1

INTRODUCTION TO FIXED-PARAMETER ALGORITHMS

Computationally hard problems are ubiquitous. The systematic study of computational (in)tractability lies at the heart of computer science. Michael R. Garey and David S. Johnson's monograph *Computers and Intractability* from the late 1970s was a landmark achievement in this direction, providing an in-depth treatment of the corresponding theory of *NP*-completeness. With the theory of *NP*-completeness at hand, we can prove meaningful statements about the computational complexity of problems. But what happens after we have succeeded in proving that a problem is *NP*-hard (that is, "intractable"), but which nevertheless must be solved in practice? In other words, how do we cope with computational intractability?

The approach followed in this book is based on worst-case analysis of deterministic, exact algorithms to solve hard problems. In the course of dealing with intractable problems, we will also refer to several other related algorithmic methodologies, such as approximation algorithms, average-case analysis, randomized algorithms, and purely heuristic methods. Each approach has advantages and disadvantages. With regard to exact *fixed-parameter algorithms*, the advantages are

- guaranteed optimality of the solution; and
- provable upper bounds on the computational complexity.

The disadvantage is that we have to take into account

- exponential running time factors.

Clearly, exponential growth quickly becomes prohibitive when running algorithms in practice. Fixed-parameter algorithmics provides guidance on the feasibility of the "exact algorithm approach" for hard problems by means of a refined, two-dimensional complexity analysis. The fundamental idea is to strive for better insight into a problem's complexity by exploring (various) problem-specific parameters to find out how they influence the problem's computational complexity. Ideally, we aim for statements such as "if some parameter k is small in problem X, then X can be solved efficiently". For instance, in the case of the *NP*-complete graph problem Vertex Cover we know that if the solution set we are searching for is "small", then this set can be found efficiently whatever the graph looks like and however big it is—the exponential factor in the running time amounts to less than 1.28^k, where the parameter k is the size of the solution set sought.

We begin with three brief case studies of computationally hard problems. In doing so, we give a concise overview of how to cope with their computational intractability, providing the first examples of the fixed-parameter approach.

1.1 The satisfiability problem

The CNF-Satisfiability problem for Boolean formulae in conjunctive normal form may be considered the "drosophila[1] of computational complexity theory". This fundamental *NP*-complete problem has been the subject of research on exact algorithms for decades and it continues to play a central role in algorithmic research—the annual "SAT" conference is devoted to theory and applications of satisfiability testing. The problem is defined as follows.

Input: A Boolean formula F in conjunctive normal form.
Task: Determine whether or not there exists a truth assignment for the Boolean variables in F such that F evaluates to true.

Example 1.1 The formula

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$$

is satisfiable, satisfied by the truth assignment $T(x_1) = \mathit{true}$, $T(x_2) = \mathit{false}$, $T(x_3) = \mathit{true}$. An alternative satisfying assignment is $T(x_1) = \mathit{false}$, $T(x_2) = \mathit{true}$, $T(x_3) = \mathit{false}$. By way of contrast, there is no satisfying assignment for the formula

$$(x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3).$$

Numerous applications such as VLSI design and model checking make CNF-Satisfiability of real practical interest. Often, however, in concrete applications the corresponding problem instances turn out to be much easier than one might expect given the fact of CNF-Satisfiability's *NP*-completeness. For instance, large CNF-Satisfiability instances arising in the validation of automotive product configuration data are efficiently solvable. Hence the question arises of whether we can learn more about the complexity of CNF-Satisfiability by means of studying various problem parameters. In particular, we are less interested in empirical results and more in provable performance bounds related to various parameterizations of CNF-Satisfiability. In this way, we hope to obtain a better understanding of the problem properties that might be of algorithmic use.

Formally, an input of CNF-Satisfiability is a conjunction of m *clauses* where each clause consists of a disjunction of *literals*, that is, negated or non-negated Boolean variables. Let there be n different variables occurring in the formula. We list a set of various "parameterizations" of CNF-Satisfiability.

[1] *Drosophila melanogaster* is a fruit fly and is one of the most valuable organisms in biological research. It has been used as a model organism for research for almost a century.

The point subsequently is to see how the running time bounds of solving algorithms depend on the given parameters. The central question is whether and how we can confine the seemingly unavoidable combinatorial explosion to the respective parameter. In particular, how small can the corresponding exponential term be kept? We list some results from the literature given in the end of this chapter.

Parameter "clause size". The maximum number k of literals in any of the given clauses is a very natural formula parameter. For $k = 3$ (that is, 3-CNF-Satisfiability), however, the problem remains *NP*-complete, whereas it is polynomial-time solvable for $k = 2$ (that is, 2-CNF-Satisfiability). Thus, for upperbounding the combinatorial explosion, this parameterization seems of little help.

Parameter "number of variables". The number n of different variables occurring in a formula significantly influences the computational complexity. Since there are 2^n different truth assignments, in essentially[2] this number of steps CNF-Satisfiability can be solved. When restricting the maximum clause size by some k, for instance, $k = 3$, better bounds are known. The currently best upper bound for a deterministic algorithm solving 3-CNF-Satisfiability is below 1.49^n. The currently best randomized algorithm for 3-CNF-Satisfiability has expected running time below 1.33^n.

Parameter "number of clauses". If the number of clauses in a formula can be bounded from above by m, then CNF-Satisfiability can be solved in 1.24^m steps.

