24 NP-completeness

Controlling complexity is the essence of computer programming.
– Brian W Kernighan
Any sufficiently advanced technology is indistinguishable from magic.
– Arthur C Clarke

In Section 14.5, I proved that the sorting problem cannot be solved in $o(n \log n)$ time using comparisons only. Algorithms like the merge sort and the heap sort solve the problem in $O(n \log n)$ time and are, therefore, optimal. In that section, I also initiated a discussion on the complexity of a problem instead of an algorithm. Classification of problems based on their complexities is a serious study in computer science. A problem, in abstract terms, is a problem, independent of any algorithm to solve it, and even of whether there exists any algorithm to solve the problem. Still, algorithms play a vital role in the classification efforts mentioned above.

In this chapter, I deal with problems that can be solved in polynomial time. Generalizing the concept of algorithms lets us talk about problems that can be solved in non-deterministic polynomial time. A bunch of open questions crops up during this study, of which the most celebrated one is the million-dollar $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ question. In an attempt to resolve this question, the concept of complete problems is introduced. The $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ question, however, stands open until now.

This apparently theoretical study has enormous practical implications. Suppose we encounter a problem that is known to be complete in the class of non-deterministic polynomial-time algorithms. We do not know how to solve this problem in (deterministic) polynomial time and even whether it can at all be solved in (deterministic) polynomial time. Still, this is a problem we need to solve. What is the way out then? In practice, one may go for approximation or randomized algorithms (discussed in the next chapter), or one may focus on a subclass of the problem, that yields polynomial-time solution. In short, it is necessary to develop an intuition about the *difficulty* of solving problems.

On the other extreme, a naive algorithm to solve a problem may be too inefficient (such as exponential-time) to be of any practical use. However, the problem may be known to have polynomial-time algorithms. In this case, we should study these algorithms and choose one suitable for our needs. In short, it is necessary to develop an intuition about the *ease* of solving problems.

For the study of complexity of problems, it is a common practice to concentrate (only) on decision problems, that is, on those problems that have yes/no answers. This sounds like a loss of generality. For example, proving an integer n as composite is not the same as explicitly finding a non-trivial factor of n. As another example, the task of a compiler is not limited to checking whether an input program is syntactically correct. In many situations, however, general functional and optimization problems can be rephrased in terms of decision problems. For instance, the integer factorization problem can be solved in polynomial time if and only if the decision problem of Exercise 5.196 can be solved in polynomial time. This means that although the problem of deciding the compositeness of n fails to correctly capture the complexity of factoring n, another decision problem achieves this goal. The trouble was not with decision problems, but with framing the correct decision problems. Nonetheless, this example illustrates why restriction to decision problems has been universally accepted as an effective and fruitful simplification.

24.1 Complexity classes P and NP

We have seen earlier that a polynomial-time algorithm performs better, both theoretically and practically, than super-polynomial-time (like exponential-time) algorithms. Polynomial-time algorithms are considered efficient. Obviously, a $\Theta(n^{100})$-time algorithm is expected to be of little practical use. Luckily enough, most polynomial-time algorithms, that we deal with in practice, do not have such huge exponents in their running times. In any case, the notion of polynomial-time solvability is usually taken in the same spirit as the notion of efficient solvability.

Definition The class of problems that can be solved in polynomial time is called the complexity class P.

The class P consists precisely of those problems that can be solved *efficiently* (see above). It is a huge (infinite) class, but we cannot precisely characterize all the problems in P. Problems that are *known* to have polynomial-time algorithms do belong to this class. However, the non-existence of a known polynomial-time algorithm for a particular problem does not automatically imply that this problem lies outside P. All we can say in this case is that we do not know whether this problem belongs to P.[24.1] As an example, consider the primality testing problem. Before August 2002, we did not know whether this problem belongs to P. The discovery of the AKS algorithm provided a proof that the primality testing problem is indeed in P.

In order to prove that a problem is in P, it suffices to supply a polynomial-time algorithm for the problem. It does not matter how large the degree of the polynomial is (it must be constant though).

24.1.1 Algorithms that can guess

With a stretch of imagination, we generalize the notion of algorithms. All algorithms discussed so far are deterministic in the sense that every step in the algorithm is definite and well-defined. We now relax this condition and allow each step to be chosen from a collection of several (albeit a finite number of) possibilities. For the moment, let us not bother about how to implement non-deterministic steps, namely who chooses a possibility for a step and on what basis. Let us instead try to conceptualize non-determinism using some examples.

Consider the problem COMPOSITE that determines whether an input integer $n > 0$ is composite. If n is composite, it has a non-trivial divisor d ($2 \leqslant d \leqslant n-1$ and $d|n$). Discovering such a divisor d establishes the compositeness of n. On the other hand, if n is not composite, no d in the range $2 \leqslant d \leqslant n-1$ divides n. The non-deterministic algorithm discussed below has to do with guessing a non-trivial divisor d of n. Let l be the bit-length of n. Then, an integer d in the range $2 \leqslant d \leqslant n-1$ can be encoded using l bits. We guess the l bits of a potential divisor d. If d indeed divides n, then we declare n as *composite*, else as *not composite*.

```
Let l be the bit-length of n;
for (i = 0; i < l; ++i) guess a bit d_i from the set {0,1};
Let d = (d_{l-1} d_{l-2} ... d_1 d_0)_2;
if ((2 ⩽ d ⩽ n-1) && (d divides n)) output "Yes"; else output "No";
```

[24.1] It is also possible that we can provide a proof that the problem cannot belong to P. There indeed exist such provably difficult computational problems.

