


Part V: Complexity

In Part IV we described the distinction between problems that are theoretically solvable and ones that are not. In this section, we will take another look at the class of solvable problems and further distinguish among them. In particular, we will contrast problems that are “practically solvable”, in the sense that programs that solve them have resource requirements (in terms of time and or space) that can generally be met, and problems that are “practically unsolvable”, at least for large inputs, since their resource requirements grow so quickly that they cannot typically be met. Throughout our discussion, we will generally assume that if resource requirements grow as some polynomial function of problem size, then the problem is practically solvable. If they grow faster than that, then, for all but very small problem instances, the problem will generally be practically unsolvable.

27 Introduction to the Analysis of Complexity


Once we know that a problem is solvable (or a language is decidable or a function is computable), we're not done. The next step is to find an efficient algorithm to solve it.

27.1 The Traveling Salesman Problem

The *traveling salesman problem*  (or TSP for short) is easy to state: Given n cities and the distances between each pair of them, find the shortest tour that returns to its starting point and visits each other city exactly once along the way. We can solve this problem using the straightforward algorithm that first generates all possible paths that meet the requirements and then returns the shortest one. Since we must make a loop through the cities, it doesn't matter what city we start in. So we can pick any one. If there are n cities, there are $n-1$ cities that could be chosen next. And $n-2$ that can be chosen after that. And so forth. So, given n cities, the number of different tours is $(n-1)!$. We can cut the number of tours we examine in half by recognizing that the cost of a tour is the same whether we traverse it forward or backward. That still leaves $(n-1)/2$ tours to consider. So this approach quickly becomes intractable as the number of cities grows. To see why, consider the following set of observations: The speed of light is $3 \cdot 10^8$ m/sec. The width of a proton is 10^{-15} m. So, if we perform one operation in the time it takes light to cross a proton, we can perform $3 \cdot 10^{23}$ operations/sec. There have been about $3 \cdot 10^{17}$ seconds since the Big Bang. So, at that rate, we could have performed about $9 \cdot 10^{40}$ operations since the Big Bang. But $36!$ is $3.6 \cdot 10^{41}$. So there hasn't been enough time since the Big Bang to have solved even a single traveling salesman problem with 37 cities. That's fewer than one city per state in the United States.

One early application of work on the TSP was of concern to farmers rather than salesmen. The task was to conduct a survey of farmlands in Bengal in 1938. One goal of the survey planners was to minimize the cost of transporting the surveyors and their equipment from one place to the next. Another early application was the scheduling of school bus routes so that all the stops were visited and the travel distance among them was minimized.

Of course, one way to make more computations possible is to exploit parallelism. For example, there are about 10^{11} neurons in the human brain. If we think of them as operating independently, then they can perform 10^{11} computations in parallel. Each of them is very slow. But if we imagined the fast operation we described above being performed by 10^{11} computers in parallel, then there would have been time for $9 \cdot 10^{51}$ operations since the Big Bang. $43! = 6 \cdot 10^{52}$. So we still could not have solved an instance of the TSP with one city per state.

In this century, manufacturing applications of the TSP are important. Consider the problem of drilling a set of holes on a board. To minimize manufacturing time, it may be important to minimize the distance that must be traveled by the drill as it moves from one hole to the next. 

Over 50 years of research on the traveling salesman problem have led to techniques for reducing the number of tours that must be examined. For example, a dynamic programming approach that reuses partial solutions leads to an algorithm that solves any TSP instance with n cities in time that grows only as $n^2 2^n$. For large n , that is substantially better than $(n-1)!$. But it still grows exponentially with n and is not efficient enough for large problems. Despite substantial work since the discovery of that approach, there still exists no algorithm that can be guaranteed to solve an arbitrary instance of the TSP exactly and efficiently. We use the term "efficiently" here to mean that the time required to execute the algorithm grows as no more than some polynomial function of the number of cities. Whether or not such an efficient algorithm exists is perhaps the most important open question in theoretical computer science. We'll have a lot more to say about this question, which is usually phrased somewhat differently: "Does $P = NP$?"

So we do not have a technique for solving the TSP that is efficient and that is guaranteed to find the optimal solution for all problem instances. But suppose that we can compromise. Then it turns out that:

1. There are techniques that are guaranteed to find an optimal solution and that run efficiently on many (although not all) problem instances, and

2. There are techniques that are guaranteed to find a good (although not necessarily optimal) solution and to do so efficiently.

TSP solvers that make the first compromise exploit the idea of linear programming [1]. Given a problem P , a solver of this sort begins by setting up a relaxed version of P (i.e., one in which it is not necessary to satisfy all of the constraints imposed by the original problem P). Then it uses the optimization techniques of linear programming to solve this relaxed problem efficiently. The solution that it finds at this step is optimal, both for the original problem P and for the relaxed problem, but it may not be a legal solution to P . If it is, the process halts with the best tour. If the solution to the relaxed problem is not also a solution to P , it can be used to make a “cut” in the space of possible solutions. The cut is a new linear constraint with the property that the solution that was just found and rejected is on one side of the constraint while all possible solutions to the original problem P are on the other. Ideally, of course, many other candidate solutions that would also have to be rejected will also be on the wrong side of the cut. The cut is then added and a new linear programming problem, again a relaxed (but this time less relaxed) version of P , is solved. This process continues until it finds a solution that meets the constraints of the original problem P . In the worst case, only a single solution will be eliminated every time and an exponential number of tours will have to be considered. When the data come from real problems, however, it usually turns out that the algorithm performs substantially better than that. In 1954, when this idea was first described, it was used to solve an instance of the TSP with 49 cities. Since then, computers have gotten faster and the technique has been improved. In 2004, the Concorde TSP solver, a modern implementation of this idea, was used to find the optimal route that visits 24,978 cities in Sweden [2].

But what about the second compromise? It often doesn’t make sense to spend months finding the perfect tour when a very good one could be found in minutes. Further, if we’re solving a problem based on real distances, then we’ve already approximated the problem by measuring the distances to some finite precision. The notion of an exact optimal solution is theoretically well defined, but it may not be very important for real problems.

If we are willing to accept a “good” solution, then there are reasonably efficient algorithms for solving the TSP. For example, suppose that the distances between the cities satisfy the triangle inequality (i.e., given any three cities a , b , and c , the length of the path that goes directly from a to b is less than or equal to the length of the path that goes from a to c and then to b). If the cities are laid out on a plane and if the distances between them correspond to Euclidean distance (i.e., the standard measure of distance in the plane), then this constraint is met. Then there is a polynomial-time algorithm that finds a minimum spanning tree (as described in Section 28.1.6) for the city graph and uses it to construct a tour whose length is no more than twice the length of an optimal tour. And there is a more sophisticated algorithm that constructs a tour whose distance is guaranteed to be no more than 1.5 times that of the optimal one. So, for all such real-world problems, we have a “pretty good” efficient algorithm. But we’d like to do better and we usually can. For example, a solution that is known to be no more than 0.1% longer than an optimal tour has been found for a problem with 1,904,711 cities [3].

In several important ways, the TSP is representative of a much larger collection of problems that are of substantial practical interest. As we consider these problems and look for efficient algorithms to solve them, we’ll typically consider the following two important questions:

1. What do we mean by efficiency? In particular, are we concerned with:
 - the time required to solve a problem, or
 - the space required to solve it?
2. How intrinsically hard is the problem? In other words, is there some reason to believe that an algorithm that is relatively inefficient, for example one whose time complexity grows exponentially with the size of the input, is the best we are likely to be able to come up with to solve the problem at hand?


In the next three chapters, we will develop a theory that helps us to answer question two, with respect to both time and space requirements.

27.2 The Complexity Zoo

We are going to discover that, just as we were able to build a hierarchy of language classes based on the power of the automaton required to solve the membership problem, we can build a hierarchy of problem classes based on the complexity of the best algorithm that could exist to solve the problem. We'll consider problems that are intrinsically "easy" or *tractable*, by which we will mean that they can be solved in time that grows only by some polynomial function of the size of the input. And we'll consider problems (like the traveling salesman problem) that appear to be intrinsically "hard" or *intractable*, by which we mean that the time required to execute the best known algorithm grows exponentially (or worse) in the size of the input.

Some of the complexity classes that we will describe are large and play important roles in characterizing the practical solvability of the problems that they contain. For example, the first class that we will define is P, the class of problems that can be solved by a deterministic algorithm in polynomial time. All of the context-free languages (including the regular ones) are in P. So is deciding whether a number is prime or whether a graph is connected.

We will also describe a large and important class called NP-complete. No efficient algorithm for solving any NP-complete problem is known. The algorithms that we do have all require some kind of nontrivial search. For example, the traveling salesman problem is NP-complete. So is deciding whether a Boolean formula is satisfiable. (A straightforward search-based approach to solving this problem simply tries all possible assignments of truth values to the variables of an input formula.)

For a variety of reasons, people have found it useful to define many other classes of problems as well. Some of these classes are large and include languages of substantial practical interest. Many others are small and contain problems of more limited interest. There are classes that are known to be subclasses of other classes. There are classes that are known to be mutually disjoint. And there are pairs of classes whose relationship to each other is unknown. The *Complexity Zoo*  is a catalogue of known complexity classes. At the time that this sentence is being written, it contains 460 classes, with new ones still being added. We will mention only a small fraction of them in the next few chapters. But the others are defined using the same kinds of techniques that we will use. In each case, the goal is to group together a set of problems that share some significant characteristic(s).

27.3 Characterizing Problems

In order to be able to compare very different kinds of problems, we will need a single framework in which to describe them. Just as we did in Parts II, III, and IV of this book, we will describe problems as languages to be decided. So we will prove complexity results for some of the languages we have already discussed, including:

- $\{w \in \{a, b\}^* : \text{no two consecutive characters are the same}\}$ (a typical regular language),
- $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i \neq j) \text{ or } (j \neq k)\}$ (an example of a context-free language),
- $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$ (an "easy" language that is not context-free), and
- $\text{SAT} = \{\langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable}\}$ (a "hard" language that is not context-free).

We will describe both time and space complexity in terms of functions that are defined only for deciding Turing machines (i.e., Turing machines that always halt). So our discussion of the complexity of languages will be restricted to the decidable languages. Thus we will not be able to make any claims about the complexity of languages such as:

- $H = \{\langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w\}$, or
- $\text{PCP} = \{\langle P \rangle : \text{the Post Correspondence Problem instance } P \text{ has a solution}\}$.

If we were not restricting our attention to decision problems (whose output is a single bit), we might discover problems that appear hard simply because they require very long answers. For example, consider the Towers of Hanoi problem, which we describe in [C 798](#). Suppose that we wanted to describe the complexity of the most efficient algorithm that, on input n , outputs a sequence of moves that would result in n disks being moved from one pole to another. It is possible to prove that the shortest such sequence contains $2^n - 1$ moves. So any algorithm that solves this problem must run for at least $2^n - 1$ steps (assuming that it takes at least one step to write each move). And it needs at least $2^n - 1$ memory cells to store the output sequence as it is being built. Regardless of how efficiently each move can be chosen,

both the time complexity and the space complexity of any algorithm that solves this problem must be exponential simply because the length of the required answer is.

Contrast this with the traveling salesman problem. Given n cities, a solution is an ordered list of the n cities. So the length of a solution is approximately the same as the length of the input. The complexity of solving the problem arises not from the need to compose a large answer but from the apparent need to search a large space of possible short answers. By choosing to cast all of our problems as decision problems, we standardize (to one bit) the length of the solutions that will be produced. Then we can compare problems by asking about the complexity, with respect to time or space or both, of computing that one bit. (We will see, at the end of the next section, how the traveling salesman problem can be converted to a decision problem.)

27.3.1 Choosing an Encoding

Recall that we argued, in Section 3.2, that restricting our attention to the broad task of language recognition did not tie our hands behind our backs since other kinds of problems can be encoded as languages to be decided. So, for example, we will prove complexity results for some languages that are derived from questions we might ask about graphs. For example, we can analyze the complexity of:

- $\text{CONNECTED} = \{ \langle G \rangle : G \text{ is an undirected graph and } G \text{ is connected} \}$. An undirected graph is *connected* iff there exists a path from each vertex to every other node.
- $\text{HAMILTONIAN-CIRCUIT} = \{ \langle G \rangle : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian circuit} \}$. A *Hamiltonian circuit* is a path that starts at some vertex s , ends back in s , and visits each other vertex in G exactly once.

When our focus was on decidability, we did not concern ourselves very much with the nature of the encodings that we used. One exception to this arose in Section 3.2, when we showed one encoding for an integer sum problem that makes the resulting language regular, while a different encoding results in a nonregular language.

But now we want to make claims not just about decidability but about the efficiency of decidability. In particular, we are going to want to describe both the time and the space requirements of a deciding program as a function of the length of the program's input. So it may matter what encoding we choose (and thus how long each input string is). Most of the time, it will be obvious what constitutes a reasonable encoding.

One important place where it may not be obvious is the question of what constitutes a reasonable encoding of the natural numbers. We will take as reasonable an encoding in any base greater than or equal to 2. So we'll allow, for example, both binary and decimal encodings. We will not consider unary encodings. The reason for this distinction is straightforward: it takes n characters to encode n in unary (letting the empty string stand for 0, 1 for 1, and so forth). But for any base $b \geq 2$, the string encoding of n base b has length $\lfloor \log_b n \rfloor + 1$ (where $\lfloor x \rfloor$, read as "floor of x ", is the largest natural number less than x). So the length of the encoding grows only as the logarithm of n , rather than as n . Looked at from the other direction, the length of the string required to encode n in unary grows as 2^k , where k is the length of the string required to encode n in any base $b \geq 2$.

As long as we consider only bases greater than 1, the choice of base changes the length of any number's encoding only by some constant factor. This is true since, for any two positive integers a and b :

$$\log_a x = \log_a b \cdot \log_b x.$$

As we'll see shortly, we are going to ignore constant factors in almost all of our analyses. So, in particular, the constant $\log_a b$ will not affect the analyses that we will do. We'll get the same analysis with any base $b \geq 2$. With this encoding decision in hand, we'll be able to analyze the complexity of languages such as:

- $\text{PRIMES} = \{ w : w \text{ is the binary encoding of a prime number} \}$.

But keep in mind one consequence of this encoding commitment: Consider any program P that implements a function on the natural numbers. Suppose that, given the number k as input, P executes $c_1 \cdot k$ steps (for some constant c_1). It

might seem natural to say P executes in time that is linear in the size of its input. But the length of the actual input to P will be $\log_b k$, where b is greater than 1. So, if we describe the number of steps P executes as a function of the length of its input, we will get $c_2 \cdot 2^{\log k}$. Thus P executes in time that grows exponentially in the length of its input.

What about encodings for graph problems such as the ones we mentioned above? We consider two reasonable encodings for a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. Let $n = |V|$ (the number of vertices in V). For both encodings, we begin by naming the vertices with the integers from 1 to n . Then we may:

- Represent G as a list of edges. This is the technique that we used in Example 3.6. We will represent each vertex with the binary string that encodes its name. We will represent an edge by the pair of binary strings corresponding to the start and the end vertices of the edge. Then we can represent G by a sequence of edges. The binary strings will be separated by the character /, and we'll begin each encoding with the binary encoding of n . Thus the string 101/1/10/10/11/1/100/10/101 would encode a graph with five vertices and four edges. The maximum number of edges in G is n^2 (or $n^2/2$ if G is undirected). The number of characters required to encode a single vertex is $\lfloor \log_2 n \rfloor + 1$. The number of characters required to encode a single edge plus the delimiter ahead of it is then $2 \cdot \lfloor \log_2 n \rfloor + 4$. So the maximum length of the string that encodes G is bounded by:

$$n^2(2 \cdot \log_2 n + 4) + \log_2 n.$$

- Represent G as an adjacency matrix, as described in § 566. The matrix will have n rows and n columns. The value stored in cell (i, j) will be 1 if G contains an edge from vertex i to vertex j ; it will be 0 otherwise. So the value of each cell can be encoded as a single binary digit and the entire matrix can be encoded as a binary string of length:

$$n^2.$$

In either case, the size of the representation of G is a polynomial function of the number of vertices in G . The main question that we are going to be asking about the problems we consider is whether or not there exists an algorithm that solves the problem in some amount of time that grows as no more than some polynomial function of the size of the input. In that case, the answer will be the same whether we describe the size of G as simply the number of vertices it contains or we describe it as the length of one of the two string encodings (an edge list or an adjacency matrix) that we just described.

27.3.2 Converting Optimization Problems into Languages

But now let's return to the traveling salesman problem. One way to think of the TSP is that it is the Hamiltonian circuit problem with a twist: We've added distances (or, more generally) costs to the edges. And we're no longer interested simply in knowing whether a circuit exists. We insist on finding the shortest (or cheapest) one. We call problems like TSP, in which we must find the "best" solution (for some appropriate definition of "best"), **optimization problems**.

We can convert an optimization problem into a decision problem by placing a bound on the cost of any solution that we will accept. So, for example, we will be able to analyze the complexity of the language:

- TSP-DECIDE = $\{ \langle G, cost \rangle : \langle G \rangle$ encodes an undirected graph with a positive distance attached to each of its edges and G contains a Hamiltonian circuit whose total cost is less than $cost$ $\}$.

It may feel that we have lost something in this transformation. Suppose that what we really want to know is how hard it will be to find the best Hamiltonian circuit in a graph. The modified form of the problem that we have described as TSP-DECIDE seems in some sense easier, since we need only answer a yes/no question and we're given a bound above which we need check no paths. If we found an efficient algorithm that decided TSP-DECIDE, we might still not have an efficient way of solving the original problem. If, on the other hand, there is no efficient procedure for deciding TSP-DECIDE, then there can be no efficient procedure for solving the original problem (since any such procedure could be turned into an efficient procedure for deciding TSP-DECIDE). The time required to decide TSP-DECIDE is a lower bound on the time required to solve the original problem. And what we're going to see is that no efficient procedure for deciding TSP-DECIDE is known and it appears unlikely that one exists.

27.4 Measuring Time and Space Complexity

Before we can begin to analyze problems to determine how fundamentally hard they are, we need a way to analyze the time and space requirements of specific programs.

27.4.1 Choosing a Model of Computation

If we are going to say that a program, running on some particular input, executes p steps or uses m memory locations, we need to know what counts as a step or a memory location. Consider, for example, the following simple function *tally*, which returns the product of the integers in an input array:

```
tally (A: vector of n integers, n: integer) =  
  result = 1.  
  For i = 1 to n do:  
    result = result * A[i].  
  end.  
  Return result.
```

Suppose that *tally* is invoked on an input vector *A* with 10 elements. How many steps does it run before it halts? One way to answer the question would be to count each line of code once for each time it is executed. So the initialization of *result* is done once. The multiplication is done 10 times. The return statement is executed once. But how shall we count the statement “for $i = 1$ to n do:” and the end statement? We could count the for statement 10 times and thus capture the fact that the index variable is incremented and compared to n 10 times. Then we could skip counting the end statement entirely. Or we could count the end statement 10 times and assume that that’s where the index variable is compared to 10. So we might end up with the answer 22 (i.e., $1 + 1 + 10 + 10$, which we get if we don’t count executions of the end statement). Or we might end up with the answer 32 (i.e., $1 + 1 + 10 + 10 + 10$, which we get if we count both the end and the for statement 10 times).

As we’ll soon see, this is a difference that won’t matter in the kinds of analyses that we will want to do, since, using either metric, we can say that the number of steps grows linearly with the number of elements in *A*. But there is another problem here. Should we say that the amount of time required to increment the index variable is the same as the amount of time required to multiply two (possibly large) numbers? That doesn’t seem to make sense. In particular, as the number of elements of *A* increases, the size of *result* increases. So, depending on how integers are represented, a real computer may require more time per multiplication as the number of elements of *A* increases. In that case, it would no longer be true that the number of steps grows only linearly with the length of *A*.

Now consider an analysis of the space requirements of *tally*. One simple way to do such an analysis is to say that, in addition to the memory that holds its inputs, *tally* requires two memory locations, one to hold the index variable *i* and another to hold the accumulated product in *result*. Looked at this way, the amount of additional space required by *A* is a constant (i.e., 2), independent of the size of its input. But what happens if we again consider that the size of *result* may grow as each new element of *A* is multiplied into it. In that case, the number of bits required to encode *result* may also grow as the number of elements of *A* grows. Again the question arises, “Exactly what should we count?”

We will solve both of these problems by choosing one specific model of computation: the Turing machine. We will count execution steps in measuring time and visited tape squares in measuring space. More precisely:

- We will allow Turing machines with any fixed size tape alphabet. Note that if we made a more restrictive assumption and allowed only two tape symbols, the number of steps and tape squares might increase but only by some constant factor.
- We will allow only one-tape Turing machines. Would it matter if we relaxed this restriction and allowed multiple tapes? Recall that we showed, in Section 17.3.1, that the number of steps required to execute a program on a one-tape machine grows as at most the square of the number of steps required to execute the same program on a multiple tape machine. So, if such a factor doesn’t matter, we can allow multiple-tape machines for convenience.

- We will consider both deterministic and nondeterministic Turing machines. We will describe different complexity functions for the two of them and explore how they relate to each other. It seems likely, but no one has yet succeeding in proving, that there are problems that require exponentially more steps to solve on a deterministic machine than they do on a nondeterministic one.

Of course, we rarely care about the efficiency of actual Turing machines. (And we know that, even for some simple problems, straightforward Turing machines may seem very inefficient.) We care about the efficiency of real computers. But we showed, in Section 17.4, that the number of steps required to simulate a simple but realistic computer architecture on a one-tape deterministic Turing machine may grow as at most the sixth power of the number of steps required by the realistic machine. Almost all of the complexity analyses that we will do will ignore polynomial factors. When we are doing that, we may therefore describe programs in a more conventional programming style and count steps in the obvious way.