Parameter "formula length". If the total length (that is, counting the number of literal occurrences in the formula) of the formula F is bounded from above by $\ell := |F|$, then CNF-Satisfiability can be solved in 1.08^ℓ steps.

The above parameterizations do not suffice to explain all the cases of good behaviour of CNF-Satisfiability in many practical situations. There are application scenarios where none of them would lead to an efficient algorithm. Hence the following way of parameterizing CNF-Satisfiability seems to offer good prospects.

Parameters exploiting "formula structure". There are several recent exact algorithms with exponential bounds depending on certain *structural* formula parameters. These parameters are based on structural graph decompositions where a graph is associated with the formula structure—for instance, one obtains so-called variable interaction graphs. The basic idea is that the way in which different variables (or literals) occur in common clauses—and thus how they interact—strongly influences the complexity of finding a satisfying assignment if any exists. Then structure and width

[2]We neglect polynomial-time factors in the running time analysis throughout the book whenever they play a minor role in our considerations.

concepts for graphs (as considered later in this book) are exploited to derive tractability results for formulae with particular structural properties on the graph side.

Finally, from the parameterized complexity *theory* point of view the following parameterization is of particular relevance.

Parameter "weight of assignment". Here it is asked whether a formula has a satisfying assignment with *exactly* k variables being set to true. Although the parameterization with this "weight parameter" seems rather artificial, it plays a major role in characterizing parameterized intractability and the corresponding structural complexity theory. It serves as an anchor point in the parameterized hardness program similar to the role that CNF-Satisfiability plays in classical *NP*-completeness theory. Thus, in particular, it is considered very unlikely that this version of CNF-Satisfiability can be solved in $f(k) \cdot |F|^{O(1)}$ steps. So far it is only known how to achieve the trivial running time $n^{O(k)}$ by testing all $\binom{n}{k}$ candidate truth assignments—it seems that the combinatorial explosion cannot be confined to a function exclusively depending on k. Note that this hardness already holds true when considering the weighted version of 2-CNF-Satisfiability.

In contrast, if one asks whether there is a satisfying assignment with *at most* k variables set true, then this weighted 2-CNF-Satisfiability can be easily solved using a size-2^k search tree: as long as there are clauses with only positive literals, take an arbitrary one of these clauses and branch into the two cases, setting either of the variables true (at least one of the two variables must be set true). In each branch, simplify the formula according to the variable chosen to be set true. After that, a formula remains where every clause has at least one negative literal and this instance is trivial to solve. Clearly, this method generalizes to arbitrary constant clause sizes.

To summarize, different ways of parameterizing CNF-Satisfiability lead to a better understanding of the problem's inherent complexity facets. In particular, it is hoped that structural parameterizations in the future will yield new insights into tractable cases of CNF-Satisfiability; but, in any case, there surely is *no "best parameterization"*. As a rule, the nature of the complexity of hard computational problems will probably almost always require such a multi-perspective view in order to gain better insight into practically relevant, efficiently solvable *special cases*. To cope with intractability in a mathematically sound way, it seems that we have to pay the price of shifting our view from considering the input through just one pair of glasses (mostly these glasses are called "complexity measurement relative to input size") to a multitude of pairs of glasses, each of them with a different focus (that is, parameter).

CNF-Satisfiability, as defined above, is a *decision problem* with answer either "yes" or "no", usually also providing a satisfying truth assignment if it exists. In most application scenarios considered in this book, however, we will have

to deal with *optimization problems* where the goal is to minimize or maximize a certain solution value. An optimization version of CNF-SATISFIABILITY is MAXIMUM (CNF-)SATISFIABILITY, in which one is asked to find a truth assignment that simultaneously satisfies as many clauses as possible. Obviously, SATISFIABILITY is a special case of MAXIMUM SATISFIABILITY in the sense that here one asks whether all clauses can be satisfied. Analogous parameterizations to those for SATISFIABILITY can also be investigated for MAXIMUM SATISFIABILITY.

Parameter "clause size". Even for maximum clause size $k = 2$ (MAXIMUM 2-SATISFIABILITY) the problem remains *NP*-complete.

Parameter "number of variables". Analogously to SATISFIABILITY, MAXIMUM SATISFIABILITY can be solved in 2^n steps, where n denotes the number of variables appearing in the given formula. It is an open problem to obtain better bounds even if the maximum clause size is 3 (MAXIMUM 3-SATISFIABILITY). A recent breakthrough shows, however, that MAXIMUM 2-SATISFIABILITY can be solved in 1.74^n steps.

Parameter "number of clauses". If the number of clauses in a formula can be bounded from above by m, then MAXIMUM SATISFIABILITY can be solved in 1.33^m steps. MAXIMUM 2-SATISFIABILITY can be solved in 1.15^m steps.

Parameter "formula length". If the total length of the formula is bounded from above by ℓ, then MAXIMUM SATISFIABILITY can be solved in 1.11^ℓ steps. MAXIMUM 2-SATISFIABILITY can be solved in 1.08^ℓ steps.

We are not aware of investigations of MAXIMUM SATISFIABILITY concerning parameterizations according to "formula structure" as discussed for SATISFIABILITY.

1.2 An example from railway optimization

In our second example, we deal with a graph problem that is directly motivated by an application in railway optimization. Although the fixed-parameter approach in this case so far is not able to prove tractability according to a reasonable parameterization, the preprocessing technique presented is of great interest as a core algorithmic tool in parameterized complexity studies. The problem is defined as follows.

> **Input:** A set of trains, a set of train stations, and for each train the stations where it stops.
> **Task:** Find a minimum size set of train stations such that each train stops in at least one of the selected stations.

