X_{i,j}$, and let $s_{i,j} = (a_{i,j}, b_{i,j})$. We claim that the pairs $s_{i,j}$ form a solution for the Grid Tiling instance. First, $s_{i,j} = (a_{i,j}, b_{i,j}) \in S_{i,j}$ follows from the way the graph is defined. To see that $a_{i,j} \leq a_{i+1,j}$, let us compute the distance between $z_{i,j}$ and $z_{i+1,j}$. The distance between $z_{i,j} = w^{i,j}_{a_{i,j},b_{i,j}}$ and $\gamma_{i,j}$ is $L+n-a_{i,j}$, the distance between $z_{i+1,j} = w^{i+1,j}_{a_{i+1,j},b_{i+1,j}}$ and $\gamma_{i,j}$ is $L+a_{i+1,j}-1$, hence the distance between $z_{i,j}$ and $z_{i+1,j}$ is $4L+n-1+(a_{i+1,j}-a_{i,j}) = d+(a_{i+1,j}-a_{i,j})$. As we know that this distance is at least d, the inequality $a_{i,j} \leq a_{i+1,j}$ follows. By a similar argument, computing the distance between $z_{i,j}$ and $z_{i,j+1}$ shows that $b_{i,j} \leq b_{i,j+1}$ holds. Thus the pairs $s_{i,j}$ indeed form a solution of the Grid Tiling with $\leq$ instance. □

We finish this section with a geometric problem. Given a set of disks of unit diameter in the plane (described by the coordinates of their centers) and an integer k, Unit Disk Independent Set asks for a subset S of k disks that are pairwise disjoint. In other words, we need to select k centers such that any two of them are at distance at least 1 from each other. It is possible to solve the problem more efficiently than the $n^{\mathcal{O}(k)}$-time brute-force algorithm.

Theorem 14.33 ([9]). Unit Disk Independent Set *can be solved in time* $n^{\mathcal{O}(\sqrt{k})}$.

We show that the exponent $\mathcal{O}(\sqrt{k})$ in Theorem 14.33 is best possible by a reduction from Grid Tiling with $\leq$. In the proof, we face a minor notational difficulty. When defining the Grid Tiling problem, we imagined the sets $S_{i,j}$

arranged in a matrix, with $S_{i,j}$ being in row i and column j. When reducing Grid Tiling to a geometric problem, the natural idea is to represent $S_{i,j}$ with a gadget located around coordinate (i, j). However, this introduces an unnatural 90 degrees rotation compared to the layout of the $S_{i,j}$'s in the matrix, which can be confusing in the presentation of a reduction. Therefore, for geometric problems, it is convenient to imagine that $S_{i,j}$ is "located" at coordinate (i, j). To emphasize this interpretation, we use the notation $S[x, y]$ to refer to the sets; we imagine that $S[x, y]$ is at location (x, y), hence sets with the same x are on a vertical line and sets with the same y are on the same horizontal line (see Fig. 14.7). The constraints of Grid Tiling are the same as before: the pairs selected from $S[x, y]$ and $S[x + 1, y]$ agree in the first coordinate, while the pairs selected from $S[x, y]$ and $S[x, y + 1]$ agree in the second coordinate. Grid Tiling with $\leq$ is defined similarly. With this notation, we can give a very clean and transparent reduction to Unit Disk Independent Set.

Theorem 14.34. Unit Disk Independent Set *is* W[1]*-hard and, unless ETH fails, it has no* $f(k)n^{o(\sqrt{k})}$*-time algorithm for any computable function* f.

Proof. It will be convenient to work with open disks of radius $\frac{1}{2}$ (that is, diameter 1) in this proof: then two disks are nonintersecting if and only if the distance between their centers is at least 1. The proof is by reduction from Grid Tiling with $\leq$. Let $I = (n, k, \mathcal{S})$ be an instance of Grid Tiling with $\leq$. We construct a set D of unit disks such that D contains a set of k^2 pairwise nonintersecting disks if and only if I is a yes-instance.

Let $\varepsilon = 1/n^2$. For every $1 \leq x, y \leq k$ and every $(a, b) \in S[x, y] \subseteq [n] \times [n]$, we introduce into D an open disk of radius $\frac{1}{2}$ centered at $(x + \varepsilon a, y + \varepsilon b)$; let $D[x, y]$ be the set of these $|S[x, y]|$ disks introduced for a fixed x and y (see Fig. 14.7). Note that the disks in $D[x, y]$ all intersect each other. Therefore, if $D' \subseteq D$ is a set of pairwise nonintersecting disks, then $|D'| \leq k^2$ and $|D'| = k^2$ is possible only if D' contains exactly one disk from each $D[x, y]$. In the following, we prove that there is such a set of k^2 pairwise nonintersecting disks if and only if I is a yes-instance.

We need the following observation first. Consider two disks centered at $(x+\varepsilon a, y+\varepsilon b)$ and $(x+1+\varepsilon a', y+\varepsilon b')$ for some $(a, b), (a', b') \in [n] \times [n]$. We claim that they are nonintersecting if and only if $a \leq a'$. Indeed, if $a > a'$, then the square of the distance of the two centers is

$$\begin{aligned}(1 + \varepsilon(a' - a))^2 + \varepsilon^2(b' - b)^2 &\leq (1 + \varepsilon(a' - a))^2 + \varepsilon^2 n^2 \\ &\leq (1 - \varepsilon)^2 + \varepsilon = 1 - \varepsilon + \varepsilon^2 < 1\end{aligned}$$

(in the first inequality, we have used $b', b \leq n$; in the second inequality, we have used $a \geq a' + 1$ and $\varepsilon = 1/n^2$). On the other hand, if $a \leq a'$, then the square of the distance is at least $(1 + \varepsilon(a' - a))^2 \geq 1$, hence the two disks do not intersect (recall that the disks are open). This proves our claim. A similar

$S[1,3]$: (1,1) (2,5) (3,3)	$S[2,3]$: (3,2) (2,3)	$S[3,3]$: (5,4) (3,4)
$S[1,2]$: (5,1) (1,4) (5,3)	$S[2,2]$: (3,1) (2,2)	$S[3,2]$: (1,1) (2,3)
$S[1,1]$: (1,1) (3,1) (2,4)	$S[2,1]$: (2,2) (1,4)	$S[3,1]$: (1,3) (2,3) (3,3)

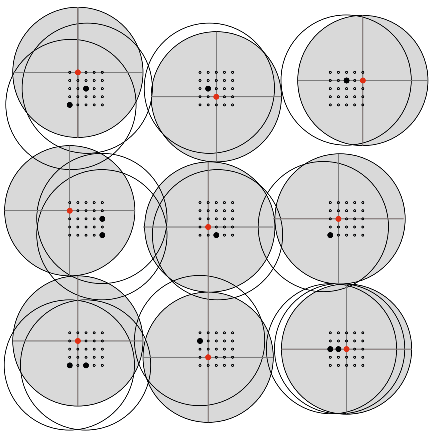


Fig. 14.7: An instance of Grid Tiling with $\leq$ with $k = 3$ and $n = 5$ and the corresponding instance of Unit Disk Independent Set constructed in the reduction of Theorem 14.34. The small dots show the potential positions for the centers, the large dots are the actual centers in the constructed instance. The shaded disks with red centers correspond to the solution of Grid Tiling with $\leq$ shown on the left

claim shows that disks centered at $(x+\varepsilon a, y+\varepsilon b)$ and $(x+\varepsilon a', y+1+\varepsilon b')$ are nonintersecting if and only if $b \leq b'$. Moreover, it is easy to see that the disks centered at $(x+\varepsilon a, y+\varepsilon b)$ and $(x'+\varepsilon a', y'+\varepsilon b')$ for some $1 \leq a, a', b, b' \leq n$ cannot intersect if $|x-x'|+|y-y'| \geq 2$: the square of the distance between the two centers is at least $2(1-\varepsilon n)^2 > 1$.

Let the pairs $s[x,y] \in S[x,y]$ form a solution of the instance I; let $s[x,y] = (a[x,y], b[x,y])$. For every $1 \leq x, y \leq k$, we select the disk $d[x,y]$ centered at $(x+\varepsilon a[x,y], y+\varepsilon b[x,y]) \in D[x,y]$. As we have seen, if $|x-x'|+|y-y'| \geq 2$, then $d[x,y]$ and $d[x',y']$ cannot intersect. As the $s[x,y]$'s form a solution of instance I, we have that $a[x,y] \leq a[x+1,y]$. Therefore, by our claim in the previous paragraph, the disks $d[x,y]$ and $d[x+1,y]$ do not intersect. Similarly, we have $b[x,y] \leq b[x,y+1]$, implying that $d[x,y]$ and $d[x,y+1]$ do not intersect either. Hence there is indeed a set of k^2 pairwise nonintersecting disks in D.

Conversely, let $D' \subseteq D$ be a set of k^2 pairwise independent disks. This is only possible if for every $1 \leq x, y \leq k$, the set D' contains a disk $d[x,y] \in D[x,y]$, that is, centered at $(x+\varepsilon a[x,y], j+\varepsilon b[x,y])$ for some $(a[x,y], b[x,y]) \in [n] \times [n]$. We claim that the pairs $s[x,y] = (a[x,y], b[x,y])$ form a solution of the instance I. First, $d[x,y] \in D[x,y]$ implies that $s[x,y] \in S[x,y]$. As we have seen above, the fact that $d[x,y]$ and $d[x+1,y]$ do not intersect implies that $a[x,y] \leq a[x+1,y]$. Similarly, the fact that $d[x,y]$ and $d[x,y+1]$ do not intersect each other implies that $b[x, y] \leq b[x, y+1]$. Thus the $s[x, y]$'s form a solution for the Grid Tiling with $\leq$ instance I. $\square$

*14.5 Lower bounds based on the Strong Exponential-Time Hypothesis

In essence, ETH enables us to estimate the optimal *magnitude* of the parametric dependency of the running time of parameterized algorithms. SETH is a more precise tool, using which one can pinpoint almost exactly the asymptotic behavior of the optimal running time, at a cost of adopting a much stronger complexity assumption. For instance, suppose we are working with a parameterized problem admitting an $\mathcal{O}^*(3^k)$ algorithm. A typical lower bound under ETH is of the form "No algorithm with running time $\mathcal{O}^*(2^{o(k)})$ can be expected". Under SETH one can prove statements like "For any $\varepsilon > 0$, there is no algorithm with running time $\mathcal{O}^*((3-\varepsilon)^k)$", thus showing that even the constant 3 in the base of the exponent is hard to improve.

Observe that for any constant c, saying that there exists an algorithm with running time $\mathcal{O}^*((c-\varepsilon)^k)$ for some $\varepsilon > 0$ is equivalent to saying that there exists an algorithm with running time $\mathcal{O}^*(c^{\delta k})$ for some $\delta < 1$. While the first notation gives a more intuitive feeling about what the lower bound actually means, the second is more convenient when it comes to formal proofs. We will hence use both notations, jumping between them implicitly.

Lower bounds under SETH can be proved by providing reductions from variants of CNF-SAT, similarly to the lower bounds under ETH. However, they are typically much more delicate. For ETH we are usually interested only in the asymptotics of how the parameter is transformed — it is relevant whether the output parameter is linear or quadratic in the input one, but the precise leading coefficient of the linear function does not play any role. For SETH it is precisely this coefficient that is crucial. To see it on an example, say that we provide two reductions from CNF-SAT to some parameterized problem L, where in the first reduction the output parameter k is equal to $2n$ (here, n is the number of variables), and in the second reduction the output parameter is equal to n. Then the first reduction shows only that, for any $\varepsilon > 0$, there is no algorithm with running time $\mathcal{O}^*((\sqrt{2}-\varepsilon)^k)$ assuming SETH, while the second one improves this lower bound to nonexistence of an $\mathcal{O}^*((2-\varepsilon)^k)$ algorithm.

Interestingly, observe that in the presented example we could state the lower bound under a slightly weaker assumption that the general CNF-SAT problem cannot be solved in time $\mathcal{O}^*((2-\varepsilon)^n)$ for some $\varepsilon > 0$. This is a common phenomenon in many lower bounds under SETH, where the reduction can be assumed to start from general CNF-SAT, rather than from q-SAT for increasing, yet constant values of q. However, there are examples where it is helpful to use the full power of SETH; we will see one such lower bound in the next section.

Before we proceed, let us comment that the problems for which SETH is known to give a tight understanding of their (parameterized) complexity are much scarcer than the problems affected by ETH. The difficulty lies in the very specific requirements that reductions proving lower bounds under SETH have to satisfy. Most known reductions from CNF-SAT or its variants encode the clauses of the input formula using some kind of gadgets. Therefore, in most cases it is hard to avoid the situation where the output parameter (or the total output size, if we are interested in the classical complexity of the problem) depends on the number of clauses instead of just variables; this is completely unacceptable when trying to prove a lower bound based on SETH. This applies in particular to problems parameterized by the requested size of the solution, like Vertex Cover or Feedback Vertex Set. In fact, basically all the known lower bounds under SETH for FPT problems concern *structural* parameterizations, such as the size of the universe of the instance (say, in the Hitting Set problem) or some width parameter of the input graph; such parameters are often much easier to control in terms of the number of variables.

Hence, one could argue that we are missing some important theoretical tool for SETH, similar to the sparsification lemma, which would enable us to unlock the full potential of this conjecture. Alternatively, this state of the art might indicate that SETH is simply too weak an assumption for proving precise statements about the running times of exponential-time algorithms, and we need stronger and better-suited working conjectures.

In the following three sections we present exemplary lower bounds that can be proved under the assumption of SETH. Actually, these examples almost cover all the significant applications of SETH in parameterized complexity that are known so far, as explained in the previous paragraphs.

14.5.1 Hitting Set ***parameterized by the size of the universe***

Recall that the Hitting Set problem can be solved by a brute-force search in time $\mathcal{O}^*(2^n)$, where n is the size of the universe.[6] It is natural to ask whether the constant 2 in the base of the exponent could be improved, or, in other words, whether one can search through significantly less than 2^n possible candidates for the solution. It appears that this is not the case under the assumption of SETH — the barrier of brute-force 2^n enumeration is hard to break.

[6] We recall that the $\mathcal{O}^*(\cdot)$ notation hides factors polynomial in the total input size, i.e., here polynomial in $|U| + |\mathcal{F}|$, and not just polynomial in $n = |U|$.

The first natural idea for a reduction from CNF-SAT would be the following. Create two elements of the universe t_x, f_x for each variable x, and add a set containing this pair to the constructed family. Set the budget for the size of the hitting set to be equal to n, so that in each of these two-element sets exactly one of the elements must be chosen. Choosing t_x corresponds to setting the variable to true, and choosing f_x corresponds to setting it to false, so we have one-to-one correspondence between assignments of the variables and candidate solutions. Now satisfaction of clauses can be encoded directly as hitting sets: for instance for a clause $x \vee \neg y \vee z$ we will create a set $\{t_x, f_y, t_z\}$, which is hit if and only if the clause is satisfied.

Unfortunately, this reduction gives only an $\mathcal{O}^*((\sqrt{2}-\varepsilon)^n)$ lower bound, since the universe in the output instance is of size $2n$. The problem is about redundancy: if we know that some t_x is chosen to the solution, then we are sure that f_x is not chosen, and thus we did not exploit the whole 2^{2n} search space of the output instance, but rather a subspace of cardinality 2^n. We hence need a smarter way of packing the information.

Suppose that the reduction we are going to construct will create a universe U of size n', which is slightly larger than n; for now think that $n' = n + \mathcal{O}(\log n)$. Suppose further that k is the intended size of the hitting set. Now observe that k must be very close to $n'/2$ for the following reason. If $k < 0.49 \cdot n'$ or $k > 0.51 \cdot n'$ for some n', then the solution might be found by brute-force verification of all the sets of $\binom{U}{<0.49n'}$ or $\binom{U}{>0.51n'}$, and there are only $\mathcal{O}((2-\varepsilon)^n)$ such sets for some $\varepsilon > 0$. Therefore, the intuition is that the 2^n search space of CNF-SAT has to be embedded roughly into $\binom{U}{n'/2}$; since $\left|\binom{U}{n'/2}\right| = \mathcal{O}\left(\frac{2^{n'}}{\sqrt{n'}}\right)$ by Stirling approximation, the additional additive $\mathcal{O}(\log n)$ factor in n' should be sufficient to ensure enough space to accommodate all possible variable assignments.

It seems, however, challenging to implement such an embedding so that the clauses can be also conveniently checked. Now is the point when it becomes useful to start with q-SAT instead of general CNF-SAT. Let us arbitrarily partition the variable set into large, but constant-size groups. Let p be the size of the group. Each group X will have a corresponding part of the universe U_X of size $p' = p + \mathcal{O}(\log p)$, and the space of assignments of variables of X will be embedded roughly into $\binom{U_X}{p'/2}$, which has size $\binom{p'}{p'/2} > 2^p$. Thus we have $n' = \frac{p'}{p} \cdot n$, and when p grows to infinity then n' becomes bounded by $(1+\alpha)n$, for any $\alpha > 0$. Finally, provided that we start with q-SAT instead of general CNF-SAT, every clause touches at most q groups of size p each, and hence it can be modelled using a constant (yet exponential in q and p) number of sets in the family.

We now proceed to a formal explanation.

Theorem 14.35. *Unless SETH fails, there is no algorithm for* Hitting Set *that achieves running time $\mathcal{O}^*((2-\varepsilon)^n)$ for any $\varepsilon > 0$, where n is the size of the universe.*

Proof. We shall present a reduction that for any constants $\alpha > 0$ and $q \geq 3$, takes an instance of q-SAT on n variables and transforms it in time polynomial in the input size into an equivalent instance of Hitting Set on a universe of size at most $(1+\alpha)n$. We note that in this reduction we will treat q and α as constants, and thus both the size of the family of the output Hitting Set instance and the running time of the reduction can depend exponentially on q and $1/\alpha$.

We first verify that the existence of such a reduction will provide us with the claimed lower bound. For the sake of contradiction, assume that Hitting Set indeed can be solved in time $\mathcal{O}^*(2^{\delta n})$ for some constant $\delta < 1$. Fix some $\alpha > 0$ so that $\alpha < 1$ and $\delta' := \delta(1+\alpha) < 1$. Assuming SETH, there exists a constant q such that q-SAT cannot be solved in time $\mathcal{O}^*(2^{\delta' n})$. Consider now the following algorithm for q-SAT: apply the presented reduction for parameters α and q, and then solve the resulting instance of Hitting Set using the assumed faster algorithm. Thus we could solve q-SAT in time $\mathcal{O}^*(2^{\delta(1+\alpha)n}) = \mathcal{O}^*(2^{\delta' n})$, which contradicts our choice of q.

We now proceed to the reduction itself. Let φ be the input instance of q-SAT, let $\texttt{Vars}$ be the variable set of φ, let $\texttt{Cls}$ be the clause set of φ, and let $n = |\texttt{Vars}|$, $m = |\texttt{Cls}|$. We fix some odd constant $p \geq 3$ such that $p + 2\lceil \log_2 p \rceil \leq (1+\alpha/3)p$. Let $p' = p + 2\lceil \log_2 p \rceil$; note that p' is also odd. Without loss of generality we will assume that n is divisible by p. This can be achieved by adding at most $p-1$ dummy variables not appearing in any clause, which increases the size of the variable set by multiplicative factor at most $1+\alpha/3$, unless the input instance is of constant size and we may solve it by brute-force.

Partition $\texttt{Vars}$ into n/p groups of variables $\texttt{Vars}_1, \texttt{Vars}_2, \ldots, \texttt{Vars}_{n/p}$, each of size p. For each group $\texttt{Vars}_i$ we create a set of elements U_i of size p'. Let $A_i^{\downarrow} = \binom{U_i}{\frac{p'-1}{2}}$ and $A_i^{\uparrow} = \binom{U_i}{\frac{p'+1}{2}}$. Note that $|A_i^{\downarrow}| = |A_i^{\uparrow}| \geq \frac{2^{p'}}{1+p'} \geq \frac{p^2}{1+p'} \cdot 2^p$. It can be readily checked that for $p \geq 3$ it holds that $p^2 \geq 1 + p'$, so $|A_i^{\downarrow}| = |A_i^{\uparrow}| \geq 2^p$. Therefore, let us fix an arbitrary injective mapping η_i from the set $\texttt{Vars}_i^{\{\bot,\top\}}$ of all the true/false assignments of the variables of $\texttt{Vars}_i$ to the set $A_i^{\downarrow}$. Let $B_i = A_i^{\downarrow} \setminus \mathrm{Im}(\eta_i)$, where $\mathrm{Im}(\eta_i)$ denotes the image of η_i.

Take any clause $C \in \texttt{Cls}$ and let $\texttt{Vars}_{j_{C,1}}, \texttt{Vars}_{j_{C,2}}, \ldots, \texttt{Vars}_{j_{C,r_C}}$ be all the variable groups that contain any variable appearing in C. Since C has at most q literals, we have that $r_C \leq q$. Hence the set $\texttt{Vars}_C := \bigcup_{t=1}^{r_C} \texttt{Vars}_{j_{C,t}}$ has cardinality at most $p'q$, which is a constant. Now consider all the true/false assignments of the variables of $\texttt{Vars}_C$, and let Z_C be the set of those assignments for which the clause C is *not* satisfied. For each $\zeta \in Z_C$ construct a set F_ζ^C by taking

$$F_\zeta^C = \bigcup_{t=1}^{r_C} \eta_{j_{C,t}}(\zeta|_{\texttt{Vars}_{j_{C,t}}}).$$

Now we construct the universe of the output Hitting Set instance by taking $U = \bigcup_{i=1}^{n/p} U_i$, while the family $\mathcal{F}$ is defined as follows. The family $\mathcal{F}$ comprises:

- all the sets contained in $A_i^{\uparrow}$, for each $i = 1, 2, \ldots, n/p$;
- all the sets contained in B_i, for each $i = 1, 2, \ldots, n/p$; and
- all the sets F_ζ^C for $C \in \texttt{Cls}$ and $\zeta \in Z_C$.

Finally, we set budget $k = n/p \cdot \frac{p'+1}{2}$ and output the instance $(U, \mathcal{F}, k)$ of Hitting Set. Note that $|U| \le (1 + \alpha/3)n$, which, together with the preliminary blow-up of the universe size due to introducing dummy variables, gives a total increase of the universe size by a multiplicative factor at most $(1 + \alpha/3)^2 \le 1 + \alpha$ (as $\alpha < 1$).

We now argue equivalence of the instances. Assume first that φ admits a satisfying assignment $\psi \colon \texttt{Vars} \to \{\bot, \top\}$. Construct a set X by taking

$$X = \bigcup_{i=1}^{n/p} U_i \setminus \eta_i(\psi|_{\texttt{Vars}_i}),$$

and observe that $|X| = k$. We claim that X is a solution to the Hitting Set instance $(U, \mathcal{F}, k)$. For any $i \in [n/p]$ and any $F \in A_i^{\uparrow}$ we have that $|F| = |X \cap U_i| = \frac{p'+1}{2}$ and $|U_i| = p'$, so $X \cap F$ must be nonempty. Moreover, we have that $U_i \setminus X \in \mathrm{Im}(\eta_i)$ by the definition of X, so $X \cap F \neq \emptyset$ for any $F \in B_i$.

Now consider any $C \in \texttt{Cls}$ and any $\zeta \in Z_C$. Since C is satisfied by the assignment ψ, there exists a variable x such that x belongs to some group $\texttt{Vars}_{j_{C,t}}$ and a literal of x satisfies C. Let $j = j_{C,t}$. Since ζ does not satisfy C, we in particular have that $\psi|_{\texttt{Vars}_j} \neq \zeta|_{\texttt{Vars}_j}$. Hence $F_\zeta^C \cap U_j = \eta_j(\zeta|_{\texttt{Vars}_j}) \neq \eta_j(\psi|_{\texttt{Vars}_j}) = U_j \setminus X$. Since $|X \cap U_j| = \frac{p'+1}{2}$, $|F_\zeta^C \cap U_j| = \frac{p'-1}{2}$, $|U_j| = p'$ and $F_\zeta^C \cap U_j \neq U_j \setminus X$, we have that $F_\zeta^C \cap X \cap U_j \neq \emptyset$ and we are done with the first implication.

Conversely, assume that there exists a set $X \subseteq U$, $|X| \le k$, that has a nonempty intersection with every set of $\mathcal{F}$. First observe that $|X \cap U_i| \ge \frac{p'+1}{2}$ for each $i \in [n/p]$, since otherwise $|U_i \setminus X| \ge \frac{p'+1}{2}$ and X would not hit any subset of $U_i \setminus X$ of size $\frac{p'+1}{2}$, but $A_i^{\uparrow} \subseteq \mathcal{F}$. Since $|X| \le k = n/p \cdot \frac{p'+1}{2}$ and sets $X \cap U_i$ are pairwise disjoint, we infer that $|X| = k$ and $|X \cap U_i| = \frac{p'+1}{2}$ for each $i \in [n/p]$. Moreover, since $X \cap F \neq \emptyset$ for each $F \in B_i$, we infer that $U_i \setminus X \in \mathrm{Im}(\eta_i)$. For each i let us construct an assignment $\psi_i \colon \texttt{Vars}_i \to \{\bot, \top\}$ by taking $\psi_i = \eta_i^{-1}(U_i \setminus X)$. We claim that $\psi = \bigcup_{i=1}^{n/p} \psi_i$ is a satisfying assignment for φ.

Take any clause $C \in \texttt{Cls}$, and assume for the sake of contradiction that ψ does not satisfy C. Let $\zeta = \psi|_{\texttt{Vars}_C}$; then we have that $\zeta \in Z_C$ by the definition of Z_C. Since X was a solution, we have that $X \cap F_\zeta^C \neq \emptyset$; hence $X \cap F_\zeta^C \cap U_j \neq \emptyset$ for some $j = j_{C,t}$. By the definition of F_ζ^C we have that

$F_\zeta^C \cap U_j = \eta_j(\zeta|_{\mathtt{Vars}_j}) = \eta_j(\psi|_{\mathtt{Vars}_j})$. On the other hand, by the definition of ψ we have that $U_j \setminus X = \eta_j(\psi|_{\mathtt{Vars}_j})$. Consequently $U_j \setminus X = F_\zeta^C \cap U_j$, and so $X \cap F_\zeta^C \cap U_j = \emptyset$, a contradiction. □

Observe that the proof of Theorem 14.35 substantially depends on the fact that the starting point is an instance of q-SAT for a *constant* q, instead of arbitrary CNF-SAT. This is used to be able to construct the families Z_C: if a clause could be arbitrarily large, then the size of the family Z_C would be exponential in n, and the reduction would use exponential time. Hence, the lower bound of Theorem 14.35 holds only under the assumption of SETH, and not under the weaker assumption that CNF-SAT cannot be solved in time $\mathcal{O}^*((2-\varepsilon)^n)$ for any $\varepsilon > 0$.

Lower bounds of a similar flavor to Theorem 14.35 can be proved for some other problems; examples include Set Splitting parameterized by the size of the universe, or NAE-SAT parameterized by the number of variables (see Exercise 14.16). Both these problems do not admit $\mathcal{O}^*((2-\varepsilon)^n)$ algorithms for any $\varepsilon > 0$, unless SETH fails.

Recall, however, that for the Hitting Set problem we have also a dynamic-programming algorithm working in time $\mathcal{O}^*(2^m)$, where m is the size of the family; equivalently, Set Cover can be solved in time $\mathcal{O}^*(2^n)$, where n is the size of the universe (see Theorem 6.1). Is the constant 2 in the base of the exponent optimal also in this case? Actually, we neither have a faster algorithm, nor are able to link non-existence of such to SETH. This motivates the following conjecture; q-Set Cover is the Set Cover problem with an additional assumption that every $F \in \mathcal{F}$ has cardinality at most q.

Conjecture 14.36 (Set Cover Conjecture, [112]). Let λ_q be the infimum of the set of constants c such that q-Set Cover can be solved in time $\mathcal{O}^*(2^{cn})$, where n is the size of the universe. Then $\lim_{q\to\infty} \lambda_q = 1$. In particular, there is no algorithm for the general Set Cover problem that runs in $\mathcal{O}^*((2-\varepsilon)^n)$ for any $\varepsilon > 0$.

It appears that several essentially tight lower bounds for classic parameterized problems can be established under the Set Cover Conjecture. Therefore, the question of finding links between SETH and the Set Cover Conjecture seems like a natural research direction.

Theorem 14.37 ([112]). *Unless the Set Cover Conjecture fails,*

- Steiner Tree *cannot be solved in time $\mathcal{O}^*((2-\varepsilon)^\ell)$ for any $\varepsilon > 0$, where ℓ is the target size of the tree;*
- Connected Vertex Cover *cannot be solved in time $\mathcal{O}^*((2-\varepsilon)^k)$ for any $\varepsilon > 0$, where k is the target size of the solution;*
- Subset Sum *cannot be solved in time $\mathcal{O}^*((2-\varepsilon)^m)$ for any $\varepsilon > 0$, where m is the number of bits of the encoding of the target sum t.*

14.5.2 Dynamic programming on treewidth

Recall that in Chapter 7 we have learned how to design dynamic-programming routines working on a tree decomposition of a graph. For many classic problems with simple formulations, the straightforward dynamic program has running time $\mathcal{O}^*(c^t)$ for some typically small constant c. For instance, for Independent Set one can obtain running time $\mathcal{O}^*(2^t)$, while for Dominating Set it is $\mathcal{O}^*(3^t)$. The intuitive feeling is that this running time should be optimal, since the natural space of states exactly corresponds to the information about a subtree of the tree decomposition that needs to be remembered for future computations. In other words, every state is meaningful on its own, and can lead to an optimal solution at the end. It is natural to ask whether under the assumption of SETH we can support this intuition by a formal lower bound.

This is indeed the case, and we can use a similar approach as that for Theorem 14.19 (we sketched the main ideas behind its proof in the discussion after the statement). In this section we will discuss in detail the case of the Independent Set problem parameterized by pathwidth, which turns out not to admit an $\mathcal{O}^*((2-\varepsilon)^p)$ algorithm for any $\varepsilon > 0$, assuming SETH. Before we proceed with a formal argumentation, let us discuss the intuition behind the approach. Note that a lower bound for pathwidth is even stronger than for treewidth.

We start the reduction from an arbitrary CNF-SAT instance on n variables and m clauses. The idea is to create a bundle of n very long paths $P^1, P^2, \ldots, P^n$ of even length, corresponding to variables $x_1, x_2, \ldots, x_n$. Assume for now that on each of these paths the solution is allowed to make one of two choices: incorporate into the independent set either all the odd-indexed vertices, or all the even-indexed vertices. Then for every clause we construct a clause verification gadget and attach it to some place of the bundle. The gadget is adjacent to paths corresponding to variables appearing in the clause, and the attachment points reflect whether the variable's appearance is positive or negative (glance at Fig. 14.9 for an idea of how this will be technically carried out). As usual, the role of the clause gadget is to verify that the clause is satisfied. Satisfaction of the clause corresponds to the condition that at least one of the attachment points of the clause gadget needs to be *not* chosen into the constructed independent set; hence the clause gadget needs to have the following property: the behavior inside the gadget can be set optimally if and only if at least one of the attachment points is free. Fortunately, it is possible to construct a gadget with exactly this property, and moreover the gadget has constant pathwidth, so it does not increase much the width of the whole construction. See Fig. 14.8 and Claim 14.39 for the implementation.

One technical problem that we still need to overcome is the first technical assumption about the choices the solution makes on the paths P^i. It is namely not true that on a path of even length there are only two maximum-size

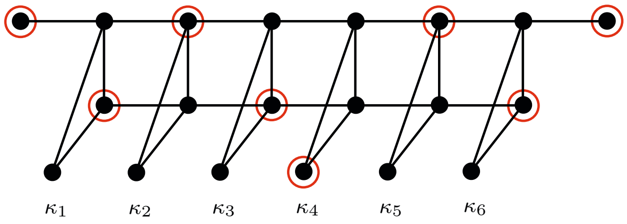


Fig. 14.8: Gadget M_6 with maximum independent set Z_4 (i.e., the one with $Z_4 \cap \mathcal{I} = \{\kappa_4\}$) highlighted

independent sets: the odd-indexed vertices and the even-indexed vertices. The solution can first start with picking only odd-indexed vertices, then make a gap of two vertices, and continue further with even-indexed vertices. Thus, on each path there can be one "cheat" where the solution flips from odd indices to even indices. The solution to this problem is a remarkably simple trick that is commonly used in similar reductions. We namely repeat the whole sequence of clause gadgets $n+1$ times, which ensures that at most n copies are spoiled by possible cheats, and hence at least one of the copies is attached to area where no cheat happens, and hence the behavior of the solution on the paths P^i correctly encodes some satisfying assignment of the variable set.

We now proceed to the formal argumentation.

Theorem 14.38. *Suppose there exists a constant $\varepsilon > 0$ and an algorithm that, given an integer k and a graph G with its path decomposition of width p, checks whether G admits an independent set of size k in time $\mathcal{O}^*((2-\varepsilon)^p)$. Then* CNF-SAT *can be solved in time $\mathcal{O}^*((2-\varepsilon)^n)$, and in particular SETH fails.*

Proof. We provide a polynomial-time reduction that takes a CNF-SAT instance φ on n variables, and constructs an instance (G, k) of Independent Set together with a path decomposition of G of width $n + \mathcal{O}(1)$. Provided that the supposed faster algorithm for Independent Set existed, we could compose the reduction with this algorithm and solve CNF-SAT in time $\mathcal{O}^*((2-\varepsilon)^n)$.

We first make an auxiliary gadget construction that will be useful in the reduction. Let M_r be the graph depicted in Fig. 14.8, consisting of a sequence of r triangles with each two consecutive ones connected by two edges, and two pendant vertices attached to the first and the last triangle. By $\kappa_1, \kappa_2, \ldots, \kappa_r$ we denote the degree-2 vertices in the triangles, as shown on Fig. 14.8, and we also denote $\mathcal{I} = \{\kappa_1, \kappa_2, \ldots, \kappa_r\}$. The following claim, whose proof is left to the reader as Exercise 14.14, summarizes all the properties of M_r that we will need.

Claim 14.39. *Assume r is even. Then the maximum size of an independent set in M_r is equal to $r + 2$, and each independent set of this size has a* *nonempty intersection with $\mathcal{I}$. Moreover, for every $a = 1, 2, \ldots, r$ there exists an independent set Z_a in M_r such that $|Z_a| = r + 2$ and $Z_a \cap \mathcal{I} = \{\kappa_a\}$. Finally, the graph M_r has pathwidth at most* 3.

Thus, graph M_r can be thought of as a gadget encoding a single 1-in-r choice. A similar behavior could be modelled by taking M_r to be a clique on r vertices, but the crucial property is that M_r has constant pathwidth. In the reduction we will use multiple copies of M_r for various values of r. For a copy H of M_r, by $\kappa_a(H)$, $Z_a(H)$, and $\mathcal{I}(H)$ we denote the respective objects in H.

We now proceed to the construction itself. Let $\texttt{Vars} = \{x_1, x_2, \ldots, x_n\}$, $\texttt{Cls} = \{C_1, C_2, \ldots, C_m\}$ be the variable and the clause set of φ, respectively. For $C \in \texttt{Cls}$, by $|C|$ we will denote the number of literals appearing in C, and let us enumerate them arbitrary as $\ell_1^C, \ell_2^C, \ldots, \ell_{|C|}^C$. By duplicating some literals if necessary we will assume that $|C|$ is even for each $C \in \texttt{Cls}$. Without loss of generality we assume that no clause of $\texttt{Cls}$ contains two opposite literals of the same variable, since such clauses are satisfied in every assignment.

We first construct an auxiliary graph G_0 as follows. Introduce n paths $P^1, P^2, \ldots, P^n$, each having $2m$ vertices. Let $p_0^i, p_1^i, \ldots, p_{2m-1}^i$ be the vertices of P^i in the order of their appearance. For each clause C_j, construct a graph H_j being a copy of $M_{|C_j|}$. For every $a = 1, 2, \ldots, |C_j|$, let x_{i_a} be the variable appearing in literal $\ell_a^{C_j}$. If $\ell_a^{C_j} = x_{i_a}$ then connect $\kappa_a(H_j)$ with $p_{2j-2}^{i_a}$, and if $\ell_a^{C_j} = \neg x_{i_a}$ then connect $\kappa_a(H_j)$ with $p_{2j-1}^{i_a}$. This concludes the construction of the graph G_0.

To construct the output graph G, we create $n + 1$ copies of the graph G_0; denote them by $G_1, G_2, G_3, \ldots, G_{n+1}$. We follow the same convention that an object from copy G_a, for $a = 1, 2, \ldots, n + 1$, is denoted by putting G_a in brackets after the object's name in G_0. Connect the copies sequentially as follows: for each $a = 1, 2, \ldots, n$ and each $i = 1, 2, \ldots, n$, connect vertices $p_{2m-1}^i(G_a)$ and $p_0^i(G_{a+1})$. This concludes that construction of graph G. The following claim, whose proof is left to the reader as Exercise 14.15, verifies that G indeed has low pathwidth.

Claim 14.40. *G admits a path decomposition of width at most $n + 3$, which furthermore can be computed in polynomial time.*

Finally, we set the expected size of the independent set to be $k = (n + 1) \cdot \left(mn + \sum_{j=1}^m (|C_j| + 2)\right)$. It now remains to argue that G admits an independent set of size k if and only if φ is satisfiable.

