

# Algorithms: An Overview of Complexity

## P, NP, NP-complete, Halting Problem and Decidability

Suppose  $n$  is the “size” of the problem, e.g. the number elements in an array to be sorted or the number of vertices in a graph, then we say that the problem solution runs in polynomial time if there is a polynomial in  $n$ ,  $p(n)$  say, which places an upper bound on the running time of the solution.

i.e.

$$\text{running time of alg} \leq p(n)$$

where  $p(n) = c_k n^k + c_{k-1} n^{k-1} + \dots + c_1 n^1 + c_0$  for some  $k \geq 0$  and constants  $c_k, c_{k-1} \dots c_0 \geq 0$ . Here  $p(n)$  is a polynomial of order  $k$ . As  $n$  increases,  $p(n)$  will be dominated by the first term  $c_k n^k$ . In practice, algorithms with  $k > 3$  turn out to be not very usable.

e.g.

Bubble sort runs in time  $\leq cn^2$  for an array of size  $n$  where  $c$  is some constant.

Given a set of values  $S = \{x_1, x_2, x_3, \dots, x_n\}$ , find all possible subsets of  $S$ . There are  $2^n$  of these. So an algorithm to do this will involve at least  $2^n$  steps. This is an exponential function and grows much faster than any polynomial.

## Tractable and Intractable Problems

A problem is tractable if it can be solved in polynomial time. If not, it is intractable. Intractable problems are solvable in principle, it's just that they are not in practice when the problem size increases. For example, bubble sort is tractable whereas finding all subsets of a finite set is not.

## Class P

P is the class of algorithms whose complexity is a polynomial function of the problems size. Examples include minimum spanning tree algorithms, finding an Eulerian cycle thru' a graph and bubble sort. In fact most useful algorithms have degree 3 or less.

## Class NP

NP means non-deterministic polynomial. Suppose a computer program could “guess” a solution to a problem and then could check if the guessed “solution” actually solved the problem and this check could be done in polynomial time, then the program is said to be in the class NP. Non-deterministic is another word for guessing.

Most problems are in this class. NP includes P or  $P \subseteq NP$ .

Some well known problems in NP are:

- find a Hamiltonian circuit thru a graph
- find all subsets of a set

- travelling salesman problem
- knapsack problem – find the most valuable subset of  $n$  items of positive integer weights and values which fit into a knapsack of a given positive integer capacity.
- partition problem – give  $n$  positive integers, determine if it is possible to partition them into two disjoint subsets of which have equal sum.
- graph colouring – for a given graph find the smallest number of colours that need to be assigned to the graphs vertices so that no two adjacent vertices share the same colour – called the chromatic number of the graph.

It is an open question in computer science whether problems in NP are in P also, i.e.  $NP = P$  ? If so, there would be a polynomial time algorithm for finding a solution to each problem in NP. For example is there a polynomial time algorithm that solves travelling salesman problem? No one has been able to prove that there is or is not although it is widely believed that there is not. Believed that  $P \subset NP$ .

### **Class NP-complete**

A problem  $X$  is NP-complete if it is in NP and if every other problem in NP both known and unknown can be transformed into  $X$ . So by finding a solution to  $X$ , we find one to all other problems in NP. The transformation should take polynomial time.

The problems listed above in NP are NP-complete. If any of them had polynomial time algorithm as a solution, then so would all the others.

### **Halting Problem and Decidability**

Alan Turing proposed the halting problem:

Given a computer program (or algorithm) and input data for it, determine if that program will terminate or halt on that input or if it will continue indefinitely working on it. Turing proved that this problem is **undecidable** which means that there is no definite method or algorithm (or computer program) which can solve it.

That is, there is no program which can accept as input another program and its input, both as input and then decide if the other program will halt when run on its input.

Many other problems in computer science are undecidable, e.g. there is no algorithm which can look at a formula in logic (predicate calculus) and decide if it is provable.

## More Detailed Overview from a Text Book

### 10.3 *P*, *NP*, and *NP*-complete Problems

In the study of the computational complexity of problems, the first concern of both computer scientists and computing professionals is whether a given problem can be solved in polynomial time by some algorithm.

**DEFINITION 1** We say that an algorithm solves the problem in polynomial time if its worst-case time efficiency belongs to  $O(p(n))$  where  $p(n)$  is a polynomial of the problem's input size  $n$ . (Note that since we are using big-oh notation here, problems solvable in, say, logarithmic time are solvable in polynomial time as well.) Problems that can be solved in polynomial time are called *tractable*, problems that cannot be solved in polynomial time are called *intractable*.