The motivation is that at the selected stations S one wants to build some supply stations for trains. To save costs, the set S shall be as small as possible. The formalization as a graph problem leads to an *NP*-complete version of the DOMINATING SET problem in bipartite graphs. To do so, one associates trains with one vertex set and stations with the other vertex set; one draws an edge between a train vertex and a station vertex iff the train stops in this station.

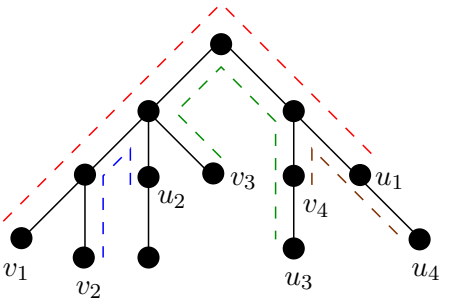

FIG. 1.3. An instance of MULTICUT IN TREES with four demand paths.

~~instance may have changed such that data reduction again applies. As a rule, data reduction is frequently so effective that there is rarely a case where one would not want to exploit it.~~

1.3 A communication problem in tree networks

Our third and last introductory example gives a concrete fixed-parameter algorithm together with its simple analysis. More specifically, we consider the problem MULTICUT restricted to trees. It is motivated by applications in communication networks. The problem is studied on general graphs as well; it is one example of a relatively small number of graph problems that remain *NP*-complete even when restricted to trees. MULTICUT IN TREES is defined as follows.

> **Input:** An undirected tree $T = (V, E)$, $n := |V|$, and a collection H of m pairs of nodes in V, $H = \{(u_i, v_i) \mid u_i, v_i \in V, u_i \neq v_i, 1 \leq i \leq m\}$.
> **Task:** Find a minimum size subset E' of E such that the removal of the edges in E' separates each pair of nodes in H.

Each pair of nodes in a tree uniquely determines a corresponding path which we subsequently refer to as a *demand path*.

Example 1.4 Figure 1.3 gives an example instance for MULTICUT IN TREES with four pairs of nodes, that is, four demand paths. There is an optimal solution which removes two edges as marked in Figure 1.4.

By trying all possibilities (and using the fact that for trees $|E| = n-1$) we can solve the problem in 2^{n-1} steps. Can we do better if there exists a small solution set E'? Consider the parameter $k := |E'|$ and study how k influences the problem complexity. Note that clearly $k < n$ but in some application scenarios $k \ll n$ seems to be a reasonable assumption. Here, the following simple algorithm works. Assume that the given tree is (arbitrarily) rooted—that is, choose an arbitrary node $r \in V$ and direct all edges from r to its neighbors (that is, children), from r's children to their children, and so on. In Figure 1.3 the chosen node r is drawn at the top of the picture. Consider a pair of nodes $(u, v) \in H$ such that the uniquely determined path p between u and v has maximum distance from

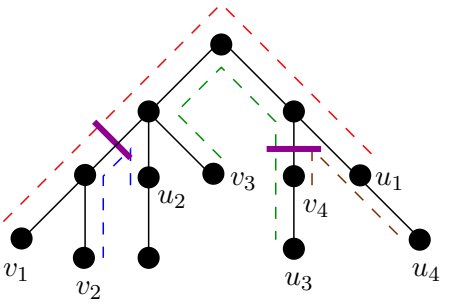

FIG. 1.4. An optimal solution that removes the two marked edges.

the root r. In Figure 1.3 (u_2, v_2) or (u_4, v_4) is such a pair. Call w the node on p that is closest to r; that is, w is the least common ancestor of u and v.

Then one may easily see (using the tree structure) that there is an optimal solution that contains at least one of the two path edges connected to w. This implies that we can build a search tree of depth bounded by k: simply search for w as specified above and then branch into the two cases (at least one of them *has to* yield an optimal solution) of taking one of the two neighboring path edges of w into the solution edge set to be constructed. Each of these two cases represents the deletion of one of the two edges. After deleting an edge, we remove all no longer connected node pairs from H. Iterating this process at most k times, we obtain a search tree of size bounded by 2^k, and thus we have a fixed-parameter algorithm for MULTICUT IN TREES with respect to parameter k. ~~Note that in a straightforward way, by taking both edges, one also obtains a polynomial-time factor-2 approximation algorithm. Nothing better is known. An in-depth treatment of depth- and thus size-bounded search trees is given in Chapter 8.~~

~~Additionally, there is a set of simple data reduction rules for MULTICUT IN TREES. In the description of the subsequent rules, we often *contract an edge* e.~~ Let $e = \{v, w\}$ and let $N(v)$ and $N(w)$ denote the sets of neighbors of v and w, ~~respectively. Then, contracting e means that we replace v and w by one new~~ vertex x and we set $N(x) := N(v) \cup N(w) \setminus \{v, w\}$. We occasionally consider paths P_1 and P_2 in the tree and we write $P_1 \subset P_2$ when the node set (and edge ~~set) of P_2 comprises that of P_1. Now, the correctness of the following four data reduction rules and their polynomial-time realizability is easy to see.~~

~~**Idle Edge.** If there is a tree edge with no path passing through it, then contract this edge.~~

~~**Unit Path.** If a demand path has length one (in other words, it consists of exactly one edge), then the corresponding edge e has to be in the solution set E'. Contract e and remove all demand paths passing e from H and decrease the parameter k by one.~~

~~**Dominated Edge.** If all demand paths that pass edge e_1 of T also pass edge e_2 of T, then contract e_1.~~

3

PARAMETERIZED COMPLEXITY THEORY—A PRIMER

We briefly sketch general aspects of the theoretical basis of the study of parameterized complexity. The focus of this chapter, however, lies on the algorithmic side of fixed-parameter tractability, and the consideration of complexity-theoretic issues remains very limited here. A more formal treatment follows in Part III.