27.4.2 Defining Functions that Measure Time and Space Requirements

If we are given some particular Turing machine M and some particular input w , then we can determine the exact number of steps that M executes when started with w on its tape. We can also determine exactly the number of tape squares that M visits in the process. But we'd like to be able to describe M more generally and ask how it behaves on an arbitrary input.

To do that, we define two functions, *timereq* and *spacereq*. The domain of both functions is the set of Turing machines that halt on all inputs. The range of both is the set of functions that map from the natural numbers to the natural numbers. The function *timereq*(M) measures the time complexity of M ; it will return a function that describes how the number of steps that M executes is related to the length of its input. Similarly, the function *spacereq*(M) will define the space complexity of M ; it will return a function that describes the number of tape squares that M visits as a function of the length of its input.

Specifically, we define *timereq* as follows:

- If M is a deterministic Turing machine that halts on all inputs, then the value of *timereq*(M) is the function $f(n)$ defined so that, for any natural number n , $f(n)$ is the maximum number of steps that M executes on any input of length n .
- If M is a nondeterministic Turing machine all of whose computational paths halt on all inputs, then think of the set of computations that M might perform as a tree, just as we did in Section 17.3.2. We will not measure the number of steps in the entire tree of computations. Instead we will consider just individual paths and we will measure the length of the longest one. So the value of *timereq*(M) is the function $f(n)$ defined so that, for any natural number n , $f(n)$ is the number of steps on the longest path that M executes on any input of length n .

Analogously, we define *spacereq* as follows:

- If M is a deterministic Turing machine that halts on all inputs, then the value of *spacereq*(M) is the function $f(n)$ defined so that, for any natural number n , $f(n)$ is the maximum number of tape squares that M reads on any input of length n .
- If M is a nondeterministic Turing machine all of whose computational paths halt on all inputs, then the value of *spacereq*(M) is the function $f(n)$ defined so that, for any natural number n , $f(n)$ is the maximum number of tape squares that M reads on any path that it executes on any input of length n .

Notice that both *timereq*(M) and *spacereq*(M), as we have just defined them, measure the worst-case performance of M . In other words they measure the resource requirements of M on the inputs that require the most resources. An alternative approach would be to define both functions to return the average over all inputs. So, for example, we might define *timereqaverage*(M) to be the function $f(n)$ that returns the average number of steps that M executes on inputs of length n .

We have chosen to focus on worst-case performance, both because we would like to know an upper bound on the resources required to solve a problem and because it is, in most cases, easier to determine. We should keep in mind, however, that it is possible for worst-case performance and average-case performance to be very different. For example, an algorithm that exploits a hash table may take, on average, constant time to look up a value. But, if all the entries happen to hash to the same location, it may take time that is proportional to the number of entries in the table.

The fact that average-case and worst-case may be very different can be exploited by hackers. © 725.

The good news about the difference between average-case and worst-case is that, for many real problems, the worst case is very rare. For example, in Chapter 30, we will describe the design of randomized algorithms that solve some hard problems quickly with probability equal almost to one.

Example 27.1 Analyzing the Turing Machine that Decides $A^nB^nC^n$

Consider the deterministic Turing machine M that we built in Example 17.8. It decides the language $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$ and it operates as follows:

1. Move right onto w . If the first character is \square , halt and accept.
2. Loop:
 - 2.1 Mark off an a with a 1.
 - 2.2 Move right to the first b and mark it off with a 2. If there isn't one or if there is a c first, halt and reject.
 - 2.3 Move right to the first c and mark it off with a 3. If there isn't one or if there is an a first, halt and reject.
 - 2.4 Move all the way back to the left, then right again past all the 1's (the marked off a 's). If there is another a , go back to the top of the loop. If there isn't, exit the loop.
3. All a 's have found matching b 's and c 's and the read/write head is just to the right of the region of marked off a 's. Continue moving left to right to verify that all b 's and c 's have been marked. If they have, halt and accept. Otherwise halt and reject.

We can analyze M and determine $\text{timereq}(M)$ as follows: Let n be the length of the input string w . First, since we must determine the number of steps that M executes in the worst case, we will not consider the cases in which it exits the loop in statement 2 prematurely. So we consider only cases where there are at least as many b 's and c 's (in the right order) as there are a 's. In all such cases, M executes the statement-2 loop once for every a in the input.

Let's continue by restricting our attention to the case where $w \in A^nB^nC^n$. Then, each time through the loop, M must, on its way to the right, visit every square that contains an a , every square that contains a b , and, on average, half the squares that contain a c . And it must revisit them all as it scans back to the left. Since each letter occurs $n/3$ times, the average number of steps executed each time through the loop is $2(n/3 + n/3 + n/6)$. The loop will be executed $n/3$ times, so the total number of steps executed by the loop is $2(n/3)(n/3 + n/3 + n/6)$. Then, in the last execution of statement 2.4, combined with the execution of statement 3, M must make one final sweep all the way through w . That takes an additional n steps. So the total number of steps M executes is:

$$2(n/3)(n/3 + n/3 + n/6) + n.$$

Now suppose instead that $w \notin A^nB^nC^n$ because it contains either extra characters after the matched regions or extra a 's or b 's embedded in the matching regions. So, for example, w might be $aaabbbbbcccc$ or $aabbcca$. In these cases, the number of steps executed by the loop of statement 2 is less than the number we computed above (because the loop is executed fewer than $n/3$ times). Since timereq must measure the number of steps in the worst case, for any input of length n , we can therefore ignore inputs such as these in our analysis. So we can say that:

$$\text{timereq}(M) = 2(n/3)(n/3 + n/3 + n/6) + n.$$

Using ideas that we will formalize shortly, we can thus say that the time required to run M on an input of length n grows as n^2 .

Analyzing $spacereq(M)$ is simpler. M uses only those tape squares that contain its input string, plus the blank on either side of it. So we have:

$$spacereq(M) = n + 2.$$

27.5 Growth Rates of Functions

Let A be a program and suppose that $timereq(A) = 2n$. By almost any standard, A is efficient. We can probably afford to run A on any inputs that anyone can afford to construct. But now consider a program B , where $timereq(B) = 2^n$. This second program is a lot less efficient than A is. And there are inputs of quite reasonable size on which B would not yet have finished if it had started at the instant of the Big Bang. Some functions grow very much faster than others, as shown in Figure 27.1 (in which both the x -axis, corresponding to n , and the y -axis, corresponding to $f(n)$, are logarithmic).

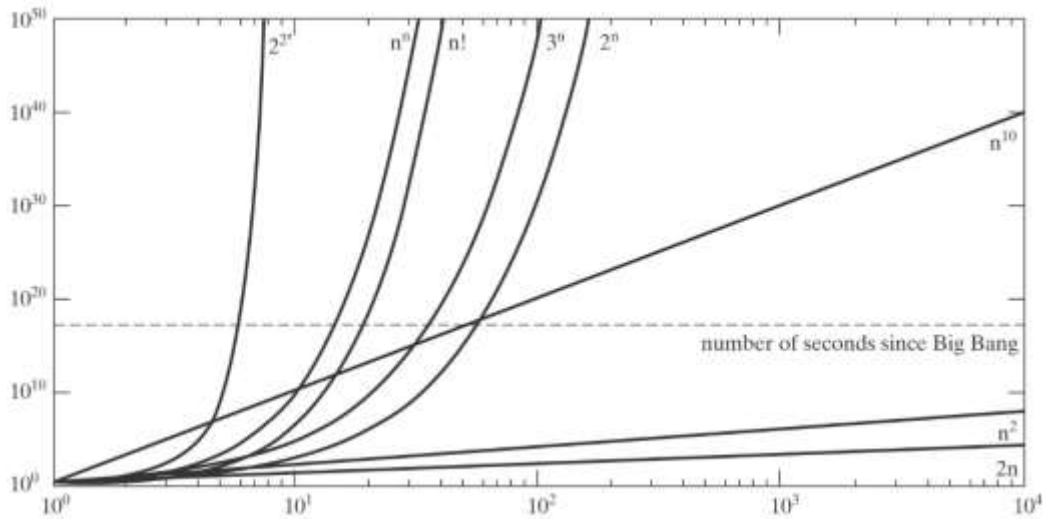


Figure 27.1 Growth rates of functions

As we develop a theory of complexity, we will find that problems that can be solved by algorithms whose time requirement is some polynomial function (e.g., $2n$) will generally be regarded as tractable. Problems for which the best known algorithm has greater than polynomial time complexity (e.g., 2^n) will generally be regarded as intractable.

Problems that are intractable in this sense are likely to remain intractable, even as computers get faster. For example, if computer speed increases by a factor of 10, we can think of $timereq$ as decreasing by a factor of 10. The only effect that has on the growth rate chart that we just presented is to shift all the lines down a barely perceptible amount.

It is possible that the one thing that might change the intractability picture for some problems is **quantum computing** [\[1\]](#). So far, the only quantum computers that have been built are so small that quantum computing has not had a practical impact on the solvability of hard problems. Someday, however, they might. But it is important to keep in mind that while quantum computing may break through intractability barriers, it cannot break through computability ones. The proof that we did of the unsolvability of the halting problem made no appeal to the physical structure of the device that was hypothesized to implement the halts function. So it applies to quantum computers as well as to current silicon-based ones.

27.6 Asymptotic Dominance

As we analyze problems and the algorithms that can solve them, we may be interested in one or both of:

- The exact amount of time or space required to run an algorithm on a problem of a given size. In this case, we may care that one algorithm runs twice as fast as another one, or that it uses half as much memory. When this happens, the functions $timereq(M)$ and $spacereq(M)$ are exactly what we need.
- The rate at which the required time or space grows as the size of the problem grows. In this case, we may be relatively unconcerned with such things as constant factors, particularly if we are facing that the total required time or space grows exponentially (or worse) with the size of the problem. In this case, $timereq(M)$ and $spacereq(M)$ provide detail that may obscure the important factors.

In the analyses that we will do in the next two chapters, we will focus on the second of these issues. Thus we will, by and large, ignore constant factors and slowly growing terms. So for example, if $timereq(M_1) = 3n^2 + 23n + 100$ and $timereq(M_2) = 25n^2 + 4n + 3$, we would like to say that the time complexity of both machines grows as n^2 . But before we embark on that analysis, we should point out that, when we are considering practical algorithms, constant factors and more slowly growing terms may matter. For instance, in Exercise 27.8), we will compare two algorithms for matrix multiplication. To multiply two $n \times n$ matrices using the obvious algorithm requires time that grows as n^3 . An alternative is Strassen's algorithm. We'll see that it requires time that grows as $n^{2.807}$. But we'll also see that Strassen's algorithm can be slower than the straightforward approach for values of n up to between 500 and 1000.

To be able to do the kind of analysis that we wish to focus on, we'll need to be able to compare two functions and ask how they behave as their inputs grow. For example, does one of them grow faster than the other? In other words, after some finite set of small cases, is one of them consistently larger than the other? In that case, we can view the larger function as describing an upper bound on the smaller one. Or perhaps they grow at the same rate. In that case, we can view either as describing a bound on the growth of the other. Or perhaps, after some finite number of small cases, one of them is consistently smaller than the other. In that case, we can view the smaller one as describing a lower bound on the other. Or maybe we can make no consistent claim about the relationship between the two functions. The theory that we are about to present is a general one that relates functions to each other. It is not tied specifically to our use, namely to measure the performance of programs. But it is exactly what we need.

One reason that we generally choose to ignore constant factors, in particular, is that the Linear Speedup Theorem tells us that, up to a point, any Turing machine can be sped up by any desired constant factor. \mathfrak{B} 658.

Consider any two functions f and g from the natural numbers to the positive reals. We define five useful relations that may hold between such functions. The first, \mathcal{O} , is introduced in \mathfrak{B} 597. We have been using it informally in the analyses of the algorithms that we have presented in Parts II, III, and IV. The other four are new:

- **Asymptotic upper bound:** $f(n) \in \mathcal{O}(g(n))$ iff there exists a positive integer k and a positive constant c such that:

$$\forall n \geq k \quad (f(n) \leq c g(n)).$$

In other words, ignoring some number of small cases (all those of size less than k), and ignoring some constant factor c , $f(n)$ is bounded from above by $g(n)$. Another way to describe this relationship, if the required limit exists, is:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty.$$

In this case, we'll say that f is "big-Oh" of g or that g asymptotically dominates or grows at least as fast as f . We can think of g as describing an upper bound on the growth of f .

- **Asymptotic strong upper bound:** $f(n) \in \mathcal{O}(g(n))$ iff, for every positive c , there exists a positive integer k such that:

$$\forall n \geq k \quad (f(n) < c g(n)).$$

In other words, whenever the required limit exists:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

In this case, we'll say that f is “little-oh” of g or that g grows strictly faster than f does.

- **Asymptotic lower bound:** $f(n) \in \Omega(g(n))$ iff there exists a positive integer k and a positive constant c such that:

$$\forall n \geq k (f(n) \geq c g(n)).$$

In other words, ignoring some number of small cases (all those of size less than k), and ignoring some constant factor c , $f(n)$ is bounded from below by $g(n)$. Another way to describe this relationship, if the required limit exists, is:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0.$$

In this case, we'll say that f is “big-Omega” of g or that g grows no faster than f .

- **Asymptotic strong lower bound:** $f(n) \in \omega(g(n))$ iff, for every positive c , there exists a positive integer k such that:

$$\forall n \geq k (f(n) > c g(n)).$$

In other words, whenever the required limit exists:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty.$$

In this case, we'll say that f is “little-omega” of g or that g grows strictly slower than f does.

- **Asymptotic tight bound:** $f(n) \in \Theta(g(n))$ iff there exists a positive integer k and positive constants c_1 , and c_2 such that:

$$\forall n \geq k (c_1 g(n) \leq f(n) \leq c_2 g(n)).$$

In other words, again assuming the limit exists:

$$0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty.$$

In this case, we'll say that f is “Theta” of g or that g is an asymptotically tight bound on f . Equivalently, we can define Θ in terms of \mathcal{O} and Ω in either of the following ways:

$f(n) \in \Theta(g(n))$ iff $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$. In other words, $f(n) \in \Theta(g(n))$ iff $g(n)$ is both an upper and a lower bound of $f(n)$.

$f(n) \in \Theta(g(n))$ iff $f(n) \in \mathcal{O}(g(n))$ and $g(n) \in \mathcal{O}(f(n))$. In other words, $f(n) \in \Theta(g(n))$ iff $f(n)$ and $g(n)$ are upper bounds of each other.

The graphs shown in Figure 27.2 may help in visualizing the bounds that are defined by \mathcal{O} , Ω , and Θ .

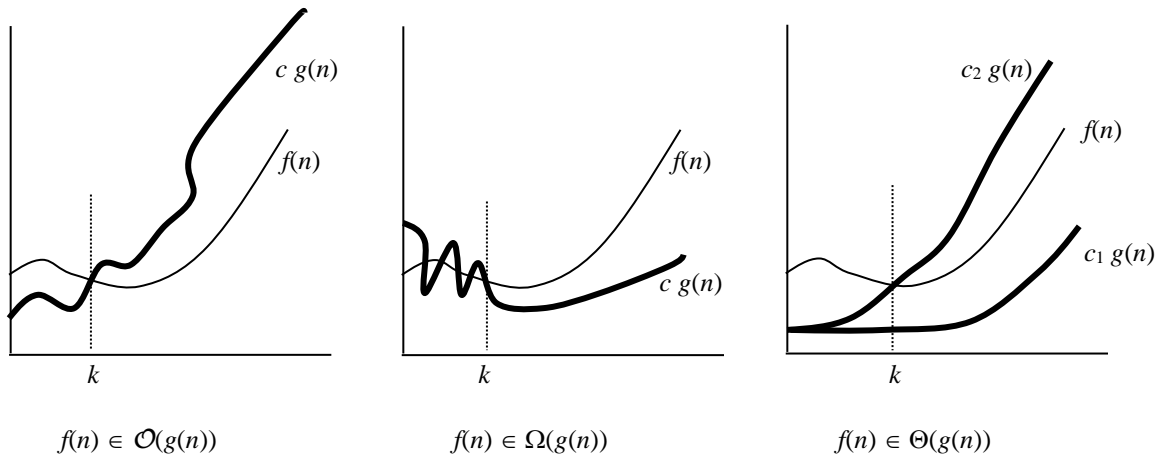


Figure 27.2 \mathcal{O} , Ω , and Θ

Example 27.2 Determining \mathcal{O} , Ω , and Θ from the Definitions

Suppose that we have analyzed the time complexity of some Turing machine M and determined that:

$$\text{timereq}(M) = 3n^2 + 23n + 100.$$

Then:

- $\text{timereq}(M) \in \mathcal{O}(n^2)$, which we can prove by finding appropriate values for c and k . A bit of experimenting will show that we could, for example, let $c = 4$ and $k = 28$ since $\forall n \geq 28 (3n^2 + 23n + 100 \leq 4n^2)$. A direct way to find c and k in the case of polynomials like this is to observe that, if $n \geq 1$, then:

$$3n^2 + 23n + 100 \leq 3n^2 + 23n^2 + 100n^2 = 126n^2.$$

So let $k = 1$ and $c = 126$.

- $\text{timereq}(M) \in \mathcal{O}(n^3)$, which we can prove by using either of the sets of values for k and c that we used above.
- $\text{timereq}(M) \in \mathcal{O}(n^3)$, which we can prove by letting, for any value of c , k be $\lceil 126/c \rceil + 1$ (where $\lceil x \rceil$, read as “ceiling of x ”, is the smallest integer greater than or equal to x). To see why such a k works, again observe that, if $n \geq 1$ then:

$$3n^2 + 23n + 100 \leq 3n^2 + 23n^2 + 100n^2 = 126n^2.$$

So we can assure that $3n^2 + 23n + 100 < c n^3$ by assuring that $126n^2 < c n^3$. Solving for n , we get $n > 126/c$.

We can guarantee that k is an integer and that it is greater than $126/c$ by setting it to $\lceil 126/c \rceil + 1$. Note that this means that $k \geq 1$, so the condition we required for the first step we did is satisfied.

- $\text{timereq}(M) \in \Omega(n)$, which we can prove by letting $c = 1$ and $k = 1$, since $\forall n \geq 1 (3n^2 + 23n + 100 \geq n)$.
- $\text{timereq}(M) \in \Omega(n^2)$, which we can prove by letting $c = 1$ and $k = 1$, since $\forall n \geq 1 (3n^2 + 23n + 100 \geq n^2)$.
- $\text{timereq}(M) \in \Theta(n^2)$, which we can prove by noting that $3n^2 + 23n + 100 \in \mathcal{O}(n^2)$ and $3n^2 + 23n + 100 \in \Omega(n^2)$. Note that $\text{timereq}(M) \notin \Theta(n)$ and $\text{timereq}(M) \notin \Theta(n^3)$.

Given two functions $f(n)$ and $g(n)$, it is possible to show that $f(n) \in \mathcal{O}(g(n))$ (and similarly for \mathcal{O} , Ω , and Θ) by showing the required constants, as we have just done. But it is much easier to prove such claims by exploiting a set of facts about arithmetic with respect to these relations. We'll state some of these facts in the next two theorems.

Theorem 27.1 Facts about \mathcal{O}

Theorem: Let f, f_1, f_2, g, g_1 , and g_2 be functions from the natural numbers to the positive reals, let a and b be arbitrary real constants, and let $c, c_0, c_1, \dots, c_k, s, t$ be any positive real constants. Then:

1. $f(n) \in \mathcal{O}(f(n))$.
2. Addition:
 - 2.1. $\mathcal{O}(f(n)) = \mathcal{O}(f(n) + c_0)$ (if we make the assumption, which will always be true for the functions we will be considering, that $1 \in \mathcal{O}(f(n))$).
 - 2.2. If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $f_1(n) + f_2(n) \in \mathcal{O}(g_1(n) + g_2(n))$.
 - 2.3. $\mathcal{O}(f_1(n) + f_2(n)) = \mathcal{O}(\max(f_1(n), f_2(n)))$.
3. Multiplication:
 - 3.1. $\mathcal{O}(f(n)) = \mathcal{O}(c_0 f(n))$.
 - 3.2. If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $f_1(n) f_2(n) \in \mathcal{O}(g_1(n) g_2(n))$.
4. Polynomials:
 - 4.1. If $a \leq b$ then $\mathcal{O}(n^a) \subseteq \mathcal{O}(n^b)$.
 - 4.2. If $f(n) = c_j n^j + c_{j-1} n^{j-1} + \dots + c_1 n + c_0$ then $f(n) \in \mathcal{O}(n^j)$.
5. Logarithms:
 - 5.1. For a and $b > 1$, $\mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$.
 - 5.2. If $0 < a < b$ and $c > 1$ then $\mathcal{O}(n^a) \subseteq \mathcal{O}(n^a \log_c n) \subseteq \mathcal{O}(n^b)$.
6. Exponentials (including the fact that exponentials dominate polynomials):
 - 6.1. If $1 < a \leq b$ then $\mathcal{O}(a^n) \subseteq \mathcal{O}(b^n)$.
 - 6.2. If $a \geq 0$ and $b > 1$ then $\mathcal{O}(n^a) \subseteq \mathcal{O}(b^n)$.
 - 6.3. If $f(n) = c_{j+1} 2^n + c_j n^j + c_{j-1} n^{j-1} + \dots + c_1 n + c_0$, then $f(n) \in \mathcal{O}(2^n)$.
 - 6.4. $\mathcal{O}(n^r 2^n) \subseteq \mathcal{O}(2^{(n^s)})$, for some s .
7. Factorial dominates exponentials: If $a \geq 1$ then $\mathcal{O}(a^n) \subseteq \mathcal{O}(n!)$.
8. Transitivity: If $f(n) \in \mathcal{O}(f_1(n))$ and $f_1(n) \in \mathcal{O}(f_2(n))$ then $f(n) \in \mathcal{O}(f_2(n))$.

