1 Fixed-Parameter Tractability

In this chapter, we introduce parameterized problems and the notion of fixed-parameter tractability. We start with an informal discussion that highlights the main issues behind the definition of fixed-parameter tractability. In Sect. 1.2, we begin the formal treatment. In Sect. 1.3, we consider a larger example that introduces some of the most fundamental parameterized problems and the most basic technique for establishing fixed-parameter tractability, the method of bounded search trees. In Sect. 1.4 and Sect. 1.5 we exemplify how the parameterized approach may help to gain a better understanding of the complexity of fundamental algorithmic problems by considering applications in two different areas, approximation algorithms and automated verification. Finally, in Sect. 1.6, we give several equivalent characterizations of fixed-parameter tractability.

1.1 Introduction

Before we start our formal development of the theory, in this section we informally discuss a few motivating examples. All notions discussed informally in this introduction will be made precise later in this book.

The first example is the problem of evaluating a database query, which we have already mentioned in the preface. To be a bit more precise, let us say that we want to evaluate a *conjunctive query* φ in a *relational database* $\mathcal{D}$.[1] Conjunctive queries form the most fundamental class of database queries,

[1]If the reader has never heard of "conjunctive queries" or "relational databases" before, there is no need to worry. All that is required here is some vague idea about "database query" and "database." (For example, a database might store flights between airports, and a conjunctive query might ask if there is connection from Berlin to Beijing with two stopovers.) Of course, the query will be written in some formal *query language*, such as the language SQL, and thus is a well-defined mathematical object. Its *size* is simply the number of symbols it contains.

and many queries that occur in practice are conjunctive queries. Classical complexity quickly tells us that this problem is intractable; it is NP-complete.

As a second example, we consider a central algorithmic problem in automated verification, the problem of checking that a finite state system, for example, a circuit, has a certain property. The state space $\mathcal{S}$ of the system can be described by a so-called *Kripke structure*, which is nothing but a vertex-labeled directed graph. The property to be checked is typically specified as a formula φ in a temporal logic, for example, *linear temporal logic* LTL.[2] Then the problem is to decide whether the structure $\mathcal{S}$ satisfies the formula φ. This problem is known as the LTL *model-checking problem*. Again, classical complexity tells us that the problem is intractable; it is PSPACE-complete.

The two problems are fairly similar, and they both lend themselves naturally to a parameterized complexity analysis. As we explained in the introduction, in parameterized complexity theory the complexity of a problem is not only measured in terms of the input size, but also in terms of a parameter. The theory's focus is on situations where the parameter can be assumed to be small. The inputs of both the query evaluation problem and the model-checking problem consist of two parts, which typically have vastly different sizes. The database and the state space are usually very large, whereas the query and the LTL-formula tend to be fairly small. As parameters, we choose the size of the query and the size of the formula. In the following discussion, we denote the parameter by k and the input size by n. Note that the input size n will usually be dominated by the size of the database and the size of the state space, respectively.

It is easy to show that the query evaluation problem can be solved in time $O(n^k)$. Furthermore, the model-checking problem can be solved in time $O(k \cdot 2^{2k} \cdot n)$. The latter result requires more effort; we will reconsider it in Sect. 1.5 and again in Sect. 10.1. In both cases, the constants hidden in the $O(\,\cdot\,)$ notation ("big-Oh notation") are fairly small. At first sight, these results look very similar: Both running times are exponential, and both are polynomial for fixed k. However, there is an important difference between the two results: Let us assume that n is large and k is small, say, $k = 5$. Then an exponent k in the running time, as in $O(n^k)$, is prohibitive, whereas an exponential factor 2^{2k} as in $O(k \cdot 2^{2k} \cdot n)$ may be unpleasant, but is acceptable for a problem that, after all, is PSPACE-complete.[3] In the terminology of parameterized complexity theory, the LTL model-checking problem is *fixed-parameter tractable.*

Up to this point, except for the occasional use of the term "parameter," the discussion did not require any parameterized complexity theory. As a matter

[2]Again, there is no need to know about Kripke structures or temporal logic here. A vague intuition of "systems" and "specification languages" is sufficient.

[3]Actually, in practice the dominant factor in the $k \cdot 2^{2k} \cdot n$ running time of the LTL model-checking algorithm is not the exponential 2^{2k}, but the size n of the state space.

of fact, researchers in database theory and automated verification were well aware of the issues we discussed above before parameterized complexity was first introduced. (The LTL model-checking algorithm due to Lichtenstein and Pnueli was published in 1985.) But now we ask if the conjunctive query evaluation problem is also fixed-parameter tractable, that is, if it can be solved by an algorithm with a similar running time as the LTL model-checking algorithm, say, $2^{O(k)} \cdot n$ or $2^{p(k)} \cdot q(n)$ for some polynomials $p(X), q(X)$, or at least $2^{2^k} \cdot q(n)$. Classical complexity provides us with no means to support a negative answer to this question, and this is where the new theory of parameterized intractability is needed. To cut the story short, the conjunctive query evaluation problem can be shown to be complete for the parameterized complexity class W[1]. This result, which will be proved in Chap. 6, can be interpreted as strong evidence that the problem is not fixed-parameter tractable.

As a third example of a parameterized problem we consider the satisfiability problem for formulas of propositional logic. We parameterize this problem by the number of variables of the input formula. Again denoting the parameter by k and the size of the input by n, this problem can clearly be solved in time $O(2^k \cdot n)$ and hence is fixed-parameter tractable. However, this problem is of a different nature from the parameterized problems we have discussed so far, because here the parameter cannot be expected to be small in typical applications. If, in some specific application, we had to solve large instances of the satisfiability problem with few variables, then the fixed-parameter tractability would help, but such a scenario seems rather unlikely. The purpose of the parameterization of the satisfiability problem by the number of variables is to obtain a more precise measure for the “source of the (exponential) complexity” of the problem, which is not the size of the input formula, but the number of variables. The typical question asked for such parameterizations is not if the problem is fixed-parameter tractable, but if it can be solved by (exponential) algorithms better than the trivial brute-force algorithms. Specifically, we may ask if the satisfiability problem can be solved in time $2^{o(k)} \cdot n$. Here parameterized complexity theory is closely connected with exact exponential (worst-case) complexity analysis. We will give an introduction into this area in the last chapter of this book.

1.2 Parameterized Problems and Fixed-Parameter Tractability

We start by fixing some notation and terminology: The set of all integers is denoted by $\mathbb{Z}$, the set of nonnegative integers by $\mathbb{N}_0$, and the set of natural numbers (that is, positive integers) by $\mathbb{N}$. For integers n, m with $n \leq m$, we let $[n, m] := \{n, n+1, \dots, m\}$ and $[n] := [1, n]$. Unless mentioned explicitly otherwise, we encode integers in binary.

As is common in complexity theory, we describe decision problems as languages over finite alphabets Σ. To distinguish them from parameterized problems, we refer to sets $Q \subseteq \Sigma^*$ of strings over Σ as *classical problems.* We always assume Σ to be nonempty.

Definition 1.1. Let Σ be a finite alphabet.
(1) A *parameterization* of Σ^* is a mapping $\kappa : \Sigma^* \to \mathbb{N}$ that is polynomial time computable.
(2) A *parameterized problem* (over Σ) is a pair (Q, κ) consisting of a set $Q \subseteq \Sigma^*$ of strings over Σ and a parameterization κ of Σ^*. $\dashv$

Example 1.2. Let SAT denote the set of all satisfiable propositional formulas, where propositional formulas are encoded as strings over some finite alphabet Σ. Let $\kappa : \Sigma^* \to \mathbb{N}$ be the parameterization defined by

$$\kappa(x) := \begin{cases} \text{number of variables of } x, & \text{if } x \text{ is (the encoding of) a propositional} \\ & \text{formula (with at least one variable)}^4, \\ 1, & \text{otherwise,} \end{cases}$$

for $x \in \Sigma^*$. We denote the parameterized problem $(\textsc{Sat}, \kappa)$ by p-SAT. $\dashv$

If (Q, κ) is a parameterized problem over the alphabet Σ, then we call strings $x \in \Sigma^*$ *instances* of Q or (Q, κ) and the numbers $\kappa(x)$ the corresponding *parameters*. Usually, we represent a parameterized problem (Q, κ) in the form

Instance:	$x \in \Sigma^*$.
Parameter:	$\kappa(x)$.
Problem:	Decide whether $x \in Q$.

For example, the problem p-SAT would be represented as follows:

p-SAT

Instance:	A propositional formula α.
Parameter:	Number of variables of α.
Problem:	Decide whether α is satisfiable.

As in this case, the underlying alphabet will usually not be mentioned explicitly.

As a second example, we consider a parameterized version of the classical INDEPENDENT-SET problem. Recall that an *independent set* in a graph is a set of pairwise non-adjacent vertices. An instance of INDEPENDENT-SET consists of a graph $\mathcal{G}$ and a positive integer k; the problem is to decide if $\mathcal{G}$ has an independent set of k elements.

[4]Our notation concerning propositional logic will be explained in detail in Sect. 4.1. In particular, we will not admit Boolean constants in propositional formulas, so every formula has at least one variable.

Example 1.3. A natural parameterization κ of INDEPENDENT-SET is defined by $\kappa(\mathcal{G}, k) = k$. It yields the following parameterized problem:

p-INDEPENDENT-SET	
Instance:	A graph $\mathcal{G}$ and $k \in \mathbb{N}$.
Parameter:	k.
Problem:	Decide whether $\mathcal{G}$ has an independent set of cardinality k.

$\dashv$

Before we define fixed-parameter tractability, let us briefly comment on the technical condition that a parameterization be polynomial time computable. For almost all natural parameterizations, the condition will obviously be satisfied. In any case, we can always make the parameter an explicit part of the input: If $Q \in \Sigma^*$ and $K : \Sigma^* \to \mathbb{N}$ is a function, then we can consider the problem

$$Q' := \{(x, k) \mid x \in Q, \text{ and } k = K(x)\} \subseteq \Sigma^* \times \mathbb{N},$$

with the parameterization κ defined by $\kappa(x, k) := k$. Indeed, parameterized problems are often defined as subsets of $\Sigma^* \times \mathbb{N}$, with the parameter being the second component of the instance. A typical example is p-INDEPENDENT-SET.

