Chapter 6

NP-completeness

In this chapter, we introduce the complexity classes that are of paramount importance for algorithm designers: P, NP, and NPC. We take a strictly practical approach and determinedly skip the detour through Turing machines. In other words, we limit ourselves to NP-completeness, explaining its importance and detailing how to prove that a problem is NP-complete.

After introducing our approach in Section 6.1, we define the complexity classes P and NP in Section 6.2. NP-complete problems are introduced in Section 6.3, along with the practical reasoning to prove that a problem is NP-complete. Several examples are provided in Section 6.4. We discuss subtleties in problem definitions in Section 6.5 and strong NP-completeness in Section 6.6. Finally, we make our conclusions in Section 6.7.

6.1 A practical approach to complexity theory

This chapter introduces the key complexity classes that algorithm designers are confronted with: P, which stands for *Polynomial*, and NP, which stands for *Nondeterministic Polynomial*. In fact, we depart from the original definition of the class NP and use the (equivalent) characterization *Polynomial with Certificate*. Within the NP class, we focus on the subclass NPC of *NP-complete* problems.

When writing this chapter, we faced a cruel dilemma. Either we use a formal approach, which requires an introduction to Turing machines, explain their characteristics, and classify the languages that they can recognize, or we use a practical approach that completely skips the detour through the theoretical computer science framework and defines complexity classes out of nowhere (almost!). We firmly believe that there is no trade-off in between, and that a comprehensive exposure does require Turing machines. However, given the main objectives of this book, we chose the latter approach. The price to pay is that the reader will have to take for granted a key result, namely Cook's theorem [25], which we will state without proof. Cook's theorem provides the first NP-complete problem, and we will have to trust him on this. However, the main advantage is that we can concentrate on the art of the algorithm designer, namely *polynomial reduction*.

First, why Turing machines? To assess the complexity of a problem, we need to define its size and the number of time steps required to solve it. But what is appropriate within a time step? A formal answer relies on Turing machines. The size of a problem is the number of consecutive positions used to store its data on the (infinite) ribbon of the machine. The number of time steps is the number of moves before the Turing machine terminates the execution of its program, given the data initially stored on its ribbon. Instead, in the practical approach, we simply define the size of a problem as the number of memory locations, or bits, that are needed to store its data, and we define a time step as the maximum time needed to execute an elementary operation. Here, an elementary operation is defined as any *reasonable* computation. And, the trouble begins. Fetching the values of two memory locations, adding them and storing them back into some memory location, is that an elementary operation? Yes—well, provided that the access to the memory locations takes constant time, which may require that the total memory is bounded, or at least that two different memory locations used to solve the problem are not too far apart in storage. We are not far from moving the head of the Turing machine from one position to another! Similarly, adding two bits or two bytes or two double-precision floating point numbers (64 bits) is indeed an elementary operation, but adding two integers of unbounded length is not. In fact, an elementary operation is anything that can be done in polynomial time by a Turing machine, but this statement is helpful mostly to those who are familiar with Turing machines. Here is an example of an operation that is not reasonable. If we have two prime numbers p and q of r bits, we can compute their product $n = p \times q$ in $O(r^2)$, but given n, we cannot find p and q in time polynomial in r.

We refer the reader interested in the formal approach to some excellent books. The big classic is the book by Garey and Johnson [38] with a comprehensive treatment of NP-completeness. A very intuitive proof of Cook's theorem is given by Wilf [108]. More on complexity theory is provided by Papadimitriou [82].

6.2 Problem classes

In this section, we first emphasize the importance of polynomials in the theory. Then, we discuss how to define the problem size and how to encode data. This is illustrated through classical examples; integers are coded in a logarithmic size, but we should be careful if objects must be enumerated (set of nodes in a graph, list of tasks, etc).

6.2.1 Problems in P

The following remark, admittedly simple, is fundamental: *The composition of two polynomials is a polynomial.* Thanks to this observation, key values (time, size) can be defined up to a polynomial factor. From the point of view of complexity classes, values like n, n^3, or $n^{27} + 17n^5 + 42$ are totally equivalent; all these values are polynomial in n. Hence, there is no difference if an elementary operation of the algorithm would cost n^3 or $n^{27}+17n^5+42$ time steps of a Turing machine; as long as there is a polynomial number of such operations, the total number of time steps for the Turing machine remains polynomial.

The theory deals with decision problems, with a yes/no answer, rather than with optimization problems (this is related to languages that are accepted by Turing machines). A decision problem is in the complexity class P if it can be solved in *polynomial time*. Owing to the previous remark, we do not need to specify the degree of the polynomial, which is not relevant as far as theory is concerned (we come back to this last point below). Hence, the key for understanding this class P is the notion of "polynomial time." As mentioned before, one must decide what can be done within one unit of time. One usually assumes that one can add, multiply, or access memory in constant time, but the multiplication of large numbers (respectively, memory accesses) can depend on the size of the numbers (respectively, of the memory).

From an algorithmic point of view, we usually suppose that we can add, multiply, access memory within one unit of time, as long as numbers and memory size are bounded, which seems reasonable. These operations are then of polynomial time, and thus this model is polynomial with respect to the theoretical one with the Turing machine, as long as we are careful when dealing with nonbounded integers.

Also, the resolution time must be a polynomial of the *data size*, so one needs to define this "data size" carefully. This data size can strongly depend on the way an instance is encoded. Intuitively, integers can be coded in binary, therefore requiring a logarithmic size rather than a linear one (when encoded in unary). The encoding with any other basis $b \neq 2$ has the same size as the binary encoding, up to a constant factor ($\log_2(n)/\log_b(n) = 1/\log_b(2)$). However, some integers describing a problem instance should not be encoded in binary when they code objects to be enumerated. Otherwise, some "elementary" operations would have a cost exponential in the data size. We illustrate this by detailing two problem examples.

Example: 2-partition

DEFINITION 6.1 (2-PARTITION)**.** Given n positive integers $a_1, \dots, a_n$, is there a subset I of $\{1, \dots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

The input data of a problem instance is a set of n integers. In theory, these n integers could be encoded either in unary or in binary. However, by convention, in complexity theory, any integer appearing in the coding of an instance must be encoded in binary. The only exception is for data whose encoding in unary would not change the overall data size of the instance (i.e., if the new data size remains polynomial in the original data size). For instance, for 2-PARTITION, the choice of encoding for the value n itself does not matter because encoding n integers requires a data size of at least one per integer and thus of at least n. Therefore, for the sake of simplicity, one usually encodes n in unary. Then, with the mandatory binary encoding of the n integers, the data size is $\sum_{1 \leqslant i \leqslant n} \log(a_i)$. With a unary encoding of the integers, the data size of an instance would have been $\sum_{1 \leqslant i \leqslant n} a_i$.

The choice of the encoding is vital for such a problem. Indeed, one can find an algorithm whose time is polynomial in $n \times \sum_{1 \leqslant i \leqslant n} a_i$. We design a simple dynamic-programming algorithm; we solve the problems $c(i, T)$, where $c(i, T)$ equals true if there is a subset of $\{a_1, \ldots, a_i\}$ of sum T (and false otherwise), for $1 \leqslant i \leqslant n$ and $0 \leqslant T \leqslant S = \sum_{1 \leqslant i \leqslant n} a_i$. The solution to the original problem is $c(n, \frac{S}{2})$. The recurrence relation is $c(i, T) = c(i-1, T-a_i) \vee c(i-1, T)$. This algorithm is in $O(nS)$. Therefore, this algorithm runs in a time that would be polynomial in the size of the data if we had allowed the integers to be coded in unary. However, the algorithm running time is exponential in the data size when integers are coded in binary, as mandated. Such an algorithm is said to be *pseudopolynomial*.