It is outrageous to conclude about the compositeness of n from a single guess of d. We are not going to do so anyway. We instead demand that if there is at least one guess for d for which the output is *Yes*, then we say that the non-deterministic algorithm *accepts* n as composite. On the other hand, if all guesses for d yield the answer *No*, then the algorithm *rejects* n as composite.

Let us also conceptualize the running time of the above non-deterministic algorithm. Henceforth, we assume that a guess of each bit takes one unit of time (or $O(1)$ time). The algorithm starts by computing the bit length l of n. This can be performed in $O(\lg n)$ time using bit operations. This is followed by guessing l bits, taking a total of $l = O(\lg n)$ time. Construction of d from the bits can also be accomplished in $O(\lg n)$ time using bit operations. One can compare d with 2 and $n-1$ in $O(\lg n)$ time (including the computation of $n-1$ from n). Finally, the division of n by d can be carried out in $O(\lg^2 n)$ time. To sum up, the above non-deterministic algorithm runs in $O(\lg^2 n)$ time which is polynomial in the input size $\lg n$.

I mentioned earlier that we need to consider all possible guesses in order to determine whether n is really composite or not. I still say so, but warn you that it is the deterministic way of looking at the algorithm. In the non-deterministic sense, a single set of guesses *confirms* the compositeness or otherwise of n.

How to implement? Assume that in the case when n is composite, somebody tells the algorithm how to guess the bits d_i correctly so as to generate a non-trivial divisor d of n. We will later call this prescription of correct guesses a certificate for the compositeness of n. If n is not composite, whatever guesses the algorithm makes, the output will be *No*, that is, issuing a *No* as output based on a single guess of d is not unjustified. In fact, there does not exist a certificate for the compositeness of n in this case.

Imagine that you are asked to compute a tricky integration. If you plan to solve it deterministically (honestly, I meant), you keep on employing substitutions, integration by parts, and all tools available to you. On many occasions, you get stuck. You track back, refine and/or modify your substitution and other choices, get stuck again, track back, and so on. Eventually, a trick works. You present to your teacher your final solution which shows only the successful sequence.

Suppose, on the contrary, you are non-deterministic (dishonest!). You go to a friend who divulges to you the basic steps (for example, first substitute $y = x^2/\ln\sin\sqrt{x}$, then make a substitution $z = e^y/(y^2 + \tan^{-1} y)$, and so on). In this case too, you can generate your final solution, expectedly much faster than your honest attempt.

There, however, do exist functions that are not integrable. If such a function is given by your calculus teacher, no hint from any friend can help. Any attempt you make (honestly or dishonestly) is bound to fail!

Another way to look at non-determinism is by using parallel computation. A k-way choice for a step may be viewed as replacing each process by k child processes running independently of one another and each handling a particular choice of the step. Each child process may encounter non-deterministic steps and, if so, an appropriate number of grandchild processes are created and executed in parallel. We may assume that we have an infinite number of processors so that all processes thus created can run in parallel (as long as there are only finitely many processes).

Let us look at some other examples. Let $G = (V,E)$ be a graph (say, undirected, but not necessarily simple). Let $|V| = n$ and $|E| = m$. We define a *Hamiltonian cycle* in G to be a permutation $v_0, v_1, \ldots, v_{n-1}$ of the vertices in G such that each (v_i, v_{i+1}) and also (v_{n-1}, v_0) belong to the edge set

E. In other words, a Hamiltonian cycle is a cycle in G that visits each vertex of G (exactly once). We denote, by HAM-CYCLE, the problem of determining whether a given graph has a Hamiltonian cycle. Here is a non-deterministic algorithm for this problem.

```
for (i=0; i<n; ++i) guess the vertex v_i;
Check whether v_0,v_1,...,v_{n-1},v_0 is a Hamiltonian cycle in G;
if so, output "Yes"; else output "No";
```

The non-deterministic part of the above algorithm is the choice (guessing) of the order of the vertices in G. The check whether a given sequence $v_0, v_1, \ldots, v_{n-1}, v_0$ yields a Hamiltonian cycle can be carried out deterministically. Here is a possible strategy. First, unmark all the vertices of G. Mark v_0. For $i = 1, 2, \ldots, n-1$, check whether v_i is already marked. If so, the vertex v_i is already visited, so report failure. If not, check whether the edge (v_{i-1}, v_i) belongs to E. If not, return failure, else mark v_i, increment i and repeat. When the loop terminates successfully, check whether $(v_{n-1}, v_0) \in E$. If so, return success, else return failure.