Fixed-Parameter Tractability

Recall that the motivation for the notion of *fixed-parameter tractability* is that if the parameter is small then the dependence of the running time of an algorithm on the parameter is not so significant. A fine point of the notion is that it draws a line between running times such as $2^k \cdot n$ on one side and n^k on the other, where n denotes the size of the input and k the parameter.

The length of a string $x \in \Sigma^*$ is denoted by $|x|$.

Definition 1.4. Let Σ be a finite alphabet and $\kappa : \Sigma^* \to \mathbb{N}$ a parameterization.

(1) An algorithm $\mathbb{A}$ with input alphabet Σ is an *fpt-algorithm with respect to* κ if there is a computable function $f : \mathbb{N} \to \mathbb{N}$ and a polynomial $p \in \mathbb{N}_0[X]$ such that for every $x \in \Sigma^*$, the running time of $\mathbb{A}$ on input x is at most

$$f\big(\kappa(x)\big) \cdot p\big(|x|\big).$$

(2) A parameterized problem (Q, κ) is *fixed-parameter tractable* if there is an fpt-algorithm with respect to κ that decides Q.
FPT denotes the class of all fixed-parameter tractable problems. $\dashv$

If the parameterization is clear from the context, we do not explicitly mention it and just speak of fpt-algorithms. We often use a less explicit terminology when bounding the running time of an algorithm or the complexity of a problem. For example, we might say that an algorithm is an fpt-algorithm if its running time is $f(\kappa(x)) \cdot |x|^{O(1)}$ for some computable function f. Formally, $n^{O(1)}$ denotes the class of all polynomially bounded functions on the natural numbers. The reader not familiar with the $O(\,\cdot\,)$ ("big-Oh") and $o(\,\cdot\,)$ ("little-oh") notation will find its definition in the Appendix. Occasionally, we also use the corresponding $\Omega(\,\cdot\,)$ ("big-Omega") and $\omega(\,\cdot\,)$ ("little-omega") notation for the corresponding lower bounds and the $\Theta(\,\cdot\,)$ ("big-Theta") notation for simultaneous upper and lower bounds, which all are explained in the Appendix as well.

Example 1.5. The parameterized satisfiability problem p-SAT is fixed-parameter tractable. Indeed, the obvious brute-force search algorithm decides if a formula α of size m with k variables is satisfiable in time $O(2^k \cdot m)$. ⊣

Clearly, if $Q \in$ PTIME then $(Q, \kappa) \in$ FPT for every parameterization κ. Thus fixed-parameter tractability relaxes the classical notion of tractability, polynomial time decidability.

Another trivial way of generating fixed-parameter tractable problems is shown in the following example:

Example 1.6. Let Σ be a finite alphabet and $\kappa_{\text{size}} : \Sigma^* \to \mathbb{N}$ the parameterization defined by

$$\kappa_{\text{size}}(x) := \max\{1, |x|\}$$

for all $x \in \Sigma^*$. (Remember that parameterizations always take nonnegative values.) Then for every decidable set $Q \subseteq \Sigma^*$, the problem $(Q, \kappa_{\text{size}})$ is fixed-parameter tractable. ⊣

The example can be generalized to the following proposition. A function $f : \mathbb{N} \to \mathbb{N}$ is *nondecreasing* (*increasing*) if for all $m, n \in \mathbb{N}$ with $m < n$ we have $f(m) \leq f(n)$ ($f(m) < f(n)$, respectively). A function f is *unbounded* if for all $n \in \mathbb{N}$ there exists an $m \in \mathbb{N}$ such that $f(m) \geq n$.

Proposition 1.7. *Let $g : \mathbb{N} \to \mathbb{N}$ be a computable nondecreasing and unbounded function, Σ a finite alphabet, and $\kappa : \Sigma^* \to \mathbb{N}$ a parameterization such that $\kappa(x) \geq g(|x|)$ for all $x \in \Sigma^*$.*

Then for every decidable set $Q \subseteq \Sigma^*$, the problem (Q, κ) is fixed-parameter tractable.

Proof: Let $h : \mathbb{N} \to \mathbb{N}$ be defined by

$$h(n) := \begin{cases} \max\{m \in \mathbb{N} \mid g(m) \leq n\}, & \text{if } n \geq g(1), \\ 1, & \text{otherwise.} \end{cases}$$

Since g is nondecreasing and unbounded, h is well-defined, and since g is nondecreasing and computable, h is also computable. Observe that h is nondecreasing and that $h(g(n)) \geq n$ for all $n \in \mathbb{N}$. Thus for all $x \in \Sigma^*$ we have

$$h(\kappa(x)) \geq h(g(|x|)) \geq |x|.$$

Let $f : \mathbb{N} \to \mathbb{N}$ be a computable function such that $x \in Q$ is decidable in time $f(|x|)$. Without loss of generality we may assume that f is nondecreasing. Then $x \in Q$ is decidable in time $f(h(\kappa(x))$, and hence (Q, κ) is fixed-parameter tractable. □

Thus every parameterized problem where the parameter increases monotonically with the size of the input is fixed-parameter tractable. The following example illustrates the other extreme of a parameterization that does not grow at all:

Example 1.8. Let Σ be a finite alphabet and $\kappa_{\text{one}} : \Sigma^* \to \mathbb{N}$ the parameterization defined by

$$\kappa_{\text{one}}(x) := 1.$$

for all $x \in \Sigma^*$.

Then for every $Q \subseteq \Sigma^*$, the problem (Q, κ_{one}) is fixed-parameter tractable if and only if Q is polynomial time decidable. ⊣

The parameterizations κ_{size} and κ_{one} introduced in Examples 1.6 and 1.8 will be quite convenient later to construct "pathological" examples of parameterized problems with various properties.

Exercise 1.9. Prove that the condition that g is nondecreasing is necessary in Proposition 1.7. That is, construct a decidable problem Q and a parameterization κ such that $(Q, \kappa) \notin$ FPT and $\kappa(x) \geq g(|x|)$ for all x and some function g that is computable and unbounded, but not nondecreasing.
Hint: Let Q be a problem that only contains strings of even length and that is decidable, but not decidable in polynomial time. Let $\kappa(x) := \kappa_{\text{one}}(x)$ if $|x|$ is even and $\kappa(x) := |x|$ if $|x|$ is odd. ⊣

Parameterized complexity theory provides methods for proving problems to be fixed-parameter tractable, but also gives a framework for dealing with apparently intractable parameterized problems in a similar way as the theory of NP-completeness does in classical complexity theory.

A very simple criterion for fixed-parameter intractability is based on the observation that the slices of a fixed-parameter tractable problem are solvable in polynomial time:

Definition 1.10. Let (Q, κ) be a parameterized problem and $\ell \in \mathbb{N}$. The ℓth *slice* of (Q, κ) is the classical problem

$$(Q, \kappa)_\ell := \{x \in Q \mid \kappa(x) = \ell\}. \qquad ⊣$$

Proposition 1.11. *Let (Q, κ) be a parameterized problem and $\ell \in \mathbb{N}$. If (Q, κ) is fixed-parameter tractable, then $(Q, \kappa)_\ell \in$ PTIME.*

We leave the simple proof to the reader (recall that κ is computable in polynomial time).

Example 1.12. Recall that a graph $\mathcal{G} = (V, E)$ is *k-colorable*, where $k \in \mathbb{N}$, if there is a function $C : V \to [k]$ such that $C(v) \neq C(w)$ for all $\{v, w\} \in E$. We parameterize the colorability problem for graphs by the number of colors:

p-COLORABILITY	
Instance:	A graph $\mathcal{G}$ and $k \in \mathbb{N}$.
Parameter:	k.
Problem:	Decide whether $\mathcal{G}$ is k-colorable.

The third slice of this problem is the classical 3-colorability problem, which is well-known to be NP-complete. Hence, by the preceding proposition, p-COLORABILITY is not fixed-parameter tractable unless PTIME = NP. $\dashv$

Unfortunately, for most parameterized problems that are believed to be intractable there is no such easy reduction to the classical theory of NP-completeness. For example, it is widely believed that p-INDEPENDENT-SET is not fixed-parameter tractable, but all slices of the problem are decidable in polynomial time.

Some remarks concerning the definition of fixed-parameter tractability are in order. We allow an *arbitrary* computable function f to bound the dependence of the running time of an fpt-algorithm on the parameter. While indeed a running time such as

$$O(2^k \cdot n),$$

where k denotes the parameter and n the size of the instance, can be quite good for small values of k, often better than the polynomial $O(n^2)$, a running time of, say,

$$2^{2^{2^{2^{2^k}}}} \cdot n,$$

cannot be considered tractable even for $k = 1$. The liberal definition of fixed-parameter tractability is mainly justified by the following two arguments, which are similar to those usually brought forward to justify polynomial time as a model of classical tractability:

(1) FPT is a robust class that does not depend on any particular machine model, has nice closure properties, and has a mathematically feasible theory.
(2) "Natural" problems in FPT will have "low" parameter dependence.

While by and large, both of these arguments are valid, we will see later in this book that (2) has some important exceptions. Indeed, we will see in Chap. 10 that there are natural fixed-parameter tractable problems that can only be solved by fpt-algorithms with a nonelementary parameter dependence. In Chap. 15, we will investigate more restrictive notions of fixed-parameter tractability.

However, among the known fixed-parameter tractable problems, problems that require a larger than exponential parameter dependence are rare exceptions. Furthermore, much of the theory is concerned with proving intractability (more precisely, hardness results), and, of course, such results are even stronger for our liberal definition.