Proof: Proofs of these claims, based on the definition of \mathcal{O} , are given in \mathfrak{B} 653 or left as exercises. ■

We can summarize some of the key facts from Theorem 27.1 as follows, with the caveat that the constants a, b, c , and d must satisfy the constraints given in the theorem:

$$\mathcal{O}(c) \subseteq \mathcal{O}(\log_a n) \subseteq \mathcal{O}(n^b) \subseteq \mathcal{O}(d^n) \subseteq \mathcal{O}(n!).$$

In other words, factorial dominates exponentials, which dominate polynomials, which dominate logarithms, which dominate constants.

Theorem 27.2 Facts about \mathcal{O}

Theorem: Given any functions f and g from the natural numbers to the positive reals:

- $f(n) \notin \mathcal{O}(f(n))$
- $\mathcal{O}(f(n)) \subset \mathcal{O}(f(n))$

Proof: Proofs of these claims, based on the definitions of \mathcal{O} and \mathcal{O} , are given in \mathfrak{B} 658. ■

Example 27.3 Determining \mathcal{O} and \mathcal{o} from the Properties Theorems

In Example 27.1, we analyzed the time complexity of the Turing machine M and determined that:

$$\begin{aligned} \text{timereq}(M) &= 2(n/3)(n/3 + n/3 + n/6) + n. \\ &= (5/9)n^2 + n. \end{aligned}$$

So:

- $\text{timereq}(M) \in \mathcal{O}(n^2)$. It is also true that $\text{timereq}(M) \in \mathcal{O}(n^3)$.
- $\text{timereq}(M) \in \mathcal{o}(n^3)$.

We've defined the relations \mathcal{O} , \mathcal{o} , Ω , and Θ because each of them is useful in characterizing the way in which $\text{timereq}(M)$ and $\text{spacereq}(M)$ grow as the length of the input to M increases. $\Theta(f(n))$ provides the most information since it describes the tightest bound on the growth of $f(n)$. But most discussions of complexity rely more extensively on \mathcal{O} for two reasons:

- Even when analyzing a particular machine M , it may be easier to prove a claim about $\mathcal{O}(\text{timereq}(M))$ than about $\Theta(\text{timereq}(M))$ (and similarly about $\text{spacereq}(M)$). In this case, it is conventional to make the strongest claim that can be proved. So, for example, if $\text{timereq}(M) \in \mathcal{O}(n^3)$ then it must also be true that $\text{timereq}(M) \in \mathcal{O}(n^4)$. But if we can prove the former claim, then that is the one we will make. This is the convention that we have used in analyzing algorithms in Parts II, III, and IV of this book.
- In Chapters 28, 29, and 30, we will move from discussing individual algorithms for deciding a language to making claims about the inherent complexity of a language itself. We'll base those claims on the best known algorithm for deciding the language. Since we often cannot prove that no better algorithm can exist, we will be unable to make any claim about a lower bound on the complexity of the language. Thus \mathcal{O} will be the best that we can do.

It is common to say, informally, “ M is $\mathcal{O}(f(n))$ ”, when we mean that $\text{timereq}(M) \in \mathcal{O}(f(n))$. We will do this when it causes no confusion. Similarly, we'll say that M is *polynomial* or that M implements a *polynomial-time algorithm* whenever $\text{timereq}(M) \in \mathcal{O}(f(n))$ for some polynomial function f .

27.7 Algorithmic Gaps

Our goal, in the next three chapters, is to characterize problems by their inherent difficulty. We can close the book on the complexity of a problem L if we can show all of the following:

1. There exists an algorithm that decides L and that has complexity C_1 .
2. Any algorithm that decides L must have complexity at least C_2 .
3. $C_1 = C_2$.

The existence of an algorithm as described in point 1 imposes an upper bound on the inherent complexity of L since it tells us that we can achieve C_1 . The existence of a proof of a claim as described in point 2 imposes a lower bound on the inherent complexity of L since it tells us that we can't do better than C_2 . If $C_1 = C_2$, we are done.

What we are about to see is that, for many interesting problems, we are not done. For all of the problems we will consider, some algorithm is known. So we have an upper bound on inherent complexity. But, for many of these problems, only very weak lower bounds are known. Proving lower bounds turns out to be a lot harder than proving upper bounds. So, for many problems, there is a gap, and sometimes a very significant one, between the best known lower bound and the best known upper bound. For example, the best known deterministic algorithm for solving the traveling salesman problem exactly has $\text{timereq} \in \mathcal{O}(2^{(n^k)})$. But it is unknown whether this is the best we can do. In particular, no one has been able to prove that there could not exist a deterministic, polynomial time algorithm for TSP-DECIDE.

The complexity classes that we are about to define will necessarily be based on the facts that we have. Thus they will primarily be defined in terms of upper bounds. We will group together problems for which algorithms of similar complexity are known. We must remain agnostic, for now, on several questions of the form, “Is class CL_1 equal to class CL_2 ?” Such questions will only be able to be answered by the discovery of new algorithms that prove stronger upper bounds or by the discovery of new proofs of stronger lower bounds.

27.8 Examples

Suppose that we have a problem that we wish to solve and an algorithm that solves it. But we’d like a more efficient one. We might be happy with one that runs, say, twice as fast as the original one does. But we would be even happier if we could find one for which the required time grew more slowly as the size of the problem increased. For example, the original algorithm might be $\mathcal{O}(2^n)$, while another one might be $\mathcal{O}(n^3)$. Sometimes we will succeed in finding such an algorithm. As we’ll see in the next couple of chapters, sometimes we won’t.

27.8.1 Polynomial Speedup

We begin with two examples for which we start with a polynomial algorithm but are nevertheless able to improve its running time.

Example 27.4 Finding the Minimum and Maximum in a List

We first consider an easy problem: Given a list of n numbers, find the minimum and the maximum elements in the list. We can convert this problem into a language recognition problem by defining the language $L = \{ \langle \text{list of numbers}, \text{number}_1, \text{number}_2 \rangle : \text{number}_1 \text{ is the minimum element of the list and } \text{number}_2 \text{ is the maximum element} \}$.

We’ll focus on the core of the decision procedure. Its job is to examine a list and find its minimum and maximum elements. We begin with a simple approach:

```

simplecompare(list: list of numbers) =
    max = list[1].
    min = list[1].
    For i = 2 to length(list) do:
        If list[i] < min then min = list[i].
        If list[i] > max then max = list[i].

```

Rather than trying to count every operation, we’ll assume that the time required by all the other operations is dominated by the time required to do the comparisons. The straightforward algorithm that we just presented requires $2(n-1)$ comparisons. So we can say that *simplecompare* is $\mathcal{O}(2n)$. Or, eliminating the constant, it is $\mathcal{O}(n)$. Can we do better? We notice that if $\text{list}[i] < \text{min}$ then it cannot also be true that $\text{list}[i] > \text{max}$. So that comparison can be skipped. We can do even better, though, if we consider the elements of the list two at a time. We first compare $\text{list}[i]$ to $\text{list}[i+1]$. Then we compare the smaller of the two to *min* and the larger of the two to *max*. This new algorithm requires only $(3/2)(n-1)$ comparisons. So, while the time complexity of all three algorithms is $\mathcal{O}(n)$, the last one requires 25% fewer comparisons than the first one did.

In the next example we return to a problem we considered in Chapter 5: given a pattern string and an input text string, does the pattern match anywhere in the text? We know that this question is decidable and that one way to answer it is to use a finite state machine. We now consider another way and examine its efficiency.

Example 27.5 String Search and the Knuth-Morris-Pratt Algorithm

Define the language:

- **STRING-SEARCH** = $\{ \langle t, p \rangle : \text{the string } p \text{ (the pattern) exists as a substring somewhere in } t \text{ (the text string)} \}$.

The following straightforward algorithm decides **STRING-SEARCH** by looking for at least one occurrence of the pattern p somewhere in t . It starts at the left and shifts p one character to the right each time it fails to find a match. (Note that the characters in the strings are numbered starting with 0.)


```

simple-string-search(t, p: strings) =
  i = 0.
  j = 0.
  While i ≤ |t| - |p| do:
    While j < |p| do:
      If t[i+j] = p[j] then j = j + 1.           /* Continue the match
      Else exit this loop.                               /* Match failed. Need to slide the pattern to the right.
    If j = |p| then halt and accept.                   /* The entire pattern matched.
    Else:
      i = i + 1.                                       /* Slide the pattern one character to the right.
      j = 0.                                             /* Start over again matching pattern characters.
  Halt and reject.                                       /* Checked all the way to the end and didn't find a match.

```

Let n be $|t|$ and let m be $|p|$. In the worst case (in which it doesn't find an early match), *simple-string-search* will go through its outer loop almost n times and, for each of those iterations, it will go through its inner loop m times. So $\text{timereq}(\text{simple-string-search}) \in \mathcal{O}(nm)$.

Can we do better? The answer is yes. We know, from Section 5.4.2, that, given a particular pattern p , we can build a deterministic finite state machine that looks for p in t and executes only n steps. But constructing that machine by hand for each new p isn't feasible if the pattern itself must also be an input to the program. We could use the following algorithm to decide STRING-SEARCH (where both t and p are input to the program):

```

string-search-using-FSMs(t, p: strings) =
  1. Build the simple nondeterministic FSM M that accepts any string that contains p as a substring.
  2. Let M' = ndfsmtodfsm(M).                               /* Make an equivalent deterministic FSM.
  3. Let M'' = minDFSM(M').                               /* Minimize it.
  4. Run M'' on t.
  5. If it accepts, accept. Else reject.

```

Step 0 of *string-search-using-FSMs* runs in n steps. And it is true that steps 1-3 need only be done once for each pattern p . The resulting machine M'' can then be used to scan as many input strings as we want. But steps 1-3 are expensive since the number of states of M' may grow exponentially with the number of states of M (i.e., with the number of characters in p).

So can we beat *string-search-using-FSMs*? In particular, can we design a search algorithm whose matching time is linear in n (the length of t) but that can be efficient in performing any necessary preprocessing of p ? The answer to this second question is also yes. One way to do it is to use the *buildkeywordFSM* algorithm, which we presented in Section 6.2.4, to build a deterministic FSM directly from the pattern. An alternative is to search directly without first constructing an FSM.

The Knuth-Morris-Pratt algorithm \boxtimes does the latter. It is a variant of *simple-string-search* that is efficient both in preprocessing and in searching. To see how it works, we'll begin with an example. Let t and p be as shown here. *Simple-string-search* begins by trying to match p starting in position 0:

```

      0 1 2 3 4 5 6 7 8
t:   a b c a b a b c a b d
p:   a b c a b d
           ×

```

We've marked with an \times the point at which *simple-string-search* notices that its first attempt to find a match has failed. *Simple-string-search* will increment i by 1, thus shifting the pattern one character to the right, and then it will try again, this time checking:

```

      0 1 2 3 4 5 6 7 8
t:    a b c a b a b c a b d
p:    a b c a b d
      x

```

But it shouldn't have had to bother doing that. It already knows what the first five characters of t are. The first one doesn't matter since the pattern is going to be shifted past it to the right. But the next four characters, $bcab$, tell it something. They are not the beginning of the pattern it is trying to match. It makes no sense to try again to match starting with the b or with the c .

Assume that a match fails. When it does, the current value of j is exactly the number of characters that were successfully matched before the failure was detected. We ignore the first of those characters since we will slide the pattern at least one character to the right and so the first matched character will never be considered again. Call the remaining $j-1$ characters the kernel. In our example, when the first mismatch was detected, j was 5, so the kernel is $bcab$. Now notice that, given a value for j , we can compute the only possible kernel just from the pattern p . It is independent of t . Specifically, the kernel that corresponds to j is composed of characters 1 through $j-1$ of p (numbering from 0 again).

Given a kernel from the last match, how do we know how far to the right we can slide the pattern before we have to try again to match it against t ? The answer is that we can slide the beginning of the pattern to the right until it is just past the kernel. But then we have to slide it back to the left to account for any overlap between the end of the kernel and the beginning of the pattern. So how far is that? To answer that question, we do the following. Start by placing the kernel on one line and the pattern, immediately to the right of it, on the line below it. So we have, in our example:

```

  b c a b
    a b c a b d

```

Now slide the pattern as far to the left as it can go subject to the constraint that, when we stop, any characters that are lined up in a single column must be identical. So, in this example, we can slide the pattern leftward by two characters, producing:

```

  b c a b
    a b c a b d

```

Thus, given this particular pattern p , if j is five when a mismatch is detected, then the next match we should try is the one that we get if we shift the pattern five characters to the right minus the two overlap characters. So we slide it three character to the right and we try:

```

      0 1 2 3 4 5 6 7 8
t:    a b c a b a b c a b d
p:    a b c a b d
      x

```

Again remember that this analysis of sliding distance is independent of the text string t . So we can preprocess a pattern p to determine what the overlap numbers are for each value of j . We will store those numbers in a table we will call T . Note that if $j = 0$ or 1, the corresponding kernel will be empty. For reasons that will become clear when we see exactly how the table T is going to be used, set $T[0]$ to -1 and $T[1]$ to 0. For the pattern $abcabd$ that we have been considering, T will be:

j	0	1	2	3	4	5
$T[j]$	-1	0	0	0	1	2
the kernel	ϵ	ϵ	b	bc	bca	bcab

Now, continuing with our example, notice something else about what should happen on the next match attempt. There were two characters of overlap between the pattern and the kernel. That means that we already know that the first two pattern characters match against the last two kernel characters and that those last two kernel characters are identical to the two text characters we would look at first. We don't need to check them again. So, each time we reposition the pattern on the text string (thus changing the index i in the search algorithm we presented above), we can also compute j , the first character pair we need to check. Rather than resetting it to 0 every time, we can jump it past the known characters and start it at the first character we actually need to check. So how far can we jump? The answer is that the new value of j can be computed by using its previous value as an index into T . The new value of j is exactly $T[j]$, since the size of the overlap is exactly the length of the substring we have already examined and thus can skip.

We can now state our new search algorithm based on these two optimizations (i.e., sliding the pattern to the right as far as possible and starting to check the next match as far to the right as possible):

```

Knuth-Morris-Pratt( $t, p$ : strings) =
   $i = 0.$ 
   $j = 0.$ 
  while  $i \leq |t| - |p|$  do:
    while  $j < |p|$  do:
      if  $t[i+j] = p[j]$  then  $j = j + 1.$            /* Continue the match
      Else exit this loop.                       /* Match failed. Need to slide the pattern to the right.
    if  $j = |p|$  then halt and accept.           /* The entire pattern matched.
    Else:
  *            $i = i + j - T[j].$                /* Slide the pattern as far as possible to the right.
  *            $j = \max(0, T[j]).$              /* Start  $j$  at the first character we actually need to check.
  Halt and reject.                             /* Checked all the way to the end and didn't find a match.

```

Knuth-Morris-Pratt is identical to *simple-string-search* except in the two lines marked on the left with asterisks. The only difference is in how i and j are updated each time a new match starts.

Looking at the algorithm, it should be clear why we assigned $T[0]$ the value -1. If a match fails immediately, we have to guarantee that the pattern gets shifted one character to the right for the next match. Assigning $T[0]$ the value -1 does that. Unfortunately though, that assignment does mean that we must treat $j = 0$ as a special case in computing the next value for j . That value must be 0, not -1. Thus the use of the *max* function in the expression that defines the next value for j .

Assuming that T can be computed and that it has the values shown above, we can now illustrate the operation of *Knuth-Morris-Pratt* on our example. At each iteration, we show the value of j (i.e., the position at which we start comparing the pattern to the text), with an underline:

```

      0 1 2 3 4 5 6 7 8
t:   a b c a b a b c a b d
p:   a b c a b d           Start with  $i = 0, j = 0.$ 
      *
      Mismatch found:  $i = 0, j = 5.$ 
      Compute new values for next match:  $i = i + j - T[j] = 0 + 5 - 2 = 3.$ 
       $j = \max(0, T[j]) = 2.$ 

t:   a b c a b a b c a b d
p:           a b c a b d
      *
      Mismatch found immediately:  $i = 3, j = 2.$ 
      Compute new values for next match:  $i = i + j - T[j] = 3 + 2 - 0 = 5.$ 
       $j = \max(0, T[j]) = 0.$ 

t:   a b c a b a b c a b d
p:           a b c a b d
      Complete match will now be found.

```

How much we can slide the pattern each time we try a match depends on the structure of the pattern. The worst case is a pattern like `aaaaaab`. Notice that every kernel for this pattern will be a string of zero or more `a`'s. That means that the pattern overlaps all the way to the left on every kernel. This is going to mean that it is never possible to slide the pattern more than one character to the right on each new match attempt. Using the technique we described above, we can build T (which describes the number of characters of overlap) for this pattern:

j	0	1	2	3	4	5	6
$T[j]$	-1	0	1	2	3	4	5
the kernel	ϵ	ϵ	a	aa	aaa	aaaa	aaaaa

Now consider what happens when we run *Knuth-Morris-Pratt* on the following example using this new pattern:

```

      0 1 2 3 4 5 6 7 8 9 10 ...
t    a a a a a a a a a a a a a a a b
p    a a a a a a b
      *
      Mismatch found:  $i = 0, j = 6$ .
      Compute new values for next match:  $i = i + j - T[j] = 0 + 6 - 5 = 1$ 
                                           $j = \max(0, T[j]) = 5$ 

t    a a a a a a a a a a a a a a a b
p    a a a a a a b
      *
      Mismatch found almost immediately:  $i = 1, j = 6$ .
      Compute new values for next match:  $i = i + j - T[j] = 1 + 6 - 5 = 2$ 
                                           $j = \max(0, T[j]) = 5$ 

t    a a a a a a a a a a a a a a a b
p    a a a a a a b
      *
      Mismatch found almost immediately:  $i = 2, j = 6$ .

```

This process continues, shifting the pattern one character to the right each time, until it finds a match at the very end of the string. But notice that, even though we weren't able to advance the pattern more than one character at each iteration, we were able to start j out at 5 each time. So we did skip most of the comparisons that *simple-string-search* would have done.

Analyzing the complexity of *Knuth-Morris-Pratt* is straightforward. Ignore for the moment the complexity of computing the table T . We will discuss that below. Assuming that T has been computed, we can count the maximum number of comparisons that will be done given a text t of length n and a pattern p of length m . Consider each character c of t . If the first comparison of p to c succeeds, then one of the following things must happen next:

- The rest of the pattern also matches. No further match attempts will be made so c will never be examined again.
- Somewhere later the pattern fails. But, in that case, c becomes part of the kernel that will be produced by that failed match. No kernel characters are ever reexamined. So c will never be examined again.

So the number of successful comparisons is no more than n . The number of unsuccessful comparisons is also no more than n since every unsuccessful comparison forces the process to stop and start over, sliding the pattern at least one character to the right. That can happen no more than n times. So the total number of comparisons is no more than $2n$ and so is $\mathcal{O}(n)$.

It remains to describe the algorithm that constructs the table T . The obvious approach is to try matching p against each possible kernel, starting in each possible position. But we would like a technique that is $\mathcal{O}(m)$, i.e., linear in the length of the pattern. Such an algorithm exists. It builds up the entries in T one at a time starting with $T[2]$ (since $T[0]$ is always -1 and $T[1]$ is always 0). The idea is the following: Assume that we have already considered a kernel of length $k-1$ and we are now considering one of length k . This new kernel is identical to the previous one except that one more character from p has been added to the right. So, returning to our first example, assume we have already processed the kernel of length 3 and observed a one character overlap (shown in the box) with the pattern:

kernel: b c a
 pattern: a b c a b d

To form the next longer kernel we add a b to the right of the previous kernel:

kernel: b c a b
 pattern: a b c a b d

Notice that there is no chance that there is now an overlap that starts to the left of the one we found at the last step. If the pattern didn't match those earlier characters of the kernel before, it still won't. There are only three possibilities:

- The match we found at the previous step can be extended by one character. That is what happens in this case. When this happens, the value of T for the current kernel is one more than it was for the last one.
- The match we found on the previous step cannot be extended. In that case, we check to see whether a new, shorter match can be started.
- Neither can the old match be extended nor a new one started. In this case, the value of T corresponding to the current kernel is 0.

Based on this observation, we can define the following algorithm for computing the table T :

```

buildoverlap(p: pattern string) =
   $T[0] = -1.$ 
   $T[1] = 0.$ 
   $j = 2$                                      /*  $j$  is the index of the element of  $T$  we are currently computing.
                                             /* It is the entry for a kernel of length  $j - 1$ .
   $k = 0.$                                      /*  $k$  is the length of the overlap from the previous element of  $T$ .
  While  $j < |p|$  do:                          /* When  $j$  equals  $|p|$ , all elements of  $T$  have been filled in.
    Compare  $p[j - 1]$  to  $p[k].$                 /* Compare the character that just got appended to the kernel to
                                             /* the next character of  $p$  to see if the current match can be
                                             /* extended.
                                             /* Extend the previous overlap by one character.
    If they are equal then:
       $T[j] = k + 1.$ 
       $j = j + 1.$                              /* We know the answer for this cell and can go on to the next.
       $k = k + 1.$                              /* The overlap length just increased by one.
    If they are not equal but  $k > 0$  then:    /* See if a shorter match is possible, starting somewhere
                                             /* in the box that enclosed the match we had before.
       $k = T[k].$                                /* Don't increment  $j$  since we haven't finished this entry yet.
    If they are not equal and  $k = 0$  then:    /* No overlap exists.
       $T[j] = 0.$ 
       $j = j + 1.$                              /* We know the answer for this cell and can go on to the next.
       $k = 0.$                                  /* The overlap length is back to 0.
  
```

Buildoverlap executes at most $2m$ comparisons (where m is the length of the pattern p). So the total number of comparisons executed by *Knuth-Morris-Pratt* on a text of length n and a pattern of length m is $\mathcal{O}(n + m)$. Particularly if either n or m is very large, this is a substantial improvement over *simple-string-search*, which required $\mathcal{O}(nm)$ comparisons.

