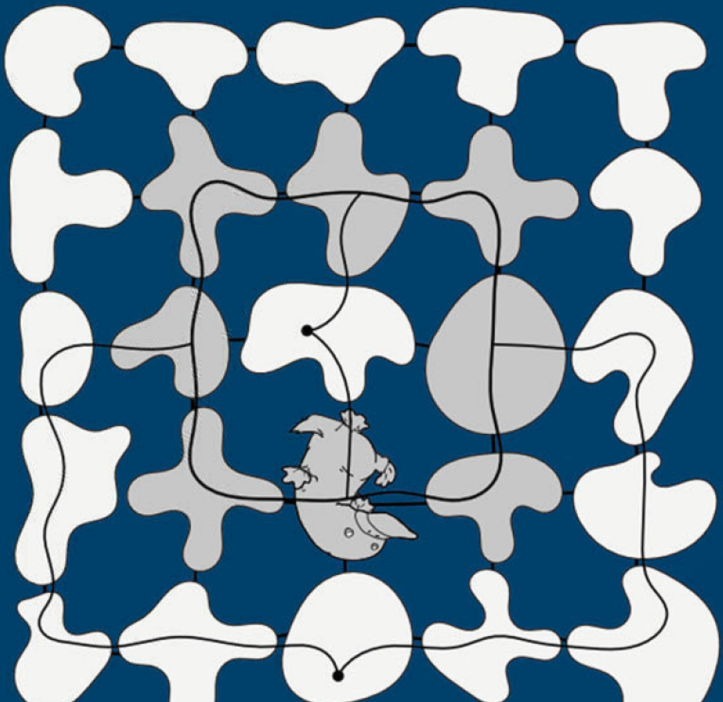
Marek Cygan · Fedor V. Fomin
Łukasz Kowalik · Daniel Lokshtanov
Dániel Marx · Marcin Pilipczuk
Michał Pilipczuk · Saket Saurabh

Parameterized Algorithms



Springer

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

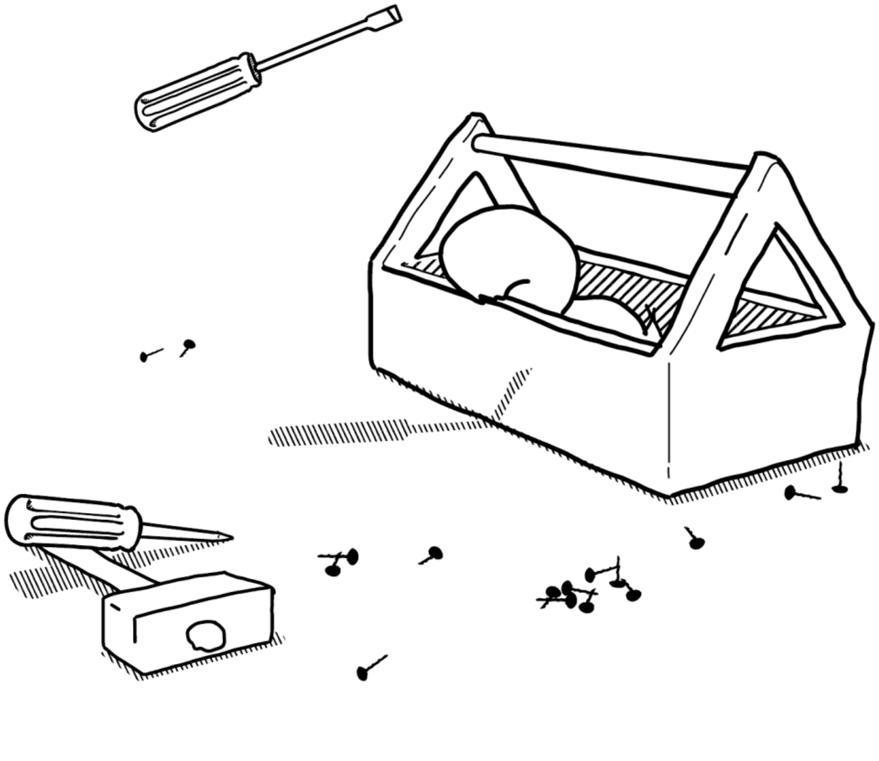
Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

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

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 $k = 2$.