No one knows an algorithm that is polynomial in the data size (i.e., in $O(n \log(S))$), so the question whether the 2-PARTITION problem is or isn't in P is left open.

Example: Bipartite graphs

DEFINITION 6.2 (BIPARTITE)**.** Given a graph $G = (V, E)$, is G a bipartite graph?

This is a decision problem; the answer must be yes or no. The input data of a problem instance is a graph (V, E), where V is the set of vertices and E the set of edges. The size of the data depends on how the graph is stored (or encoded). The graph consists of $|V| = n$ vertices. One usually codes n in unary rather than in binary. Independent vertices are vertices that are not endpoints of any edge. Independent vertices play a trivial role with respect to the problem, and they can therefore be safely discounted. Then, each vertex is the endpoint of at least one edge, $|E| \geqslant n/2$, and encoding n in binary does not change the data size. Therefore, for the sake of simplicity, in any graph problem the number of vertices is always encoded in unary for the same reason.

Then, the identifier of a vertex can be encoded in binary, thus in $\log(n)$ for one vertex leading to a total of $n \log(n)$, which is still polynomial in n. The number of edges is also polynomial in n because there are at most n^2 edges. Then, altogether, the total size of the problem data is a polynomial in n, where n is the number of vertices of the graph. When designing graph algorithms, one often denotes the size of data of a graph as $|V| + |E|$ (strictly speaking, it should be $|V| + |E|\log(|V|)$, but each expression is polynomial in the other one). This allows us to refine the cost study of the algorithms, in particular when $|E| \ll |V|^2$. However, $|V| + |E|$ is still polynomial in n, so this refinement does not alter the problem classification.

Now, given a graph, in order to answer the question (yes or no), we need to perform a number of operations that is polynomial in n (greedy graph coloring). This problem is, therefore, in the complexity class P because it can be solved in a time that is polynomial in the data size.

6.2.2 Problems in NP

To define the complexity class NP, we need to define the *certificate* of a problem, which is (an encoding of) a solution to the problem.

Problem solution: Certificate

Back to the 2-PARTITION problem, if we are given a subset $I \subseteq \{1, \dots, n\}$, we can check in polynomial time (even in linear time) whether $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$, and, therefore, we can answer whether the problem has a positive answer in polynomial time. Moreover, the size of the certificate I is $O(n\log(n))$, which is polynomial in the problem size (the certificate contains $O(n)$ identifiers, each coded on $O(\log(n))$ bits).

Another way to provide a solution to 2-PARTITION would be to give the certificate $\{a_i\}_{i \in I}$, but if the a_is are coded in unary in the certificate, it is of exponential size. However, if the a_is are coded in binary, then the certificate has polynomial size and it is perfectly acceptable; a certificate is valid if it is polynomial in the problem size.

For the bipartite graph problem, the certificate would be the set of indices of vertices of one of the two subsets of the graph, whose size is polynomial in the problem size. Given this set, it is then easy to check that it is a correct solution by looking at each edge of the graph, which takes a polynomial time.

(See also Section 6.4.4 on scheduling problems for an illustration of the care that must be taken to design a certificate of polynomial size.)

Definition of NP

We are now ready to define the problem class NP. This is the class of decision problems for which we can verify a certificate in a time that is polynomial in the problem size. By *verify*, we mean *check that the certificate is indeed a solution*, i.e., that the answer to the problem is *yes*. Both previous examples are, therefore, in NP because, if we are given a certificate of polynomial size, we can check in polynomial time whether it is a solution to the problem.

We make a short digression to explain that NP stands for *Nondeterministic Polynomial*, for reference to nondeterministic Turing machines that were originally used to define the class. As already mentioned, we define NP as *Polynomial with Certificate* in this book, and we ignore equivalent characterizations of the NP class, either older (nondeterministic Turing machines) or newer (the famous PCP theorem). (See [4] for more information.)

It is time to recapitulate; we have defined two classes of decision problems:

P: Given an instance I of the problem of size $|I|$ when encoded in binary, there is an algorithm whose running time is polynomial in $|I|$ and which reports whether the instance has a solution or not;

NP: Given an instance I of the problem of size $|I|$ when encoded in binary, and a certificate of size polynomial in $|I|$, there is an algorithm whose running time is polynomial in $|I|$ and which reports whether the certificate is indeed a solution to the instance.

We observe that P $\subseteq$ NP. If we can find a solution in polynomial time, then we can verify the solution in polynomial time, with an empty certificate. Most researchers believe that the inclusion is strict, i.e., P $\neq$ NP, because it should be easier to check whether a certificate is a solution to the problem than to find a solution to that problem. As you may have heard before, this question is open at the time of this writing.

We have already seen that BIPARTITE is in P; therefore, it is in NP. Also, 2-PARTITION is in NP, but we do not know whether it is in P or not. Below are a few more examples to illustrate the class NP.

Examples: Problems in NP

DEFINITION 6.3 (COLOR)**.** Given a graph $G = (V, E)$ and an integer k $(1 \leqslant k \leqslant |V|)$, can we color G with at most k colors?

This is a graph coloring problem; two vertices connected with an edge cannot be assigned the same color. The size of the data is a polynomial in $|V| + \log(k)$. Indeed, we need to enumerate all vertices similarly to the problem bipartite, hence the term $|V|$, while the integer k is encoded in binary. Since $k \leqslant |V|$ (one never needs more colors than vertices), the size of the data is a polynomial in $|V|$. A certificate can be the list of the vertices together with their color, whose size is linear in the size of the problem instance. The verification would amount to checking that no two adjacent vertices are assigned the same color, and that no more than k colors are used in total, which can be done in linear time as well.

DEFINITION 6.4 (HC – Hamiltonian Cycle)**.** Given a graph $G = (V, E)$, is there a cycle that goes through each vertex once and only once?

Similarly to other graph problems, the size of the data is a polynomial in $|V|$. A certificate can be the ordered list of the vertices that constitute the cycle (with linear size again). As before, the verification is easy: Check that the cycle is built with existing edges in the graph, and that each vertex is visited once and only once.

DEFINITION 6.5 (TSP – Traveling Salesman Problem)**.** Given a complete graph $G = (V, E)$, a cost function $w : E \to \mathbb{N}$ and an integer k, is there a cycle $\mathcal{C}$ going through each vertex once and only once, with $\sum_{e \in \mathcal{C}} w(e) \leqslant k$?

This classical traveling salesman problem is a weighted version of the HC problem. There are several variants of the problem with various constraints on the cost function w: The weights can be arbitrary, satisfy the triangular inequality, or correspond to the Euclidean distance. The variants do not change the problem complexity. The size of the data is a polynomial in $|V| + \sum_{e \in E} \log(w(e)) + \log(k)$. We need to enumerate vertices, and other integers are coded in binary. A certificate can be the ordered list of the vertices that constitute the cycle, and the verification is similar to that for the HC problem.

No one knows how to find a solution to these three problems in polynomial time.

Problems not in NP?

One rarely encounters a problem whose membership status, with respect to NP, is unknown. It is even rarer to come across a problem that is known not to belong to NP. These problems are usually not very interesting from an algorithmic point of view. They are, however, fundamental for the theory of complexity. We provide a few examples below.