3.1 Basic theory

The *NP*-complete minimization problem VERTEX COVER is the best studied problem in the field of fixed-parameter algorithms:

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Task:** Find a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.

Figure 3.1 illustrates some basic graph problems occurring in the course of this section.

In what follows, let $n := |V|$ denote the number of graph vertices. VERTEX COVER is *fixed-parameter tractable*: there are algorithms solving it in $O(1.28^k + kn)$ time. By way of contrast, consider the also *NP*-complete maximization problem CLIQUE:

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Task:** Find a subset of vertices $C \subseteq V$ with k or more vertices such that C forms a clique, that is, C induces a complete subgraph of G.

CLIQUE appears to be *fixed-parameter intractable*: it is *not* known whether it can be solved in time $f(k) \cdot n^{O(1)}$, where f may be a computable but arbitrarily fast growing function only depending on k. The well-founded conjecture is that no such fixed-parameter algorithm for CLIQUE exists. The best known algorithm solving CLIQUE runs in time $O(n^{ck/3})$, where c is the exponent on the time bound for multiplying two integer $n \times n$ matrices (currently, $c = 2.38$). Note that $O(n^{k+1})$ is the running time for the straightforward algorithm that just checks all size-k subsets. The decisive point is that k appears in the exponent of n, and there seems to be no way "to shift the combinatorial explosion only into k", independent from n.

The observation that *NP*-complete problems like VERTEX COVER and CLIQUE behave completely differently in a "parameterized sense" lies at the very heart of parameterized complexity, a theory pioneered by Rod G. Downey and Michael R. Fellows. In the remainder of the book, we will mainly concentrate

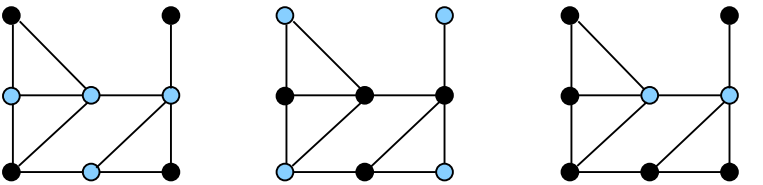

FIG. 3.1. From left to right, the three central parameterized problems VERTEX COVER, INDEPENDENT SET, and DOMINATING SET with optimal solution sets marked. Their parameterized complexity ranges from *FPT* over $W[1]$-complete to $W[2]$-complete. The parameter is always defined as the size of the solution set and the respective solution sets are marked by light shading (opposite to the dark shading of the remaining vertices).

on the world of fixed-parameter tractable problems such as those exhibited by VERTEX COVER. Here, we only briefly sketch some very basic ideas from the theory of parameterized intractability in order to provide some background on parameterized complexity theory and the ideas behind it.

We begin with some basic definitions of parameterized complexity theory.

Definition 3.1 *A* parameterized problem *is a language $L \subseteq \Sigma^* \times \Sigma^*$, where Σ is a finite alphabet. The second component is called the parameter of the problem.*

In practically all the examples in this work the parameter is a nonnegative integer or a set of nonnegative integers. Hence we will usually write $L \subseteq \Sigma^* \times \mathbb{N}$ instead of $L \subseteq \Sigma^* \times \Sigma^*$. In principle, however, the above definition leaves open the possibility of also defining more complicated parameters; for instance the subgraphs one is searching for in a graph. For $(x, k) \in L$, the two dimensions of parameterized complexity analysis are constituted by the input size n, that is, $n := |(x, k)|$ and the parameter value k (usually a nonnegative integer) or its size.

Definition 3.2 *A parameterized problem L is* fixed-parameter tractable *if it can be determined in $f(k) \cdot n^{O(1)}$ time whether or not $(x, k) \in L$, where f is a computable function only depending on k. The corresponding complexity class is called FPT.*

In the most general form of fixed-parameter tractability we have a multiplicative connection between $f(k)$ and the polynomial running time part. The running time for VERTEX COVER is of the form $f(k)+n^{O(1)}$. Using "asymptotic considerations" or, more importantly, by data reduction through preprocessing, an algorithm with $f(k) \cdot n^{O(1)}$ running time can be transferred into an algorithm with $g(k) + n^{O(1)}$ running time, g again being a function depending only on parameter k. See Chapter 7 in Part II for more on this.

Fixed-parameter tractability is the key notion in this book. In Definition 3.2, one may assume any standard model of sequential deterministic computation

such as deterministic Turing machines or RAMs. For the sake of convenience, if not stated otherwise, we will always take the parameter, denoted by k, as a nonnegative integer encoded with a unary alphabet. Unary encoding avoids "dirty tricks" in the analysis of the computational complexity of the algorithms that might be possible using binary encoding.

A core tool in the development of fixed-parameter algorithms is polynomial-time preprocessing by *data reduction rules*, often yielding a *reduction to a problem kernel*. Here, the goal is, given any problem instance x with parameter k, to transform it into a new instance x' with parameter k' such that the size of x' is bounded from above by some function depending only on k, $k' \leq k$, and (x, k) has a solution iff (x', k') has a solution. Observe that data reduction (as we have already discussed for MULTICUT IN TREES in Section 1.3) as defined here is not to be confused with the subsequently defined (Definition 3.3) concept of parameterized reduction. The first is an algorithmic tool important for fixed-parameter tractability and the second is a complexity-theoretic tool important for fixed-parameter intractability.

