1

What Is a Kernel?

> Every block of stone has a statue inside it and it is the task of the sculptor to discover it.
>
> *(—Michelangelo Buonarroti (1475–1564))*

1.1 Introduction

Preprocessing (*data reduction* or *kernelization*) is a computation that transforms input data into "something simpler" by partially solving the problem associated with it. Preprocessing is an integral part of almost any application: Both systematic and intuitive approaches to tackle difficult problems often involve it. Even in our everyday lives, we often rely on preprocessing, sometimes without noticing it. Before we delve into the formal details, let us start our acquaintance with preprocessing by considering several examples.

Let us first look at the simple chess puzzle depicted in Fig. 1.1. In the given board position, we ask if White can checkmate the Black king in two moves. A naive approach for solving this puzzle would be to try all possible moves of White, all possible moves of Black, and then all possible moves of White. This gives us a huge number of possible moves—the time required to solve this puzzle with this approach would be much longer than a human life. However, a reader with some experience of playing chess will find the solution easily: First we move the white knight to f7, checking the black king. Next, the black king has to move to either h8 or to h7, and in both cases it is checkmated once the white rook is moved to h5. So how we are able to solve such problems? The answer is that while at first look the position on the board looks complicated, most of the pieces on the board, like white pieces on the first three rows or black pieces on the first three columns, are irrelevant to the solution. See the right-hand board in Fig. 1.1. An experienced player could see the important patterns

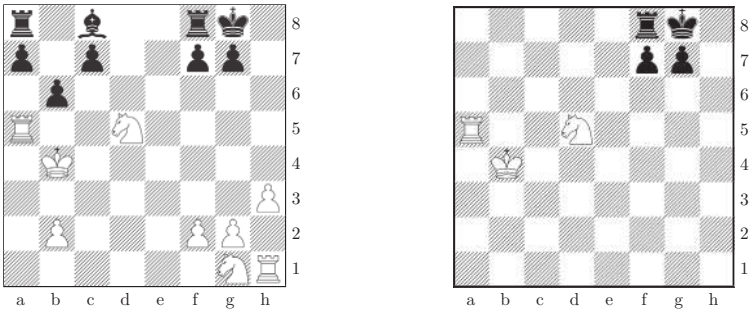

Figure 1.1 Can White checkmate in two moves? The initial puzzle and an "equivalent" reduced puzzle.

immediately, which allows the player to ignore the irrelevant information and concentrate only on the essential part of the puzzle. In this case, the player reduces the given problem to a seemingly simpler problem, and only then tries to solve it.

In this example, we were able to successfully simplify a problem by relying on intuition and acquired experience. However, we did not truly give a *sound* rule, having provable correctness, to reduce the complexity of the problem—this will be our goal in later examples. Moreover, in this context we also ask ourselves whether we can turn our intuitive arguments into generic rules that can be applied to all chess compositions. While there exist many rules for good openings, middle games and endings in chess, turning intuition into generic rules is not an easy task, and this is why the game is so interesting!

Let consider more generic rules in the context of another popular game, Sudoku (see Fig. 1.2). Sudoku is a number-placement puzzle, which is played over a 9×9 grid that is divided into 3×3 subgrids called "boxes." Some of the cells of the grid are already filled with some numbers. The objective is to fill each of the empty cells with a number between 1 and 9 such that each number appears exactly once in each row, column and box. While an unexperienced Sudoku-solver will try to use a brute force to guess the missing numbers this approach would work only for very simple examples. The experienced puzzle-solver has a number of preprocessing rules under her belt that allow her to reduce the puzzle to a state in which a brute-force approach can solve the problem within reasonable time.

Several known such preprocessing techniques solve most easy puzzles. For more difficult puzzles preprocessing is used to decrease the number of cases one should analyze to find a solution, whereas the solution is obtained by

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

4	2	***6***	5	***7***	1	***3***	9	***8***
8	***5***	***7***	2	***9***	3	***1***	***4***	6
1	3	***9***	***4***	6	***8***	***2***	7	***5***
9	***7***	1	***3***	***8***	***5***	6	***2***	***4***
5	4	***3***	***7***	***2***	***6***	***8***	1	9
6	***8***	2	***1***	***4***	***9***	7	***5***	***3***
7	9	***4***	***6***	3	***2***	***5***	8	***1***
2	***6***	***5***	8	***1***	4	***9***	***3***	7
3	1	***8***	9	***5***	7	***4***	6	***2***

Figure 1.2 A solution to a Sudoku puzzle.