Let us also mention that Downey and Fellows' standard notion of fixed-parameter tractability does not even require the parameter dependence of an fpt-algorithm to be computable. However, the notion of fixed-parameter tractability adopted here (called *strongly uniform fixed-parameter tractability* in [83]) leads to a more robust theory, which for all natural problems is the same anyway.

We want to clarify a possible source of ambiguity in our notation. Often the instances of problems are taken from a certain class of instances, such as the class of planar graphs. Suppose we have a parameterized problem (Q, κ) over the alphabet Σ and then consider the restriction of Q to a class $I \subseteq \Sigma^*$ of instances. Formally, this restriction is the problem $(I \cap Q, \kappa)$. Informally, we usually introduce it as follows:

Instance:	$x \in I$.
Parameter:	$\kappa(x)$.
Problem:	Decide whether $x \in Q$.

Let us emphasize that this notation specifies the same problem as:

Instance:	$x \in \Sigma^*$.
Parameter:	$\kappa(x)$.
Problem:	Decide whether $x \in Q \cap I$.

Example 1.13. For every class A of propositional formulas, we consider the following restriction of the problem p-SAT:

p-SAT(A)

Instance:	$\alpha \in A$.
Parameter:	Number of variables in α.
Problem:	Decide whether α is satisfiable.

$\dashv$

We close this section with a technical remark that will frequently be used tacitly. Many parameterized problems, for instance, p-INDEPENDENT-SET, have the following form:

Instance:	$x \in \Sigma^*$ and $k \in \mathbb{N}$.
Parameter:	k.
Problem:	$\ldots$.

Note that the size of an instance of such a problem is of order $|x| + \log k$.[5] Nevertheless, a simple computation shows that the problem is fixed-parameter tractable if and only if for some computable function f it can be decided in time $f(k) \cdot |x|^{O(1)}$ (instead of $f(k) \cdot (|x| + \log k)^{O(1)}$).

Similarly, a problem of the form

Instance:	$x \in \Sigma^*$ and $y \in (\Sigma')^*$.
Parameter:	$|y|$.
Problem:	$\dots$.

is in FPT if and only if it can be solved in time $f(|y|) \cdot |x|^{O(1)}$.

1.3 Hitting Sets and the Method of Bounded Search Trees

Let us consider the following problem, which we may call the *panel problem*: We have to form a small panel of leading experts in some research area A that we do not know well. We only have a publication database for that area at our disposal.[6] Here are three ideas of approaching the problem:

(1) We could try to find a small panel of scientists such that every paper in the area A is coauthored by some scientist of the panel. Clearly, the members of such a panel must know the area very well.
(2) We could try to find a small panel such that everybody working in the area has a joint publication with at least one panel member. Then the panel members should have a good overview over the area (maybe not as good as in (1), but still good enough).
(3) If neither (1) nor (2) works out, we could try to form a panel of scientists working in the area such that no two of them have a joint publication. To guarantee a certain breadth the panel should have a reasonable size.

But how can we find such a panel for either of the three approaches, given only the publication database? As trained complexity theorists, we model the problems we need to solve by the well-known *hitting set*, *dominating set*, and *independent set* problems.

For approach (1), we consider the *collaboration hypergraph* of the publication database. A hypergraph is a pair $\mathcal{H} = (V, E)$ consisting of a set V of *vertices* and a set E of *hyperedges* (sometimes also called *edges*), each of which is a subset of V. Thus *graphs* are hypergraphs with (hyper)edges of cardinality two. A *hitting set* in a hypergraph $\mathcal{H} = (V, E)$ is a set $S \subseteq V$

[5] If we write $\log n$ where an integer is expected, we mean $\lceil \log n \rceil$.

[6] If the reader feels the need for further motivation, here are two suggestions: Think of being a publisher who wants to start a new book series in the area A and is looking for an editorial board, or think of being a university official who wants to evaluate the A department with the help of a panel of external experts.

of vertices that intersects each hyperedge (that is, $S \cap e \neq \emptyset$ for all $e \in E$). Hitting-Set is the problem of finding a hitting set of a given cardinality k in a given hypergraph $\mathcal{H}$. The vertices of the collaboration hypergraph are all authors appearing in the publication database, and the hyperedges are all sets of authors of publications in the database. Approach (1) to the panel problem amounts to solving Hitting-Set for the collaboration hypergraph and the desired panel size k.

For approaches (2) and (3), all the information we need is contained in the *collaboration graph.* The vertices of this graph are again all authors, and there is an edge between two authors if they have a joint publication.[7] Recall that a *dominating set* in a graph $\mathcal{G} = (V, E)$ is a set $S \subseteq V$ of vertices such that every vertex in $V \setminus S$ is adjacent to a vertex in S. Dominating-Set is the problem of finding a dominating set of a given cardinality k in a given graph $\mathcal{G}$. Approach (2) to the panel problem amounts to solving Dominating-Set for the collaboration graph and panel size k. Finally, approach (3) to the panel problem amounts to solving Independent-Set for the collaboration graph and panel size k.

Unfortunately, all three problems are NP-complete. At first sight, this suggests that unless the publication database is fairly small there is not much hope for solving the panel problem with any of the three approaches. However, we only have to solve the problem for a *small* panel size k. We parameterize the problems by k and consider the following parameterized problems:

p-Hitting-Set
Instance: A hypergraph $\mathcal{H}$ and $k \in \mathbb{N}$.
Parameter: k.
Problem: Decide whether $\mathcal{H}$ has a hitting set of k elements.

p-Dominating-Set
Instance: A graph $\mathcal{G}$ and $k \in \mathbb{N}$.
Parameter: k.
Problem: Decide whether $\mathcal{G}$ has a dominating set of k elements.

We have already defined p-Independent-Set in Example 1.3.

If we assume the size of the panel to be small, a good fpt-algorithm for any of the three problems would let us solve the panel problem with the corresponding idea. Unfortunately, we will see later in this book that most likely none of the three problems is fixed-parameter tractable.

A great strength of the parameterized approach to the design of algorithms is its flexibility. If the first, obvious parameterization of a problem has been

[7]In hypergraph-theoretical terms, the collaboration graph is the *primal graph* of the collaboration hypergraph.

classified as "intractable," there is no need to give up. We can always look for further, maybe "hidden," parameters. In our example, we notice that we can expect the hyperedges of the collaboration hypergraph, that is, the author sets of publications in our database, to be fairly small. This suggests the following more-refined parameterization of the hitting set problem (we denote the cardinality of a finite set M by $|M|$):

p-card-HITTING-SET	
Instance:	A hypergraph $\mathcal{H} = (V, E)$ and $k \in \mathbb{N}$.
Parameter:	$k + d$, where $d := \max\{|e| \mid e \in E\}$.
Problem:	Decide whether $\mathcal{H}$ has a hitting set of k elements.

What we actually would like to do here is parameterize the problem by two parameters, k and d. However, admitting several parameters would further complicate the theory, and to avoid this we can use the sum of all intended parameters as the only actual parameter. We do the same whenever we consider problems with several parameters. This is sufficient for all our purposes, and it keeps the theory feasible.

The *size* of a hypergraph $\mathcal{H} = (V, E)$ is the number

$$\|\mathcal{H}\| := |V| + \sum_{e \in E} |e|,$$

this roughly corresponds to the size of a reasonable representation of $\mathcal{H}$.[8]

Theorem 1.14. *p-card*-HITTING-SET *is fixed-parameter tractable. More precisely, there is an algorithm solving* HITTING-SET *in time*

$$O(d^k \cdot \|\mathcal{H}\|).$$

Proof: Without loss of generality, we always assume that $d \geq 2$. For hypergraphs with hyperedges of cardinality at most 1, the hitting set problem is easily solvable in linear time.

We apply a straightforward recursive algorithm. Let e be a hyperedge of the input hypergraph $\mathcal{H}$. We know that each hitting set of $\mathcal{H}$ contains at least one vertex in e. We branch on these vertices: For $v \in e$, let $\mathcal{H}_v$ be the hypergraph obtained from $\mathcal{H}$ by deleting v and all hyperedges that contain v. Then $\mathcal{H}$ has a k-element hitting set that contains v if and only if $\mathcal{H}_v$ has a $(k-1)$-element hitting set. Thus $\mathcal{H}$ has a k-element hitting set if and only if there is a $v \in e$ such that $\mathcal{H}_v$ has a $(k-1)$-element hitting set. A recursive algorithm based on this observation is displayed as Algorithm 1.1. The algorithm returns TRUE if the input hypergraph has a hitting set of cardinality k and FALSE otherwise.

[8]As our machine model underlying the analysis of concrete algorithms we use random access machines with a standard instruction set and the uniform cost measure (cf. the Appendix). The assumption underlying the definition of the size of a hypergraph is that each vertex can be stored in one or a constant number of memory cells. See p. 74 in Chap. 4 for a detailed discussion of the size of structures.

```
HS($\mathcal{H}, k$)
 // $\mathcal{H} = (V, E)$ hypergraph, $k \geq 0$
 1. if $|V| < k$ then return FALSE
 2. else if $E = \emptyset$ then return TRUE
 3. else if $k = 0$ then return FALSE
 4. else
 5.     choose $e \in E$
 6.     for all $v \in e$ do
 7.         $V_v \leftarrow V \setminus \{v\}$; $E_v \leftarrow \{e \in E \mid v \notin e\}$; $\mathcal{H}_v \leftarrow (V_v, E_v)$
 8.         if HS($\mathcal{H}_v, k-1$) then return TRUE
 9.     return FALSE
```