Proving nontrivial, absolute lower bounds on the computational complexity of problems appears to be very difficult and has made relatively little progress so far. Hence it is probably not surprising that up to now there is no proof that *no* $f(k) \cdot n^{O(1)}$ time algorithm for CLIQUE exists. In a more complexity-theoretical language, this can be rephrased by saying that it is unknown whether CLIQUE $\in FPT$. Analogously to classical complexity theory, Downey and Fellows developed some way out of this quandary by providing a reducibility and completeness program. The completeness theory of parameterized intractability involves significantly more technical effort than the classical one. We very briefly sketch some integral parts of this theory in the following.

To prove the "relative hardness" of parameterized problems, we first need a reducibility concept:

Definition 3.3 *Let $L_1, L_2 \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems. We say that L_1* reduces to *L_2* by a (standard) parameterized (many-one-)reduction *if there are functions $k \mapsto k'$ and $k \mapsto k''$ from $\mathbb{N}$ to $\mathbb{N}$ and a function $(x, k) \mapsto x'$ from $\Sigma^* \times \mathbb{N}$ to Σ^* such that*

1. *$(x, k) \mapsto x'$ is computable in $k'' \cdot |(x, k)|^c$ time for some constant c and*
2. *$(x, k) \in L_1$ iff $(x', k') \in L_2$.*

Notably, most reductions from classical complexity turn out *not* to be parameterized ones. Consider the *NP*-complete maximization problem INDEPENDENT SET (also known as STABLE SET).

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Task:** Find a subset of vertices $I \subseteq V$ with k or more vertices that form an independent set, that is, I induces an edgeless subgraph of G.

Refer to Figure 3.1 for an example. The well-known reduction from INDEPENDENT SET to VERTEX COVER, which is given by letting $G' = G$ and $k' = |V| - k$

7

DATA REDUCTION AND PROBLEM KERNELS

Every phone book is sorted. In a sorted structure, by using binary search we can find items in logarithmic time. Sorting is the fundamental algorithmic problem in computer science. Indeed, "when in doubt, sort" is considered as one of the first rules of algorithm design. Thus sorting is a prime example of simplifying or, in this case more specifically, structuring the input by efficient preprocessing. In this chapter we will present a special form of data simplification. Whereas in the case of sorting the simplification of the input consists in restructuring it to make it sorted, the focus in the parameterized complexity context is on simplifying the input by shrinking it in size.

Whereas the sorting problem is a computationally feasible problem, we focus on "intractable" problems. If a computationally hard problem must be solved in practice, one of the first things usually done is to try to perform a reduction of the size of the input data. Many input instances consist of some parts that are relatively easy to cope with and other parts that form the "really hard" core of the problem. Hence, before starting a cost-intensive algorithm to solve the difficult problem, a polynomial-time preprocessing phase is executed in order to shrink the given input data as much (and as fast) as possible.

In Part I we have already seen two examples of such data reduction techniques that can be executed in polynomial time. In Section 1.2, considering a problem from railway optimization, the two simple reductions given by the Train Rule and the Station Rule were presented as efficient (polynomial-time) preprocessing rules with enormous success in empirical studies. A theoretical confirmation of the strength of these rules is still missing today, though. In Section 1.3, considering the communication network problem Multicut in Trees, we discussed several simple data reduction rules. In this case, by way of contrast, the "quality" of data reduction can actually be proven by showing a size-bounded "problem kernel"—this form of "guaranteed quality of data reduction" will be the core concept of this chapter.

Observe the "universal importance" of data reduction by preprocessing. It is not only a ubiquitous topic for the design of efficient fixed-parameter algorithms, but it is of importance for basically *any* method (such as approximation or purely heuristic algorithms) that tries to cope with hard problems. It is important to emphasize, however, that the use of data reduction techniques is not restricted to a preprocessing phase only. On the contrary, there is empirical as well as theoretical evidence that it is beneficial to combine or interleave data reduction techniques with the "main algorithm" for problem solution, achieving significant speedups in this way. Data reduction is one of the most important techniques in

designing fixed-parameter algorithms and is also useful for other paradigms in the field of algorithmics for hard problems.

We start with two concrete examples of data reduction. First, consider CNF-Satisfiability, where one is given a Boolean formula F in conjunctive normal form and the task is to decide whether or not F has a satisfying truth assignment (refer to Section 1.1). Clearly, if there are clauses consisting of only one literal then to satisfy F one must satisfy these "unit-clauses" by setting the value of the corresponding variable accordingly. There is no choice here. This can be accomplished by a simple scan through the formula phase, and it may shrink the original input formula considerably, resulting in a reduced formula.

Second, let us return to our running example Vertex Cover:

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Task:** Find a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.