27.8.2 Replacing an Exponential Algorithm with a Polynomial One

Sometimes we can get substantially greater speedup than we did in the last two examples. We may be able to replace one algorithm with another whose asymptotic complexity is much better. We've already seen two important examples of this:

- Given a string w , and a context-free language L , described by a grammar G , an obvious way to decide whether w is an element L is to try all the possible ways in which w might be derived using the rules of G . Alternatively, we

could try all paths through the nondeterministic PDA that can be constructed from G . But both of these approaches are $\mathcal{O}(2^n)$. Practical parsers must be substantially more efficient than that. In Chapter 15 we saw that, for many useful context-free languages, we can build linear-time parsers. We also saw that it is possible to retain generality and to parse an arbitrary context-free language in $\mathcal{O}(n^3)$ time using techniques, such as the Cocke-Kasami-Younger algorithm and the Earley algorithm, that exploit dynamic programming.

- Given a hidden Markov model (HMM) M and an observed output sequence O , an obvious way to determine the path through M that was most likely to have produced O is to try all paths through M of length $|O|$, compute their probabilities, and then choose the one with the highest such probability. But, letting n be $|O|$, this approach is $\mathcal{O}(2^n)$. If HMMs are to be useful, particularly in real-time applications like speech understanding, they have to be substantially faster than that. But, again, we can exploit dynamic programming. The Viterbi and the forward algorithms, which we described in Section 5.11.2, run in $\mathcal{O}(k^2n)$ time, where k is the number of states in M .

Whenever our first attempt to solve a problem yields an exponential-time algorithm, it will be natural to try to do better. The next example is a classic case in which that effort succeeds.

Example 27.6 Greatest Common Divisor and Euclid's Algorithm

One of the earliest problems for which an efficient algorithm replaced a very inefficient, but obvious one, is greatest common divisor (or gcd). Let n and m be integers. Then $\text{gcd}(n, m)$ is the largest integer k such that k is a factor of both n and m . The obvious way to compute gcd is:

$\text{gcd-obvious}(n, m: \text{integers}) =$

1. Compute the prime factors of both n and m .
2. Let k be the product of all factors common to n and m (including duplicates).
3. Return k .

So, for example, the prime factors of 40 are $\{2, 2, 2, 5\}$. The prime factors of 60 are $\{2, 2, 3, 5\}$. So $\text{gcd}(40, 60) = 2 \cdot 2 \cdot 5 = 20$.

Unfortunately, no efficient (i.e., polynomial-time) algorithm for prime factorization is known. So the obvious solution to the gcd problem is also inefficient.

But there is a better way. The following technique \square was known to the ancient Greeks. Although probably discovered before Euclid, one version of it appeared in Euclid's *Elements* in about 300 B.C. and so the technique is commonly called Euclid's algorithm:

$\text{gcd-Euclid}(n, m: \text{integers}) =$

1. If $m = 0$ return n .
2. Else return $\text{gcd-Euclid}(m, n \pmod{m})$, where $n \pmod{m}$ is the remainder after integer division of n by m .

To see that gcd-Euclid must eventually halt, observe that $n \pmod{m} < m$. So the second argument to gcd-Euclid is strictly decreasing. Since it can never become negative, it must eventually become 0. The proof that gcd-Euclid halts with the correct result rests on the observation that, for any integers n and m , if some integer k divides both n and m it must also divide $n \pmod{m}$. To see why this is so, notice that there exists some natural number j such that $n = jm + (n \pmod{m})$. So, if both n and jm are divisible by k , $n \pmod{m}$ must also be.

Next we analyze the time complexity of gcd-Euclid . Again, the key is that its second argument is strictly decreasing. The issue is, "How fast?" The answer is based on the observation that $n \pmod{m} \leq n/2$. To see why this is so, consider two cases:

- $m \leq n/2$: We have $n \pmod{m} < m \leq n/2$ and thus $n \pmod{m} \leq n/2$.
- $m > n/2$: Then $n \pmod{m} = n - m$. So $n \pmod{m} \leq n/2$.

We note that *gcd-Euclid* swaps its arguments on each recursive call. So, after each pair of calls, the second argument is cut at least in half. Thus, after at most $2 \cdot \log_2 m$ calls, the second argument will be equal to 0 and *gcd-Euclid* will halt. If we assume that each division has constant cost, then $\text{timereq}(\text{gcd-Euclid}) \in \mathcal{O}(\log_2(\max(n, m)))$.

We can turn the gcd problem into a language to be recognized by defining:

- RELATIVELY-PRIME = $\{\langle n, m \rangle : n \text{ and } m \text{ are integers and they are relatively prime}\}$. Two integers are *relatively prime* iff their gcd is 1.

The following procedure decides RELATIVELY-PRIME:

```
REL-PRIMEdecide( $\langle n, m \rangle$ : integers) =
  If  $\text{gcd-Euclid}(n, m) = 1$  then accept; else reject.
```

We already know that $\text{timereq}(\text{gcd-Euclid}) \in \mathcal{O}(\log_2(\max(n, m)))$. But recall that the length of the string encoding of an integer k is $\mathcal{O}(\log k)$. So, if the input to REL-PRIMEdecide has length $|\langle n, m \rangle|$, then $\max(n, m)$ may be $2^{|\langle n, m \rangle|}$. Thus $\text{timereq}(\text{REL-PRIMEdecide}) \in \mathcal{O}(\log_2(2^{|\langle n, m \rangle|})) = \mathcal{O}(|\langle n, m \rangle|)$. So REL-PRIMEdecide runs in linear time.

In Section 28.1, we will see other examples of problems that can be solved in an obvious way using an exponential-time algorithm but for which more efficient, polynomial-time algorithms also exist. But then, in Section 28.2, we'll consider a large family of problems for which no efficient solutions are known, despite the fact that substantial effort has gone into searching for them.

27.8.3 Time-Space Tradeoffs

Space efficiency and time efficiency affect the utility of an algorithm in different ways. In the early days of computing, when memory was expensive, programmers worried about small factors (and even constants) in the amount of memory required by their programs. But, in modern computers, memory is cheap, fast, and plentiful. So while it may matter to us whether one program takes twice as long as another one to run, we rarely care whether it takes twice as much memory. That is we don't care *until* our program runs out of memory and stops dead in its tracks. Time inefficiency may lead to a graceful degradation in system performance. Memory inefficiency may make a program's performance "fall off a cliff". So there are cases where we have no choice but to choose a less time-efficient algorithm in place of a more time-efficient one because the former uses less space. This is particularly likely to happen when we are solving intrinsically hard problems, in other words those where, no matter what we do, the amount of time and/or memory grows very quickly as the size of the problem increases.

Example 27.7 Search: Depth-First, Breadth-First, and Iterative Deepening

Consider the problem of searching a tree. We have discussed this problem at various points throughout this book. For example, Theorem 17.2 tells us that, for any nondeterministic deciding or semideciding Turing machine M , there exists an equivalent deterministic one. The proof (given in \mathfrak{B} 643) is by construction of a deterministic machine that conducts a search through the computational paths of M . If it finds an accepting path, then it accepts.

What search algorithm shall we use to solve problems such as this?

Depth-first search chooses one branch and follows it until it reaches either a solution or a dead-end. In the latter case, it backs up to the most recent decision point from which there still exists an unexplored branch. Then it picks one such branch and follows it. This process continues until either a solution is found or no unexplored alternatives remain. Depth-first search is easy to implement and it requires very little space (just a stack whose depth equals the length of the path that is currently being considered). But depth-first search can get stuck exploring a bad path and miss exploring a better one. For example, in the proof of Theorem 17.2, we must consider the case in which some of M 's paths do not halt. A depth-first search could get stuck in one of them and never get around to finding some other path that halts and accepts. So depth-first search cannot be used to solve this problem.

Breadth-first search explores all paths to depth one, storing each of the nodes it generates. Next it expands each of those nodes one more level, generating a new fringe of leaf nodes. Then it returns to those leaf nodes and expands all

of them one more level. This process continues until either a solution is found or no leaf nodes have any successors. Breadth-first search cannot get stuck since it explores all paths of length k before considering any paths of length $k+1$. But breadth-first search must store every partial path in memory. So the amount of space it requires grows exponentially with the depth of the search tree that it is exploring. A Turing machine has an infinite tape, so it will never run out of room. However, managing it and shifting its contents around are difficult. Real computers, though, have finite memory. So, for practical problems, breadth-first search can work very well as long as the available memory is adequate for storing all the partial paths. As soon as it is not, the search process unceremoniously stops.

Iterative deepening is a compromise between breadth-first search and depth-first search. It first explores all paths of length 1 using depth-first search. Then it starts over and explores all paths of length 2 using depth-first search. And then all paths of length 3, and so forth. Whenever it finds a solution, at some depth, it halts. The space complexity of iterative deepening is the same as for depth-first search. And its time complexity is only slightly worse than that of breadth-first search. This may seem counterintuitive, since, for each k , the search to depth k starts over; it doesn't use any of the results from the search to depth $k-1$. We present the algorithm in detail in [§ 643](#), and we analyze its complexity in [§ 647](#). In a nutshell, the reason that starting the search over every time isn't such a bad idea is that the top part of the search tree is the part that must be generated many times. But the top part is very small compared to the bottom part. Iterative deepening is the technique that we use to prove Theorem 17.2.

27.9 Exercises

- 1) Let M be an arbitrary Turing machine.
 - a) Suppose that $\text{timereq}(M) = 3n^3(n+5)(n-4)$. Circle all of the following statements that are true:
 - i) $\text{timereq}(M) \in \mathcal{O}(n)$.
 - ii) $\text{timereq}(M) \in \mathcal{O}(n^6)$.
 - iii) $\text{timereq}(M) \in \mathcal{O}(n^5/50)$.
 - iv) $\text{timereq}(M) \in \Theta(n^6)$.
 - b) Suppose that $\text{timereq}(M) = 5^n \cdot 3n^3$. Circle all of the following statements that are true:
 - i) $\text{timereq}(M) \in \mathcal{O}(n^5)$.
 - ii) $\text{timereq}(M) \in \mathcal{O}(2^n)$.
 - iii) $\text{timereq}(M) \in \mathcal{O}(n!)$.
- 2) Show a function f , from the natural numbers to the reals, that is $\mathcal{O}(1)$ but that is not constant.
- 3) Assume the definitions of the variables given in the statement of Theorem 27.1. Prove that there exists some s such that:

$$\mathcal{O}(n^s 2^n) \subseteq \mathcal{O}(2^{(n^s)}).$$
- 4) Prove that, if $0 < a < b$, then $n^b \notin \mathcal{O}(n^a)$.
- 5) Let M be the Turing machine shown in Example 17.9. M accepts the language $W \subset W = \{w \subset w : w \in \{a, b\}^*\}$. Analyze $\text{timereq}(M)$.
- 6) Assume a computer that executes 10^{10} operations/second. Make the simplifying assumption that each operation of a program requires exactly one machine instruction. For each of the following programs P , defined by its time requirement, what is the largest size input on which P would be guaranteed to halt within a year?
 - a) $\text{timereq}(P) = 5243n+649$.
 - b) $\text{timereq}(P) = 5n^2$.
 - c) $\text{timereq}(P) = 5^n$.
- 7) Let each line of the following table correspond to a problem for which two algorithms, A and B , exist. The table entries correspond to timereq for each of those algorithms. Determine, for each problem, the smallest value of n (the length of the input) such that algorithm B runs faster than algorithm A .

A	B
n^2	$572n + 4171$
n^2	$1000n \log_2 n$
$450n^2$	$n!$
$3^n + 2$	$n!$

- 8) Show that $L = \{ \langle M \rangle : M \text{ is a Turing machine and } \text{timereq}(M) \in \mathcal{O}(n^2) \}$ is not in D.
- 9) Consider the problem of multiplying two $n \times n$ matrices. The straightforward algorithm *multiply* computes $C = A \cdot B$ by computing the value for each element of C using the formula:

$$C_{i,j} = \sum_{k=1}^n A_{i,k} B_{k,j} \text{ for } i, j = 1, \dots, n.$$

Multiply uses n multiplications and $n-1$ additions to compute each of the n^2 elements of C . So it uses a total of n^3 multiplications and $n^3 - n^2$ additions. Thus $\text{timereq}(\text{multiply}) \in \Theta(n^3)$.

We observe that any algorithm that performs at least one operation for each element of C must take at least n^2 steps. So we have an n^2 lower bound and an n^3 upper bound on the complexity of matrix multiplication. Because matrix multiplication plays an important role in many kinds of applications (including, as we saw in Section 15.3.2, some approaches to context-free parsing), the question naturally arose, “Can we narrow that gap?” In particular, does there exist a better than $\Theta(n^3)$ matrix multiplication algorithm? In [Strassen 1969], Volker Strassen showed that the answer to that question is yes.

Strassen’s algorithm exploits a divide-and-conquer strategy in which it computes products and sums of smaller submatrices. Assume that $n = 2^k$, for some $k \geq 1$. (If it is not, then we can make it so by expanding the original matrix with rows and columns of zeros, or we can modify the algorithm presented here and divide the original matrix up differently.) We begin by dividing A , B , and C into 2×2 blocks. So we have:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \text{ and } C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix},$$

where each A_{ij} , B_{ij} , and C_{ij} is a $2^{k-1} \times 2^{k-1}$ matrix.

With this decomposition, we can state the following equations that define the values for each element of C :

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1}. \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2}. \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1}. \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}. \end{aligned}$$

So far, decomposition hasn’t bought us anything. We must still do eight multiplications and four additions, each of which must be done on matrices of size 2^{k-1} . Strassen’s insight was to define the following seven equations:

$$\begin{aligned} Q_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}). \\ Q_2 &= (A_{2,1} + A_{2,2})B_{1,1}. \\ Q_3 &= A_{1,1}(B_{1,2} - B_{2,2}). \\ Q_4 &= A_{2,2}(B_{2,1} - B_{1,1}). \\ Q_5 &= (A_{1,1} + A_{1,2})B_{2,2}. \\ Q_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}). \\ Q_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}). \end{aligned}$$

These equations can then be used to define the values for each element of C as follows:

$$\begin{aligned}
C_{1,1} &= Q_1 + Q_4 - Q_5 + Q_7. \\
C_{1,2} &= Q_3 + Q_5. \\
C_{2,1} &= Q_2 + Q_4. \\
C_{2,2} &= Q_1 - Q_2 + Q_3 + Q_6.
\end{aligned}$$

Now, instead of eight matrix multiplications and four matrix additions, we do only seven matrix multiplications, but we must also do eighteen matrix additions (where a subtraction counts as an addition). We've replaced twelve matrix operations with 25. But matrix addition can be done in $\mathcal{O}(n^2)$ time, while matrix multiplication remains more expensive.

Strassen's algorithm applies these formulas recursively, each time dividing each matrix of size 2^k into four matrices of size 2^{k-1} . The process halts when $k = 1$. (Efficient implementations \square of the algorithm actually stop the recursion sooner and use the simpler *multiply* procedure on small submatrices. We'll see why in part (e) of this problem.) We can summarize the algorithm as follows:

```

Strassen(A, B, k: where A and B are matrices of size 2^k) =
  If k = 1 then compute the Q's using scalar arithmetic. Else, compute them as follows:
  Q1 = Strassen((A1,1 + A2,2), (B1,1 + B2,2), k-1).
  Q2 = Strassen((A2,1 + A2,2), B1,1, k-1).
  ...
  Q7 =
  C1,1 =
  ...
  C2,2 =
  Return C.

```

/* Compute all the Q matrices as described above.

/* Compute all the C matrices as described above.

In the years following Strassen's publication of his algorithm, newer ones that use even fewer operations have been discovered \square . The fastest known technique is the Coppersmith-Winograd algorithm, whose time complexity is $\mathcal{O}(n^{2.376})$. But it is too complex to be practically useful. There do exist algorithms with better performance than *Strassen*, but, since it opened up this entire line of inquiry, we should understand its complexity. In this problem, we will analyze *timereq* of *Strassen* and compare it to *timereq* of the standard algorithm *multiply*. We should issue two caveats before we start, however: the analysis that we are about to do just counts scalar multiplies and adds. It does not worry about such things as the behavior of caches and the use of pipelining. In practice, it turns out that the crossover point for *Strassen* relative to *multiply* \square is lower than our results suggest. In addition, *Strassen* may not be as numerically stable as *multiply* is, so it may not be suitable for all applications.

- We begin by defining $mult'(k)$ to be the number of scalar multiplications that will be performed by *Strassen* when it multiplies two $2^k \times 2^k$ matrices. Similarly, let $add'(k)$ be the number of scalar additions. Describe both $mult'(k)$ and $add'(k)$ inductively by stating their value for the base case (when $k = 1$) and then describing their value for $k > 1$ as a function of their value for $k-1$.
- To find closed form expressions for $mult'(k)$ and add' requires solving the recurrence relations that were given as answers in part (a). Solving the one for $mult'(k)$ is easy. Solving the one for $add'(k)$ is harder. Prove that the following are correct:

$$\begin{aligned}
mult'(k) &= 7^k. \\
add'(k) &= 6 \cdot (7^k - 4^k).
\end{aligned}$$

- We'd like to define the time requirement of *Strassen*, when multiplying two $n \times n$ matrices, as a function of n , rather than as a function of $\log_2 n$, as we have been doing. So define $mult(n)$ to be the number of multiplications that will be performed by *Strassen* when it multiplies two $n \times n$ matrices. Similarly, let $add(n)$ be the number of additions. Using the fact that $k = \log_2 n$, state $mult(n)$ and $add(n)$ as functions of n .
- Determine values of α and β , each less than 3, such that $mult(k) \in \Theta(n^\alpha)$ and $add(k) \in \Theta(n^\beta)$.
- Let $ops(n) = mult(n) + add(n)$ be the total number of scalar multiplications and additions that *Strassen* performs to multiply two $n \times n$ matrices. Recall that, for the standard algorithm *multiply*, this total operation

count is $2n^3 - n^2$. We'd like to find the crossover point, i.e., the point at which *Strassen* performs fewer scalar operations than *multiply* does. So find the smallest value of k such that $n = 2^k$ and $ops(n) < 2n^3 - n^2$. (Hint: Once you have an equation that describes the relationship between the operation counts of the two algorithms, just start trying candidates for k , starting at 1.)

- 10) In this problem, we will explore the operation of the Knuth-Morris-Pratt string search algorithm that we described in Example 27.5. Let p be the pattern `cbacbcc`.
- Trace the execution of *buildoverlap* and show the table T that it builds.
 - Using T , trace the execution of *Knuth-Morris-Pratt*(`cbaccbacbcc`, `cbacbcc`).

28 Time Complexity Classes

Some problems are easy. For example, every regular language can be decided in linear time (by running the corresponding DFSA). Some problems are harder. For example, the best known algorithm for deciding the Traveling Salesman language TSP-DECIDE takes, in the worst case, time that grows exponentially in the size of the input. In this chapter, we will define a hierarchy of language classes based on the time required by the best known decision algorithm.

28.1 The Language Class P

The first important complexity class that we will consider is the class P, which includes all and only those languages that are decidable by a deterministic Turing machine in polynomial time. So we have:

The Class P: $L \in P$ iff there exists some deterministic Turing machine M that decides L and $\text{timereq}(M) \in \mathcal{O}(n^k)$ for some constant k .

It is common to think of the class P as containing exactly the *tractable* problems. In other words, it contains those problems that are not only solvable in principle (i.e., they are decidable) but also solvable in an amount of time that makes it reasonable to depend on solving them in real application contexts.

Of course, suppose that the best algorithm we have for deciding some language L is $\Theta(n^{1000})$ (i.e., its running time grows at the same rate, to within a constant factor, as n^{1000}). It is hard to imagine using that algorithm on anything except a toy problem. But the empirical fact is that we don't tend to find algorithms of this sort. Most problems of practical interest that are known to be in P can be solved by programs that are no worse than $\mathcal{O}(n^3)$ if we are analyzing running times on conventional (random access) computers. And so they're no worse than $\mathcal{O}(n^{18})$ when run on a one-tape, deterministic Turing machine. Furthermore, it often happens that, once some polynomial time algorithm is known, a faster one will be discovered. For example, consider the problem of matrix multiplication. If we count steps on a random access computer, the obvious algorithm for matrix multiplication (based on Gaussian elimination) is $\mathcal{O}(n^3)$. Strassen's algorithm [Str68] is more efficient; it is $\mathcal{O}(n^{2.81})$. Other algorithms whose asymptotic complexity is even lower (approaching $\mathcal{O}(n^2)$) are now known, although they are substantially more complex.

So, as we consider languages that are in P, we will generally discover algorithms whose time requirement is some low-order polynomial function of the length of the input. But we should be clear that not all languages in P have this property. In Section 28.9.1, we'll describe the time hierarchy theorems. One consequence of the Deterministic Time Hierarchy Theorem is that, for any integer $k > 1$, there are languages that can be decided by a deterministic Turing machine in $\mathcal{O}(n^k)$ time but not in $\mathcal{O}(n^{k-1})$ time. It just happens that if $k = 5000$, we are unlikely to care.

Going the other direction, if we have a problem that we cannot show to be in P, is it necessarily intractable in practice? Often it is. But there may be algorithms that solve it quickly most of the time or that solve it quickly and return the right answer most of the time. For example, prior to the recent proof (which we will mention in Section 28.1.7) that primality testing can be done in polynomial time, randomized algorithms that performed primality testing efficiently were known and commonly used [AKS02]. We'll return to this approach in Chapter 30.

28.1.1 Closure of P under Complement

One important property of the class P is that, if a language L is in P, so is its complement:

Theorem 28.1 P is Closed under Complement

Theorem: The class P is closed under complement.