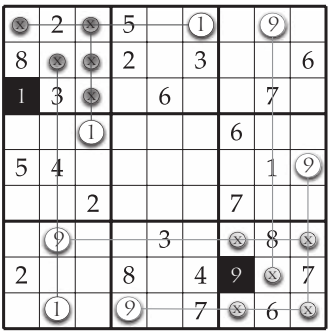

Figure 1.3 Applying the cross-hatching rule to the top-left and bottom-right boxes.

combining preprocessing with other approaches. For example, one such well-known preprocessing technique is *cross-hatching*. The cross-hatching rule is applied to 3×3 boxes. Let us look at the top-left box of the sample puzzle in Fig. 1.2. Because all numbers between 1 and 9 must appear in this box, the six empty cells should be filled with the numbers $1, 4, 5, 6, 7$ and 9. Let us attempt to find an empty cell that can be filled in with the missing number 1. To identify such a cell, we use the fact that any number can appear only once per row and once per column. As illustrated in Fig. 1.3, we thus discover a unique cell that can accommodate the number 1. In the bottom-right box, cross-hatching identifies a unique cell that can accommodate the number 9.

Although many rules were devised for solving Sudoku puzzles, none provides a generic solution to every puzzle. Thus while, for Sudoku, one can formalize what a reduction is, we are not able to predict whether reductions will solve the puzzle or even if they simplify the instance.

In both examples, to solve the problem at hand, we first simplify, and only then go about solving it. While in the chess puzzle we based our reduction solely on our intuition and experience, in the Sudoku puzzle we attempted to formalize the preprocessing rules. But is it possible not only to formalize what a preprocessing rule is but also to analyze the impact of preprocessing rigorously?

In all examples discussed so far, we did not try to analyze the potential impact of implemented reduction rules. We know that in some cases reduction rules will simplify instances significantly, but we have no idea if they will be useful for all instances or only for some of them. We would like to examine this issue in the context of NP-complete problems, which constitute a very large class of interesting combinatorial problems. It is widely believed that no NP-complete problem can be solved efficiently, that is, by a polynomial time algorithm. Is it possible to design reduction rules that can reduce a hard problem, say, by 5 percent while not solving it? At first glance, this idea can never work unless P is equal to NP. Indeed, consider for example the following NP-complete problem Vertex Cover. Here we are given an n-vertex graph G and integer k. The task is to decide whether G contains a vertex cover S of size at most k, that is a set such that every edge of G has at least one endpoint in S. Vertex Cover is known to be NP-complete. Suppose that we have a polynomial time algorithm that is able to reduce the problem to an equivalent instance of smaller size. Say, this algorithm outputs a new graph G' on $n - 1$ vertices and integer k' such that G has a vertex cover of size at most k if and only if G' has a vertex cover of size at most k'. In this situation, we could have applied the algorithm repeatedly at most n times, eventually solving the problem optimally in polynomial time. This would imply that P is equal to NP and thus the existence of such a preprocessing algorithm is highly unlikely. Similar arguments are valid for any NP-hard problem. However, before hastily determining that we have reached a dead end, let us look at another example.

In our last example, we have a set of pebbles lying on a table, and we ask if we can cover all pebbles with k sticks. In other words, we are given a finite set of points in the plane, and we need to decide if all these points can be covered by at most k lines (see Fig. 1.4). This problem is known under the name Point Line Cover. We say that the integer k is the *parameter* associated with our problem instance. If there are n points, we can trivially solve the problem by trying all possible ways to draw k lines. Every line is characterized by two points, so this procedure will require roughly $n^{\mathcal{O}(k)}$ steps.