346

Limitations of Algorithm Power

There are several reasons for drawing the intractability line in this way. First, the entries of Table 2.1 and their discussion in Section 2.1 imply that we cannot solve arbitrary instances of intractable problems in a reasonable amount of time unless such instances are very small. Second, although there might be a huge difference between the running times in  $O(p(n))$  for polynomials of drastically different degrees, there are very few useful polynomial-time algorithms with the degree of a polynomial higher than three. In addition, polynomials that bound running times of algorithms do not usually involve extremely large coefficients. Third, polynomial functions possess many convenient properties; in particular, both the sum and composition of two polynomials are always polynomials too. Fourth, the choice of this class has led to a development of an extensive theory called *computational complexity*, which seeks to classify problems according to their inherent difficulty. And according to this theory, a problem's intractability remains the same for all principal models of computations and all reasonable input encoding schemes for the problem under consideration. We just touch on some basic notions and ideas of complexity theory in this section. If you are interested in a more formal treatment of this theory, you will have no trouble finding a wealth of textbooks devoted to the subject (e.g., [Sip97], [Pap94]).

#### *P* and *NP* Problems

Most problems discussed in this book can be solved in polynomial time by some algorithm. They include computing the product and the greatest common divisor of two integers, sorting, searching (for a particular key in a list or for a given pattern in a text string), checking connectivity and acyclicity of a graph, finding a minimum spanning tree, and finding the shortest paths in a weighted graph. (You are invited to add more examples to this list.) Informally, we can think about problems that can be solved in polynomial time as the set that computer science theoreticians call *P*. A more formal definition includes in *P* only *decision problems*, which are problems with yes/no answers.

**DEFINITION 2** Class *P* is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called *polynomial*.

The restriction of *P* to decision problems can be justified by the following reasons. First, it is sensible to exclude problems not solvable in polynomial time because of their exponentially large output. Such problems do arise naturally—e.g., generating subsets of a given set or all the permutations of  $n$  distinct items—but it is apparent from the outset that they cannot be solved in polynomial time. Second, many important problems that are not decision problems in their most natural formulation can be reduced to a series of decision problems that are easier to study. For example, instead of asking about the smallest number of colors needed to color the vertices of a graph so that no two adjacent vertices are colored

the same color, we can ask whether there exists such a coloring of the graph's vertices with no more than  $m$  colors for  $m = 1, 2, \dots$  (The problem of vertex coloring with  $m$  colors is called the ***m-coloring problem***.) The first value of  $m$  in this series for which the decision problem of  $m$ -coloring has a solution solves the optimization version of the graph-coloring problem as well.

It is natural to wonder whether *every* decision problem can be solved in polynomial time. The answer to this question turns out to be no. In fact, some decision problems cannot be solved at all by any algorithm. Such problems are called ***undecidable***. A famous example was given by Alan Turing in 1936.<sup>1</sup> The problem in question is called the ***halting problem***: given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

Here is a surprisingly short proof of this remarkable fact. By way of contradiction, assume that  $A$  is an algorithm that solves the halting problem. That is, for any program  $P$  and input  $I$ ,

$$A(P, I) = \begin{cases} 1, & \text{if program } P \text{ halts on input } I; \\ 0, & \text{if program } P \text{ does not halt on input } I. \end{cases}$$

We can consider program  $P$  as an input to itself and use the output of algorithm  $A$  for pair  $(P, P)$  to construct a program  $Q$  as follows:

$$Q(P) = \begin{cases} \text{halts, if } A(P, P) = 0, \text{ i.e., if program } P \text{ does not halt on input } P; \\ \text{does not halt, if } A(P, P) = 1, \text{ i.e., if program } P \text{ halts on input } P. \end{cases}$$

Then on substituting  $Q$  for  $P$ , we obtain

$$Q(Q) = \begin{cases} \text{halts, if } A(Q, Q) = 0, \text{ i.e., if program } Q \text{ does not halt on input } Q; \\ \text{does not halt, if } A(Q, Q) = 1, \text{ i.e., if program } Q \text{ halts on input } Q. \end{cases}$$

This is a contradiction because neither of the two outcomes for program  $Q$  is possible, and it completes the proof.

Are there decidable but intractable problems? Yes, there are, but the number of known examples is small, especially of those that arise naturally rather than being constructed for the sake of a theoretical argument.