Negation of TSP: Given a problem instance of TSP, is it true that there is no cycle in the graph of length $|V|/2$?

This problem is similar to TSP, but the question is asked in the reverse way. It is difficult to think of a certificate of polynomial size that would allow us to check in polynomial time that the answer to the question is yes. Whether this problem belongs to NP is an open question.

Square: Given n squares whose areas sum up to 1, can we partition the unit square into these n squares?

We are interested in this problem because it plays a prominent role in the case study of Chapter 14. Its complexity depends on the exact definition that is used. First, we give the variant of the problem that is used in Chapter 14; we are given n squares of size a_i, with $\sum_{1 \leqslant i \leqslant n} a_i^2 = 1$. The a_i are rational numbers, $a_i = b_i/c_i$, and the problem size is $\sum_{1 \leqslant i \leqslant n} \log(b_i) + \sum_{1 \leqslant i \leqslant n} \log(c_i)$. A certificate can be the position of each square, for $1 \leqslant i \leqslant n$, for instance the coordinates of its top left corner. This certificate is of polynomial size. We can then check in polynomial time whether it is a solution of the problem or not (however, writing such a verification procedure requires some care). Hence, this variant is in NP.

Another variant consists of having as input m_i squares of size a_i, for $1 \leqslant i \leqslant p$, with $n = \sum_{1 \leqslant i \leqslant p} m_i$ and $\sum_{1 \leqslant i \leqslant p} m_i a_i^2 = 1$. The size of the data is then $n + \sum_{1 \leqslant i \leqslant p} \log(m_i) + \sum_{1 \leqslant i \leqslant p} \log(b_i) + \sum_{1 \leqslant i \leqslant p} \log(c_i)$. We do not need to enumerate all squares but only the p basic squares, while the m_is can be coded in binary. Then, in a certificate of polynomial size, we cannot enumerate all the n squares to give their coordinates. There might exist a compact analytical formula that would characterize solutions (say, the j-th square of size a_i is placed at coordinates $f(i, j)$), but this is far from being obvious. We do not know whether this latter variant is in NP or not.

It is much harder to identify a problem that is known not to be in NP, at least without making any assumption like P $\neq$ NP. A (complicated) example is the problem of deciding whether two regular expressions represent different languages, where the expressions are limited to four operators: union, concatenation, the Kleene star (zero or more copies of an expression), and squaring (two copies of an expression). Any algorithm for this problem requires exponential space, hence, exponential verification time [77].

Another problem that is not in NP is the program termination problem, or halting problem (decide whether a program will terminate on a given input). However, this example is a little excessive because no algorithm can exist to solve it, regardless of its complexity [82].

6.3 NP-complete problems and reduction theory

As explained in the previous section, we do not know whether the inclusion P $\subseteq$ NP is strict or not. However, we are able to compare the complexity of problems in NP; Cook's idea was to prove that some problems of the NP class are at least as difficult as all other problems of the same class. These problems are called NP-complete and form the subclass NPC of the class NP. They are *the most difficult problems* of NP. If we are able to solve one NP-complete problem in polynomial time, then we will be able to solve all problems of NP in polynomial time, and we will have P = NP. The main objective of this section is to explain this line of reasoning in full detail and to explore some consequences.

We detail the theory of reduction, which aims at proving that a problem is *more difficult* than another one. However, if we want to prove that a problem is more difficult than any other one, we need to identify the first NP-complete problem, as explained in Section 6.3.2. Note that a set of NP-complete problems with the corresponding reductions is presented in Section 6.4.

6.3.1 Polynomial reduction

We start by explaining the mechanism of polynomial reduction, i.e., how to prove that a problem is more difficult than another. Consider two decision problems P_1 and P_2. How can we prove that P_1 is more difficult than P_2? We say that P_2 is polynomially reducible to P_1 and write $P_2 \xrightarrow{pr} P_1$ if, whenever we are given an instance $\mathcal{I}_2$ of problem P_2, we can convert it, with only a polynomial-time algorithm, into an instance $\mathcal{I}_1$ of P_1, in such a way that $\mathcal{I}_2$ has the answer "Yes" if and only if $\mathcal{I}_1$ has the answer "Yes."

Now, if P_2 is polynomially reducible to P_1, then P_1 must be more difficult than P_2 (or more precisely, at least as difficult as P_2). Indeed, if there exists a polynomial algorithm to solve P_1, then by applying the polynomial reduction, and because the composition of two polynomials is a polynomial, there exists a polynomial algorithm to solve P_2. Given an instance $\mathcal{I}_2$ of P_2, we can indeed convert it into instance $\mathcal{I}_1$ of P_1, and since there is an equivalence between solutions of $\mathcal{I}_1$ and $\mathcal{I}_2$, the polynomial algorithm for P_1 executed on instance $\mathcal{I}_1$ returns the solution for instance $\mathcal{I}_2$. Take the contrapositive of this statement. If there is no polynomial algorithm to solve P_2, then there is none to solve P_1 either, so P_1 is more difficult.

We point out that polynomial reduction is a transitive operation: If $P_3 \xrightarrow{pr} P_2$ and $P_2 \xrightarrow{pr} P_1$, then $P_3 \xrightarrow{pr} P_1$. Again, this is because the composition of two polynomials is a polynomial, nothing more.

Note also that it is mandatory to have the equivalence of solutions, i.e., if $\mathcal{I}_1$ has a solution then $\mathcal{I}_2$ has one, and if $\mathcal{I}_2$ has a solution then $\mathcal{I}_1$ has one. Otherwise, the polynomial reduction $P_2 \xrightarrow{pr} P_1$ would not imply that P_1 is more difficult than P_2.

6.3.2 Cook's theorem

The fundamental result of the P versus NP theory is Cook's theorem [25], which shows that the satisfiability problem SAT is the most difficult problem in NP. This means that all other problems in NP are polynomially reducible to SAT. We introduce SAT and give a brief intuitive sketch of Cook's proof.

DEFINITION 6.6 (SAT)**.** Let F be a Boolean formula with n variables $x_1, \ldots, x_n$ and p clauses $C_1, \ldots, C_p$: $F = C_1 \wedge C_2 \wedge \cdots \wedge C_p$, with, for $1 \leqslant i \leqslant p$, $C_i = x^*_{i_1} \vee x^*_{i_2} \vee \cdots \vee x^*_{i_{f(i)}}$, $1 \leqslant i_k \leqslant n$ for $1 \leqslant k \leqslant f(i)$, and $x^* = x$ or $\overline{x}$. Does there exist an instantiation of the n variables such that F is true (i.e., C_i is true for $1 \leqslant i \leqslant p$)?

Clearly, SAT is in NPC, and a certificate can simply be the list of the instantiation of each variable (whether a given x_i is instantiated to true or false). However, it seems difficult to solve SAT without a certificate; because some clauses have x_i and other $\overline{x_i}$, we may have to try all 2^n possible instantiations to find one that satisfies the formula. In other words, SAT seems to be a hard problem indeed.

Cook's theorem states that all problems in NP are polynomially reducible to SAT. The main idea of the proof is the following: Consider any problem P in NP, and take an arbitrary instance I, together with its certificate C. The proof goes by simulating the execution of the Turing machine that accepts the couple (I, C) as input and outputs "Yes" after a polynomial number of steps.