Algorithm 1.1. A recursive hitting set algorithm

The correctness of the algorithms follows from the discussion above. To analyze the running time, let $T(k, n, d)$ denote the maximum running time of HS($\mathcal{H}', k'$) for $\mathcal{H}' = (V', E')$ with $\|\mathcal{H}'\| \leq n$, $\max\{|e| \mid e \in E'\} \leq d$, and $k' \leq k$. We get the following recurrence:

$$T(0, n, d) = O(1) \tag{1.1}$$

$$T(k, n, d) = d \cdot T(k-1, n, d) + O(n) \tag{1.2}$$

(for $n, k \in \mathbb{N}$). Here the term $d \cdot T(k-1, n, d)$ accounts for the at most d recursive calls in line 8. The hypergraph $\mathcal{H}_v$ can easily be computed in time $O(n)$, and all other commands can be executed in constant time. Let $c \in \mathbb{N}$ be a constant such that the term $O(1)$ in (1.1) and the term $O(n)$ in (1.2) are bounded by $c \cdot n$. We claim that for all $d \geq 2$ and $k \geq 0$,

$$T(k, n, d) \leq (2d^k - 1) \cdot c \cdot n. \tag{1.3}$$

We prove this claim by induction on k. For $k = 0$, it is immediate by the definition of c. For $k > 0$, we have

$$\begin{aligned} T(k, n, d) &\leq d \cdot T(k-1, n, d) + c \cdot n \\ &\leq d \cdot (2d^{k-1} - 1) \cdot c \cdot n + c \cdot n \\ &= (2d^k - d + 1) \cdot c \cdot n \\ &\leq (2d^k - 1) \cdot c \cdot n. \end{aligned}$$

This proves (1.3) and hence the theorem. □

Exercise 1.15. Modify algorithm HS($\mathcal{H}, k$) in such a way that it returns a hitting set of $\mathcal{H}$ of size at most k if such a hitting set exists and NIL otherwise and that the running time remains $O(d^k \cdot \|\mathcal{H}\|)$. ⊣

The recursive algorithm described in the proof exhaustively searches for a hitting set of size at most k. Of course, instead of recursive, such a search can also be implemented by explicitly building a search tree (which then corresponds to the recursion tree of the recursive version). A nonrecursive implementation that traverses the tree in a breadth-first manner rather than depth-first (as the recursive algorithm) may be preferable if we are interested in a hitting set of minimum cardinality (a *minimum hitting set* for short). The important fact is that the search tree is at most d-ary, that is, every node has at most d children, and its height is at most k. (The *height* of a tree is the number of edges on the longest path from the root to a leaf.) Thus the size of the tree is bounded in terms of the parameter $k + d$. This is why the method underlying the algorithm is often called the *method of bounded search trees*.

The following example illustrates the construction of the search tree:

Example 1.16. Let

$$\mathcal{H} := \big(\{a, b, c, d, e, f, g, h\}, \{e_1, \dots, e_5\}\big),$$

where $e_1 := \{a, b, c\}$, $e_2 := \{b, c, d\}$, $e_3 := \{c, e, f\}$, $e_4 := \{d, f\}$, and $e_5 := \{d, g\}$. The hypergraph is displayed in Fig. 1.2.

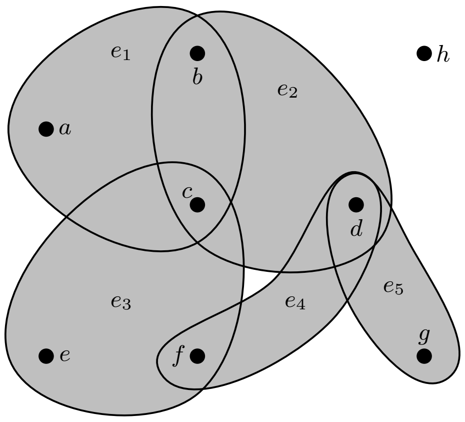

Fig. 1.2. The hypergraph of Example 1.16

The search tree underlying the execution of our algorithm on input $(\mathcal{H}, 3)$ is displayed in Fig. 1.3. We assume that on all branches of the tree the edges are processed in the same order $e_1, \dots, e_5$. Each inner node of the tree is labeled by the hyperedge e processed at that node, and each edge of the tree is labeled by the vertex $v \in e$ that determines the next recursive call. We leave it to the reader to compute the subhypergraphs $\mathcal{H}_v$ for which the recursive calls are made. The color of a leaf indicates the return value: It is TRUE for black and gray leaves and FALSE for white leaves. Each black or gray leaf corresponds to a hitting set of $\mathcal{H}$, which consists of the vertices labeling the edges on the path from the root to the leaf. A leaf is black if this hitting set is minimal with respect to set inclusion. $\dashv$

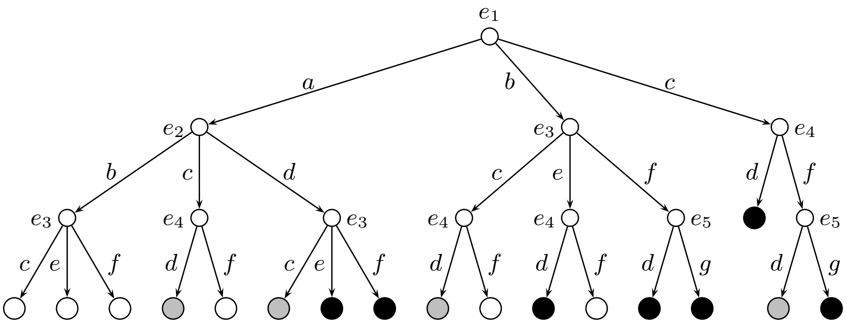

Fig. 1.3. A search tree

For later reference, we note that a slight modification of our hitting set algorithm yields the following lemma.

Lemma 1.17. *There is an algorithm that, given a hypergraph $\mathcal{H}$ and a natural number k, computes a list of all minimal (with respect to set inclusion) hitting sets of $\mathcal{H}$ of at most k elements in time*

$$O(d^k \cdot k \cdot \|\mathcal{H}\|),$$

where d is the maximum hyperedge cardinality. The list contains at most d^k sets.

Proof: Consider the algorithm EnumerateHS displayed as Algorithm 1.4.

Claim 1. Let $\mathcal{H} = (V, E)$ be a hypergraph and $k \in \mathbb{N}_0$. Then

$$\text{EnumerateHS}(\mathcal{H}, k, \emptyset)$$

returns a set $\mathcal{S}$ of hitting sets of $\mathcal{H}$ such that each minimal hitting set of $\mathcal{H}$ of cardinality at most k appears in $\mathcal{S}$.

Proof: By induction on the number $|E|$ of hyperedges of $\mathcal{H}$ we prove the slightly stronger statement that for all sets X disjoint from V,

$$\text{EnumerateHS}(\mathcal{H}, k, X)$$

returns a set $\mathcal{S}$ of sets such that:

```
EnumerateHS($\mathcal{H}, k, X$)
 // $\mathcal{H} = (V, E)$ hypergraph, $k \geq 0$, set $X$ of vertices (not of $\mathcal{H}$)
 1. if $E = \emptyset$ then return $\{X\}$
 2. else if $k = 0$ then return $\emptyset$
 3. else
 4.     choose $e \in E$
 5.     $\mathcal{S} \leftarrow \emptyset$
 6.     for all $v \in e$ do
 7.         $V_v \leftarrow V \setminus \{v\}$; $E_v \leftarrow \{e \in E \mid v \notin e\}$; $\mathcal{H}_v \leftarrow (V_v, E_v)$
 8.         $\mathcal{S} \leftarrow \mathcal{S} \cup$ EnumerateHS($\mathcal{H}_v, k-1, X \cup \{v\}$)
 9.     return $\mathcal{S}$
```

Algorithm 1.4. An algorithm enumerating hitting sets

(i) For all $S \in \mathcal{S}$ we have $X \subseteq S$, and the set $S \setminus X$ is a hitting set of $\mathcal{H}$ of cardinality at most k.

(ii) For each minimal hitting set S of $\mathcal{H}$ of cardinality at most k the set $S \cup X$ is contained in $\mathcal{S}$.

This is obvious for $|E| = 0$, because the only minimal hitting set of a hypergraph with no hyperedges is the empty set.

So suppose that $|E| > 0$ and that the statement is proved for all hypergraphs with fewer hyperedges. Let $\mathcal{S}$ be the set returned by the algorithm. To prove (i), let $S \in \mathcal{S}$. Let e be the edge chosen in line 4 and $v \in e$ such that S enters the set $\mathcal{S}$ in the corresponding execution of line 8, that is, $S \in \mathcal{S}_v$, where $\mathcal{S}_v$ is the set returned by EnumerateHS($\mathcal{H}_v, k-1, X \cup \{v\}$). By the induction hypothesis, $X \cup \{v\} \subseteq S$ and $S \setminus (X \cup \{v\})$ is a hitting set of $\mathcal{H}_v$. Then $e' \cap (S \setminus X) \neq \emptyset$ for all edges $e' \in E_v$ and also for all edges $e' \in E \setminus E_v$, because $v \in S$ and $v \notin X$.

To prove (ii), let S be a minimal hitting set of $\mathcal{H}$ of cardinality at most k. Note that the existence of such a set implies $k > 0$ because $E \neq \emptyset$. Let e be the hyperedge chosen by the algorithm in line 4 and $v \in S \cap e$, and let $\mathcal{S}_v$ be the set returned by the recursive call EnumerateHS($\mathcal{H}_v, k-1, X \cup \{v\}$) in line 8.

$S \setminus \{v\}$ is a minimal hitting set of the hypergraph $\mathcal{H}_v$. Hence by the induction hypothesis,

$$S \cup X = (S \setminus \{v\}) \cup (X \cup \{v\}) \in \mathcal{S}_v \subseteq \mathcal{S}.$$

This proves (ii) and hence the claim. $\dashv$

The analysis of the algorithm is completely analogous to the analysis of the algorithm HS in the proof of Theorem 1.14, only the constant c changes. Hence the running time is $O(d^k \cdot \|\mathcal{H}\|)$.