There is a large number of important problems, however, for which no polynomial-time algorithm has been found, nor has the impossibility of such an algorithm been proved. The classic monograph by M. Garey and D. Johnson [GJ79] contains a list of several hundred such problems from different areas of computer

1. This was just one of many breakthrough contributions to theoretical computer science made by the English logician and computer science pioneer Alan Turing (1912–1954). In recognition of this, the ACM—the principal society of computing professionals and researchers—has named after him an award given to researchers for outstanding contributions to theoretical computer science. A lecture given on such an occasion by Richard Karp [Kar86] provides an interesting historical account of the complexity theory development.

science, mathematics, and operations research. Here is just a small sample of some of the best-known problems that fall into this category:

**Hamiltonian circuit** Determine whether a given graph has a Hamiltonian circuit (a path that starts and ends at the same vertex and passes through all the other vertices exactly once).

**Traveling salesman** Find the shortest tour through  $n$  cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer weights).

**Knapsack problem** Find the most valuable subset of  $n$  items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.

**Partition problem** Given  $n$  positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.

**Bin packing** Given  $n$  items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.

**Graph coloring** For a given graph, find its chromatic number (the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color).

**Integer linear programming** Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and/or inequalities.

Some of these problems are decision problems. Those that are not have decision-version counterparts (e.g., the  $m$ -coloring problem for the graph-coloring problem). What all these problems have in common is an exponential (or worse) growth of choices, as a function of input size, from which a solution needs to be found. Note, however, that some problems that also fall under this umbrella can be solved in polynomial time. For example, the Eulerian circuit problem—the problem of the existence of a cycle that traverses all the edges of a given graph exactly once—can be solved in  $O(n^2)$  time by checking, in addition to the graph's connectivity, whether all the graph's vertices have even degrees. This example is particularly striking: it is quite counterintuitive to expect that the problem about cycles traversing all the edges exactly once (Eulerian circuits) can be so much easier than the seemingly similar problem about cycles visiting all the vertices exactly once (Hamiltonian circuits).

Another common feature of a vast majority of decision problems is the fact that while solving such problems can be computationally difficult, checking whether a proposed solution actually solves the problem is computationally easy; it can be done in polynomial time. (We can think of such a proposed solution as being randomly generated by somebody leaving us with the task of verifying its validity.) For example, it is easy to check whether a proposed list of vertices is a Hamiltonian circuit for a given graph with  $n$  vertices. All we need to check

is that the list contains  $n + 1$  vertices of the graph in question, the first  $n$  vertices are distinct while the last one is the same as the first, and every consecutive pair of the list's vertices is connected by an edge. This general observation about decision problems has led computer scientists to the notion of a nondeterministic algorithm.

**DEFINITION 3** A *nondeterministic algorithm* is a two-stage procedure that takes as its input an instance  $I$  of a decision problem and does the following.

Nondeterministic (“guessing”) stage: An arbitrary string  $S$  is generated that can be thought of as a candidate solution to the given instance  $I$  (but may be complete gibberish as well).

Deterministic (“verification”) stage: A deterministic algorithm takes both  $I$  and  $S$  as its input and outputs yes if  $S$  represents a solution to instance  $I$ . (If  $S$  is not a solution to instance  $I$ , the algorithm either returns no or is allowed not to halt at all.)

We say that a nondeterministic algorithm solves a decision problem if and only if for every yes instance of the problem it returns yes on some execution. (In other words, we require a nondeterministic algorithm to be capable of “guessing” a solution at least once and to be able to verify its validity. And, of course, we do not want it to ever output a yes answer on an instance for which the answer should be no.) Finally, a nondeterministic algorithm is said to be *nondeterministic polynomial* if the time efficiency of its verification stage is polynomial.

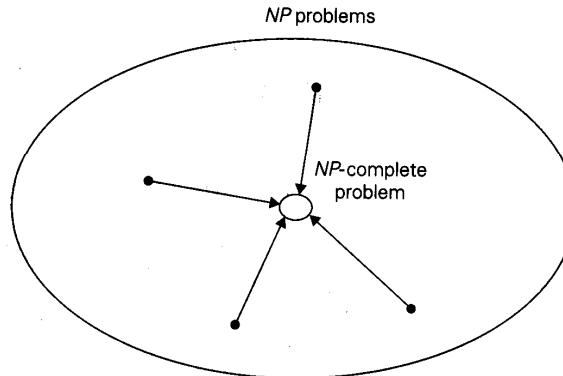
Now we can define the class of  $NP$  problems.

**DEFINITION 4** Class  $NP$  is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called *nondeterministic polynomial*.