It is clearly permissible to remove isolated vertices, that is, vertices with no adjacent edges. Moreover, if one is looking for only *one* optimal vertex cover and *not all* of them, then vertices with only one adjacent edge, and thus one adjacent vertex, can easily be dealt with by putting the neighboring vertex into the cover. This is correct because in order to cover the corresponding edge one of the two endpoints *must* be in the vertex cover and a vertex with higher degree has the potential to cover more edges. In the fixed-parameter setting, where we ask for a vertex cover of size at most k, we can further do the following. If there is a vertex of degree at least $k+1$, that is, a vertex with more than k adjacent edges, then, if a vertex cover of size k exists, this particular vertex must be part of it. Otherwise, to cover all its adjacent edges would require all its at least $k + 1$ neighbors, a contradiction. This is known as *Buss's reduction to a problem kernel* for Vertex Cover. One can easily verify that after performing the above rules, in order to have a Vertex Cover of size at most k the remaining graph can have at most $k^2 + k$ vertices and at most k^2 edges. For the time being, however, our sole point of interest here is that all three of the above rules (concerning isolated vertices, vertices of degree one, and vertices of degree at least $k + 1$) can be executed in polynomial time. Thus, we may obtain an efficiently executable data reduction. In Section 7.4, we will see that Vertex Cover even allows for a much more sophisticated and stronger data reduction.

The methodological approach, including various techniques of data reduction or problem kernelization is best learned by a series of concrete examples. This will be the main contents of this chapter. The examples are selected in such a way that it is hoped that they well represent standard techniques that have been employed in this context. So, our extensive discussions are based on diverse problems such as Maximum Satisfiability, Cluster Editing, 3-Hitting Set, Vertex Cover, and Dominating Set in Planar Graphs. To facilitate illustration, the majority of the chosen problems are graph problems, although data reduction techniques are of interest and apply to all sorts of hard problems.

In summary, data reduction is a topic with practical importance that cannot be overrated and it belongs as an important key technique in *every* algorithm designer's toolbox. Formal studies of data reduction techniques are still underrepresented in the algorithms literature and it will become a growing field of research on its own. In the context of fixed-parameter algorithms, this basically comes down to what is known as reduction to a problem kernel. We formally introduce this notion in the first section.

7.1 Basic definitions and facts

Besides the notion of fixed-parameter tractability itself, the concept of *reduction to a problem kernel* or, equivalently, *kernelization*, leads to the most important definition in this book. It captures data reduction taken from a fixed-parameter complexity point of view. To simplify matters, it is assumed that the parameter is a positive integer. Cases with more than one parameter being a number are each handled analogously.

Definition 7.1 *Let $\mathcal{L}$ be a parameterized problem, that is, $\mathcal{L}$ consists of input pairs (I, k), where I is the problem instance and k is the parameter.* Reduction to a problem kernel *then means to replace instance (I, k) by a "reduced" instance (I', k') (called* problem kernel*) such that*

$$k' \leq k, \qquad |I'| \leq g(k)$$

for some function g only depending on k, and

$$(I, k) \in \mathcal{L} \text{ iff } (I', k') \in \mathcal{L}.$$

Furthermore, the reduction from (I, k) to (I', k') must be computable in polynomial time $T_K(|I|, k)$. The function $g(k)$ is called the size *of the problem kernel.*

Clearly, the aforementioned kernelization for Vertex Cover due to Buss fits into the given definition.

Two remarks:

- Definition 7.1 requires that $k' \leq k$. In principle, it could even be allowed that $k' = f(k)$ for some arbitrary function f. We are not aware of a concrete, natural parameterized problem with a problem kernel where $k' > k$.
- This remark concerns the type of parameterizations for which we study reduction to a problem kernel. All kernelizations we are aware of refer to parameterized problems where the parameter reflects the value to be optimized in the corresponding optimization problem.

To further substantiate and to state more precisely the last remark, recall the problem Multicut in Trees introduced in Section 1.3:

Input: An undirected tree $T = (V, E)$, $n := |V|$, and a collection H of m pairs of vertices in V, $H = \{(u_i, v_i) \mid u_i, v_i \in V, u_i \neq v_i, 1 \leq i \leq m\}$.
Task: Find a minimum size subset E' of E such that the removal of the edges in E' separates each pair of vertices in H.

In Section 1.3 we discussed the parameterization by $k := |E'|$. In Section 5.1 we briefly introduced another parameterization, namely the maximum number d of demand paths passing through a tree node. By employing dynamic programming techniques also with respect to parameter d we can achieve fixed-parameter tractability. For parameter k, however, with some technical expenditure a problem kernel can be proven. By way of contrast, we see no point in searching for a problem kernel with respect to d. Note that k corresponds to the optimization value in Multicut in Trees, whereas d corresponds to a "structural property" of the input which is basically unrelated to the optimization value. Hence it is not conceivable to get a reduced instance with an upper bound on its size that depends only on the parameter d because arbitrarily large instances with small d but a large solution set exist.

For computationally hard problems, the best one can hope for is that a problem kernel has size linear in k, a so-called *linear problem kernel*. According to Definition 7.1, this formally means that, for a given problem parameter k, the reduced instance I' is of size $O(k)$. For graph problems, however, this is often confused with the property that the reduced graph $G' = (V', E')$ fulfills $|V'| = O(k)$, whereas the total instance size also including the number of edges still may be $O(k^2)$. Nevertheless, one frequently terms this a linear problem kernel. A concrete example of this is given with the problem kernel for Vertex Cover that contains only $2k$ vertices; see Section 7.4.

One can classify kernelization algorithms or, more specifically, data reduction rules, into two types:

- *parameter-independent* and
- *parameter-dependent* ones.

