Chapter 1
Introduction

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

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

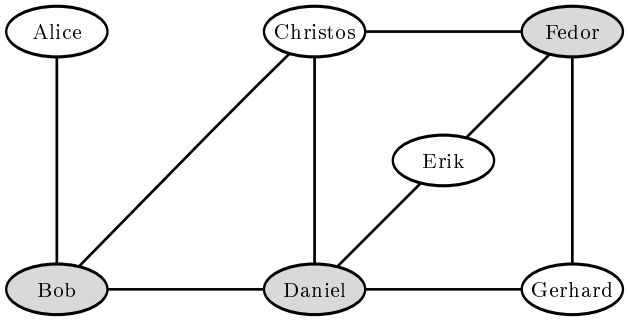

Fig. 1.1: An instance of the Bar Fight Prevention problem with a solution for $k = 3$. An edge between two guests means that they will fight if both are admitted

Efficient algorithms for Bar Fight Prevention

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

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

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

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

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

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

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

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

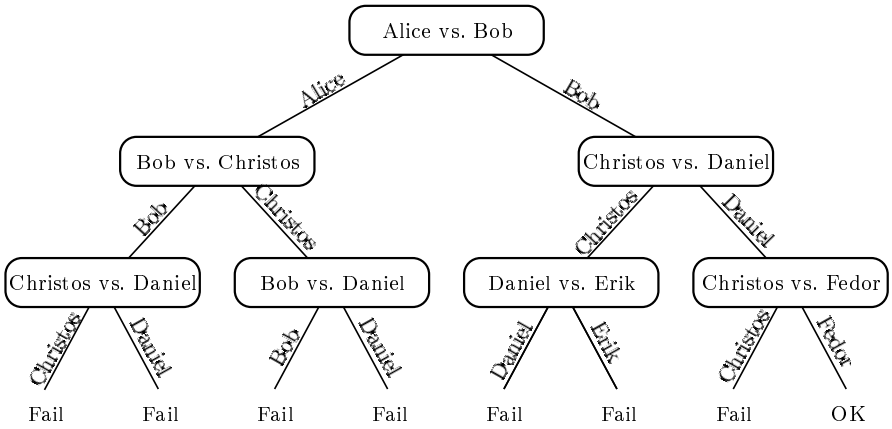

Fig. 1.2: The search tree for Bar Fight Prevention with $k = 3$. In the leaves marked with "Fail", the parameter k is decreased to zero, but there are still unresolved conflicts. The rightmost branch of the search tree finds a solution: after rejecting Bob, Daniel, and Fedor, no more conflicts remain

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

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

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

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

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

A negative example: vertex coloring

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

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

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

A hard parameterized problem: finding cliques

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

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

is, a set of k vertices with an edge between every pair of them. This problem is known as the Clique problem. For example, if we interpret now the edges of Fig. 1.1 as showing friendships between people, then Bob, Christos, and Daniel form a clique of size 3.

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

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

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

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

Fig. 1.3: Overview of the problems in this chapter

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

Finding cliques — with a different parameter

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

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

The art of parameterization

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

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

a problem involving a set of geometric objects (say, points in space, disks, or polygons), one may parameterize by the maximum number of vertices of each polygon or the dimension of the space where the problem is defined. For each problem, with a bit of creativity, one can come up with a large number of (combinations of) parameters worth studying.

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

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

1.1 Formal definitions

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

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

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

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

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

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

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

We now define the complexity class XP.

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

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

The definition of a parameterized problem, as well as the definitions of the classes FPT and XP, can easily be generalized to encompass multiple parameters. In this setting we simply allow k to be not just one nonnegative

integer, but a vector of d nonnegative integers, for some fixed constant d. Then the functions f and g in the definitions of the complexity classes FPT and XP can depend on all these parameters.

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

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

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

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

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

Bibliographic notes

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