Because Turing machines are simple, their behavior can be characterized by clauses linking a set of variables. We can define $x_{t,j,s}$ as a variable that is true if after t steps of computation, symbol s is in position j of the ribbon, and we can simulate the operation of the machine using these variables. There are many such variables, but only a polynomial number in $|I|$, and a polynomial number of clauses as well. A detailed, but easy-to-follow, proof is given by Wilf [108].

6.3.3 Growing the class NPC of NP-complete problems

Now that we have the first NP-complete problem handy, how can we find more? To prove that a problem, P_1, is in NPC, we merely have to prove that SAT is polynomially reducible to this problem. Indeed, by composition, all problems in NP are reducible to SAT, hence, to P_1. The reduction takes several steps:

1. Prove that $P_1 \in NP$: We must be able to build a certificate of polynomial size, and then, for any instance $\mathcal{I}_1$ of problem P_1, we must be able to check in polynomial time whether the certificate is a solution. Usually, this first step is easy, but it should not be forgotten.
2. Prove the completeness of P_1: We transform an arbitrary instance $\mathcal{I}$ of SAT into an instance $\mathcal{I}_1$ of P_1 in polynomial time, and such that:
   (a) the size of $\mathcal{I}_1$ is polynomial in the size of $\mathcal{I}$;
   (b) $\mathcal{I}_1$ has a solution $\Leftrightarrow$ $\mathcal{I}$ has a solution.

Let us come back to the construction of instance $\mathcal{I}_1$. The construction should be done in polynomial time, but this is usually implicit because the size of $\mathcal{I}_1$ should be polynomial in the size of $\mathcal{I}$, and because we perform only "reasonable" operations.

Assume that we have polynomially reduced SAT to P_1. We now have two problems in NPC, namely, P_1 and SAT. If we want to extend the class to a third problem, $P_2 \in NP$, should we reduce SAT or P_1 to P_2? Of course, the answer is that either reduction works. Indeed, we have so far:

- $P_1 \xrightarrow{pr} SAT$ and $P_2 \xrightarrow{pr} SAT$ (both by Cook's theorem).
- $SAT \xrightarrow{pr} P_1$ (our previous reduction).

We can prove that $SAT \xrightarrow{pr} P_2$ either directly or via the reduction $P_1 \xrightarrow{pr} P_2$ because $SAT \xrightarrow{pr} P_1$ and because, as we have already stated, polynomial reduction is a transitive operation.

In other words, to show that some problem P_2 in NP is in NPC, we can pick any NP-complete problem P_1 in NPC and show that $P_1 \xrightarrow{pr} P_2$. This will show that P_2 is in NPC, and P_2 will itself become a candidate NP-complete problem to pick up for later reductions.

A decision problem is said to be NP-hard when it can be polynomially reduced from an NP-complete problem, but it is not known whether it belongs to NP.

6.3.4 Optimization problems versus decision problems

We have been focusing so far on decision problems, but, in many practical situations, we have to solve an optimization problem in which we want to maximize or minimize a given criterion. Optimization problems (also called search problems) are more complex than decision problems, but one can always restrict an optimization problem so that it becomes a decision problem.

For instance, the graph coloring problem is usually an optimization problem: What is the minimum number of colors required to color the graph? The restriction to the decision problem is the COLOR problem of Definition 6.3: Can we color the graph with at most k colors? If we can solve the optimization problem, we have immediately the solution to the decision problem, for any value of k. In this particular case, we also can go the other way round. If we are able to solve the decision problem, then we can find the answer to the optimization problem by performing a binary search on k ($1 \leqslant k \leqslant |V|$) and computing the answer of the decision problem for each value of k. The binary search adds a factor $\log(|V|)$ to the algorithm complexity, so that if we had a polynomial algorithm, it remains polynomial. The two problems (optimization and decision) have the same complexity. In most cases, the optimization problem can be solved using a binary search as described above. However, this result is not always true; it can be difficult to find the answer to the optimization problem, even though we can solve the decision problem. In some extreme situations, there may be no solution to the optimization problem. For instance, there is no solution to the problem "Find the smallest rational number x such that $x^2 \geqslant 2$" because $\sqrt{2}$ is irrational, while it is easy to solve the decision problem in polynomial time: "Given a rational number x, do we have $x^2 \geqslant 2$?" (simply compute a square and compare it to 2).

Transforming a decision problem into an optimization problem may not be natural or even possible. However, we can always define the *associated* decision problem of an optimization problem: If the optimization problem aims at minimizing a value x with some constraints, the decision problem adds a value x_0 as an input to the problem, and the question is whether there is a solution achieving a value $x \leqslant x_0$.

A typical example is based on 2-PARTITION. Consider the scheduling problem with two processors, where we want to schedule n tasks of length a_i. Ideally, we want to 2-partition the tasks so that the execution finishes as soon as possible, but if this is not possible, we minimize the difference of finish times, which amounts to minimizing the global finish time. Formally, the objective is to find a subset I that minimizes $x = |\sum_{i \in I} a_i - \sum_{i \notin I} a_i|$. The associated decision problem with target value $x_0 = 0$ is exactly 2-PARTITION, that will be shown to be NP-complete (Exercise 7.20, p. 155). By misuse of language, we say that an optimization problem is NP-complete if the associated decision problem for some well-chosen target value is NP-complete. Hence, the scheduling problem with two processors as defined above is NP-complete.

6.4 Examples of NP-complete problems and reductions

At this point, we know that SAT is NP-complete. As already discussed, we proceed by reduction to increase the list of NP-complete problems. In this section, we show that 3-SAT, CLIQUE, and VERTEX-COVER are in NPC. We also give references for the NP-completeness of 2-PARTITION, HC (Hamiltonian Cycle), and then show that TSP (Traveling Salesman Problem) is NP-complete.

6.4.1 3-SAT

DEFINITION 6.7 (3-SAT)**.** Let F be a Boolean formula with n variables $x_1, \dots, x_n$ and p clauses $C_1, \dots, C_p$: $F = C_1 \wedge C_2 \wedge \cdots \wedge C_p$, with, for $1 \leqslant i \leqslant p$, $C_i = x^*_{i_1} \vee x^*_{i_2} \vee x^*_{i_3}$, $1 \leqslant i_k \leqslant n$ for $1 \leqslant k \leqslant 3$, and $x^* = x$ or $\overline{x}$. Does there exist an instantiation of the variables such that F is true (i.e., C_i is true for $1 \leqslant i \leqslant p$)?

This problem is the restriction of SAT to the case where each clause consists of three variables, i.e., following the notations of Section 6.3.2, $f(i) = 3$ for $1 \leqslant i \leqslant p$. In fact, 3-SAT is so close to SAT that one might wonder why consider 3-SAT in addition to, or replacement of, SAT. The reason is that it is much easier to manipulate clauses with exactly three variables. Furthermore, proving the NP-completeness of 3-SAT is also a good exercise for our first reduction.

THEOREM 6.1. *3-SAT is NP-complete.*

Proof. This proof, as well as the next ones, follows the reduction method to prove that a problem is NP-complete.

First, we prove that 3-SAT is in NP. We can simply claim that it is in NP because it is a restriction of SAT, which itself is in NP. It also is easy to prove it directly. We consider an instance I of 3-SAT, which is of size $O(n + p)$. A certificate is a set of truth values, one for each variable. Therefore, it is of size $O(n)$, which is polynomial in the size of the instance. It is easy to check whether the certificate is a solution, and this takes a time $O(n + p)$. Altogether, 3-SAT is in NP.