The distinguishing factor is whether or not the kernelization makes explicit use of the parameter value. For instance, consider Vertex Cover. The notion of "high-degree vertices" as employed in the kernelization attributed to Buss is parameter-dependent because it needs to know the value of parameter k. By way of contrast, the rule which simply put the neighbor of a degree-one vertex into the vertex cover does not need to know the value of k; it is parameter-independent. Clearly, "parameter-independence" is preferable, but it seems hard to achieve in all cases as we will see in the case studies to follow. For parameter-dependent rules, to find a solution with *optimal* parameter value one has to try several candidates for parameter k in a systematic fashion. For the time being, just note that currently the only known kernelization of Multicut in Trees is based on a mixture of parameter-dependent and parameter-independent data reduction rules.

To get further acquainted also with the pitfalls of problem kernels, let us now have a brief look at a somewhat strange kind of problem kernelization. Consider Independent Set in Planar Graphs:

> **Input:** A planar graph $G = (V, E)$ and a nonnegative integer k.
> **Task:** Find a subset of vertices $I \subseteq V$ with k or more vertices that form an independent set, that is, I induces an edgeless subgraph of G.

Independent Set in Planar Graphs has a problem kernel consisting of only $4k$ vertices: due to the famous four-color theorem for planar graphs and the corresponding polynomial-time coloring algorithm one can color the vertices of a given planar graph with four colors such that no two neighboring vertices possess the same color. Hence each "color class" forms an independent set of the graph, and, since we need only four color classes, one of them must contain at least one fourth of all vertices.

Thus the reduction to a problem kernel may simply proceed as follows: if for a given planar graph with n vertices and parameter k it holds that $k \leq \lceil n/4 \rceil$, then answer "yes" and produce an independent set using the polynomial-time coloring algorithm. If $k > \lceil n/4 \rceil$, then $n < 4k$ and, voilà, we have a problem kernel with $4k$ vertices. On the one hand, in a way, this kind of kernelization is not satisfactory because in the second case we did not reduce the size of the input graph at all, but simply made the indirect observation that it must have a big (!) independent set that contains at least one quarter of all graph vertices. This contradicts the common assumption of parameters being "small" and directs our attention to the now more natural parameterization above the guaranteed value $\lceil n/4 \rceil$, see Section 5.2. On the other hand, this example also shows that there may be deep theory behind the construction of problem kernels: here, the famous four-color theorem and the corresponding intricate coloring algorithm.

This problem kernelization for Independent Set in Planar Graphs based on the four-color theorem also has a "global flavor", whereas the majority of known reductions to a problem kernel work by using "local rules". More specifically, the corresponding data reduction rules—we usually need more than one single rule to prove a problem kernel—explore local structures within an input instance. Based exclusively on this, these rules decide whether the input can be be simplified (or, equivalently, reduced) here. For instance, in the case of Vertex Cover such a local structure may be a vertex together with all its neighbors. If it has only one neighbor, a very simple data reduction rule says that we should put its neighbor into the vertex cover and remove it from the given instance together with all its incident edges, no matter what the remaining graph looks like.

Finally, let us mention in passing that in parameterized complexity theory it has become a commonplace that "every fixed-parameter tractable problem is kernelizable". To begin with, note that it is obvious that if there is a reduction to a problem kernel for a decidable parameterized problem, then it is fixed-parameter tractable: simply perform a brute-force search algorithm on the

remaining problem kernel. The opposite direction is a little less obvious: assume that the given fixed-parameter algorithm has running time $f(k) \cdot n^c$ for some positive constant c. The idea is to run this algorithm on the problem for at most n^{c+1} steps and then to consider the two cases that either the algorithm has finished its task within that time or it has not finished. In the first case, we directly obtain a kernelization algorithm running in polynomial time n^{c+1}, which simply outputs either a trivial "no"-instance or a trivial "yes"-instance. In the second case, we can argue that $n < f(k)$. Thus, our problem kernel is the original input instance itself. We can summarize these arguments as follows.

Proposition 7.2 *A decidable parameterized problem $\mathcal{L}$ is fixed-parameter tractable with respect to parameter k iff there exists a reduction to a problem kernel for $\mathcal{L}$ with respect to k.*

Proof Following the discussion preceding Proposition 7.2, it remains to be shown that $n < f(k)$ in the case that the fixed-parameter algorithm has not finished its task within n^{c+1} steps for a positive constant c. If so, this means that

$$n^{c+1} < f(k) \cdot n^c,$$

which yields $n < f(k)$. □

Clearly, Proposition 7.2 is of no direct practical use and gives only a trivial problem kernel with no algorithmic impact. As a matter of experience, one may conclude that, although it is often not hard to find some simple data reduction rules for a fixed-parameter tractable problem, to prove that they really yield a non-trivial problem kernel in the sense of Definition 7.1 frequently becomes a mathematically challenging task—it is worth the effort! We will elaborate several concrete problem kernelizations in the remainder of this chapter.

7.2 Maximum Satisfiability

To start our series of example kernelizations, we present a simple reduction to a problem kernel for the *NP*-complete Maximum Satisfiability (MaxSat) problem (see also Section 1.1):

> **Input:** A Boolean formula in conjunctive normal form consisting of m clauses and a nonnegative integer k.
> **Task:** Find a truth assignment satisfying at least k clauses.