Most decision problems are in  $NP$ . First of all, this class includes all the problems in  $P$ :

$$P \subseteq NP.$$

This is true because, if a problem is in  $P$ , we can use the deterministic polynomial-time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simply ignores string  $S$  generated in its nondeterministic (“guessing”) stage. But  $NP$  also contains the Hamiltonian circuit problem, the partition problem, as well as decision versions of the traveling salesman, the knapsack, graph coloring and many hundreds of other difficult combinatorial optimization problems cataloged in [GJ79]. The halting problem, on the other hand, is among the rare examples of decision problems that are known not to be in  $NP$ .



**FIGURE 10.6** Notion of an *NP*-complete problem. Polynomial-time reductions of *NP* problems to an *NP*-complete problem are shown by arrows.

This leads to the most important open question of theoretical computer science: Is *P* a proper subset of *NP*, or are these two classes, in fact, the same? We can put this symbolically as

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

Let us discuss these two alternatives.  $P = NP$  would imply that each of many hundreds of difficult combinatorial decision problems can be solved by a polynomial-time algorithm, although computer scientists have failed to find such algorithms despite their persistent efforts over many years. Moreover, many well-known decision problems are known to be ***NP-complete***. Informally, an *NP*-complete problem is a problem in *NP* that is as difficult as any other problem in this class because, by definition, any other problem in *NP* can be reduced to it in polynomial time (shown symbolically in Figure 10.6).

Here are more formal definitions of these concepts.

**DEFINITION 5** A decision problem  $D_1$  is said to be ***polynomially reducible*** to a decision problem  $D_2$  if there exists a function  $t$  that transforms instances of  $D_1$  to instances of  $D_2$  such that

1.  $t$  maps all yes instances of  $D_1$  to yes instances of  $D_2$  and all no instances of  $D_1$  to no instances of  $D_2$ ;
2.  $t$  is computable by a polynomial-time algorithm.

This definition immediately implies that if a problem  $D_1$  is polynomially reducible to some problem  $D_2$  that can be solved in polynomial time, then problem  $D_1$  can also be solved in polynomial time (why?).

### NP-complete Problems

Let us introduce another important notion in the computational complexity theory—that of NP-completeness.

**DEFINITION 6** A decision problem  $D$  is said to be **NP-complete** if

1. it belongs to class  $NP$ ;
2. every problem in  $NP$  is polynomially reducible to  $D$ .

The fact that closely related decision problems are polynomially reducible to each other is not very surprising. For example, let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem. The latter can be stated as the problem to determine whether there exists a Hamiltonian circuit in a given complete graph with positive integer weights whose length is not greater than a given positive integer  $m$ . We can map a graph  $G$  of a given instance of the Hamiltonian circuit problem to a complete weighted graph  $G'$  representing an instance of the traveling salesman problem by assigning 1 as the weight to each edge in  $G$  and adding an edge of weight 2 between any pair of not adjacent vertices in  $G$ . As the upper bound  $m$  on the Hamiltonian circuit length, we take  $m = n$ , where  $n$  is the number of vertices in  $G$  (and  $G'$ ). Obviously, this transformation can be done in polynomial time.

Let  $G$  be a yes instance of the Hamiltonian circuit problem. Then  $G$  has a Hamiltonian circuit, and its image in  $G'$  will have length  $n$ , making the image a yes instance of the decision traveling salesman problem. Conversely, if we have a Hamiltonian circuit of the length not larger than  $n$  in  $G'$ , then its length must be exactly  $n$  (why?) and hence the circuit must be made up of edges present in  $G$ , making the inverse image of the yes instance of the decision traveling salesman problem be a yes instance of the Hamiltonian circuit problem. This completes the proof.

The notion of NP-completeness requires, however, polynomial reducibility of *all* problems in  $NP$ , both known and unknown, to the problem in question. Given the bewildering variety of decision problems, it is nothing short of amazing that the specific examples of NP-complete problems have been actually found. Nevertheless, this mathematical feat was accomplished independently by Stephen Cook in the United States and Leonid Levin in the Soviet Union.<sup>2</sup> In his 1971

2. As it often happens in the history of science, breakthrough discoveries are made independently and almost simultaneously by several scientists. In fact, Levin introduced a more general notion than NP-completeness, which was not limited to decision problems, but his paper [Lev73] was published two years after Cook's.



paper, Cook [Coo71] showed that the so-called **CNF-satisfiability problem** is *NP*-complete. The CNF-satisfiability problem deals with boolean expressions. Each boolean expression can be represented in conjunctive normal form, such as the following expression involving three boolean variables  $x_1$ ,  $x_2$ , and  $x_3$  and their negations denoted  $\bar{x}_1$ ,  $\bar{x}_2$ , and  $\bar{x}_3$ , respectively:

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \& (\bar{x}_1 \vee x_2) \& (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3).$$

The CNF-satisfiability problem asks whether or not we can assign values *true* and *false* to variables of a given boolean expression in its CNF form to make the entire expression *true*. (It is easy to see that this can be done for the above formula: if  $x_1 = \text{true}$ ,  $x_2 = \text{true}$ , and  $x_3 = \text{false}$ , the entire expression is *true*.)