As the search tree traversed by the algorithm is a d-ary tree of height k, it has at most d^k leaves, which implies that the set $\mathcal{S}$ of hitting sets returned by the algorithm has cardinality at most d^k. But not all hitting sets in $\mathcal{S}$ are necessarily minimal; we only know that all minimal hitting sets of cardinality at most k appear in $\mathcal{S}$. However, we can easily test in time $O(k \cdot n)$ if a hitting set of $\mathcal{H}$ of cardinality k is minimal. Thus we can extract a list of all minimal hitting sets from $\mathcal{S}$ in time $O(|\mathcal{S}| \cdot k \cdot n) = O(d^k \cdot k \cdot n)$. □

Before we give further applications of the method of bounded search trees, let us briefly return to our panel problem. The reader may object that the algorithm of Theorem 1.14, though an fpt-algorithm, is still not very efficient. After all, there may be publications with ten authors or more, and if the panel is supposed to have 10 members, this yields an unpleasantly large constant factor of 10^{10}. Note, however, that a simple heuristic optimization will improve the algorithm considerably if most hyperedges of the input hypergraph are small and only a few are a bit larger: We first sort the hyperedges of the hypergraph by cardinality and then process them in this order. Then chances are that the algorithm stops before it has to branch on the large hyperedges, and even if it has to branch on them this will only happen close to the leaves of the tree. In particular, in our collaboration hypergraph, probably only few papers will have many authors.

For approach (1) to the panel problem, we have thus constructed a reasonable algorithm. How about (2) and (3), that is, p-DOMINATING-SET and p-INDEPENDENT-SET on the collaboration graph? Note that the fact that the maximum hyperedge cardinality d can be expected to be small has no impact on the collaboration graph. To see this, observe that for every collaboration graph $\mathcal{G}$ there is a matching collaboration hypergraph with maximum edge cardinality 2: the graph $\mathcal{G}$ itself viewed as a hypergraph. As we shall see below, an important parameter for the DOMINATING-SET and INDEPENDENT-SET problems is the *degree* of the input graph. Unfortunately, we cannot expect the degree of the collaboration graph, that is, the maximum number of coauthors of an author in the publication database, to be small.

An immediate consequence of Theorem 1.14 is that the hitting set problem restricted to input hypergraphs of bounded hyperedge cardinality is fixed-parameter tractable:

Corollary 1.18. *For every $d \geq 1$, the following problem is fixed-parameter tractable:*

p-d-HITTING-SET	
Instance:	A hypergraph $\mathcal{H} = (V, E)$ with $\max\{|e| \mid e \in E\} \leq d$ and $k \in \mathbb{N}$.
Parameter:	k.
Problem:	Decide whether $\mathcal{H}$ has a hitting set of k elements.

A *vertex cover* of a graph $\mathcal{G} = (V, E)$ is a set $S \subseteq V$ of vertices such that for all edges $\{v, w\} \in E$ either $v \in S$ or $w \in S$. The *parameterized vertex cover problem* is defined as follows:

p-Vertex-Cover	
Instance:	A graph $\mathcal{G}$ and $k \in \mathbb{N}$.
Parameter:	k.
Problem:	Decide whether $\mathcal{G}$ has a vertex cover of k elements.

As graphs are hypergraphs of hyperedge cardinality 2, Theorem 1.14 yields:

Corollary 1.19. *p*-Vertex-Cover *is fixed-parameter tractable. More precisely, there is an algorithm solving p*-Vertex-Cover *in time* $O(2^k \cdot \|\mathcal{G}\|)$.

The *degree* $\deg(v)$ of a vertex v in a graph $\mathcal{G} = (V, E)$ is the number of edges incident with v. The *(maximum) degree* of $\mathcal{G}$ is $\deg(\mathcal{G}) := \max\{\deg(v) \mid v \in V\}$. Consider the following refined parameterization of Dominating-Set:

p-deg-Dominating-Set	
Instance:	A graph $\mathcal{G} = (V, E)$ and $k \in \mathbb{N}$.
Parameter:	$k + \deg(\mathcal{G})$.
Problem:	Decide whether $\mathcal{G}$ has a dominating set of k elements.

To prove the following corollary of Theorem 1.14, observe that the dominating sets of a graph $\mathcal{G} = (V, E)$ are precisely the hitting sets of the hypergraph

$$\big(V, \{e_v \mid v \in V\}\big), \quad \text{where } e_v := \{v\} \cup \big\{w \in V \bigm| \{v, w\} \in E\big\} \text{ for } v \in V.$$

Corollary 1.20. *p-deg*-Dominating-Set *is fixed-parameter tractable. More precisely, there is an algorithm solving* Dominating-Set *in time*

$$O((d + 1)^k \cdot \|\mathcal{G}\|).$$

A different application of the method of bounded search trees shows that the refined parameterization of Independent-Set by the degree is also fixed-parameter tractable.

p-deg-Independent-Set	
Instance:	A graph $\mathcal{G} = (V, E)$ and $k \in \mathbb{N}$.
Parameter:	$k + \deg(\mathcal{G})$.
Problem:	Decide whether $\mathcal{G}$ has an independent set of k elements.

Exercise 1.21. Prove that *p-deg*-Independent-Set is fixed-parameter tractable.
Hint: Construct the first k levels of a search tree that, if completed, describes all *maximal* independent sets of the input graph. $\dashv$

Exercise 1.22. A *cover* for a hypergraph $\mathcal{H} = (V, E)$ is a set $X \subseteq V$ such that $|e \setminus X| \leq 1$ for all $e \in E$. Note that if $|e| \geq 2$ for all $e \in E$ then every cover is a hitting set. Show that there is an algorithm that, given a hypergraph $\mathcal{H}$ and a natural number k, computes a list of all minimal covers of $\mathcal{H}$ of cardinality at most k in time $O((k+1)^k \cdot k \cdot \|\mathcal{H}\|)$. $\dashv$

Exercise 1.23. Let

$$A = (a_{ij})_{\substack{i \in [m] \\ j \in [n]}} \in \{0,1\}^{m \times n}$$

be an $m \times n$ matrix with 0–1-entries. A *dominating set* for A is a set $S \subseteq [m] \times [n]$ such that

- $a_{ij} = 1$ for all $(i,j) \in S$,
- if $a_{ij} = 1$ for some $i \in [m], j \in [n]$, then there is an $i' \in [m]$ such that $(i', j) \in S$, or there is a $j' \in [n]$ such that $(i, j') \in S$.

That is, S is a set of 1-entries of the matrix such that each 1-entry is either in the same row or in the same column as an element of S.

Prove that the following problem is fixed-parameter tractable:

p-Matrix-Dominating-Set	
Instance:	A matrix $A \in \{0,1\}^{m \times n}$ and $k \in \mathbb{N}$.
Parameter:	k.
Problem:	Decide whether A has a dominating set of cardinality k.

Hint: View A as the adjacency matrix of a bipartite graph. A dominating set for the matrix corresponds to a set S of edges of the graph such that each edge has an endpoint with an edge in S in common. Show that if the matrix has a dominating set of cardinality k, then the graph has only few vertices of degree larger than k. Build a bounded search tree whose leaves describe minimal sets of edges that cover all edges except those that have an endpoint of degree larger than k. Try to extend the covers at the leaves to covers of all edges. $\dashv$

1.4 Approximability and Fixed-Parameter Tractability

In this section, we will show how the point of view of parameterized complexity may provide a better understanding of certain complexity issues in the theory of approximation algorithms for combinatorial optimization problems. First, we recall the definition of optimization problems. A binary relation $R \subseteq \Sigma_1^* \times \Sigma_2^*$, for alphabets Σ_1, Σ_2, is *polynomially balanced* if there is a polynomial $p \in \mathbb{N}_0[X]$ such that for all $(x, y) \in R$ we have $|y| \leq p(|x|)$.

Definition 1.24. Let Σ be a finite alphabet. An NP-*optimization problem* (over Σ) is a triple $O = (\mathrm{sol}, \mathrm{cost}, \mathrm{goal})$, where

(1) sol is a function defined on Σ^* such that the relation

$$\{(x, y) \mid x \in \Sigma^* \text{ and } y \in \mathrm{sol}(x)\}$$

is polynomially balanced and decidable in polynomial time. For every instance $x \in \Sigma^*$, we call the elements of the set $\mathrm{sol}(x)$ *solutions* for x.

(2) cost is a polynomial time computable function defined on $\{(x, y) \mid x \in \Sigma^*, \text{ and } y \in \mathrm{sol}(x)\}$; the values of cost are positive natural numbers.

(3) $\mathrm{goal} \in \{\max, \min\}$.

If $\mathrm{goal} = \max$ ($\mathrm{goal} = \min$) we speak of a *maximization* (*minimization*) problem. The function opt_O on Σ^* is defined by

$$\mathrm{opt}_O(x) := \mathrm{goal}\{\mathrm{cost}(x, y) \mid y \in \mathrm{sol}(x)\}.$$

A solution $y \in \mathrm{sol}(x)$ for an instance $x \in \Sigma^*$ is *optimal* if $\mathrm{cost}(x, y) = \mathrm{opt}_O(x)$. The objective of an optimization problem O is to find an optimal solution for a given instance. ⊣

Let us remark that for many problems O, finding an optimal solution for a given instance x is polynomial time equivalent to computing the cost $\mathrm{opt}_O(x)$ of an optimal solution. This, in turn, is often equivalent to deciding for a given k if $\mathrm{opt}_O(x) \geq k$ for $\mathrm{goal} = \max$ or $\mathrm{opt}_O(x) \leq k$ for $\mathrm{goal} = \min$.

Example 1.25. Recall that a *complete graph* is a graph in which all vertices are pairwise adjacent. A *clique* in a graph is the vertex set of a complete subgraph. The decision problem CLIQUE asks whether a given graph $\mathcal{G}$ has a clique of given cardinality k. It is derived from the maximization problem MAX-CLIQUE, which asks for a clique of maximum cardinality.

We usually use an informal notation for introducing optimization problems that is similar to our notation for parameterized problems. For example, for the maximum clique problem we write:

MAX-CLIQUE
Instance: A graph $\mathcal{G} = (V, E)$.
Solutions: Nonempty cliques $S \subseteq V$ of $\mathcal{G}$.
Cost: $|S|$.
Goal: max.