We represent the Boolean values true and false by 1 and 0, respectively. A *truth assignment* I can be defined as a set of literals that contains no pairs of complementary literals. Then for a variable x we have $I(x) = 1$ iff $x \in I$ and $I(x) = 0$ iff $\bar{x} \in I$. We deal only with propositional formulae in conjunctive normal form. These are often represented in *clause form*, that is, as a set of clauses, where a clause is a set of literals. We represent formulae as *multi-sets* of sets since a formula might contain some identical clauses. For the Satisfiability problem

(see Section 1.1) multiple clauses can be eliminated, but this is of course no longer true if we are interested in the number of satisfiable clauses. The formula

$$(x \vee y \vee \bar{z}) \wedge (x \vee y \vee \bar{z}) \wedge (\bar{x} \vee z) \wedge (\bar{y} \vee z)$$

will be represented as

$$\{\{x, y, \bar{z}\}, \{x, y, \bar{z}\}, \{\bar{x}, z\}, \{\bar{y}, z\}\}.$$

Note that the outer curly brackets denote the multi-set and the inner curly brackets denote the sets of literals. The *length of a clause* is its cardinality, and the *length of a formula* is the sum of the lengths of its clauses. To simplify presentation, we assume that each clause contains at most one occurrence of a variable x. Note that a clause where both literals x and $\bar{x}$ occur is trivially always satisfied.

Suppose that we are given an input instance for MaxSat. The first simple observation is that if $k \leq \lceil m/2 \rceil$, then the desired truth assignment trivially exists: take a random truth assignment. If it satisfies at least k clauses then we are done. Otherwise, "flipping" each bit in this truth assignment to its opposite value yields a new truth assignment that now satisfies at least $\lceil m/2 \rceil$ clauses.

Hence from now on we can assume that $k > \lceil m/2 \rceil$, which implies that $m < 2k$. The next observation gives a problem kernel of quadratic size: partition the clauses of the given formula F into long clauses (that is, clauses containing k or more literals) and into short clauses (that is, clauses containing less than k literals). Thus,

$$F = F_l \wedge F_s,$$

where F_l contains all long clauses and F_s contains all short clauses. Let L be the number of long clauses. If $L \geq k$ then again at least k clauses can be satisfied by picking (in the worst case) in each long clause another variable and setting its value accordingly such that the corresponding clause gets satisfied.

If $L < k$, then an important point is that now it is sufficient to focus attention on the new MaxSat instance $(F_s, k-L)$, which already gives our problem kernel due to the following observations. Note that (F, k) is a yes-instance of MaxSat iff $(F_s, k-L)$ is a yes-instance of MaxSat: the same reasoning as in the preceding paragraph shows that the remaining L large clauses can always be satisfied. This is true because to satisfy $k-L$ clauses at most $k-L$ variables (at most one variable per satisfied clause) are needed. Thus, for the L large clauses at least $k-(k-L) = L$ variables remain to be freely set and the claimed equivalence is shown.

It remains to show that the size of the formula F_s (making the reduced problem instance) is bounded from above by $O(k^2)$: we have that there are $m - L \leq m$ small clauses, each containing at most k literals, and we have that $m < 2k$ as explained before. Hence the total number of literal occurrences in F_s is bounded from above by $2k \cdot k = 2k^2$. This means a quadratic-size problem kernel for MaxSat with respect to parameter k.

Proposition 7.3 MAXIMUM SATISFIABILITY *has a problem kernel of size* $O(k^2)$, *and it can be found in linear time.*

Proof The above described method and its correctness are clear. With regard to the running time, it is easy to see that the determination of the small and large clauses as well as the determination of an assignment satisfying all large clauses can be done in linear time by simply scanning the given formula. □

In summary, we observe that the given problem kernelization for MAXIMUM SATISFIABILITY makes explicit use of parameter value k to perform the partitioning into a small and a large formula; hence the described data reduction is parameter-dependent. Moreover, the reduction to a problem kernel for MAXSAT has the same unsatisfactory flavor as that which INDEPENDENT SET IN PLANAR GRAPHS (see Section 7.1) had. The point is that again it is the use made of the structural property of the input instance which guarantees "beforehand" that at least half of all clauses are always satisfiable. Hence, similar to the case of INDEPENDENT SET IN PLANAR GRAPHS, parameterizing above guaranteed values would probably be more appropriate here (see Section 5.2). This is also discussed in the related literature, see the bibliographical remarks in Section 7.10.

7.3 Cluster Editing

Our next example of problem kernelization deals with a graph modification problem arising in the field of data clustering. Different from the MAXIMUM SATISFIABILITY case, in this new example we work with with mainly *local* data reduction rules. Again, however, the rules are parameter dependent. The problem definition is based on the notion of a *similarity graph* whose vertices correspond to data elements and in which there is an edge between two vertices iff the similarity of their corresponding elements exceeds a predefined threshold. The goal is to obtain a *cluster graph* by as few edge modifications (that is, edge deletions or edge additions) as possible; a cluster graph is a graph in which each of the connected components is a clique. Thus we arrive at the *NP*-complete edge modification problem CLUSTER EDITING:

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Task:** Find out whether we can transform G, by deleting or adding at most k edges, into a graph that consists of a disjoint union of cliques.

In our data reduction rules we employ two annotations for unordered vertex pairs which are defined as follows.

"*permanent*": In this case, $\{u, v\} \in E$ and it is not allowed to delete $\{u, v\}$;
"*forbidden*": In this case, $\{u, v\} \notin E$ and it is not allowed to add $\{u, v\}$.

Note that whenever we delete an edge $\{u, v\}$ from E, we make vertex pair $\{u, v\}$ *forbidden*, since it would not make sense to reintroduce previously deleted edges. In the same way, whenever we add an edge $\{u, v\}$ to E, we make $\{u, v\}$ *permanent.*