Proof: For any language L , if $L \in P$ then there exists some deterministic Turing machine M that decides L in polynomial time. From M , we can build a new deterministic Turing machine M' that decides $\neg L$ in polynomial time. We use the same construction that we used to prove Theorem 20.4 (which tells us that the decidable languages are

closed under complement). M' is simply M with accepting and nonaccepting states swapped. M' will always halt in exactly the same number of steps M would take and it will accept $\neg L$. ■

For many problems that we are likely to care about, this closure theorem doesn't give us exactly the result we need. For example, we'll show below that the language $\text{CONNECTED} = \{\langle G \rangle : G \text{ is an undirected graph and } G \text{ is connected}\}$ is in P. We'd like then to be able to conclude that the related language $\text{NOTCONNECTED} = \{\langle G \rangle : G \text{ is an undirected graph and } G \text{ is not connected}\}$ is also in P. But we have the same problem that we had in analyzing languages that are defined in terms of a Turing machine's behavior:

- $\neg\text{CONNECTED} = \text{NOTCONNECTED} \cup \{\text{strings that are not syntactically legal descriptions of undirected graphs}\}$.

If, however, we can check for legal syntax in polynomial time, then we can consider the universe with respect to which the complement of CONNECTED is computed to be just those strings whose syntax is legal. Then we can conclude that NOTCONNECTED is in P if CONNECTED is. In all the examples we will consider in this book, such a syntax check can be done in polynomial time. So we will consider the complement of some language L to be the language consisting of strings of the correct syntax but without the property that defines L .

28.1.2 Languages That Are in P

We have already discussed many examples of languages that are in P:

- Every regular language is in P since every regular language can be decided in linear time. We'll prove this claim as Theorem 28.2 below.
- Every context-free language is in P since there exist context-free parsing (and deciding) algorithms that run in $\mathcal{O}(n^3)$ time on a conventional computer (and thus run in $\mathcal{O}(n^{18})$ time on a single-tape Turing machine). We'll prove this claim as Theorem 28.3 below.
- Some languages that are not context-free are also in P. One example of such a language is $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$. In Example 27.1, we analyzed M , the Turing machine that we had built to decide $A^n B^n C^n$. We showed that $\text{time}_{req}(M) = 2(n/3)(n/3 + n/3 + n/6) + n$, which, as we showed in Example 27.3, is in $\mathcal{O}(n^2)$.

The game of Nim (appropriately encoded as a decision problem in which we ask whether there is a guaranteed win for the current player) is in P. But it appears that very few “interesting” games are. © 780.

Many other languages are also in P. In the rest of this section, we show examples of proofs that languages are in P. If we can construct a one-tape, deterministic Turing machine M that decides some language L in polynomial time, then we have a proof that L is in P. But we will generally find it substantially easier just to describe a decision procedure as it would be implemented on a conventional, random access computer. Then we can appeal to Theorem 17.4, which tells us that a deterministic random access program that executes t steps can be simulated by a seven-tape Turing machine in $\mathcal{O}(t^3)$ steps. We also showed, in Theorem 17.1, that t steps of a k -tape Turing machine can be simulated in $\mathcal{O}(t^2)$ steps of a standard Turing machine. Composing these results, we have that if a random access program runs in t steps, it can be simulated by a standard Turing machine in $\mathcal{O}(t^6)$ steps. Since the composition of two polynomials is a polynomial, if we have a random access algorithm that runs in polynomial time, then it can be simulated in a polynomial number of steps on a deterministic one-tape Turing machine and so the language that it decides is in P.

We'll make one other simplifying assumption as well. It takes $\mathcal{O}(n^2)$ steps to compare two strings of length n on a one-tape Turing machine. It takes $\mathcal{O}(\log^2 n)$ to compare two integers of size n (since their string descriptions have length $\log n$). We can do similar analyses of the time required to perform arithmetic operations on numbers of size n . The key is that all of these operations can be performed in polynomial time. So, if our goal is to show that an algorithm

runs in polynomial time, we can assume that all such operations are performed in constant time (which they generally are on real computers). While not strictly true, this assumption will have no effect on any claim we may make that an algorithm runs in polynomial time.

28.1.3 Regular and Context-Free Languages

Theorem 28.2 Every Regular Language Can be Decided in Linear Time.

Theorem: Every regular language can be decided in linear time. So every regular language is in P.

Proof: Given any regular language L , there exists some deterministic finite state machine $M = (K, \Sigma, \delta, s, A)$ that decides L . From M , we can construct a deterministic Turing machine $M' = (K \cup \{s', y, n\}, \Sigma, \Sigma \cup \{\square\}, \delta', s', \{y\})$ that decides L in linear time. Roughly, M' simply simulates the steps of M , moving its read/write head one square to the right at each step and making no change to the tape. When M' reads a \square , it halts. If it is in an accepting state, it accepts; otherwise it rejects. So, if (q, a, p) is a transition in M , M' will contain the transition $((q, a), (p, a, \rightarrow))$. Because of our convention that the read/write head of M' will be just to the left of the first input character when it begins, M' will need a new start state, s' , in which it will read a \square and move right to the first input character. Also, since FSMs halt when they run out of input, while Turing machines halt only when they enter an explicit halting state, M' will need two new states: y , which will halt and accept, and n , which will halt and reject. Finally, M' will need transitions into y and n labeled \square . So, if q is a state in M (and thus also in M') and q is an accepting state in M , M' will contain the transition $((q, \square), (y, \square, \rightarrow))$. If, on the other hand, q is not an accepting state in M , M' will contain the transition $((q, \square), (n, \square, \rightarrow))$.

On any input of length n , M' will execute $n + 2$ steps. So $\text{timereq}(M') \in \mathcal{O}(n)$. ■

It is significant that every regular language can be decided in linear time. But the fact that every regular language is in P is also a consequence of the more general fact that we prove next: every context-free language is in P.

Theorem 28.3 Every Context-Free Language is in P

Theorem: Every context-free language can be decided in $\mathcal{O}(n^{18})$ time. So every context-free language is in P.

Proof: In Chapter 14, we showed that every context-free language is decidable. Unfortunately, neither of the algorithms that we presented there (*decideCFLusingGrammar* and *decideCFLusingPDA*) runs in polynomial time. But, in Section 15.3, we presented the Cocke-Kasami-Younger (CKY) algorithm, which can parse any context-free language in time that is $\mathcal{O}(n^3)$ if we count operations on a conventional computer. That algorithm can be simulated on a standard, one-tape Turing machine in $\mathcal{O}(n^{18})$ steps. ■

28.1.4 Connected Graphs

We next consider languages that describe significant properties of graphs.

One of the simplest questions we can ask about a graph is whether it is connected. A graph is **connected** iff there exists a path from each vertex to each other vertex. We consider here the problem for undirected graphs. (We can also define the related language for directed graphs.) Define the language:

- CONNECTED = $\{\langle G \rangle : G \text{ is an undirected graph and } G \text{ is connected}\}$.

Theorem 28.4 The Problem of Identifying Connected Graphs is in P

Theorem: CONNECTED is in P.

Proof: We prove that CONNECTED is in P by exhibiting a deterministic, polynomial-time algorithm that decides it:

connected($\langle G: \text{graph with vertices } V \text{ and edges } E \rangle$) =

1. Set all vertices to be unmarked.
2. Mark vertex 1.
3. Initialize L (a list that will contain vertices that have been marked but whose successors have not yet been examined) to contain just vertex 1.
4. Initialize *marked-vertices-counter* to 1.
5. Until L is empty do:
 - 5.1. Remove the first element from L . Call it *current-vertex*.
 - 5.2. For each edge e that has *current-vertex* as an endpoint do:
 - Call the other endpoint of e *next-vertex*. If *next-vertex* is not already marked then do:
 - Mark *next-vertex*.
 - Add *next-vertex* to L .
 - Increment *marked-vertices-counter* by 1.
6. If *marked-vertices-counter* = $|V|$ accept; else reject.


Connected will mark and count the vertices that are reachable from vertex 1. Since G is undirected, if there is a path from vertex 1 to some vertex n , then there is also a path from vertex n back to vertex 1. So, if there is a path from vertex 1 to every other vertex, then there is a path from every other vertex back to vertex 1 and from there to each other vertex. Thus G is connected. If, on the other hand, there is some vertex that is not reachable from vertex 1, G is unconnected.

So it remains to show that the runtime of *connected* is some polynomial function of $|\langle G \rangle|$:

- Step 1 takes time that is $\mathcal{O}(|V|)$.
- Steps 2, 3, and 4 each take constant time.
- The outer loop of step 5 can be executed at most $|V|$ times since no vertex can be put on L more than once.
 - Step 5.1 takes constant time.
 - The loop in step 5.2 can be executed at most $|E|$ times. Each time through, it requires at most $\mathcal{O}(|V|)$ time (depending on how the vertices are represented and marked).
- Step 6 takes constant time.

So the total time required to execute *connected* is $|V| \cdot \mathcal{O}(|E|) \cdot \mathcal{O}(|V|) = \mathcal{O}(|V|^2|E|)$. But note that $|E| \leq |V|^2$. So the time required to execute *connected* is $\mathcal{O}(|V|^4)$. ■

28.1.5 Eulerian Paths and Circuits

The Seven Bridges of Königsberg problem  is inspired by the geography of a town once called Königsberg, in Germany, now called Kaliningrad, in Russia. The town straddled the banks of the river Pregel and there were two islands in the river. There were seven bridges connecting the river banks and the islands as shown in Figure 28.1. The problem is this: can a citizen of Königsberg take a walk through the town (starting anywhere she likes) and cross each bridge exactly once?

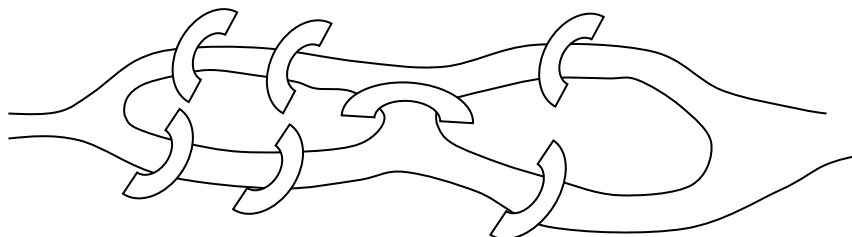


Figure 28.1 The Seven Bridges of Königsberg

In 1736, Leonhard Euler showed that the answer to this question is no. To prove this, he abstracted the map to a graph whose vertices correspond to the land masses and whose edges correspond to the bridges between them. So, in Euler's representation, the town became the graph shown in Figure 28.2. Vertices 1 and 2 represent the river banks and vertices 3 and 4 represent the two islands.

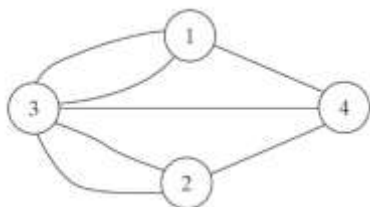


Figure 28.2 The Seven Bridges of Königsberg as a graph

We can now restate the Seven Bridges of Königsberg problem as, “Does there exist a path through the graph such that each edge is traversed exactly once?”

Generalizing to an arbitrary graph, we give the following definitions:

- An **Eulerian path** through a graph G is a path that traverses each edge in G exactly once.
- An **Eulerian circuit** through a graph G is a path that starts at some vertex s , ends back in s , and traverses each edge in G exactly once. (Note the difference between an Eulerian circuit and a Hamiltonian one: An Eulerian circuit visits each *edge* exactly once. A Hamiltonian circuit visits each *vertex* exactly once.)

Bridge inspectors, road cleaners, and network analysts can minimize their effort if they traverse their systems by following an Eulerian circuit. © 701.

We'd now like to determine the computational complexity of deciding, given an arbitrary graph G , whether or not it possesses an Eulerian path (or circuit). Both questions can be answered with a similar technique, so we'll pick the circuit problem and define the following language:

- EULERIAN-CIRCUIT = $\{ \langle G \rangle : G \text{ is an undirected graph and } G \text{ contains an Eulerian circuit} \}$.

We'll show next that EULERIAN-CIRCUIT is in P. The algorithm that we will use to prove this claim is based on an observation that Euler made in studying the Königsberg bridge problem. Define the **degree** of a vertex to be the number of edges with it as an endpoint. For example, in the Königsberg graph, vertices 1, 2, and 4 have degree 3. Vertex 3 has degree 5. Euler observed that:

- A connected graph possesses an Eulerian path that is not a circuit iff it contains exactly two vertices of odd degree. Those two vertices will serve as the first and last vertices of the path.
- A connected graph possess an Eulerian circuit iff all its vertices have even degree. Because each vertex has even degree, any path that enters it can also leave it without reusing an edge.

It should now be obvious why Euler knew (without explicitly exploring all possible paths) that there existed no path that crossed each of the Königsberg bridges exactly once.

Theorem 28.5 The Problem of Finding an Eulerian Circuit in a Graph is in P

Theorem: EULERIAN-CIRCUIT is in P.

Proof: We prove that EULERIAN-CIRCUIT is in P by exhibiting a deterministic, polynomial-time algorithm that decides it:

Eulerian($\langle G \rangle$: graph with vertices V and edges E) =

1. If $connected(G)$ rejects, reject (since an unconnected graph cannot have an Eulerian circuit). Else:
2. For each vertex v in G do:
 - 2.1. Count the number of edges that have v as one endpoint but not both.
 - 2.2. If the count is odd, exit the loop and reject.
3. If all counts are even, accept.

The correctness of *Eulerian* follows from Euler's observations as stated above.

We show that *Eulerian* runs in polynomial time as follows:

- We showed in the proof of Theorem 28.4 that *connected* runs in time that is polynomial in $|V|$.
- The loop in step 2 is executed at most $|V|$ times. Each time through, it requires time that is $\mathcal{O}(|E|)$.
- Step 3 takes constant time.

So the total time required to execute steps 2 - 3 of *Eulerian* is $|V| \cdot \mathcal{O}(|E|)$. But $|E| \leq |V|^2$. So the time required to execute steps 2-3 of *Eulerian* is $\mathcal{O}(|V|^3)$. ■

28.1.6 Minimum Spanning Trees ✦

Consider an arbitrary undirected graph G . A *spanning tree* T of G is a subset of the edges of G such that:

- T contains no cycles and
- Every vertex in G is connected to every other vertex using just the edges in T .

An unconnected graph (i.e., a graph in which there exist at least two vertices with no path between them) has no spanning trees. A connected graph G will have at least one spanning tree; it may have many.

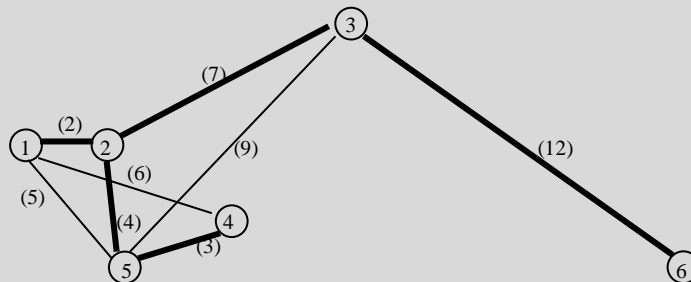
Define a *weighted graph* to be a graph that has a weight (a number) associated with each edge. Typically the weight represents some sort of cost or benefit associated with traversing the edge. Define an *unweighted graph* to be a graph that does not associate weights with its edges.

If G is a weighted graph, we can compare the spanning trees of G by defining the cost of a tree to be the sum of the costs (weights) of its edges. Then a tree T is a *minimum spanning tree* of G iff it is a spanning tree and there is no other spanning tree whose cost is lower than that of T . Note that, if all edge costs are positive, a minimum spanning tree is also a minimum cost subgraph that connects all the vertices of G since any connected subgraph that contains cycles (i.e., any connected subgraph that is not a tree) must have higher cost than T does.

The cheapest way to lay cable that connects a set of points is along a minimum spanning tree that connects those points. € 701.

Example 28.1 A Minimum Spanning Tree

Let G be the following graph, in which the edge costs are shown in parentheses next to each edge:



The subgraph shown with heavy lines is a minimum spanning tree of G .

Given a connected graph G , how shall we go about trying to find a minimum spanning tree for it? The most obvious thing to do is to try all subgraphs of G . We can reject any that do not connect all of G 's vertices. Of the remaining ones, we can choose the one with the lowest total cost. This procedure works but does not run in time that is polynomial in the size of G . Can we do better? The answer is yes.

One of the simplest reasonable algorithms for finding a minimum spanning tree is Kruskal's algorithm, defined as follows:

Kruskal(G : connected graph with vertices V and edges E) =

1. Sort the edges in E in ascending order by their cost. Break ties arbitrarily.
2. Initialize T to a forest with an empty set of edges.
3. Until all the edges in E have been considered do:
 - 3.1. Select e , the next edge in E . If the endpoints of e are not connected in T then add e to T .
4. Return T .

To show that Kruskal's algorithm finds a minimum spanning tree, we must show that the graph T that it returns is a tree (i.e., it is connected and it contains no cycles), that it is a spanning tree (i.e., that it includes all the vertices of the original graph G), and that there is no lower cost spanning tree of G . T cannot contain cycles because step 3.1 can add an edge only if its two endpoints are not already connected. T must be connected and it must be a spanning tree because we assumed that the input graph G is connected. This means that if we used all of G 's edges, every one of G 's vertices would be in T and T would be connected. But we do use all of G 's edges except ones whose endpoints are already connected in T .

So all that remains is to prove that we have found a minimum spanning tree. Kruskal's algorithm is an example of a **greedy algorithm**: It attempts to find an optimal global solution by grabbing the best (local) pieces, in this case short edges, and putting them together. Greedy algorithms tend to run quickly because they may do little or no search. But they cannot always be guaranteed to find the best global solution. For example, there exist greedy algorithms for solving the traveling salesman problem. Although they may produce fairly reasonable solutions quickly, they cannot be guaranteed to find a shortest path.

It turns out, however, that Kruskal's algorithm is guaranteed to find a minimum spanning tree. To see why, we make the following observation, which holds for any graph G with a single minimum spanning tree. It can be extended, with a bit more complexity, to graphs that have multiple spanning trees. Suppose that *Kruskal* generated a tree T_K that is not a minimal spanning tree. Then there was the first point at which it inserted an edge (n, m) that prevented T_K from being the same as some minimal spanning tree. Pick a minimum spanning tree that is identical to T_K up to, but not including that point, and call it T_{min} . Because T_{min} is a spanning tree, it must contain exactly one path between n and m . That path does not contain the edge (n, m) . Suppose that we add it. T_{min} now contains a cycle. That cycle must contain some edge e that *Kruskal* would have considered after considering (n, m) (since otherwise it would have been chosen instead of (n, m) as a way to connect n and m). Thus the weight of that edge must be at least the weight of (n, m) . Remove e from T_{min} . Call the result T_{min}' . T_{min}' is a spanning tree. It contains the edge (n, m) instead of the edge e . Since the weight of (n, m) is less than or equal to the weight of e , the weight of T_{min}' must be less than or equal to the weight of T_{min} . But we assumed that T_{min} was minimal, so it can't be less. It must, therefore be equal. But then adding (n, m) did not prevent T_K , the tree that *Kruskal* built, from being minimal. This contradicts the assumption that it did.

We are now ready to ask the question, "How computationally hard is finding a minimum spanning tree?" Since this is an optimization problem, we'll use the same technique we used to convert the traveling salesman problem into the decision problem TSP-DECIDE: we'll give a cost bound and ask whether there exists a minimum spanning tree whose cost is less than the bound we provide.

Define the language:

- $MST = \{ \langle G, cost \rangle : G \text{ is an undirected graph with a positive cost attached to each of its edges and there exists a minimum spanning tree of } G \text{ with total cost less than } cost \}$.

Theorem 28.6 The Problem of Finding a Minimum Spanning Tree with an Acceptable Cost is in P

Theorem: MST is in P.

Proof: We prove that MST is in P by exhibiting a polynomial time algorithm that decides it:

$MSTdecide(\langle G: \text{graph with vertices } V \text{ and edges } E, \text{cost: number} \rangle) =$

1. Invoke $Kruskal(G)$. Let T be the minimum spanning tree that is returned.
2. If the total cost of $T < \text{cost}$ then accept, else reject.

$MSTdecide$ runs in polynomial time if each of its two steps does. Step 2 can be done in constant time.

So it remains to analyze Kruskal's algorithm, which we do as follows:

- Step 1, the sorting step, can be done with $|E| \cdot \log |E|$ comparisons and each comparison takes constant time.
- Step 2 takes constant time.
- The loop in step 3 is executed $|E|$ times. The time required at each step to test whether two vertices are disconnected depends on the data structure that is used to represent T . A straightforward way to do it is to maintain, for each tree in the forest T , a set that contains exactly the vertices that are present in that tree. Each vertex in V will be in at most one such set. So, in considering an edge (n, m) , we examine each of the sets. If we find one that contains n , we look just in that set to see whether m is also there. If it is, then n and m are already connected; otherwise they are not. To find n , we may have to look through all the sets, so we may have to examine $|V|$ vertices and, if all the vertices are in the same set, we might have to do that again to look for m . So we might examine $\mathcal{O}(|V|)$ vertices to do the check for disconnectedness. Further, if we take this approach, then we must maintain these sets. But, even doing that, the cost of adding e to T is constant. So step 3 takes a total number of steps that is $\mathcal{O}(|E| \cdot |V|)$.

So the total time required to execute Kruskal's algorithm is $\mathcal{O}(|E| \cdot |V|)$ and so $\mathcal{O}(|\langle G \rangle|^2)$. With a more efficient implementation of step 3, it is possible to show¹³ that it is also $\mathcal{O}(|E| \cdot \log |V|)$.

Kruskal's algorithm proves that MST is in P. And it is very easy to implement. There also exist other algorithms for finding minimum spanning trees that run even faster than Kruskal's algorithm does \blacksquare .

28.1.7 Primality Testing

Prime numbers \blacksquare have fascinated mathematicians since the time of the ancient Greeks. It turns out that some key problems involving prime numbers are known to be solvable in polynomial time, while some others are not now known to be.

Prime numbers are of more than theoretical interest. They play a critical role in modern encryption systems \mathbb{C} 722.

In Example 27.6, we introduced the language:

- RELATIVELY-PRIME = $\{\langle n, m \rangle : n \text{ and } m \text{ are integers and they are relatively prime}\}$. Recall that two integers are relatively prime iff their greatest common divisor is 1.