Let us now look at the running time of this algorithm. The choice of each v_i involves selecting one of the n vertices of G. If we rename the vertices of G by the integers $0, 1, \ldots, n-1$, a choice of v_i is equivalent to a choice of l bits, where $l = \lceil \lg(n+1) \rceil$. This choice can be performed in $O(\log n)$ time, that is, n vertices can be chosen in a total of $O(n \log n)$ time. As mentioned in the last paragraph, the check whether the sequence $v_0, v_1, \ldots, v_{n-1}, v_0$ corresponds to a Hamiltonian cycle can be carried out in polynomial time. (The exact degree depends on the implementation.) To sum up, the above non-deterministic algorithm runs in polynomial time in the worst case.

Finally, I introduce a problem of a similar description. Let $G = (V, E)$ be again an undirected graph with $|E| = m$. An *Eulerian tour* in G is a permutation $e_0, e_1, \ldots, e_{m-1}$ of the edges of G such that the end vertex of e_i is the start vertex of e_{i+1}, and the end vertex of e_{m-1} is the start vertex of e_0. Thus, an Eulerian tour is a closed walk in the graph with each edge traversed once and only once. By EULER-TOUR, we denote the problem of deciding whether an input graph G has an Eulerian tour. A non-deterministic algorithm to solve this problem follows.

```
for (i=0; i<m; ++i) guess the edge e_i;
Check whether e_0,e_1,...,e_{m-1} is an Eulerian tour in G;
if so, output "Yes"; else output "No";
```

This algorithm has an identical structure as the algorithm provided for solving HAM-CYCLE. It is an easy matter to check that the non-deterministic algorithm for EULER-TOUR runs in polynomial time (in m).

Let me now provide a concrete definition: a non-deterministic algorithm is one which involves a finite number of guesses. We also allow zero guesses, indicating that a deterministic algorithm is also treated as non-deterministic. If all sequences of guesses lead to the answer *No*, we say that the decision of the algorithm is *No*. On the other hand, if at least one sequence of guesses lets the algorithm output *Yes*, we say that the decision of the algorithm is *Yes*.

The running time of a non-deterministic algorithm is expressed as a function of the size n of the input. We assume that guessing each single bit is an elementary operation, that is, takes one unit of time. The worst-case running time of a non-deterministic algorithm is its maximum possible running time, where the maximum is taken over all possible sequences of guesses and over all possible inputs of size n. This concept of running time leads to the following important definition.

Definition The class of problems that can be solved in polynomial time using non-deterministic algorithms is called the complexity class NP.

Since we call deterministic algorithms (those that make no guesses) non-deterministic also, it trivially follows that

$$\mathrm{P} \subseteq \mathrm{NP}.$$

What about the converse, that is, is $\mathrm{NP} \subseteq \mathrm{P}$ too? The answer is not known. There exist many problems in NP, that do not have known polynomial-time deterministic algorithms. Intense research on these problems for several decades has not revealed any clue whether any polynomial-time deterministic algorithms may exist for these problems. We are equally clueless about a proof that some of these problems cannot be solved in polynomial time. This leads to the famous

$$\mathrm{P} \stackrel{?}{=} \mathrm{NP}$$

question which turns out to be the deepest unsolved problem in computer science. The Clay Mathematical Institute has identified seven unsolved problems on May 24, 2000 and declared a prize worth $1 million for solving each of these problems.[24.2] Most computer scientists (and mathematicians) strongly believe that $\mathrm{P} \neq \mathrm{NP}$, but nobody seems to be anywhere near a concrete proof.

Let us try to *simulate* a non-deterministic algorithm N by a deterministic algorithm D. An obvious strategy is to enumerate all possible sequences of guesses made by the non-deterministic algorithm. D runs N once for each guess sequence. If all guess sequences yield the answer *No*, the simulating algorithm D says *No* too. On the other hand, if for some sequence of guesses, the answer *Yes* is encountered, then D stops immediately after declaring *Yes* as its decision.

Let n be the input size, and $f(n)$ (an upper bound on) the number of steps taken by N on an input of size n. Furthermore, let g denote the (maximum) number of guesses (of bits) made by N on an input of size n. Since guessing each bit takes one unit of time, it follows that $g \leqslant f(n)$. Each guess of a bit gives two possibilities: 0 and 1, that is, g guesses yield 2^g possibilities. Thus, an upper bound on the running time of the simulating algorithm D is $2^g f(n) \leqslant 2^{f(n)} f(n) = 2^{f(n) \lg f(n)}$. If $f(n) = \mathrm{O}(n^d)$ for some $d \in \mathbb{N}$, this running time is $\mathrm{O}((2^d)^{n^d \lg n})$. This is an expression exponential in n.

It is not clear whether this exponential blow-up in the running time is unavoidable for some of the problems in NP. Simulating a non-deterministic algorithm by a deterministic algorithm need not be the *only* way to solve a problem in NP. A problem in NP is not ruled out to have independent polynomial-time algorithms.

Both the problems HAM-CYCLE and EULER-TOUR belong to NP. We do not know how to solve HAM-CYCLE in polynomial time. As we will see shortly, HAM-CYCLE is one of the most difficult problems in NP. On the other hand, there exist polynomial-time algorithms for solving EULER-TOUR. Exercise 6.39 deals with one such algorithm. It is surprising that two problems of similar descriptions have (presumably) different algorithmic behaviors.