But before trying all possible combinations, let us perform some much less time-consuming operations. Toward this end, let us consider the following simple yet powerful observation: If there is a line L covering at least $k + 1$

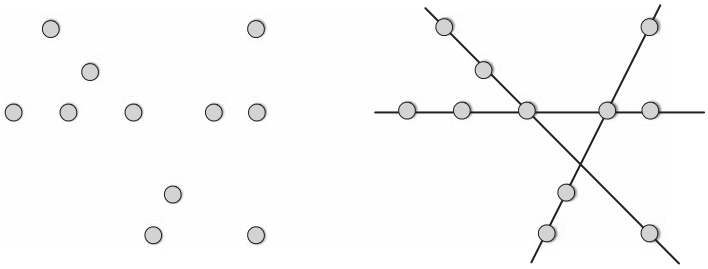

Figure 1.4 Covering all points with three lines.

points, then this line should belong to every solution (that is, at most k lines that cover all points). Indeed, if we do not use this line L, then all the points it covers have to be covered by other lines, which will require at least $k+1$ lines. Specifically, this means that if our instance has a solution, then it necessarily contains L, and therefore the instance obtained by deleting all points covered by L and decrementing the budget k by 1 also has a solution. In the other direction, it is clear that if our new instance has a solution, then the original one also has a solution. We thus conclude that solving the original problem instance is equivalent to solving the instance obtained by deleting all points covered by L and decrementing the budget k by 1. In other words, we can apply the following *reduction rule*.

Reduction Rule Reduction Rule If there is a line L covering more than k points, remove all points covered by L and decrement the parameter k by one.

This reduction rule is *sound*: The reduced problem instance has a solution if and only if the original problem instance has a solution. The naive implementation of the reduction rule takes time $\mathcal{O}(n^3)$: For each pair of points, we check if the line through it covers at least $k+1$ points. After each application of the reduction rule, we obtain an instance with a smaller number of points. Thus, after exhaustive repeated application of this rule, we arrive at one of the following situations.

- We end up having an instance in which no points are left, in which case the problem has been solved.
- The parameter k is zero but some points are left. In this case, the problem does not have solution.

- Neither of the two previous conditions is true, yet the reduction rule cannot be applied.

What would be the number of points in an irreducible instance corresponding to the last case? Because no line can cover more than k points, we deduce that if we are left with more than k^2 points, the problem does not have solution. We have thus managed, without solving the problem, to reduce the size of the problem from n to k^2! Moreover, we were able to estimate the size of the reduced problem as a function of the parameter k. This leads us to the striking realization that polynomial-time algorithms hold provable power over exact solutions to hard problems; rather than being able to find those solutions, they are able to provably reduce input sizes without changing the answer.

It is easy to show that the decision version of our puzzle problem—determining whether a given set of points can be covered by at most k lines—is NP-complete. While we cannot claim that our reduction rule always reduces the number of points by 5 percent, we are still able to prove that the size of the reduced problem does not exceed some function of the parameter k. Such a reduced instance is called a *kernel* of the problem, and the theory of efficient parameterized reductions, also known as *kernelization*, is the subject of this book.

1.2 Kernelization: Formal Definition

To define kernelization formally, we need to define what a parameterized problem is. The algorithmic and complexity theory studying parameterized problems is called *parameterized complexity*.

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

For example, an instance of Point Line Cover parameterized by the solution size is a pair (S, k), where we expect S to be a set of points on a plane encoded as a string over Σ, and k is a positive integer. Specifically, a pair (S, k) is a yes-instance, which belongs to the Point Line Cover parameterized language, if and only if the string S correctly encodes a set of points, which we will also denote by S, and moreover this set of points can be covered by k lines. Similarly, an instance of the CNF-SAT problem (satisfiability of propositional formulas in conjunctive normal form), parameterized by the number of variables, is a pair (φ, n), where we expect φ to be the input formula

encoded as a string over Σ and n to be the number of variables of φ. That is, a pair (φ, n) belongs to the CNF-SAT parameterized language if and only if the string φ correctly encodes a CNF formula with n variables, and the formula is satisfiable.