Theorem 28.7 RELATIVELY-PRIME is in P

Theorem: RELATIVELY-PRIME is in P.

Proof: RELATIVELY-PRIME can be decided in linear time by the algorithm $REL-PRIMEdecide$, described in Example 27.6. \blacksquare

¹³ For a proof of this claim, see [Cormen et al. 2001].

But now consider the problem of determining whether or not a number is prime. We have encoded that problem as the language:

- $\text{PRIMES} = \{w : w \text{ is the binary encoding of a prime number}\}$.

The obvious way to decide PRIMES is, when given the number k , to consider all the natural numbers between 2 and \sqrt{k} . Check each to see whether it divides evenly into k . If any such number does, then k isn't prime. If none does, then k is prime. The time required to implement this approach is $\mathcal{O}(\sqrt{k})$. But n , the length of the string that encodes k , is $\log k$. So this simple algorithm is $\mathcal{O}(2^{n/2})$. Because of the practical significance of primality testing, particularly in cryptography, substantial effort has been devoted to finding a more efficient technique for deciding PRIMES. It turns out that there exist randomized algorithms that can decide PRIMES in polynomial time if we allow an exceedingly small, but nonzero, probability of making an error. We'll describe such an approach in Chapter 30. Such techniques are widely used in practice.

Until very recently, however, the question of whether PRIMES is in P (i.e., whether a provably correct, polynomial-time algorithm for it exists) remained unanswered and it continued to be of theoretical interest. We now know the answer to the question. We can state it as the following theorem:

Theorem 28.8 PRIMES is in P

Theorem: PRIMES is in P.

Proof: Various proofs of this claim have been proposed. Most have relied on hypotheses that, although widely believed to be true, remained unproven. But [Agrawal, Kayal and Saxena 2004] contains a proof that relies on no unproven assumptions. It describes an algorithm for deciding PRIMES that runs in deterministic $\mathcal{O}((\log n)^{12} \cdot f(\log \log n))$ time, where f is a polynomial. The details of the proof are beyond the scope of this book. Since the original algorithm was described, modifications of it that further improve its performance have been discovered. ■

The class P is closed under complement. So we also have that the following language is in P:

- $\text{COMPOSITES} = \{w : w \text{ is the binary encoding of a composite number}\}$. A composite number is a natural number greater than 1 that is not prime.

Unfortunately, the results we have just presented do not close the book on the problem of working with prime and composite numbers. We now know that there exists a polynomial-time algorithm to check whether a number is prime and we continue to exploit randomized algorithms to answer the question in practice. The fact that we can, in polynomial time, tell whether or not a number is prime does not tell us that there exists a polynomial-time algorithm to *discover* the factors of a number that is not prime. No efficient algorithm for factoring using a conventional computer is currently known. Were a practical and efficient algorithm to be discovered, modern encryption techniques that rely on factorization would no longer be effective. One approach to constructing such an algorithm is to exploit quantum computing. Shor's algorithm [Sho95], for example, factors a number k in $\mathcal{O}((\log k)^3)$ time on a quantum computer. But the largest number that has so far been able to be factored on a quantum computer is 15 [Sho95].

28.2 The Language Class NP

Now suppose that, in our quest for polynomial-time deciding Turing machines, we allow nondeterminism. Will this increase the number of languages for which it is possible to build polynomial-time deciders? No one knows. But it appears likely that it does. For example, consider again the traveling salesman language $\text{TSP-DECIDE} = \{w \text{ of the form: } \langle G, \text{cost} \rangle, \text{ where } \langle G \rangle \text{ encodes an undirected graph with a positive distance attached to each of its edges and } G \text{ contains a Hamiltonian circuit whose total cost is less than } \text{cost}\}$. Recall that a Hamiltonian circuit is a path that starts at some vertex s , ends back in s , and visits each other vertex in G exactly once. We know of no *deterministic* Turing machine that can decide TSP-DECIDE in polynomial time. But there is a *nondeterministic* Turing machine that does. It works by using nondeterminism to guess the best path. (We'll describe it in detail below.)

28.2.1 Defining the Class NP

TSP-DECIDE is typical of a large class of problems that are of considerable practical interest. All of them share the following three properties:

1. The problem can be solved by searching through a space of partial solutions (such as routes), looking for a complete solution that satisfies all of the given constraints. The size of the space that must be explored in this way grows exponentially with the size of the problem that is being considered.
2. No better (i.e., not based on search) technique for finding an exact solution is known.
3. But, if a proposed solution were suddenly to appear, it could be checked for correctness very efficiently.

The next language class that we will define is called NP. It will include TSP-DECIDE and its cousins, as well as the “easier” languages that are also in P. In Section 28.5.1, we’ll define a subset of NP called NP-complete. It will include only those languages, like TSP-DECIDE, that are the “hardest” of the NP languages.

Properties 1 and 3 suggest two superficially quite different ways to define NP. It turns out that the two definitions are equivalent. Because each of them is useful in some contexts, we provide them both.

Nondeterministic Deciding

The first definition we present is based on the idea of search. Nondeterministic Turing machines perform search. So we will define the class NP to include all and only those languages that are decidable by a nondeterministic Turing machine in polynomial time. (The name NP stands for Nondeterministic Polynomial.) Remember that in defining the time requirement of a nondeterministic Turing machine M , we don’t consider the total number of steps that M executes on all of its computational paths; instead we measure just the length of its longest path. Thus we’ll say that a language L is in NP iff there is some nondeterministic Turing machine M that decides L and the length of the longest computational path that M must follow on any input of length n grows as some polynomial function of n . So we have:

The Class NP: $L \in \text{NP}$ iff there exists some nondeterministic Turing machine M that decides L and $\text{timereq}(M) \in \mathcal{O}(n^k)$ for some constant k .

Again consider the language TSP-DECIDE. Given a string $w = \langle G, \text{cost} \rangle$, we can build a nondeterministic Turing machine M that decides whether w is in TSP-DECIDE. M ’s job is to decide whether there is a Hamiltonian circuit through G whose cost is less than cost . M will nondeterministically guess a path through G with length equal to the number of vertices in G . There is a finite number of such paths and each of them has finite length, so all paths of M will eventually halt. M will accept w if it finds at least one path that corresponds to a Hamiltonian circuit whose cost is less than cost . Otherwise it will reject. We will show below that $\text{timereq}(M) \in \mathcal{O}(n)$. So TSP-DECIDE \in NP.

Deterministic Verifying

But now suppose that (somehow) we find ourselves in possession of a particular path c , along with the claim that c proves that $w = \langle G, \text{cost} \rangle$ is in TSP-DECIDE. In other words, there is a claim that c is a Hamiltonian circuit through G with cost less than cost . Our only job now is to check c and verify that it does in fact prove that w is in TSP-DECIDE. We can build a deterministic Turing machine M' that does this in time that is polynomial in the length of w . The input to M' will be the string $\langle G, \text{cost}, c \rangle$. Assume that c is represented as a sequence of vertices. Then M' will simply walk through c , one vertex at a time, checking that G does in fact contain the required edge from each vertex to its successor. As it does this, it will keep track of the length of c so far. If that length ever exceeds the number of vertices in G , M' will halt and reject. Also, as it goes along, it will use a list of the vertices in G and mark each one as it is visited, checking to make sure that every vertex is visited exactly once. And, finally, it will keep a running total of the costs of the edges that it follows. If every step of c follows an edge in G , every vertex in G is visited once except that c starts and ends at the same vertex, and the cost of c is less than cost , M' will accept its input and thus report that c does in fact prove that w is in TSP-DECIDE. Otherwise M' will reject its input and thus report that c fails to do that. (But note that this doesn’t mean that w is not in TSP-DECIDE; we know only that c fails to show that it is.) We’ll call M' a verifier for TSP-DECIDE. We can analyze the complexity of M as follows: Let n be the number of vertices in G . Then M' executes its outer loop at most n times (since it will quit if the length of c exceeds n). As it checks each step in c , it may take $\mathcal{O}(|G|^2)$ steps to check the edges of G , to mark the visited

vertices, and to update the cost total. So the total time for the main loop is $\mathcal{O}(|G|^3)$. The final check takes time that is $\mathcal{O}(|G|^2)$. So the total is $\mathcal{O}(|G|^3)$.

So we have M , a *nondeterministic* polynomial time *decider* for TSP-DECIDE. And we have M' , a *deterministic* polynomial time *verifier* for it. The relationship between M and M' is typical for problems like TSP-DECIDE. The decider works by nondeterministically searching a space of candidate structures (in the case of TSP-DECIDE, candidate paths) and accepting iff it finds at least one that meets the requirements imposed by the language that is being decided. The verifier works by simply checking a single candidate structure (e.g., a path) and verifying that it meets the language's requirements.

The existence of verifiers like M' suggests an alternative way to define the class NP. We first define exactly what we mean by a verifier: A Turing machine V is a *verifier* for a language L iff:

$$w \in L \text{ iff } \exists c (\langle w, c \rangle \in L(V)).$$

We'll call c , the candidate structure that we provide to the verifier V , a *certificate*. Think of it as a certificate of proof that w is in L . So V verifies L precisely in case it accepts at least one certificate for every string in L and accepts no certificate for any string that is not in L . Since the string we are actually interested in is w , we will define $\text{timereq}(V)$, when V is a verifier, as a function just of $|w|$, not of $|\langle w, c \rangle|$.

Now, using the idea of a verifier, we can state the following alternative definition for the class NP:

The Class NP: $L \in \text{NP}$ iff there exists a deterministic Turing machine V such that V is a verifier for L and $\text{timereq}(V) \in \mathcal{O}(n^k)$ for some constant k (i.e., V is a deterministic polynomial-time verifier for L).

Note that, since the number of steps that a polynomial-time V executes is bounded by some polynomial function of the length of the input string w , the number of certificate characters it can look at is also bounded by the same function. So, when we are considering polynomial time verifiers, we will consider only certificates whose length is bounded by some polynomial function of the length of the input string w .

The Two Definitions are Equivalent

Now that we have two definitions for the class NP, we would like to be able use whichever one is more convenient. However, it is not obvious that the two definitions are equivalent. So we must prove that they are.

Theorem 28.9 The Two Definitions of the Class NP are Equivalent

Theorem: The following two definitions are equivalent:

- (1) $L \in \text{NP}$ iff there exists a nondeterministic, polynomial-time Turing machine that decides it.
- (2) $L \in \text{NP}$ iff there exists a deterministic, polynomial-time verifier for it.

Proof: We must prove that if there exists a nondeterministic decider for L than there exists a deterministic verifier for it, and vice versa:

- (1) Let L be a language that is in NP by definition (1). Then there exists a nondeterministic, polynomial-time Turing machine M that decides it. Using M , we construct V , a deterministic polynomial time verifier for L . On the input $\langle w, c \rangle$, V will simulate M running on w except that, every time M would have to make a choice, V will simply follow the path that corresponds to the next symbol of c . V will accept iff M would have accepted on that path. Thus V will accept iff c is a certificate for w . V runs in polynomial time because the length of the longest path M can follow is bounded by some polynomial function of the length of w . So V is a deterministic polynomial-time verifier for L .

- (2) Let L be a language that is in NP by definition (2). Then there exists a deterministic Turing machine V such that V is a verifier for L and $\text{timereq}(V) \in \mathcal{O}(n^k)$ for some k . Using V , we construct a nondeterministic polynomial-time Turing machine M that will decide L . On input w , M will nondeterministically select a certificate c whose length is bounded by the greatest number of steps V could execute on any input of length at most $\text{timereq}(V)(|w|)$. (It need not consider any longer certificates since V would not be able to evaluate them.) It will then run V on $\langle w, c \rangle$. M follows a finite number of computational paths, each of which halts in time that is $\mathcal{O}(n^k)$. So M is a nondeterministic polynomial-time Turing machine that decides L . ■

In the next section we will see several examples of languages that are in NP. Theorem 28.9 tells us that we can prove a claim of the form, “ L is in NP” by exhibiting for L either a nondeterministic polynomial-time decider or a deterministic polynomial-time verifier.

28.2.2 Languages That Are in NP

The class NP is important because it contains many languages that arise naturally in a variety of applications. We’ll mention several here. None of these languages is known to be in P. In fact, all of them are in the complexity class NP-complete, which contains the hardest NP languages. We’ll define NP-completeness in Section 28.5.

TSP-DECIDE is typical of a large class of graph-based languages that are in NP. This class includes:

- HAMILTONIAN-PATH = $\{\langle G \rangle : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian path}\}$. A Hamiltonian path through G is a path that visits each vertex in G exactly once.
- HAMILTONIAN-CIRCUIT = $\{\langle G \rangle : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian circuit}\}$. A **Hamiltonian circuit** is a path that starts at some vertex s , ends back in s , and visits each other vertex in G exactly once.
- CLIQUE = $\{\langle G, k \rangle : G \text{ is an undirected graph with vertices } V \text{ and edges } E, k \text{ is an integer, } 1 \leq k \leq |V|, \text{ and } G \text{ contains a } k\text{-clique}\}$. A **clique** in G is a subset of V with the property that every pair of vertices in the clique is connected by some edge in E . A k -clique is a clique that contains exactly k vertices.

NP includes other kinds of languages as well. Typically a language is in NP if you can imagine deciding it by exploring a well-defined search space looking for at least one value that meets some clear requirement. So, for example, the following language based on an important property of Boolean logic formulas is in NP:

- SAT = $\{\langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable}\}$. We can show that a string w is in SAT by finding a satisfying assignment of values to the variables in the wff that it encodes.

Sets of almost any type can lead to problems that are in NP. We’ve just mentioned examples based on graphs (sets of vertices and edges) and on logical wffs (sets of variables connected by operators). The following NP language is based on sets of integers:

- SUBSET-SUM = $\{\langle S, k \rangle : S \text{ is a multiset (i.e., duplicates are allowed) of integers, } k \text{ is an integer, and there exists some subset of } S \text{ whose elements sum to } k\}$. For example:
 - $\langle \{1256, 45, 1256, 59, 34687, 8946, 17664\}, 35988 \rangle$ is in SUBSET-SUM.
 - $\langle \{101, 789, 5783, 6666, 45789, 996\}, 29876 \rangle$ is not in SUBSET-SUM.

The SUBSET-SUM problem can be used as the basis for a simple encryption system, that could be used, for example, to store password files. We start with a set of say 1000 integers. Call them the base integers. Then suppose that each password can be converted (for example by looking at pairs of symbols) to a multiset of base integers. Then a password checker need not store actual passwords. It can simply store the sum of the base integers that the password generates. When a user enters a password, it is converted

to base integers and the sum is computed and checked against the stored sum. But if someone breaks in and gets access to the stored password sums, they won't be able to reconstruct any of the passwords, even if they know how passwords are mapped to base integers, unless they can (reasonably efficiently) take a sum and find a subset of the base integers that add to form it.

The next example of an NP language is based on sets of anything as long as the objects have associated costs:

- SET-PARTITION = $\{ \langle S \rangle : S \text{ is a multiset (i.e., duplicates are allowed) of objects, each of which has an associated cost, and there exists a way to divide } S \text{ into two subsets, } A \text{ and } S - A, \text{ such that the sum of the costs of the elements in } A \text{ equals the sum of the costs of the elements in } S - A \}$.

SET-PARTITION arises in many sorts of resource allocation contexts. For example, suppose that there are two production lines and a set of objects that need to be manufactured as quickly as possible. Let the objects' costs be the time required to make them. Then the optimum schedule divides the work evenly across the two machines. Load balancing in a dual processor computer system can also be described as a set-partition problem.

Our final example is based on sets of anything as long as the objects have associated costs and values:

- KNAPSACK = $\{ \langle S, v, c \rangle : S \text{ is a set of objects each of which has an associated cost and an associated value, } v \text{ and } c \text{ are integers, and there exists some way of choosing elements of } S \text{ (duplicates allowed) such that the total cost of the chosen objects is at most } c \text{ and their total value is at least } v \}$. Notice that, if the cost of each item equals its value, then the KNAPSACK problem becomes very similar to the SUBSET-SUM problem.

The KNAPSACK problem derives its name from the problem of choosing the best way to pack a knapsack with limited capacity in such a way as to maximize the utility of the contents. For example, imagine a thief trying to decide what to steal or a backpacker trying to decide what food to take. The KNAPSACK problem arises in a wide variety of applications in which resources are limited and utility must be maximized. For example, what ads should a company buy? What products should a factory make? How should a company expand its workforce?

In the next three sections we'll prove that TSP-DECIDE, CLIQUE and SAT are in NP. We'll consider HAMILTONIAN-CIRCUIT in Theorem 28.22. We leave the rest as exercises.

28.2.3 TSP

We argued above that there exists a nondeterministic, polynomial-time Turing machine that decides TSP-DECIDE. Now we prove that claim. Just to make it clear how such a machine might work, we will describe in detail the Turing machine *TSPdecide*. Let V be the vertices in G and E be its edges. *TSPdecide* will nondeterministically consider all paths through G with length equal to $|V|$. There is a finite number of such paths and each of them has finite length, so all paths of *TSPdecide* will eventually halt. *TSPdecide* will accept w if it finds at least one path that corresponds to a Hamiltonian circuit and that has cost less than $cost$. Otherwise it will reject. *TSPdecide* will use three tapes. The first will store the input G . The second will keep track of the path that is being built. And the third will contain the total cost of the path so far. We define *TSPdecide* as follows:

TSPdecide($\langle G: \text{graph with vertices } V \text{ and edges } E, cost: \text{integer} \rangle$) =

1. Initialize by nondeterministically choosing a vertex in G . Put that vertex on the path that is stored on tape 2. Write 0 on tape 3.
2. Until the number of vertices on the path on tape 2 is equal to $|V| + 1$ or this path fails do:
 - 2.1. Nondeterministically choose an edge e in E .
 - 2.2. Check that one endpoint of e is the last vertex on the current path.
 - 2.3. Check that either:

- the number of vertices on the path equals $|V|$ and the other endpoint of e is the same as the first vertex in the path, or
 - the number of vertices on the path is less than $|V|$ and the other endpoint of e is not on already on the path.
- 2.4. Add the cost of e to the path cost that is stored on tape 3 and check that the result is less than $cost$.
 - 2.5. If conditions 2.2, 2.3, and 2.4, are satisfied then add the second endpoint of e to the current path.
 - 2.6. Else this path fails. Exit the loop.
3. If the loop ended normally, accept. If it ended by the path failing, reject.

We analyze $timereq(TSPdecide)$ as follows: the initialization in step 1 takes $\mathcal{O}(|\langle G, cost \rangle|)$ time. The longest path that $TSPdecide$ will consider contains $|V| + 1$ vertices. (It may also consider some shorter paths if they fail before completing a circuit). So $TSPdecide$ goes through the step 2 loop at most $\mathcal{O}(|\langle G, cost \rangle|)$ times. Each step of that loop takes $\mathcal{O}(|\langle G, cost \rangle|)$ time. So $timereq(TSPdecide) \in \mathcal{O}(|\langle G, cost \rangle|^2)$.

We've now described both a nondeterministic decider and a deterministic verifier for TSP-DECIDE. So proving the next theorem is straightforward.

Theorem 28.10 TSP-DECIDE is in NP

Theorem: $TSP-DECIDE = \{\langle G, cost \rangle : \langle G \rangle \text{ encodes an undirected graph with a positive distance attached to each of its edges and } G \text{ contains a Hamiltonian circuit whose total cost is less than } cost\}$ is in NP.

Proof: The nondeterministic Turing machine $TSPdecide$ decides TSP-DECIDE in polynomial time. ■

While it is sometimes instructive to describe a decider or a verifier in detail, as a Turing machine, as we have just done, we will generally describe them simply as well-specified algorithms. We will do that in the following examples.

28.2.4 Clique Detection

Recall that, given a graph G with vertices V and edges E , a **clique** in G is a subset of V with the property that every pair of vertices in the clique is connected by some edge in E . A **k -clique** is a clique that contains exactly k vertices.

Clique detection, particularly the detection of maximally large cliques, plays an important role in many applications in computational biology.

Theorem 28.11 CLIQUE is in NP

Theorem: $CLIQUE = \{\langle G, k \rangle : G \text{ is an undirected graph with vertices } V \text{ and edges } E, k \text{ is an integer, } 1 \leq k \leq |V|, \text{ and } G \text{ contains a } k\text{-clique}\}$ is in NP.

Proof: We can prove this claim by describing a deterministic polynomial time verifier, $clique-verify(\langle G, k, c \rangle)$, that takes three inputs, a graph G , an integer k , and a set of vertices c , where c is a proposed certificate for $\langle G, k \rangle$. The job of $clique-verify$ is to check that c is a clique in G and that it contains k vertices. The first step of $clique-verify$ is to count the number of vertices in c . If the number is greater than $|V|$ or not equal to k , it will immediately reject. Otherwise, it will go on to step 2, where it will consider all pairs of vertices in c . For each, it will go through the edges in E and check that there is an edge between the two vertices of the pair. If there is any pair that is not connected by an edge, $clique-verify$ will reject. If all pairs are connected, it will accept. Step 1 takes time that is linear in $|c|$, which is bounded by some polynomial function of $|\langle G, k \rangle|$. Step 2 must consider $|c|^2$ vertex pairs. For each it must examine at most $|E|$ edges. Since both $|c|$ and $|E|$ are bounded by $|\langle G, k \rangle|$, $timereq(clique-verify) \in \mathcal{O}(|\langle G, k \rangle|^3)$. So $clique-verify$ is a deterministic polynomial-time verifier for CLIQUE. ■

28.2.5 Boolean Satisfiability

In Section 22.4.1, we showed that several key questions concerning Boolean formulas are decidable. In particular, we showed that SAT is in D. We can now consider the complexity of SAT. We'll prove here that it and one of its cousins, 3-SAT, are NP. You may recall that, in Section 22.4.1, we also showed that the problem of deciding whether a Boolean formula is valid (i.e., whether it is true for all assignments of values to its variables) is decidable. It turns out that that problem appears to be harder than the problem of deciding satisfiability. We'll consider the language $\text{VALID} = \{\langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is valid}\}$ in Section 28.8.