Assume first that φ admits a satisfying assignment $\psi\colon \texttt{Vars} \to \{\bot, \top\}$. We first construct an independent set X_0 of size $mn + \sum_{j=1}^m (|C_j| + 2)$ in G_0. For every $i = 1, \ldots, n$, if $\psi(x_i) = \bot$ then in X_0 we include all vertices p_b^i with even b, and if $\psi(x_i) = \top$ then we include all vertices p_b^i with odd b. For each clause C_j fix an index a_j such that the literal $\ell_{a_j}^{C_j}$ satisfies C_j in the

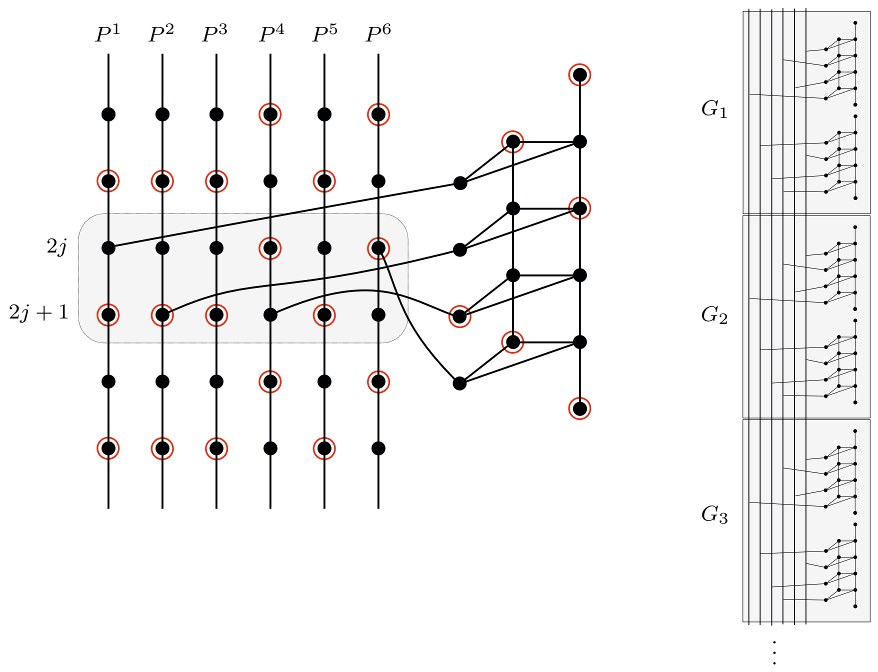


Fig. 14.9: Constructions in the proof of Theorem 14.38. Left side: Connections adjacent to a clause gadget H_j for $C_j = x_1 \vee \neg x_2 \vee \neg x_4 \vee x_6$, together with the constructed independent set for variable assignment $\psi(x_4) = \psi(x_6) = \bot$ and $\psi(x_1) = \psi(x_2) = \psi(x_3) = \psi(x_5) = \top$. Note that the behavior in the gadget H_j has been set according to the choice of literal $\neg x_4$ as the literal satisfying C_j, but we could also choose literal x_1. Right side: Arranging copies of G_0 into G

assignment ψ; let x_{i_j} be the variable appearing in this literal. Then include into X_0 the independent set $Z_{a_j}(H_j)$ given by Claim 14.39.

Clearly $|X_0| = mn + \sum_{j=1}^{m}(|C_j|+2)$. We now verify that X_0 is independent in G_0. From the construction it directly follows that the vertices of X lying on paths P^i are pairwise nonadjacent, and also vertices lying in different gadgets H_j are nonadjacent. Vertices of X_0 lying in the same gadget H_j are nonadjacent by Claim 14.39. It remains to verify that no vertex of X_0 belonging to a gadget H_j can see any vertex of X_0 lying on any path P^i. The only vertices of H_j adjacent to the vertices outside H_j are vertices of $\mathcal{I}(H_j)$, and we have $X_0 \cap \mathcal{I}(H_j) = \{\kappa_{a_j}(H_j)\}$. The only neighbor of $\kappa_{a_j}(H_j)$ outside H_j is the vertex $p^{i_j}_{2j-2}$ if $\ell^{C_j}_{a_j} = x_{i_j}$, or $p^{i_j}_{2j-1}$ if $\ell^{C_j}_{a_j} = \neg x_{i_j}$. Since $\ell^{C_j}_{a_j}$ satisfies C_j in the assignment ψ, it follows from the definition of X_0 that precisely this vertex on P^{i_j} is not chosen to X_0. So indeed X_0 is an independent set in G_0.

Let $X_1, X_2, \ldots, X_{n+1}$ be the images of X_0 in the copies $G_1, G_2, \ldots, G_{n+1}$, and let $X = \bigcup_{i=1}^{n+1} X_i$. Clearly $|X| = k$, and it follows easily from the construction that X is independent: for every edge $p^i_{2m-1}(G_a)p^i_0(G_{a+1})$ connecting each two consecutive copies, exactly one of the assertions holds: $p^i_{2m-1}(G_a) \in X_a$ or $p^i_0(G_{a+1}) \in X_{a+1}$.

We are left with the converse implication. For $i = 1, 2, \ldots, n$, let $\hat{P}^i = G[\bigcup_{a=1}^{n+1} V(P^i(G_a))]$. Observe that $\hat{P}^i$ is a path on $(n+1) \cdot 2m$ vertices, and thus the maximum size of an independent set in $\hat{P}^i$ is $(n+1)m$. Assume that G admits an independent set X of size k. We then have that $|X \cap V(\hat{P}^i)| \leq (n+1)m$ for each $i = 1, 2, \ldots, n$, and by Claim 14.39 we have that $|X \cap V(H_j(G_a))| \leq |C_j| + 2$ for each $j = 1, 2, \ldots, m$ and $a = 1, 2, \ldots, n+1$. Since the sum of these upper bounds through all the paths $\hat{P}^i$ and gadgets $H_j(G_a)$ is exactly equal to k, we infer that all these inequalities are in fact equalities: $|X \cap V(\hat{P}^i)| = (n+1)m$ for each $i = 1, 2, \ldots, n$, and $|X \cap V(H_j(G_a))| = |C_j|+2$ for each $j = 1, 2, \ldots, m$ and $a = 1, 2, \ldots, n+1$.

Examine now one path $\hat{P}^i$ and observe that any independent set of size $(n+1)m$ in $\hat{P}^i$ can leave at most one pair of consecutive vertices on $\hat{P}^i$ not chosen. We will say that the copy G_a is *spoiled by the path* $\hat{P}^i$ if $V(P^i(G_a)) \setminus X$ contains a pair of consecutive vertices on $P^i(G_a)$. It then follows that each path $\hat{P}^i$ can spoil at most one copy G_a, so there exists at least one copy G_{a_0} that is not spoiled at all. Recall that G_{a_0} is a copy of G_0; let $X_0 \subseteq V(G_0)$ be the set of originals of the vertices of the set $V(G_{a_0}) \cap X$. We then have that X_0 contains every second vertex on each path P^i, beginning with p^i_0 and finishing on p^i_{2m-2}, or beginning with p^i_1 and finishing on p^i_{2m-1}. Moreover, $X_0 \cap V(H_j)$ is a maximum-size independent set in each gadget H_j. By Claim 14.39 it follows that for each $j = 1, 2, \ldots, m$ there exists an index a_j such that $\kappa_{a_j}(H_j) \in X_0$.

Let $\psi\colon \texttt{Vars} \to \{\bot, \top\}$ be an assignment defined as follows: $\psi(x_i) = \bot$ if $X_0 \cap V(P^i) = \{p^i_0, p^i_2, \ldots, p^i_{2m-2}\}$, and $\psi(x_i) = \top$ if $X_0 \cap V(P^i) = \{p^i_1, p^i_3, \ldots, p^i_{2m-1}\}$. Consider now any clause C_j, and let x_{i_j} be the variable appearing in the literal $\ell^{C_j}_{a_j}$. Recall that $\kappa_{a_j}(H_j) \in X_0$, and it is adjacent to $p^{i_j}_{2j-2}$ if $\ell^{C_j}_{a_j} = x_{i_j}$, or to $p^{i_j}_{2j-1}$ if $\ell^{C_j}_{a_j} = \neg x_{i_j}$. As X_0 is independent, we infer that this vertex on P^{i_j} does not belong to X_0. Hence $\psi(x_{i_j}) = \top$ if $\ell^{C_j}_{a_j} = x_{i_j}$ and $\psi(x_{i_j}) = \bot$ if $\ell^{C_j}_{a_j} = \neg x_{i_j}$, by the definition of ψ. In both cases we can conclude that the literal $\ell^{C_j}_{a_j}$ satisfies C_j in the assignment ψ. Since C_j was chosen arbitrarily, we infer that ψ satisfies φ. $\square$

Under SETH we can also prove the optimality of known dynamic-programming routines for many other problems. The following theorem provides a handful of the most important examples (note that the lower bound for q-Coloring is given only for tree decompositions and not path decompositions).

Theorem 14.41 ([113, 118, 325]). *Assume that* CNF-SAT *cannot be solved in time* $\mathcal{O}^*((2-\varepsilon')^n)$ *for any* $\varepsilon' > 0$. *Then for every* $\varepsilon > 0$ *the following holds (p/t is the width of a given path/tree decomposition of the input graph):*

- Dominating Set *cannot be solved in time* $\mathcal{O}^*((3-\varepsilon)^p)$*;*
- Odd Cycle Transversal *cannot be solved in time* $\mathcal{O}^*((3-\varepsilon)^p)$*;*
- q-Coloring *cannot be solved in time* $\mathcal{O}^*((q-\varepsilon)^t)$*, for any constant* $q \geq 3$*;*
- Steiner Tree *cannot be solved in time* $\mathcal{O}^*((3-\varepsilon)^p)$*;*
- Feedback Vertex Set *cannot be solved in time* $\mathcal{O}^*((3-\varepsilon)^p)$*;*
- Connected Vertex Cover *cannot be solved in time* $\mathcal{O}^*((3-\varepsilon)^p)$*;*
- Hamiltonian Cycle *cannot be solved in time* $\mathcal{O}^*((2+\sqrt{2}-\varepsilon)^p)$*.*

Note that for the last three points, naive dynamic-programming routines work in slightly super-exponential time, which can be improved to single exponential using the techniques of Chapter 11. In particular, the lower bounds given by Theorem 14.41 for Steiner Tree, Feedback Vertex Set, and Connected Vertex Cover match the randomized upper bounds given by the Cut & Count technique. Therefore, in these examples the bases of exponents in running times given by Cut & Count are not just coincidental numbers stemming from the technique, but match inherent properties of the problems themselves.

Observe also that all the lower bounds mentioned in Theorem 14.41 prove that the optimal base of the exponent is a number larger than 2, in most cases 3. In the case of Independent Set the reduction intuitively embedded the 2^n search space into 2^p states of the dynamic program, which was very convenient as we could use a one-to-one correspondence between variables x_i and paths P_i. In the case of, say, Dominating Set, the reduction should embed 2^n possible variable assignments into $3^{n'}$ states of the dynamic program, where $n' = n \cdot \log_3 2$. Hence, in the resulting decomposition we cannot have one element of a bag per each variable, but every element of a bag has to bear information about *more* than one variable; more precisely, about $\log_2 3 \approx 1.585$ variables. Needless to say, this creates technical difficulties in the proof.

These difficulties can be overcome using a similar technique as in the proof of the lower bound for Hitting Set parameterized by the size of the universe (Theorem 14.35). We namely partition the variables into groups of size q for some large constant q, and assign $q' = \left\lceil \frac{q}{\log_2 3} \right\rceil$ vertices of a bag to each group. In this manner, for each group we can encode all $2^q \leq 3^{q'}$ variable assignments as possible interactions of the solution with the assigned subset of the bag. Also, the pathwidth of the resulting graph will be approximately $n \cdot \frac{q'}{q} \leq \frac{n}{\log_2 3} \cdot (1+\alpha)$ for some α converging to zero as q grows to infinity. As in the case of Theorem 14.35, it is somewhat problematic to extract an assignment of a particular clause from a group in order to check it against a clause gadget. This is done via an additional layer of *group gadgets* that are responsible for decoding the assignment for a group; we refer the reader to [118, 325] for details.

14.5.3 A refined lower bound for Dominating Set

In Section 14.4 we have seen that ETH can be used to provide estimates of the magnitude of the function of k appearing in the exponent of n for W[1]-hard problems. In particular, in Theorem 14.23 we have proved that, unless ETH fails, Dominating Set does not admit an algorithm with running time $f(k) \cdot n^{o(k)}$ for any computable function f. Again, it appears that having assumed SETH we can establish even more precise lower bounds. This is exemplified in the following theorem.

Theorem 14.42. *Unless* CNF-SAT *can be solved in time* $\mathcal{O}^*((2-\varepsilon')^n)$ *for some* $\varepsilon' > 0$*, there do not exist constants* $\varepsilon > 0$*,* $k \geq 3$ *and an algorithm solving* Dominating Set *on instances with parameter equal to* k *that runs in time* $\mathcal{O}(N^{k-\varepsilon})$*, where* N *is the number of vertices of the input graph.*

Proof. For the sake of contradiction assume that such constants $\varepsilon > 0$, $k \geq 3$ and such an algorithm does exist. We are going to present an algorithm for CNF-SAT working in time $\mathcal{O}^*(2^{\delta n})$ for $\delta = 1 - \frac{\varepsilon}{k} < 1$.

Let φ be the input instance of CNF-SAT, let $\texttt{Vars}$ be the variable set of φ, and let n, m be the numbers of variables and clauses of φ, respectively. Partition $\texttt{Vars}$ into k parts $\texttt{Vars}_1, \texttt{Vars}_2, \ldots, \texttt{Vars}_k$, each of size at most $\lceil n/k \rceil$. For every part $\texttt{Vars}_i$ and each of $2^{|\texttt{Vars}_i|}$ possible assignments $\psi \colon \texttt{Vars}_i \to \{\bot, \top\}$, create a vertex v^i_ψ. For each $i \in [k]$, create an additional vertex w^i and let V_i be the set of vertices v^i_ψ created for the part $\texttt{Vars}_i$ plus the vertex w^i. Make V_i into a clique. For each clause C of φ, create a vertex u_C. For every $i \in [k]$ and every assignment ψ of $\texttt{Vars}_i$, create an edge $u_C v^i_\psi$ if and only if $\texttt{Vars}_i$ contains a variable x_i whose literal appears in C, and moreover ψ evaluates x_i so that this literal satisfies C. Let G be the resulting graph and let $N = |V(G)|$. Observe that $N \leq k \cdot (1 + 2^{\lceil n/k \rceil}) + m = \mathcal{O}^*(2^{n/k})$, so G contains at most $\binom{N}{2} \leq \mathcal{O}^*(2^{2n/k}) \leq \mathcal{O}^*(2^{2n/3})$ edges. Moreover, G can be constructed in time $\mathcal{O}^*(2^{2n/k}) \leq \mathcal{O}^*(2^{2n/3})$.

The intuition is as follows. Vertices w_i ensure that in the optimum solution we will need to pick exactly one vertex from every clique V_i. The vertex picked from V_i encodes the assignment of variables of $\texttt{Vars}_i$, and domination of each vertex u_C checks satisfaction of the corresponding clause C. Thus, the space of all the $\prod_{i=1}^{k} |V_i|$ k-tuples that are candidates for a dominating set in G corresponds to the space of all the 2^n variable assignments for the input formula φ.

We now claim that G admits a dominating set of size k if and only if φ is satisfiable. If φ admits a satisfying assignment $\psi\colon \mathtt{Vars} \to \{\bot, \top\}$, then let us construct a set $X = \{v^i_{\psi|_{V_i}} : i \in [k]\}$. Observe that $|X| = k$ and X is a dominating set in G: each clique V_i is dominated by $v^i_{\psi|_{V_i}}$, while each vertex u_C is dominated by the vertex $v^i_{\psi|_{V_i}}$ for any part V_i that contains a variable whose literal satisfies C.

Conversely, let X be a dominating set of size k in G. Since every vertex w^i is dominated by X and $N[w_i] = V_i$, we infer that X contains exactly one vertex from each clique V_i, and no other vertex. Let us define assignment $\psi_i\colon \mathtt{Vars}_i \to \{\bot, \top\}$ for each $i \in [k]$ as follows: if $X \cap V_i = \{w_i\}$ then put an arbitrary assignment as ψ_i, and otherwise ψ_i is such that $X \cap V_i = \{v^i_{\psi_i}\}$. Let $\psi = \bigcup_{i\in[k]} \psi_i$; we claim that the assignment ψ satisfies the input formula φ. Take any clause C and examine the vertex u_C. Since $u_C \notin X$, u_C is dominated by a vertex $u^i_{\psi_i} \in X$ for some $i \in [k]$. By the definition of the edge set of G we infer that $\mathtt{Vars}_i$ contains a variable whose literal is present in C, and which is evaluated under the assignment ψ_i so that C becomes satisfied. Since $\psi_i = \psi|_{\mathtt{Vars}_i}$, the same literal satisfies C in the assignment ψ.

To conclude the proof, observe that we could solve the CNF-SAT problem by pipelining the presented reduction with the supposed algorithm for Dominating Set on instances with parameter k. As argued, the reduction runs in time $\mathcal{O}^*(2^{2n/3})$, while the application of the algorithm for Dominating Set takes time $\mathcal{O}(N^{k-\varepsilon}) = 2^{n\cdot(1-\varepsilon/k)} \cdot (n+m)^{\mathcal{O}(k)}$. Since $\varepsilon > 0$ and k is a constant, this means that CNF-SAT can be solved in time $\mathcal{O}^*(2^{\delta n})$ for $\delta = 1 - \frac{\varepsilon}{k} < 1$. $\square$

Note that the trivial algorithm for Dominating Set that verifies all k-tuples of vertices runs in time $\mathcal{O}(N^k \cdot (N+M))$ on graphs with N vertices and M edges. However, it is known that this algorithm can be slightly improved, to running time $\mathcal{O}(N^{k+o_k(1)})$. Thus, Theorem 14.42 essentially states that it is hard to break the barrier of the cardinality of the brute-force search space.

Exercises

14.1 (✎). Prove the randomized analogues of Theorems 14.4 and 14.5:

(a) Assuming randomized ETH, there exists a constant $c > 0$ such that no randomized two-sided error algorithm for 3-SAT can achieve running time $\mathcal{O}^*(2^{c(n+m)})$.
(b) Randomized SETH implies randomized ETH.

14.2. Prove Theorem 14.6, that is, verify that the following problems admit linear reductions from 3-SAT:

(a) Vertex Cover (✎),
(b) Dominating Set (✎),
(c) Feedback Vertex Set (✎),
(d) 3-Coloring,

(e) Hamiltonian Cycle.

14.3. Prove Theorem 14.9, that is, verify that the following problems admit reductions with linear output size from the Planar 3-SAT problem (assuming the existence of a variable circuit, if needed):

(a) Planar Vertex Cover,
(b) Planar Dominating Set,
(c) Planar Feedback Vertex Set,
(d) Planar 3-Coloring (☠),
(e) Planar Hamiltonian Cycle (☠).

14.4 (✎). Using Theorem 14.13 as a black box, prove that the existence of an $\mathcal{O}^*(2^{o(k \log k)})$ algorithm for $k \times k$ Permutation Clique would contradict randomized ETH.

14.5. In the $2k \times 2k$ Bipartite Permutation Independent Set we are given a graph G with the vertex set $[2k] \times [2k]$, where every edge is between $I_1 = [k] \times [k]$ and $I_2 = ([2k] \setminus [k]) \times ([2k] \setminus [k])$. The question is whether there exists an independent set $X \subseteq I_1 \cup I_2$ in G that induces a permutation of $[2k]$. Construct a polynomial-time reduction that takes an instance of $k \times k$ Permutation Clique and outputs an equivalent instance of $2k \times 2k$ Bipartite Permutation Independent Set. Infer that $2k \times 2k$ Bipartite Permutation Independent Set does not admit an $\mathcal{O}^*(2^{o(k \log k)})$ algorithm unless ETH fails.

14.6. Construct a polynomial-time reduction that takes an instance of $2k \times 2k$ Bipartite Permutation Independent Set and outputs an equivalent instance of $2k \times 2k$ Permutation Hitting Set with thin sets. Infer that $k \times k$ Permutation Hitting Set with thin sets does not admit an $\mathcal{O}^*(2^{o(k \log k)})$ algorithm unless ETH fails, which is the first half of Theorem 14.16.

14.7 (✎). Construct a polynomial-time reduction that takes an instance of $k \times k$ Permutation Hitting Set with thin sets and outputs an equivalent instance of $k \times k$ Hitting Set with thin sets. Infer that $k \times k$ Hitting Set with thin sets does not admit an $\mathcal{O}^*(2^{o(k \log k)})$ algorithm unless ETH fails, which is the second half of Theorem 14.16.

14.8 (☠). In the Maximum Cycle Cover problem the input consists of a graph G and an integer k, and the question is whether one can find a collection of *at least* k vertex-disjoint cycles in the graph so that every vertex belongs to exactly one of these cycles. Prove that unless ETH fails, Maximum Cycle Cover does not admit an FPT algorithm with running time $\mathcal{O}^*(2^{o(p \log p)})$, where p is the width of a given path decomposition of G.

14.9. In the Component Order Integrity problem we are given a graph G and two integers k and ℓ. The task is to determine whether there exists a set of vertices X of cardinality at most k such that every connected component of $G \setminus X$ contains at most ℓ vertices. Prove that Component Order Integrity admits an FPT algorithm with running time $\mathcal{O}^*(2^{\mathcal{O}(k \log \ell)})$, but the existence of an algorithm with running time $\mathcal{O}^*(2^{o(k \log \ell)})$ would contradict ETH.

14.10 (☠). Using Theorem 14.20 as a black box, prove the following statement: There exists a universal constant $\delta > 0$ such that, unless $\mathrm{P} \neq \mathrm{NP}$, there is no constant λ and a kernelization algorithm for Edge Clique Cover that reduces the number of vertices to at most $\lambda \cdot 2^{\delta k}$. In particular, Edge Clique Cover does not admit a $2^{o(k)}$ kernel unless $\mathrm{P} = \mathrm{NP}$.

14.11 (✎). Modify the construction in the proof of Theorem 13.31 to get a parameterized reduction from Subgraph Isomorphism to Odd Set where the parameter of the constructed instance is linear in the number of edges of H in the Subgraph Isomorphism instance.

14.12 (✎). Modify the construction in the proof of Theorem 13.33 to get a parameterized reduction from Subgraph Isomorphism to Strongly Connected Steiner Subgraph where the parameter of the constructed instance is linear in the number of edges of H in the Subgraph Isomorphism instance.

14.13. The input of the Unit Square Independent Set problem is a set of axis-parallel unit squares in the plane and an integer k; the task is to select a set of k pairwise disjoint squares. Show that Unit Square Independent Set is W[1]-hard and, unless ETH fails, it has no $f(k)n^{o(\sqrt{k})}$-time algorithm for any computable function f.

14.14 (✎). Prove Claim 14.39.

14.15 (✎). Prove Claim 14.40.

14.16. In the NAE-SAT problem the input consists of a formula φ in the conjunctive normal form (CNF), and the question is whether there exists an assignment of the variables such that in each clause of φ there is at least one literal evaluated to true and at least one evaluated to false. Prove that, unless SETH fails, for any $\varepsilon > 0$ there does not exist an algorithm that solves NAE-SAT in time $\mathcal{O}^*((2-\varepsilon)^n)$, where n is the number of variables appearing in φ.

14.17. Show that the standard reduction from the q-SAT problem to Independent Set proves that unless SETH fails, for any $\varepsilon > 0$ there is no $\mathcal{O}^*((\sqrt{2}-\varepsilon)^p)$-time algorithm for the Independent Set problem given a path decomposition of width p.

14.18 (☠). In the Connected Dominating Set problem the input consists of a graph G and an integer k, and the question is whether there exists a subset $X \subseteq V(G)$ of at most k vertices such that $G[X]$ is connected and $N[X] = V(G)$. Prove that, unless SETH fails, for any $\varepsilon > 0$ there does not exist an algorithm that, given a Connected Dominating Set instance (G,k) together with a path decomposition of G of width p, solves (G,k) in time $\mathcal{O}^*((4-\varepsilon)^p)$.

14.19. Prove that, given a graph G, one can check whether G has a dominating set of size at most 2 in $\mathcal{O}(|V(G)|^\omega)$ time, where ω is the matrix multiplication exponent.

Hints

14.1 Follow the lines of the proofs of Theorems 14.4 and 14.5, but, whenever some algorithm $\mathcal{A}_c$ is used, repeat its usage many times in order to reduce the error probability.

14.2 Look at classic NP-hardness reductions for these problems that can be found in the literature.

14.3

(a) Replace variables and clauses by gadgets, but watch out not to spoil planarity around a variable gadget when edges corresponding to positive and negative occurrences interleave around the vertex corresponding to the variable. To fix this, use a cycle of length $2p$ for a variable gadget, where p is the number of occurrences of the variable.
(b) Design a similar construction as for Planar Vertex Cover, but use a cycle of length $3p$.
(c) Modify slightly the construction for Planar Vertex Cover by applying the standard reduction from Vertex Cover to Feedback Vertex Set at the end.

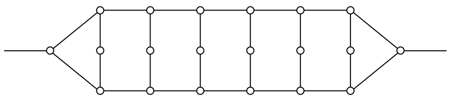


Fig. 14.10: The ladder graph

(d) Use the variable circuit $x_1 - x_2 - \ldots - x_n - x_1$ to propagate colors chosen to represent true and false assignment. Carefully design a planar variable gadget of size $\Theta(p)$, where p is the number of occurrences of the variable. Make sure that your gadget both propagates colors representing true and false to the next variable, and also makes use of them in the check against the clause gadget. The clause gadget can be of constant size.

(e) Use the variable circuit $x_1 - x_2 - \ldots - x_n - x_1$ as the 'spine' of the intended Hamiltonian cycle. The Hamiltonian cycle should be able to make small detours from the spine into clause gadgets, and all clause gadgets can be visited if and only if the input formula is satisfiable. Use the ladder graph depicted in Fig. 14.10 as your variable gadget. In the proof of correctness you will need to be very careful when arguing that the Hamiltonian cycle cannot misbehave and jump between distant variable gadgets via clause gadgets.

14.4 Start with the randomized version of Theorems 14.4, i.e., Exercise 14.1.(a), and verify that the chain of reductions to $k \times k$ Clique via 3-Coloring gives also a randomized lower bound.

14.5 By taking the complement of the graph, you may start with $k \times k$ Permutation Independent Set. In the constructed instance of $2k \times 2k$ Bipartite Permutation Independent Set, the only possible independent sets should induce the same permutation pattern on I_1 and on I_2, and this permutation pattern should correspond to a solution to the input instance of $k \times k$ Permutation Independent Set.

14.6 Encode the constraint imposed by each edge of G using one thin set. Remember to force the solution to be contained in $I_1 \cup I_2$, which can also be done using thin sets.

14.7 The requirement that each column must contain at least one vertex of the solution can easily be encoded using thin sets.

14.8 Follow the strategy sketched in the paragraph after Theorem 14.19. Start from $k \times k$ Hitting Set with thin sets and encode choice of the solution as a choice of a perfect matching in a $K_{k,k}$-biclique in the first bag. The main spine of the decomposition should be formed by $2k$ paths originating in the $2k$ vertices of the biclique; along these paths k cycles should 'travel' at each point. A gadget verifying a set should allow closing one cycle and beginning a new one in its place if and only if the set is hit by the solution.

14.9 For the positive result, apply a simple branching strategy. For the negative result, start the reduction from $k \times k$ Clique and have $\ell = k^{\mathcal{O}(1)}$.

14.10 Since Edge Clique Cover is in NP, there exists a polynomial-time reduction from Edge Clique Cover to 3-SAT. Consider the composition of (a) the reduction of Theorem 14.20, (b) a supposed kernelization algorithm, and (c) the reduction back from Edge Clique Cover to 3-SAT. Calculate an upper bound on the size of the output instance in terms of the input one.

14.11 Let $v_1, \ldots, v_{|V(H)|}$ be the vertices of H. We include $E_{i,j}$ in the universe only if v_i and v_j are adjacent. The parameter is set to $k' = k + |E(H)|$.

14.12 Let $v_1, \ldots, v_{|V(H)|}$ be the vertices of H. We include the vertex $y_{i,j}$ and the set of vertices $E'_{i,j}$ in G' only if v_i and v_j are adjacent. The parameter is set to $k' = k + 1 + 2|E(H)| = |K| + k + |E(H)|$.

14.13 The reduction in the proof of Theorem 14.34 does not work with squares: the squares at $(x + a\varepsilon, y + a\varepsilon)$ and $(x + 1 + a\varepsilon, y + 1 + a\varepsilon)$ can intersect at their corners, which was not an issue for unit disks. We show two possible workarounds.

Solution 1: Represent each pair $(a, b) \in S[x, y]$ with the five squares at $(3x+a\varepsilon, 3y+a\varepsilon)$, $(3x + 1 + a\varepsilon, 3y + a\varepsilon)$, $(3x + a\varepsilon, 3y + 1 + a\varepsilon)$, $(3x - 1 + a\varepsilon, 3y + a\varepsilon)$, $(3x + a\varepsilon, 3y - 1 + a\varepsilon)$. Set the new parameter to $k' = 5k$.

Solution 2: Represent each pair $(a, b) \in S[x, y]$ with a square at $(x+a\varepsilon-y/2, y+a\varepsilon+x/2)$.

14.17 Create a clique of size q for each clause and a clique of size 2 for each variable. Connect each vertex of each clique of size q with its negation in the corresponding clique of size 2. Prove that the resulting graph has pathwidth at most $2n + \mathcal{O}(1)$ for constant q.

14.18 As in the lower bound for Independent Set, the main part of the construction consists of roughly p long "strings" that transmit a fixed choice of an assignment ψ. However, as we aim at s $(4 - \varepsilon)^p$ lower bound, each "string" needs to transmit information about the assignment of *two* variables. In the Connected Dominating Set problem, the four states of a vertex on a string may be chosen as follows: (1) not in the solution, but dominated; (2) not in the solution, and not yet dominated; (3) in the solution, and already connected to some "root vertex" of the constructed solution"; and (4) in the solution, but not yet connected to the "root vertex".

Bibliographic notes

The $\mathcal{O}^*(2^{0.387n})$ algorithm for 3-SAT was given by Hertli [261], building upon the earlier work of Paturi, Pudlák, Saks, and Zane [382]. ETH and SETH were first introduced in the work of Impagliazzo and Paturi [272], which built upon earlier work of Impagliazzo, Paturi and Zane [273]. In particular, the sparsification lemma (Theorem 14.3) together with its most important corollary, Theorem 14.4, were proved in [273]. The work of Impagliazzo, Paturi and Zane [273] also describes in detail and with larger generality the consequences of ETH that can be obtained via linear reductions; Theorem 14.6 is a direct consequence of this discussion. The applicability of ETH to planar graph problems and to basic parameterized problems was first observed by Cai and Juedes [68]. In particular, Theorems 14.9 and 14.11 should be attributed to them.

The technique of proving NP-hardness of problems on planar graphs via Planar 3-SAT was proposed by Lichtenstein [320], and it unified several results that were obtained before. In particular, Exercise 14.3 is motivated by the work of Lichtenstein, apart from the NP-hardness of Planar 3-Coloring. The classic reduction for this problem was first given by Stockmeyer [419], and proceeds directly from general 3-Coloring by embedding the graph on the plane and introducing a crossover gadget for each crossing of edges. However, it is possible to prove NP-hardness also using Planar 3-SAT, as proposed in Exercise 14.3.

The lower bounds for slightly super-exponential parameterized complexity were introduced by Lokshtanov, Marx, and Saurabh [326]. In particular, all the lower bounds contained in Section *14.3.2 originate in [326], apart from the lower bound for Cycle Packing (first part of Theorem 14.19) that was proved by Cygan, Nederlof, Pilipczuk, Pilipczuk, van Rooij, and Wojtaszczyk [118] as one of complementary results for their Cut & Count technique. Exercise 14.8 originates also in the work of Cygan, Nederlof, Pilipczuk,

Pilipczuk, van Rooij, J.M.M., and Wojtaszczyk [118]. Exercise 14.9 comes from the work of Drange, Dregi, and van 't Hof [157].

The $\mathcal{O}^*(d^d)$ algorithm for Distortion was proposed by Fellows, Fomin, Lokshtanov, Losievskaja, Rosamond, and Saurabh [176], and the matching lower bound included in Theorem 14.18 was given by Lokshtanov, Marx, and Saurabh [326]. We refer to a survey of Gupta, Newman, Rabinovich, and Sinclair [246] for more information about metric embeddings and their applications in algorithms.

The doubly exponential lower bound for Edge Clique Cover was given by Cygan, Pilipczuk, and Pilipczuk [120]. The nonexistence of a subexponential kernel under $P \neq NP$, i.e., Exercise 14.10, was given in the PhD thesis of Pilipczuk [385].

The lower bound of Theorem 14.21 was proved by Chen, Huang, Kanj, and Xia [80] (conference version is [78]). A weaker lower bound, which does not allow for the factor $f(k)$ in the running time, was proved by Chen, Chor, Fellows, Huang, Juedes, Kanj, and Xia [76]. These papers, as well as [79], noted that lower bounds of this form for W[1]-hard problems can be transferred to other problems via parameterized reductions that increase the parameter at most linearly.

The lower bound Theorem 14.24 for Subgraph Isomorphism was proved by Marx [352] as a corollary of a much more general result on restricted constraint satisfaction problems where the constraint graph has to belong to a certain class. The lower bounds of Theorem 14.27 are due to Marx [349].

Alber and Fiala [9] presented an $n^{\mathcal{O}(\sqrt{k})}$ time algorithm for Unit Disk Independent Set using a geometric separator theorem. However, as discussed by Marx and Sidiropoulos [359], one can obtain this result by a simple application of the shifting technique. The W[1]-hardness of Unit Disk Independent Set was proved by Marx [345], but with a reduction that increases the parameter more than quadratically, hence giving only a lower bound weaker than the one stated in Theorem 14.34. The Grid Tiling problem was formulated by Marx [348], obtaining the tight lower bound of Theorem 14.34 for Unit Disk Independent Set, as well as an analogous result for Unit Square Independent Set. The lower bound of Theorem 14.32 and the algorithm of Theorem 14.31 for Scattered Set is by Marx and Pilipczuk [355]. The Grid Tiling problem was used by Marx [354] to show that, assuming ETH, there is no $f(k) \cdot n^{o(\sqrt{\ell})}$-time algorithm for Edge Multicut on planar graphs with ℓ terminals for any computable function f, essentially matching the $2^{\mathcal{O}(\ell)} \cdot n^{\mathcal{O}(\sqrt{\ell})}$-time algorithm of Klein and Marx [295]. Chitnis, Hajiaghayi, and Marx [92] used Grid Tiling to prove that, assuming ETH, there is no $f(k) \cdot n^{o(\sqrt{k})}$-time algorithm for Strongly Connected Steiner Subgraph on planar graphs, parameterized by the number k of terminals, for any computable function f. Additionally, they presented a $2^{\mathcal{O}(k \log k)} \cdot n^{\mathcal{O}(\sqrt{k})}$-time algorithm for planar graphs (on general graphs, an earlier algorithm of Feldman and Ruhl [173] solves the problem in time $n^{\mathcal{O}(k)}$ time). Marx and Pilipczuk [356] presented a number of W[1]-hardness results for various parameterizations of Subgraph Isomorphism; many of these reductions start from Grid Tiling. The Grid Tiling with $\leq$ problem was formalized by Marx and Sidiropoulos [359], who also studied higher-dimensional generalizations of Grid Tiling and Grid Tiling with $\leq$ and obtained tight lower bounds for higher-dimensional geometric problems.

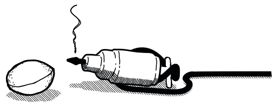
The lower bound for Hitting Set parameterized by the size of the universe was given by Cygan, Dell, Lokshtanov, Marx, Nederlof, Okamoto, Paturi, Saurabh, and Wahlström [112]. The work of Cygan et al. contains a much larger net of reductions, which links many classic parameterized problems either to SETH or to the Set Cover Conjecture (Conjecture 14.36). In particular, the Set Cover Conjecture is formulated in [112]. Notably, the authors consider also parity variants of the considered problems. In this setting the analogue of the Set Cover Conjecture is indeed equivalent to the analogue of SETH, which provides further evidence in favor of the Set Cover Conjecture.

The framework for proving SETH-based lower bounds for treewidth/pathwidth parameterizations was introduced by Lokshtanov, Marx, and Saurabh [325]. In particular, the presented lower bound for Independent Set (Theorem 14.38) is due to them. The first three lower bounds of Theorem 14.41 were also obtained in [325], while the next three, as well as that of Exercise 14.18, was given by Cygan, Nederlof, Pilipczuk, Pilipczuk, van Rooij, and Wojtaszczyk [118] as complementary results for their Cut & Count technique. The last lower bound, for Hamiltonian Cycle, was given by Cygan, Kratsch, and Nederlof [113]. Interestingly, in the same work the authors give a matching algorithm with running time $\mathcal{O}^*((2+\sqrt{2})^p)$, where p is the width of a given path decomposition.

An $\mathcal{O}(N^{k+o_k(1)})$ algorithm for Dominating Set was proposed by Eisenbrand and Grandoni [163]. The matching lower bound under SETH, i.e., Theorem 14.42, together with the algorithm of Exercise 14.19, is due to Pătraşcu and Williams [390]. Interestingly, Pătraşcu and Williams viewed this theorem not as a lower bound, but rather as a possible approach to obtaining faster algorithms for CNF-SAT.

Chapter 15
Lower bounds for kernelization

In this chapter we discuss methodologies for proving lower bounds for kernelization. We describe the technique of compositionality, which is the main tool for proving negative results about the existence of polynomial kernels. We also give an introduction to weak compositions, which can be used to provide more precise lower bounds on kernel sizes.



In Chapters 2, 9, and 12 we have learned many different approaches to designing efficient kernelization algorithms. While some parameterized problems are naturally amenable to various preprocessing techniques, we have so far not discussed what properties of a given problem make it resistant from the point of view of efficient kernelization. In Chapter 2 we have seen that the existence of *any* kernelization algorithm is equivalent to the existence of a fixed-parameter tractable algorithm for a problem. Therefore, the theory of parameterized intractability that was discussed in Chapter 13 can be used to investigate whether a problem admits any kernel at all. This answer is, however, quite unsatisfactory: we have argued that the interesting kernelization algorithms are those that have good (say, polynomial) guarantees on the output size, and for many problems such algorithms can be indeed designed. Under what conditions and what assumptions can we refute the existence of a polynomial kernel for a particular fixed-parameter tractable problem?

The best known answer to this question so far comes via the framework of *compositionality*. The development of this technique around the year 2008 greatly increased the interest in kernelization as a research area. Indeed, the existence of a methodology for proving negative results about kernelization made it possible to perform a systematic classification of problems into those that do admit polynomial kernels, and those that probably do not. Before this breakthrough, one could only investigate how different combinatorial properties of problems can be used in designing efficient preprocessing routines, but understanding their limitations was out of reach.

In this chapter we introduce the technique of compositionality. Fortunately, the complexity foundations of the framework are quite simple to explain, so many of the fundamental results to be presented will be proved in a self-contained way. However, our main goal in this chapter, as in Chapters 13 and 14, is to provide a basic practitioner's toolbox. Therefore, after presenting the complexity foundations for the technique we illustrate its usage with a few selected examples that reflect various scenarios appearing in "real life". In particular, we discuss the notion of *cross-composition*, which is a convenient formalism for applying compositionality; we describe different strategies for designing compositions, focusing on the concept of an *instance selector*; and we present *polynomial parameter transformations*, which serve the role of reductions in the world of kernelization lower bounds.