24.1.2 Polynomial-time verifiability

If you are uncomfortable with the notion of non-deterministic algorithms, here is an alternative and equivalent way of characterizing the problems in NP. The problems in P can be *solved* easily,

[24.2] Visit `http://www.claymath.org/millennium/` to know more about the *Millennium Problems*.

whereas the problems in NP can be *verified* easily. In order to explain what verification means, I introduce the concept of certificates.

Let P be a decision problem and I an input for P. The set of all inputs for P, for which the decision is *Yes*, is denoted by Accept(P), and the set of those, for which the decision is *No*, by Reject(P). A certificate C applies to I only if $I \in$ Accept(P). It is an assertion about I, that proves that I does belong to Accept(P). No certificate is meant for proving $I \in$ Reject(P).

As a real-life example, let P be the problem of being able to swim, and let all human beings form the set of inputs for P. Those who can swim belong to Accept(P), and those who cannot swim belong to Reject(P). Now, consider an input I (perhaps I or you) for P. If Mr. I can really swim, he can prove it. However, proving that requires a swimming pool with sufficient water, swimming costumes and other accessories, a watchful and capable guard who can save people in case of accidents, a nice weather, and so on. That is already a lot of arrangement. Instead, suppose that Mr. I can provide a certificate from a competent authority (a certificate from the Gym). That proves that Mr. I is indeed capable of swimming. There is no need to set up a swimming test for this proof. On the other hand, if Mr. I cannot swim, he cannot furnish a swimmer's certificate, and so can in no way prove that he is capable of swimming. Moreover, no certificate can prove that he *cannot* swim.

Let us now look at computational examples. First, consider the problem COMPOSITE. A positive integer n is composite if and only if it admits a non-trivial divisor, that is, a divisor d in the range $2 \leqslant d \leqslant n-1$. If n is indeed composite, such a divisor proves that n is composite. All we need to do for the verification is that we divide n by d and determine that this division leads to the zero remainder. One (multiple-precision) division can be carried out in polynomial time. A prime number, on the other hand, cannot possess such a certificate. The non-existence of a non-trivial divisor cannot be certified in an easily provable manner. However, note that there exist certificates for primality too, but primality certificates are based on concepts different from those involved in compositeness certificates. Moreover, we now know that primality testing can be carried out in polynomial time (recall the AKS algorithm).

For a graph $G \in$ Accept(HAM-CYCLE), a certificate is a Hamiltonian cycle in G. Such a cycle is specified by a sequence of vertices in G. We have described earlier how such a sequence can establish, in polynomial time, the fact that $G \in$ Accept(HAM-CYCLE). If G does not contain a Hamiltonian cycle, then no sequence of vertices of G can prove that $G \in$ Reject(HAM-CYCLE).

Finally, for $G \in$ Accept(EULER-TOUR), an Eulerian tour (a sequence of edges) in G proves that G indeed belongs to Accept(EULER-TOUR). A graph containing no Eulerian tours cannot possess such certificates for proving its membership in Reject(EULER-TOUR).

There is a commonness in all these examples. Given an instance $I \in$ Accept(P) of size n and a certificate C for I, one can verify, in polynomial time, that $I \in$ Accept(P). With an abuse of convention, the input size for the verification algorithm is taken as n itself. We neglect the size of the certificate. However, since a verification algorithm needs to read the certificate and still must finish in polynomial time, we require that the certificate must be of length bounded by a polynomial function of n. Certificates that are longer (that is, of super-polynomial length) cannot lead to polynomial-time verification. Certificates verifiable in polynomial time are called *succinct certificates*.

I now prove a very important fact relating problems in NP with succinct certificates.

Theorem A problem P is in NP if and only if every instance $I \in$ Accept(P) possesses a succinct certificate.

Proof [If] We need to furnish a non-deterministic algorithm for P. The basic idea is that we guess certificates for the input instance. Succinctness of certificates guarantees that there are only finitely many candidates for certificates. More precisely, let n be the size of the input I. Suppose that n^k is a bound on the size of the certificate for an instance of size n. We non-deterministically generate a string of size $\leqslant n^k$ and check whether this string certifies the membership of I in Accept(P). If any one of these candidate strings proves this membership, we know $I \in$ Accept(P). If all candidates fail, then I does not possess any certificate and so belongs to Reject(P).

```
Non-deterministically generate a candidate C (a string of length ⩽ n^k);
Verify whether C is a certificate for I;
if so, output "Yes"; else output "No";
```

[Only if] Let A be a polynomial-time non-deterministic algorithm for P. A makes some non-deterministic guesses during its execution. If the input instance $I \in$ Accept(P), then there exists (at least) one sequence of guesses that lets A output *Yes*. A specification of this sequence (say, in the form of a bit sequence, assuming that each guess is that of a bit) is a certificate for I. Since A runs in polynomial time, the number of guesses it makes must also be bounded by a polynomial in n. Therefore, the certificate for $I \in$ Accept(P) mentioned above is succinct. •

One can use the above characterization for defining the class NP, that is, the class of problems P for which every $I \in$ Accept(P) possesses a succinct certificate is defined to be the class NP. In other words, the problems in NP are precisely those that can be verified in polynomial time.