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

The notion of kernelization is tightly linked to the notion of *fixed-parameter tractability* (FPT) of parameterized problems. Before we formally define what a kernel is, let us first briefly discuss this basic notion, which serves as background to our story. Fixed-parameter algorithms are the class of exact algorithms where the exponential blowup in the running time is restricted to a small parameter associated with the input instance. That is, the running time of such an algorithm on an input of size n is of the form $\mathcal{O}(f(k)\, n^c)$, where k is a parameter that is typically small compared to n, $f(k)$ is a (typically super-polynomial) function of k that does not involve n, and c is a constant. Formally,

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

The assumption that f is a computable function is aligned with the book by Cygan et al. (2015). This assumption helps to avoid running into trouble when developing complexity theory for fixed-parameter tractability.

We briefly remark that there is a hierarchy of intractable parameterized problem classes above FPT. The main ones are the following.

$$\text{FPT} \subseteq \text{M[1]} \subseteq \text{W[1]} \subseteq \text{M[2]} \subseteq \text{W[1]} \subseteq \cdots \subseteq \text{W[P]} \subseteq \text{XP}$$

The principal analog of the classical intractability class NP is W[1]. In particular, a fundamental problem complete for W[1] is the k-Step Halting Problem for Nondeterministic Turing Machines (with unlimited nondeterminism and alphabet size). This completeness result provides an analog of Cook's theorem in classical complexity. A convenient source of W[1]-hardness reductions is provided by the result that Clique is complete for

W[1]. Other highlights of this theory are that Dominating Set is complete for W[2], and that FPT=M[1] if and only if the *exponential time hypothesis* fails. The classical reference on parameterized complexity is the book by Downey and Fellows (1999). A rich collection of books for further reading about parameterized complexity is provided in the "Bibliographic Notes" to this chapter.

Let us now turn our attention back to the notion of *kernelization*, which is formally defined as follows.

Definition 1.3 Let L be a parameterized problem over a finite alphabet Σ. A *kernelization algorithm*, or in short, a *kernelization*, for L is an algorithm with the following property. For any given $(x,k) \in \Sigma^* \times \mathbb{N}$, it outputs in time polynomial in $|(x,k)|$ a string $x' \in \Sigma^*$ and an integer $k' \in \mathbb{N}$ such that

$$((x,k) \in L \iff (x',k') \in L) \text{ and } |x'|, k' \leq h(k),$$

where h is an arbitrary computable function. If K is a kernelization for L, then for every instance (x,k) of L, the result of running K on the input (x,k) is called the kernel of (x,k) (under K). The function h is referred to as the *size* of the kernel. If h is a polynomial function, then we say that the kernel is polynomial.

We remark that in the preceding definition, the function h is not unique. However, in the context of a specific function h known to serve as an upper bound on the size of our kernel, it is conventional to refer to this function h as the size of the kernel.

We often say that a problem L admits a kernel of size h, meaning that every instance of L has a kernel of size h. We also often say that L admits a kernel with property Π, meaning that every instance of L has a kernel with property Π. For example, saying that Vertex Cover admits a kernel with $\mathcal{O}(k)$ vertices and $\mathcal{O}(k^2)$ edges is a short way of saying that there is a kernelization algorithm K such that for every instance (G,k) of the problem, K outputs a kernel with $\mathcal{O}(k)$ vertices and $\mathcal{O}(k^2)$ edges.

While the running times of kernelization algorithms are of clear importance, the optimization of this aspect is not the topic of this book. However, we remark that, lately, there is some growing interest in optimizing this aspect of kernelization as well, in particular in the design of linear-time kernelization algorithms. Here, linear time means that the running time of the algorithm is linear in $|x|$, but it can be nonlinear in k.

It is easy to see that if a decidable (parameterized) problem admits a kernelization for some function f, then the problem is FPT: For every instance of the problem, we call a polynomial time kernelization algorithm, and then we use a decision algorithm to identify if the resulting instance is valid. Because the size of the kernel is bounded by some function of the parameter, the running time of the decision algorithm depends only on the parameter. Interestingly, the converse also holds, that is, if a problem is FPT then it admits a kernelization. The proof of this fact is quite simple, and we present it here.