Finally, at the end of this chapter we present the technique of *weak compositions*. It is a variant of compositionality, using which one can prove lower bounds on kernel sizes for problems that actually do admit polynomial kernels. We provide full details of the two probably most important results of this area: that the Vertex Cover and Feedback Vertex Set problems do not admit kernels with bitsize $\mathcal{O}(k^{2-\varepsilon})$ for any $\varepsilon > 0$, unless NP $\not\subseteq$ coNP/ poly.

15.1 Compositionality

Since the principles of compositionality are very natural, whereas the formal layer can be more challenging to understand, we first give a description of the intuition behind the methodology before going into formal details.

Let us take as the working example the Longest Path problem: given a graph G and an integer k, we ask whether G contains a simple path on k vertices. Assume for a moment that this problem admits a kernelization algorithm that reduces the number of vertices to at most k^3. More formally, given an instance (G, k) it outputs an equivalent instance (G', k') such that $|V(G')|, k' \leq k^3$. Suppose somebody gave us k^7 instances of Longest Path with the same parameter k; denote them by $(G_1, k), (G_2, k), \ldots, (G_{k^7}, k)$. Consider now a graph H obtained by taking the disjoint union of graphs $G_1, G_2, \ldots, G_{k^7}$, and observe that H contains a path on k vertices if and only if *at least* one of the graphs $G_1, G_2, \ldots, G_{k^7}$ does. In other words, the answer to the instance (H, k) is equal to the logical OR of the answers to instances $(G_1, k), (G_2, k), \ldots, (G_{k^7}, k)$. We will also say that the instance (H, k) is an OR-*composition* of instances $(G_1, k), (G_2, k), \ldots, (G_{k^7}, k)$.

Now apply the assumed kernelization algorithm to the instance (H, k), and obtain some new instance (H', k') such that $|V(H')|, k' \leq k^3$. Observe that (H', k') can be encoded in roughly $k^6/2 + 3\log k$ bits: we can encode the adjacency matrix of H' in $\binom{k^3}{2}$ bits, while the encoding of k' takes $3 \log k$ bits. This is, however, much less than k^7, the number of the input instances!

Therefore, during the kernelization process the algorithm must have 'forgotten' information about a vast majority of instances. The desired k-path, however, can lure in any of them. Losing information about this instance can change the global answer to the problem, if it is the only one with the positive outcome. Consequently, it seems that the kernelization algorithm would need to be able to solve some of the given instances, or at least to reason about which of them are less prone to having a positive answer and can be safely discarded. Such an ability would be highly suspicious for an algorithm running in polynomial time, since the LONGEST PATH problem is NP-hard.

Of course, there is nothing magical in the numbers 3 and 7 that we picked in this example, and the reasoning is much more general: a kernelization algorithm for LONGEST PATH with polynomial guarantees on the output size would need to discard some information that it should not be able to identify as useless. Our goal is therefore to put into formal terms why the described behavior is unlikely, and link this understanding to some well-established assumptions from complexity theory.

15.1.1 Distillation

We first aim to formally capture the intuition that some information has to be lost when packing too many instances into a too small space in the kernel. In order to achieve this, we introduce the concept of *distillation*, which will be a result of pipelining a composition for a parameterized problem with a hypothetical polynomial kernel for it. Thus, by (a) designing a composition, and (b) proving that the existence of a distillation is unlikely, we will be able to refute the existence of a polynomial kernel. This section is devoted to understanding issue (b). Since parameterizations will be needed only to formally capture the concept of kernelization, the definition of distillation considers classic (that is, not parameterized) languages.

Let Σ be some fixed, constant-size alphabet that we will use for encoding the instances.

Definition 15.1. Let $L, R \subseteq \Sigma^*$ be two languages. An OR-*distillation* of L into R is an algorithm that, given a sequence of strings $x_1, x_2, \ldots, x_t \in \Sigma^*$, runs in time polynomial in $\sum_{i=1}^{t} |x_i|$ and outputs one string $y \in \Sigma^*$ such that

(a) $|y| \leq p(\max_{i=1}^{t} |x_i|)$ for some polynomial $p(\cdot)$, and
(b) $y \in R$ if and only if there exists at least one index i such that $x_i \in L$.

Thus, the answer to the output instance of R is equivalent to the logical OR of the answers to the input instances of L.

In what follows we will be considering the complexity class coNP/ poly. We assume that the reader is familiar with the concepts of co-nondeterminism and nonuniform complexity classes; we refer her or him to any textbook on complexity theory for an introduction to these topics. However, for the sake of clarity we recall now the definition of coNP/ poly.

Definition 15.2. We say that a language L belongs to the complexity class coNP/ poly if there is a Turing machine M and a sequence of strings $(\alpha_n)_{n=0,1,2,\ldots}$, called the *advice*, such that:

- Machine M, when given input x of length n, has access to string α_n and has to decide whether $x \in L$. Machine M works in *co-nondeterministic polynomial time*. That is, it makes a polynomial number of steps that may be chosen co-nondeterministically: If $x \in L$, the algorithm should derive this conclusion for *every* possible run, whereas if $x \notin L$, then *at least one* run needs to finish with this conclusion.
- $|\alpha_n| \leq p(n)$ for some polynomial $p(\cdot)$.

Note that in this definition α_n depends on n only; in other words, for each n we need to design a "global" advice α_n that will work for all the inputs of length n. We are now ready to state the crucial result standing behind the compositionality framework.

Theorem 15.3. *Let $L, R \subseteq \Sigma^*$ be two languages. If there exists an* OR-*distillation of L into R, then $L \in$* coNP/ poly.

Before we proceed to the proof, let us see the consequences of the theorem in the form of the following corollary. In the following, whenever we talk about NP-hardness we mean NP-hardness with respect to Karp reductions.

Corollary 15.4. *If an* NP-*hard language $L \subseteq \Sigma^*$ admits an* OR-*distillation into some language $R \subseteq \Sigma^*$, then* NP $\subseteq$ coNP/ poly.

Proof. For any language $L' \in$ NP we can construct an algorithm resolving membership in L' in coNP/ poly as follows. We first apply the NP-hardness reduction from L' to L, and then run the algorithm resolving membership in L in coNP/ poly implied by Theorem 15.3. Consequently NP $\subseteq$ coNP/ poly, which is a highly unexpected outcome from the point of view of the complexity theory. □

The assumption that NP $\not\subseteq$ coNP/ poly may be viewed as a stronger variant of the statement that NP $\neq$ coNP. The latter claim means that verification of certificates in polynomial time cannot be simulated by verification of counterexamples in polynomial time. In the claim that NP $\not\subseteq$ coNP/ poly we assume further that such a simulation is impossible even if the co-nondeterministic polynomial-time algorithm resolving a problem in NP is given access to some polynomial-size advice string that depends on the input length only. It is known that NP $\subseteq$ coNP/poly implies that $\Sigma_3^{\mathrm{P}} = \mathrm{PH}$, that is, the polynomial hierarchy collapses to its third level. This is not as dramatic a collapse as P = NP, but it is serious enough to be highly implausible.

Note here that we do not assume *anything* about the target language R — it can be even undecidable. Therefore, in some sense the theorem has an information-theoretical character. It shows that the possibility of packing a lot of information into small space gives rise to a surprising algorithm resolving the problem, lying roughly on the opposite side of the polynomial hierarchy than expected. The way of organizing the information in the output instance is not relevant at all.

Proof (of Theorem 15.3). Without loss of generality assume that $\Sigma = \{0,1\}$; this can be easily done by encoding symbols of a larger alphabet using constant-length binary strings. Let $\mathcal{A}$ be the assumed OR-distillation algorithm, and let $p(\cdot)$ be a polynomial upper bounding the length of the output of $\mathcal{A}$. Without loss of generality we may assume that $p(\cdot)$ is nondecreasing. Let $K = p(n)$ so that algorithm $\mathcal{A}$ run on a sequence of strings, each of length at most n, outputs a string of length at most K. Let us fix $t = K+1$. Thus, algorithm $\mathcal{A}$ maps the set $D = (\Sigma^{\leq n})^t$ of t-tuples of input strings into the set $\Sigma^{\leq K}$.[1]

Let now $A = L \cap \Sigma^{\leq n}$ and $\overline{A} = \Sigma^{\leq n} \setminus L$ be the sets of yes- and no-instances of L of length at most n, respectively. Similarly, let $B = R \cap \Sigma^{\leq K}$ and $\overline{B} = \Sigma^{\leq K} \setminus R$. By the assumption that $\mathcal{A}$ is an OR-distillation, we have that $\mathcal{A}$ maps $(\overline{A})^t$ into $\overline{B}$, and $D \setminus (\overline{A})^t$ into B.

For strings $x \in \Sigma^{\leq n}$ and $y \in \Sigma^{\leq K}$, we will say that x is *covered* by y if there exists a t-tuple $(x_1, x_2, \ldots, x_t) \in D$ such that (a) $x = x_i$ for some i, and (b) $\mathcal{A}(x_1, x_2, \ldots, x_t) = y$. In other words, x is contained in some tuple from the preimage of y. More generally, for $X \subseteq \Sigma^{\leq n}$ and $Y \subseteq \Sigma^{\leq K}$ we will say that X is covered by Y if every element of X is covered by at least one element of Y. The following claim will be the core argument in the proof.

Claim 15.5. *There exists a set $Y \subseteq \overline{B}$ such that*

(a) $|Y| \leq n+1$, *and*
(b) Y *covers* $\overline{A}$.

Since $\mathcal{A}^{-1}(\overline{B}) = (\overline{A})^t$, no element of $\overline{B}$ can cover any string outside $\overline{A}$. Of course, $\overline{B}$ itself covers $\overline{A}$, but the size of $\overline{B}$ can be huge in terms of n. The gist of the proof is that we can find a covering set of size polynomial in n, rather than exponential.

[1] For a set S, notation S^t and $S^{\leq t}$ denotes the set of sequences of elements from S that have length exactly t, resp. at most t.

Before we proceed to the proof of Claim 15.5, let us shed some light on its motivation by showing how it implies Theorem 15.3. That is, we would like to prove that $L \in \mathrm{coNP/poly}$. According to the definition, we need to construct (a) an algorithm resolving membership in L in co-nondeterministic polynomial time, and (b) a sequence of advice strings α_n for $n = 0, 1, 2, \ldots$ that will be given to the algorithm.

In our case, advice α_n will be an encoding of the covering set Y. Since Y contains at most $n+1$ strings, each of length at most K, the advice string α_n will be of length polynomial in n. The algorithm works as follows. Given the input string x of length n, it tries to prove that $x \notin L$. If this is the case, then there exists a tuple $(x_1, x_2, \ldots, x_t) \in D$ such that $x = x_i$ for some i, and $\mathcal{A}(x_1, x_2, \ldots, x_t) \in Y$. The algorithm guesses co-nondeterministically this tuple, computes $\mathcal{A}(x_1, x_2, \ldots, x_t)$, and checks whether the result is contained in the advice string α_n. If indeed $x \notin L$, then for at least one guess we will compute a string belonging to Y, which is a certificate that $x \notin L$ since $Y \subseteq \overline{B}$. If $x \in L$, however, then by the definition of an OR-distillation every tuple containing x is mapped to a string contained in B, so in particular outside of Y.

Let us examine how the polynomial advice is used in this argument: Claim 15.5 provides us only a proof of the *existence* of a small covering set Y, and no means of computing it efficiently. Indeed, computation of this set would be extremely difficult, as it depends on the actual behavior of algorithm $\mathcal{A}$ on all the inputs from D. Instead, this set is provided to the algorithm in the advice string, so there is no need to construct it. The crux is that the constructed covering set Y works for *all* the input strings x of length n, and hence we can have the same advice for all the inputs of the same length.

We now proceed to the proof of Claim 15.5. We will consecutively construct strings $y_1, y_2, y_3, \ldots \in \overline{B}$ up to the point when the set $Y_i = \{y_1, y_2, \ldots, y_i\}$ already covers the whole $\overline{A}$. Let S_i be the set of strings from $\overline{A}$ that are not covered by Y_i; we have that $S_0 = \overline{A}$. During the construction we will guarantee that $|S_i| \leq \frac{|\overline{A}|}{2^i}$, which is of course satisfied at the beginning. Since $|\overline{A}| \leq |\Sigma^{\leq n}| < 2^{n+1}$ (here we use the assumption that $|\Sigma| = 2$), the construction will terminate after at most $n+1$ steps with S_i being empty.

It remains to show how to construct string y_i based on the knowledge of Y_{i-1}. Recall that algorithm $\mathcal{A}$ maps every tuple from the set $(S_{i-1})^t \subseteq (\overline{A})^t$ into $\overline{B}$. Since $|\overline{B}| \leq |\Sigma^{\leq K}| < 2^{K+1}$, it follows by the pigeonhole principle that there exists some $y \in \overline{B}$ such that $|\mathcal{A}^{-1}(y) \cap (S_{i-1})^t| \geq \frac{|(S_{i-1})^t|}{2^{K+1}} = \left(\frac{|S_{i-1}|}{2}\right)^t$. Observe now that every string from S_{i-1} that is contained in any tuple from $\mathcal{A}^{-1}(y) \cap (S_{i-1})^t$ is covered by y. Therefore, if we set $y_i = y$, then every string from every tuple from the set $\mathcal{A}^{-1}(y) \cap (S_{i-1})^t$ is contained in $S_{i-1} \setminus S_i$, since it gets covered. Consequently $\mathcal{A}^{-1}(y) \cap (S_{i-1})^t \subseteq (S_{i-1} \setminus S_i)^t$, and hence

$$\left(\frac{|S_{i-1}|}{2}\right)^t \leq \left|\mathcal{A}^{-1}(y) \cap (S_{i-1})^t\right| \leq \left|(S_{i-1} \setminus S_i)^t\right| = |S_{i-1} \setminus S_i|^t.$$

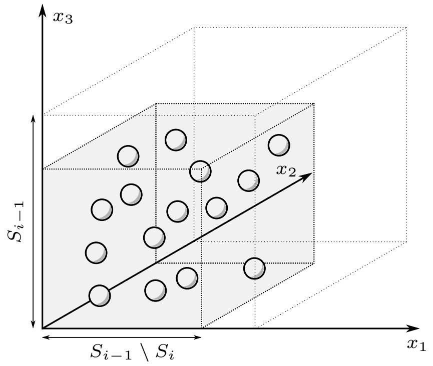


Fig. 15.1: The core combinatorial argument in the proof of Claim 15.5: if a small t-dimensional product hypercube is contained in a large t-dimensional product hypercube, and it contains at least a $\frac{1}{2^t}$ fraction of all the tuples, then the side of the small hypercube must be at least half as long as the side of the large one. The balls within the small hypercube depict tuples from $\mathcal{A}^{-1}(y) \cap (S_{i-1})^t$

We infer that $|S_{i-1} \setminus S_i| \geq \frac{|S_{i-1}|}{2}$, which means that $|S_i| \leq |S_{i-1}|/2$ and the condition $|S_i| \leq \frac{|\overline{A}|}{2^i}$ follows by a trivial induction. As we have argued, this means that the construction will terminate after at most $n+1$ steps, yielding a feasible set Y. This concludes the proof of Claim 15.5, and of Theorem 15.3. □

15.1.2 Composition

We now formalize what it means to compose instances. Recall that in the Longest Path example we assumed that all the input instances have the same parameter. It appears that we can make even stronger assumptions, which is formalized in the following definition.

Definition 15.6. An equivalence relation $\mathcal{R}$ on the set Σ^* is called a *polynomial equivalence relation* if the following conditions are satisfied:

(a) There exists an algorithm that, given strings $x, y \in \Sigma^*$, resolves whether $x \equiv_{\mathcal{R}} y$ in time polynomial in $|x| + |y|$.
(b) Relation $\mathcal{R}$ restricted to the set $\Sigma^{\leq n}$ has at most $p(n)$ equivalence classes, for some polynomial $p(\cdot)$.

Assume, for instance, that the input to our problem is some fixed encoding of an undirected graph. We can then define $\mathcal{R}$ as follows. First, we put into one equivalence class all the strings from Σ^* that do not encode any graph. We call such instances *malformed*, while all the other instances are *well-formed*. Next, note that we can choose the encoding in such a manner that graphs with encoding of length at most n have at most n vertices and at most n edges. Therefore, if we say that two well-formed instances are $\mathcal{R}$-equivalent if the corresponding graphs have the same numbers of vertices and edges, then $\Sigma^{\leq n}$ is divided by $\mathcal{R}$ into at most $n^2 + 1$ classes (+1 comes from the malformed class) and condition (b) is satisfied. Of course, provided that we chose a reasonable encoding, condition (a) holds as well. If the input to the problem was a graph and an integer k, encoded in unary, then we could in addition refine $\mathcal{R}$ by requiring that the instances x and y have the same value of k — we would just increase the bound on the number of equivalence classes in $\Sigma^{\leq n}$ from n^2+1 to n^3+1. In the later sections we will see that the ability to make such assumptions about the input instances can be very useful for streamlining the composition.

We now proceed to the main definition.

Definition 15.7. Let $L \subseteq \Sigma^*$ be a language and $Q \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized language. We say that L *cross-composes* into Q if there exists a polynomial equivalence relation $\mathcal{R}$ and an algorithm $\mathcal{A}$, called the *cross-composition*, satisfying the following conditions. The algorithm $\mathcal{A}$ takes as input a sequence of strings $x_1, x_2, \ldots, x_t \in \Sigma^*$ that are equivalent with respect to $\mathcal{R}$, runs in time polynomial in $\sum_{i=1}^{t} |x_i|$, and outputs one instance $(y, k) \in \Sigma^* \times \mathbb{N}$ such that:

(a) $k \leq p(\max_{i=1}^{t} |x_i| + \log t)$ for some polynomial $p(\cdot)$, and
(b) $(y, k) \in Q$ if and only if there exists at least one index i such that $x_i \in L$.

Note that in this definition, contrary to OR-distillation, it is only the output parameter that is small, while the whole output string y may be even as huge as the concatenation of the input instances. Moreover, the output parameter can also depend poly-logarithmically on the number of input instances. This seemingly innocent nuance will be heavily exploited in the later sections.

To give an example, observe that the Hamiltonian Path problem cross-composes into Longest Path parameterized by the requested path length as follows. For the equivalence relation $\mathcal{R}$ we take a relation that puts all malformed instances into one equivalence class, while all the well-formed instances are partitioned with respect to the number of vertices of the graph. The cross-composition algorithm, given a bunch of malformed instances, returns some trivial no-instance of Longest Path, while given a sequence of graphs on n vertices, it returns the encoding of their disjoint union together with parameter $k = n$. For a reasonable encoding we have that n is bounded by the maximum length of an input string, and so condition (a) holds. Clearly, the disjoint union of graphs on n vertices has an n-vertex path if and only if one of the input graphs admitted a Hamiltonian path. Hence condition (b) is satisfied as well.

As the reader expects, pipelining the cross-composition from an NP-hard problem with a polynomial kernel will yield an OR-distillation of the NP-hard problem, implying that NP $\subseteq$ coNP/ poly. However, since the output of an OR-distillation can be an instance of *any* language, we will be able to refute the existence of a larger class of preprocessing routines.

Definition 15.8. A *polynomial compression* of a parameterized language $Q \subseteq \Sigma^* \times \mathbb{N}$ into a language $R \subseteq \Sigma^*$ is an algorithm that takes as input an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, works in time polynomial in $|x| + k$, and returns a string y such that:

(a) $|y| \leq p(k)$ for some polynomial $p(\cdot)$, and
(b) $y \in R$ if and only if $(x, k) \in Q$.

If $|\Sigma| = 2$, the polynomial $p(\cdot)$ will be called the *bitsize* of the compression.

Whenever we talk about the existence of a polynomial compression and we do not specify the target language R, we mean the existence of a polynomial compression into *any* language R.

Obviously, a polynomial kernel is also a polynomial compression by treating the output kernel as an instance of the unparameterized version of Q. Here, by an *unparameterized version* of a parameterized language Q we mean a classic language $\tilde{Q} \subseteq \Sigma^*$ where the parameter is appended in unary after the instance. The main difference between polynomial compression and kernelization is that the polynomial compression is allowed to output an instance of *any* language R, even an undecidable one. Clearly, if R is reducible in polynomial time back to Q, then pipelining the compression with the reduction gives a polynomial kernel for Q. However, R can have much higher complexity than Q, and there are examples of natural problems for which there exists a polynomial compression, but a polynomial kernel is not known.

We are now ready to state and prove the main theorem of this section.

Theorem 15.9. *Assume that an* NP*-hard language L cross-composes into a parameterized language Q. Then Q does not admit a polynomial compression, unless* NP $\subseteq$ coNP/ poly.

Proof. Let $\mathcal{A}$ be a cross-composition of L into Q, let $p_0(\cdot)$ be the polynomial bounding the parameter of the output instance of $\mathcal{A}$, and let $\mathcal{R}$ be the polynomial equivalence relation used by $\mathcal{A}$. Assume further that Q admits a polynomial compression $\mathcal{C}$ into some language R. Let OR(R) be a language consisting of strings of the form $d_1\#d_2\#\ldots\#d_q$ such that for at least one index i it holds that $d_i \in R$. Here, # is some special symbol that is artificially added to Σ and is not used in any string from R. We are going to construct an OR-distillation of L into OR(R), which by Corollary 15.4 implies that NP $\subseteq$ coNP/ poly. The algorithm is depicted in Fig. 15.2.

Let $x_1, x_2, \ldots, x_t$ be the sequence of input strings and let $n = \max_{i=1}^{t} |x_i|$. Apply the following polynomial-time preprocessing: examine the strings pairwise and remove all the duplicates. Note that this step does not change

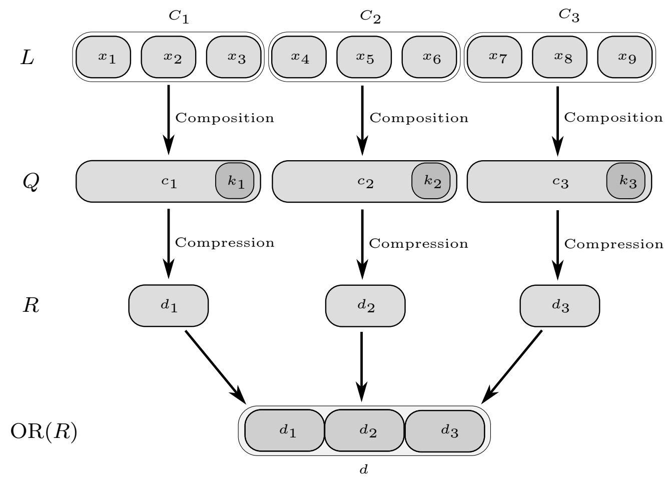


Fig. 15.2: The workflow of the algorithm in the proof of Theorem 15.9

whether at least one of the strings belongs to L. Observe that the number of different strings over Σ of length at most n is bounded by $\sum_{i=0}^{n} |\Sigma|^i \leq |\Sigma|^{n+1}$. Hence, after removing the duplicates we have that $t \leq |\Sigma|^{n+1}$, and so we can assume that $\log t = \mathcal{O}(n)$.

Partition the input strings into equivalence classes $C_1, C_2, \ldots, C_q$ with respect to the relation $\mathcal{R}$. By the definition of the polynomial equivalence relation, this step can be performed in polynomial time, and q, the number of the obtained classes, is bounded by some polynomial $p_1(n)$.

For $j = 1, 2, \ldots, q$, apply the cross-composition $\mathcal{A}$ to the strings contained in the class C_j, obtaining a parameterized instance (c_j, k_j) such that (a) $k_j \leq p_0(\max_{x \in C_j} |x| + \log |C_j|)$, which is polynomially bounded in n and $\log t$, and (b) $(c_j, k_j) \in Q$ if and only if there exists at least one instance $x \in C_j$ such that $x \in L$. Consequently, there exists an index i, $1 \leq i \leq t$, such that $x_i \in L$ if and only if there exists an index j, $1 \leq j \leq q$, such that $(c_j, k_j) \in Q$. Since $\log t = \mathcal{O}(n)$, we infer that for some polynomial $p_2(\cdot)$ it holds that $k_j \leq p_2(n)$ for $j = 1, 2, \ldots, q$.

Now apply the assumed compression algorithm $\mathcal{C}$ to each instance (c_j, k_j), obtaining a string d_j such that $d_j \in R$ if and only if $(c_j, k_j) \in Q$. Since $k_j \leq p_2(n)$ and $|d_j| \leq p_3(k_j)$ for some polynomial $p_3(\cdot)$, we have that $|d_j| \leq p_4(n)$ for some polynomial $p_4(\cdot)$.

Finally, merge all the strings d_j into one instance $d = d_1\#d_2\#\ldots\#d_q$. From the construction it immediately follows that $d \in \mathrm{OR}(R)$ if and only if there exists an index i, $1 \leq i \leq t$, such that $x_i \in L$. Hence, condition (b) of OR-distillation is satisfied. For condition (a), we have that $q \leq p_1(n)$ and $|d_j| \leq p_4(n)$. Consequently $|d| \leq p_1(n) \cdot (p_4(n) + 1) - 1$, and condition (a) follows. $\square$

As an immediate corollary of the described cross-composition of Hamiltonian Path into Longest Path, the NP-hardness of the Hamiltonian Path problem, and Theorem 15.9, we obtain the following result.

Corollary 15.10. Longest Path *does not admit a polynomial kernel unless* NP $\subseteq$ coNP/ poly.

We remark that it is very rare for a problem to admit such a simple cross-composition as in the case of Longest Path. Designing a cross-composition usually requires a very good understanding of the combinatorics of the problem together with a few ingenious insights into how the construction can be performed. One often needs to make a clever choice of the starting language L, and/or do a lot of technical work. In Section 15.2 we shall present in a number of examples what kind of strategies can be employed for designing a cross-composition.

It is somewhat interesting that the composition framework, which is basically the only tool that we have so far for proving kernelization lower bounds, seems to be unable to distinguish polynomial kernelization from the weaker notion of polynomial compression. It remains open whether there are natural problems that do admit polynomial compression algorithms, but for which the existence of a polynomial kernel can be refuted.

15.1.3 AND-distillations and AND-compositions

In the previous sections we have been always using the OR function as the template of behavior of distillations and compositions. It is very tempting to conjecture that the same results can be obtained when we substitute OR with AND. We could in the same manner define AND-distillation by replacing condition (b) of OR-distillation with a requirement that the output string y belongs to R if and only if *all* the input strings x_i belong to L. Likewise, AND-cross-composition is defined by replacing condition (b) of cross-composition by requiring that the resulting instance (y, k) belongs to Q if and only if *all* the input strings x_i belong to L.

The proof of Theorem 15.9 is oblivious to whether we use the OR function or the AND function; the outcome of the constructed distillation will be just an instance of AND(R) instead of OR(R). Our arguments behind Theorem 15.3, however, completely break when translating them to the AND setting. It turns out that Theorem 15.3 is still true when we replace OR-distillations with AND-distillations, but the proof is much more difficult, and hence omitted in this book.

Theorem 15.11 ([160]). *Let $L, R \subseteq \Sigma^*$ be two languages. If there exists an* AND*-distillation of L into R, then $L \in$* coNP/ poly.

As we have already mentioned, the proof of Theorem 15.9 can be translated to the AND setting in a straightforward manner; we leave checking this fact as an easy exercise. Therefore, by replacing usage of Theorem 15.3 with Theorem 15.11 we obtain the following result.

Theorem 15.12. *Assume that an* NP*-hard language L* AND*-cross-composes into a parameterized language Q. Then Q does not admit a polynomial compression, unless* $\mathrm{NP} \subseteq \mathrm{coNP}/\mathrm{poly}$.

As an example of a problem that admits a trivial AND-cross-composition take the TREEWIDTH problem: given a graph G and parameter k, verify whether $\mathrm{tw}(G) \leq k$. Since treewidth of a disjoint union of a family of graphs is equal to the maximum over treewidths of these graphs, the disjoint union yields an AND-cross-composition from the unparameterized version of TREEWIDTH into the parameterized one. Computing treewidth of a graph is NP-hard, so by Theorem 15.12 we infer that TREEWIDTH does not admit a polynomial kernel, unless $\mathrm{NP} \subseteq \mathrm{coNP}/\mathrm{poly}$. The same reasoning can be performed for other graph parameters that behave similarly under the disjoint union, for instance for pathwidth or rankwidth.

It is noteworthy that problems known to admit AND-cross-compositions are much scarcer than those amenable to OR-cross-compositions. We hardly know any natural examples other than the aforementioned problems of computing graph parameters closed under taking the disjoint union. Therefore, when trying to refute the existence of a polynomial kernel for a particular problem, it is more sensible to first try to design an OR-cross-composition.

15.2 Examples

We now show four selected proofs of kernelization lower bounds, chosen to present different aspects of the compositions framework. We discuss instance selectors, a common theme in many compositions appearing in the literature. We also present the so-called *polynomial parameter transformations*, which serve the role of hardness reductions in the world of kernelization lower bounds. We conclude with an example of a problem parameterized by some structural measure of the input graph and see that the composition framework works well also in such cases.

15.2.1 Instance selector: SET SPLITTING

Recall that, when designing a cross-composition, we are to encode a logical OR of multiple instances $x_1, x_2, \ldots, x_t$ of the input language as a single parameterized instance (y, k) of the output language. Clearly, we need to somehow include all input instances in (y, k), but we need also to encode the OR behavior.

A natural approach would be to design some gadget that "chooses" which instance x_i we are going to actually solve. A bit more precisely, such an *instance selector* Γ should interact with a potential solution to (y, k) in t different ways, called *states*. In one direction, if we choose state i on Γ and solve the instance x_i, we should be able to obtain a valid solution to (y, k). In the second direction, a valid solution to (y, k) that sets Γ in state i should yield a solution to x_i.

In the design of instance selectors we will often heavily rely on the fact that the parameter k of the output instance (y, k) may depend polylogarithmically on the number of input instances. In particular, the state i of the instance selector will often reflect the binary representation of the integer i.

We now illustrate the concept of an instance selector with a simple example of the Set Splitting problem. Here, we are given a universe U and a family $\mathcal{F}$ of subsets of U; the goal is to choose a subset $X \subseteq U$ that *splits* each set in $\mathcal{F}$. We say that a set X *splits* a set A if both $A \cap X \neq \emptyset$ and $A \setminus X \neq \emptyset$. We are interested in the parameterization $k = |U|$. We remark that this problem has a trivial brute-force FPT algorithm running in time $\mathcal{O}^*(2^k)$.

It is not hard to prove that Set Splitting is NP-hard by a direct reduction from the satisfiability of CNF formulas. In this section we prove the following.

Theorem 15.13. *There exists a cross-composition from* Set Splitting *into itself, parameterized by the size of the universe. Consequently,* Set Splitting *parameterized by $|U|$ does not admit a polynomial compression, unless* NP $\subseteq$ coNP/ poly.

Before we dive into the composition, let us comment what Theorem 15.13 actually means: There are instances of Set Splitting where the family $\mathcal{F}$ is much larger than the universe U, and we do not expect that any efficient (polynomial-time) algorithm is able to sparsify such a family $\mathcal{F}$ without changing the answer in some cases.

Proof (Proof of Theorem 15.13). By choosing an appropriate polynomial equivalence relation $\mathcal{R}$, we may assume that we are given a family of t Set Splitting instances $(\mathcal{F}_i, U_i)_{i=0}^{t-1}$ where $|U_i| = k$ for each i. More precisely, relation $\mathcal{R}$ considers as equivalent all the malformed instances, and all the well-formed instances with the same cardinality of the universe. Let us arbitrarily identify all sets U_i; henceforth we assume that all input instances operate on the same universe U of size k.

Note that we have numbered the input instances starting from 0: in this way it is more convenient to work with the binary representations of indices i.

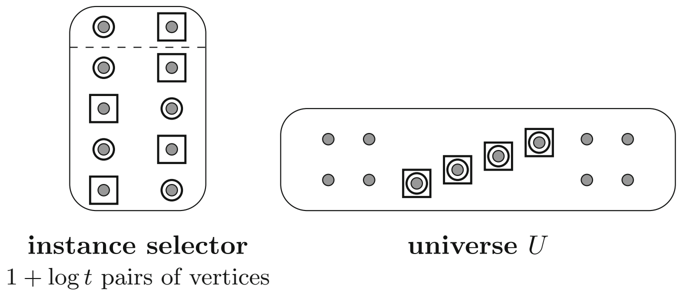


Fig. 15.3: Part of the construction in the proof of Theorem 15.13 for $s = \log t = 4$. The depicted sets are A^0 (elements in circles) and A^1 (elements in rectangles) for some 4-element set $A \in \mathcal{F}_i$ and $i = 5 = 0101_2$.

For the same reason, we duplicate some input instances so that $t = 2^s$ for some integer s. Note that this step at most doubles the number of input instances.

We now proceed to the construction of an instance selector. The selector should have t states; if it attains state i, it should make all input instances 'void' except for the i-th one. We create $2(s+1)$ new elements $\Gamma = \{d_\alpha^\beta : 0 \leq \alpha \leq s, \beta \in \{0,1\}\}$ and define $\hat{U} = U \cup \Gamma$. The new elements are equipped with a family $\mathcal{F}^\Gamma = \{\{d_\alpha^0, d_\alpha^1\} : 0 \leq \alpha \leq s\}$ of sets that force us to choose exactly one element from each pair $\{d_\alpha^0, d_\alpha^1\}$ into the solution X. Intuitively, the choice of which element to include into the solution X for $1 \leq \alpha \leq s$ corresponds to the choice of the input instance $(\mathcal{F}_i, U)$; the additional pair for $\alpha = 0$ is needed to break the symmetry.

We are now ready to formally define the family $\hat{\mathcal{F}}$ for the output instance. We start with $\hat{\mathcal{F}} = \mathcal{F}^\Gamma$ and then, for each input instance $(\mathcal{F}_i, U)$, we proceed as follows. Let $i = b_1 b_2 \ldots b_s$ be the binary representation of the index i, $b_\alpha \in \{0,1\}$ for $1 \leq \alpha \leq s$; note that we add leading zeroes so that the length of the representation is exactly s. Define $b_0 = 0$. For each $A \in \mathcal{F}_i$ we insert the following two sets into the family $\hat{\mathcal{F}}$ (see also Fig. 15.3):

$$A^0 = A \cup \{d_\alpha^{b_\alpha} : 0 \leq \alpha \leq s\} \quad \text{and} \quad A^1 = A \cup \{d_\alpha^{1-b_\alpha} : 0 \leq \alpha \leq s\}.$$

We output the instance $(\hat{\mathcal{F}}, \hat{U})$. Note that $|\hat{U}| = |U| + 2(s+1) = k + 2(\log t + 1)$ and, moreover, the construction can be performed in polynomial time. Thus, to finish the proof we need to show that $(\hat{\mathcal{F}}, \hat{U})$ is a yes-instance of Set Splitting if and only if one of the instances $(\mathcal{F}_i, U)$ is.

In one direction, let X be a solution to the instance $(\mathcal{F}_i, U)$ for some $0 \leq i < t$. Moreover, let $b_1 b_2 \ldots b_s$ be the binary representation of the index i. Define $b_0 = 0$ and consider the set $\hat{X}$ defined as

$$\hat{X} = X \cup \{d_\alpha^{b_\alpha} : 0 \leq \alpha \leq s\}.$$

Clearly, $\hat{X}$ splits each set in $\mathcal{F}^{\Gamma}$, as for each α, $0 \leq \alpha \leq s$, it holds that exactly one of elements d_α^0 and d_α^1 belongs to $\hat{X}$ and exactly one does not. The assumption that X is a solution to $(\mathcal{F}_i, U)$ implies that $\hat{X}$ splits also each set $A \in \mathcal{F}_i$, so in particular it splits A^0 and A^1. Finally, take any $j \neq i$ and any $A \in \mathcal{F}_j$. Since $j \neq i$, indices i and j differ on at least one binary position, say α. Now observe that $d_0^0 \in \hat{X} \cap A^0$ and $d_\alpha^{1-b_\alpha} \in A^0 \setminus \hat{X}$, and $d_0^1 \in A^1 \setminus \hat{X}$ and $d_\alpha^{b_\alpha} \in A^1 \cap \hat{X}$. Consequently, $\hat{X}$ splits both A^0 and A^1.

In the other direction, let $\hat{X}$ be a solution to $(\hat{\mathcal{F}}, \hat{U})$. As $\hat{U} \setminus \hat{X}$ is a solution as well, without loss of generality we may assume that $d_0^0 \in \hat{X}$. Note that since $\hat{X}$ splits each set of $\mathcal{F}^{\Gamma}$, for each $0 \leq \alpha \leq s$ exactly one of the elements d_α^0 and d_α^1 belongs to $\hat{X}$ and one does not. Let then $b_\alpha \in \{0,1\}$ be an index such that $d_\alpha^{b_\alpha} \in \hat{X}$ and $d_\alpha^{1-b_\alpha} \notin \hat{X}$, for $0 \leq \alpha \leq s$. Note that $b_0 = 0$.

Let $0 \leq i < t$ be such that $b_1 b_2 \ldots b_s$ is the binary representation of the index i. We claim that $X = \hat{X} \cap U$ is a solution to the input instance $(\mathcal{F}_i, U)$. Consider any $A \in \mathcal{F}_i$. Observe that $\hat{X} \cap \Gamma = A^0 \cap \Gamma$ and, since $\hat{X}$ splits A^0, we have $A \setminus X = A^0 \setminus \hat{X} \neq \emptyset$. Moreover, $\hat{X} \cap A^1 \cap \Gamma = \emptyset$ and, since $\hat{X}$ splits A^1, we have $A \cap X = A^1 \cap \hat{X} \neq \emptyset$. Hence X splits A and the lemma is proved. □

15.2.2 Polynomial parameter transformations: COLORFUL GRAPH MOTIF *and* STEINER TREE

In the classic theory of NP-hardness we very rarely show that a particular problem L is NP-hard using the definition. Instead of proving from scratch that every problem in NP is polynomial-time reducible to L, we can benefit from the work of Cook and Levin who proved this for satisfiability of propositional formulas, and just construct a polynomial-time reduction from a problem that is already known to be NP-hard. Thus we can focus on one well-chosen hard problem as the source language of the reduction, rather than on an arbitrary problem in NP.

In the world of kernel lower bounds, the situation is somewhat similar: an explicit lower bound via the composition framework does not seem as challenging as re-proving the Cook-Levin theorem (since we in fact reuse the result of Fortnow and Santhanam), but in many cases it is easier to 'transfer' hardness from one problem to another. To be able to do so, we need an appropriate notion of reduction.