That P $\subseteq$ NP follows easily from this characterization too. Suppose $P \in$ P, and $I \in$ Accept(P) has a certificate C. A verification algorithm can be designed as follows. (Read but) ignore the certificate C, and solve P on I. Since $P \in$ P, there exists a (deterministic) polynomial-time algorithm for P. We use this algorithm for solving P on I. We finally output the decision of the deterministic algorithm as the decision of the verification algorithm.

24.2 Polynomial-time reductions and NP-complete problems

Let us once again review the fundamental question P $\stackrel{?}{=}$ NP. If P = NP, every problem that can be verified easily can also be solved easily. On the other hand, if P $\neq$ NP, there exist problems that can be verified easily but cannot be solved easily. Which one of these two possibilities is true is not known to us. The problem continues to remain unsolved for several decades.

A giant step towards understanding this question is proposed independently by Stephen Cook and Leonid Levin in 1971. They tried to identify a class of problems in NP, that can be rightfully called the class of the computationally most difficult problems in NP. Any polynomial-time algorithm for any of these problems leads to the collapse P = NP. On the other hand, if we can prove that some of these difficult problems cannot be solved in polynomial time, then we have P $\neq$ NP. Cook and Levin's theory does not lead to an immediate settlement of the P $\stackrel{?}{=}$ NP question, but provides a potentially precious backdrop for the study of the unsolved question. Moreover, our capability of identifying the most difficult problems in NP has profound practical significance. If we do not know how to solve a problem easily, the best thing we can do is to concentrate our effort on the design of approximate and/or restricted solutions.

Let me first introduce the notion of reduction between problems. Let P and P' be decision problems. A reduction from P to P' is an algorithm that converts an input I for P to an input I'

for P' such that $I \in \text{Accept}(P)$ if and only if $I' \in \text{Accept}(P')$. In order that reductions can be done efficiently, we require the reduction algorithm to run in time polynomially bounded by n, where n is the size of the input I for P. If there exists a polynomial-time reduction from P to P', we say that P is reducible to P' and denote this as $P \leqslant P'$. The reduction $P \leqslant P'$ implies that if we know a way to solve P', we also know a way to solve P, namely, first convert the input I for P to an input I' for P', run an algorithm for P' on input I', and output the decision taken by this algorithm. Notice that this reduce-then-solve strategy is not necessarily the only (or the easiest) way to solve P. Most importantly, P may be solved more efficiently using other algorithms. This explains the sign $\leqslant$ in $P \leqslant P'$. In words, the problem P is computationally no harder than the problem P'.

Examples

(1) The notion of reduction can be illustrated by the following joke. A mathematician once entered a room and found an empty bucket on the floor. There were a tap and a table in the room. The task of the mathematician was to put a bucket full of water on the top of the table. He opened the tap, filled the bucket with water and placed the bucket on the table. Another day, the mathematician entered the room on an identical mission. This time, he discovered that the bucket was already full of water. He thought for a while, and then drained out the water from the bucket, put the empty bucket on the floor, and exclaimed with immense pleasure: *I have reduced this new problem to the old problem which I can solve!*

(2) Let G be an undirected graph and let s and t be vertices in G. A *Hamiltonian path* in G is a path from s to t using edges of G, on which each vertex of G appears once and only once. By HAM-PATH we denote the problem of determining, given G, s and t, whether G contains a Hamiltonian path from s to t. I now explain a reduction

$$\text{HAM-PATH} \leqslant \text{HAM-CYCLE}.$$

Let G,s,t constitute an input for HAM-PATH. We want to convert it to an input G' (an undirected graph) for HAM-CYCLE. We add a new vertex u to the vertex set of G in order to obtain the vertex set for G'. The edges of G' are all the edges of G plus two extra edges (u,s) and (t,u). I leave it to the reader to visualize that G' contains a Hamiltonian cycle if and only if G contains a Hamiltonian path from s to t.

The analogous problems of deciding the existence of Hamiltonian cycles and paths in directed graphs are denoted respectively by D-HAM-CYCLE and D-HAM-PATH. The above reduction for undirected graphs can be applied *mutatis mutandis* to directed graphs, and we conclude that

$$\text{D-HAM-PATH} \leqslant \text{D-HAM-CYCLE}.$$

(3) We can propose reductions in the directions opposite to those of Part (2). I first show that

$$\text{HAM-CYCLE} \leqslant \text{HAM-PATH}.$$

Let G be an undirected graph. We plan to construct an undirected graph G' with two vertices s',t' such that G' has an s',t' Hamiltonian path if and only if G has a Hamiltonian cycle. We

pick an arbitrary vertex u in G. We split u in two vertices u_1 and u_2. The vertex set of G' comprises u_1, u_2 and all vertices of G other than u. If (u,v) is an edge of G (for the particular vertex u chosen above), we add two edges (u_1,v) and (u_2,v) to G'. Finally, we add to G' each edge (v,w) of G, where neither v nor w is the same as u. We take $s' = u_1$ and $t' = u_2$. Here, splitting u to u_1, u_2 translates a Hamiltonian cycle in G to a u_1, u_2 Hamiltonian path in G'. The reader is urged to formally establish the correctness of this reduction algorithm.