To prove the completeness, we reduce an instance of SAT. So far, it is the only problem that we know to be NP-complete, thanks to Cook's theorem, so we have no choice.

Let $\mathcal{I}_1$ be an instance of SAT. First, we need to build an instance $\mathcal{I}_2$ of 3-SAT that will have a solution if and only if $\mathcal{I}_1$ has one. $\mathcal{I}_1$ consists of p clauses $C_1, \dots, C_p$, of lengths $f(1), \dots, f(p)$, and each clause is made of some of the n variables $x_1, \dots, x_n$.

Instance $\mathcal{I}_2$ initially consists of the n variables $x_1, \dots, x_n$. Then, we add to $\mathcal{I}_2$ variables and clauses corresponding to each clause C_i of $\mathcal{I}_1$. We build a set of clauses made of exactly three variables, and the goal is to have the equivalence between C_i and the constructed clauses. We consider various cases:

- If C_i has a single variable x, we add to instance $\mathcal{I}_2$ two new variables a_i and b_i and four clauses: $x \vee a_i \vee b_i$, $x \vee \overline{a_i} \vee b_i$, $x \vee a_i \vee \overline{b_i}$, and $x \vee \overline{a_i} \vee \overline{b_i}$.
- If C_i has two variables $x_1 \vee x_2$, we add to instance $\mathcal{I}_2$ one new variable c_i and two clauses: $x_1 \vee x_2 \vee c_i$ and $x_1 \vee x_2 \vee \overline{c_i}$.
- If C_i has three variables, we add it to $\mathcal{I}_2$.
- If C_i has k variables, with $k > 3$, $C_i = x_1 \vee x_2 \vee \cdots \vee x_k$, then we add $k-3$ new variables $z_1^i, z_2^i, \dots, z_{k-3}^i$ and $k-2$ clauses: $x_1 \vee x_2 \vee z_1^i$, $x_3 \vee \overline{z_1^i} \vee z_2^i$, ..., $x_{k-2} \vee \overline{z_{k-4}^i} \vee z_{k-3}^i$, and $x_{k-1} \vee x_k \vee \overline{z_{k-3}^i}$.

Note that all clauses that are added to $\mathcal{I}_2$ are exactly made of three variables, and that the construction is done in polynomial time. Then, we must check the different points of the reduction.

First, note that $size(\mathcal{I}_2)$ is polynomial in $size(\mathcal{I}_1)$ (and even linear); indeed, $size(\mathcal{I}_2) = O(n + \sum_{i=1}^{p} f(i))$.

Then, we start with the easy side, which consists of proving that if $\mathcal{I}_1$ has a solution, then $\mathcal{I}_2$ has a solution. Let us assume that $\mathcal{I}_1$ has a solution. We have an instantiation of variables $x_1, \dots, x_n$ such that C_i is true for $1 \leqslant i \leqslant p$. Then, a solution for $\mathcal{I}_2$ keeps the same values for the x_is, and set all a_j, b_j and c_j values to true. Therefore, if a clause with at most three variables is true in $\mathcal{I}_1$, all corresponding clauses in $\mathcal{I}_2$ are true. Consider now a clause C_i in $\mathcal{I}_1$ with $k > 3$ variables: $C_i = x_1 \vee x_2 \vee \cdots \vee x_k$. Let x_j be the first variable of the clause that is true. Then, for the solution of $\mathcal{I}_2$, we instantiate $z_1^i, \dots, z_{j-2}^i$ to true and $z_{j-1}^i, \dots, z_{k-3}^i$ to false. With this instantiation, all clauses of $\mathcal{I}_2$ are true, and thus $\mathcal{I}_1 \Rightarrow \mathcal{I}_2$.

For the other side, let us assume that $\mathcal{I}_2$ has a solution. We have an instantiation of all variables x_i, a_i, b_i, c_i, and z_j^i that is a solution of $\mathcal{I}_2$. Then, we prove that the same instantiation of $x_1, \dots, x_n$ is a solution of the initial instance $\mathcal{I}_1$. First, for a clause with one or two variables, whatever the values of a_i, b_i, and c_i, we necessarily have x or $x_1 \vee x_2$ equal to true because we have added clauses constraining the extra variables. The clauses with three variables remain true since we have not modified them. Finally, let C_i be a clause of $\mathcal{I}_1$ with $k > 3$ variables, $C_i = x_1 \vee x_2 \vee \cdots \vee x_k$. We reason by contradiction. If this clause is false, then, necessarily, because of the first clause added to $\mathcal{I}_2$ when processing clause C_i, z_1^i must be true, and similarly we can prove that all z_j^i variables must be true. The contradiction arises for the last clause because it imposes that $\overline{z_{k-3}^i}$ should be true if x_{k-1} and x_k are both false. Therefore, by contradiction, at least one of the x_js must be true and the clause of $\mathcal{I}_1$ is true. We finally have $\mathcal{I}_2 \Rightarrow \mathcal{I}_1$, which concludes the proof. □

As a final remark, we point out that not all restrictions of a given NP-complete problem remain NP-complete. For instance, 2-SAT, the SAT problem where each clause contains exactly two variables, belongs to P. Several variants of 3-SAT are shown NP-complete in the exercises.

6.4.2 CLIQUE

We now consider a problem that is very different from SAT.

DEFINITION 6.8 (CLIQUE)**.** Let $G = (V, E)$ be a graph and k be an integer such that $1 \leqslant k \leqslant |V|$. Does there exist a clique of size k (i.e., a complete subgraph of G with k vertices)?

This is a graph problem, and the size of the instance is polynomial in $|V|$ (recall that $|E| \leqslant |V|^2$, so we do not need to consider $|E|$ in the instance size).

THEOREM 6.2. *CLIQUE is NP-complete.*

Proof. First we prove that CLIQUE is in NP. The certificate is the list of vertices of a clique, and we can check in polynomial time (even quadratic time) whether it is a clique or not. For each vertex pair of the certificate, the edge between these vertices must be in E.

The completeness is obtained with a reduction from 3-SAT. We could do a reduction from SAT, but 3-SAT is more regular, so we give it preference for the reduction. Let $\mathcal{I}_1$ be an instance of 3-SAT with n variables and p clauses. Then we build an instance $\mathcal{I}_2$ of CLIQUE. We add three vertices to the graph for each clause (each vertex corresponds to one of the literals of the clause) and then we add an edge between two vertices if and only if (i) they are not part of the same clause and (ii) they are not antagonist (i.e., one corresponding to a variable x_i and the other to its negation $\overline{x_i}$). An example is shown in Figure 6.1, with the graph obtained for a formula with three clauses $C_1 \wedge C_2 \wedge C_3$, with $C_1 = x_1 \vee \overline{x_2} \vee \overline{x_3}$, $C_2 = \overline{x_1} \vee x_2 \vee x_3$, and $C_3 = x_1 \vee x_2 \vee x_3$.

Note that $\mathcal{I}_2$ is a graph with $3p$ vertices; the size of this instance, therefore, is polynomial in the size of $\mathcal{I}_1$. Moreover, we fix in instance $\mathcal{I}_2$ the integer k of the CLIQUE definition such that $k = p$. We are now ready to check the equivalence of the solutions.