Definition 15.14. Let $P, Q \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems. An algorithm $\mathcal{A}$ is called a *polynomial parameter transformation* (PPT, for short) from P to Q if, given an instance (x,k) of problem P, $\mathcal{A}$ works in polynomial time and outputs an equivalent instance $(\hat{x}, \hat{k})$ of problem Q, i.e., $(x,k) \in P$ if and only if $(\hat{x}, \hat{k}) \in Q$, such that $\hat{k} \leq p(k)$ for some polynomial $p(\cdot)$.

Let us verify that the reduction defined as above indeed serves our needs. Assume that, apart from a PPT $\mathcal{A}$ from P to Q, there exists a compression algorithm $\mathcal{B}$ for problem Q that, given an instance $(\hat{x}, \hat{k})$, in polynomial time outputs an instance y of some problem R of size bounded by $p_{\mathcal{B}}(\hat{k})$ for some polynomial $p_{\mathcal{B}}(\cdot)$. Consider pipelining algorithms $\mathcal{A}$ and $\mathcal{B}$: given an instance (x, k) of P, we first use algorithm $\mathcal{A}$ to reduce it to an equivalent instance $(\hat{x}, \hat{k})$ of Q, and then use algorithm $\mathcal{B}$ to compress it to an equivalent instance y of R. Observe that $|y| \leq p_{\mathcal{B}}(\hat{k})$ and $\hat{k} \leq p(k)$, hence $|y|$ is bounded polynomially in k. Since all computations take polynomial time, we obtain a polynomial compression of P into R. This reasoning allows us to state the following.

Theorem 15.15. *Let $P, Q \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems and assume there exists a polynomial parameter transformation from P to Q. Then, if P does not admit a polynomial compression, neither does Q.*

Hence, if we want to show kernelization hardness for problem Q, it suffices to 'transfer' the hardness from some other problem P, for which we already know that (probably) it does not admit a polynomial compression. We remark that Theorem 15.15 uses the notion of compression instead of kernelization: if we liked to prove an analogous statement for polynomial kernelization, we would need to provide a way to reduce 'back' Q to P (in an unparameterized way). This, in turn, requires some additional assumptions, for example NP-hardness of P and $Q \in$ NP. Luckily, the composition framework refutes the existence of not only polynomial kernelization, but also any polynomial compression, and we can use the (simpler) statement of Theorem 15.15.

We illustrate the use of PPTs on the example of the Steiner Tree problem. Here, given a graph G with some terminals $K \subseteq V(G)$ and an integer ℓ, one asks for a connected subgraph of G that both contains all terminals and has at most ℓ edges. Note that without loss of generality we may assume that the subgraph in question is a tree, as we care only about connectivity. We are interested in the parameterization by ℓ.

A direct composition for Steiner Tree may be quite involved and difficult. However, we make our life much easier by choosing an appropriate auxiliary problem: for this problem a composition algorithm is almost trivial, and a PPT reduction to Steiner Tree is also relatively simple.

The auxiliary problem that will be of our interest is called Colorful Graph Motif. In this problem we are given a graph G, an integer k, and a coloring c of vertices of G into k colors (formally, any function $c : V(G) \to [k]$). The task is to find a connected subgraph H of G that contains exactly one vertex of each color (and hence has exactly k vertices). The Colorful Graph Motif problem is NP-hard even if we restrict the input graphs to being trees of maximum degree 3.

Observe now that a disjoint union yields a composition algorithm for Colorful Graph Motif. More precisely, given any number of Colorful Graph Motif instances $(G_i, k_i, c_i)_{i=1}^{t}$ with the same number of colors $k =$

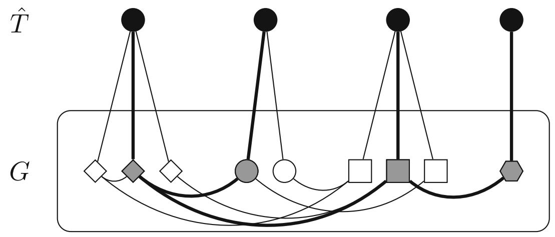


Fig. 15.4: Reduction from Colorful Graph Motif to Steiner Tree. Black vertices are terminals. The shape of a non-terminal vertex corresponds to its color in the input Colorful Graph Motif instance. A solution to Colorful Graph Motif with additional edges for the corresponding solution to Steiner Tree is highlighted with thick edges and grey nodes

k_i, the connectivity requirement ensures that an instance $(\biguplus_{i=1}^{t} G_i, k, \bigcup_{i=1}^{t} c_i)$ is a yes-instance of Colorful Graph Motif if and only if one of the instances (G_i, k_i, c_i) is; here, $\biguplus_{i=1}^{t} G_i$ denotes the disjoint union of graphs G_i. Hence, using the aforementioned NP-hardness of Colorful Graph Motif we may state the following.

Theorem 15.16. Colorful Graph Motif *cross-composes into itself parameterized by k, the number of colors. Consequently,* Colorful Graph Motif *parameterized by k does not admit a polynomial compression, unless* $\mathrm{NP} \subseteq \mathrm{coNP}/\mathrm{poly}$.

Thus, we have obtained a hardness result for Colorful Graph Motif almost effortlessly. It remains to 'transfer' this result to Steiner Tree.

Theorem 15.17. *There exists a PPT from* Colorful Graph Motif *parameterized by the number of colors to* Steiner Tree *parameterized by the size of the tree. Consequently,* Steiner Tree *parameterized by the size of the tree does not admit a polynomial compression, unless* $\mathrm{NP} \subseteq \mathrm{coNP}/\mathrm{poly}$.

Proof. Let (G, k, c) be a Colorful Graph Motif instance. Without loss of generality assume that $k > 1$, since otherwise we are dealing with a trivial yes-instance. We construct a graph $\hat{G}$ from G by creating k new terminals $\hat{K} = \{t_1, t_2, \ldots, t_k\}$, one for each color, and making each terminal t_i adjacent to $c^{-1}(i)$, that is, to all vertices of color i; see Fig. 15.4. Let $\hat{\ell} = 2k - 1$. We claim that $(\hat{G}, \hat{K}, \hat{\ell})$ is a yes-instance of Steiner Tree if and only if (G, k, c) is a yes-instance of Colorful Graph Motif.

In one direction, let H be a solution of the Colorful Graph Motif instance (G, k, c). Without loss of generality assume that H is a tree; hence $|V(H)| = k$ and $|E(H)| = k - 1$. Let $v_i \in V(H)$ be the vertex of color i in the subgraph H. Observe that $\hat{H} = H \cup \{v_i t_i \ : \ 1 \leq i \leq k\}$ is a connected subgraph of $\hat{G}$, containing all terminals t_i and having exactly $2k-1=\hat{\ell}$ edges.

In the other direction, let $\hat{H}$ be a solution of the Steiner Tree instance $(\hat{G},\hat{K},\hat{\ell})$. As $\hat{H}$ is connected and has at most $2k-1$ edges, it contains at most k non-terminal vertices. Recall that we assumed $k>1$. Since $t_i \in V(\hat{H})$ for any $1 \leq i \leq k$, $\hat{H}$ needs to contain a neighbor v_i of t_i in $\hat{G}$. By construction, $v_i \in V(G)$ and $c(v_i)=i$; in particular, all the vertices v_i are pairwise distinct. However, $|V(\hat{H})| \leq \hat{\ell}+1=2k$, hence the vertex set of $\hat{H}$ is *exactly* $\hat{K} \cup \{v_i : 1 \leq i \leq k\}$. In particular, each vertex t_i is of degree 1 in $\hat{H}$ and, consequently, $H=\hat{H} \setminus \hat{K}$ is connected. Observe that H is a subgraph of G, and contains exactly one vertex v_i of each color i. Hence, (G,k,c) is a yes-instance of Colorful Graph Motif, and the claim is proved.

Finally, observe that the aforementioned construction can be performed in polynomial time, and $\hat{\ell}=2k-1$ is bounded polynomially in k. Hence, the presented construction is indeed a PPT from Colorful Graph Motif to Steiner Tree. ☐

We remark that there are also other natural parameterizations of Steiner Tree: we can parameterize by $|K|$, the number of terminals, or by $\ell+1-|K|$, the number of non-terminal vertices of the tree in question. While the latter parameterization yields a $W[2]$-hard problem, recall that for the parameterization by $|K|$ there exists an FPT algorithm with running time $\mathcal{O}^*(2^{|K|})$, see Theorem 10.6. Since $|K| \leq \ell+1$, in Theorem 15.17 we have considered a *weaker* parameterization than by $|K|$, and so Theorem 15.17 refutes also the existence of a polynomial compression for the parameterization by $|K|$, under $\mathrm{NP} \not\subseteq \mathrm{coNP}/\mathrm{poly}$.

15.2.3 A more involved one: Set Cover

In the case of Set Splitting the construction of the instance selector was somehow natural. Here we present a more involved reduction that shows a lower bound for a very important problem, namely Set Cover. An input to Set Cover consists of a universe U, a family $\mathcal{F}$ of subsets of U, and an integer ℓ. The goal is to choose at most ℓ sets from the family $\mathcal{F}$ that cover the entire universe U. That is, we are looking for a subfamily $\mathcal{X} \subseteq \mathcal{F}$ such that $|\mathcal{X}| \leq \ell$ and $\bigcup \mathcal{X}=U$.

It is more convenient for us to work with a graph-theoretic formulation of Set Cover, namely the Red-Blue Dominating Set problem (RBDS for short): Given a bipartite graph G with sides of the bipartition R and B (henceforth called *red* and *blue* vertices, respectively) and an integer ℓ, find a set $X \subseteq R$ of at most ℓ vertices that together dominate the entire set B, i.e., $N(X)=B$. To see that these problems are equivalent, note that given a Set Cover instance $(U,\mathcal{F},\ell)$, one may set $R=\mathcal{F}$, $B=U$, and make each set $A \in \mathcal{F}$ adjacent to all its elements; the condition $\bigcup \mathcal{X} = U$ translates to $N(\mathcal{X}) = U$ in the constructed graph.

The goal of this section is to show a kernelization lower bound for Set Cover, parameterized by the size of the universe, $|U|$. Recall that for this parameterization we have a simple FPT algorithm running in time $\mathcal{O}^*(2^{|U|})$ that performs dynamic programming on subsets of U, see Theorem 6.1. Equivalently, we are interested in RBDS, parameterized by $|B|$.

Theorem 15.18. Set Cover*, parameterized by the size of the universe, does not admit a polynomial compression unless* NP $\subseteq$ coNP/ poly*.*

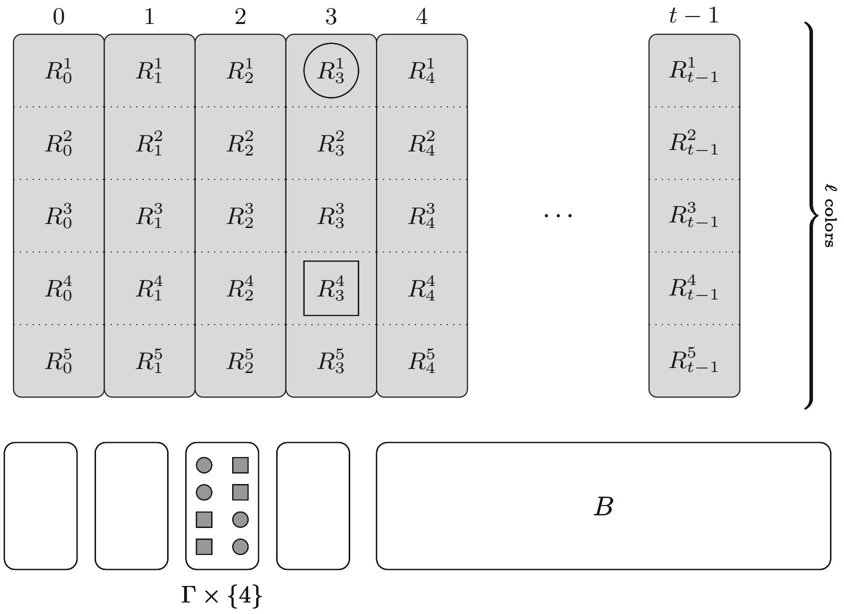
Our first step is to move to a variant of this problem, called Colored Red-Blue Dominating Set (Col-RBDS for short), where the red side R is partitioned into ℓ sets, $R = R^1 \cup R^2 \cup \ldots \cup R^\ell$, and the solution set X is required to contain exactly one vertex from each set R^i. We will often think of sets $R^1, R^2, \ldots, R^\ell$ as of *colors*, and we are looking for a colorful solution in a sense that the solution needs to pick exactly one vertex of each color. We parameterize Col-RBDS by $|B| + \ell$, that is, we add the number of colors to the parameter. Observe that RBDS becomes polynomial for $\ell \geq |B|$ (each vertex $w \in B$ may simply greedily pick one 'private' neighbor in R), so in this problem the parameterization by $|B| + \ell$ is equivalent to the parameterization by $|B|$. However, this is not the case for Col-RBDS, and therefore we explicitly add ℓ to the parameter.

We now formally show that RBDS and Col-RBDS are equivalent.

Lemma 15.19. RBDS*, parameterized by* $|B|$*, and* Col-RBDS*, parameterized by* $|B| + \ell$*, are equivalent with respect to polynomial parameter transformations. That is, there exists a PPT from one problem to the other, and vice versa.*

Proof. In one direction, observe that, given a RBDS instance (G, ℓ), we may replace each vertex $v \in R$ with its ℓ copies $\{v^j \ : \ 1 \leq j \leq \ell\}$ (keeping their neighborhood in B) and define $R^j = \{v^j \ : \ v \in R\}$ for $1 \leq j \leq \ell$. This gives a PPT from RBDS to Col-RBDS with considered parameterizations, provided that $\ell < |B|$. Otherwise, as discussed before, the input RBDS instance is polynomial-time solvable.

The second direction is a bit more involved. Let $(\hat{G}, \ell)$ be an instance of Col-RBDS. Create a graph G from $\hat{G}$ by adding ℓ new vertices $w^1, w^2, \ldots, w^\ell$ to the blue side B, and making each w^j adjacent to all the vertices of R^j, for $1 \leq j \leq \ell$. We claim that (G, ℓ) is an equivalent instance of RBDS; as we parameterize Col-RBDS by $|B| + \ell$, this construction will yield a desired PPT from Col-RBDS to RBDS. To see the claim, note that to dominate each vertex w^j it is sufficient and necessary to take a vertex of R^j into the constructed solution X. Therefore, any solution of the RBDS instance (G, ℓ) needs to contain at least one vertex from each set R^j, and since it contains at most ℓ vertices in total, it must contain exactly one vertex from each set



0
1
2
3
4
$t-1$
R_0^1 R_1^1 R_2^1 R_3^1 R_4^1 R_{t-1}^1
R_0^2 R_1^2 R_2^2 R_3^2 R_4^2 R_{t-1}^2
R_0^3 R_1^3 R_2^3 R_3^3 R_4^3 R_{t-1}^3
R_0^4 R_1^4 R_2^4 R_3^4 R_4^4 R_{t-1}^4
R_0^5 R_1^5 R_2^5 R_3^5 R_4^5 R_{t-1}^5
ℓ colors
B
$\Gamma \times \{4\}$

Fig. 15.5: Gadget in the cross-composition for Col-RBDS with $\ell = 5$ colors. The vertices from R_3^1 are adjacent to the circle-shaped vertices, and the vertices from R_3^4 to the square-shaped ones

R^j. Consequently, a set $X \subseteq V(\hat{G})$ of size at most ℓ is a solution of the Col-RBDS instance $(\hat{G}, \ell)$ if and only if it is a solution of the RBDS instance (G, ℓ), and we are done. □

Lemma 15.19 allows us to focus on Col-RBDS. The enhanced structure of this problem allows us to design a cross-composition, concluding the proof of Theorem 15.18.

Lemma 15.20. Col-RBDS *cross-composes into itself, parameterized by* $|B| + \ell$.

Proof. By choosing an appropriate polynomial equivalence relation, we may assume that we are dealing with a sequence $(G_i, \ell_i)_{i=0}^{t-1}$ of Col-RBDS instances with an equal number of blue vertices, denoted by n, and equal budget $\ell = \ell_i$. For a graph G_i, we denote the sides of its bipartition as B_i and R_i, and let $R_i = R_i^1 \cup R_i^2 \cup \ldots \cup R_i^\ell$ be the partition of the set of red vertices into ℓ colors. We remark that we again number the input instances starting from 0 in order to make use of the binary representation of the index i.

Our first step is to construct a graph H as follows. We first take a disjoint union of the graphs G_i, and then we arbitrarily identify all blue sides B_i into one set B of size n, which constitutes the blue side of H. The red side of H is defined as $R = \bigcup_{i=0}^{t-1} R_i$, and partitioned into colors $R^j = \bigcup_{i=0}^{t-1} R_i^j$, for $1 \le j \le \ell$. We remark that the Col-RBDS instance (H, ℓ) is *not* necessarily equivalent to the logical OR of all input instances (G_i, ℓ_i): a solution to (H, ℓ) may contain vertices that originate from different input instances G_i. We now design a gadget — an instance selector in fact — that will prohibit such behavior.

Let s be the smallest integer such that $t \le 2^s$; note that $s = \mathcal{O}(\log t)$. Let $\Gamma = \{d_\alpha^\beta \ : \ 1 \le \alpha \le s, \beta \in \{0,1\}\}$. For each $1 \le i \le t$, let $b_1^i b_2^i \ldots b_s^i$ be the binary representation of the index i, and let $\gamma(i) = \{d_\alpha^{b_\alpha^i} \ : \ 1 \le \alpha \le s\}$ and $\overline{\gamma}(i) = \Gamma \setminus \gamma(i)$.

We define

$$\hat{B} = B \cup (\Gamma \times \{2, 3, \ldots, \ell\}).$$

That is, we make $\ell - 1$ copies of the set Γ, indexed by integers $2, 3, \ldots, \ell$, and add them to the blue side of the graph H. Finally, for each $1 \le i \le t$ we make each vertex of R_i^1 adjacent to all the vertices of $\gamma(i) \times \{2, 3, \ldots, \ell\}$ and for each $2 \le j \le \ell$ we make each vertex of R_i^j adjacent to all the vertices of $\overline{\gamma}(i) \times \{j\}$; see Fig. 15.5. Let us denote the resulting graph by $\hat{G}$. We claim that the Col-RBDS instance $(\hat{G}, \ell)$ is a yes-instance if and only if one of the instances (G_i, ℓ_i) is a yes-instance. As the construction can be performed in polynomial time, and $|\hat{B}| = |B| + |\Gamma|(\ell - 1) = \mathcal{O}((|B| + \ell) \log t)$, this claim would conclude the proof of the lemma.

Before we start the formal proof, let us give some intuition. To solve $(\hat{G}, \ell)$, we need to pick a vertex x^j from each color R^j. We would like to ensure that whenever $x^1 \in R_i$, then $x^j \in R_i$ as well for all j. This is ensured by the blue vertices in $\Gamma \times \{j\}$: only x^1 and x^j may dominate them, and each such vertex dominates only $s = |\Gamma|/2$ vertices from $\Gamma \times \{j\}$. Hence, to dominate the entire $\Gamma \times \{j\}$, we need to pick a vertex x^j that dominates the complement of the neighborhood of x^1. This, in turn, is only satisfied if both x^1 and x^j originate from the same input instance.

We now proceed to the formal argumentation.

In one direction, let $X = \{x^1, x^2, \ldots, x^\ell\}$ be a solution to an instance (G_i, ℓ_i), where $x^j \in R_i^j$ for $1 \le j \le \ell$. We claim that X is a solution to $(\hat{G}, \ell)$ as well. Clearly, it contains exactly one vertex in each set R^j and $B \subseteq N_{\hat{G}}(X)$. It remains to show that X dominates also the new vertices of $\hat{B}$. To this end, consider an index j, where $2 \le j \le \ell$. By construction, x^1 is adjacent to $\gamma(i) \times \{j\}$, whereas x^j is adjacent to $\overline{\gamma}(i) \times \{j\}$. Hence, $\Gamma \times \{j\} \subseteq N_{\hat{G}}(X)$. As the choice of j was arbitrary, we have that $\Gamma \times \{2, 3, \ldots, \ell\} \subseteq N_{\hat{G}}(X)$, and X is a solution to $(\hat{G}, \ell)$.

In the second direction, let $X = \{x^1, x^2, \ldots, x^\ell\}$ be a solution in the instance $(\hat{G}, \ell)$. We claim that there exists $1 \leq i \leq t$ such that $X \subseteq R_i$; this would imply that X is a solution to (G_i, ℓ_i) as well, finishing the proof of the equivalence. Assume that $x^1 \in R_i$ and $x^j \in R_{i'}$ for some $1 \leq j \leq \ell$. Consider the vertices of $\Gamma \times \{j\}$. By the construction, the neighbors of x^1 in this set are $\gamma(i) \times \{j\}$, the neighbors of x^j in this set are $\overline{\gamma}(i') \times \{j\}$, and no other vertex of X has a neighbor in $\Gamma \times \{j\}$. Since $|\gamma(i)| = |\overline{\gamma}(i')| = s = |\Gamma|/2$ and $\gamma(i) \cup \overline{\gamma}(i') = \Gamma$, we infer that $\gamma(i) = \gamma(i')$ and, consequently, $i = i'$. This finishes the proof of the claim, of Lemma 15.20 and of Theorem 15.18. $\square$

We remark here that there are other natural parameters of Set Cover to consider: we may parameterize it by ℓ, or by $|\mathcal{F}|$ (which in the language of RBDS means parameterization by $|R|$). It is easy to see that the first parameterization generalizes Dominating Set parameterized by the solution size, and hence is $W[2]$-hard. For parameterization by $|\mathcal{F}|$ we have a trivial FPT algorithm working in $\mathcal{O}^*(2^{|\mathcal{F}|})$ time that checks all subsets of $\mathcal{F}$. It can be proved that also for this parameterization the existence of a polynomial kernel may be refuted under the assumption of NP $\not\subseteq$ coNP/poly; see Exercise 15.4.10.

15.2.4 Structural parameters: Clique *parameterized by the vertex cover number*

Our last example concerns the so-called structural parameters: instead of considering a natural parameter such as solution size, we parameterize by some structural measure of the input graph, such as treewidth or cliquewidth. In this section we focus on the classic Clique problem, parameterized by the vertex cover number of the input graph.

More formally, we assume that an input consists of a graph G, and integer ℓ and a vertex cover $Z \subseteq V(G)$ of the graph G. The task is to check whether G contains a complete subgraph on ℓ vertices. We parameterize by $|Z|$, the size of the vertex cover given as an input.

One may wonder why we assume a vertex cover is given on the input: a second option would be to consider a variant where no vertex cover is present, and we only (theoretically) analyze the algorithm in terms of the size of minimum vertex cover of the input graph. However, as the minimum vertex cover problem admits an efficient 2-approximation algorithm, from the point of view of parameterized algorithms and polynomial kernelization these variants are equivalent: we may always compute an approximate vertex cover and use it as the input set Z.

We remark also that Clique, parameterized by the vertex cover number, is trivially fixed-parameter tractable: any clique in G may contain at most one vertex outside Z, so a brute-force algorithm may consider only $2^{|Z|}(|V(G)| -$

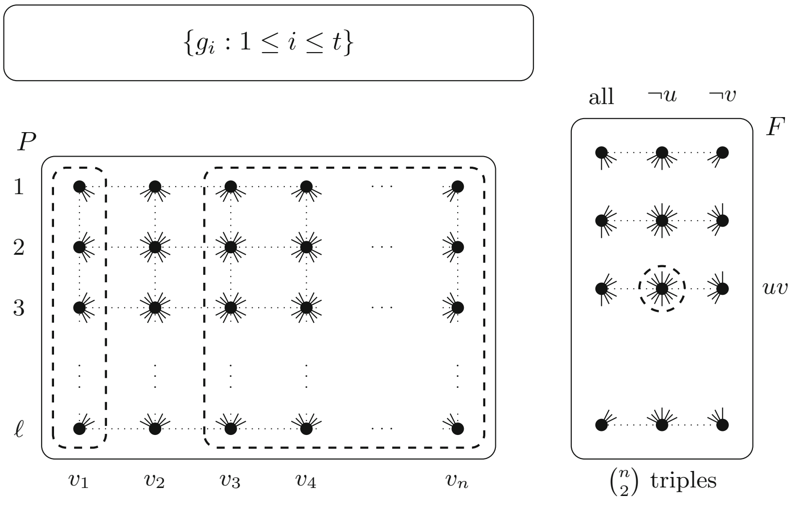


Fig. 15.6: An illustration of the cross-composition for Clique parameterized by the vertex cover number. The vertex $f_{uv}^{\neg u}$ for $u = v_2$ is adjacent to all vertices in P except for the column with subscript u

$|Z|+1)$ possible solutions. Recall that, in contrast, the standard solution-size parameterization of Clique yields a $W[1]$-hard problem.

Our goal in this section is to show that our structural parameterization of Clique does not admit a polynomial compression. That is, we show the following.

Theorem 15.21. Clique *cross-composes into itself, parameterized by the vertex cover number. Consequently,* Clique *parameterized by the vertex cover number does not admit a polynomial compression, unless* NP $\subseteq$ coNP/ poly.

Proof. By choosing an appropriate polynomial equivalence relation, we may assume that we are given a sequence $(G_i, \ell_i)_{i=1}^{t}$ of (unparameterized) Clique instances, where $n = |V(G_i)|$ and $\ell = \ell_i$ for all i. Without loss of generality assume $\ell \leq n$. Let us arbitrarily identify the vertex sets of the input graphs, so henceforth we assume that $V(G_i) = V$ for each i, $1 \leq i \leq t$. We are going to construct a clique instance $(\hat{G}, \hat{\ell})$ with vertex cover $\hat{Z}$ of size bounded polynomially in n.

The graph $\hat{G}$ will consist of two parts: one small central part, where the clique in question is chosen, and a large scattered part, where we choose in which input instance the clique resides. The scattered part will be an independent set: this ensures that the small central part is actually a vertex cover of $\hat{G}$.

For the scattered part, we construct simply a vertex g_i for each input graph G_i. As promised, these vertices are pairwise nonadjacent. The choice of which vertex g_i we include in the solution to $(\hat{G}, \hat{\ell})$ determines the choice of the instance where we actually exhibit a solution.

For the central part, we start with creating $n\ell$ vertices $P = \{p_v^a : v \in V, 1 \le a \le \ell\}$. We make p_v^a adjacent to p_u^b if and only if $v \neq u$ and $a \neq b$. Note that this ensures that every maximum clique in $\hat{G}[P]$ has ℓ vertices: it contains exactly one vertex p_v^a for each index a, and at most one vertex p_v^a for each index v. We also create $3\binom{n}{2}$ vertices $F = \{f_{uv}^{\mathrm{all}}, f_{uv}^{\neg u}, f_{uv}^{\neg v} : uv \in \binom{V}{2}\}$ and make every two of them adjacent if they have different subscripts. Moreover, the vertex f_{uv}^{all} is adjacent to all vertices of P, whereas vertex $f_{uv}^{\neg u}$ is adjacent to $P \setminus \{p_u^a : 1 \le a \le \ell\}$ and $f_{uv}^{\neg v}$ is adjacent to $P \setminus \{p_v^a : 1 \le a \le \ell\}$; see Fig. 15.6.

For each input graph i, $1 \le i \le t$, we make g_i adjacent to all the vertices of P. Then, for each $uv \in \binom{n}{2}$, we make g_i adjacent to f_{uv}^{all} if $uv \in E(G_i)$, and to both $f_{uv}^{\neg u}$ and $f_{uv}^{\neg v}$ otherwise. Finally, we set $\hat{\ell} = 1 + \ell + \binom{n}{2}$. Clearly, the construction can be performed in polynomial time. Moreover, $P \cup F$ is a vertex cover of $\hat{G}$ of size $n\ell + 3\binom{n}{2} = \mathcal{O}(n^2)$. To finish the proof of the theorem it suffices to show that there exists an $\hat{\ell}$-vertex clique in $\hat{G}$ if and only if one of the input graphs G_i contains an ℓ-vertex clique.

To give some intuition, let us first explore how an $\hat{\ell}$-vertex clique may appear in graph $\hat{G}$. Such a clique $\hat{X}$ may contain only one vertex g_i, at most ℓ vertices of P (one for each index a) and at most $\binom{n}{2}$ vertices of F (one for each $uv \in \binom{V}{2}$). However, $\hat{\ell} = 1 + \ell + \binom{n}{2}$; hence, $\hat{X}$ contains *exactly* one vertex g_i, *exactly* one vertex p_v^a for each $1 \le a \le \ell$, and *exactly* one vertex among vertices f_{uv}^{all}, $f_{uv}^{\neg u}$ or $f_{uv}^{\neg v}$ for each $uv \in \binom{V}{2}$. The choice of g_i determines the instance we are actually solving, the choice of the vertices of P encodes a supposed clique in G_i, whereas the vertices in F verify that the chosen vertices of P indeed correspond to a clique in G_i.

Let us now proceed to the formal argumentation.

Assume first that $X = \{x^1, x^2, \ldots, x^\ell\}$ induces an ℓ-vertex clique in some input graph G_i. Construct $\hat{X}$ as follows: first, take $\hat{X} = \{g_i\} \cup \{p_{x^a}^a : 1 \le a \le \ell\}$ and then, for each $uv \in \binom{V}{2}$, add f_{uv}^{all} to $\hat{X}$ if $uv \in E(G_i)$, add $f_{uv}^{\neg u}$ to $\hat{X}$ if $uv \notin E(G_i)$ and $u \notin X$ and add $f_{uv}^{\neg v}$ to $\hat{X}$ if $uv \notin E(G_i)$, $u \in X$ and $v \notin X$. Note that the fact that X induces a clique implies that it is not possible that $uv \notin E(G_i)$ and $u, v \in X$ and hence the presented case distinction is exhaustive. It is straightforward to verify that $\hat{X}$ induces a clique in $\hat{G}$ of size $\hat{\ell} = 1 + \ell + \binom{n}{2}$.

In the other direction, let $\hat{X}$ induce an $\hat{\ell}$-vertex clique in $\hat{G}$. As discussed, $\hat{X}$ needs to contain exactly one vertex g_i, exactly one of the vertices f_{uv}^{all}, $f_{uv}^{\neg u}$ or $f_{uv}^{\neg v}$ for each $uv \in \binom{V}{2}$, and exactly one vertex $p_{x^a}^a$ for each $1 \le a \le \ell$, where the vertices x^a are pairwise different. We claim that $X = \{x^a : 1 \le a \le \ell\}$

is an ℓ-vertex clique in G_i. Clearly $|X| = \ell$, so it remains to prove that $G_i[X]$ is a clique.

By contrary, assume that $x^a x^b \notin E(G_i)$ for some $1 \leq a, b \leq \ell$, $a \neq b$. By the construction, g_i is not adjacent to $f^{\mathrm{all}}_{x^a x^b}$ and, consequently, either $f^{\neg x^a}_{x^a x^b}$ or $f^{\neg x^b}_{x^a x^b}$ belongs to $\hat{X}$. However, the first one is not adjacent to $p^a_{x^a} \in \hat{X}$, and the latter is not adjacent to $p^b_{x^b} \in \hat{X}$. This contradicts the assumption that $\hat{X}$ induces a clique in $\hat{G}$, and concludes the proof of Theorem 15.21. $\square$

15.3 Weak compositions

The tools that we have seen so far are only able to distinguish problems having polynomial kernels from those that do not admit such preprocessing routines. However, is it possible to prove more refined lower bounds for problems which actually *do* possess polynomial kernels? For example, in Section 2.5 we have learned a basic kernelization algorithm for Vertex Cover that returns an instance with at most $2k$ vertices, which therefore can be encoded using $\mathcal{O}(k^2)$ bits. At first glance, it seems implausible to obtain a kernel that can be encoded in a subquadratic number of bits. Can we refine our methodology to develop techniques for proving lower bounds of such type?

The answer to this question is affirmative, and comes via the framework of so-called *weak compositions*. Again, the intuition is very simple. In the previous sections we have not used the full power of the proof of Theorem 15.3. Intuitively, an OR-distillation algorithm has to lose some information from the input instances $x_1, x_2, \ldots, x_t$ even if the bitsize of the output is bounded by $p(\max_{i=1}^t |x_i|) \cdot t^{1-\varepsilon}$ for a polynomial $p(\cdot)$ and any $\varepsilon > 0$. And indeed, it is easy to verify that our proof of Theorem 15.3 can be adjusted even to this weaker notion of OR-distillation. Now assume that somebody gave us a version of cross-composition from some NP-hard language L to the Vertex Cover problem, whose output parameter k is allowed to be roughly proportional to the square root of the number of input instances. More precisely, for any $\delta > 0$ we have some polynomial $p(\cdot)$ such that $k \leq p(\max_{i=1}^t |x_i|) \cdot t^{\frac{1}{2}+\delta}$. If the Vertex Cover problem admitted a compression into bitsize $\mathcal{O}(k^{2-\varepsilon_0})$ for some fixed $\varepsilon_0 > 0$, then by choosing $\delta = \varepsilon_0/4$ and pipelinining the composition with the compression, we would obtain a contradiction with the strengthened version of Theorem 15.3.

This motivates the following definition.

Definition 15.22. Let $L \subseteq \Sigma^*$ be a language and $Q \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized language. We say that L *weakly-cross-composes* into Q if there exists a real constant $d \geq 1$, called the *dimension*, a polynomial equivalence relation $\mathcal{R}$, and an algorithm $\mathcal{A}$, called the *weak cross-composition*, satisfying the following conditions. The algorithm $\mathcal{A}$ takes as input a sequence of strings $x_1, x_2, \ldots, x_t \in \Sigma^*$ that are equivalent with respect to $\mathcal{R}$, runs in time polynomial in $\sum_{i=1}^{t} |x_i|$, and outputs one instance $(y, k) \in \Sigma^* \times \mathbb{N}$ such that:

(a) for every $\delta > 0$ there exists a polynomial $p(\cdot)$ such that for every choice of t and input strings $x_1, x_2, \ldots, x_t$ it holds that $k \le p(\max_{i=1}^{t} |x_i|) \cdot t^{\frac{1}{d}+\delta}$, and
(b) $(y, k) \in Q$ if and only if there exists at least one index i such that $x_i \in L$.

The following theorem easily follows from the described adjustments in the proofs of Theorem 15.3 and Theorem 15.9. We leave its proof as an easy exercise for the reader (Exercises 15.2 and 15.3).

Theorem 15.23. *Assume that an* NP*-hard language L admits a weak cross-composition of dimension d into a parameterized language Q. Assume further that Q admits a polynomial compression with bitsize $\mathcal{O}(k^{d-\varepsilon})$, for some $\varepsilon > 0$. Then* $\mathrm{NP} \subseteq \mathrm{coNP}/\mathrm{poly}$.

Therefore, to refute the existence of a polynomial compression for Vertex Cover with strictly subquadratic bitsize, it suffices to design a weak cross-composition of dimension 2 from some NP-hard problem to Vertex Cover. It appears that we will be able to elegantly streamline the construction by choosing the starting problem wisely. The problem of our interest is called Multicolored Biclique: We are given a bipartite graph $G = (A, B, E)$ such that A is partitioned into k color classes $A_1, A_2, \ldots, A_k$, each of size n, and B is partitioned into k color classes $B_1, B_2, \ldots, B_k$, each of size n. The question is whether one can pick one vertex from each class A_i and one vertex from each class B_i so that the graph induced by the chosen vertices is isomorphic to a complete bipartite graph $K_{k,k}$. The Multicolored Biclique problem is known to be NP-hard.

Lemma 15.24. *There exists a weak cross-composition of dimension* 2 *from the* Multicolored Biclique *problem into the* Vertex Cover *problem parameterized by the solution size.*

Proof. By choosing the polynomial equivalence relation appropriately, we can assume that we are given a sequence of well-formed instances $I^1, I^2, \ldots, I^t$ having the same values of k and n. More precisely, each I^j contains a bipartite graph $G^j = (A^j, B^j, E^j)$, each A^j is partitioned into classes $A_1^j, A_2^j, \ldots, A_k^j$, each B^j is partitioned into classes $B_1^j, B_2^j, \ldots, B_k^j$, and $|A_i^j| = |B_i^j| = n$ for each $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, t$. By duplicating some instances if necessary, we assume that $t = s^2$ for some integer s.

Construct now a graph H as follows. The vertex set of H consists of $2s$ classes of vertices $\hat{A}^1, \hat{A}^2, \ldots, \hat{A}^s, \hat{B}^1, \hat{B}^2, \ldots, \hat{B}^s$, each of size kn. Each class $\hat{A}^\ell$ is further partitioned into classes $\hat{A}_1^\ell, \hat{A}_2^\ell, \ldots, \hat{A}_k^\ell$, each of size n, and the same is performed for the classes $\hat{B}^\ell$ as well. The intuition now is that we have s^2 input instances I^j, and s^2 pairs $(\hat{A}^{j_1}, \hat{B}^{j_2})$ that look exactly as the sides of the bipartition of the graph G^j. Hence, we will pack the information

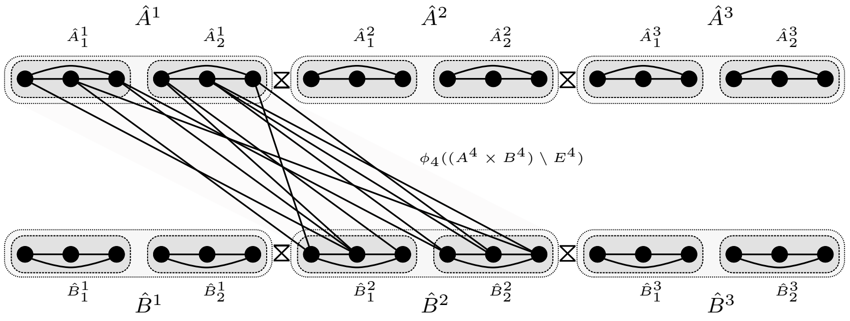


Fig. 15.7: The graph H in the proof of Theorem 15.23 constructed for $k = 2$, $n = 3$, and $t = 9$; the highlighted part contains the set of edges between $\hat{A}^1$ and $\hat{B}^2$ that is constructed in (c) to encode the input graph G^4. To avoid confusion, the complete bipartite graphs between every pair of classes $\hat{A}^\ell$ and between every pair of classes $\hat{B}^\ell$ have been depicted symbolically as small $K_{2,2}$s

about the edges between A^j and B^j from each input instance I^j into the edge space between $\hat{A}^{j_1}$ and $\hat{B}^{j_2}$ for a different pair of indices (j_1, j_2).