SAT has applications in such domains as computer-aided design, computer-aided manufacturing, robotics, machine vision, scheduling, and hardware and software verification. It is particularly useful in verifying the correctness of digital circuits using a technique called model checking. $\text{C } 679$.

Theorem 28.12 SAT is in NP

Theorem: $\text{SAT} = \{\langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable}\}$ is in NP.

Proof: SAT is in NP because there exists a deterministic polynomial time verifier for it. $\text{SAT-verify}(\langle w, c \rangle)$ takes two inputs, a wff w and a certificate c , which is a list of assignments of truth values to the variables of w . The job of SAT-verify is to determine whether w evaluates to *True* given the assignments provided by c . For example:

- The wff $w = P \wedge Q \wedge \neg R$ is satisfiable. The string $c = (P = \text{True}, Q = \text{True}, R = \text{False})$ is a certificate for it, since the expression $\text{True} \wedge \text{True} \wedge \neg \text{False}$ simplifies to *True*. $\text{SAT-verify}(\langle w, c \rangle)$ will accept.
- The wff $w = P \wedge Q \wedge R$ is satisfiable. But the string $c = (P = \text{True}, Q = \text{True}, R = \text{False})$ is not a certificate for it, since the expression $\text{True} \wedge \text{True} \wedge \text{False}$ simplifies to *False*. So $\text{SAT-verify}(\langle w, c \rangle)$ will reject.
- The wff $w = P \wedge \neg P$ is not satisfiable. So for any c , $\text{SAT-verify}(\langle w, c \rangle)$ will reject.

Let vars be the number of distinct variables in w . Let ops be the number of operators in w . Then SAT-verify behaves as follows: For each assignment in c , it makes one pass through w , replacing all occurrences of the current variable with the value that c assigns to it. Then it makes at most ops passes through w , on each pass replacing every operator whose arguments have already been evaluated by the result of applying the operator to its arguments. Then it checks to see whether $w = \text{True}$. The first step must consider vars variables and each can be processed in $\mathcal{O}(|w|)$ time. Since $\text{vars} \leq |w|$, this first step takes $\mathcal{O}(|w|^2)$ time. The second step executes at most ops passes and each pass can be done in $\mathcal{O}(|w|)$ time. Since $\text{ops} \leq |w|$, the second step takes $\mathcal{O}(|w|^2)$ time. Thus SAT-verify takes time $\mathcal{O}(|w|^2)$ and is a deterministic polynomial-time verifier for SAT.

Alternatively, we can build a nondeterministic polynomial-time decider for SAT. It decides whether a string $\langle w \rangle$ is in SAT by nondeterministically choosing a set of assignments to the variables in w . Then it uses SAT-verify to check whether that assignment proves that w is satisfiable. ■

As far as we know, SAT is not also in P. No polynomial time algorithm to decide it in the general case is known. But very efficient SAT solvers work well in practice. They take advantage of the fact that it is typically not necessary to enumerate all possible assignments of values to the variables. One technique that exploits this observation relies on a clever data structure, the ordered binary decision diagram (or OBDD), which we describe in $\text{B } 612$.

We next describe 3-SAT, a variant of SAT that we will find useful in our upcoming discussion of the complexity of several other languages. Before we can define 3-SAT we must define conjunctive normal form for Boolean formulas:

- A **literal** is either a variable or a variable preceded by a single negation symbol.
- A **clause** is either a single literal or the disjunction of two or more literals.
- A well-formed formula (or wff) of Boolean logic is in **conjunctive normal form** (or CNF) iff it is either a single clause or the conjunction of two or more clauses.

- A wff is in **3-conjunctive normal form** (or 3-CNF) iff it is in conjunctive normal form and each clause contains exactly three literals.

Table 28.1 illustrates these definitions. The symbol • indicates that the corresponding formula is in the matching form.

	3-CNF	CNF
$(P \vee \neg Q \vee R)$	•	•
$(P \vee \neg Q \vee R) \wedge (\neg P \vee Q \vee \neg R)$	•	•
P		•
$(P \vee \neg Q \vee R \vee S) \wedge (\neg P \vee \neg R)$		•
$P \rightarrow Q$		
$(P \wedge \neg Q \wedge R \wedge S) \vee (\neg P \wedge \neg R)$		
$\neg(P \vee Q \vee R)$		

Table 28.1 3-CNF and CNF formulas

Every wff can be converted to an equivalent wff in 3-CNF. See § 606 for a proof of this claim, as well as more examples of all of the terms that we have just defined.

Theorem 28.13 3-SAT is in NP

Theorem: 3-SAT = { $\langle w \rangle$: w is a wff in Boolean logic, w is in 3-conjunctive normal form and w is satisfiable} is in NP.

Proof: 3-SAT is in NP because there exists a deterministic polynomial time verifier for it. *3-SAT-verify*($\langle w, c \rangle$) first checks to make sure that w is in 3-CNF. It can do that in linear time. Then it calls *SAT-verify*($\langle w, c \rangle$) to check that c is a certificate for w . ■

28.3 Does $P = NP$?

While we know some things about the relationship between P and NP, a complete answer to the question, “Are they equal?” has, so far, remained elusive.

We begin by describing what we do know:

Theorem 28.14 Every Language in P is also in NP

Theorem: $P \subseteq NP$.

Proof: Let L be an arbitrary language in P. Then there exists a deterministic polynomial time decider M for L . But M is also a nondeterministic polynomial time decider for L . (It just doesn’t have to make any guesses.) So L is in NP. ■

So all of the following languages are in NP:

- Every context-free language,
- EULERIAN-CIRCUIT,
- MST, and
- PRIMES.

But what about the other direction? Are there languages that are in NP but that are not in P? Alternatively (since we just showed that $P \subseteq NP$), does $P = NP$? No one knows. There are languages, like TSP-DECIDE, CLIQUE, and SAT

that are known to be in NP but for which no deterministic polynomial time decision procedure exists. But no one has succeeded in proving that those languages, or many others that are in NP, are not also in P.

The question, “Does $P = NP$?” is one of seven Millennium Problems [\[1\]](#); a \$1,000,000 prize awaits anyone who can solve it. By the way, most informed bets are on the answer to the question being, “No”. Further, it is widely believed that even if it should turn out to be possible to prove that every language that is in NP is also in P, it is exceedingly unlikely that that proof will lead to the development of practical polynomial time algorithms to decide languages like TSP-DECIDE and SAT. There is widespread consensus that if such algorithms existed they would have been discovered by now given the huge amount of effort that has been spent looking for them.

While we do not know with certainty whether $P = NP$, we do know something about how the two classes relate to other complexity classes. In particular, define:

- PSPACE: For any language L , $L \in \text{PSPACE}$ iff there exists some deterministic Turing machine M that decides L and $\text{space}_{req}(M) \in \mathcal{O}(n^k)$ for some k .
- NPSPACE: For any language L , $L \in \text{NPSPACE}$ iff there exists some nondeterministic Turing machine M that decides L and $\text{space}_{req}(M) \in \mathcal{O}(n^k)$ for some k .
- EXPTIME: For any language L , $L \in \text{EXPTIME}$ iff there exists some deterministic Turing machine M that decides L and $\text{time}_{req}(M) \in \mathcal{O}(2^{(n^k)})$ for some k . We’ll consider the class EXPTIME in Section 28.9.

Chapter 29 is devoted to a discussion of space complexity classes, including PSPACE and NPSPACE. We’ll mention here just one important result: Savitch’s Theorem (which we state as Theorem 29.2) tells us that any nondeterministic Turing machine can be converted to a deterministic one that uses at most quadratically more space. So, in particular, $\text{PSPACE} = \text{NPSPACE}$ and we can simplify our discussion by considering just PSPACE.

We can summarize what is known about P, NP, and the new classes PSPACE and EXPTIME as follows:

$$P \subseteq NP \subseteq \text{PSPACE} \subseteq \text{EXPTIME}.$$

In addition, in Section 28.9.1, we will prove the Deterministic Time Hierarchy Theorem, which tells us that $P \neq \text{EXPTIME}$. So at least one of the inclusions shown above must be proper. It is generally assumed that all of them are, but no proofs of those claims exist.

Because we know that $P \neq \text{EXPTIME}$, we also know that there exist decidable but intractable problems.

28.4 Using Reduction in Complexity Proofs

In Chapter 21, we used reduction to prove decidability properties of new languages by reducing other languages to them. Since all we cared about then was decidability, we accepted as reductions any Turing machines that implemented computable functions. We were not concerned with the efficiency of those Turing machines. We can also use reduction to prove complexity properties of new languages based on the known complexity properties of other languages. When we do that, though, we will need to place bounds on the complexity of the reductions that we use. In particular, it is important that the complexity of any reduction we use be dominated by the complexity of the language we are reducing to. To guarantee that, we will exploit only deterministic, polynomial-time reductions.

All of the reductions that we will use to prove complexity results will be mapping reductions. Recall, from Section 21.2.2, that a *mapping reduction* R from L_1 to L_2 is a Turing machine that implements some computable function f with the property that:

$$\forall x (x \in L_1 \leftrightarrow f(x) \in L_2).$$

Now suppose that R is a mapping reduction from L_1 to L_2 and that there exists a Turing machine M that decides L_2 . Then to decide whether some string x is in L_1 we first apply R to x and then invoke M to decide membership in L_2 . So $C(x) = M(R(x))$ will decide L_1 .

Suppose that there exists a deterministic, polynomial-time mapping reduction R from L_1 to L_2 . Then we'll say that L_1 is deterministic, polynomial-time **reducible** to L_2 , which we'll write as $L_1 \leq_P L_2$. And, whenever such an R exists, we note that:

- L_1 must be in P if L_2 is: if L_2 is in P then there exists some deterministic, polynomial-time Turing machine M that decides it. So $M(R(x))$ is also a deterministic, polynomial-time Turing machine and it decides L_1 .
- L_1 must be in NP if L_2 is: if L_2 is in NP then there exists some nondeterministic, polynomial-time Turing machine M that decides it. So $M(R(x))$ is also a nondeterministic, polynomial-time Turing machine and it decides L_1 .

Given two languages L_1 and L_2 , we can use reduction to:

- Prove that L_1 is in P or in NP because we already know that L_2 is.
- Prove that L_1 would be in P or in NP if we could somehow show that L_2 is. When we do this we cluster languages of similar complexity (even if we're not yet sure what that complexity is).

In many of the reductions that we will do, we will map objects of one sort to objects that appear to be of a very different sort. For example, the first reduction that we show will be from 3-SAT (a language of Boolean wffs) to the graph language INDEPENDENT-SET. On the surface, Boolean formulas and graphs seem quite different. So how should the reduction proceed? The strategy we'll typically use is to exploit gadgets. A **gadget** is a structure in the target language that mimics the role of a corresponding structure in the source language. In the 3-SAT to INDEPENDENT-SET reduction, strings in the source language describe formulas that contain literals and clauses. Strings in the target language describe graphs that contain vertices and edges. So we need one gadget that looks like a graph but that mimics a literal and another gadget that looks like a graph but that mimics a clause. Very simple gadgets will work in this case. In some others that we'll see later, more clever constructions will be required.

Consider the language:

- INDEPENDENT-SET = $\{ \langle G, k \rangle : G \text{ is an undirected graph and } G \text{ contains an independent set of at least } k \text{ vertices} \}$.

An **independent set** is a set of vertices no two of which are adjacent (i.e., connected by a single edge). So, in the graph shown in Figure 28.3, the circled vertices form an independent set.

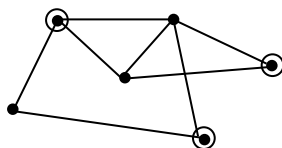


Figure 28.3 An independent set

Consider a graph in which the edges represent conflicts between the objects that correspond to the vertices. For example, in a scheduling program the vertices might represent tasks. Then two vertices will be connected by an edge if their corresponding tasks cannot be scheduled at the same time because their resource requirements conflict. We can find the largest number of tasks that can be scheduled at the same time by finding the largest independent set in the task graph.

Notice, by the way, that there is an obvious relationship between INDEPENDENT-SET and CLIQUE. If S is an independent set in some graph G with vertices V and edges E , then S is also a clique in the graph G' with vertices V and edges E' , where E' contains an edge between each pair of nodes n and m iff V does not contain such an edge. There is also a relationship between INDEPENDENT-SET and the language CHROMATIC-NUMBER, which we'll define in Section 28.7.6. While INDEPENDENT-SET asks for the maximum number of vertices in any one independent set in G , CHROMATIC-NUMBER asks how many nonintersecting independent sets are required if every vertex in G is to be in one.

Theorem 28.15 3-SAT is Reducible to INDEPENDENT-SET

Theorem: $3\text{-SAT} \leq_p \text{INDEPENDENT-SET}$

Proof: We show a deterministic, polynomial-time reduction R from 3-SAT to INDEPENDENT-SET. R must map from a Boolean formula to a graph. Let f be a Boolean formula in 3-conjunctive normal form. Let k be the number of clauses in f . R is defined as follows:

$R(\langle f \rangle) =$

1. Build a graph G by doing the following:
 - 1.1. Create one vertex for each instance of each literal in f .
 - 1.2. Create an edge between each pair of vertices that correspond to literals in the same clause.
 - 1.3. Create an edge between each pair of vertices that correspond to complementary literals (i.e., two literals that are the negation of each other).
2. Return $\langle G, k \rangle$.

For example, consider the formula $(P \vee \neg Q \vee W) \wedge (\neg P \vee S \vee T)$. From this formula, R will build the graph shown in Figure 28.4.

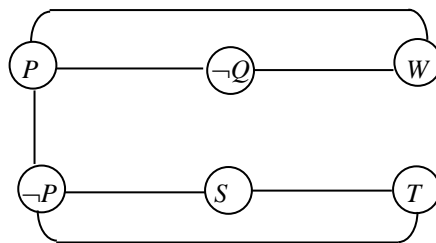


Figure 28.4 Graph gadgets represent a Boolean formula

So each literal gadget is a single vertex and each clause gadget is a set of three vertices plus the edges that connect them.

R runs in polynomial time. To show that it is correct, we must show that $f \in 3\text{-SAT}$ iff $R(\langle f \rangle) \in \text{INDEPENDENT-SET}$.

We first show that $\langle f \rangle \in 3\text{-SAT} \rightarrow R(\langle f \rangle) \in \text{INDEPENDENT-SET}$. If $\langle f \rangle \in 3\text{-SAT}$ then there exists a satisfying assignment A of values to the variables in f . We can use that assignment to show that G , the graph that R builds, contains an independent set S of size at least (in fact, exactly) k . Build S as follows: From each clause gadget choose one literal that is made positive by A . (There must be one since A is a satisfying assignment.) Add the vertex corresponding to that literal to S . S will contain exactly k vertices. And it is an independent set because:

- No two vertices come from the same clause. So step 1.2 could not have created an edge between them.
- No two vertices correspond to complementary literals. So step 1.3 could not have created an edge between them.

Next we show that $R(\langle f \rangle) \in \text{INDEPENDENT-SET} \rightarrow \langle f \rangle \in 3\text{-SAT}$. If $R(\langle f \rangle) \in \text{INDEPENDENT-SET}$ then the graph G that R builds contains an independent set S of size at least (again, in fact, exactly) k . We can use that set to show that there exists some satisfying assignment A for f . Notice that no two vertices in S come from the same clause gadget (because, if they did, they would be connected in G). Since S contains at least k vertices, no two are from the same clause, and f contains k clauses, S must contain one vertex from each clause. So build A as follows: Assign the value *True* to each literal that corresponds to a vertex in S . This is possible because no two vertices in S correspond to complementary literals (again because, if they did, they would be connected in G). Assign arbitrary values to all other literals. Since each clause will contain at least one literal whose value is *True*, the value of f will be *True*. ■

If we want to decide 3-SAT, it is unlikely that we would choose to do so by reducing it to INDEPENDENT-SET and then deciding INDEPENDENT-SET. That wouldn't make sense since none of our techniques for deciding INDEPENDENT-SET run any faster than some obvious methods for deciding 3-SAT. But, having done this reduction, we're in a new position if somehow a fast technique for deciding INDEPENDENT-SET were to be discovered. Then we would instantly also have a fast way to decide 3-SAT.

28.5 NP-Completeness and the Cook-Levin Theorem

We don't know whether $P = NP$. Substantial effort has been expended both in looking for a proof that the two classes are the same and in trying to find a counterexample (i.e., a language that is in NP but not in P) that proves that they are not. Neither of those efforts has succeeded. But what has emerged from that work is a class of NP languages that are maximally "hard" in the sense that if any one of them should turn out to be in P then every NP language would also be in P (and thus P would equal NP). This class of maximally hard NP languages is called NP-complete.

28.5.1 NP-Complete and NP-Hard Languages

Consider two properties that a language L might possess:


1. L is in NP.
2. Every language in NP is deterministic, polynomial-time reducible to L .

Using those properties, we will define:

The Class NP-hard: L is NP-hard iff it possesses property 2. Any NP-hard language L is at least as hard as every language in NP in the sense that if L should turn out to be in P, every NP language must also be in P. Languages that are NP-hard are generally viewed as being *intractable*, meaning that it is unlikely that any efficient (i.e., deterministic, polynomial-time) decision procedure for any of them is likely to exist.

The Class NP-complete: L is NP-complete iff it possesses *both* property 1 and property 2. All NP-complete languages can be viewed as being equivalently hard in the sense that all of them can be decided in nondeterministic, polynomial time and, if any one of them can also be decided in deterministic polynomial time, then all of them can.

Note that the difference between the classes NP-hard and NP-complete is that NP-hard contains some languages that appear to be harder than the languages in NP (in the sense that no nondeterministic, polynomial-time decider for them is known to exist). To see the difference, consider two families of languages whose definitions are based on popular games:

The languages that correspond to generalizations of many one-person games (or puzzles) are NP-complete. For example, consider the generalization of Sudoku  (described in § 781) to an $n \times n$ grid (where n is a perfect square). Then define the following language:

- $\text{SUDOKU} = \{ \langle b \rangle : b \text{ is a configuration of an } n \times n \text{ grid and } b \text{ has a solution under the rules of Sudoku} \}$.

SUDOKU is in NP because there exists a straightforward verifier that checks a proposed solution. It has also been shown to be NP-complete.

The complexity of Sudoku is similar to that of other interesting puzzles. C 781.

On the other hand, the languages that correspond to generalizations of many two-person games are NP-hard but thought not to be in NP. For example, consider the generalization of chess to an $n \times n$ board. Then define the language:

- CHESS = $\{ \langle b \rangle : b \text{ is a configuration of an } n \times n \text{ chess board and there is a guaranteed win for the current player} \}$.

The complexity of the language CHESS explains the fact that it took almost 50 years between the first attempts at programming computers to play chess and the first time that a program beat a reigning chess champion. Some other games, like Go, are still dominated by human players. C 780.

In Section 28.9, we'll return to the issue of problems that appear not to be in NP. For now we can just notice that the reason that CHESS appears not to be in NP is that it is not possible to verify that a winning move sequence exists just by checking a single sequence. It appears necessary to check all sequences that result from the choices that could be made by the opposing player. This can be done in exponential time using depth-first search, so CHESS is an element of the complexity class EXPTIME (which we'll define precisely later).


The class NP-complete is important. Many of its members correspond to problems, like the traveling salesman problem, that have substantial practical significance. It is also one of the reasons that it appears unlikely that $P = NP$. A deterministic, polynomial-time decider for any member of NP-complete would prove that P and NP are the same. Yet, despite substantial effort on many of the known NP-complete problems, no such decider has been found.

But how can we prove that a language L is NP-complete? To do so requires that we show that every other language that is in NP is deterministic, polynomial-time reducible to it. We can't show that just by taking some list of known NP languages and cranking out the reductions. There is an infinite number of NP languages.

If we had even one NP-complete language L' , then we could show that a new language L is NP-complete by showing that L' is deterministic, polynomial-time reducible to it. Then every other NP language could be reduced first to L' and then to L . But how can we get that process started? We need a "first" NP-complete language.

28.5.2 The Cook-Levin Theorem and the NP-Completeness of SAT

Steven Cook and Leonid Levin solved the problem of finding a first NP-complete language by showing that SAT is NP-complete. Their proof does not depend on reducing individual NP languages to SAT. Instead it exploits the fact that a language is in NP precisely in case there exists some nondeterministic, polynomial-time Turing machine M that decides it. Cook and Levin showed that there exists a polynomial-time algorithm that, given $\langle M \rangle$, maps any string w to a Boolean formula that describes the sequence of steps that M executes on input w . They showed further that this reduction guarantees that the formula it constructs is satisfiable iff M ends its computation by accepting w . So, if there exists a deterministic, polynomial-time algorithm that decides SAT, then any NP language (decided by some Turing machine M), can be decided in deterministic, polynomial time by running the reduction (based on $\langle M \rangle$) on w and then running the SAT decider on the resulting formula.

Because SAT is NP-complete, it is unlikely that a polynomial-time decider for it exists. But Boolean satisfiability is a sufficiently important practical problem that there exists an entire annual conference  devoted to the study of both theoretical and applied research in this area. We'll say more about the development of efficient SAT solvers in B 612.