Assume first that the instance $\mathcal{I}_1$ of 3-SAT has a solution. Then, we pick a vertex corresponding to a variable that is true in each clause, and it is easy to check that the subgraph made of these p vertices is a clique. Indeed, two of such vertices are not in the same clause, and they are not antagonistic; therefore, there is an edge between them.

On the other side, if there is a clique of size k in instance $\mathcal{I}_2$, then necessarily there is one vertex of the clique in each clause (otherwise, the two vertices

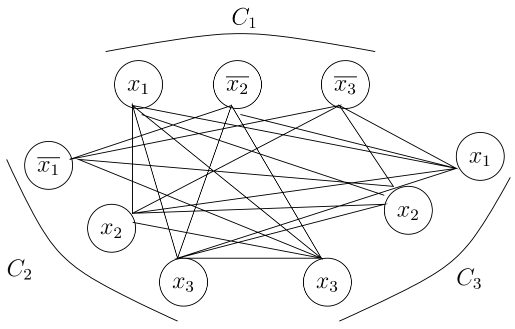

FIGURE 6.1: Example: Reduction of an instance of 3-SAT to an instance of CLIQUE.

within the same clause would not be connected). We choose these vertices to instantiate the variables, and we obtain a solution because we never make contradictory choices (because two antagonistic vertices cannot be part of the clique, there is no edge between them). This concludes the proof. □

We discuss variants of the CLIQUE problem in Section 6.5.

6.4.3 VERTEX-COVER

We continue to enrich the class NPC with another graph problem. We say that an edge $e = (u, v)$ is *covered* by its endpoints u and v.

DEFINITION 6.9 (VERTEX-COVER)**.** Let $G = (V, E)$ be a graph and k be an integer such that $1 \leqslant k \leqslant |V|$. Do there exist k vertices $v_{i_1}, \ldots, v_{i_k}$ such that each edge $e \in E$ is covered by (at least) one of the v_{i_j}, for $1 \leqslant j \leqslant k$?

THEOREM 6.3. *VERTEX-COVER is NP-complete.*

Proof. It is easy to check that VERTEX-COVER is in NP. The certificate is a set of k vertices, $V_c \subseteq V$, and for each edge $(v_1, v_2) \in E$, we check whether $v_1 \in V_c$ or $v_2 \in V_c$. The verification is done in time $|E| \times k$, and, therefore, it is polynomial in the problem size.

This problem is once again a graph problem, so we choose to use a reduction from CLIQUE, which turns out to be straightforward. Let $\mathcal{I}_1$ be an instance of CLIQUE: It consists of a graph $G = (V, E)$ and an integer k. We consider the following instance $\mathcal{I}_2$ of VERTEX-COVER. The graph is $\overline{G} = (V, \overline{E})$, which is the complementary graph of G, i.e., an edge is in $\overline{G}$ if and only if it is not in G (see the example in Figure 6.2). Moreover, we set the size of the covering set to $|V| - k$.

If instance $\mathcal{I}_1$ has a solution, G has a clique of size k, and, therefore, the $|V| - k$ vertices that are not part of the clique form a covering set of $\overline{G}$. Reciprocally,

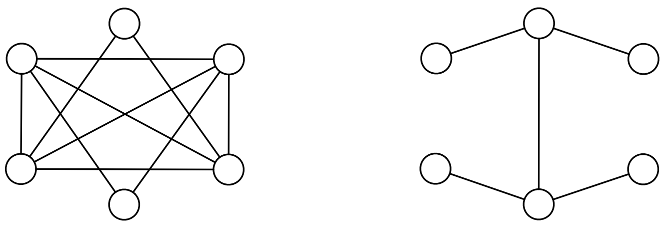

FIGURE 6.2: Example: Reduction of an instance of CLIQUE (on the left, graph G, $k = 4$) to an instance of VERTEX-COVER (on the right, graph $\overline{G}$, size of the cover $|V| - k = 2$).

if $\mathcal{I}_2$ has a solution, then the vertices that are not part of the covering set form a clique in the original graph G. This concludes the proof. □

6.4.4 Scheduling problems

Scheduling is the activity that consists of mapping an application onto a target platform and of assigning execution times to its constitutive parts. The application can often be represented as a task graph, where nodes denote computational tasks and edges model precedence constraints between tasks. For each task, an assignment (choose the processor that will execute the task) and a schedule (decide when to start the execution) are determined. The goal is to obtain an efficient execution of the application, which translates into optimizing some objective function. The traditional objective function in the scheduling literature is the minimization of the total execution time, or *makespan*; however, we will see examples with other objectives, such as those of the case study devoted to online scheduling (Chapter 15).

Traditional scheduling assumes that the target platform is a set of p identical processors, and that no communication cost is paid. In that context, a task graph is a directed acyclic vertex-weighted graph $G = (V, E, w)$, where the set V of vertices represents the tasks, the set E of edges represents precedence constraints between tasks ($e = (u, v) \in E$ if and only if $u \prec v$, where $\prec$ is the precedence relation), and the weight function $w : V \longrightarrow \mathbb{N}^*$ gives the weight (or duration) of each task. Task weights are assumed to be positive integers. A schedule σ of a task graph is a function that assigns a start time to each task: $\sigma : V \longrightarrow \mathbb{N}^*$ such that $\sigma(u) + w(u) \leqslant \sigma(v)$ whenever $e = (u, v) \in E$. In other words, a schedule preserves the *precedence constraints* induced by the precedence relation $\prec$ and embodied by the edges of the precedence graph. If $u \prec v$, then the execution of u begins at time $\sigma(u)$ and requires $w(u)$ units of time, and the execution of v at time $\sigma(v)$ must start after the end of the execution of u. Obviously, if there were a cycle in the task graph, no schedule could exist, hence, the restriction to acyclic graphs and, thus, the focus on Directed Acyclic Graphs (DAGs).

There are other constraints that must be met by schedules, namely, *resource constraints*. When there is an infinite number of processors (in fact, when there are as many processors as tasks), the problem is *with unlimited processors*, and denoted $P\infty|prec|Cmax$ in the literature [44]. We use the shorter notation SCHED(∞) in this book; each task can be assigned to its own processor. When there is a fixed number $p < n$ of available processors, the problem is *with limited processors*, and the general problem is denoted SCHED(p). SCHED(2) represents the scheduling problem with only two processors. Note that SCHED(∞) is equivalent to SCHED(q) for any value $q \geqslant n$, where n is the number of tasks. In the case with limited processors, a problem is defined by the task graph and the number of processors p. An allocation function $\mathsf{alloc} : V \longrightarrow \mathcal{P}$ is then required, where $\mathcal{P} = \{1, \ldots, p\}$ denotes the set of available processors. This function assigns a target processor to each task. The resource constraints simply specify that no processor can be allocated more than one task at the same time:

$$\mathsf{alloc}(T) = \mathsf{alloc}(T') \Rightarrow \begin{cases} \quad \sigma(T) + w(T) \leqslant \sigma(T') \\ \text{or } \sigma(T') + w(T') \leqslant \sigma(T). \end{cases}$$

This condition expresses the fact that if two tasks T and T' are allocated to the same processor, then their executions cannot overlap in time.

