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

Parameterized Algorithms

May 30, 2016

Springer

Chapter 1
Introduction

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

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

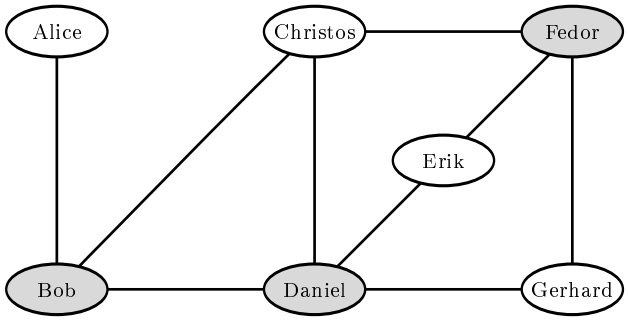


Fig. 1.1: An instance of the Bar Fight Prevention problem with a solution for $k = 3$. An edge between two guests means that they will fight if both are admitted

Efficient algorithms for Bar Fight Prevention

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

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

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

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

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

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

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

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

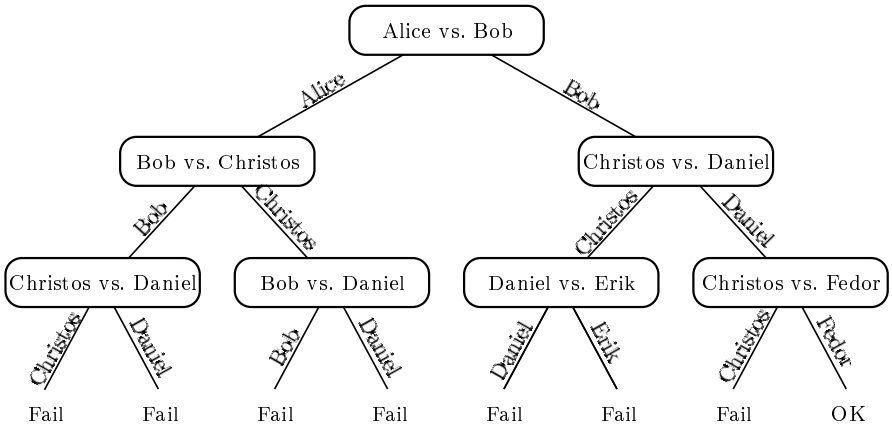


Fig. 1.2: The search tree for Bar Fight Prevention with $k = 3$. In the leaves marked with "Fail", the parameter k is decreased to zero, but there are still unresolved conflicts. The rightmost branch of the search tree finds a solution: after rejecting Bob, Daniel, and Fedor, no more conflicts remain

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

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

in k, but depends only polynomially (actually, linearly) on n. Our goal is to find algorithms of this form.

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

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

A negative example: vertex coloring

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

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

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

A hard parameterized problem: finding cliques

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

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

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:

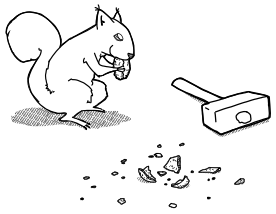
- optimize the *parametric dependence* of the running time, i.e., try to design an algorithm where function f grows as slowly as possible; or
- optimize the *polynomial factor* in the running time, i.e., try to design an algorithm where constant c is as small as possible.

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.

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

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

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

2.1 Formal definitions

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

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

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

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

Definition 2.1 (Kernelization, kernel). A *kernelization algorithm*, or simply a *kernel*, for a parameterized problem Q is an algorithm $\mathcal{A}$ that, given

an instance (I, k) of Q, works in polynomial time and returns an equivalent instance (I', k') of Q. Moreover, we require that $\text{size}_{\mathcal{A}}(k) \leq g(k)$ 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 of-

ten 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. □

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.

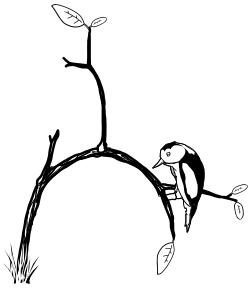
For tournaments, the deletion of edges results in directed graphs which are not tournaments anymore. Because of that, it is much more convenient to use the characterization of a feedback arc set in terms of "reversing edges". We start with the following well-known result about *topological orderings* 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 $\mathrm{rev}(F) = \{(u, v) \ : \ (v, u) \in F\}$, then for $G \circledast F$ the vertex set is $V(G)$

Chapter 3
Bounded search trees

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



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

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

1. Every feasible solution S of I_i, $i \in \{1, \ldots, \ell\}$, corresponds to a feasible solution $h_i(S)$ of I. Moreover, the set

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

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

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

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

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

We apply the idea of bounded search trees to Vertex Cover in Section 3.1. Section 3.2 briefly discusses methods of bounding the number of

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

3.1 Vertex Cover

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

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

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

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

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

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

The running time of the algorithm is bounded by

(the number of nodes in the search tree) $\times$ (time taken at each node).

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

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

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

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

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

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

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

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

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

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

since then we will have

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

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

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

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

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

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

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

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

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

3.2 How to solve recursive relations

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