Let us fix an arbitrarily chosen bijection τ between $[s^2]$ and $[s] \times [s]$. The edge set of H is defined as follows:

(a) For every $1 \leq \ell_1 < \ell_2 \leq s$, introduce an edge between every vertex of $\hat{A}^{\ell_1}$ and every vertex of $\hat{A}^{\ell_2}$, and an edge between every vertex of $\hat{B}^{\ell_1}$ and every vertex of $\hat{B}^{\ell_2}$.
(b) For every $1 \leq \ell \leq s$ and every $1 \leq i \leq k$, turn the sets $\hat{A}^\ell_i$ and $\hat{B}^\ell_i$ into cliques.
(c) For every $1 \leq j \leq t$, let $\tau(j) = (j_1, j_2)$. Fix any bijection ϕ_j that, for each $i = 1, 2, \ldots, k$, matches vertices of A^j_i with vertices of $\hat{A}^{j_1}_i$, and vertices of B^j_i with vertices of $\hat{B}^{j_2}_i$. For every $u \in A^j$ and $v \in B^j$, put an edge between $\phi_j(u)$ and $\phi_j(v)$ if and only if the edge uv is *not* present in G^j.

The role of edges introduced in (a) is to enforce that the solution has a nontrivial behavior for exactly one of the pairs $(\hat{A}^{j_1}, \hat{B}^{j_2})$, which will emulate the choice of the instance to be satisfied. Edges introduced in (b) and (c) encode the instance itself.

Let us fix the budget for the resulting Vertex Cover instance as $\hat{k} = |V(H)| - 2k = 2snk - 2k$. Since $s = \sqrt{t}$, this means that the obtained cross-composition will have dimension 2. We are left with formally verifying that the obtained instance $(H, \hat{k})$ of Vertex Cover is equivalent to the OR of the input instances of Multicolored Biclique.

Claim 15.25. *If there exists an index j, $1 \leq j \leq t$, such that I^j is a yes-instance of* Multicolored Biclique*, then $(H, \hat{k})$ is a yes-instance of* Vertex Cover*.*

Proof. Let Y^j be the vertex set of a $K_{k,k}$-biclique in G^j, where Y^j contains exactly one vertex from each color class $A_1^j, A_2^j, \ldots, A_k^j, B_1^j, B_2^j, \ldots, B_k^j$. Let us define $X = V(H) \setminus \phi_j(Y^j)$. Since $|Y^j| = 2k$, we have that $|X| = \hat{k}$. We are going to show that X is a vertex cover of H, which will certify that $(H, \hat{k})$ is a yes-instance. We prove an equivalent statement that $Y = V(H) \setminus X = \phi_j(Y^j)$ is an independent set in H.

This, however, follows directly from the construction of H. Let $(j_1, j_2) = \tau(j)$ and let us take any distinct $u, v \in Y$. If $u, v \in \hat{A}^{j_1}$, then u and v have to belong to two different classes among $\hat{A}_1^{j_1}, \hat{A}_2^{j_1}, \ldots, \hat{A}_k^{j_1}$, and there is no edge between any pair of these classes. The same reasoning also holds when $u, v \in \hat{B}^{j_2}$. However, if $u \in \hat{A}^{j_1}$ and $v \in \hat{B}^{j_2}$, then we have that $\phi_j^{-1}(u)\phi_j^{-1}(v) \in E^j$, because Y^j induces a biclique in G^j. By the construction of H it follows that $uv \notin E(H)$. ⌟

Claim 15.26. *If $(H, \hat{k})$ is a yes-instance of* Vertex Cover*, then there exists an index j, $1 \leq j \leq t$, such that I^j is a yes-instance of* Multicolored Biclique*.*

Proof. Let X be a vertex cover of H of size at most $\hat{k}$. Hence, $Y = V(H) \setminus X$ is an independent set in H of size at least $2k$.

We first claim that there are two indices $1 \leq j_1, j_2 \leq s$ such that $Y \subseteq \hat{A}^{j_1} \cup \hat{B}^{j_2}$. Indeed, if there were two different indices j_1' and j_1'' such that there existed some $u \in Y \cap \hat{A}^{j_1'}$ and $v \in Y \cap \hat{A}^{j_1''}$, then by the construction of H we would have that $uv \in E(H)$, which contradicts the fact that Y is an independent set. The same reasoning can be performed for the sets $\hat{B}^{\ell}$. Let $j = \tau^{-1}(j_1, j_2)$ then; since we have that $Y \subseteq \hat{A}^{j_1} \cup \hat{B}^{j_2}$, we can define $Y^j = \phi_j^{-1}(Y)$. The goal is to show that Y^j is a solution to the instance I^j.

We now observe that Y^j contains at most one vertex from each color class $A_1^j, A_2^j, \ldots, A_k^j, B_1^j, B_2^j, \ldots, B_k^j$. This follows directly from the fact that each of the classes $\hat{A}_1^{j_1}, \hat{A}_2^{j_1}, \ldots, \hat{A}_k^{j_1}, \hat{B}_1^{j_2}, \hat{B}_2^{j_2}, \ldots, \hat{B}_k^{j_2}$ induces a clique in H. Since $|Y^j| = |Y| \geq 2k$ and there are $2k$ color classes, we infer that Y^j contains *exactly* one vertex from each color class.

Finally, pick any $u \in Y^j \cap A^j$ and $v \in Y^j \cap B^j$, and observe that since $\phi_j(u)$ and $\phi_j(v)$ are not adjacent in H, the vertices u and v are adjacent in G^j. Consequently, Y^j induces a $K_{k,k}$-biclique in G^j that contains one vertex from each color class, and I_j is hence a yes-instance of Multicolored Biclique. ⌟

Claims 15.25 and 15.26 conclude the proof of Lemma 15.24. □

From Theorem 15.23 and Lemma 15.24 we can derive the following corollary.

Corollary 15.27. *For any $\varepsilon > 0$, the* VERTEX COVER *problem parameterized by the solution size does not admit a polynomial compression with bitsize $\mathcal{O}(k^{2-\varepsilon})$, unless* $\mathrm{NP} \subseteq \mathrm{coNP/poly}$.

Observe now that the lower bounds on the bitsize of a compression can be transferred in a similar manner via polynomial parameter transformations as we have seen in Section 15.2.2. It is just the polynomial dependence of the output parameter on the input one that determines the strength of the obtained lower bound for the target problem. For instance, there is a simple polynomial parameter transformation that takes as input an instance (G, k) of VERTEX COVER and outputs an equivalent instance (G', k) of FEEDBACK VERTEX SET: To obtain G', take every edge $uv \in E(G)$ and add a vertex w_{uv} that is adjacent only to u and to v, thus creating a triangle that must be hit by the solution. By pipelining this reduction with Corollary 15.27, we obtain the following result.

Corollary 15.28. *For any $\varepsilon > 0$, the* FEEDBACK VERTEX SET *problem parameterized by the solution size does not admit a polynomial compression with bitsize $\mathcal{O}(k^{2-\varepsilon})$, unless* $\mathrm{NP} \subseteq \mathrm{coNP/poly}$.

As with classic cross-compositions, we can refute the possibility that an algorithm with a too weak computational power packs the information contained in the input instance into a too small bitsize of the output instance. However, again we have no control over how the information needs to be organized in, say, the kernel. For instance, Corollaries 15.27 and 15.28 show that obtaining kernels for the VERTEX COVER and FEEDBACK VERTEX SET problems with a strictly subquadratic number of edges is implausible (since $\mathcal{O}(k^{2-\varepsilon})$ edges can be encoded in $\mathcal{O}(k^{2-\varepsilon} \log k)$ bits). However, for VERTEX COVER we know a kernel that has a linear number of vertices, namely $2k$, whereas no such kernel is known for FEEDBACK VERTEX SET: the best known one (given in Chapter 9) has $\mathcal{O}(k^2)$ vertices. It seems hard to understand, using solely weak compositions as a tool, whether obtaining a kernel for FEEDBACK VERTEX SET with a subquadratic number of vertices is possible or not.

We state the following theorem for the convenience of the reader, as it provides a good set of basic problems for further reductions. In the PERFECT d-SET MATCHING problem we are given a universe U, of size divisible by d, and a family $\mathcal{F} \subseteq 2^U$ of subsets of U, each of size d. The question is whether there exists a subfamily $\mathcal{G} \subseteq \mathcal{F}$ of size $|U|/d$ such that sets in $\mathcal{G}$ are pairwise disjoint.

Theorem 15.29 ([129, 130]). *Let $\varepsilon > 0$ be any constant. Unless* $\mathrm{NP} \subseteq \mathrm{coNP/poly}$, *the following statements hold:*

(a) For any $q \geq 3$, the q-SAT problem parameterized by the number of variables n does not have a compression with bitsize $\mathcal{O}(n^{q-\varepsilon})$.

*(b) For any $d \geq 2$, the d-*Hitting Set *problem parameterized by $|U|$ does not have a compression with bitsize $\mathcal{O}(|U|^{d-\varepsilon})$.*
(c) For any $d \geq 3$, the Perfect d-Set Matching *problem parameterized by $|U|$ does not have a compression with bitsize $\mathcal{O}(|U|^{d-\varepsilon})$.*

Note that for all the considered problems, already the trivial kernelization algorithms, which just remove duplicates of clauses or sets in the input, match the presented lower bounds.

The proofs of the presented statements are more difficult, and hence omitted from this book. Note that statement (b) for $d = 2$ is equivalent to Corollary 15.27. The proof for $d > 2$ can be obtained using the same strategy, or derived by an easy reduction from statement (a). The known proofs of statement (a), which was actually the original motivation for introducing the weak composition framework, use quite advanced results from number theory. The proof of statement (c) is a fairly complicated weak cross-composition; however, it is self-contained and does not use any external results.

Exercises

15.1 (✎). In a few places (e.g., Steiner Tree parameterized by either ℓ or $|K|$) we have used the following argument: If a problem P admits two parameterizations k_1 and k_2, and k_1 is always bounded polynomially in k_2 in nontrivial instances, then a result that refutes the existence of a polynomial compression for the parameterization by k_2 also does the same for the parameterization by k_1. Prove this statement formally, using the notion of polynomial parameter transformation.

15.2. Prove that Theorem 15.3 holds even if the output string of an OR-distillation is allowed to have length at most $p(\max_{i=1}^{t} |x_i|) \cdot t^{1-\varepsilon}$, for some polynomial $p(\cdot)$ and any $\varepsilon > 0$. Would it be possible to use $o(t)$ instead of $t^{1-\varepsilon}$?

15.3 (✎). Derive Theorem 15.23 from Exercise 15.2.

15.4. In this meta-exercise you are asked to prove that the following parameterized problems do not admit polynomial compressions unless NP $\subseteq$ coNP/ poly. In each case, we give problem name, the parameter in parentheses, and the problem definition. For each problem, you may assume that it is NP-hard (without proof).

Some important definitions: We say that a graph G is *d-degenerate* for some positive integer d if each subgraph of G contains a vertex of degree at most d. For example, forests are 1-degenerate and planar graphs are 5-degenerate. A *bull* is a 5-vertex graph H with $V(H) = \{a, b, c, d, e\}$ and $E(H) = \{ab, bc, ac, bd, ce\}$. A *proper coloring* of a graph G with q colors is a function $c : V(G) \to [q]$ such that $c(u) \neq c(v)$ whenever $uv \in E(G)$.

1. (✎) Directed Max Leaf (k): Does there exist an outbranching with at least k leaves in a given directed graph G? An *outbranching* is a directed graph that (a) is a tree if we suppress the directions of the edges, and (b) is rooted at one vertex, and all edges are directed away from the root.
2. (✎) Max Leaf Subtree (k): Does there exist a subtree of a given graph G with at least k leaves?
3. Connected Vertex Cover (k): Does there exist a set X of at most k vertices in the given graph G, such that $G[X]$ is connected and $G \setminus X$ is edgeless?

4. Connected Feedback Vertex Set (k): Does there exist a set X of at most k vertices in the given graph G, such that $G[X]$ is connected and $G \setminus X$ is a forest?
5. Connected Bull Hitting (k): Does there exist a set of vertices X in the given graph G such that $G[X]$ is connected and $G \setminus X$ does not contain a bull as an induced subgraph?
6. 2-deg Connected Feedback Vertex Set (k): The same as Connected Feedback Vertex Set, but the input graph is required to be 2-degenerate.
7. 2-deg Steiner Tree(k): The same as Steiner Tree, but the input graph is required to be 2-degenerate.
8. 2-deg Connected Dominating Set(k): Does there exist a set of vertices X in the given 2-degenerate graph G such that $G[X]$ is connected and $N_G[X] = V(G)$?
9. Set Packing ($|U|$): Do there exist k pairwise disjoint sets in a given family $\mathcal{F}$ of subsets of some universe U?
10. Set Cover ($|\mathcal{F}|$): Do there exist ℓ sets in a given family $\mathcal{F}$ of subsets of some universe U that together cover the whole U?
11. Directed Edge Multiway Cut with 2 terminals(k): Does there exist a set X of at most k edges of the given directed graph G with designated terminals s and t, such that in $G \setminus X$ there does not exist a (directed) path from s to t nor from t to s?
12. Disjoint Factors ($|\Gamma|$): Do there exist pairwise disjoint subwords $u_1, u_2, \ldots, u_s$ of the input word w over alphabet $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_s\}$ such that each u_i is of length at least 2 and begins and ends with γ_i?
13. Vertex Disjoint Paths (k): Do there exist vertex-disjoint paths $P_1, P_2, \ldots, P_k$ in the input graph G with designated terminal pairs (s_1, t_1), (s_2, t_2), $\ldots$, (s_k, t_k), such that P_i connects s_i with t_i for each i?
14. (☠) Chromatic Number (VC): Does there exist a proper coloring of the input graph G with q colors? The input consists of the graph G, the integer q and a set $Z \subseteq V(G)$ that is a vertex cover of G (i.e., $G \setminus Z$ is edgeless).
15. (☠) Feedback Vertex Set (Cluster Distance): Does there exist a set $X \subseteq V(G)$ of size at most ℓ such that $G \setminus X$ is a forest? The input consists of the graph G, the integer ℓ, and a set $Z \subseteq V(G)$ such that $G \setminus Z$ is a cluster graph (i.e., each connected component of $G \setminus Z$ is a clique).
16. (☠) Edge Clique Cover (k): Do there exist k subgraphs $K_1, K_2, \ldots, K_k$ of the input graph G such that each K_i is a clique and $E(G) = \bigcup_{i=1}^{k} E(K_i)$?
17. Subset Sum ($k + \log m$): Does there exist a subset X of at most k numbers from the given set S of positive integers, such that $\sum_{x \in X} x$ equals exactly a number m prescribed in the input?
18. (☠) s-Way Cut (k): Does there exist a set X of at most k edges of the input graph G, such that $G \setminus X$ has at least s connected components? Both numbers s and k are given as input.
19. (☠) Eulerian Deletion (k): Does there exist a set of at most k edges in the input directed graph G, such that $G \setminus X$ is Eulerian (i.e., weakly connected and each vertex has equal in- and outdegree)?

15.5. The Odd Cycle Transversal problem asks for a set X of at most k vertices of the input graph G such that $G \setminus X$ is bipartite. Prove that, unless NP $\subseteq$ coNP/ poly, for any $\varepsilon > 0$ Odd Cycle Transversal does not admit a kernel with $\mathcal{O}(k^{2-\varepsilon})$ edges.

15.6. Prove that there exists a function $f : \mathbb{Z}_+ \to \mathbb{Z}_+$ with $f(d) \to \infty$ as $d \to \infty$ such that if for some $d \geq 2$ there exists a kernel of size $\mathcal{O}(k^{f(d)})$ for any of the following two problems, restricted to d-degenerate graphs, then NP $\subseteq$ coNP/ poly:

1. Connected Vertex Cover,
2. Dominating Set.

Hints

15.2 Follow the same proof strategy, but modify the choice of t.

15.4

1 Try disjoint union.
2 Try disjoint union.
3 Reduce from Set Cover parameterized by the size of the universe.
4, 5, 6 Reduce from Connected Vertex Cover.
7, 8 Recall that Colorful Graph Motif is NP-hard already on trees. Use this result to improve the kernelization lower bound for Colorful Graph Motif and reduce from this problem.
9 Adjust the cross-composition for Set Cover parameterized by the size of the universe.
10 Define a colorful version of the problem and prove that it is equivalent to the original one. Focus on proving hardness for the colorful version. Design an instance selector consisting of $2 \log t$ additional sets in $\log t$ new colors. The chosen sets of these colors should cover completely the universes of all the input instances apart from exactly one, which should be covered by 'regular' sets.
11 Compose the input instances 'sequentially'. However, enrich the input instances before performing the composition, in order to make sure that the solution to the composed instance will not be spread among several of the input instances, but will rather be concentrated in one instance.
12 Start the composition with instances over the same alphabet Γ, and add $\log t$ symbols to Γ to emulate an instance selector.
13 Reduce from Disjoint Factors.
14 Carefully prepare your input language for composition. For example, consider the task of checking whether a planar graph is properly colorable with four colors, which is NP-hard. Note that it is definitely colorable with four colors. Design an instance selector that allows all input instances to use four colors, except for one instance that may use only three.
15 Reduce from a colorful variant of the Set Packing problem.
16 Use an AND-cross-composition rather than an OR-cross-composition. You need a careful construction to ensure that all the edges of the resulting instance are covered. For this, use the following observation: If the input graph contains a simplicial vertex, then we can assume that any solution contains its closed neighborhood as one of the cliques. Thus, by adding simplicial vertices we can create cliques of edges that will be covered for sure.
17 Make a cross-composition from the same problem, but watch out for the bound on the number of equivalence classes in the polynomial equivalence relation — you cannot directly classify the instances according to their target value m. How to go around this problem? Then, design an instance selector to ensure that all integers in the solution come from the same input instance.
18 First, show that a weighted variant, where each edge has small integer weight, is equivalent. Then, cross-compose from Clique to the weighted variant. Your intended solution should pick one input instance, and cut out all vertices *not contained* in the supposed clique as isolated vertices, and leave the rest of the graph as one large connected component.
19 Prove NP-hardness and cross-compose the following auxiliary problem: given a directed graph G with terminals s and t and some *forbidden pairs* $F \subseteq \binom{E(G)}{2}$, does there exist a path from s to t that, for any $e_1e_2 \in F$, does not contain *both* edges e_1 and e_2.

15.5 Reduce from Vertex Cover.

15.6 Reduce from any of the problems considered in Theorem 15.29.

Bibliographic notes

For an introduction to complexity theory, and in particular to the concepts used in this chapter, we refer the reader to the recent book of Arora and Barak [21]. The fact that $\mathrm{NP} \subseteq \mathrm{coNP/poly}$ implies that the polynomial hierarchy collapses to its third level is known as Yap's theorem [440].

Theorem 15.3 was proved by Fortnow and Santhanam [213]. The first framework for proving kernelization lower bounds that builds upon the work of Fortnow and Santhanam was proposed by Bodlaender, Downey, Fellows, and Hermelin [50]. This technique, called OR-*composition*, is a slightly weaker form of the cross-composition framework that was presented in this chapter. More precisely, the formalism presented in [50] does not use polynomial equivalence relations, assumes that the source and target languages of a composition are the same, and does not allow the output parameter to depend poly-logarithmically on the number of input instances. Later it turned out that the lacking features are helpful in streamlining compositions for many problems, and many authors developed ad hoc, problem-dependant methods for justifying them. The cross-composition framework, which was eventually proposed by Bodlaender, Jansen, and Kratsch [53], elegantly explains why these features can be added in general. The fact that the whole approach should work when the OR function is replaced with AND was initially conjectured by Bodlaender et al. [50]; the statement had been dubbed the *AND-conjecture*. The conjecture was eventually resolved by Drucker [160], who proved Theorem 15.11. We note that Drucker only states that the existence of an AND-distillation for an NP-hard problem implies that $\mathrm{NP} \subseteq \mathrm{coNP/poly}$, but the slightly stronger statement of Theorem 15.11 also follows from his results. A shorter proof of the result of Drucker was recently announced by Dell [128].

The idea of an instance selector originates in the work of Dom, Lokshtanov, and Saurabh [141]. The lower bound for Set Splitting was first obtained by Cygan, Pilipczuk, Pilipczuk, and Wojtaszczyk [123] via a PPT from Set Cover parameterized by the size of the family, i.e., Exercise 15.4.10. The lower bound for Steiner Tree was first proved by Dom, Lokshtanov, and Saurabh [141], again via a PPT from Set Cover parameterized by the size of the family. The proof using Graph Motif as a pivot problem was proposed by Cygan, Pilipczuk, Pilipczuk, and Wojtaszczyk [121], and the main point was that the resulting graph is 2-degenerate. The example of Set Cover parameterized by the size of the universe has been taken from Dom, Lokshtanov, and Saurabh [141]; we note that the original composition was slightly more complicated. Clique parameterized by vertex cover was one of the first examples of the application of the cross-composition framework, and was given by Bodlaender, Jansen and Kratsch [53]. For Exercise 15.4, point 1 originates in [35], points 3, 9, 10, 17 originate in [141], points 7, 8 originate in [121], points 11, 16, 18 originate in [114], points 12, 13 originate in [56], point 14 originates in [276], point 15 originates in [53], and point 19 originates in [117].

The idea of weak compositions originates in the work of Dell and van Melkebeek [130], who proved statements (a) and (b) of Theorem 15.29, and derived Corollary 15.27 from it. The weak composition framework, as presented in this chapter, was introduced later, independently by Dell and Marx [129] and by Hermelin and Wu [260]. In particular, Dell and Marx [129] proved statement (c) of Theorem 15.29 and re-proved statement (b) in a simplified manner. The presented lower bound for kernelization of Vertex Cover is also taken from [129].

References

1. Abrahamson, K.R., Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness IV: On completeness for W[P] and PSPACE analogues. Ann. Pure Appl. Logic **73**(3), 235–276 (1995)
2. Abu-Khzam, F.N.: An improved kernelization algorithm for r-Set Packing. Information Processing Letters **110**(16), 621–624 (2010)
3. Abu-Khzam, F.N.: A kernelization algorithm for d-Hitting Set. J. Computer and System Sciences **76**(7), 524–531 (2010)
4. Abu-Khzam, F.N., Collins, R.L., Fellows, M.R., Langston, M.A., Suters, W.H., Symons, C.T.: Kernelization algorithms for the vertex cover problem: Theory and experiments. In: Proceedings of the 6th Workshop on Algorithm Engineering and Experiments and the 1st Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALC), pp. 62–69. SIAM (2004)
5. Adler, I., Grohe, M., Kreutzer, S.: Computing excluded minors. In: Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 641–650. SIAM (2008)
6. Adler, I., Kolliopoulos, S.G., Krause, P.K., Lokshtanov, D., Saurabh, S., Thilikos, D.M.: Tight bounds for linkages in planar graphs. In: Proceedings of the 38th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 6755, pp. 110–121. Springer (2011)
7. Aigner, M.: A course in enumeration, *Graduate Texts in Mathematics*, vol. 238. Springer, Berlin (2007)
8. Alber, J., Bodlaender, H.L., Fernau, H., Kloks, T., Niedermeier, R.: Fixed parameter algorithms for dominating set and related problems on planar graphs. Algorithmica **33**(4), 461–493 (2002)
9. Alber, J., Fiala, J.: Geometric separation and exact solutions for the parameterized independent set problem on disk graphs. J. Algorithms **52**(2), 134–151 (2004)
10. Alon, N., Babai, L., Itai, A.: A fast and simple randomized parallel algorithm for the maximal independent set problem. J. Algorithms **7**(4), 567–583 (1986)
11. Alon, N., Gutin, G., Kim, E.J., Szeider, S., Yeo, A.: Solving MAX-r-SAT above a tight lower bound. Algorithmica **61**(3), 638–655 (2011)
12. Alon, N., Gutner, S.: Balanced hashing, color coding and approximate counting. In: Proceedings of the 4th International Workshop on Parameterized and Exact Computation (IWPEC), *Lecture Notes in Comput. Sci.*, vol. 5917, pp. 1–16. Springer (2009)
13. Alon, N., Gutner, S.: Balanced families of perfect hash functions and their applications. ACM Transactions on Algorithms **6**(3) (2010)
14. Alon, N., Lokshtanov, D., Saurabh, S.: Fast FAST. In: Proceedings of the 36th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 5555, pp. 49–58. Springer (2009)
15. Alon, N., Yuster, R., Zwick, U.: Color-coding. J. ACM **42**(4), 844–856 (1995)
16. Amini, O., Fomin, F.V., Saurabh, S.: Counting subgraphs via homomorphisms. SIAM J. Discrete Math. **26**(2), 695–717 (2012)
17. Amir, E.: Approximation algorithms for treewidth. Algorithmica **56**, 448–479 (2010)
18. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Alg. Disc. Meth. **8**, 277–284 (1987)
19. Arnborg, S., Lagergren, J., Seese, D.: Easy problems for tree-decomposable graphs. J. Algorithms **12**(2), 308–340 (1991)
20. Arnborg, S., Proskurowski, A.: Linear time algorithms for NP-hard problems restricted to partial k-trees. Discrete Applied Mathematics **23**(1), 11–24 (1989)
21. Arora, S., Barak, B.: Computational Complexity — A Modern Approach. Cambridge University Press (2009)

22. Arora, S., Grigni, M., Karger, D.R., Klein, P.N., Woloszyn, A.: A polynomial-time approximation scheme for weighted planar graph TSP. In: Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 33–41. SIAM (1998)
23. Baker, B.S.: Approximation algorithms for NP-complete problems on planar graphs. J. ACM **41**(1), 153–180 (1994)
24. Balasubramanian, R., Fellows, M.R., Raman, V.: An improved fixed-parameter algorithm for vertex cover. Information Processing Letters **65**(3), 163–168 (1998)
25. Baste, J., Sau, I.: The role of planarity in connectivity problems parameterized by treewidth. In: Proceedings of the 9th International Symposium on Parameterized and Exact Computation (IPEC), *Lecture Notes in Comput. Sci.*, vol. 8894, pp. 63–74. Springer (2014)
26. Bateni, M., Hajiaghayi, M., Marx, D.: Approximation schemes for Steiner forest on planar graphs and graphs of bounded treewidth. J. ACM **58**(5), 21 (2011)
27. Bax, E.T.: Inclusion and exclusion algorithm for the Hamiltonian path problem. Information Processing Letters **47**(4), 203–207 (1993)
28. Bax, E.T.: Algorithms to count paths and cycles. Information Processing Letters **52**(5), 249–252 (1994)
29. Becker, A., Bar-Yehuda, R., Geiger, D.: Randomized algorithms for the loop cutset problem. J. Artificial Intelligence Res. **12**, 219–234 (2000)
30. Ben-Ari, M.: Mathematical Logic for Computer Science, 3rd edn. Springer (2012)
31. Bertelè, U., Brioschi, F.: Nonserial dynamic programming. Academic Press, New York (1972). Mathematics in Science and Engineering, Vol. 91
32. Bessy, S., Fomin, F.V., Gaspers, S., Paul, C., Perez, A., Saurabh, S., Thomassé, S.: Kernels for feedback arc set in tournaments. J. Computer and System Sciences **77**(6), 1071–1078 (2011)
33. Bienstock, D., Langston, M.A.: Algorithmic implications of the graph minor theorem. Handbooks in Operations Research and Management Science **7**, 481–502 (1995)
34. Binkele-Raible, D.: Amortized analysis of exponential time and parameterized algorithms: Measure and conquer and reference search trees. Ph.D. thesis, University of Trier (2010)
35. Binkele-Raible, D., Fernau, H., Fomin, F.V., Lokshtanov, D., Saurabh, S., Villanger, Y.: Kernel(s) for problems with no kernel: On out-trees with many leaves. ACM Transactions on Algorithms **8**(4), 38 (2012)
36. Björklund, A.: Determinant sums for undirected hamiltonicity. SIAM J. Comput. **43**(1), 280–299 (2014)
37. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Fourier meets Möbius: fast subset convolution. In: Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC), pp. 67–74. ACM, New York (2007)
38. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Evaluation of permanents in rings and semirings. Information Processing Letters **110**(20), 867–870 (2010)
39. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Narrow sieves for parameterized paths and packings. CoRR **abs/1007.1161** (2010)
40. Björklund, A., Husfeldt, T., Koivisto, M.: Set partitioning via inclusion-exclusion. SIAM J. Computing **39**(2), 546–563 (2009)
41. Björklund, A., Kaski, P., Kowalik, L.: Probably Optimal Graph Motifs. In: Proceedings of the 30th International Symposium on Theoretical Aspects of Computer Science (STACS), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 20, pp. 20–31. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2013)
42. Bliznets, I., Fomin, F.V., Pilipczuk, M., Pilipczuk, M.: A subexponential parameterized algorithm for Interval Completion. CoRR **abs/1402.3473** (2014)
43. Bliznets, I., Fomin, F.V., Pilipczuk, M., Pilipczuk, M.: A subexponential parameterized algorithm for Proper Interval Completion. In: Proceedings of the 22nd Annual European Symposium on Algorithms (ESA), *Lecture Notes in Comput. Sci.*, vol. 8737, pp. 173–184. Springer (2014)

44. Bodlaender, H.L.: On linear time minor tests with depth-first search. J. Algorithms **14**(1), 1–23 (1993)
45. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Computing **25**(6), 1305–1317 (1996)
46. Bodlaender, H.L.: A partial k-arboretum of graphs with bounded treewidth. Theoretical Computer Science **209**(1-2), 1–45 (1998)
47. Bodlaender, H.L., Bonsma, P., Lokshtanov, D.: The fine details of fast dynamic programming over tree decompositions. In: Proceedings of the 8th International Symposium on Parameterized and Exact Computation (IPEC), *Lecture Notes in Comput. Sci.*, vol. 8246, pp. 41–53. Springer (2013)
48. Bodlaender, H.L., Cygan, M., Kratsch, S., Nederlof, J.: Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. In: Proceedings of the 40th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 7965, pp. 196–207. Springer (2013)
49. Bodlaender, H.L., van Dijk, T.C.: A cubic kernel for feedback vertex set and loop cutset. Theory of Computing Systems **46**(3), 566–597 (2010)
50. Bodlaender, H.L., Downey, R.G., Fellows, M.R., Hermelin, D.: On problems without polynomial kernels. J. Computer and System Sciences **75**(8), 423–434 (2009)
51. Bodlaender, H.L., Downey, R.G., Fomin, F.V., Marx, D. (eds.): The multivariate algorithmic revolution and beyond - Essays dedicated to Michael R. Fellows on the occasion of his 60th birthday, *Lecture Notes in Comput. Sci.*, vol. 7370. Springer (2012)
52. Bodlaender, H.L., Drange, P.G., Dregi, M.S., Fomin, F.V., Lokshtanov, D., Pilipczuk, M.: An $O(c^k n)$ 5-approximation algorithm for treewidth. In: Proceedings of the 54th Annual Symposium on Foundations of Computer Science (FOCS), pp. 499–508. IEEE (2013)
53. Bodlaender, H.L., Jansen, B.M.P., Kratsch, S.: Kernelization lower bounds by cross-composition. SIAM J. Discrete Math. **28**(1), 277–305 (2014)
54. Bodlaender, H.L., Kloks, T.: Efficient and constructive algorithms for the pathwidth and treewidth of graphs. J. Algorithms **21**(2), 358–402 (1996)
55. Bodlaender, H.L., van Leeuwen, E.J., van Rooij, J.M.M., Vatshelle, M.: Faster algorithms on branch and clique decompositions. In: Proceedings of the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS), *Lecture Notes in Comput. Sci.*, vol. 6281, pp. 174–185 (2010)
56. Bodlaender, H.L., Thomassé, S., Yeo, A.: Kernel bounds for disjoint cycles and disjoint paths. Theoretical Computer Science **412**(35), 4570–4578 (2011)
57. Bollobás, B.: On generalized graphs. Acta Math. Acad. Sci. Hungar **16**, 447–452 (1965)
58. Borie, R.B., Parker, R.G., Tovey, C.A.: Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. Algorithmica **7**(5&6), 555–581 (1992)
59. Bousquet, N., Daligault, J., Thomassé, S.: Multicut is FPT. In: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC), pp. 459–468. ACM (2011)
60. Bui-Xuan, B.M., Jones, N.S.: How modular structure can simplify tasks on networks. CoRR **abs/1305.4760** (2013)
61. Burrage, K., Estivill-Castro, V., Fellows, M.R., Langston, M.A., Mac, S., Rosamond, F.A.: The undirected feedback vertex set problem has a poly(k) kernel. In: Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC), *Lecture Notes in Comput. Sci.*, vol. 4169, pp. 192–202. Springer (2006)
62. Buss, J.F., Goldsmith, J.: Nondeterminism within P. SIAM J. Computing **22**(3), 560–572 (1993)
63. Cai, L.: Fixed-parameter tractability of graph modification problems for hereditary properties. Information Processing Letters **58**(4), 171–176 (1996)

64. Cai, L.: Parameterized complexity of cardinality constrained optimization problems. Comput. J. **51**(1), 102–121 (2008)
65. Cai, L., Chan, S.M., Chan, S.O.: Random separation: A new method for solving fixed-cardinality optimization problems. In: Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC), *Lecture Notes in Comput. Sci.*, vol. 4169, pp. 239–250. Springer (2006)
66. Cai, L., Chen, J., Downey, R.G., Fellows, M.R.: Advice classes of parameterized tractability. Ann. Pure Appl. Logic **84**(1), 119–138 (1997)
67. Cai, L., Chen, J., Downey, R.G., Fellows, M.R.: On the parameterized complexity of short computation and factorization. Arch. Math. Log. **36**(4-5), 321–337 (1997)
68. Cai, L., Juedes, D.W.: On the existence of subexponential parameterized algorithms. J. Computer and System Sciences **67**(4), 789–807 (2003)
69. Calinescu, G., Fernandes, C.G., Reed, B.A.: Multicuts in unweighted graphs and digraphs with bounded degree and bounded tree-width. J. Algorithms **48**(2), 333–359 (2003)
70. Cao, Y., Chen, J., Liu, Y.: On feedback vertex set new measure and new structures. In: Proceedings of the 12th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT), *Lecture Notes in Comput. Sci.*, vol. 6139, pp. 93–104 (2010)
71. Cesati, M.: Perfect code is W[1]-complete. Information Processing Letters **81**(3), 163–168 (2002)
72. Cesati, M.: The Turing way to parameterized complexity. J. Computer and System Sciences **67**(4), 654–685 (2003)
73. Chekuri, C., Chuzhoy, J.: Polynomial bounds for the Grid-Minor Theorem. In: Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC), pp. 60–69. ACM (2014)
74. Chekuri, C., Ene, A.: Approximation algorithms for submodular multiway partition. In: Proceedings of the 52nd Annual Symposium on Foundations of Computer Science (FOCS), pp. 807–816. IEEE (2011)
75. Chekuri, C., Ene, A.: Submodular cost allocation problem and applications. In: Proceedings of the 38th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 6755, pp. 354–366. Springer (2011)
76. Chen, J., Chor, B., Fellows, M.R., Huang, X., Juedes, D.W., Kanj, I.A., Xia, G.: Tight lower bounds for certain parameterized NP-hard problems. Information and Computation **201**(2), 216–231 (2005)
77. Chen, J., Fomin, F.V., Liu, Y., Lu, S., Villanger, Y.: Improved algorithms for feedback vertex set problems. J. Computer and System Sciences **74**(7), 1188–1198 (2008)
78. Chen, J., Huang, X., Kanj, I.A., Xia, G.: Linear FPT reductions and computational lower bounds. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC), pp. 212–221. ACM (2004)
79. Chen, J., Huang, X., Kanj, I.A., Xia, G.: On the computational hardness based on linear FPT-reductions. J. Comb. Optim. **11**(2), 231–247 (2006)
80. Chen, J., Huang, X., Kanj, I.A., Xia, G.: Strong computational lower bounds via parameterized complexity. J. Computer and System Sciences **72**(8), 1346–1367 (2006)
81. Chen, J., Kanj, I.A., Jia, W.: Vertex cover: further observations and further improvements. J. Algorithms **41**(2), 280–301 (2001)
82. Chen, J., Kanj, I.A., Xia, G.: Improved upper bounds for vertex cover. Theoretical Computer Science **411**(40-42), 3736–3756 (2010)
83. Chen, J., Kneis, J., Lu, S., Mölle, D., Richter, S., Rossmanith, P., Sze, S.H., Zhang, F.: Randomized divide-and-conquer: improved path, matching, and packing algorithms. SIAM J. Computing **38**(6), 2526–2547 (2009)
84. Chen, J., Liu, Y., Lu, S.: An improved parameterized algorithm for the minimum node multiway cut problem. Algorithmica **55**(1), 1–13 (2009)
85. Chen, J., Liu, Y., Lu, S., O'Sullivan, B., Razgon, I.: A fixed-parameter algorithm for the directed feedback vertex set problem. J. ACM **55**(5) (2008)