(We only admit nonempty cliques as solutions because we require costs to be positive integers.) ⊣

Example 1.26. The *minimum vertex cover problem* is defined as follows:

MIN-VERTEX-COVER
Instance: A graph $\mathcal{G} = (V, E)$.
Solutions: Nonempty vertex covers $S \subseteq V$ of $\mathcal{G}$.
Cost: $|S|$.
Goal: min.

⊣

There is a canonical way to associate a parameterized problem with each optimization problem:

Definition 1.27. Let $O = (\mathrm{sol}, \mathrm{cost}, \mathrm{opt})$ be an NP-optimization problem over the alphabet Σ. The *standard parameterization* of O is the following parameterized problem:

p-O	
Instance:	$x \in \Sigma^*$ and $k \in \mathbb{N}$.
Parameter:	k.
Problem:	Decide whether $\mathrm{opt}_O(x) \geq k$ if goal $=$ max or $\mathrm{opt}_O(x) \leq k$ if goal $=$ min.

$\dashv$

Example 1.28. The standard parameterization of MIN-VERTEX-COVER is the following problem:

p-MIN-VERTEX-COVER	
Instance:	A graph $\mathcal{G}$ and $k \in \mathbb{N}$.
Parameter:	k.
Problem:	Decide whether $\mathcal{G}$ has a vertex cover of at most k elements.

Observe that p-MIN-VERTEX-COVER is almost exactly the same problem as p-VERTEX-COVER introduced in the previous section, because a graph with at least k vertices has a vertex cover of exactly k elements if and only if it has a vertex cover of at most k elements. This is because vertex covers are *monotone* in the sense that supersets of vertex covers are also vertex covers. The two problems only differ on instances $(\mathcal{G}, k)$, where $\mathcal{G}$ has less than k vertices. For both problems, such instances are trivial, though in different directions.

Similarly, p-HITTING-SET and p-DOMINATING-SET are essentially the standard parameterizations of the minimization problems MIN-HITTING-SET and MIN-DOMINATING-SET. For maximization problems, instead of monotonicity we need antimonotonicity, which means that subsets of solutions are still solutions. p-INDEPENDENT-SET is essentially the standard parameterization of the antimonotone maximization problem MAX-INDEPENDENT-SET, and the following problem p-CLIQUE is essentially the standard parameterization of MAX-CLIQUE:

p-CLIQUE	
Instance:	A graph $\mathcal{G}$ and $k \in \mathbb{N}$.
Parameter:	k.
Problem:	Decide whether $\mathcal{G}$ has a clique of k elements.

$\dashv$

Note that for minimization problems that are not monotone and maximization problems that are not antimonotone, the standard parameterization is not the same as the parameterized problem that asks for solutions of cardinality exactly k. As a matter of fact, the two problems may have very different complexities, as the following exercise shows:

Exercise 1.29. A propositional formula is in *3-disjunctive normal form* (*3-DNF*) if it is of the form $\bigvee_{i \in I}(\lambda_{i1} \wedge \lambda_{i2} \wedge \lambda_{i3})$, where the λ_{ij} are literals. The conjunctions $(\lambda_{i1} \wedge \lambda_{i2} \wedge \lambda_{i3})$ are called the *terms* of the formula. Consider the following maximization problem:

MAX-3-DNF-SAT	
Instance:	A propositional formula α in 3-DNF.
Solutions:	Assignments to the variables of α.
Cost:	1 + number of terms satisfied.
Goal:	max.

(a) Prove that the standard parameterization of MAX-3-DNF-SAT is fixed-parameter tractable.
Hint: Prove that the expected number of terms satisfied by a random assignment is at least $(1/8)m$, where m is the total number of terms of the input formula. Conclude that for $m \geq 8k$, the formula has an assignment that satisfies at least k terms.

(b) Show that unless PTIME = NP, the following parameterized problem associated with MAX-3-DNF-SAT is not fixed-parameter tractable:

Instance:	A propositional formula α in 3-DNF and $k \in \mathbb{N}$.
Parameter:	k.
Problem:	Decide whether there is an assignment that satisfies exactly $k-1$ terms of α.

Hint: Prove that it is NP-complete to decide if a given formula in 3-disjunctive normal form has an assignment that satisfies no term. $\dashv$

Let us now turn to approximability. Let $O = (\mathrm{sol}, \mathrm{cost}, \mathrm{goal})$ be an optimization problem over Σ. For any instance x of O and for any $y \in \mathrm{sol}(x)$, the *approximation ratio* $r(x, y)$ of y with respect to x is defined as

$$r(x,y) := \max\left\{\frac{\mathrm{opt}_O(x)}{\mathrm{cost}(x,y)}, \frac{\mathrm{cost}(x,y)}{\mathrm{opt}_O(x)}\right\}.$$

For example, for minimization problems, we have $\mathrm{cost}(x,y) = r(x,y) \cdot \mathrm{opt}_O(x)$. The approximation ratio is always a number ≥ 1; the better a solution is, the closer the ratio is to 1.

Definition 1.30. Let $O = (\mathrm{sol}, \mathrm{cost}, \mathrm{goal})$ be an NP-optimization problem over the alphabet Σ.

(1) Let $\epsilon > 0$ be a real number. A *polynomial time ϵ-approximation algorithm* for O is a polynomial time algorithm that, given an instance $x \in \Sigma^*$, computes a solution $y \in \mathrm{sol}(x)$ such that $r(x,y) \leq (1+\epsilon)$.
Problem O is *constant approximable* if, for some $\epsilon > 0$, there exists a polynomial time ϵ-approximation algorithm for O.
(2) A *polynomial time approximation scheme (ptas)* for O is an algorithm $\mathbb{A}$ that takes as input pairs $(x,k) \in \Sigma^* \times \mathbb{N}$ such that for every fixed k, the algorithm is a polynomial time $(1/k)$-approximation algorithm. ⊣

Most of the known polynomial time approximation schemes have a running time of $n^{\Omega(k)}$, which means that for reasonably close approximations the running times quickly get infeasible on large instances. A ptas $\mathbb{A}$ is a *fully polynomial time approximation scheme (fptas)* if its running time is polynomial in $|x| + k$. Unfortunately, only few known optimization problems have an fptas. However, if we do not need a very precise approximation and can live with an error $1/k$, say, of 10%, we are in the situation where we have a small problem parameter k. We parameterize our approximation schemes by this error parameter and obtain the following intermediate notion:

Definition 1.31. Let $O = (\mathrm{sol}, \mathrm{cost}, \mathrm{goal})$ be an NP-optimization problem over the alphabet Σ. An *efficient polynomial time approximation scheme (eptas)* for O is a ptas $\mathbb{A}$ for O for which there exists a computable function $f : \mathbb{N} \to \mathbb{N}$ and a polynomial $p(X)$ such that the running time of $\mathbb{A}$ on input $(x,k) \in \Sigma^* \times \mathbb{N}$ is at most $f(k) \cdot p(|x|)$. ⊣

Thus an eptas is an fpt-algorithm with respect to the parameterization $(x,k) \mapsto k$ of $\Sigma^* \times \mathbb{N}$. Clearly,

$$\text{FPTAS} \subseteq \text{EPTAS} \subseteq \text{PTAS},$$

that is, if an optimization problem O has an fptas then it has an eptas, and if it has an eptas then it has a ptas. The notion of an eptas seems to be a reasonable intermediate notion of approximation schemes between the very strict fptas and the general ptas. One well-known example of an eptas is Arora's approximation scheme for the Euclidean traveling salesman problem [16]. We will see an example of an eptas for a scheduling problem in Sect. 9.4.

The following result establishes a connection between the existence of an eptas for an optimization problem and the fixed-parameter tractability of its standard parameterization. The result is simple, but interesting because it connects two completely different parameterized problems derived from the same optimization problem:

Theorem 1.32. *If the* NP*-optimization problem O has an eptas then its standard parameterization p-O is fixed-parameter tractable.*

Proof: Let us assume that $O = (\mathrm{sol}, \mathrm{cost}, \min)$ is a minimization problem over the alphabet Σ. (The proof for maximization problems is similar.) Let

$\mathbb{A}$ be an eptas for O with running time $f(k) \cdot |x|^{O(1)}$ for some computable function f. The following algorithm $\mathbb{A}'$ is an fpt-algorithm for p-O: Given an instance (x, k), algorithm $\mathbb{A}'$ computes the output y of $\mathbb{A}$ on input $(x, k+1)$. If $\mathrm{cost}(x, y) \leq k$ it accepts; otherwise it rejects.

Clearly, $\mathbb{A}'$ is an fpt-algorithm, because $\mathbb{A}$ is an fpt-algorithm. To see that $\mathbb{A}'$ is correct, we distinguish between two cases: If $\mathrm{cost}(x, y) \leq k$, then $\mathrm{opt}_O(x) \leq k$, and hence (x, k) is a "yes"-instance of p-O. If $\mathrm{cost}(x, y) \geq k+1$, then

$$\mathrm{opt}_O(x) = \frac{\mathrm{cost}(x,y)}{r(x,y)} \geq \frac{\mathrm{cost}(x,y)}{1+\frac{1}{k+1}} \geq \frac{k+1}{1+\frac{1}{k+1}} = \frac{k+1}{k+2} \cdot (k+1) > k.$$

As $\mathrm{opt}_O(x) > k$, the pair (x, k) is a "no"-instance of p-O. □

Together with hardness results from parameterized complexity theory, Theorem 1.32 can be used to establish the nonexistence of efficient polynomial time approximation schemes (under assumptions from parameterized complexity). Let us remark that the converse of Theorem 1.32 does not hold. It is known that MIN-VERTEX-COVER has no ptas unless PTIME = NP and hence no eptas, while in Sect. 1.3 we saw that the standard parameterization p-VERTEX-COVER is in FPT.