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_1

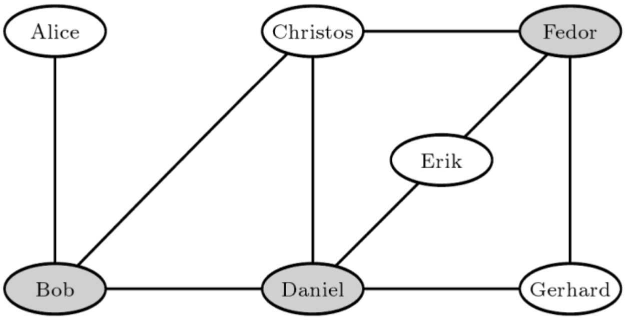


Fig. 1.1: An instance of the BAR FIGHT PREVENTION problem with a solution for $k = 3$. An edge between two guests means that they will fight if both are admitted

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

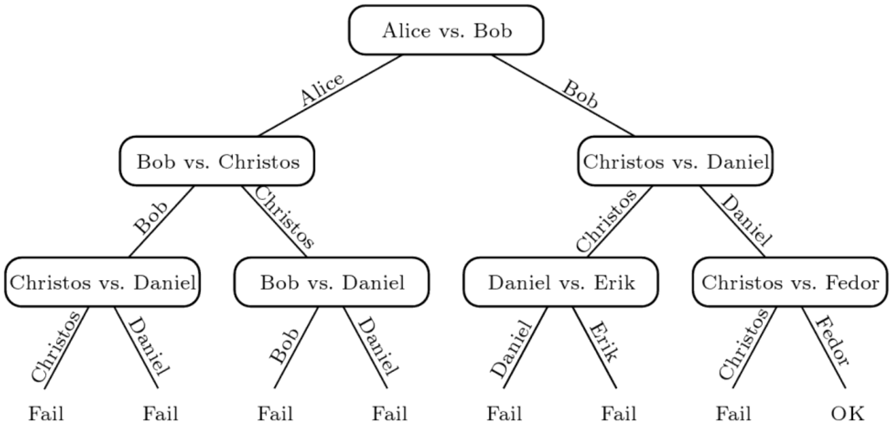


Fig. 1.2: The search tree for Bar Fight Prevention with $k = 3$. In the leaves marked with "Fail", the parameter k is decreased to zero, but there are still unresolved conflicts. The rightmost branch of the search tree finds a solution: after rejecting Bob, Daniel, and Fedor, no more conflicts remain

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

© Springer International Publishing Switzerland 2015
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 $\text{size}_{\mathcal{A}}\colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined as follows:

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

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 $\text{size}_{\mathcal{A}}(k) \leq g(k)$ 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. □

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 $E(G \circledast F) = (E(G) \cup \text{rev}(F)) \setminus F$. 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 $C \cap \text{rev}(F)$ 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 $f_i \in \text{rev}(F)$ (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

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. $\square$

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. $\square$

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.

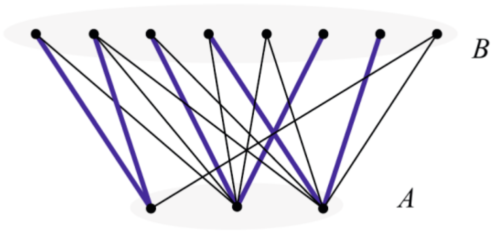


Fig. 2.2: Set A has a 2-expansion into B

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. □

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 $0 \leq y_v \leq 1$ 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 $y_u \geq x_u$ and $y_v \geq x_v$, so $y_u + y_v \geq x_u + x_v \geq 1$. 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

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

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

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

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 $d!k^d \cdot d^2$. ☐

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

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 $\mathrm{NP} \subseteq \mathrm{coNP}/\mathrm{poly}$ (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 $\mathrm{P} = \mathrm{NP}$.

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.

© Springer International Publishing Switzerland 2015
M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_3

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

$$\Big\{ h_i(S) \ : \ 1 \le i \le \ell \text{ and } S \text{ is a feasible solution of } I_i \Big\}$$

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 $\mu(I_i) \le \mu(I) - c$ 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 nonpositive 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

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

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 $(v_1, v_2, \ldots, v_n)$ 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 $V_{3k} = \{v_1, \ldots, v_{3k}\}$. 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):

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

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:

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