86. Chen, J., Lu, S., Sze, S.H., Zhang, F.: Improved algorithms for path, matching, and packing problems. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 298–307. SIAM (2007)
87. Cheung, H.Y., Lau, L.C., Leung, K.M.: Algebraic algorithms for linear matroid parity problems. ACM Transactions on Algorithms **10**(3), 10 (2014)
88. Chimani, M., Mutzel, P., Zey, B.: Improved Steiner tree algorithms for bounded treewidth. J. Discrete Algorithms **16**, 67–78 (2012)
89. Chitnis, R.H., Cygan, M., Hajiaghayi, M., Marx, D.: Directed subset feedback vertex set is fixed-parameter tractable. In: Proceedings of the 39th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 7391, pp. 230–241. Springer (2012)
90. Chitnis, R.H., Egri, L., Marx, D.: List H-coloring a graph by removing few vertices. In: Proceedings of the 21st Annual European Symposium on Algorithms (ESA), *Lecture Notes in Comput. Sci.*, vol. 8125, pp. 313–324. Springer (2013)
91. Chitnis, R.H., Hajiaghayi, M., Marx, D.: Fixed-parameter tractability of directed multiway cut parameterized by the size of the cutset. SIAM J. Computing **42**(4), 1674–1696 (2013)
92. Chitnis, R.H., Hajiaghayi, M., Marx, D.: Tight bounds for planar strongly connected Steiner subgraph with fixed number of terminals (and extensions). In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1782–1801. SIAM (2014)
93. Chlebík, M., Chlebíková, J.: Crown reductions for the minimum weighted vertex cover problem. Discrete Applied Mathematics **156**(3), 292–312 (2008)
94. Chor, B., Fellows, M.R., Juedes, D.W.: Linear kernels in linear time, or how to save k colors in $O(n^2)$ steps. In: Proceedings of the 30th Workshop on Graph-Theoretic Concepts in Computer Science (WG), *Lecture Notes in Comput. Sci.*, vol. 3353, pp. 257–269. Springer (2004)
95. Clarkson, K.L.: Las Vegas algorithms for linear and integer programming when the dimension is small. J. ACM **42**(2), 488–499 (1995)
96. Cohen, N., Fomin, F.V., Gutin, G., Kim, E.J., Saurabh, S., Yeo, A.: Algorithm for finding k-vertex out-trees and its application to k-internal out-branching problem. J. Computer and System Sciences **76**(7), 650–662 (2010)
97. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (3rd ed.). MIT Press (2009)
98. Courcelle, B.: The monadic second-order logic of graphs I: Recognizable sets of finite graphs. Information and Computation **85**, 12–75 (1990)
99. Courcelle, B.: The monadic second-order logic of graphs III: Treewidth, forbidden minors and complexity issues. Informatique Théorique **26**, 257–286 (1992)
100. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach. Cambridge University Press (2012)
101. Courcelle, B., Makowsky, J.A., Rotics, U.: On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. Discrete Applied Mathematics **108**(1-2), 23–52 (2001)
102. Courcelle, B., Olariu, S.: Upper bounds to the clique width of graphs. Discrete Applied Mathematics **101**(1-3), 77–114 (2000)
103. Crowston, R., Fellows, M.R., Gutin, G., Jones, M., Kim, E.J., Rosamond, F.A., Ruzsa, I.Z., Thomassé, S., Yeo, A.: Satisfying more than half of a system of linear equations over GF(2): A multivariate approach. J. Computer and System Sciences **80**(4), 687–696 (2014)
104. Crowston, R., Gutin, G., Jones, M., Kim, E.J., Ruzsa, I.Z.: Systems of linear equations over $\mathbb{F}_2$ and problems parameterized above average. In: Proceedings of the 12th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT), *Lecture Notes in Comput. Sci.*, vol. 6139, pp. 164–175. Springer (2010)
105. Crowston, R., Gutin, G., Jones, M., Raman, V., Saurabh, S.: Parameterized complexity of MaxSat above average. Theoretical Computer Science **511**, 77–84 (2013)

106. Crowston, R., Gutin, G., Jones, M., Yeo, A.: A new lower bound on the maximum number of satisfied clauses in Max-SAT and its algorithmic applications. Algorithmica **64**(1), 56–68 (2012)
107. Crowston, R., Jones, M., Mnich, M.: Max-cut parameterized above the Edwards-Erdős bound. In: Proceedings of the 39th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 7391, pp. 242–253. Springer (2012)
108. Crowston, R., Jones, M., Muciaccia, G., Philip, G., Rai, A., Saurabh, S.: Polynomial kernels for lambda-extendible properties parameterized above the Poljak-Turzik bound. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 24, pp. 43–54. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2013)
109. Cunningham, W.H.: Improved bounds for matroid partition and intersection algorithms. SIAM Journal on Computing **15**(4), 948–957 (1986)
110. Cunningham, W.H., Edmonds, J.: A combinatorial decomposition theory. Canadian J. Math. **32**, 734–765 (1980)
111. Cygan, M.: Deterministic parameterized connected vertex cover. In: Proceedings of the 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT), *Lecture Notes in Comput. Sci.*, vol. 7357, pp. 95–106. Springer (2012)
112. Cygan, M., Dell, H., Lokshtanov, D., Marx, D., Nederlof, J., Okamoto, Y., Paturi, R., Saurabh, S., Wahlström, M.: On problems as hard as CNF-SAT. In: Proceedings of the 27th IEEE Conference on Computational Complexity (CCC), pp. 74–84. IEEE (2012)
113. Cygan, M., Kratsch, S., Nederlof, J.: Fast hamiltonicity checking via bases of perfect matchings. In: Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC), pp. 301–310. ACM (2013)
114. Cygan, M., Kratsch, S., Pilipczuk, M., Pilipczuk, M., Wahlström, M.: Clique cover and graph separation: New incompressibility results. ACM Transactions on Computation Theory **6**(2), 6 (2014)
115. Cygan, M., Lokshtanov, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: Minimum Bisection is fixed parameter tractable. In: Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC), pp. 323–332. ACM (2014)
116. Cygan, M., Marx, D., Pilipczuk, M., Pilipczuk, M.: The planar directed k-Vertex-Disjoint Paths problem is fixed-parameter tractable. In: Proceedings of the 54th Annual Symposium on Foundations of Computer Science (FOCS), pp. 197–207. IEEE (2013)
117. Cygan, M., Marx, D., Pilipczuk, M., Pilipczuk, M., Schlotter, I.: Parameterized complexity of Eulerian deletion problems. Algorithmica **68**(1), 41–61 (2014)
118. Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., van Rooij, J.M.M., Wojtaszczyk, J.O.: Solving connectivity problems parameterized by treewidth in single exponential time. In: Proceedings of the 52nd Annual Symposium on Foundations of Computer Science (FOCS), pp. 150–159. IEEE (2011)
119. Cygan, M., Philip, G., Pilipczuk, M., Pilipczuk, M., Wojtaszczyk, J.O.: Dominating Set is fixed parameter tractable in claw-free graphs. Theoretical Computer Science **412**(50), 6982–7000 (2011)
120. Cygan, M., Pilipczuk, M., Pilipczuk, M.: Known algorithms for Edge Clique Cover are probably optimal. In: Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1044–1053. SIAM (2013)
121. Cygan, M., Pilipczuk, M., Pilipczuk, M., Wojtaszczyk, J.O.: Kernelization hardness of connectivity problems in d-degenerate graphs. Discrete Applied Mathematics **160**(15), 2131–2141 (2012)
122. Cygan, M., Pilipczuk, M., Pilipczuk, M., Wojtaszczyk, J.O.: On multiway cut parameterized above lower bounds. Theory of Computing Systems **5**(1), 3 (2013)

123. Cygan, M., Pilipczuk, M., Pilipczuk, M., Wojtaszczyk, J.O.: Solving the 2-Disjoint Connected Subgraphs problem faster than 2^n. Algorithmica **70**(2), 195–207 (2014)
124. Dahlhaus, E., Johnson, D.S., Papadimitriou, C.H., Seymour, P.D., Yannakakis, M.: The complexity of multiterminal cuts. SIAM J. Computing **23**(4), 864–894 (1994)
125. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5**, 394–397 (1962)
126. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**, 201–215 (1960)
127. Dehne, F.K.H.A., Fellows, M.R., Langston, M.A., Rosamond, F.A., Stevens, K.: An $O(2^{O(k)}n^3)$ FPT algorithm for the undirected feedback vertex set problem. In: Proceedings of the 11th Annual International Conference on Computing and Combinatorics (COCOON), *Lecture Notes in Comput. Sci.*, vol. 3595, pp. 859–869. Springer, Berlin (2005)
128. Dell, H.: AND-compression of NP-complete problems: Streamlined proof and minor observations. In: Proceedings of the 9th International Symposium on Parameterized and Exact Computation (IPEC), *Lecture Notes in Comput. Sci.*, vol. 8894, pp. 184–195. Springer (2014)
129. Dell, H., Marx, D.: Kernelization of packing problems. In: Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 68–81. SIAM (2012)
130. Dell, H., van Melkebeek, D.: Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. J. ACM **61**(4), 23 (2014)
131. Demaine, E.D., Fomin, F.V., Hajiaghayi, M., Thilikos, D.M.: Subexponential parameterized algorithms on graphs of bounded genus and H-minor-free graphs. J. ACM **52**(6), 866–893 (2005)
132. Demaine, E.D., Hajiaghayi, M.: Bidimensionality: new connections between FPT algorithms and PTASs. In: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 590–601. SIAM (2005)
133. Demaine, E.D., Hajiaghayi, M.: The bidimensionality theory and its algorithmic applications. Comput. J. **51**(3), 292–302 (2008)
134. Demaine, E.D., Hajiaghayi, M.: Linearity of grid minors in treewidth with applications through bidimensionality. Combinatorica **28**(1), 19–36 (2008)
135. Demaine, E.D., Hajiaghayi, M., Kawarabayashi, K.: Contraction decomposition in H-minor-free graphs and algorithmic applications. In: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC), pp. 441–450. ACM (2011)
136. DeMillo, R.A., Lipton, R.J.: A probabilistic remark on algebraic program testing. Information Processing Letters **7**, 193–195 (1978)
137. Dendris, N.D., Kirousis, L.M., Thilikos, D.M.: Fugitive-search games on graphs and related parameters. Theoretical Computer Science **172**(1-2), 233–254 (1997)
138. Diestel, R.: Graph theory, *Graduate Texts in Mathematics*, vol. 173, 3rd edn. Springer-Verlag, Berlin (2005)
139. Dirac, G.A.: On rigid circuit graphs. Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg **25**, 71–76 (1961)
140. Dom, M., Guo, J., Hüffner, F., Niedermeier, R., Truß, A.: Fixed-parameter tractability results for feedback set problems in tournaments. J. Discrete Algorithms **8**(1), 76–86 (2010)
141. Dom, M., Lokshtanov, D., Saurabh, S.: Incompressibility through colors and IDs. In: Proceedings of the 36th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 5555, pp. 378–389. Springer (2009)
142. Dorn, F.: Dynamic programming and fast matrix multiplication. In: Proceedings of the 14th Annual European Symposium on Algorithms (ESA), *Lecture Notes in Comput. Sci.*, vol. 4168, pp. 280–291. Springer, Berlin (2006)

143. Dorn, F.: Planar subgraph isomorphism revisited. In: Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science (STACS), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 5, pp. 263–274. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2010)
144. Dorn, F., Fomin, F.V., Lokshtanov, D., Raman, V., Saurabh, S.: Beyond bidimensionality: Parameterized subexponential algorithms on directed graphs. Information and Computation **233**, 60–70 (2013)
145. Dorn, F., Fomin, F.V., Thilikos, D.M.: Subexponential parameterized algorithms. Computer Science Review **2**(1), 29–39 (2008)
146. Dorn, F., Fomin, F.V., Thilikos, D.M.: Catalan structures and dynamic programming in H-minor-free graphs. J. Computer and System Sciences **78**(5), 1606–1622 (2012)
147. Dorn, F., Penninkx, E., Bodlaender, H.L., Fomin, F.V.: Efficient exact algorithms on planar graphs: Exploiting sphere cut decompositions. Algorithmica **58**(3), 790–810 (2010)
148. Downey, R.G., Fellows, M.R.: Fixed-parameter intractability. In: Proceedings of the 7th Annual Structure in Complexity Theory Conference, pp. 36–49. IEEE Computer Society (1992)
149. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness. Proceedings of the 21st Manitoba Conference on Numerical Mathematics and Computing. Congr. Numer. **87**, 161–178 (1992)
150. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness I: Basic results. SIAM J. Computing **24**(4), 873–921 (1995)
151. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness II: On completeness for W[1]. Theoretical Computer Science **141**(1&2), 109–131 (1995)
152. Downey, R.G., Fellows, M.R.: Parameterized computational feasibility. In: P. Clote, J. Remmel (eds.) Proceedings of the Second Cornell Workshop on Feasible Mathematics. Feasible Mathematics II, pp. 219–244. Birkhäuser, Boston (1995)
153. Downey, R.G., Fellows, M.R.: Parameterized complexity. Springer-Verlag, New York (1999)
154. Downey, R.G., Fellows, M.R.: Fundamentals of Parameterized Complexity. Texts in Computer Science. Springer (2013)
155. Downey, R.G., Fellows, M.R., Vardy, A., Whittle, G.: The parametrized complexity of some fundamental problems in coding theory. SIAM J. Computing **29**(2), 545–570 (1999)
156. Dragan, F.F., Fomin, F.V., Golovach, P.A.: Spanners in sparse graphs. J. Computer and System Sciences **77**(6), 1108–1119 (2011)
157. Drange, P.G., Dregi, M.S., van 't Hof, P.: On the computational complexity of vertex integrity and component order connectivity. In: Proceedings of the 25th International Symposium on Algorithms and Computation (ISAAC), *Lecture Notes in Comput. Sci.*, vol. 8889, pp. 285–297. Springer (2014)
158. Drange, P.G., Fomin, F.V., Pilipczuk, M., Villanger, Y.: Exploring subexponential parameterized complexity of completion problems. In: Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 25, pp. 288–299. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2014)
159. Dreyfus, S.E., Wagner, R.A.: The Steiner problem in graphs. Networks **1**(3), 195–207 (1971)
160. Drucker, A.: New limits to classical and quantum instance compression. In: Proceedings of the 53rd Annual Symposium on Foundations of Computer Science (FOCS), pp. 609–618. IEEE (2012)
161. Edmonds, J.: Minimum partition of a matroid into independent subsets. J. Res. Nat. Bur. Standards Sect. B **69**, 67–72 (1965)
162. Edmonds, J., Fulkerson, D.R.: Transversals and Matroid Partition. Defense Technical Information Center (1965)

163. Eisenbrand, F., Grandoni, F.: On the complexity of fixed parameter clique and dominating set. Theor. Comput. Sci. **326**(1-3), 57–67 (2004)
164. Ellis, J.A., Sudborough, I.H., Turner, J.S.: The vertex separation and search number of a graph. Information and Computation **113**(1), 50–79 (1994)
165. Eppstein, D.: Subgraph isomorphism in planar graphs and related problems. J. Graph Algorithms and Applications **3**, 1–27 (1999)
166. Erdős, P.: On a problem of graph theory. The Mathematical Gazette **47**(361), 220–223 (1963)
167. Erdős, P., Rado, R.: Intersection theorems for systems of sets. J. London Math. Soc. **35**, 85–90 (1960)
168. Erickson, R.E., Monma, C.L., Veinott, A.F.J.: Send-and-split method for minimum-concave-cost network flows. Mathematics of Operations Research **12**(4), 634–664 (1987)
169. Estivill-Castro, V., Fellows, M.R., Langston, M.A., Rosamond, F.A.: FPT is P-time extremal structure I. In: Proceedings of the First Workshop Algorithms and Complexity in Durham (ACID), vol. 4, pp. 1–41 (2005)
170. Fafianie, S., Bodlaender, H.L., Nederlof, J.: Speeding up dynamic programming with representative sets - an experimental evaluation of algorithms for Steiner tree on tree decompositions. In: Proceedings of the 8th International Symposium on Parameterized and Exact Computation (IPEC), *Lecture Notes in Comput. Sci.*, vol. 8246, pp. 321–334. Springer (2013)
171. Feige, U.: Faster FAST (Feedback Arc Set in Tournaments). CoRR **abs/0911.5094** (2009)
172. Feige, U., Hajiaghayi, M., Lee, J.R.: Improved approximation algorithms for minimum weight vertex separators. SIAM J. Computing **38**(2), 629–657 (2008)
173. Feldman, J., Ruhl, M.: The directed Steiner network problem is tractable for a constant number of terminals. SIAM J. Computing **36**(2), 543–561 (2006)
174. Fellows, M.R.: Blow-ups, win/win's, and crown rules: Some new directions in FPT. In: Proceedings of the 29th Workshop on Graph-Theoretic Concepts in Computer Science (WG), *Lecture Notes in Comput. Sci.*, vol. 2880. Springer (2003)
175. Fellows, M.R.: The lost continent of polynomial time: Preprocessing and kernelization. In: Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC), *Lecture Notes in Computer Science*, vol. 4169, pp. 276–277. Springer (2006)
176. Fellows, M.R., Fomin, F.V., Lokshtanov, D., Losievskaja, E., Rosamond, F.A., Saurabh, S.: Distortion is fixed parameter tractable. ACM Transactions on Computation Theory **5**(4), 16 (2013)
177. Fellows, M.R., Fomin, F.V., Lokshtanov, D., Rosamond, F.A., Saurabh, S., Szeider, S., Thomassen, C.: On the complexity of some colorful problems parameterized by treewidth. Inf. Comput. **209**(2), 143–153 (2011)
178. Fellows, M.R., Heggernes, P., Rosamond, F.A., Sloper, C., Telle, J.A.: Finding k disjoint triangles in an arbitrary graph. In: Proceedings of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG), *Lecture Notes in Comput. Sci.*, vol. 3353, pp. 235–244. Springer (2004)
179. Fellows, M.R., Hermelin, D., Rosamond, F.A., Vialette, S.: On the parameterized complexity of multiple-interval graph problems. Theoretical Computer Science **410**(1), 53–61 (2009)
180. Fellows, M.R., Langston, M.A.: Nonconstructive advances in polynomial-time complexity. Information Processing Letters **26**(3), 155–162 (1987)
181. Fellows, M.R., Langston, M.A.: Fast self-reduction algorithms for combinatorical problems of VLSI-design. In: 3rd Aegean Workshop on Computing VLSI Algorithms and Architectures (AWOC), *Lecture Notes in Comput. Sci.*, vol. 319, pp. 278–287. Springer (1988)
182. Fellows, M.R., Langston, M.A.: Nonconstructive tools for proving polynomial-time decidability. J. ACM **35**(3), 727–739 (1988)

183. Fellows, M.R., Langston, M.A.: An analogue of the Myhill-Nerode theorem and its use in computing finite-basis characterizations (extended abstract). In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS), pp. 520–525. IEEE (1989)
184. Fellows, M.R., Langston, M.A.: On search, decision and the efficiency of polynomial-time algorithms (extended abstract). In: Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC), pp. 501–512. ACM (1989)
185. Fellows, M.R., Lokshtanov, D., Misra, N., Rosamond, F.A., Saurabh, S.: Graph layout problems parameterized by vertex cover. In: Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC), *Lecture Notes in Comput. Sci.*, vol. 5369, pp. 294–305. Springer (2008)
186. Fernau, H., Fomin, F.V., Lokshtanov, D., Mnich, M., Philip, G., Saurabh, S.: Social choice meets graph drawing: how to get subexponential time algorithms for ranking and drawing problems. Tsinghua Sci. Technol. **19**(4), 374–386 (2014)
187. Fernau, H., Manlove, D.: Vertex and edge covers with clustering properties: Complexity and algorithms. J. Discrete Algorithms **7**(2), 149–167 (2009)
188. Flum, J., Grohe, M.: The parameterized complexity of counting problems. SIAM J. Computing **33**(4), 892–922 (2004)
189. Flum, J., Grohe, M.: Parameterized Complexity Theory. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin (2006)
190. Fomin, F.V., Gaspers, S., Saurabh, S., Thomassé, S.: A linear vertex kernel for maximum internal spanning tree. J. Computer and System Sciences **79**(1), 1–6 (2013)
191. Fomin, F.V., Golovach, P.A., Lokshtanov, D., Saurabh, S.: Intractability of clique-width parameterizations. SIAM J. Computing **39**(5), 1941–1956 (2010)
192. Fomin, F.V., Golovach, P.A., Lokshtanov, D., Saurabh, S.: Almost optimal lower bounds for problems parameterized by clique-width. SIAM J. Computing **43**(5), 1541–1563 (2014)
193. Fomin, F.V., Golovach, P.A., Thilikos, D.M.: Contraction obstructions for treewidth. J. Combinatorial Theory Ser. B **101**(5), 302–314 (2011)
194. Fomin, F.V., Grandoni, F., Kratsch, D.: A Measure & Conquer approach for the analysis of exact algorithms. J. ACM **56**(5) (2009)
195. Fomin, F.V., Høie, K.: Pathwidth of cubic graphs and exact algorithms. Information Processing Letters **97**(5), 191–196 (2006)
196. Fomin, F.V., Kaski, P.: Exact exponential algorithms. Communications of the ACM **56**(3), 80–88 (2013)
197. Fomin, F.V., Kratsch, D.: Exact Exponential Algorithms. Springer (2010). An EATCS Series: Texts in Theoretical Computer Science
198. Fomin, F.V., Kratsch, D., Woeginger, G.J.: Exact (exponential) algorithms for the dominating set problem. In: Proceedings of the 30th Workshop on Graph Theoretic Concepts in Computer Science (WG), *Lecture Notes in Comput. Sci.*, vol. 3353, pp. 245–256. Springer, Berlin (2004)
199. Fomin, F.V., Kratsch, S., Pilipczuk, M., Pilipczuk, M., Villanger, Y.: Tight bounds for parameterized complexity of cluster editing with a small number of clusters. J. Computer and System Sciences **80**(7), 1430–1447 (2014)
200. Fomin, F.V., Lokshtanov, D., Misra, N., Saurabh, S.: Planar F-deletion: Approximation, kernelization and optimal FPT algorithms. In: Proceedings of the 53rd Annual Symposium on Foundations of Computer Science (FOCS), pp. 470–479. IEEE (2012)
201. Fomin, F.V., Lokshtanov, D., Panolan, F., Saurabh, S.: Representative sets of product families. In: Proceedings of the 22nd Annual European Symposium on Algorithms (ESA), *Lecture Notes in Comput. Sci.*, vol. 8737, pp. 443–454. Springer (2014)
202. Fomin, F.V., Lokshtanov, D., Raman, V., Saurabh, S.: Bidimensionality and EPTAS. In: Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 748–759. SIAM (2011)
203. Fomin, F.V., Lokshtanov, D., Raman, V., Saurabh, S.: Subexponential algorithms for partial cover problems. Information Processing Letters **111**(16), 814–818 (2011)

204. Fomin, F.V., Lokshtanov, D., Raman, V., Saurabh, S., Rao, B.V.R.: Faster algorithms for finding and counting subgraphs. J. Computer and System Sciences **78**(3), 698–706 (2012)
205. Fomin, F.V., Lokshtanov, D., Saurabh, S.: Bidimensionality and geometric graphs. In: Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1563–1575. SIAM (2012)
206. Fomin, F.V., Lokshtanov, D., Saurabh, S.: Efficient computation of representative sets with applications in parameterized and exact algorithms. In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 142–151. SIAM (2014)
207. Fomin, F.V., Lokshtanov, D., Saurabh, S., Thilikos, D.M.: Bidimensionality and kernels. In: Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 503–510. SIAM (2010)
208. Fomin, F.V., Lokshtanov, D., Saurabh, S., Thilikos, D.M.: Linear kernels for (connected) dominating set on H-minor-free graphs. In: Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 82–93. SIAM (2012)
209. Fomin, F.V., Pilipczuk, M.: Subexponential parameterized algorithm for computing the cutwidth of a semi-complete digraph. In: Proceedings of the 21st Annual European Symposium on Algorithms (ESA), LNCS, *Lecture Notes in Comput. Sci.*, vol. 8125, pp. 505–516. Springer (2013)
210. Fomin, F.V., Thilikos, D.M.: An annotated bibliography on guaranteed graph searching. Theoretical Computer Science **399**(3), 236–245 (2008)
211. Fomin, F.V., Villanger, Y.: Subexponential parameterized algorithm for minimum fill-in. SIAM J. Computing **42**(6), 2197–2216 (2013)
212. Ford Jr., L.R., Fulkerson, D.R.: Maximal flow through a network. Canad. J. Math. **8**, 399–404 (1956)
213. Fortnow, L., Santhanam, R.: Infeasibility of instance compression and succinct PCPs for NP. J. Computer and System Sciences **77**(1), 91–106 (2011)
214. Frank, A.: Connections in Combinatorial Optimization. Oxford Lecture Series in Mathematics and Its Applications. Oxford University Press (2011)
215. Frank, A., Tardos, É.: An application of simultaneous Diophantine approximation in combinatorial optimization. Combinatorica **7**(1), 49–65 (1987)
216. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with 0(1) worst case access time. J. ACM **31**(3), 538–544 (1984)
217. Freedman, M., Lovász, L., Schrijver, A.: Reflection positivity, rank connectivity, and homomorphism of graphs. J. Amer. Math. Soc. **20**(1), 37–51 (2007)
218. Frick, M., Grohe, M.: Deciding first-order properties of locally tree-decomposable structures. J. ACM **48**(6), 1184–1206 (2001)
219. Fuchs, B., Kern, W., Mölle, D., Richter, S., Rossmanith, P., Wang, X.: Dynamic programming for minimum Steiner trees. Theory of Computing Systems **41**(3), 493–500 (2007)
220. Gabow, H.N., Nie, S.: Finding a long directed cycle. ACM Transactions on Algorithms **4**(1), 7 (2008)
221. Gabow, H.N., Stallmann, M.: An augmenting path algorithm for linear matroid parity. Combinatorica **6**(2), 123–150 (1986)
222. Gagarin, A., Myrvold, W.J., Chambers, J.: The obstructions for toroidal graphs with no $K_{3,3}$'s. Discrete Mathematics **309**(11), 3625–3631 (2009)
223. Gallai, T.: Maximum-minimum Sätze und verallgemeinerte Faktoren von Graphen. Acta Math. Acad. Sci. Hungar. **12**, 131–173 (1961)
224. Garey, M.R., Johnson, D.S.: Computers and Intractability, A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York (1979)
225. Garg, N., Vazirani, V.V., Yannakakis, M.: Primal-dual approximation algorithms for integral flow and multicut in trees. Algorithmica **18**(1), 3–20 (1997)
226. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra. Cambridge University Press, New York, NY, USA (1999)

227. Geelen, J., Gerards, B., Whittle, G.: Excluding a planar graph from $GF(q)$-representable matroids. J. Combinatorial Theory Ser. B **97**(6), 971–998 (2007)
228. Ghosh, E., Kolay, S., Kumar, M., Misra, P., Panolan, F., Rai, A., Ramanujan, M.S.: Faster parameterized algorithms for deletion to split graphs. In: Proceedings of the 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT), *Lecture Notes in Comput. Sci.*, vol. 7357, pp. 107–118. Springer (2012)
229. Godlin, B., Kotek, T., Makowsky, J.A.: Evaluations of graph polynomials. In: Proceedings of the 34th International Workshop on Graph-Theoretic Concepts in Computer Science (WG), *Lecture Notes in Comput. Sci.*, vol. 5344, pp. 183–194 (2008)
230. Golovach, P.A., van 't Hof, P., Paulusma, D.: Obtaining planarity by contracting few edges. Theoretical Computer Science **476**, 38–46 (2013)
231. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. Academic Press, New York (1980)
232. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete mathematics: A foundation for computer science, 2nd edn. Addison-Wesley Publishing Company, Reading, MA (1994)
233. Graham, R.L., Spencer, J.H.: A constructive solution to a tournament problem. Canadian Mathematical Bulletin **14**, 45–47 (1971)
234. Gramm, J., Guo, J., Hüffner, F., Niedermeier, R.: Data reduction and exact algorithms for clique cover. ACM Journal of Experimental Algorithmics **13** (2008)
235. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for CLOSEST STRING and related problems. Algorithmica **37**(1), 25–42 (2003)
236. Grohe, M.: Local tree-width, excluded minors, and approximation algorithms. Combinatorica **23**(4), 613–632 (2003)
237. Grohe, M.: Logic, graphs, and algorithms. In: J. Flum, E. Grädel, T. Wilke (Eds), Logic and Automata – History and Perspectives, pp. 357 – 422. Amsterdam University Press, Amsterdam (2007)
238. Grohe, M., Kawarabayashi, K., Marx, D., Wollan, P.: Finding topological subgraphs is fixed-parameter tractable. In: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC), pp. 479–488. ACM (2011)
239. Gu, Q.P., Tamaki, H.: Improved bounds on the planar branchwidth with respect to the largest grid minor size. Algorithmica **64**(3), 416–453 (2012)
240. Guo, J.: Problem kernels for NP-complete edge deletion problems: Split and related graphs. In: Proceedings of the 18th International Symposium on Algorithms and Computation (ISAAC), *Lecture Notes in Comput. Sci.*, vol. 4835, pp. 915–926. Springer (2007)
241. Guo, J., Gramm, J., Hüffner, F., Niedermeier, R., Wernicke, S.: Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. J. Computer and System Sciences **72**(8), 1386–1396 (2006)
242. Guo, J., Niedermeier, R.: Fixed-parameter tractability and data reduction for multicut in trees. Networks **46**(3), 124–135 (2005)
243. Guo, J., Niedermeier, R.: Linear problem kernels for NP-hard problems on planar graphs. In: Proceedings of the 34th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 4596, pp. 375–386. Springer (2007)
244. Guo, J., Niedermeier, R., Suchý, O.: Parameterized complexity of arc-weighted directed Steiner problems. SIAM J. Discrete Math. **25**(2), 583–599 (2011)
245. Guo, J., Niedermeier, R., Wernicke, S.: Parameterized complexity of generalized vertex cover problems. In: Proceedings of the 9th International Workshop on Algorithms and Data Structures (WADS), *Lecture Notes in Comput. Sci.*, vol. 3608, pp. 36–48. Springer, Berlin (2005)
246. Gupta, A., Newman, I., Rabinovich, Y., Sinclair, A.: Cuts, trees and ℓ_1-embeddings of graphs. Combinatorica **24**(2), 233–269 (2004)

247. Gutin, G., van Iersel, L., Mnich, M., Yeo, A.: All ternary permutation constraint satisfaction problems parameterized above average have kernels with quadratic numbers of variables. In: Proceedings of the 18th Annual European Symposium on Algorithms (ESA), *Lecture Notes in Comput. Sci.*, vol. 6346, pp. 326–337. Springer (2010)
248. Gutin, G., van Iersel, L., Mnich, M., Yeo, A.: Every ternary permutation constraint satisfaction problem parameterized above average has a kernel with a quadratic number of variables. J. Computer and System Sciences **78**(1), 151–163 (2012)
249. Gutin, G., Kim, E.J., Lampis, M., Mitsou, V.: Vertex cover problem parameterized above and below tight bounds. Theory of Computing Systems **48**(2), 402–410 (2011)
250. Gutin, G., Kim, E.J., Szeider, S., Yeo, A.: A probabilistic approach to problems parameterized above or below tight bounds. J. Computer and System Sciences **77**(2), 422–429 (2011)
251. Gutin, G., Yeo, A.: Note on maximal bisection above tight lower bound. Information Processing Letters **110**(21), 966–969 (2010)
252. Gutin, G., Yeo, A.: Constraint satisfaction problems parameterized above or below tight bounds: A survey. In: H.L. Bodlaender, R. Downey, F.V. Fomin, D. Marx (eds.) The Multivariate Algorithmic Revolution and Beyond, *Lecture Notes in Computer Science*, vol. 7370, pp. 257–286. Springer (2012)
253. Gyárfás, A.: A simple lower bound on edge coverings by cliques. Discrete Mathematics **85**(1), 103–104 (1990)
254. Halin, R.: S-functions for graphs. J. Geometry **8**(1-2), 171–186 (1976)
255. Hall, P.: A contribution to the theory of groups of prime-power order. Proc. London Math. Soc. **36**, 24–80 (1934)
256. Hall, P.: On representatives of subsets. J. London Math. Soc. **10**, 26–30 (1935)
257. Halmos, P.R., Vaughan, H.E.: The marriage problem. American Journal of Mathematics **72**(1), 214–215 (1950)
258. Hermelin, D., Kratsch, S., Sołtys, K., Wahlström, M., Wu, X.: A completeness theory for polynomial (Turing) kernelization. In: Proceedings of the 8th International Symposium on Parameterized and Exact Computation (IPEC), *Lecture Notes in Comput. Sci.*, vol. 8246, pp. 202–215. Springer (2013)
259. Hermelin, D., Mnich, M., van Leeuwen, E.J., Woeginger, G.J.: Domination when the stars are out. In: Proceedings of the 38th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 6755, pp. 462–473. Springer (2011)
260. Hermelin, D., Wu, X.: Weak compositions and their applications to polynomial lower bounds for kernelization. In: Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 104–113. SIAM (2012)
261. Hertli, T.: 3-SAT faster and simpler—Unique-SAT bounds for PPSZ hold in general. SIAM J. Computing **43**(2), 718–729 (2014)
262. Hliněný, P.: Branch-width, parse trees, and monadic second-order logic for matroids. J. Combinatorial Theory Ser. B **96**(3), 325–351 (2006)
263. Hlinený, P., Oum, S.: Finding branch-decompositions and rank-decompositions. SIAM J. Computing **38**(3), 1012–1032 (2008)
264. Hlinený, P., Oum, S., Seese, D., Gottlob, G.: Width parameters beyond tree-width and their applications. Comput. J. **51**(3), 326–362 (2008)
265. Hochbaum, D.S., Maass, W.: Approximation schemes for covering and packing problems in image processing and VLSI. J. ACM **32**(1), 130–136 (1985)
266. Hodges, W.: A Shorter Model Theory. Cambridge University Press (1997)
267. van der Holst, H.: A polynomial-time algorithm to find a linkless embedding of a graph. J. Combinatorial Theory Ser. B **99**(2), 512–530 (2009)
268. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J. Computing **2**, 225–231 (1973)
269. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley Longman Publishing, Boston (2006)

270. Horn, R.A., Johnson, C.R.: Topics in matrix analysis. Cambridge University Press (1991)
271. Hüffner, F., Komusiewicz, C., Moser, H., Niedermeier, R.: Fixed-parameter algorithms for Cluster Vertex Deletion. Theory of Computing Systems **47**(1), 196–217 (2010)
272. Impagliazzo, R., Paturi, R.: On the complexity of k-SAT. J. Computer and System Sciences **62**(2), 367–375 (2001)
273. Impagliazzo, R., Paturi, R., Zane, F.: Which problems have strongly exponential complexity. J. Computer and System Sciences **63**(4), 512–530 (2001)
274. Jansen, B.M.P.: Kernelization for maximum leaf spanning tree with positive vertex weights (2009). Technical Report UU-CS-2009-027, available at `http://www.cs.uu.nl/research/techreps/repo/CS-2009/2009-027.pdf`.
275. Jansen, B.M.P.: Turing kernelization for finding long paths and cycles in restricted graph classes. In: Proceedings of the 22nd Annual European Symposium on Algorithms (ESA), *Lecture Notes in Comput. Sci.*, vol. 8737, pp. 579–591. Springer (2014)
276. Jansen, B.M.P., Kratsch, S.: Data reduction for graph coloring problems. Inf. Comput. **231**, 70–88 (2013)
277. Jansen, B.M.P., Lokshtanov, D., Saurabh, S.: A near-optimal planarization algorithm. In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1802–1811. SIAM (2014)
278. Jensen, P.M., Korte, B.: Complexity of matroid property algorithms. SIAM Journal on Computing **11**(1), 184–190 (1982)
279. Kabanets, V., Impagliazzo, R.: Derandomizing polynomial identity tests means proving circuit lower bounds. Computational Complexity **13**(1-2), 1–46 (2004)
280. Kannan, R.: Minkowski's convex body theorem and integer programming. Mathematics of Operations Research **12**(3), 415–440 (1987)
281. Kaplan, H., Shamir, R., Tarjan, R.E.: Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. SIAM J. Computing **28**(5), 1906–1922 (1999)
282. Karp, R.M.: Dynamic programming meets the principle of inclusion and exclusion. Oper. Res. Lett. **1**(2), 49–51 (1982)
283. Karpinski, M., Schudy, W.: Faster algorithms for Feedback Arc Set Tournament, Kemeny Rank Aggregation and Betweenness Tournament. In: Proceedings of the 21st International Symposium on Algorithms and Computation (ISAAC), *Lecture Notes in Comput. Sci.*, vol. 6506, pp. 3–14. Springer (2010)
284. Kaski, P.: Linear and bilinear transformations for moderately exponential algorithms (2009). Manuscript, lecture notes for AGAPE'09 summer school
285. Kawarabayashi, K.: Planarity allowing few error vertices in linear time. In: Proceedings of the 50th Annual Symposium on Foundations of Computer Science (FOCS), pp. 639–648. IEEE (2009)
286. Kawarabayashi, K., Kobayashi, Y.: Linear min-max relation between the treewidth of H-minor-free graphs and its largest grid. In: Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science (STACS), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 14, pp. 278–289. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2012)
287. Kawarabayashi, K., Kreutzer, S., Mohar, B.: Linkless and flat embeddings in 3-space. Discrete & Computational Geometry **47**(4), 731–755 (2012)
288. Khachiyan, L., Porkolab, L.: Integer optimization on convex semialgebraic sets. Discrete & Computational Geometry **23**(2), 207–224 (2000)
289. Khuller, S.: Algorithms column: the vertex cover problem. SIGACT News **33**(2), 31–33 (2002)
290. Kim, E.J., Williams, R.: Improved parameterized algorithms for above average constraint satisfaction. In: Proceedings of the 6th International Symposium on Parameterized and Exact Computation (IPEC), *Lecture Notes in Comput. Sci.*, vol. 7112, pp. 118–131. Springer (2012)

291. Kinnersley, N.G.: The vertex separation number of a graph equals its path-width. Information Processing Letters **42**(6), 345–350 (1992)
292. Kirousis, L.M., Papadimitriou, C.H.: Interval graphs and searching. Discrete Math. **55**(2), 181–184 (1985)
293. Kirousis, L.M., Papadimitriou, C.H.: Searching and pebbling. Theoretical Computer Science **47**(2), 205–218 (1986)
294. Klein, P.N.: A linear-time approximation scheme for TSP in undirected planar graphs with edge-weights. SIAM J. Computing **37**(6), 1926–1952 (2008)
295. Klein, P.N., Marx, D.: Solving planar k-terminal cut in $O(n^{c\sqrt{k}})$ time. In: Proceedings of the 39th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 7391, pp. 569–580. Springer (2012)
296. Kleinberg, J., Tardos, É.: Algorithm Design. Addison-Wesley (2005)
297. Kloks, T.: Treewidth, Computations and Approximations, *Lecture Notes in Comput. Sci.*, vol. 842. Springer (1994)
298. Kneis, J., Langer, A., Rossmanith, P.: A new algorithm for finding trees with many leaves. Algorithmica **61**(4), 882–897 (2011)
299. Kneis, J., Mölle, D., Richter, S., Rossmanith, P.: Divide-and-color. In: Proceedings of the 34th International Workshop Graph-Theoretic Concepts in Computer Science (WG), *Lecture Notes in Comput. Sci.*, vol. 4271, pp. 58–67. Springer (2008)
300. Knuth, D.E.: The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd edn. Addison-Wesley (1997)
301. Kociumaka, T., Pilipczuk, M.: Faster deterministic feedback vertex set. Inf. Process. Lett. **114**(10), 556–560 (2014)
302. Kohn, S., Gottlieb, A., Kohn, M.: A generating function approach to the traveling salesman problem. In: Proceedings of ACM annual conference (ACM 1977), pp. 294–300. ACM Press (1977)
303. Kőnig, D.: Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. Math. Ann. **77**(4), 453–465 (1916)
304. Kotek, T., Makowsky, J.A.: Connection matrices and the definability of graph parameters. Logical Methods in Computer Science **10**(4) (2014)
305. Koutis, I.: Faster algebraic algorithms for path and packing problems. In: Proceedings of the 35th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 5125, pp. 575–586 (2008)
306. Koutis, I., Williams, R.: Limits and applications of group algebras for parameterized problems. In: Proceedings of the 36th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 5555, pp. 653–664. Springer (2009)
307. Kowalik, L., Pilipczuk, M., Suchan, K.: Towards optimal kernel for connected vertex cover in planar graphs. Discrete Applied Mathematics **161**(7-8), 1154–1161 (2013)
308. Kratsch, S., Philip, G., Ray, S.: Point line cover: The easy kernel is essentially tight. In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1596–1606. SIAM (2014)
309. Kratsch, S., Pilipczuk, M., Pilipczuk, M., Wahlström, M.: Fixed-parameter tractability of multicut in directed acyclic graphs. In: Proceedings of the 39th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 7391, pp. 581–593. Springer (2012)
310. Kratsch, S., Wahlström, M.: Representative sets and irrelevant vertices: New tools for kernelization. In: Proceedings of the 53rd Annual Symposium on Foundations of Computer Science (FOCS), pp. 450–459. IEEE (2012)
311. Kratsch, S., Wahlström, M.: Matroid theory and kernelization (presentation slides). http://worker2013.mimuw.edu.pl/slides/magnus.pdf (2013)
312. Kratsch, S., Wahlström, M.: Compression via matroids: A randomized polynomial kernel for odd cycle transversal. ACM Transactions on Algorithms **10**(4), 20 (2014)