The *makespan* MS(σ, p) of a schedule σ that uses p processors is its total execution time: $\mathrm{MS}(\sigma, p) = \max_{v \in V}\{\sigma(v) + w(v)\}$ (assuming that the first task(s) is (are) scheduled at time 0). The makespan is the total execution time, or finish time, of the schedule. Let $\mathrm{MS}_{opt}(p)$ be the value of the makespan of an optimal schedule with p processors: $\mathrm{MS}_{opt}(p) = \min_\sigma \mathrm{MS}(\sigma, p)$. Because schedules respect precedence constraints, we have $\mathrm{MS}_{opt}(p) \geqslant w(\Phi)$ for all paths Φ in G (weights extend to paths in G as usual). We also have $\mathrm{Seq} \leqslant p \times \mathrm{MS}_{opt}(p)$, where $\mathrm{Seq} = \sum_{v \in V} w(v) = \mathrm{MS}_{opt}(1)$ is the sum of all task weights.

While SCHED(∞) has polynomial complexity (simply traverse the graph and start each task as soon as possible using a fresh processor), problems with a fixed amount of resources are known to be difficult. Letting DEC be the decision problem associated with SCHED, and INDEP the restriction of DEC to independent tasks (no precedence constraints), i.e., $E = \emptyset$, well-known complexity results are summarized below:

- INDEP(2) is NP-complete but can be solved by a pseudopolynomial algorithm. Moreover, $\forall\, \varepsilon > 0$, INDEP(2) admits a $(1+\varepsilon)$-approximation whose complexity is polynomial in $\frac{1}{\varepsilon}$ (see Section 8.1.5, p. 187).
- INDEP is NP-complete in the strong sense (see Exercise 7.10, p. 152) but can be approximated up to some constant factor (see Exercise 9.5, p. 215). Moreover, $\forall \varepsilon > 0$, there is a $(1 + \varepsilon)$-approximation algorithm for this problem [50].
- DEC(2) (and hence DEC) is NP-complete in the strong sense (see Exercise 7.11, p. 152).

All these results are gathered here for the sake of comprehensiveness. The impatient reader who wonders what is the meaning of NP-complete *in the strong sense* may refer to Section 6.6, p. 145, and to understand what is *an approximation algorithm*, she/he may have a quick look at Section 8.1, p. 179 right now.

Scheduling and certificates

Scheduling problems provide a nice illustration of the attention that must be paid to certificates. Consider the DEC decision problem, namely, scheduling a task graph with p processors and a given deadline D. For the schedule to be valid, both precedence and resource constraints must be enforced. The question is to decide whether there exists a schedule whose makespan does not exceed the deadline.

A naive verification of the schedule is to describe which tasks are executed onto which processors at each time step. Unfortunately, this description may lead to a certificate of exponential size; the time steps range from 1 to D, and the size of the scheduling problem is $O(n + p + \log W + \log D)$, where $W = \sum_{v \in V} w(v)$.

A polynomial size verification of the schedule can be easily obtained using *events*, which are time steps where a new task begins or ends. There is a polynomial number of such events ($2n$), and for each of them we perform a polynomial number of checks. From the definition of the schedule, we first construct the ordered list of events in polynomial time. The basic idea is to maintain the set of tasks that have been completed and the set of processors that are currently idle. If the event corresponds to starting a new task, we check that all its predecessors have been completed, and that the target processor belongs to the set of idle processors (and then we remove it from this set). If the event corresponds to completing a task, we mark the task accordingly, and we re-insert the target processor into the set of idle processors. Note that if several events take place at the same time step, we should start with those that correspond to task completions. We perform these checks one event after the other until we reach the last one, which corresponds to the completion of the last task, and which much take place not later than D.

In summary, we see that the weights of the tasks (given by the function w) prevent us from using a naive verification of the validity of a schedule at each step of its execution. This is because the makespan is not polynomial in the problem size.

6.4.5 Other famous NP-complete problems

We have initiated discussions with the 2-PARTITION problem (Definition 6.1) that is one of the most widely used problems to perform reductions, since it turns out to be NP-complete while being quite simple in its formulation. The NP-completeness of 2-PARTITION will be shown in Exercise 7.20, p. 155, but from now on, we assume that this problem is indeed NP-complete.

The COLOR problem (see Definition 6.3) given in Section 6.2.2 is also NP-complete and the proof is the purpose of Exercise 7.7, p. 151. Other problems will discuss variants of this graph coloring problem.

Another useful problem is HC (Hamiltonian Cycle, see Definition 6.4). We have already shown that HC is in NP (see Section 6.2.2). For the completeness, we refer the interested reader to involved reduction in [27]. There is a nice reduction from 3-SAT in the first edition of the book, and the current edition performs a reduction from VERTEX-COVER.

Starting from HC, it is easy to prove that TSP (see Definition 6.5) also is NP-complete. It is clear that TSP is in NP; a certificate is an ordered list of vertices. The reduction comes from HC. Let $\mathcal{I}_1$ be an instance of HC: This is a graph $G = (V, E)$. We build the following instance $\mathcal{I}_2$ of TSP. The graph $G' = (V, E')$ has the same set of vertices as G, but it is a complete graph. We set $k = 0$, i.e., we want to find a cycle of weight 0. Finally, for $e \in E'$ we define the cost function w such that $w(e) = 0$ if $e \in E$, and $w(e) = 1$ otherwise. This reduction is obviously of polynomial time, and the equivalence of solutions is straightforward. Note that this last NP-completeness result comes from the fact that TSP is a weighted version of HC.

For a reference list of problems known to be NP-complete, we refer the reader to the book by Garey and Johnson [38].

6.5 Importance of problem definition

In this section, we point out subtleties in problem definitions. A parameter can be either fixed for the problem or part of the problem instance. Consider the problem CLIQUE introduced in Section 6.4.2. Given a graph $G = (V, E)$, we introduce the notion of β-clique of size k, where β is a rational such that $0 < \beta \leqslant 1$ [83]; a β-clique is a subgraph of G of size k (k vertices), with edge density at least β. The edge density is the ratio of the number of edges in the subgraph over the number of edges in a clique of size k, i.e., $\binom{k}{2}$. We can now define a variant of the CLIQUE problem:

DEFINITION 6.10 (BCLIQUE)**.** Let $G = (V, E)$ be a graph, β be a rational number such that $0 < \beta \leqslant 1$, and k be an integer such that $1 \leqslant k \leqslant |V|$. Does there exist a β-clique of size k in G?

In the BCLIQUE problem, β is part of the instance. Therefore, we can do a trivial reduction from CLIQUE, letting $\beta = 1$, to prove that it is NP-complete. However, we may define the problem in a different way, where β is given. For a constant β such that $0 < \beta \leqslant 1$, we define:

DEFINITION 6.11 (BCLIQUE(β))**.** Let $G = (V, E)$ be a graph and k be an integer such that $1 \leqslant k \leqslant |V|$. Does there exist a β-clique of size k in G?

We have CLIQUE = BCLIQUE(1). However, the NP-completeness of CLIQUE does not imply the NP-completeness of BCLIQUE(β) for any value of β. We prove this NP-completeness for any fixed value $0 < \beta < 1$ in the following theorem:

THEOREM 6.4. *BCLIQUE(β) is NP-complete for any rational number $\beta = \frac{p}{q}$, where p and q are positive integer constants and $p < q$.*

Proof. It is clear that BCLIQUE(β) is in NP, and the reduction comes logically from the classical CLIQUE problem. The idea is to construct an auxiliary graph $G' = (V', E')$ and to prove that G has a clique of size k if and only if $G \cup G'$ has a β-clique of size $|V'| + k$.