1.5 Model-Checking Problems

The parameterized approach is particularly well-suited for a certain type of logic-based algorithmic problems such as model-checking problems in automated verification or database query evaluation. In such problems one has to evaluate a formula of some logic in a finite structure. Typically, the formula (for example, a database query or a description of a system property) is small and the structure (for example, a database or a transition graph of a finite state system) is large. Therefore, a natural parameterization of such problems is by the length of the formula.

In this section, we shall study the *model-checking problem for linear temporal logic* in some more detail. In the model-checking approach to automated verification, finite state systems are modeled as *Kripke structures* (or transition systems). Formally, a *Kripke structure* is a triple $\mathcal{K} = (V, E, \lambda)$ that consists of a directed graph (V, E) together with a mapping λ that associates a set of *atomic propositions* with each vertex. The vertices of the structure represent the states of the system, the edges represent possible transitions between states, and the atomic propositions represent properties of the states, such as "the printer is busy" or "the content of register R1 is 0." Walks in the graph describe possible computations of the system. (Throughout this book, we distinguish between *walks* and *paths* in graphs. On a walk, vertices and edges may be repeated, whereas on a path each vertex and hence each edge may appear at most once.)

Linear temporal logic (LTL) is a language for specifying properties of such systems. Besides static properties of the states, which can be specified by Boolean combinations of the atomic propositions, the logic also allows it to specify temporal properties of the computations, such as: "Whenever the reset-button is pressed, eventually the system reboots." Here "the reset-button is pressed" and "the system reboots" are atomic propositions. If we represent them by the symbols *Reset* and *Reboot*, an LTL-formula specifying the property is

$$\mathsf{G}(\mathit{Reset} \to \mathsf{F}\, \mathit{Reboot}).$$

Here the G-operator says that the subformula it is applied to holds at all states of the computation following and including the current state. The subformula $\mathit{Reset} \to \mathsf{F}\, \mathit{Reboot}$ says that if *Reset* holds at a state, then $\mathsf{F}\, \mathit{Reboot}$ also holds. The F-operator says that the subformula it is applied to holds at some state of the computation after or equal to the current state. Thus $\mathsf{F}\, \mathit{Reboot}$ says that at some point in the future *Reboot* holds. We say that an LTL*-formula* φ *holds at a state* $v \in V$ *in a Kripke structure* $\mathcal{K} = (V, E, \lambda)$ (and write $\mathcal{K}, v \models \varphi$) if all walks of $\mathcal{K}$ starting at v satisfy φ. There is no need to give further details or a formal definition of LTL here.

The LTL *model-checking problem* MC(LTL) asks whether a given Kripke structure satisfies a given LTL-formula at a given state. We are mostly interested in the following parameterization:

p-MC(LTL)	
Instance:	A finite Kripke structure $\mathcal{K} = (V, E, \lambda)$, a state $v \in V$, and an LTL-formula φ.
Parameter:	Length of φ.
Problem:	Decide whether $\mathcal{K}, v \models \varphi$.

Theorem 1.33. p-MC(LTL) *is fixed-parameter tractable. More precisely, there is an algorithm solving* MC(LTL) *in time*

$$2^{O(k)} \cdot n,$$

where k is the length of the input formula φ and n the size of the input structure $\mathcal{K}$.

It is known that the unparameterized problem MC(LTL) is PSPACE-complete and thus intractable from the perspective of classical complexity. However, the problem is solved on large instances in practice. Thus the parameterized complexity analysis much better captures the "practical" complexity of the problem.

While a full proof of the theorem would mean too much of a digression at this point, it is useful to sketch the general strategy of the proof. The first step is to translate the input formula φ into a *Büchi automaton* $\mathfrak{A}$. Büchi automata are finite automata that run on infinite words. The second step is to check if the Büchi automaton $\mathfrak{A}$ accepts all walks in $\mathcal{K}$ starting at v. The first step requires exponential time, as the size m of the automaton $\mathfrak{A}$ may be exponential in the length of φ. The second step can be carried out by a fairly straightforward algorithm in time $O(m \cdot n)$, where n denotes the size of $\mathcal{K}$.

The automata-theoretic approach, as laid out here, is one of the most successful algorithmic techniques for solving logic-based problems. It has various practical applications in automated verification, database systems, and other areas. Often, the algorithms obtained by this approach are fpt-algorithms. We will study the automata-theoretic approach in more detail in Chap. 10.

Model-checking problems for various fragments of first-order logic play a very important role in the theory of parameterized intractability and will re-occur at many places in this book.

1.6 Alternative Characterizations of Fixed-Parameter Tractability

In this section we derive various characterizations of fixed-parameter tractability that emphasize different aspects of this notion. The first result shows that the standard "multiplicative" notion of fixed-parameter tractability is equivalent to an "additive" notion.

Proposition 1.34. *Let (Q, κ) be a parameterized problem. The following are equivalent:*
(1) $(Q, \kappa) \in \mathrm{FPT}$.
(2) There is an algorithm deciding $x \in Q$ in time

$$g\big(\kappa(x)\big) + f\big(\kappa(x)\big) \cdot p\big(|x| + \kappa(x)\big)$$

for computable functions f, g and a polynomial $p(X)$.
(3) There is an algorithm deciding $x \in Q$ in time

$$g\big(\kappa(x)\big) + p\big(|x|\big)$$

for a computable function g and a polynomial $p(X)$.

Proof: Clearly, (3) implies (2). We turn to the implication (2) $\Rightarrow$ (1). We may assume that the polynomial in (2) is a monomial, $p(X) = X^d$. Let Σ be the alphabet of (Q, κ) and $x \in \Sigma^*$ an instance, $n := |x|$ and $k := \kappa(x)$. Using the inequality $a + b \leq a \cdot (b + 1)$ (for $a, b \in \mathbb{N}$), we get

$$g(k) + f(k) \cdot (n + k)^d \leq (g(k) + f(k) \cdot (k + 1)^d) \cdot (n + 1)^d \leq h(k) \cdot p(n),$$

where $h(k) := g(k) + f(k) \cdot (k + 1)^d$ and $p(n) := (n + 1)^d$. Finally, from the inequality $a \cdot b \leq a^2 + b^2$ (for $a, b \in \mathbb{N}_0$), we get the implication (1) $\Rightarrow$ (3), since

$$f(k) \cdot p(n) \le f(k)^2 + p(n)^2. \qquad \square$$

A function $f : \mathbb{N} \to \mathbb{N}$ is *time constructible* if there is a deterministic Turing machine that for all $n \in \mathbb{N}$ on every input of length n halts in exactly $f(n)$ steps. Note that if f is time constructible then $f(n)$ can be computed in $O(f(n))$ steps. The following simple lemma implies that if $f(k) \cdot p(n)$ bounds the running time of an fpt-algorithm, then we can always assume the function f to be increasing and time constructible. We will often apply this lemma tacitly.

Lemma 1.35. *Let $f : \mathbb{N} \to \mathbb{N}$ be a computable function. Then there exists a computable function $g : \mathbb{N} \to \mathbb{N}$ such that:*
(1) $f(k) \le g(k)$ for all $k \in \mathbb{N}$,
(2) g is increasing,
(3) g is time constructible.

Proof: Let $g(k)$ be the running time of a Turing machine that, given k in unary, consecutively computes $f(1), f(2), \dots, f(k)$ in unary and then halts.

$\square$

We are now ready to give the two alternative characterizations of fixed-parameter tractability, which form the main result of this section.

Definition 1.36. Let (Q, κ) be a parameterized problem over Σ.

(1) (Q, κ) *is in* PTIME *after a precomputation on the parameter* if there exist an alphabet Π, a computable function $\pi : \mathbb{N} \to \Pi^*$, and a problem $X \subseteq \Sigma^* \times \Pi^*$ such that $X \in$ PTIME and for all instances x of Q we have

$$x \in Q \iff \big(x, \pi(\kappa(x))\big) \in X.$$

(2) (Q, κ) *is eventually in* PTIME if there are a computable function $h : \mathbb{N} \to \mathbb{N}$ and a polynomial time algorithm $\mathbb{A}$ that on input $x \in \Sigma^*$ with $|x| \ge h(\kappa(x))$ correctly decides whether $x \in Q$. The behavior of $\mathbb{A}$ on inputs $x \in \Sigma^*$ with $|x| < h(\kappa(x))$ is arbitrary. $\dashv$

Theorem 1.37. *Let (Q, κ) be a parameterized problem. Then the following statements are equivalent:*
(1) (Q, κ) is fixed-parameter tractable.
(2) (Q, κ) is in PTIME *after a precomputation on the parameter.*
(3) Q is decidable and (Q, κ) is eventually in PTIME.

Proof: Let Σ be the alphabet of (Q, κ).

(1) $\Rightarrow$ (2): Let $\mathbb{A}_Q$ be an algorithm deciding $x \in Q$ in time $f(\kappa(x)) \cdot |x|^c$ with computable f and $c \in \mathbb{N}$. Let Π be the alphabet $\{1, \S\}$ and define $\pi : \mathbb{N} \to \Pi^*$ by

$$\pi(k) := k \S f(k)$$

where k and $f(k)$ are written in unary. Let $X \subseteq \Sigma^* \times \Pi^*$ be the set of tuples accepted by the following algorithm $\mathbb{A}$.

Given $(x, y) \in \Sigma^* \times \Pi^*$, first $\mathbb{A}$ checks whether $y = \kappa(x)\S u$ for some $u \in \{1\}^*$. If this is not the case, then $\mathbb{A}$ rejects, otherwise $\mathbb{A}$ simulates $|u| \cdot |x|^c$ steps of the computation of $\mathbb{A}_Q$ on input x. If $\mathbb{A}_Q$ stops in this time and accepts, then $\mathbb{A}$ accepts, otherwise $\mathbb{A}$ rejects.