Theorem 1.4 *If a parameterized problem L is* FPT *then it admits a kernelization.*

Proof: Suppose that there is an algorithm deciding if $(x,k) \in L$ in time $f(k)|x|^c$ for some computable function f and constant c. On the one hand, if $|x| \geq f(k)$, then we run the decision algorithm on the instance in time $f(k)|x|^c \leq |x|^{c+1}$. If the decision algorithm outputs yes, the kernelization algorithm outputs a constant size yes-instance, and if the decision algorithm outputs no, the kernelization algorithm outputs a constant size no-instance. On the other hand, if $|x| < f(k)$, then the kernelization algorithm outputs x. This yields a kernel of size $f(k)$ for the problem. □

Theorem 1.4 shows that kernelization can be seen as an alternative definition of FPT problems. So to decide if a parameterized problem has a kernel, we can employ many known tools already given by parameterized complexity. But what if we are interested in kernels that are as small as possible? The size of a kernel obtained using Theorem 1.4 equals the dependence on k in the running time of the best known fixed-parameter algorithm for the problem, which is usually exponential. Can we find better kernels? The answer is yes, we can, but not always. For many problems we can obtain polynomial kernels, but under reasonable complexity-theoretic assumptions, there exist FPT problems that do not admit kernels of polynomial size.

Finally, if the input and output instances are associated with *different* problems, then the weaker notion of *compression* replaces the one of kernelization. In several parts of this book polynomial compression will be used to obtain polynomial kernels. Also the notion of compression will be very useful in the theory of lower bounds for polynomial kernels. Formally, we have the following weaker form of Definition 1.3.

Definition 1.5 A *polynomial compression* of a parameterized language $Q \subseteq \Sigma^* \times \mathbb{N}$ into a language $R \subseteq \Sigma^*$ is an algorithm that takes as input an instance $(x,k) \in \Sigma^* \times \mathbb{N}$, works in time polynomial in $|x| + k$, and returns a string y such that:

(i) $|y| \leq p(k)$ for some polynomial $p(\cdot)$, and
(ii) $y \in R$ if and only if $(x,k) \in Q$.

If $|\Sigma| = 2$, the polynomial $p(\cdot)$ will be called the *bitsize* of the compression.

In some cases, we will write of a polynomial compression without specifying the target language R. This means that there exists a polynomial compression into some language R.

Of course, a polynomial kernel is also a polynomial compression. We just treat the output kernel as an instance of the unparameterized version of Q. Here, by an *unparameterized version* of a parameterized language Q we mean a classic language $\tilde{Q} \subseteq \Sigma^*$ where the parameter is appended in unary after the instance (with some separator symbol to distinguish the start of the parameter from the end of the input). The main difference between polynomial compression and kernelization is that the polynomial compression is allowed to output an instance of *any* language R, even an undecidable one.

When R is reducible in polynomial time back to Q, then the combination of compression and the reduction yields a polynomial kernel for Q. In particular, every problem in NP can be reduced in polynomial time by a deterministic Turing machine to any NP-hard problem. The following theorem about polynomial compression and kernelization will be used in several places in this book.

Theorem 1.6 *Let $Q \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized language and $R \subseteq \Sigma^*$ be a language such that the unparameterized version of $Q \subseteq \Sigma^* \times \mathbb{N}$ is* NP*-hard and $R \subseteq \Sigma^*$ is in* NP*. If there is a polynomial compression of Q into R, then Q admits a polynomial kernel.*

Proof: Let (x,k) be an instance of Q. Then the application of a polynomial compression to (x,k) results in a string y such that $|y| = k^{\mathcal{O}(1)}$ and $y \in R$ if and only if $(x,k) \in Q$. Because $\tilde{Q}$ is NP-hard and R is in NP, there is a polynomial time many-to-one reduction f from R to $\tilde{Q}$. Let $z = f(y)$. Because the time of the reduction is polynomial in the size of y, we have that it runs in time $k^{\mathcal{O}(1)}$ and hence $|z| = k^{\mathcal{O}(1)}$. Also we have that $z \in \tilde{Q}$ if and only if $y \in R$. Let us remind that z is an instance of the unparameterized version of Q, and thus we can rewrite z as an equivalent instance $(x',k') \in Q$. This two-step polynomial-time algorithm is the desired kernelization algorithm for Q. □

Two things are worth a remark. Theorem 1.6 does not imply a polynomial kernel when we have a polynomial compression in a language that is not in NP. There are examples of natural problems for which we are able to obtain