313. Kreutzer, S.: Algorithmic meta-theorems. In: Finite and algorithmic model theory, *London Math. Soc. Lecture Note Ser.*, vol. 379, pp. 177–270. Cambridge Univ. Press, Cambridge (2011)
314. Kuratowski, K.: Sur le problème des courbes gauches en topologie. Fund. Math. **15**, 271–283 (1930)
315. LaPaugh, A.S.: Recontamination does not help to search a graph. J. ACM **40**(2), 224–245 (1993)
316. Lay, D.C.: Linear Algebra and Its Applications. Pearson Education (2002)
317. Leaf, A., Seymour, P.D.: Treewidth and planar minors. https://web.math.princeton.edu/ pds/papers/treewidth/paper.pdf (2012)
318. Lengauer, T.: Black-white pebbles and graph separation. Acta Inf. **16**, 465–475 (1981)
319. Lenstra Jr, H.W.: Integer programming with a fixed number of variables. Mathematics of operations research **8**(4), 538–548 (1983)
320. Lichtenstein, D.: Planar formulae and their uses. SIAM J. Computing **11**(2), 329–343 (1982)
321. Lidl, R., Niederreiter, H.: Introduction to Finite Fields and Their Applications. Cambridge University Press (1994)
322. Lin, B.: The parameterized complexity of k-Biclique. In: Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 605–615. SIAM (2015)
323. Lokshtanov, D.: New methods in parameterized algorithms and complexity. Ph.D. thesis, University of Bergen (2009)
324. Lokshtanov, D., Marx, D.: Clustering with local restrictions. Information and Computation **222**, 278–292 (2013)
325. Lokshtanov, D., Marx, D., Saurabh, S.: Known algorithms on graphs on bounded treewidth are probably optimal. In: Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 777–789. SIAM (2011)
326. Lokshtanov, D., Marx, D., Saurabh, S.: Slightly superexponential parameterized problems. In: Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 760–776. SIAM (2011)
327. Lokshtanov, D., Misra, N., Saurabh, S.: Imbalance is fixed parameter tractable. Information Processing Letters **113**(19-21), 714–718 (2013)
328. Lokshtanov, D., Narayanaswamy, N.S., Raman, V., Ramanujan, M.S., Saurabh, S.: Faster parameterized algorithms using linear programming. ACM Transactions on Algorithms **11**(2), 15 (2014)
329. Lokshtanov, D., Nederlof, J.: Saving space by algebraization. In: Proceedings of the 42nd Annual ACM Symposium on Theory of Computing (STOC), pp. 321–330. ACM (2010)
330. Lokshtanov, D., Ramanujan, M.S.: Parameterized tractability of multiway cut with parity constraints. In: Proceedings of the 39th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 7391, pp. 750–761. Springer (2012)
331. Lovász, L.: Flats in matroids and geometric graphs. In: Combinatorial surveys (Proc. Sixth British Combinatorial Conf., Royal Holloway Coll., Egham), pp. 45–86. Academic Press, London (1977)
332. Lovász, L.: The matroid matching problem. Algebraic methods in graph theory **1**, 495–517 (1978)
333. Lovász, L.: On determinants, matchings, and random algorithms. In: Proceedings of the International Symposium on Fundamentals of Computation Theory (FCT), pp. 565–574 (1979)
334. Lovász, L.: Matroid matching and some applications. Journal of Combinatorial Theory, Series B **28**(2), 208–236 (1980)
335. Lovász, L.: Communication complexity: a survey. In: Paths, flows, and VLSI-layout (Bonn, 1988), *Algorithms Combin.*, vol. 9, pp. 235–265. Springer, Berlin (1990)

336. Lovász, L.: Connection matrices. In: G. Grimmet, C. McDiarmid (eds.) Combinatorics, Complexity and Chance, A Tribute to Dominic Welsh, *Oxford Lecture Series in Mathematics and Its Applications*, vol. 34, pp. 179–190. New York; Oxford University Press (2007)
337. Lovász, L.: Large networks and graph limits, vol. 60. American Mathematical Soc. (2012)
338. Lovász, L., Plummer, M.D.: Matching theory. AMS Chelsea Publishing, Providence (2009)
339. Lovász, L., Saks, M.E.: Communication complexity and combinatorial lattice theory. J. Computer and System Sciences **47**(2), 322–349 (1993)
340. Luks, E.M.: Isomorphism of graphs of bounded valence can be tested in polynomial time. J. Computer and System Sciences **25**(1), 42–65 (1982)
341. Ma, B., Sun, X.: More efficient algorithms for closest string and substring problems. SIAM J. Computing **39**(4), 1432–1443 (2009)
342. Mahajan, M., Raman, V.: Parameterizing above guaranteed values: MaxSat and MaxCut. J. Algorithms **31**(2), 335–354 (1999)
343. Mahajan, M., Raman, V., Sikdar, S.: Parameterizing above or below guaranteed values. J. Computer and System Sciences **75**(2), 137–153 (2009)
344. Makowsky, J.A.: Connection matrices for MSOL-definable structural invariants. In: Third Indian Conference on Logic and Its Applications (ICLA), *Lecture Notes in Comput. Sci.*, vol. 5378, pp. 51–64. Springer (2009)
345. Marx, D.: Efficient approximation schemes for geometric problems? In: Proceedings of the 13th Annual European Symposium on Algorithms (ESA), *Lecture Notes in Comput. Sci.*, vol. 3669, pp. 448–459 (2005)
346. Marx, D.: Parameterized coloring problems on chordal graphs. Theoretical Computer Science **351**(3), 407–424 (2006)
347. Marx, D.: Parameterized graph separation problems. Theoretical Computer Science **351**(3), 394–406 (2006)
348. Marx, D.: On the optimality of planar and geometric approximation schemes. In: Proceedings of the 48th Annual Symposium on Foundations of Computer Science (FOCS), pp. 338–348. IEEE (2007)
349. Marx, D.: Closest substring problems with small distances. SIAM J. Computing **38**(4), 1382–1410 (2008)
350. Marx, D.: Parameterized complexity and approximation algorithms. Comput. J. **51**(1), 60–78 (2008)
351. Marx, D.: A parameterized view on matroid optimization problems. Theoretical Computer Science **410**(44), 4471–4479 (2009)
352. Marx, D.: Can you beat treewidth? Theory of Computing **6**(1), 85–112 (2010)
353. Marx, D.: Chordal deletion is fixed-parameter tractable. Algorithmica **57**(4), 747–768 (2010)
354. Marx, D.: A tight lower bound for planar multiway cut with fixed number of terminals. In: Proceedings of the 39th International Colloquium of Automata, Languages and Programming (ICALP), *Lecture Notes in Comput. Sci.*, vol. 7391, pp. 677–688. Springer (2012)
355. Marx, D., Pilipczuk, M.: (2014). Manuscript
356. Marx, D., Pilipczuk, M.: Everything you always wanted to know about the parameterized complexity of Subgraph Isomorphism (but were afraid to ask). In: Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 25, pp. 542–553. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2014)
357. Marx, D., Razgon, I.: Fixed-parameter tractability of multicut parameterized by the size of the cutset. SIAM J. Comput. **43**(2), 355–388 (2014)
358. Marx, D., Schlotter, I.: Obtaining a planar graph by vertex deletion. Algorithmica **62**(3-4), 807–822 (2012)

359. Marx, D., Sidiropoulos, A.: The limited blessing of low dimensionality: when $1 - 1/d$ is the best possible exponent for d-dimensional geometric problems. In: Proceedings of the 30th Annual Symposium on Computational Geometry (SOCG), pp. 67–76. ACM (2014)
360. Mathieson, L.: The parameterized complexity of degree constrained editing problems. Ph.D. thesis, Durham University (2009)
361. Mazoit, F.: Tree-width of hypergraphs and surface duality. J. Combinatorial Theory Ser. B **102**(3), 671–687 (2012)
362. Megiddo, N., Hakimi, S.L., Garey, M.R., Johnson, D.S., Papadimitriou, C.H.: The complexity of searching a graph. J. ACM **35**(1), 18–44 (1988)
363. Mehlhorn, K.: Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness, *EATCS Monographs on Theoretical Computer Science*, vol. 2. Springer (1984)
364. Mnich, M., Zenklusen, R.: Bisections above tight lower bounds. In: Proceedings of 38th International Workshop on Graph-Theoretic Concepts in Computer Science (WG), *Lecture Notes in Computer Science*, vol. 7551, pp. 184–193. Springer (2012)
365. Mohar, B.: A linear time algorithm for embedding graphs in an arbitrary surface. SIAM J. Discrete Math. **12**(1), 6–26 (1999)
366. Mohar, B., Thomassen, C.: Graphs on surfaces. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press (2001)
367. Mölle, D., Richter, S., Rossmanith, P.: Enumerate and expand: Improved algorithms for connected vertex cover and tree cover. Theory of Computing Systems **43**(2), 234–253 (2008)
368. Monien, B.: How to find long paths efficiently. In: Analysis and design of algorithms for combinatorial problems, *North-Holland Math. Stud.*, vol. 109, pp. 239–254. North-Holland, Amsterdam (1985)
369. Moser, H.: A problem kernelization for graph packing. In: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), *Lecture Notes in Comput. Sci.*, vol. 5404, pp. 401–412. Springer (2009)
370. Mulmuley, K., Vazirani, U.V., Vazirani, V.V.: Matching is as easy as matrix inversion. Combinatorica **7**(1), 105–113 (1987)
371. Murota, K.: Matrices and matroids for systems analysis, *Algorithms and Combinatorics*, vol. 20. Springer-Verlag, Berlin (2000)
372. Naor, M., Schulman, L.J., Srinivasan, A.: Splitters and near-optimal derandomization. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS), pp. 182–191. IEEE (1995)
373. Narayanaswamy, N.S., Raman, V., Ramanujan, M.S., Saurabh, S.: LP can be a cure for parameterized problems. In: Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science (STACS), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 14, pp. 338–349. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2012)
374. Nederlof, J.: Fast polynomial-space algorithms using inclusion-exclusion. Algorithmica **65**(4), 868–884 (2013)
375. Nemhauser, G.L., Trotter Jr., L.E.: Properties of vertex packing and independence system polyhedra. Math. Programming **6**, 48–61 (1974)
376. Niedermeier, R.: Invitation to fixed-parameter algorithms, *Oxford Lecture Series in Mathematics and its Applications*, vol. 31. Oxford University Press (2006)
377. Oum, S.: Approximating rank-width and clique-width quickly. ACM Transactions on Algorithms **5**(1) (2008)
378. Oum, S., Seymour, P.D.: Approximating clique-width and branch-width. J. Combinatorial Theory Ser. B **96**(4), 514–528 (2006)
379. Oxley, J.G.: Matroid theory, *Oxford Graduate Texts in Mathematics*, vol. 21, 2nd edn. Oxford University Press (2010)
380. Papadimitriou, C.H.: Computational complexity. Addison-Wesley Publishing, Reading (1994)

381. Papadimitriou, C.H., Yannakakis, M.: On limited nondeterminism and the complexity of the V-C dimension. J. Computer and System Sciences **53**(2), 161–170 (1996)
382. Paturi, R., Pudlák, P., Saks, M.E., Zane, F.: An improved exponential-time algorithm for k-SAT. J. ACM **52**(3), 337–364 (2005)
383. Philip, G., Raman, V., Sikdar, S.: Polynomial kernels for dominating set in graphs of bounded degeneracy and beyond. ACM Transactions on Algorithms **9**(1), 11 (2012)
384. Pietrzak, K.: On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. J. Computer and System Sciences **67**(4), 757–771 (2003)
385. Pilipczuk, M.: Tournaments and optimality: New results in parameterized complexity. Ph.D. thesis, University of Bergen, Norway (2013)
386. Plehn, J., Voigt, B.: Finding minimally weighted subgraphs. In: Proceedings of the 16th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1990), *Lecture Notes in Comput. Sci.*, vol. 484, pp. 18–29. Springer, Berlin (1990)
387. Prieto, E.: Systematic kernelization in FPT algorithm design. Ph.D. thesis, The University of Newcastle, Australia (2005)
388. Prieto, E., Sloper, C.: Reducing to independent set structure: the case of k-internal spanning tree. Nordic J. of Computing **12**, 308–318 (2005)
389. Prieto, E., Sloper, C.: Looking at the stars. Theoretical Computer Science **351**(3), 437–445 (2006)
390. Pătraşcu, M., Williams, R.: On the possibility of faster SAT algorithms. In: Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1065–1075. SIAM (2010)
391. Quine, W.V.: The problem of simplifying truth functions. Amer. Math. Monthly **59**, 521–531 (1952)
392. Raman, V., Saurabh, S.: Short cycles make W-hard problems hard: FPT algorithms for W-hard problems in graphs with no short cycles. Algorithmica **52**(2), 203–225 (2008)
393. Raman, V., Saurabh, S., Subramanian, C.R.: Faster fixed parameter tractable algorithms for finding feedback vertex sets. ACM Transactions on Algorithms **2**(3), 403–415 (2006)
394. Razgon, I., O'Sullivan, B.: Almost 2-SAT is fixed-parameter tractable. J. Computer and System Sciences **75**(8), 435–450 (2009)
395. Reed, B.A.: Tree width and tangles: a new connectivity measure and some applications. In: Surveys in combinatorics, *London Math. Soc. Lecture Note Ser.*, vol. 241, pp. 87–162. Cambridge University Press (1997)
396. Reed, B.A.: Algorithmic aspects of tree width. In: Recent advances in algorithms and combinatorics, *CMS Books Math./Ouvrages Math. SMC*, vol. 11, pp. 85–107. Springer, New York (2003)
397. Reed, B.A., Smith, K., Vetta, A.: Finding odd cycle transversals. Operations Research Letters **32**(4), 299–301 (2004)
398. Robertson, N., Seymour, P.D.: Graph minors. III. Planar tree-width. J. Combinatorial Theory Ser. B **36**, 49–64 (1984)
399. Robertson, N., Seymour, P.D.: Graph minors. II. Algorithmic aspects of tree-width. J. Algorithms **7**(3), 309–322 (1986)
400. Robertson, N., Seymour, P.D.: Graph minors. V. Excluding a planar graph. J. Combinatorial Theory Ser. B **41**(1), 92–114 (1986)
401. Robertson, N., Seymour, P.D.: Graph minors. X. Obstructions to tree-decomposition. J. Combinatorial Theory Ser. B **52**(2), 153–190 (1991)
402. Robertson, N., Seymour, P.D.: Graph minors. XIII. The disjoint paths problem. J. Combinatorial Theory Ser. B **63**(1), 65–110 (1995)
403. Robertson, N., Seymour, P.D., Thomas, R.: Quickly excluding a planar graph. J. Combinatorial Theory Ser. B **62**(2), 323–348 (1994)

404. Robertson, N., Seymour, P.D., Thomas, R.: Sachs' linkless embedding conjecture. J. Combinatorial Theory Ser. B **64**(2), 185–227 (1995)
405. van Rooij, J.M.M., Bodlaender, H.L., Rossmanith, P.: Dynamic programming on tree decompositions using generalised fast subset convolution. In: Proceedings of the 17th Annual European Symposium on Algorithms (ESA), *Lecture Notes in Comput. Sci.*, vol. 5757, pp. 566–577. Springer (2009)
406. Rosen, K.: Discrete Mathematics and Its Applications. McGraw-Hill (1999)
407. Rué, J., Sau, I., Thilikos, D.M.: Dynamic programming for graphs on surfaces. ACM Transactions on Algorithms **10**(2), 8 (2014)
408. Ryser, H.J.: Combinatorial mathematics. The Carus Mathematical Monographs, No. 14. The Mathematical Association of America (1963)
409. Saurabh, S.: Exact algorithms for optimization and parameterized versions of some graph-theoretic problems. Ph.D. thesis, Homi Bhaba National Institute, Mumbai, India (2007)
410. Schönhage, A., Strassen, V.: Schnelle Multiplikation großer Zahlen. Computing **7**(3-4), 281–292 (1971)
411. Schrijver, A.: A short proof of Mader's $\mathcal{S}$-paths theorem. J. Combinatorial Theory Ser. B **82**(2), 319–321 (2001)
412. Schrijver, A.: Combinatorial optimization. Polyhedra and efficiency. Vol. A. Springer-Verlag, Berlin (2003)
413. Schwartz, J.T.: Fast probabilistic algorithms for verification of polynomial identities. J. ACM **27**(4), 701–717 (1980)
414. Seymour, P.D., Thomas, R.: Graph searching and a min-max theorem for tree-width. J. Combinatorial Theory Ser. B **58**(1), 22–33 (1993)
415. Seymour, P.D., Thomas, R.: Call routing and the ratcatcher. Combinatorica **14**(2), 217–241 (1994)
416. Shachnai, H., Zehavi, M.: Faster computation of representative families for uniform matroids with applications. CoRR **abs/1402.3547** (2014)
417. Sipser, M.: Introduction to the Theory of Computation, 1st edn. International Thomson Publishing (1996)
418. Stanley, R.P.: Enumerative combinatorics. Vol. 1, *Cambridge Studies in Advanced Mathematics*, vol. 49. Cambridge University Press (1997)
419. Stockmeyer, L.: Planar 3-colorability is polynomial complete. SIGACT News **5**(3), 19–25 (1973)
420. Thomassé, S.: A $4k^2$ kernel for feedback vertex set. ACM Transactions on Algorithms **6**(2) (2010)
421. Tollis, I.G., Di Battista, G., Eades, P., Tamassia, R.: Graph drawing. Prentice Hall, Upper Saddle River (1999)
422. Tyszkiewicz, J.: A simple construction for tournaments with every k players beaten by a single player. The American Mathematical Monthly **107**(1), 53–54 (2000)
423. Ueno, S., Kajitani, Y., Gotoh, S.: On the nonseparating independent set problem and feedback set problem for graphs with no vertex degree exceeding three. Discrete Mathematics **72**(1-3), 355–360 (1988)
424. Valiant, L.G., Vazirani, V.V.: NP is as easy as detecting unique solutions. Theoretical Computer Science **47**(3), 85–93 (1986)
425. Vassilevska Williams, V.: Multiplying matrices faster than Coppersmith-Winograd. In: Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC), pp. 887–898. ACM (2012)
426. Vatshelle, M.: New Width Parameters of Graphs. Ph.D. thesis, University of Bergen, Norway (2012)
427. Vazirani, V.V.: Approximation Algorithms. Springer (2001)
428. Wagner, K.: Über eine Eigenschaft der ebenen Komplexe. Math. Ann. **114**(1), 570–590 (1937)

429. Wahlström, M.: A tighter bound for counting max-weight solutions to 2SAT instances. In: Proceedings of the 3rd International Workshop on Parameterized and Exact Computation (IWPEC), *Lecture Notes in Comput. Sci.*, vol. 5018, pp. 202–213. Springer (2008)
430. Wahlström, M.: Half-integrality, LP-branching and FPT algorithms. In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1762–1781. SIAM (2014)
431. Wang, J., Ning, D., Feng, Q., Chen, J.: An improved kernelization for P_2-packing. Information Processing Letters **110**(5), 188–192 (2010)
432. Wang, J., Yang, Y., Guo, J., Chen, J.: Planar graph vertex partition for linear problem kernels. J. Computer and System Sciences **79**(5), 609–621 (2013)
433. Weisner, L.: Abstract theory of inversion of finite series. Transactions of the American Mathematical Society **38**(3), 474–484 (1935)
434. Welsh, D.J.A.: Matroid theory. Courier Dover Publications (2010)
435. Williams, R.: Finding paths of length k in $O(2^k)$ time. Information Processing Letters **109**(6), 315–318 (2009)
436. Woeginger, G.J.: Exact algorithms for NP-hard problems: A survey. In: Combinatorial Optimization - Eureka, you shrink!, *Lecture Notes in Comput. Sci.*, vol. 2570, pp. 185–207. Springer-Verlag, Berlin (2003)
437. Woeginger, G.J.: Space and time complexity of exact algorithms: Some open problems. In: Proceedings of the 1st International Workshop on Parameterized and Exact Computation (IWPEC), *Lecture Notes in Comput. Sci.*, vol. 3162, pp. 281–290. Springer-Verlag, Berlin (2004)
438. Wu, Y., Austrin, P., Pitassi, T., Liu, D.: Inapproximability of treewidth and related problems. J. Artif. Intell. Res. **49**, 569–600 (2014)
439. Xiao, M.: Simple and improved parameterized algorithms for multiterminal cuts. Theory of Computing Systems **46**(4), 723–736 (2010)
440. Yap, C.K.: Some consequences of non-uniform conditions on uniform classes. Theor. Comput. Sci. **26**, 287–300 (1983)
441. Yates, F.: The Design and Analysis of Factorial Experiments. Imperial Bureau of Soil Sciences, Harpenden (1937)
442. Zehavi, M.: Deterministic parameterized algorithms for matching and packing problems. CoRR **abs/1311.0484** (2013)
443. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: Proc. International Symposium on Symbolic and Algebraic Computation, *Lecture Notes in Comput. Sci.*, vol. 72, pp. 216–226 (1979)

Appendix: Notation

Graph notation

We try to use as much as possible graph notation compatible with the textbook of Diestel [138]. Here we recall the most basic definitions.

An *undirected graph* is a pair $G = (V, E)$, where V is some set and E is a family of 2-element subsets of V. The elements of V are the *vertices* of G, and the elements of E are the *edges* of G. The vertex set of a graph G is denoted by $V(G)$ and the edge set of G is denoted by $E(G)$. In this book all the graphs are finite, i.e., the sets $V(G)$ and $E(G)$ are finite. We use shorthands $n = |V(G)|$ and $m = |E(G)|$ whenever their use is not ambiguous. An edge of an undirected graph with endpoints u and v is denoted by uv; the endpoints u and v are said to be *adjacent*, and one is said to be a *neighbor* of the other. We also say that vertices u and v are *incident* to edge uv.

Unless specified otherwise, all the graphs are simple, which means that no two elements of $E(G)$ are equal, and that all edges from $E(G)$ have two different endpoints. If we allow multiple edges with the same endpoints (allowing $E(G)$ to be a multiset), or edges having both endpoints at the same vertex (allowing elements of $E(G)$ to be multisets of size 2 over $E(G)$, called *loops*), then we arrive at the notion of a *multigraph*. Whenever we use multigraphs, we state it explicitly, making clear whether multiple edges or loops are allowed.

Similarly, a *directed graph* is a pair $G = (V, E)$, where V is the set of vertices of G, and E is the set of edges of G, which this time are ordered pairs of two different vertices from V. Edges in directed graphs are often also called *arcs*. For an edge $(u, v) \in E(G)$, we say that it is *directed* from u to v, or simply that it goes from u to v. Then v is an *outneighbor* of u and u is an *inneighbor* of v. For an arc $a = (u, v)$, u is the *tail* and v is the *head* of a. Again, we can allow multiple edges with the same endpoints, as well as edges with the same head as tail (loops). Thus we arrive at the notion of a *directed multigraph*

.

For an undirected graph G and edge $uv \in E(G)$, by *contracting* edge uv we mean the following operation. We remove u and v from the graph, introduce a new vertex w_{uv}, and connect it to all the vertices u or v were adjacent to. Note that the operation, as we defined it here, transforms a simple graph into a simple graph; in other words, if u and v had a common neighbor x, then the new vertex w_{uv} will be connected to x only via one edge. We may also view the operation of contraction as an operation on multigraphs: every edge ux and vx for some $x \in V(G)$ gives rise to one new edge $w_{uv}x$, and in particular every copy of the edge uv apart from the contracted one gives rise to a new loop at w_{uv}. For us the default type of contraction is the one defined on simple graphs, and whenever we use the multigraph version, we state it explicitly.

Graph H is a *subgraph* of graph G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Similarly, graph H is a *supergraph* of graph G if $V(H) \supseteq V(G)$ and $E(H) \supseteq E(G)$. For a subset $S \subseteq V(G)$, the subgraph of G *induced* by S is denoted by $G[S]$; its vertex set is S and its edge set consists of all the edges of $E(G)$ that have both endpoints in S. For $S \subseteq V(G)$, we use $G - S$ to denote the graph $G[V \setminus S]$, and for $F \subseteq E(G)$ by $G - F$ we denote the graph $(V(G), E(G) \setminus F)$. We also write $G - v$ instead of $G - \{v\}$. The *neighborhood* of a vertex v in G is $N_G(v) = \{u \in V \ : \ uv \in E(G)\}$ and the *closed neighborhood* of v is $N_G[v] = N_G(v) \cup \{v\}$. For a vertex set $S \subseteq V$, we define $N_G[S] = \bigcup_{v \in S} N[v]$ and $N_G(S) = N_G[S] \setminus S$. We denote by $d_G(v)$ the *degree* of a vertex v in graph G, which is just the number of edges incident to v. We may omit indices if the graph under consideration is clear from the context. A graph G is called *r-regular* if all vertices of G have degree r. For directed graphs, the notions of *inneighborhood*, *outneighborhood*, *indegree*, and *outdegree* are defined analogously.

In an undirected graph G, a *walk* of length k is a nonempty sequence of vertices $v_0, \ldots, v_k$ such that for every $i = 0, \ldots, k-1$ we have $v_i v_{i+1} \in E(G)$. If $v_0 = v_k$, then the walk is *closed*. A *path* is a walk where no two vertices appear twice, and a *cycle* is a closed walk where no two vertices appear twice, apart from the vertex that appears at the beginning and at the end. We often think of paths and cycles as graphs induced by vertices and edges traversed by them. Then a path is a graph P of the form

$$V(P) = \{v_0, v_1, \ldots, v_k\} \quad E(P) = \{v_0v_1, v_1v_2, \ldots, v_{k-1}v_k\}.$$

If $P = v_0v_1 \ldots v_k$ is a path, then the graph obtained from P by adding edge x_kx_0 is a cycle. Again, the *length* of a path or cycle is equal to the cardinality of its edge set. We denote by $\mathrm{dist}(u, v)$ the *distance* between u and v in a graph G, which is the shortest length of a path between u and v. The notions of walks, paths, and cycles can be naturally generalized to directed graphs by requesting compliance with the edge directions.

The *girth* of a graph G is the shortest length of a cycle in G. A *Hamiltonian path (cycle)* in a graph G is a path (cycle) passing through all the vertices of

G. A G graph is *bipartite* if $V(G)$ can be partitioned into two subsets (A, B) such that every edge of $E(G)$ has one endpoint in A and the second in B. Partition (A, B) is also called a *bipartition* of G. Equivalently, G is bipartite if and only if G does not have any cycle of odd length.

An undirected graph G is connected if for every pair u, v of its vertices there is a path between u and v. A vertex set $X \subseteq V(G)$ is *connected* if the subgraph $G[X]$ is connected.

A *tree* T is a connected undirected graph without cycles. A tree T can be *rooted* at a vertex $r \in V(T)$, which imposes on $V(T)$ natural *parent-child* and *ancestor-descendant* relations. A *forest* F is an undirected graph without cycles; thus all the connected components of F are trees. A *spanning tree* T of a graph G is a tree such that $V(T) = V(G)$ and $E(T) \subseteq E(G)$. A directed graph G is *acyclic*, or a *DAG*, if it does not contain any directed cycles.

A *matching* M in a graph G is a set of edges that pairwise do not share endpoints. A vertex of G is *saturated* if it is incident to an edge in the matching. Otherwise the vertex is *unsaturated*. For a given matching M, an *alternating path* is a path in which the edges alternately belong to the matching and do not belong to the matching. An *augmenting* path is an alternating path that starts from and ends at an unsaturated vertex. A *perfect matching* is a matching M that saturates all the vertices of the graph, i.e., every vertex of the graph is an endpoint of an edge in M.

An *independent set* I in a graph G is a subset of the vertex set $V(G)$ such that the vertices of I are pairwise nonadjacent. A *clique* C in a graph G is a subset of the vertex set $V(G)$ such that the vertices of C are pairwise adjacent. A *vertex cover* X of a graph G is a subset of the vertex set $V(G)$ such that X covers the edge set $E(G)$, i.e., every edge of G has at least one endpoint in X. A *dominating set* D of a graph G is a subset of the vertex set $V(G)$ such that every vertex of $V(G) \setminus D$ has a neighbor in D. A *proper coloring* of a graph G assigns a *color* to each vertex of G in such a manner that adjacent vertices receive distinct colors. The *chromatic number* of G is the minimum χ such that there is a proper coloring of G using χ colors.

We now define the notion of a *planar graph* and an embedding of a graph into the plane. First, instead of embedding into the plane we will equivalently embed our graphs into a sphere: in this manner, we do not distinguish unnecessarily the outer face of the embedding. Formally, an *embedding* of a graph G into a sphere is a mapping that maps (a) injectively each vertex of G into a point of the sphere, and (b) each edge uv of G into a Jordan curve connecting the images of u and v, such that the curves are pairwise disjoint (except for the endpoints) and do not pass through any other image of a vertex. A *face* is a connected component of the complement of the image of G in the sphere; if G is connected, each face is homeomorphic to an open disc. A *planar graph* is a graph that admits an embedding into a sphere, and a *plane graph* is a planar graph together with one fixed embedding.

The classic theorems of Kuratowski and Wagner state that a graph is planar if and only if it does not contain K_5 or $K_{3,3}$ as a minor (equivalently, as a topological minor). In Chapters 6 and 7 we briefly mention graphs embeddable into more complicated surfaces than plane or sphere. We refer to the bibliographic notes at the end of Chapter 6 for pointers to the literature on this topic.

SAT notation

Let $\mathtt{Vars} = \{x_1, x_2, \ldots, x_n\}$ be a set of *Boolean variables.* A variable x or a negated variable $\neg x$ is called a *literal.* A propositional formula φ is in *conjunctive normal form*, or is a *CNF formula*, if it is of the form:

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_m.$$

Here, each C_i is a *clause* of the form

$$C_i = \ell_1^i \vee \ell_2^i \vee \ldots \vee \ell_{r_i}^i,$$

where ℓ_j^i are literals of some variables of $\mathtt{Vars}$. The number of literals r_i in a clause C_i is called the *length* of the clause, and is denoted by $|C_i|$. The *size* of formula φ is defined as $|\varphi| = \sum_{i=1}^{m} |C_i|$. The set of clauses of a CNF formula is usually denoted by $\mathtt{Cls}$.

For $q \geq 2$, a CNF formula φ is in *q-CNF* if every clause from φ has at most q literals. If φ is a formula and X a set of variables, then we denote by $\varphi - X$ the formula obtained from φ after removing all the clauses that contain a literal of a variable from X.

For a CNF formula φ on variables $\mathtt{Vars}$, a *truth assignment* is a mapping $\psi\colon \mathtt{Vars} \to \{\bot, \top\}$. Here, we denote the false value as $\bot$, and the truth value as $\top$. This assignment can be naturally extended to literals by taking $\psi(\neg x) = \neg\psi(x)$ for each $x \in \mathtt{Vars}$. A truth assignment ψ *satisfies* a clause C of φ if and only if C contains some literal ℓ with $\psi(\ell) = \top$; ψ satisfies formula φ if it satisfies all the clauses of φ. A formula is *satisfiable* if it is satisfied by some truth assignment; otherwise it is *unsatisfiable.*

The notion of a truth assignment can be naturally generalized to partial assignments that valuate only some subset $X \subseteq \mathtt{Vars}$; i.e., ψ is a mapping from X to $\{\bot, \top\}$. Here, a clause C is satisfied by ψ if and only if C contains some literal ℓ whose variable belongs to X, and which moreover satisfies $\psi(\ell) = \top$.

Problem definitions

(p,q)-Cluster

Input:
A graph G, a vertex $v \in V(G)$ and integers p and q.
Question:
Does there exist a (p,q)-cluster containing v, that is, a set $C \subseteq V(G)$ such that $v \in C$, $|C| \leq p$, and $d(C) \leq q$?

(p,q)-Partition

Input:
A graph G and integers p and q.
Question:
Does there exist a partition of $V(G)$ into (p,q)-clusters? Here, a set $C \subseteq V(G)$ is a (p,q)-cluster if $|C| \leq p$ and $d(C) \leq q$.

2-Matroid Intersection

Input:
A universe U, two matrices representing matroids $\mathcal{M}_1, \mathcal{M}_2$ over U, and an integer k.
Question:
Does there exist a set $S \subseteq U$ of size at least k that is independent both in $\mathcal{M}_1$ and $\mathcal{M}_2$?

2-SAT

Input:
A CNF formula φ, where every clause consists of at most two literals.
Question:
Does there exist a satisfying assignment for φ?

2-degenerate Vertex Deletion

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is 2-degenerate?

$2k \times 2k$ Bipartite Permutation Independent Set

Input:
An integer k and a graph G with the vertex set $[2k] \times [2k]$, where every edge is between $I_1 = [k] \times [k]$ and $I_2 = ([2k] \setminus [k]) \times ([2k] \setminus [k])$.
Question:
Does there exist an independent set $X \subseteq I_1 \cup I_2$ in G that induces a permutation of $[2k]$?

3-Coloring

Input:
A graph G.
Question:
Does there exist a coloring $c : V(G) \to \{1,2,3\}$ such that $c(u) \neq c(v)$ for every $uv \in E(G)$?

3-Hitting Set

Input:
A universe U, a family $\mathcal{A}$ of sets over U, where each set in $\mathcal{A}$ is of size at most 3, and an integer k.
Question:
Does there exist a set $X \subseteq U$ of size at most k that has a nonempty intersection with every element of $\mathcal{A}$?

3-Matroid Intersection

Input:
A universe U, three matrices representing matroids $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$ over U, and an integer k.
Question:
Does there exist a set $S \subseteq U$ of size at least k that is independent in every matroid $\mathcal{M}_i$, $i = 1, 2, 3$?

3-SAT

Input:
A CNF formula φ, where every clause consists of at most three literals.
Question:
Does there exist a satisfying assignment for φ?

$\mathcal{G}$ Vertex Deletion

Input:
A graph G, an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X \in \mathcal{G}$? Here, $\mathcal{G}$ denotes any graph class that is a part of the problem definition.

ℓ-Matroid Intersection

Input:
A universe U, ℓ matrices representing matroids $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_\ell$ over U, and an integer k.
Question:
Does there exist a set $S \subseteq U$ of size at least k that is independent in every matroid $\mathcal{M}_i$, $1 \le i \le \ell$?

ϕ-Maximization

Input:
A graph G and an integer k.
Question:
Does there exist a set S of at least k vertices of G such that $\phi(G, S)$ is true? Here, ϕ denotes any computable Boolean predicate that is a part of the problem definition.

ϕ-Minimization

Input:
A graph G and an integer k.
Question:
Does there exist a set S of at most k vertices of G such that $\phi(G, S)$ is true? Here, ϕ denotes any computable Boolean predicate that is a part of the problem definition.

d-Bounded-Degree Deletion

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that the maximum degree of $G - X$ is at most d?

d-Clustering

Input:
A graph G and an integer k.
Question:
Does there exist a set $A \subseteq \binom{V(G)}{2}$ of size at most k such that the graph $(V(G), (E(G) \setminus A) \cup (A \setminus E(G)))$ is a d-cluster graph? Here, a d-cluster graph is a graph that contains exactly d connected components, each being a clique.

d-Hitting Set

Input:
A universe U, a family $\mathcal{A}$ of sets over U, where each set in $\mathcal{A}$ is of size at most d, and an integer k.
Question:
Does there exist a set $X \subseteq U$ of size at most k that has a nonempty intersection with every element of $\mathcal{A}$?

d-Set Packing

Input:
A universe U, a family $\mathcal{A}$ of sets over U, where each set in $\mathcal{A}$ is of size at most d, and an integer k.
Question:
Does there exist a family $\mathcal{A}' \subseteq \mathcal{A}$ of k pairwise disjoint sets?

k-Tree

A synonym for Tree Subgraph Isomorphism.

$k \times k$ Clique

Input:
An integer k and a graph G with a vertex set $[k] \times [k]$.
Question:
Does there exist a set $X \subseteq V(G)$ that is a clique in G and that contains exactly one element in each row $\{i\} \times [k]$, $1 \le i \le k$?

$k \times k$ Hitting Set

Input:
An integer k and a family $\mathcal{A}$ of subsets of $[k] \times [k]$.
Question:
Does there exist a set $X \subseteq [k] \times [k]$ that has a nonempty intersection with every element of $\mathcal{A}$ and that contains exactly one element in each row $\{i\} \times [k]$, $1 \leq i \leq k$?

$k \times k$ Hitting Set with thin sets

Input:
An integer k and a family $\mathcal{A}$ of subsets of $[k] \times [k]$, where every element of $\mathcal{A}$ contains at most one element in each row $\{i\} \times [k]$, $1 \leq i \leq k$.
Question:
Does there exist a set $X \subseteq [k] \times [k]$ that has a nonempty intersection with every element of $\mathcal{A}$ and that contains exactly one element in each row $\{i\} \times [k]$, $1 \leq i \leq k$?

$k \times k$ Permutation Clique

Input:
An integer k and a graph G with a vertex set $[k] \times [k]$.
Question:
Does there exist a set $X \subseteq V(G)$ that is a clique in G and that induces a permutation of $[k]$?

$k \times k$ Permutation Hitting Set

Input:
An integer k and a family $\mathcal{A}$ of subsets of $[k] \times [k]$.
Question:
Does there exist a set $X \subseteq [k] \times [k]$ that has a nonempty intersection with every element of $\mathcal{A}$ and that induces a permutation of $[k]$?