A similar reduction works for directed graphs, and we obtain

$$\text{D-HAM-CYCLE} \leqslant \text{D-HAM-PATH}.$$

(4) In the last two parts, we have proved the polynomial-time equivalence of the Hamiltonian cycle and the Hamiltonian path problems, for both directed and undirected graphs. Now, I will show that it does not matter whether we consider directed graphs or undirected graphs. I first reduce HAM-CYCLE to D-HAM-CYCLE. Let G be an undirected graph with at least three vertices (an input for HAM-CYCLE). We plan to generate a directed graph G' to be used as an input for D-HAM-CYCLE. The vertex set of G' is the same as that of G. We replace each edge (u,v) of G by two directed edges (u,v) and (v,u) for G'. A little thought reveals that G' contains a directed Hamiltonian cycle if and only if G contains an undirected Hamiltonian cycle. This proves that

$$\text{HAM-CYCLE} \leqslant \text{D-HAM-CYCLE}.$$

An analogous construction for Hamiltonian paths yields

$$\text{HAM-PATH} \leqslant \text{D-HAM-PATH}.$$

(5) In Part (4) of this example, I have reduced HAM-CYCLE to D-HAM-CYCLE. I will now show a reduction in the opposite direction, that is, I will show that

$$\text{D-HAM-CYCLE} \leqslant \text{HAM-CYCLE}.$$

We start with a directed graph G and construct an undirected graph G', such that G' has an undirected Hamiltonian cycle if and only if G has a directed Hamiltonian cycle. The vertex set of G is constructed by triplicating each vertex of G. More precisely, each vertex u of G is replaced by three vertices u_-, u, u_+. For each u, we add the undirected edges (u,u_-) and (u,u_+). Moreover, for each directed edge (u,v) of G, we add the undirected edge (u_+,v_-) in G'. Thus, u_- corresponds to edges coming to u, u_+ to edges going out of u, and u to a bridge between u_- and u_+. This is how the directions in the edges of G are simulated in the undirected graph G'. It is easy to argue formally that G' satisfies the desired property mentioned above.

A similar construction for Hamiltonian paths establishes

$$\text{D-HAM-PATH} \leqslant \text{HAM-PATH}.$$

We will work with sufficiently more complicated reduction algorithms later in this chapter. Let us now look at the following definitions.

Definition A problem P is called NP-Hard if every problem $P' \in$ NP is reducible in polynomial-time to P, that is, if $P' \leqslant P$ for every $P \in$ NP.

Definition A problem P is called NP-Complete if it satisfies the following two conditions.

(1) $P \in$ NP.

(2) P is NP-Hard.

The importance of NP-Complete problems stems from the following result.

Proposition If any NP-Complete problem can be solved in polynomial time, then P = NP. Conversely, if some NP-Complete problem is not solvable in polynomial time, then P $\neq$ NP.

Proof Let P be an NP-Complete problem that has a polynomial-time algorithm A. Take any problem $P' \in$ NP. There exists a polynomial-time reduction algorithm R' from P' to P. But then, R' followed by A solves P' in polynomial time. Thus, NP $\subseteq$ P.

For proving the converse, note that an NP-Complete problem is, by definition, in NP. •

Solving the P $\stackrel{?}{=}$ NP question is, therefore, equivalent to checking for the polynomial-time solvability of NP-Complete problems. However, being merely able to define the class of NP-Complete problems does not suffice. We need specific examples of such problems. Working on concrete problems is expected to offer better insight.

Are there specific NP-Complete problems? Yes, there are! Stephen Cook and Leonid Levin were the first to identify such problems. They independently proved that a problem known as SAT is NP-Complete. It is no joke to furnish such a proof, since the class NP is quite large (infinite), and the fact that every problem in NP reduces to SAT cannot be proved by prescribing reductions on a problem-by-problem basis. On the contrary, we need a *generic* reduction algorithm from an arbitrary problem in NP to SAT. It requires a considerable amount of formalism (like the notion of Turing machines) in order to explain the proof of Cook and Levin. This book does not deal with these formalisms, and so a proof of the Cook-Levin theorem is omitted here. I believe that a proof of this theorem fits better in a course/book on formal languages and automata theory.

I will, however, explain what the problem SAT is. It is a decision problem that deals with the satisfiability of Boolean formulas. A *Boolean variable* is a variable that can assume only two possible values: 0 (false/no) and 1 (true/yes). A *Boolean function* in n Boolean variables $x_1, x_2, \ldots, x_n$ is a function that evaluates to 0 or 1 for each of the 2^n possible values assumed by $x_1, x_2, \ldots, x_n$. Any Boolean function can be expressed as a function of the variables x_i joined together by the Boolean operators $\vee$ (*disjunction* a.k.a. *OR*), $\wedge$ (*conjunction* a.k.a. *AND*), and $\neg$ (*negation*, also denoted by bar). The following tables describe these basic operations on Boolean variables.

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	$\bar{x}$
0	1
1	0

A Boolean function ϕ is said to be *satisfiable* if ϕ evaluates to 1 for some value of the input variables. It is called *unsatisfiable* if it evaluates to 0 for all possible values of the input variables.

Example

The function $\phi(x_1,x_2,x_3) = \overline{x_1} \wedge (x_2 \vee \overline{x_3})$ is satisfiable since it evaluates to 1, for example, for $x_1 = 0$, $x_2 = 1$ and $x_3 = 1$.