Since the Cook-Levin discovery of the first known *NP*-complete problems, computer scientists have found many hundreds, if not thousands, of other examples. In particular, the well-known problems (or their decision versions) mentioned above—Hamiltonian circuit, traveling salesman, partition, bin packing, and graph coloring—are all *NP*-complete. It is known, however, that if  $P \neq NP$ , there must exist *NP* problems that neither are in *P* nor are *NP*-complete.

For many years, the leading candidate to be such an example was the problem of determining whether a given integer prime or composite. Their algorithm does not solve, however, the related problem of factoring large composite integers, which lies at the heart of the widely used encryption method called the **RSA algorithm** [RSA78]. When this book was almost ready to go to print, Professor Manindra Agrawal and his students, Neeraj Kayal and Nitin Saxena, of the Indian Institute of Technology in Kanpur announced a discovery of a deterministic polynomial-time algorithm for primality testing [AKS02].

Showing that a decision problem is *NP*-complete can be done in two steps. First, one needs to show that the problem in question is in *NP*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents a solution to the problem. Typically, this step is easy. The second step is to show that every problem in *NP* is reducible to the problem in question in polynomial time. Because of transitivity of polynomial reduction, this step can be done by showing that a known *NP*-complete problem can be transformed to the problem in question in polynomial time (see Figure 10.7). Although such a transformation may need to be quite ingenious, it is incomparably simpler than proving the existence of a transformation for every problem in *NP*. For example, if we already know that the Hamiltonian circuit problem is *NP*-complete, its polynomial reducibility to the decision traveling salesman problem implies that the latter is also *NP*-complete (after an easy check that the decision traveling salesman problem is in class *NP*).

The definition of *NP*-completeness immediately implies that if we find a deterministic polynomial-time algorithm for just one *NP*-complete problem, then every problem in *NP* can be solved in polynomial time by a deterministic algorithm, and hence  $P = NP$ . In other words, finding a polynomial-time algorithm for one

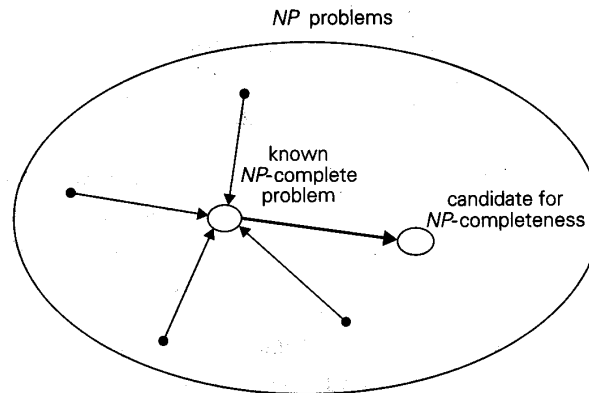


FIGURE 10.7 Proving NP-completeness by reduction

NP-complete problem would mean that there is no qualitative difference between the complexity of checking a proposed solution and finding it in polynomial time for the vast majority of decision problems of all kinds. Such implications make most computer scientists believe that  $P \neq NP$ , although nobody has been successful so far in finding a mathematical proof of this intriguing conjecture. Surprisingly, in interviews with the authors of a book about the lives and discoveries of 15 prominent computer scientists [SL98], Cook seemed to be uncertain about the eventual resolution of this dilemma while Levin contended that we should expect the  $P = NP$  outcome.

Whatever the eventual answer to the  $P \stackrel{?}{=} NP$  question proves to be, knowing that a problem is NP-complete has important practical implications for today. It means that faced with a problem known to be NP-complete, we should probably not aim at creating a revolution in computer science by designing a polynomial-time algorithm for solving all its instances. Rather, we should concentrate on several approaches that seek to alleviate the intractability of such problems. These approaches are outlined in the next chapter of the book.

### Exercises 10.3

1. A game of chess can be posed as the following decision problem: given a legal positioning of chess pieces and information about which side is to move, determine whether that side can win. Is this decision problem decidable?
2. A certain problem can be solved by an algorithm whose running time is in  $O(n^{\log_2 n})$ . Which of the following assertions is true?