$k \times k$ Permutation Hitting Set with thin sets

Input:
An integer k and a family $\mathcal{A}$ of subsets of $[k] \times [k]$, where every element of $\mathcal{A}$ contains at most one element in each row $\{i\} \times [k]$, $1 \leq i \leq k$.
Question:
Does there exist a set $X \subseteq [k] \times [k]$ that has a nonempty intersection with every element of $\mathcal{A}$ and that induces a permutation of $[k]$?

$k \times k$ Permutation Independent Set

Input:
An integer k and a graph G with a vertex set $[k] \times [k]$.
Question:
Does there exist a set $X \subseteq V(G)$ that is an independent set in G and that induces a permutation of $[k]$?

q-Coloring

Input:
A graph G.
Question:
Does there exist a coloring $c : V(G) \to [q]$ such that $c(u) \neq c(v)$ for every $uv \in E(G)$?

q-SAT

Input:
A CNF formula φ, where every clause consists of at most q literals.
Question:
Does there exist a satisfying assignment for φ?

q-Set Cover

Input:
A universe U, a family $\mathcal{F}$ over U, where every element of $\mathcal{F}$ is of size at most q, and an integer k.
Question:
Does there exist a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ of size at most k such that $\bigcup \mathcal{F}' = U$?

r-Center

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that every vertex of G is within distance at most r from at least one vertex of X?

s-Way Cut

Input:
A graph G and integers s and k.
Question:
Does there exist a set X of at most k edges of G such that $G - X$ has at least s connected components?

Almost 2-SAT

Input:
A CNF formula φ, where every clause consists of at most two literals, and an integer k.
Question:
Is it possible to make φ satisfiable by deleting at most k clauses?

Annotated Bipartite Coloring

Input:
A bipartite graph G, two sets $B_1, B_2 \subseteq V(G)$, and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ has a proper 2-coloring $c : V(G) \setminus X \to \{1,2\}$ (i.e., $c(u) \neq c(v)$ for every edge uv) that agrees with the sets B_1 and B_2, that is, $f(v) = i$ whenever $v \in B_i \setminus X$ and $i = 1, 2$?

Annotated Satisfiable Almost 2-SAT

Input:
A *satisfiable* formula φ in CNF form, where every clause of φ consists of at most two literals, two sets of variables $V^\top$ and $V^\perp$, and an integer k.
Question:
Does there exist a set X of at most k variables of φ and a satisfying assignment ψ of $\varphi - X$ such that $\psi(x) = \top$ for every $x \in V^\top \setminus X$ and $\psi(x) = \perp$ for every $x \in V^\perp \setminus X$? Here, $\varphi - X$ denotes the formula φ with every clause containing at least one variable of X deleted.

Bar Fight Prevention

Input:
A bar, a list of n potential guests, for every pair of guests a prediction whether they will fight if they are admitted together to the bar, and an integer k.
Question:
Can you identify a set of at most k troublemakers, so that if you admit to the bar everybody except for the troublemakers, no fight breaks out among the admitted guests?

Bipartite Matching

Input:
A bipartite graph G and an integer k.
Question:
Does there exist an edge set $S \subseteq E(G)$ of size at least k such that no two edges in S share an endpoint?

CNF-SAT

Input:
A formula φ in conjunctive normal form (CNF).
Question:
Does there exist a satisfying assignment for φ?

Chordal Completion

Input:
A graph G and an integer k.
Question:
Can one add at most k edges to G to turn it into a chordal graph?

Chromatic Number

Input:
A graph G.
Question:
What is the minimum integer q, such that there exists a coloring $c : V(G) \to [q]$ satisfying $c(u) \neq c(v)$ for every edge $uv \in E(G)$?

Clique

Input:
A graph G and an integer k.
Question:
Does there exist a set of k vertices of G that is a clique in G?

Clique$_{\log}$

Input:
A graph G and an integer k such that $k \leq \log_2 |V(G)|$.
Question:
Does there exist a set of k vertices of G that is a clique in G?

Closest String

Input:
A set of k strings $x_1, \ldots, x_k$, each string over an alphabet Σ and of length L, and an integer d.
Question:
Does there exist string y of length L over Σ such that $d_H(y, x_i) \leq d$ for every $1 \leq i \leq k$. Here, $d_H(x, y)$ is the *Hamming distance* between strings x and y, that is, the number of positions where x and y differ.

Closest Substring

Input:
A set of k strings $x_1, \ldots, x_k$ over an alphabet Σ, and integers L and d
Question:
Does there exist a string s of length L such that, for every $1 \leq i \leq k$, x_i has a substring x'_i of length L with $d_H(s, s_i) \leq d$. Here, $d_H(x, y)$ is the *Hamming distance* between strings x and y, that is, the number of positions where x and y differ.

Cluster Editing

Input:
A graph G and an integer k.
Question:
Does there exist a set $A \subseteq \binom{V(G)}{2}$ of size at most k such that the graph $(V(G), (E(G) \setminus A) \cup (A \setminus E(G)))$ is a cluster graph? Here, a cluster graph is a graph where every connected component is a clique.

Cluster Vertex Deletion

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is a cluster graph? Here, a cluster graph is a graph where every connected component is a clique.

Cochromatic Number

Input:
A graph G.
Question:
What is the minimum integer q, such that there exists a coloring $c : V(G) \to [q]$ satisfying $c(u) \neq c(v)$ for every pair of nonadjacent vertices u, v?

Colored Red-Blue Dominating Set

Input:
A bipartite graph G with bipartition classes $R \uplus B = V(G)$, an integer ℓ and a partition of R into ℓ sets $R^1, R^2, \ldots, R^\ell$.
Question:
Does there exist a set $X \subseteq R$ that contains exactly one element of every set R^i, $1 \leq i \leq \ell$ and such that $N_G(X) = B$?

Colorful Graph Motif

Input:
A graph G, an integer k and a coloring $c : V(G) \to [k]$.
Question:
Does there exist a connected subgraph of G that contains exactly one vertex of each color?

Component Order Integrity

Input:
A graph G and two integers k and ℓ.
Question:
Does there exist a set X of at most k vertices of G such that every connected component of $G - X$ contains at most ℓ vertices?

Connected Bull Hitting

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G[X]$ is connected and $G - X$ does not contain a bull as an induced subgraph? Here, a bull is a 5-vertex graph H with $V(H) = \{a, b, c, d, e\}$ and $E(H) = \{ab, bc, ac, bd, ce\}$.

Connected Dominating Set

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G[X]$ is connected and $N_G[X] = V(G)$?

Connected Feedback Vertex Set

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G[X]$ is connected and $G - X$ is a forest?

Connected Vertex Cover

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G[X]$ is connected and $G - X$ is edgeless?

Cycle Packing

Input:
A graph G and an integer k.
Question:
Does there exist in G a family of k pairwise vertex-disjoint cycles?

Digraph Pair Cut

Input:
A directed graph G, a designated vertex $s \in V(G)$, a family of pairs of vertices $\mathcal{F} \subseteq \binom{V(G)}{2}$, and an integer k.
Question:
Does there exist a set X of at most k edges of G, such that for each pair $\{u, v\} \in \mathcal{F}$, either u or v is not reachable from s in the graph $G - X$.

Directed Edge Multicut

Input:
A directed graph G, a set of pairs $(s_i, t_i)_{i=1}^{\ell}$ of vertices of G, and an integer k.
Question:
Does there exist a set X of at most k edges of G such that for every $1 \leq i \leq \ell$, vertex t_i is not reachable from vertex s_i in the graph $G - X$?

Directed Edge Multiway Cut

Input:
A directed graph G, a set $T \subseteq V(G)$, and an integer k.
Question:
Does there exist a set X of at most k edges of G such that for every two distinct vertices $t_1, t_2 \in T$, vertex t_2 is not reachable from vertex t_1 in the graph $G - X$?

Directed Feedback Arc Set

Input:
A directed graph G and an integer k.
Question:
Does there exist a set X of at most k edges of G such that $G - X$ is acyclic?

Directed Feedback Arc Set Compression

Input:
A directed graph G, a set $W \subseteq V(G)$ such that $G - W$ is acyclic, and an integer k.
Question:
Does there exist a set X of at most k edges of G such that $G - X$ is acyclic?

Directed Feedback Vertex Set

Input:
A directed graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is acyclic?

Directed Max Leaf

Input:
A directed graph G and an integer k.
Question:
Does there exist an out-tree in G with at least k leaves? Here, an out-tree is a directed graph whose underlying undirected graph is a tree, and every vertex of the out-tree, except for one, has indegree exactly one; a leaf of an out-tree is a vertex with outdegree zero.

Directed Steiner Tree

Input:
A directed graph G, a designated root vertex r, a set $T \subseteq V(G)$, and an integer ℓ.
Question:
Does there exist a subgraph H of G with at most ℓ edges, such that every vertex $t \in T$ is reachable from r in H?

Directed Vertex Multiway Cut

Input:
A directed graph G, a set $T \subseteq V(G)$, and an integer k.
Question:
Does there exist a set $X \subseteq V(G) \setminus T$ of size at most k such that for every two distinct vertices $t_1, t_2 \in T$, vertex t_2 is not reachable from vertex t_1 in the graph $G - X$?

Disjoint Factors

Input:
A word w over an alphabet $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_s\}$.
Question:
Does there exist pairwise disjoint subwords $u_1, u_2, \ldots, u_s$ of w such that each u_i is of length at least two and begins and ends with γ_i?

Disjoint Feedback Vertex Set

Input:
A graph G, a set $W \subseteq V(G)$ such that $G - W$ is a forest, and an integer k.
Question:
Does there exist a set $X \subseteq V(G) \setminus W$ of size at most k such that $G - X$ is a forest?

Disjoint Feedback Vertex Set in Tournaments

Input:
A tournament G, a set $W \subseteq V(G)$ such that $G - W$ is acyclic, and an integer k.
Question:
Does there exist a set $X \subseteq V(G) \setminus W$ of size at most k such that $G - X$ is acyclic?

Disjoint Odd Cycle Transversal

Input:
A graph G, a set $W \subseteq V(G)$ such that $G - W$ is bipartite, and an integer k.
Question:
Does there exist a set $X \subseteq V(G) \setminus W$ of size at most k such that $G - X$ is bipartite?

Disjoint Planar Vertex Deletion

Input:
A graph G, a set $W \subseteq V(G)$ such that $G - W$ is planar, and an integer k.
Question:
Does there exist a set $X \subseteq V(G) \setminus W$ of size at most k such that $G - X$ is planar?

Disjoint Vertex Cover

Input:
A graph G, a set $W \subseteq V(G)$ such that $G - W$ is edgeless, and an integer k.
Question:
Does there exist a set $X \subseteq V(G) \setminus W$ of size at most k such that $G - X$ is edgeless?

Distortion

Input:
A graph G and an integer d.
Question:
Does there exist an embedding $\eta : V(G) \to \mathbb{Z}$ such that for every $u, v \in V(G)$ we have $\mathrm{dist}_G(u,v) \leq |\eta(u) - \eta(v)| \leq d \cdot \mathrm{dist}_G(u,v)$?

Dominating Set

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $N_G[X] = V(G)$?

Dominating Set on Tournaments

Input:
A tournament G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that for every $v \in V(G) \setminus X$ there exists $u \in X$ with $(u,v) \in E(G)$?

Dominating Set with Pattern

Input:
A graph G, an integer k, and a graph H on vertex set $[k]$
Question:
Does there exist a tuple $(v_1, v_2, \ldots, v_k)$ of distinct vertices of G such that $N_G[\{v_1, v_2, \ldots, v_k\}] = V(G)$ and $v_i v_j \in E(G)$ if and only if $ij \in E(H)$?

Dual-Coloring

Input:
A graph G and an integer k.
Question:
Does there exist a coloring $c : V(G) \to [n-k]$ such that $c(u) \neq c(v)$ for every edge uv?

Ed-Hitting Set

Input:
A universe U, a family $\mathcal{A}$ of sets over U, where each set in $\mathcal{A}$ is of size exactly d, and an integer k.
Question:
Does there exist a set $X \subseteq U$ of size at most k that has a nonempty intersection with every element of $\mathcal{A}$?

Ed-Set Packing

Input:
A universe U, a family $\mathcal{A}$ of sets over U, where each set in $\mathcal{A}$ is of size exactly d, and an integer k.
Question:
Does there exist a family $\mathcal{A}' \subseteq \mathcal{A}$ of k pairwise disjoint sets?

Edge Bipartization

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k edges of G such that $G - X$ is bipartite?

Edge Clique Cover

Input:
A graph G and an integer k.
Question:
Does there exist k subgraphs $H_1, H_2, \ldots, H_k$ of G, such that each H_i is a clique and $E(G) = \bigcup_{i=1}^{k} E(H_i)$?

Edge Disjoint Cycle Packing

Input:
A graph G and an integer k.
Question:
Does there exist in G a family of k pairwise edge-disjoint cycles?

Edge Dominating Set

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k edges of G such that $G - V(X)$ is edgeless?

Edge Multicut

Input:
A graph G, a set of pairs $(s_i, t_i)_{i=1}^{\ell}$ of vertices of G, and an integer k.
Question:
Does there exist a set X of at most k edges of G such that for every $1 \leq i \leq \ell$, vertices s_i and t_i lie in different connected components of $G - X$?

Edge Multiway Cut

Input:
A graph G, a set $T \subseteq V(G)$, and an integer k.
Question:
Does there exist a set X of at most k edges of G such that every element of T lies in a different connected component of $G - X$?

Edge Multiway Cut for Sets

Input:
A graph G, pairwise disjoint sets $T_1, \ldots, T_p$ of vertices of G, and an integer k.
Question:
Does there exist a set X of at most k edges of G such that for every $1 \leq i < j \leq p$ there is no path between any vertex of T_i and any vertex of T_j in $G - X$?

Eulerian Deletion

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k edges of G such that $G - X$ contains an Euler tour?

Even Set

Input:
A universe U, a family $\mathcal{F}$ of subsets of U, and an integer k.
Question:
Does there exist a nonempty set $X \subseteq U$ of size at most k such that $|A \cap X|$ is even for every $A \in \mathcal{F}$?

Exact CNF-SAT

Input:
A formula φ in CNF and an integer k.
Question:
Does there exist an assignment ψ that satisfies φ and sets to true exactly k variables?

Exact Even Set

Input:
A universe U, a set $\mathcal{F}$ of subsets of U, and an integer k.
Question:
Does there exist a nonempty set $X \subseteq U$ of size exactly k such that $|A \cap X|$ is even for every $A \in \mathcal{F}$?

Exact Odd Set

Input:
A universe U, a set $\mathcal{F}$ of subsets of U, and an integer k.
Question:
Does there exist a nonempty set $X \subseteq U$ of size exactly k such that $|A \cap X|$ is odd for every $A \in \mathcal{F}$?

Exact Unique Hitting Set

Input:
A universe U, a set $\mathcal{A}$ of subsets of U, and an integer k.
Question:
Does there exist a set $X \subseteq U$ of size exactly k such that $|A \cap X| = 1$ for every $A \in \mathcal{A}$?

FAST

An abbreviation for Feedback Arc Set in Tournaments.

FVST

An abbreviation for Feedback Vertex Set in Tournaments.

Face Cover

Input:
A graph G embedded on a plane and an integer k.
Question:
Does there exist a set X of at most k faces of the embedding of G such that every vertex of G lies on at least one face of X?

Feedback Arc Set in Tournaments

Input:
A tournament G and an integer k.
Question:
Does there exist a set X of at most k edges of G such that $G - X$ is acyclic?

Feedback Vertex Set

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is a forest?

Feedback Vertex Set in Tournaments

Input:
A tournament G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is acyclic?

Feedback Vertex Set in Tournaments Compression

Input:
A tournament G, a set $W \subseteq V(G)$ such that $G - W$ is acyclic, and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is acyclic?

Graph Genus

Input:
A graph G and an integer g.
Question:
Can G be embedded on a surface of Euler genus g?

Graph Isomorphism

Input:
Two graphs G and H.
Question:
Are G and H isomorphic?

Graph Motif

Input:
A graph G, an integer k, a coloring $c : V(G) \to [k]$, and a multiset M with elements from $[k]$.
Question:
Does there exist a set $X \subseteq V(G)$ such that $G[X]$ is connected and the multiset $\{c(u) : u \in X\}$ equals M?

Grid Tiling

Input:
An integer k, an integer n, and a collection $\mathcal{S}$ of k^2 nonempty sets $S_{i,j} \subseteq [n] \times [n]$ $(1 \leq i, j \leq k)$.
Question:
Can one choose for each $1 \leq i, j \leq k$ a pair $s_{i,j} \in S_{i,j}$ such that

- If $s_{i,j} = (a, b)$ and $s_{i+1,j} = (a', b')$, then $a = a'$.
- If $s_{i,j} = (a, b)$ and $s_{i,j+1} = (a', b')$, then $b = b'$.

Grid Tiling with $\leq$

Input:
An integer k, an integer n, and a collection $\mathcal{S}$ of k^2 nonempty sets $S_{i,j} \subseteq [n] \times [n]$ $(1 \leq i, j \leq k)$.
Question:
Can one choose for each $1 \leq i, j \leq k$ a pair $s_{i,j} \in S_{i,j}$ such that

- If $s_{i,j} = (a, b)$ and $s_{i+1,j} = (a', b')$, then $a \leq a'$.
- If $s_{i,j} = (a, b)$ and $s_{i,j+1} = (a', b')$, then $b \leq b'$.

Hall Set

Input:
A bipartite graph G with bipartition classes $A \uplus B = V(G)$ and an integer k.
Question:
Does there exist a set $X \subseteq A$ of size at most k such that $|N_G(S)| < |S|$?

Halting

Input:
A description of a deterministic Turing machine M, an input word x.
Question:
Does M halt on x?

Hamiltonian Cycle

Input:
A graph G.
Question:
Does there exist a simple cycle C in G such that $V(C) = V(G)$?

Hamiltonian Path

Input:
A graph G.
Question:
Does there exist a simple path P in G such that $V(P) = V(G)$?

Hitting Set

Input:
A universe U, a family $\mathcal{A}$ of sets over U, and an integer k.
Question:
Does there exist a set $X \subseteq U$ of size at most k that has a nonempty intersection with every element of $\mathcal{A}$?

Imbalance

Input:
A graph G and an integer k.
Question:
Does there exist a bijective function $\pi : V(G) \to \{1, 2, \dots, |V(G)|\}$ (called an *ordering*) whose imbalance is at most k? Here, the imbalance of a vertex $v \in V(G)$ in an ordering π is defined as

$$\iota_\pi(v) = ||\{u \in N_G(v) \ : \ \pi(u) < \pi(v)\}| - \\ |\{u \in N_G(v) \ : \ \pi(u) > \pi(v)\}||,$$

and the imbalance of an ordering π is defined as

$$\iota(\pi) = \sum_{v \in V(G)} \iota_\pi(v).$$

Independent Dominating Set

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G[X]$ is edgeless and $N_G[X] = V(G)$?

Independent Feedback Vertex Set

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G[X]$ is edgeless and $G - X$ is a forest?

Independent Set

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G[X]$ is edgeless?

Induced Matching

Input:
A graph G and an integer k.
Question:
Does there exist a set X of exactly $2k$ vertices of G such that $G[X]$ is a matching consisting of k edges?

Integer Linear Programming

Input:
Integers m and p, a matrix $A \in \mathbb{Z}^{m \times p}$, and vectors $b \in \mathbb{Z}^m$, $c \in \mathbb{Z}^p$.
Question:
Find a vector $x \in \mathbb{Z}^p$ that satisfies $Ax \leq b$ and that minimizes the value of the scalar product $c \cdot x$.

Integer Linear Programming Feasibility

Input:
Integers m and p, a matrix $A \in \mathbb{Z}^{m \times p}$, and a vector $b \in \mathbb{Z}^m$.
Question:
Does there exist a vector $x \in \mathbb{Z}^p$ such that $Ax \leq b$?

Linear Programming

Input:
Integers m and p, a matrix $A \in \mathbb{R}^{m \times p}$, and vectors $b \in \mathbb{R}^m$, $c \in \mathbb{R}^p$.
Question:
Find a vector $x \in \mathbb{R}^p$ that satisfies $Ax \leq b$ and that minimizes the value of the scalar product $c \cdot x$.

Linkless Embedding

Input:
A graph G and an integer k.
Question:
Does there exist an embedding of G into $\mathbb{R}^3$ such that any pairwise linked family of cycles in G has size at most k? Here, we say that two vertex-disjoint cycles of G are linked in an embedding into $\mathbb{R}^3$ if they cannot be separated by a continuous deformation (i.e., they look like two consecutive links of a chain).

List Coloring

Input:
A graph G and a set $L(v)$ for every $v \in V(G)$
Question:
Can one choose a color $c(v) \in L(v)$ for every $v \in V(G)$ such that $c(u) \neq c(v)$ for every $uv \in E(G)$?

Long Directed Cycle

Input:
A directed graph G and an integer k.
Question:
Does there exist a directed cycle in G of length at least k?

Long Induced Path

Input:
A graph G and an integer k.
Question:
Does there exist an induced subgraph of G that is isomorphic to a path on k vertices?

Longest Common Subsequence

Input:
Two strings a and b.
Question:
What is the length of the longest common subsequence of a and b? That is, we ask for the largest possible integer n for which there exist indices $1 \leq i_1 < i_2 < \ldots < i_n \leq |a|$ and $1 \leq j_1 < j_2 < \ldots < j_n \leq |b|$ such that $a[i_r] = b[j_r]$ for every $1 \leq r \leq n$.

Longest Cycle

Input:
A graph G and an integer k.
Question:
Does there exist a cycle in G of length at least k?

Longest Path

Input:
A graph G and an integer k.
Question:
Does there exist a path in G consisting of k vertices?

Matroid Ed-Set Packing

Input:
An integer k and a matrix M representing a matroid $\mathcal{M}$ of rank $d \cdot k$ over a universe U, and a family $\mathcal{A}$ of sets over U, where each set in $\mathcal{A}$ is of size exactly d.
Question:
Does there exist a family $\mathcal{A}' \subseteq \mathcal{A}$ of k pairwise disjoint sets, such that $\bigcup_{A \in \mathcal{A}'} A$ is independent in $\mathcal{M}$?

Matroid Parity

Input:
A matrix A representing a matroid $\mathcal{M}$ over a universe U of size $2n$, a partition of U into n pairs $P_1, P_2, \ldots, P_n$, and an integer k.
Question:
Does there exist an independent set X in $\mathcal{M}$ of size $2k$ that is a union of k pairs P_i?

Max Leaf Spanning Tree

Input:
A graph G and an integer k.
Question:
Does there exist a spanning tree of G with at least k leaves?

Max Leaf Subtree

Input:
A graph G and an integer k.
Question:
Does G contain a tree with at least k leaves as a subgraph?

Max-r-SAT

Input:
A CNF formula φ, where every clause consists of at most r literals, and an integer k.
Question:
Does there exist an assignment ψ that satisfies at least k clauses of φ?

Max-Er-SAT

Input:
A CNF formula φ, where every clause consists of exactly r literals with pairwise different variables, and an integer k.
Question:
Does there exist an assignment ψ that satisfies at least k clauses of φ?

Max-Internal Spanning Tree

Input:
A graph G and an integer k.
Question:
Does there exist a spanning tree of G with at least k internal vertices?

MaxCut

Input:
A graph G and an integer k.
Question:
Does there exist a partition $A \uplus B$ of $V(G)$ such that at least k edges of G have one endpoint in A and the second endpoint in B?

Maximum Bisection

Input:
A graph G and an integer k.
Question:
Does there exist a partition $A \uplus B$ of $V(G)$ such that $||A| - |B|| \leq 1$ and at least k edges of G have one endpoint in A and the second endpoint in B?

Maximum Cycle Cover

Input:
A graph G and an integer k.
Question:
Does there exist a family $\mathcal{C}$ of pairwise vertex-disjoint cycles in G such that $|\mathcal{C}| \geq k$ and every vertex of G lies on some cycle of $\mathcal{C}$?

Maximum Flow

Input:
A directed graph G with edge capacity function $c : E(G) \to \mathbb{R}_{>0}$ and two designated vertices $s, t \in V(G)$.
Question:
Find a maximum flow from s to t. That is, find a function $f : E(G) \to \mathbb{R}_{>0}$ that satisfies the capacity constraints $0 \leq f(e) \leq c(e)$ for every $e \in E(G)$, flow conservation constraints $\sum_{(v,u)\in E(G)} f(v,u) = \sum_{(u,v\in E(G)} f(u,v)$ for every $v \in V(G) \setminus \{s,t\}$, and that maximizes $\sum_{(s,u)\in E(G)} f(s,u) - \sum_{(u,s)\in E(G)} f(u,s)$.

Maximum Matching

Input:
A graph G and an integer k.
Question:
Does there exist an edge set $S \subseteq E(G)$ of size at least k such that no two edges in S share an endpoint?

Maximum Satisfiability

Input:
A CNF formula φ and an integer k.
Question:
Does there exist an assignment ψ that satisfies at least k clauses of φ?

Min-2-SAT

Input:
A CNF formula φ, where every clause consists of at most two literals, and an integer k.
Question:
Does there exist an assignment ψ that satisfies at most k clauses of φ?

Min-Ones-2-SAT

Input:
A CNF formula φ, where every clause consists of at most two literals, and an integer k.
Question:
Does there exist an assignment ψ that satisfies φ and sets at most k variables to true?

Min-Ones-r-SAT

Input:
A CNF formula φ, where every clause consists of at most r literals, and an integer k.
Question:
Does there exist an assignment ψ that satisfies φ and sets at most k variables to true?

Minimum Bisection

Input:
A graph G and an integer k.
Question:
Does there exist a partition $A \uplus B$ of $V(G)$ such that $||A| - |B|| \leq 1$ and at most k edges of G have one endpoint in A and the second endpoint in B?

Minimum Maximal Matching

Input:
A graph G and an integer k.
Question:
Does there exist an inclusion-wise maximal matching in G that consists of at most k edges?

Multicolored Biclique

Input:
A bipartite graph G with bipartition classes $A \uplus B = V(G)$, an integer k, a partition of A into k sets $A_1, A_2, \ldots, A_k$, and a partition of B into k sets $B_1, B_2, \ldots, B_k$.
Question:
Does there exist a set $X \subseteq A \cup B$ that contains exactly one element of every set A_i and B_i, $1 \leq i \leq \ell$ and that induces a complete bipartite graph $K_{k,k}$ in G?

Multicolored Clique

Input:
A graph G, an integer k, and a partition of $V(G)$ into k sets $V_1, V_2, \ldots, V_k$.
Question:
Does there exist a set $X \subseteq V(G)$ that contains exactly one element of every set V_i and that is a clique in G?

Multicolored Independent Set

Input:
A graph G, an integer k, and a partition of $V(G)$ into k sets $V_1, V_2, \ldots, V_k$.
Question:
Does there exist a set $X \subseteq V(G)$ that contains exactly one element of every set V_i and that is an independent set in G?

Multicut

A synonym for Vertex Multicut.

NAE-SAT

Input:
A CNF formula φ.
Question:
Does there exist an assignment ψ such that for every clause C in φ, at least one literal of C is evaluated to true and at least one literal is evaluated to false by ψ?

Odd Cycle Transversal

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is bipartite?

Odd Set

Input:
A universe U, a set $\mathcal{F}$ of subsets of U, and an integer k.
Question:
Does there exist a nonempty set $X \subseteq U$ of size at most k such that $|A \cap X|$ is odd for every $A \in \mathcal{F}$?

Partial Dominating Set

Input:
A graph G and integers k and r.
Question:
Does there exist a set X of at most k vertices of G such that $|N_G[X]| \geq r$?

Partial Vertex Cover

Input:
A graph G and integers k and r.
Question:
Does there exist a set X of at most k vertices of G such that at least r edges of G are incident to at least one vertex of X?

Partitioned Clique

A synonym for Multicolored Clique.

Perfect d-Set Matching

Input:
A universe U, a family $\mathcal{A}$ of sets over U, where each set in $\mathcal{A}$ is of size exactly d, and an integer k.
Question:
Does there exist a family $\mathcal{A}' \subseteq \mathcal{A}$ of k pairwise disjoint sets such that $\bigcup \mathcal{A}' = U$?

Perfect Code

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that for every $u \in V(G)$ there exists exactly one vertex $v \in X$ for which $u \in N_G[v]$?

Permutation Composition

Input:
A family $\mathcal{P}$ of permutations of a universe U, additional permutation π of U, and an integer k.
Question:
Does there exist a sequence $\pi_1, \pi_2, \ldots, \pi_k \in \mathcal{P}$ such that $\pi = \pi_1 \circ \pi_2 \circ \ldots \circ \pi_k$?

Planar 3-Coloring

Input:
A planar graph G.
Question:
Does there exist a coloring $c : V(G) \to \{1, 2, 3\}$ such that $c(u) \neq c(v)$ for every $uv \in E(G)$?

Planar 3-SAT

Input:
A CNF formula φ, such that every clause of φ consists of at most three literals and the incidence graph of φ is planar. Here, an *incidence graph* of a formula φ is a bipartite graph with vertex sets consisting of all variables and clauses of φ where a variable is adjacent to all clauses it appears in.
Question:
Does there exist a satisfying assignment for φ?

Planar Vertex Deletion

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is planar?

Planar Deletion Compression

Input:
A graph G, a set $W \subseteq V(G)$ such that $G - W$ is planar, and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is planar?

Planar Diameter Improvement

Input:
A planar graph G and an integer d.
Question:
Does there exist a supergraph of G that is planar and has diameter at most d?

Planar Feedback Vertex Set

Input:
A planar graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is a forest?

Planar Hamiltonian Cycle

Input:
A planar graph G.
Question:
Does there exist a simple cycle C in G such that $V(C) = V(G)$?

Planar Longest Cycle

Input:
A planar graph G and an integer k.
Question:
Does there exist a cycle in G of length at least k?

Planar Longest Path

Input:
A planar graph G and an integer k.
Question:
Does there exist a path in G consisting of k vertices?

Planar Vertex Cover

Input:
A planar graph G, an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is edgeless?

Point Line Cover

Input:
A set P of points in the plane and an integer k.
Question:
Does there exist a family L of at most k lines on the plane such that every point in P lies on some line from L?

Polynomial Identity Testing

Input:
Two polynomials f and g over a field F, given as arithmetic circuits with addition, subtraction and multiplication gates.
Question:
Is it true that $f(x) = g(x)$ for every $x \in F$?

Pseudo Achromatic Number

Input:
A graph G and an integer k.
Question:
Does there exist a partition of $V(G)$ into k sets $V_1, V_2, \ldots, V_k$ such that for every $1 \leq i < j \leq k$ there exists at least one edge of G with one endpoint in V_i and the second endpoint in V_j?

Ramsey

Input:
A graph G and an integer k.
Question:
Does there exist a set X of exactly k vertices of G such that $G[X]$ is a clique or $G[X]$ is edgeless?

Red-Blue Dominating Set

Input:
A bipartite graph G with bipartition classes $R \uplus B = V(G)$ and an integer k.
Question:
Does there exist a set $X \subseteq R$ of size at most k such that $N_G(X) = B$?

Satellite Problem

Input:
A graph G, integers p and q, a vertex $v \in V(G)$, and a partition $V_0, V_1, \ldots, V_r$ of $V(G)$ such that $v \in V_0$ and there is no edge between V_i and V_j for any $1 \leq i < j \leq r$.
Question:
Does there exist a (p, q)-cluster C satisfying $V_0 \subseteq C$ such that for every $1 \leq i \leq r$, either $C \cap V_i = \emptyset$ or $V_i \subseteq C$? Here, a set $C \subseteq V(G)$ is a (p, q)-cluster if $|C| \leq p$ and $d(C) \leq q$.

Scattered Set

Input:
A graph G and integers r and k.
Question:
Does there exist a set X of at least k vertices of G such that $\mathrm{dist}_G(u, v) > r$ for every distinct $u, v \in X$?

Set Cover

Input:
A universe U, a family $\mathcal{F}$ over U, and an integer k.
Question:
Does there exist a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ of size at most k such that $\bigcup \mathcal{F}' = U$?

Set Packing

Input:
A universe U, a family $\mathcal{A}$ of sets over U, and an integer k.
Question:
Does there exist a family $\mathcal{A}' \subseteq \mathcal{A}$ of k pairwise disjoint sets?

Set Splitting

Input:
A universe U and a family $\mathcal{F}$ of sets over U.
Question:
Does there exist a set $X \subseteq U$ such that $A \cap X \neq \emptyset$ and $A \setminus X \neq \emptyset$ for every $A \in \mathcal{F}$?

Short Turing Machine Acceptance

Input:
A description of a nondeterministic Turing machine M, a string x, and an integer k.
Question:
Does there exist a computation path of M that accepts x on at most k steps?

Skew Edge Multicut

Input:
A directed graph G, a set of pairs $(s_i, t_i)_{i=1}^{\ell}$ of vertices of G, and an integer k.
Question:
Does there exist a set X of at most k edges of G such that for every $1 \leq i \leq j \leq \ell$, vertex t_j is not reachable from vertex s_i in the graph $G - X$?

Special Disjoint FVS

Input:
A graph G, a set $W \subseteq V(G)$ such that $G - W$ is edgeless and every vertex of $V(G) \setminus W$ is of degree at most three in G, and an integer k.
Question:
Does there exist a set $X \subseteq V(G) \setminus W$ of size at most k such that $G - X$ is a forest?

Split Edge Deletion

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k edges of G such that $G - X$ is a split graph? Here, a *split graph* is a graph whose vertex set can be partitioned into two parts, one inducing a clique and one inducing an independent set.

Split Vertex Deletion

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is a split graph? Here, a *split graph* is a graph whose vertex set can be partitioned into two parts, one being a clique and one being an independent set.

Steiner Tree

Input:
A graph G, a set $K \subseteq V(G)$, and an integer k.
Question:
Does there exist a connected subgraph of G that contains at most k edges and contains all vertices of K?

Strongly Connected Steiner Subgraph

Input:
A directed graph G, a set $K \subseteq V(G)$, and an integer k.
Question:
Does there exist a strongly connected subgraph of G that contains at most k edges and contains all vertices of K?

Subgraph Isomorphism

Input:
Two graphs G and H.
Question:
Does there exist a subgraph of G that is isomorphic to H?

Subset Sum

Input:
A set S of integers and integers k and m.
Question:
Does there exist a subset $X \subseteq S$ of size at most k whose elements sum up to m?

TSP

Input:
A graph G with edge weights $\mathbf{w} : E(G) \to \mathbb{R}_{>0}$.
Question:
Find a closed walk of minimum possible total weight that visits all vertices of G.

Total Dominating Set

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that for every $u \in V(G)$ there exists $v \in X$ with $uv \in E(G)$?

Tree Spanner

Input:
A graph G and an integer k.
Question:
Does there exist a spanning tree T of G such that for every $u, v \in V(G)$ it holds that $\mathrm{dist}_T(u,v) \leq k\mathrm{dist}_G(u,v)$?

Tree Subgraph Isomorphism

Input:
A graph G and a tree H.
Question:
Does there exist a subgraph of G that is isomorphic to H?

Treewidth

Input:
A graph G and an integer k.
Question:
Is the treewidth of G at most k?

Treewidth-η Modulator

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ has treewidth at most η?

Triangle Packing

Input:
A graph G and an integer k.
Question:
Does there exist in G a family of k pairwise vertex-disjoint triangles?

Unique Hitting Set

Input:
A universe U, a set $\mathcal{A}$ of subsets of U, and an integer k.
Question:
Does there exist a set $X \subseteq U$ of size at most k such that $|A \cap X| = 1$ for every $A \in \mathcal{A}$?

Unit Disk Independent Set

Input:
A set P of unit discs in the plane and an integer k.
Question:
Does there exist a set of k pairwise disjoint elements of P?

Unit Square Independent Set

Input:
A set P of axis-parallel unit squares in the plane and an integer k.
Question:
Does there exist a set of k pairwise disjoint elements of P?

Variable Deletion Almost 2-SAT

Input:
A CNF formula φ, where every clause of φ consists of at most two literals, and an integer k.
Question:
Does there exist a set X of at most k variables of φ such that $\varphi - X$ is satisfiable? Here, $\varphi - X$ denotes the formula φ with every clause containing at least one variable from X deleted.

Vertex k-Way Cut

Input:
A graph G and integer k and s.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ has at least s connected components?

Vertex Coloring

A synonym for Chromatic Number.

Vertex Cover

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is edgeless?

Vertex Cover Above Matching

Input:
A graph G, a matching M in G, and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is edgeless? Note that the question is the same as in Vertex Cover, but the Vertex Cover Above Matching problem is usually used in the context of above guarantee parameterization with the size of the given matching M as a lower bound.

Vertex Cover Above LP

Input:
A graph G and an integer k.
Question:
Does there exist a set X of at most k vertices of G such that $G - X$ is edgeless? Note that this is the same problem as Vertex Cover, but the name Vertex Cover Above LP is usually used in the context of above guarantee parameterization with an optimum solution to a linear programming relaxation as a lower bound.

Vertex Disjoint Paths

Input:
A graph G and k pairs of vertices $(s_i, t_i)_{i=1}^{k}$.
Question:
Do there exist k pairwise vertex-disjoint paths $P_1, P_2, \ldots, P_k$ such that P_i starts in s_i and ends in t_i?

Vertex Multicut

Input:
A graph G, a set of pairs $(s_i, t_i)_{i=1}^{\ell}$ of vertices of G, and an integer k.
Question:
Does there exist a set X of at most k vertices of G, not containing any vertex s_i or t_i, such that for every $1 \leq i \leq \ell$, vertices s_i and t_i lie in different connected components of $G - X$?

Vertex Multiway Cut

Input:
A graph G, a set $T \subseteq V(G)$, and an integer k.
Question:
Does there exist a set $X \subseteq V(G) \setminus T$ of size at most k such that every element of T lies in a different connected component of $G - X$?

Weighted Circuit Satisfiability

Input:
A Boolean circuit C and an integer k.
Question:
Does there exist an assignment to the input gates of C that satisfies C and that sets exactly k input gates to true?

Weighted Independent Set

Input:
A graph G with vertex weights $\mathbf{w} : V(G) \to \mathbb{R}_{\geq 0}$.
Question:
Find an independent set in G of the maximum possible total weight.

Weighted Longest Path

Input:
A graph G with vertex weights $\mathbf{w} : V(G) \to \mathbb{N}$ and an integer k.
Question:
Find a simple path in G on k vertices of the minimum possible total weight.

Index

M. Cygan et al., *Parameterized Algorithms*,

Author index

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3