On the other hand, the function $\psi(x_1,x_2) = (x_1 \vee \overline{x_2}) \wedge \overline{x_1} \wedge x_2$ is unsatisfiable. This unsatisfiability can be verified by considering all of the four values taken by x_1,x_2. Alternatively, note that in order to have $\psi(x_1,x_2) = 1$, the three sub-expressions $x_1 \vee \overline{x_2}$, $\overline{x_1}$, and x_2 must each evaluate to 1. The last two sub-expressions imply that we must have $x_1 = 0$ and $x_2 = 1$. But then, $x_1 \vee \overline{x_2}$ evaluates to 0.

A *literal* is either a Boolean variable x or its complement $\bar{x}$. A disjunction (OR) of literals is called a *clause*. A Boolean formula is said to be in the *conjunctive normal form* or in *CNF* if it is expressed as a conjunction (AND) of clauses. If each clause of a Boolean formula in CNF contains exactly k literals, the formula is said to by in the *k-conjunctive normal form* or in *k-CNF*.

Example

Every Boolean function ϕ can be expressed in CNF. One possibility is to write $\overline{\phi}$ as a disjunction of conjunction of literals by looking at the values of the input variables, for which ϕ evaluates to 0. Using the DeMorgan's law then produces a CNF for ϕ. For example, consider $\phi(x_1,x_2,x_3) = x_1 \vee \overline{x_2} \vee (\overline{x_2 \vee \overline{x_3}})$. We have the following truth table for ϕ.

x_1	x_2	x_3	$\phi(x_1,x_2,x_3)$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

We, therefore, have $\neg\phi(x_1,x_2,x_3) = (\overline{x_1} \wedge x_2 \wedge \overline{x_3}) \vee (\overline{x_1} \wedge x_2 \wedge x_3)$. Applying the DeMorgan's law gives $\phi(x_1,x_2,x_3)$ in CNF.

$$\phi(x_1,x_2,x_3) = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}).$$

This expression is also in 3-CNF.

We are now ready to define the satisfiability problem SAT formally: Given a Boolean formula $\phi(x_1,x_2,\ldots,x_n)$ in n variables, decide whether $\phi(x_1,x_2,\ldots,x_n)$ is satisfiable. The problem CNF-SAT is the following: Given a Boolean formula $\phi(x_1,x_2,\ldots,x_n)$ in CNF, decide whether $\phi(x_1,x_2,\ldots,x_n)$ is satisfiable. Finally, for $k \in \mathbb{N}$, define the problem k-CNF-SAT as: Given a Boolean formula $\phi(x_1,x_2,\ldots,x_n)$ in k-CNF, decide whether $\phi(x_1,x_2,\ldots,x_n)$ is satisfiable.

We assume that a Boolean formula is provided with a minimum number of parentheses. The length of the Boolean formula $\phi(x_1,x_2,\ldots,x_n)$ (in CNF or otherwise) is taken as the input size. If

ϕ is satisfiable, a specification of the values of the input variables, for which ϕ evaluates to 1, is a certificate. Clearly, this certificate can be verified in polynomial time of the input size. That is, each of the above satisfiability problems is in NP. However, there is a total of 2^n values for the input variables. Evaluating ϕ at all these values requires $\Omega(2^n)$ running time. If the input size is a polynomial (say, linear) in n, then this running time is not polynomial in the input size.

Let me now state (without proof) two important theorems.

Cook-Levin's Theorem SAT is NP-Complete.

Theorem CNF-SAT is NP-Complete.

What about the problem k-CNF-SAT? We will later see that 3-CNF-SAT is NP-Complete, whereas 2-CNF-SAT $\in$ P.

24.3 Some well-known NP-complete problems

How to prove an NP-Complete problem to be so? Do we always have to take a general problem in NP and prescribe a generic reduction from that problem to the NP-Complete problem? The fortunate answer is *No*! Cook and Levin did the first basic job. The rest is somewhat easier.

Suppose that a problem P is already proved to be NP-Complete, and we furnish a polynomial-time reduction from P to P'. Since every problem in NP reduces to P in polynomial time, it follows that every problem in NP reduces to P' too in polynomial time. Thus, P' is NP-Hard. If, in addition, P' belongs to NP, then P' is NP-Complete. The strategy is, therefore, to work out polynomial-time reductions from known NP-Complete problems to new NP-Complete problems. Providing reductions between specific problems is usually much simpler than providing generic reduction algorithms.

Cook and Levin have proved that SAT and CNF-SAT are NP-Complete. These are our first examples of NP-Complete problems. Reducing these problems (in polynomial time) to other problems in NP enlarges the set of known NP-Complete problems. We reduce problems in this enlarged set to other problems in NP in order to enlarge the class of NP-Complete problems further. And so on. At present, thousands of problems have been identified as NP-Complete.