Since $|u| \leq |y|$, one easily verifies that $\mathbb{A}$ runs in polynomial time; moreover:

$$\begin{aligned} x \in Q &\iff \mathbb{A} \text{ accepts } \big(x, \kappa(x)\S f(\kappa(x))\big) \\ &\iff \big(x, \pi(\kappa(x))\big) \in X. \end{aligned}$$

(2) $\Rightarrow$ (3): Assume that (Q, κ) is in PTIME after a precomputation on the parameter. Choose an alphabet Π, a computable function $\pi : \mathbb{N} \to \Pi^*$, and a problem $X \subseteq \Sigma^* \times \Pi^*$ as in Definition 1.36(1). Furthermore let $\mathbb{A}_X$ be an algorithm deciding X in polynomial time. The equivalence

$$x \in Q \iff \big(x, \pi(\kappa(x))\big) \in X$$

shows that Q is decidable. We fix an algorithm $\mathbb{A}_\pi$ computing π and let $f(k)$ be the running time of $\mathbb{A}_\pi$ on input k. We present an algorithm $\mathbb{A}$ showing that (Q, κ) is eventually in PTIME.

Given $x \in \Sigma^*$, the algorithm $\mathbb{A}$ simulates $|x|$ steps of the computation of $\pi(\kappa(x))$ by $\mathbb{A}_\pi$. If the computation of $\mathbb{A}_\pi$ does not stop in this time, then $\mathbb{A}$ rejects, otherwise it simulates $\mathbb{A}_X$ to check whether $(x, \pi(\kappa(x))) \in X$ and accepts or rejects accordingly. Clearly, $\mathbb{A}$ runs in polynomial time, and for $x \in \Sigma^*$ with $|x| \geq f(\kappa(x))$:

$$\mathbb{A} \text{ accepts } x \iff x \in Q.$$

(3) $\Rightarrow$ (1): Assume (3). Let $\mathbb{A}_Q$ be an algorithm that decides $x \in Q$ and let h be a computable function and $\mathbb{A}_h$ a polynomial time algorithm correctly deciding whether $x \in Q$ for $x \in \Sigma^*$ with $|x| \geq h(\kappa(x))$. We present an fpt-algorithm $\mathbb{A}$ deciding Q: Given $x \in \Sigma^*$, first $\mathbb{A}$ computes $h(\kappa(x))$ and checks whether $|x| \geq h(\kappa(x))$. If $|x| < h(\kappa(x))$, then $\mathbb{A}$ simulates $\mathbb{A}_Q$ on input x to decide whether $x \in Q$. In this case the running time of $\mathbb{A}$ can be bounded in terms of the parameter $\kappa(x)$ and the time invested to compute this parameter, that is, by $f(\kappa(x)) + |x|^{O(1)}$ for some computable f. If $|x| \geq h(\kappa(x))$, then $\mathbb{A}$ simulates $\mathbb{A}_h$ on input x to decide whether $x \in Q$. This simulation takes time polynomial in $|x|$. Altogether, the running time of $\mathbb{A}$ can be bounded by $f(\kappa(x)) + |x|^{O(1)}$. □

The equivalence between a problem being fixed-parameter tractable and in polynomial time after a precomputation that only involves the parameter may be a bit surprising at first sight, but the proof of Theorem 1.37 reveals that it is actually based on a trivial padding argument. However, there is a more meaningful concept behind the notion of being in polynomial time after a precomputation. The instances of many natural parameterized problems have two parts, and the parameter is the length of the second part. An example is the LTL model-checking problem, the LTL-formula being the second part of an instance. Fpt-algorithms for such problems often proceed in two steps. They first do a precomputation on the second part of the input and then solve the problem using the first part of the input and the result of the precomputation. Again, LTL model-checking is a good example: The fpt-algorithm we outlined in Sect. 1.5 transforms the input formula into a Büchi automaton in a precomputation and then runs the automaton on the input structure. Another example is database query evaluation, with the first part of the input being a database and the second part the query. A natural approach to solving this problem is to first "optimize" the query, that is, to turn it into an equivalent query that can be evaluated more efficiently, and then evaluate the optimized query.

While such algorithms are not formally polynomial time after a precomputation on the parameter—they do a precomputation on the LTL-formula and on the database query, hence on part of the input—they were the original motivation for introducing the concept of a problem being in PTIME after a precomputation.

We close this chapter with one more characterization of fixed-parameter tractability. We will see in Chap. 9 that this characterization embodies a very useful algorithmic technique.

Definition 1.38. Let (Q, κ) be a parameterized problem over Σ.

A polynomial time computable function $K : \Sigma^* \to \Sigma^*$ is a *kernelization* of (Q, κ) if there is a computable function $h : \mathbb{N} \to \mathbb{N}$ such that for all $x \in \Sigma^*$ we have

$$(x \in Q \iff K(x) \in Q) \quad \text{and} \quad |K(x)| \leq h(\kappa(x)).$$

If K is a kernelization of (Q, κ), then for every instance x of Q the image $K(x)$ is called the *kernel* of x (under K). $\dashv$

Observe that a kernelization is a polynomial time many-one reduction of a problem to itself with the additional property that the image is bounded in terms of the parameter of the argument.

Theorem 1.39. *For every parameterized problem (Q, κ), the following are equivalent:*
(1) $(Q, \kappa) \in \text{FPT}$.
(2) Q is decidable, and (Q, κ) has a kernelization.

Proof: Let Σ be the alphabet of (Q, κ).

(2) $\Rightarrow$ (1): Let K be a kernelization of (Q, κ). The following algorithm decides Q: Given $x \in \Sigma^*$, it computes $K(x)$ (in polynomial time) and uses a decision algorithm for Q to decide if $K(x) \in Q$. Since $|K(x)| \leq h(\kappa(x))$, the

running time of the decision algorithm is effectively bounded in terms of the parameter $\kappa(x)$.

(1) $\Rightarrow$ (2): Let $\mathbb{A}$ be an algorithm solving (Q, κ) in time $f(k) \cdot p(n)$ for some computable function f and polynomial $p(X)$. Without loss of generality we assume that $p(n) \geq n$ for all $n \in \mathbb{N}$. If $Q = \emptyset$ or $Q = \Sigma^*$, then (Q, κ) has the trivial kernelization that maps every instance $x \in \Sigma^*$ to the empty string ϵ. Otherwise, we fix $x_0 \in Q$ and $x_1 \in \Sigma^* \setminus Q$.

The following algorithm $\mathbb{A}'$ computes a kernelization K for (Q, κ): Given $x \in \Sigma^*$ with $n := |x|$ and $k := \kappa(x)$, the algorithm $\mathbb{A}'$ simulates $p(n) \cdot p(n)$ steps of $\mathbb{A}$. If $\mathbb{A}$ stops and accepts (rejects), then $\mathbb{A}'$ outputs x_0 (x_1, respectively). If $\mathbb{A}$ does not stop in $\leq p(n) \cdot p(n)$ steps, and hence $n \leq p(n) \leq f(k)$, then $\mathbb{A}'$ outputs x. Clearly, K can be computed in polynomial time, $|K(x)| \leq |x_0| + |x_1| + f(k)$, and $(x \in Q \iff K(x) \in Q)$. $\square$

Example 1.40. Recall that p-SAT is the satisfiability problem for propositional logic parameterized by the number of variables. The following simple algorithm computes a kernelization for p-SAT: Given a propositional formula α with k variables, it first checks if $|\alpha| \leq 2^k$. If this is the case, the algorithm returns α. Otherwise, it transforms α into an equivalent formula α' in disjunctive normal form such that $|\alpha'| \leq O(2^k)$. $\dashv$

Exercise 1.41. Prove that p-*deg*-INDEPENDENT-SET has a kernelization such that the kernel of an instance $(\mathcal{G}, k)$ with $d := \deg(\mathcal{G})$ has size $O(d^2 \cdot k)$. More precisely, if $(\mathcal{G}', k')$ with $\mathcal{G}' = (V', E')$ is the kernel of an instance $(\mathcal{G}, k)$, then $|V'| \leq (d+1) \cdot k$ and hence $|E'| \leq d \cdot (d+1) \cdot k/2$.

Hint: Prove by induction on k that every graph of degree d with at least $(d+1) \cdot k$ vertices has an independent set of cardinality k. $\dashv$

Exercise 1.42. Let $\mathcal{H} = (V, E)$ be a hypergraph. A *basis* of $\mathcal{H}$ is a set S of subsets of V such that each hyperedge $e \in E$ is the union of sets in S. That is, for all $e \in E$ there are $s_1, \ldots, s_\ell \in S$ such that $e = s_1 \cup \ldots \cup s_\ell$.

Prove that the following problem is fixed-parameter tractable:

p-HYPERGRAPH-BASIS
Instance: A hypergraph $\mathcal{H}$ and $k \in \mathbb{N}$.
Parameter: $k \in \mathbb{N}$.
Problem: Decide whether $\mathcal{H}$ has a basis of cardinality k.

$\dashv$

Notes

The central notion of this book, fixed-parameter tractability, was introduced by Downey and Fellows in [78], a preliminary version of [79].[9] Earlier papers [2, 180] dealt with the asymptotic behavior of parameterized problems.

[9]Usually, we only refer to the full version of an article.

The method of bounded search trees in the context of parameterized complexity theory goes back to Downey and Fellows [81]. Most of the results presented in Sect. 1.3 (including the exercises) can be found in [83].

The notion of an efficient polynomial time approximation scheme was introduced by Cesati and Trevisan [42]; Theorem 1.32 is due to Bazgan [21].

The model-checking algorithm for linear temporal logic mentioned in Theorem 1.33 is due to Lichtenstein and Pnueli [154]. We refer the reader to [55] for background on model-checking and linear temporal logic. Papadimitriou and Yannakakis [170] point out that parameterized complexity theory yields a productive framework for studying the complexity of database query languages, which is more realistic than the classical approach.

Theorem 1.37 is due to [100]; it builds on the advice view of Cai et al. [37]. The notion of kernelization in the context of parameterized complexity theory goes back to Downey and Fellows [81]. Theorem 1.39 was shown by Niedermeier [165]. Exercise 1.41 is due to Yijia Chen (private communication). Exercise 1.42 is due to [83].