We build the set of vertices V' of size $|V'| = 4(|V|^2 + k^2)q - k$, containing vertices v'_1 to $v'_{|V'|}$. For $1 \leqslant i \leqslant |V'|$ and $j \in [i+1, i+|V|] \mod |V'|$, we add an edge between v_i and v_j. Therefore, each node has $2|V|$ edges, and we have added a total of $|V||V'|$ edges. Next, we add random edges in order to have a total of $K = \frac{p}{q}\binom{|V'|+k}{2} - \binom{k}{2}$ edges between the $|V'|$ vertices. Because $|V'| + k$ is a multiple of $2q$, K is an integer. Moreover, $|V'|$ is large enough so that we can prove that $\binom{|V'|}{2} \geqslant K \geqslant |V||V'|$ (see [83]), i.e., there were initially fewer than K edges, and we can have a total of K edges without exceeding the maximum number of edges in $|V'|$.

There remains to prove that G has a clique of size k if and only if $G \cup G'$ has a $\frac{p}{q}$-clique of size $|V'| + k$. Suppose first that there is a clique C of size k in G. We consider the subgraph Q of $G \cup G'$ containing vertices $C \cup V'$. We have $|Q| = |V'| + k$ and the number of edges is $K + \binom{k}{2} = \frac{p}{q}\binom{|V'|+k}{2}$; therefore, Q is a $\frac{p}{q}$-clique by definition.

Suppose now that there is a $\frac{p}{q}$-clique Q of size $|V'| + k$ in $G \cup G'$. We first construct a $\frac{p}{q}$-clique Q' such that $|Q'| = |Q|$ and $V' \subset Q'$. Since $|Q| > |V'|$, $|V' \setminus Q| \leqslant |V|$, and each vertex in $V' \setminus Q$ cannot be connected to more than $|V| - 1$ vertices of $V' \setminus Q$. Moreover, each vertex of $V' \setminus Q$ is of degree at least $2|V|$ and, therefore, it is connected to at least $|V| + 1$ vertices of Q, while vertices of $Q \cap V$ are connected to at most $|V| - 1$ vertices (all of them from V). Therefore, we can replace $|V' \setminus Q|$ vertices of $Q \cap V$ with the remaining vertices of V', with no reduction in the edge density. We obtain a $\frac{p}{q}$-clique Q' such that $|Q'| = |Q|$ and $V' \subset Q'$. Then, $|Q' \cap V| = k$. To see that $Q' \cap V$ is a clique of size k in V, consider the density of G'; it is K by construction. If $Q' \cap V$ does not contribute $\binom{k}{2}$ edges, then Q' cannot have density $\frac{p}{q}$. Therefore, $Q' \cap V$ is a clique of size k, hence concluding the proof. $\square$

In scheduling problems (see Section 6.4.4, p. 140), the same distinction is often implicitly made, whether the number of processors p is part of the problem instance or not. For instance, if all tasks are unit-weighted, DEC is

NP-complete (with p in the problem instance), while DEC(2) can be solved in polynomial time and DEC(3) is an open problem [38].

6.6 Strong NP-completeness

The last technical discussion of this chapter is related to *weak* and *strong* NP-completeness. This refinement of the NPC problem class applies to problems involving numbers, such as 2-PARTITION, but also TSP, because of edge weights.

Consider a decision problem P, and let I be an instance of this problem. We have already discussed how to compute $size(I)$, the size of the instance, encoded in binary. We now define $max(I)$, which is the *maximum* size of the instance, typically corresponding to the problem instance with integers coded in unary. To give an example, consider an instance I of 2-PARTITION with n integers $a_1, \dots, a_n$. As already discussed, we can have $size(I) = n + \sum_{1 \leqslant i \leqslant n} \log(a_i)$, or any similar (polynomially related) expression. Now we can have $max(I) = n + \sum_{1 \leqslant i \leqslant n} a_i$, or $max(I) = n + \max_{1 \leqslant i \leqslant n} a_i$, or any similar (polynomially related) expression.

Then, given a polynomial p, we define P_p, the problem P restricted to p, as the problem restricted to instances such that $max(I)$ is smaller than $p(size(I))$, i.e., the size of the instance coded in unary is bounded applying p to the binary size of the instance. A problem P (in NP) is NP-complete *in the strong sense* if and only if there exists a polynomial p such that P_p remains NP-complete. Otherwise, if the problem restricted to p can be solved in polynomial time, the problem is NP-complete *in the weak sense*; intuitively, in this case, the problem is difficult only if we do not bound the size of the input in the problem instance.

Note that for a graph problem such as the bipartite graph problem, there are no numbers, so $max(I) = size(I)$ and the problem is NP-complete in the strong sense. For problems with numbers (including weighted graph problems), one must be more careful. Coming back to 2-PARTITION, we have seen in Section 6.2.1 that it can be solved by a dynamic-programming algorithm running in time $O(n \sum_{i=1}^{n} a_i)$, or equivalently in time $O(max(I))$. Therefore, any instance I of 2-PARTITION can be solved in time polynomial in $max(I)$, which is the definition of a pseudopolynomial problem. And 2-PARTITION is not NP-complete in the strong sense (one says it is NP-complete in the weak sense).

To conclude this section, we introduce a problem with numbers that is NP-complete in the strong sense: 3-PARTITION. The name of this problem is misleading because this problem is different from partitioning n integers into three sets of same size.

DEFINITION 6.12 (3-PARTITION)**.** Given an integer B, and $3n$ integers $a_1, \dots, a_{3n}$, can we partition the $3n$ integers into n triplets, each of sum B? We can assume that $\sum_{i=1}^{3n} a_i = nB$ (otherwise, there is no solution), and that $B/4 < a_i < B/2$ (so that one needs exactly three elements to obtain a sum B).

Contrary to 2-PARTITION, 3-PARTITION is NP-complete in the strong sense [38].

6.7 Why does it matter?

We conclude this chapter with a discussion on polynomial problems. Why focus on polynomial problems? If the size of the data is in n, from a practical perspective it is much better to have an algorithm in $(1.0001)^n$, which is exponential, than a polynomial-time algorithm in n^{1000}. In such a case, the polynomial-time algorithm is still slower than the exponential one for $n = 10^9$, and, therefore, the exponential algorithm is faster in any practical situation. However, n^{1000} is not practical either. In general, polynomial algorithms have a small degree, typically not exceeding 4 and almost always smaller than 10.

Polynomial-time algorithms are likely to be efficient algorithms, so when confronted with a new problem, the first thing we do is to look for an algorithm that would solve it in polynomial time. If we succeed, we are finished. If we do not succeed, we have another way to go—prove that the problem is NP-complete. Then the chance of somebody else coming later and providing an optimal solution to the problem is very small because it is very unlikely that P = NP. In other words, if we can show that our problem is more difficult than one (hence all) of these famous NP-complete problems, then we show strong evidence of the intrinsic difficulty of the problem.

Of course, proving a problem NP-complete does not make it go away. One needs to keep a constructive approach, such as proposing an algorithm that provides a near-optimal solution in polynomial time (or again, proving that no such approximation algorithm exists). This is the subject of Chapter 8.

6.8 Bibliographical notes

As already mentioned, our approach to NP-completeness is original. See the book by Garey and Johnson [38] for a comprehensive treatment of NP-completeness and a famous catalog of NP-complete problems. A very intuitive proof of Cook's theorem is given in the book by Wilf [108]. A theory-oriented approach with Turing machines and complexity results is available in the book by Papadimitriou [82]. The more adventurous reader can investigate the book by Arora and Barak [4].