24.3.1 3-CNF satisfiability problem

As a simple example of this reduction technique, I will prove that 3-CNF-SAT is NP-Complete. Clearly 3-CNF-SAT $\in$ NP. I now demonstrate a reduction from CNF-SAT to 3-CNF-SAT. Let ϕ be a Boolean formula in CNF (that is, an input for CNF-SAT). We convert ϕ to a Boolean formula ψ in 3-CNF such that ψ is satisfiable if and only if ϕ is. First, note that ϕ is already in CNF, that is, an AND of clauses. Let us write $\phi = \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_k$, where each α_i is a clause. It suffices to convert each such clause α_i to 3-CNF. Let

$$\alpha = a_1 \vee a_2 \vee \cdots \vee a_l$$

be a clause with l literals. If $l = 1$, then α is equivalent to $a_1 \vee a_1 \vee a_1$. If $l = 2$, then α is equivalent to $a_1 \vee a_2 \vee a_2$. If $l = 3$, then α is already in 3-CNF. Finally, consider $l \geqslant 4$. We introduce $l-3$ new Boolean variables $b_1, b_2, \ldots, b_{l-3}$ and consider the 3-CNF formula

$$\alpha' = (a_1 \vee a_2 \vee b_1) \wedge (\overline{b_1} \vee a_3 \vee b_2) \wedge (\overline{b_2} \vee a_4 \vee b_3) \wedge \cdots \wedge (\overline{b_{l-4}} \vee a_{l-2} \vee b_{l-3}) \wedge (\overline{b_{l-3}} \vee a_{l-1} \vee a_l).$$

I will now show that α evaluates to true for some truth assignment of the input variables if and only if α' is satisfiable (for the same truth assignment of the input variables and for some truth assignment of the new variables $b_1, b_2, \ldots, b_{l-3}$). First, assume that α is true, that is, $a_i = 1$ for some i. If $i = 1$ or $i = 2$, we take $b_1 = b_2 = \cdots = b_{l-3} = 0$. For this assignment, α' evaluates to 1. If $i = l-1$ or $i = l$, then for $b_1 = b_2 = \cdots = b_{l-3} = 1$, α' evaluates to 1. Finally, if $3 \leqslant i \leqslant l-2$, we take the satisfying assignment $b_1 = \cdots = b_{i-2} = 1$ and $b_{i-1} = \cdots = b_{l-3} = 0$.

Conversely, suppose that α is false, that is, each $a_i = 0$. Then, α' can be written as

$$(b_1) \wedge (\overline{b_1} \vee b_2) \wedge (\overline{b_2} \vee b_3) \wedge \cdots \wedge (\overline{b_{l-4}} \vee b_{l-3}) \wedge (\overline{b_{l-3}}).$$

One can easily verify that this function is not satisfiable for any truth assignment of $b_1, b_2, \ldots, b_{l-3}$.

We convert each clause in ϕ to a Boolean formula in 3-CNF. It is evident that each such conversion can be done in polynomial (in fact, linear) time in the length of the clause. (Moreover, the conversion increases the length of the formula by at most a constant factor.) We have, therefore, proved that CNF-SAT $\leqslant$ 3-CNF-SAT, that is, 3-CNF-SAT is NP-Complete.

24.3.2 Hamiltonian paths and cycles

We have already established that the problems HAM-CYCLE, HAM-PATH, D-HAM-CYCLE and D-HAM-PATH are reducible from one another in polynomial time. Proving any of these problems as NP-Complete establishes that all these problems are NP-Complete (evidently, all these problems are in NP). I now show a reduction from CNF-SAT to D-HAM-PATH, that is, given a Boolean formula ϕ in CNF, we construct a directed graph G such that G has an s,t Hamiltonian path if and only if ϕ is satisfiable, where s and t are two particular vertices in G. A reduction that converts a Boolean formula to a graph is expected to involve some tricky constructions.

Figure 117 illustrates the conversion of the CNF formula $\phi = (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2} \vee x_4) \wedge (\overline{x_3})$ to a directed graph G. For each variable, we use a diamond-shaped gadget[24.3] consisting of four vertices building the outline of the diamond and a doubly connected line of vertices (Figure 118). The variable gadgets are enclosed by dashed rectangles in Figure 117 and are connected in cascade as shown. The order of the variables in this arrangement is not important.

The central line consists of $3l + 1$ vertices (besides the two vertices of the diamond outline), where l is the number of clauses in the input Boolean formula ϕ. Each clause corresponds to a pair of consecutive vertices in the line, and there is a total of $l + 1$ separator vertices.

Each clause translates to a single vertex (c_1, c_2, c_3 on the left of Figure 117). These vertices are connected to the vertices in the variable gadgets as follows. We may assume that no clause in ϕ contains repetitions of literals, and contradictory literals (like x_i and $\overline{x_i}$) do not occur simultaneously in a clause. Suppose that the clause c_j contains the literal x_i. We then look at the two vertices l_{ij} and r_{ij} in the central line of the diamond gadget corresponding to the variable x_i. We add a (directed) edge from the left vertex l_{ij} to the clause vertex c_j and another edge from c_j to the right vertex r_{ij}. On the other hand, if $\overline{x_i}$ appears in the clause c_j, we add the two edges (r_{ij}, c_j) and (c_j, l_{ij}). Finally, if neither x_i nor $\overline{x_i}$ appears in c_j, we do not add any edges.

[24.3] A *gadget* is a (compound) object that represents the converted form of a (simple) object in the input instance.