It is interesting to note that the NP-completeness proof that we are about to do is not the first time that we have exploited a reduction that works because an arbitrary Turing machine computation can be simulated using some other (superficially quite different) structure:

- We sketched, in the proof of Theorem 22.4, Turing’s argument that the Entscheidungsproblem is not decidable. That proof makes use of a reduction that maps $\langle M \rangle$ to a first-order logic sentence that is provable (given a particular set of axioms) iff M ever prints 0. The reduction exploits a construction that builds a formula that describes the sequence of configurations that M enters as it computes.
- We showed, in the proof of Theorem 22.1, that PCP, the Post Correspondence Problem language, is not decidable. We did that by defining a reduction that maps an $\langle M, w \rangle$ pair to a PCP instance in such a way that the computation of M on w can be simulated by the process of forming longer and longer partial solutions to the PCP instance. Then we showed that that process ends with a complete solution to the PCP instance iff M halts and accepts w .
- We argued, in the proof of Theorem 22.2, that TILES, which corresponds to a set of tiling problems, is not even semidecidable. We did that by defining a reduction from a Turing machine to a set of tiles in such a way that each new row of any tiling would correspond to the next configuration of the Turing machine as it performed its computation. So there exists an infinite tiling iff the Turing machine fails to halt.
- We show, in the proof of Theorem 41.1, that a simple security model is undecidable. We do that by defining a reduction from an arbitrary Turing machine to an access control matrix. Then we show how the computation of the Turing machine could be simulated by a sequence of operations on the access control matrix in such a way that the property q_f leaks iff the Turing machine halts.

The key difference between those proofs and the one we are about to show for the Cook-Levin Theorem is that the reduction we’ll describe here works with a description of a Turing machine M that is known to halt (along all computational paths) and to do so in polynomial time.

Theorem 28.16 Cook-Levin Theorem

Theorem: $\text{SAT} = \{ \langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable} \}$ is NP-complete.

Proof: By Theorem 28.12, SAT is in NP. So it remains to show that it is NP-hard (i.e., that all NP languages are deterministic, polynomial-time reducible to it).

We’ll show a generic reduction R from any NP language to SAT. To use R as a reduction from a particular language $L \in \text{NP}$, we will provide it with M , one of the nondeterministic, polynomial-time deciding machines that must exist for L . Then, when R is applied to a particular string w , it will construct a (large but finite) Boolean formula that is satisfiable iff at least one of M ’s computational paths accepts w .

To see how R works, imagine a two-layer table that describes one computational path that M can follow on some particular input w . An example of such a table is shown in Figure 28.5. Imagine that the second tier, shown in bold, is overlaid on top of the first tier.

	$f(w)$							$\max(f(w), w)$							
$f(w) + 1$								q_0/\square	a	a	b				
									q_1/a	b	b				
										q_1/b	b				
										a	b	q_1/b			
										a	b	b	q_2/\square		
										a	b	b		y/ \square	
										a	b	b		y/ \square	

Figure 28.5 A table that describes one computational path of M on w .

Each row of the table corresponds to one configuration of M . The first row corresponds to M ’s starting configuration (in which the read/write head is positioned on top of the blank immediately to the left of the first symbol of w) and each succeeding row corresponds to the configuration that results from the one before it. Some row will correspond

to M 's halting configuration and we won't care what any of the rows after that one look like. We don't know how many steps M executes before it halts so we don't know exactly how many rows we will need. But we do know that $timereq(M)$ is some polynomial function $f(|w|)$ that is an upper bound on the number of steps. So we'll just let the table have $f(|w|) + 1$ rows.

The lower tier of the table will encode the contents of M 's tape. The upper tier will indicate M 's current state and the position of M 's read/write head. So all the cells of the upper tier will be empty except the one in each row that corresponds to the square that is under the read/write head. Each of those nonempty cells will contain a symbol that corresponds to M 's current state. So, in the table shown above, the upper tier is empty except for the cells that contain an expression of the form q_n .

In each configuration, M 's tape is infinite in both directions. But only a finite number of squares can be visited by M before it halts. We need only represent the squares that contain the original input plus the others that might be visited. It is possible that M spends all of its steps moving in a single direction. So, after $f(|w|)$ steps, the read/write head might be $f(|w|)$ squares to the left of where it began. Or it might be $f(|w|)$ squares, including the original input string, to the right of where it began. In order to allow room for either of these worst cases, we will include, in each tape description, the $f(|w|)$ tape squares to the left of the initial read/write head position and $max(f(|w|), |w|)$ tape squares to the right of it.

For example, suppose that, on input aab , one of M 's paths runs for 5 steps and halts in the accepting state y . If $f(3) = 6$, then that path might be described in the table shown above (where q_n/c means that the lower tier contains the symbol c and the upper tier contains q_n).

To make it easier to talk about this table, let $rows = f(|w|) + 1$ be the number of rows it contains and let $cols = f(|w|) + max(f(|w|), |w|) + 1$ be the number of columns it contains. Let $padleft = f(|w|)$.

The job of the reduction R , with respect to some particular Turing machine M , is to map a string w to a Boolean formula that describes a table such as the one above. R will guarantee that the formula it builds is satisfiable iff all of the following conditions are met:

1. The formula describes a legal table in the sense that:
 - 1.1. The upper tier contains exactly one state marker per row.
 - 1.2. The lower tier contains exactly one symbol per tape square.
2. The formula describes a table whose first row represents the initial configuration of M on input w .
3. The formula describes a table some row of which represents an accepting configuration of M on input w (i.e., the upper tier contains the state y).
4. The formula describes a table that simulates a computation that M could actually perform. So every row, except the first and any that come after the accepting configuration, represents a configuration that, given the transition relation that defines M , can immediately follow the configuration that is described by the preceding row.

Given these constraints, checking whether the formula that R builds is satisfiable is equivalent to checking that there exists some computation of M that accepts w .

It would be easy to write a first-order logic formula that satisfies conditions 1-4. We could write quantified formulas that said things like, "In every row there exists a square that contains a state symbol and every other square in that row does not contain a state symbol". But the key to defining R is to realize that we can also write a Boolean formula that says those same things. The reason we can is that we know that M halts and we have a bound on the number of steps it will execute before it halts. So we know the size of the table that describes its computation. That means that, instead of creating variables that can range over rows or tape squares, we can simply create individual variables for each property of each cell in the table. (Notice, by the way, that if we tried to take the approach we're using here and use it to reduce an arbitrary, i.e., not necessarily NP language, to SAT, it wouldn't work because we would then have no such bound on the size of the table. So, for example, we couldn't try this with the halting language H.)

Imagine the cells in the computation table that we described above as being labeled with a row number i and a column number j . We'll label the cell in the upper left corner $(1, 1)$. Let Γ be M 's tape alphabet and let K be the set of its states. We can now define the variables that R will use in mapping a particular input w :

- For each i and j such that $1 \leq i \leq rows$ and $1 \leq j \leq cols$ and for each symbol c in Γ , create the variable $tape_{i,j,c}$. When the variable $tape_{i,j,c}$ is used in a formula that describes a computational table, it will be assigned the value *True* if $cell(i, j)$ contains the tape symbol c . Otherwise it will be *False*. These variables then describe the lower tier of the computational table.
- For each i and j such that $1 \leq i \leq rows$ and $1 \leq j \leq cols$ and for each state q in K , create the variable $state_{i,j,q}$. When the variable $state_{i,j,q}$ is used in a formula that describes a computational table, it will be assigned the value *True* if $cell(i, j)$ contains the state symbol q . Otherwise it will be *False*. These variables then describe the upper tier of the computational table.

We're now ready to describe the process by which R maps a string w to a Boolean formula $DescribeMonw$, which will be composed of four conjuncts, each corresponding to one of the four conditions we listed above. In order to be able to state these formulas concisely, we'll define the notations:

$$\bigwedge_{1 \leq i \leq rows} tape_{i,j,k} \quad \text{and} \quad \bigvee_{1 \leq i \leq rows} tape_{i,j,k}$$

The first represents the Boolean AND of a set of propositions and the second represents their Boolean OR.

CONJUNCT 1: The first conjunct will represent the constraint that the table must describe a single computational path. Without this constraint it would be possible, if M is nondeterministic, to satisfy all the other constraints and yet describe a table that jumbles multiple computational paths together (thus telling us nothing about any of them).

For each cell (i, j) , we need to say that the variable corresponding to some tape symbol c is *True* and all of the ones corresponding to other tape symbols are *False*. So, for a given (i, j) let $T_{i,j}$ say that $cell(i, j)$ contains symbol c_1 and not any others or it contains symbol c_2 and not any others and so forth up to symbol $c_{|\Gamma|}$:

$$T_{i,j} \equiv \bigvee_{c \in \Gamma} (tape_{i,j,c} \wedge (\bigwedge_{\substack{s \in \Gamma \\ s \neq c}} \neg tape_{i,j,s}))$$

Then let $Tapes$ say that this is true for all squares in the table. So:

$$Tapes \equiv \bigwedge_{1 \leq i \leq rows} (\bigwedge_{1 \leq j \leq cols} T_{i,j})$$

We also need to say that each row contains exactly one state symbol. So let $Q_{i,j}$ say that $cell(i, j)$ contains exactly one state symbol:

$$Q_{i,j} \equiv \bigvee_{q \in K} (state_{i,j,q} \wedge (\bigwedge_{\substack{p \in K \\ p \neq q}} \neg state_{i,j,p}))$$

Then let $States$ say that, for each row, there is exactly one column for which that is true:

$$States \equiv \bigwedge_{1 \leq i \leq rows} (\bigvee_{1 \leq j \leq cols} (Q_{i,j} \wedge (\bigwedge_{\substack{1 < k \leq cols \\ k \neq j}} \bigwedge_{q \in K} \neg state_{i,k,q})))$$

So we have:

$$Conjunct_1 \equiv Tapes \wedge States$$

CONJUNCT 2: The second conjunct will represent the constraint that the first row of the table must correspond to M 's initial configuration when started on input w . Assume that M 's start state is q_0 . We'll first describe the lower tier of the table (the symbols on M 's tape). The first $padleft + 1$ squares will be blank. Then the input string w will appear and then all remaining squares will be blank. Let $w[j]$ be the j^{th} symbol in w . Let:

$$Blanks \equiv \left(\bigwedge_{1 \leq j \leq padleft+1} tape_{1,j,blank} \right) \wedge \left(\bigwedge_{padleft+|w|+2 \leq j \leq cols} tape_{1,j,blank} \right)$$

$$Initialw \equiv \bigwedge_{padleft+2 \leq j \leq padleft+|w|+1} tape_{1,j,w[j]}$$

Now we describe the upper tier of the table. We need to say that M is in state q_0 with its read/write head immediately to the left of the first square of w . Let:

$$Initialq \equiv state_{1,padleft+1,q_0}.$$

Then we have:

$$Conjunct_2 \equiv Blanks \wedge Initialw \wedge Initialq.$$

CONJUNCT 3: The third conjunct will represent the constraint that M 's computation must halt in the accepting state y . This means that some cell in the upper tier of the table must contain the state symbol y .

$$Conjunct_3 \equiv \bigvee_{1 \leq i \leq row} \bigvee_{1 \leq j \leq cols} state_{i,j,y}$$

CONJUNCT 4: The fourth and last conjunct will represent the constraint that successive rows of the table must correspond to successive configurations in a possible computation of M . To construct this conjunct, our reduction R must have access to M 's transition relation Δ .

The key to the construction of $conjunct_4$ can be seen by looking through a small window at the large computation table that we are working with. An example of such a window is shown in Figure 28.6. Notice that each successive configuration of M is nearly identical to the previous one. The only tape square that can change its value is the one under the read/write head. And the read/write head can move only one square to the left or one square to the right.

•
•

q_1/a	a	a	b	□	□
b	q_1/a	a	b	□	□
b	a	q_1/a	b	□	□
b	a	b	q_1/b	□	□
b	a	b	b	q_2/\square	□

•
•

Figure 28.6 A window into a computation table

Call the first row in which the accepting state y occurs *done*. Now consider all rows from 2 until *done*. (We don't care what happens in any rows after the one in which y appears. They are just in the table because we had to make sure there was enough room for all the rows that matter.) What we want to say is that, comparing row i to row $i-1$:

- All the tape squares that aren't under the read/write head stayed the same. Let:

$$Sames \equiv \forall 2 \leq i \leq done (\forall j (\forall c (\text{read/write head not in column } j \text{ in row } i-1 \rightarrow (\text{tape}_{i,j,c} \leftrightarrow \text{tape}_{i-1,j,c}))).$$

- The tape square under the read/write head changed in some way that is allowed by Δ :

$$\begin{aligned} ChangedTape \equiv \forall 2 \leq i \leq done (\forall j (\forall c (\text{read/write head in column } j \text{ in row } i-1 \text{ and } \text{tape}_{i,j,c} \rightarrow \\ \exists p (\text{state stored in row } i-1 = p, \text{ and} \\ \exists s (\text{character in column } j \text{ in row } i-1 = s, \text{ and} \\ \exists q (((p, s), (q, c), (\rightarrow|\leftarrow))) \in \Delta)))). \end{aligned}$$

- The state and the read/write head changed in some way that is allowed by Δ . There are two possibilities: either the read/write head moved one square to the right or it moved one square to the left:

$$ChangedStateAndHead \equiv \forall 2 \leq i \leq done (\forall j (\forall q (\text{state}_{i,j,q} \rightarrow \text{moved-right} \vee \text{moved-left}))), \text{ where:}$$

$$\begin{aligned} \text{moved-right} \equiv (\exists p (\text{state stored in row } i-1 = p, \text{ and} \\ \exists s (\text{character in column } j-1 \text{ in row } i-1 = s, \text{ and} \\ \exists c (((p, s), (q, c), (\rightarrow))) \in \Delta))). \end{aligned}$$

$$\begin{aligned} \text{moved-left} \equiv (\exists p (\text{state stored in row } i-1 = p, \text{ and} \\ \exists s (\text{character in column } j+1 \text{ in row } i-1 = s, \text{ and} \\ \exists c (((p, s), (q, c), (\leftarrow))) \in \Delta))). \end{aligned}$$

This last conjunct is the most complex of the four. So we will skip the step in which we convert the quantified formulas we've just presented to equivalent Boolean ones. By now it should be clear that since we are quantifying over a finite set of objects, doing that is straightforward, although tedious. So we have:

$$Conjunct_4 \equiv \text{the Boolean equivalent of: } Sames \wedge ChangedTape \wedge ChangedStateAndHead.$$

The final formula that R produces: We can now state R . On input w , it uses $\langle M \rangle$ and constructs a description of the Boolean formula:

$$DescribeMonw = Conjunct_1 \wedge Conjunct_2 \wedge Conjunct_3 \wedge Conjunct_4.$$

$DescribeMonw$ will have a satisfying assignment to its variables iff there exists some computational path along which M accepts w . So, for any NP language L , $L \leq SAT$.

It remains to show that $R(w)$ operates in polynomial time. The number of variables in $DescribeMonw$ can be computed as follows: We know that the number of steps that M will execute on input w is bounded by some polynomial function $f(|w|)$. So the number of cells in the computational table is $\mathcal{O}(f(|w|)^2)$. Call that number *cellcount*. To represent the bottom tier of the table requires *cellcount*· $|\Gamma|$ variables. To represent the top tier of the table requires *cellcount*· $|K|$ variables. Since both $|\Gamma|$ and $|K|$ are independent of $|w|$, the number of variables is then $\mathcal{O}(f(|w|)^2)$. So the number of characters required to encode each instance of a variable when it occurs in a literal in $DescribeMonw$ is ($\mathcal{O}(\log f(|w|)^2)$), which is polynomial in $|w|$.

Constructing each of the conjuncts that form $DescribeMonw$ is straightforward. But we must show that the length of each of them is bounded by some polynomial function of w :

- Conjunct 1: Each formula $T_{i,j}$ contains $|\Gamma|$ literals. So $Tapes$ contains $cellcount \cdot |\Gamma|$ literals. Each formula $W_{i,j}$ contains $|K|$ literals. So $States$ contains $cellcount \cdot cols \cdot |K| \in \mathcal{O}(f(|w|)^3)$ literals and $Conjunct_1$ contains $\mathcal{O}(f(|w|)^3)$ literals.
- Conjunct 2: We require $cols$ literals to describe the tape contents and 1 to describe the state and read/write head. So $Conjunct_2$ contains $\mathcal{O}(f(|w|))$ literals.
- Conjunct 3: $Conjunct_3$ contains $\mathcal{O}(f(|w|)^2)$ literals.
- Conjunct 4: The straightforward way to convert the quantified expressions we have provided into the required Boolean formulas nests ANDs and ORs to correspond to the nested universal and existential quantifiers. If we do that, then we will get formulas with at most $cellcount^2 \cdot |K|^2 \cdot |\Gamma|^2$ literals. Again, since $|K| \cdot |\Gamma|$ are independent of w , we have that $Conjunct_4$ contains $\mathcal{O}(f(|w|)^2)$ literals.

So $|DescribeMonw|$ is polynomial in $|w|$ and it can be constructed in polynomial time. ■

28.6 Other NP-Complete Problems

The Cook-Levin Theorem gives us our first NP-complete language. In this section we'll see that it is not alone ☐.

28.6.1 A Sampling of NP-Complete Languages

We've already described many languages that can be shown to be NP-complete. In fact every NP language that we have mentioned, except for those that we have said are in P, is provably NP-complete. So all of the following languages are NP-complete:

- SAT = $\{ \langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable} \}$.
- 3-SAT = $\{ \langle w \rangle : w \text{ is a wff in Boolean logic, } w \text{ is in 3-conjunctive normal form and } w \text{ is satisfiable} \}$.
- TSP-DECIDE = $\{ \langle G, cost \rangle, \text{ where } \langle G \rangle \text{ encodes an undirected graph with a positive distance attached to each of its edges and } G \text{ contains a Hamiltonian circuit whose total cost is less than } cost \}$.
- HAMILTONIAN-PATH = $\{ \langle G \rangle : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian path} \}$.
- HAMILTONIAN-CIRCUIT = $\{ \langle G \rangle : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian circuit} \}$.
- CLIQUE = $\{ \langle G, k \rangle : G \text{ is an undirected graph with vertices } V \text{ and edges } E, k \text{ is an integer, } 1 \leq k \leq |V|, \text{ and } G \text{ contains a } k\text{-clique} \}$.
- INDEPENDENT-SET = $\{ \langle G, k \rangle : G \text{ is an undirected graph and } G \text{ contains an independent set of at least } k \text{ vertices} \}$.
- SUBSET-SUM = $\{ \langle S, k \rangle : S \text{ is a multiset (i.e., duplicates are allowed) of integers, } k \text{ is an integer, and there exists some subset of } S \text{ whose elements sum to } k \}$.
- SET-PARTITION = $\{ \langle S \rangle : S \text{ is a multiset (i.e., duplicates are allowed) of objects each of which has an associated cost and there exists a way to divide } S \text{ into two subsets, } A \text{ and } S - A, \text{ such that the sum of the costs of the elements in } A \text{ equals the sum of the costs of the elements in } S - A \}$.
- KNAPSACK = $\{ \langle S, v, c \rangle : S \text{ is a set of objects each of which has an associated cost and an associated value, } v \text{ and } c \text{ are integers, and there exists some way of choosing elements of } S \text{ (duplicates allowed) such that the total cost of the chosen objects is at most } c \text{ and their total value is at least } v \}$.

- SUDOKU = $\{ \langle b \rangle : b \text{ is a configuration of an } n \times n \text{ Sudoku grid and } b \text{ has a solution} \}$.

Examples of other languages that are also NP-complete include:

- SUBGRAPH-ISOMORPHISM = $\{ \langle G_1, G_2 \rangle : G_1 \text{ is isomorphic to some subgraph of } G_2 \}$. Two graphs G and H are *isomorphic* to each other iff there exists a way to rename the vertices of G so that the result is equal to H . Another way to think about isomorphism is that two graphs are isomorphic iff their drawings are identical except for the labels on the vertices.

The subgraph isomorphism problem arises naturally in many domains. For example, consider the problem of matching two chemical structures to see if one occurs within another.

- BIN-PACKING = $\{ \langle S, c, k \rangle : S \text{ is a set of objects each of which has an associated size and it is possible to divide the objects so that they fit into } k \text{ bins, each of which has size } c \}$.

The bin packing problem can be extended to two and three dimensions and it remains NP-complete. The two-dimensional problem arises, for example, in laying out a newsletter with k pages and a set of stories and pictures that need to be placed on the pages. The three-dimensional problem arises, for example, in assigning cargo to a set of trucks or train cars.

- SHORTEST-SUPERSTRING = $\{ \langle S, k \rangle : S \text{ is a set of strings and there exists some superstring } T \text{ such that every element of } S \text{ is a substring of } T \text{ and } T \text{ has length less than or equal to } k \}$.

The shortest superstring problem arises naturally during DNA sequencing. The problem there is to find the most likely larger molecule from which a set of fragments were derived. © 738.

- BOUNDED-PCP = $\{ \langle P, k \rangle : P \text{ is an instance of the Post Correspondence problem (as described in Section 22.2) that has a solution of length less than or equal to } k \}$.

28.6.2 Proving That a Language is NP-Complete

To prove that a new language is NP-complete, we will exploit the following theorem. Recall that when we write $L_1 \leq_P L_2$, we mean that L_1 is polynomial-time mapping reducible to L_2 .

Theorem 28.17 Using Reduction to Prove NP-Completeness

Theorem: If L_1 is NP-complete, $L_1 \leq_P L_2$, and L_2 is in NP, then L_2 is also NP-complete.

Proof: If L_1 is NP-complete then every other NP language is deterministic, polynomial-time reducible to it. So let L be any NP language and let R_L be the Turing machine that reduces L to L_1 . If $L_1 \leq_P L_2$, let R_2 be the Turing machine that implements that reduction. Then L can be deterministic, polynomial-time reduced to L_2 by first applying R_L and then applying R_2 . Since L_2 is in NP and every other language in NP is deterministic, polynomial-time reducible to it, it is NP-complete. ■

Theorem 28.17 tells us that we can use reduction from any known NP-complete language to show that a new language is also NP-complete. At this point, we have only one such language: SAT. So we will begin by using it. Once we have others, we can use whichever one makes the required reduction easy. In fact, the first thing we will do is to show that 3-SAT, a close relative of SAT, is NP-complete. Then we'll have 3-SAT as a tool to use in our other reductions.

28.6.3 3-SAT

In Section 28.2.5 we defined:

- $3\text{-SAT} = \{\langle w \rangle : w \text{ is a wff in Boolean logic, } w \text{ is in 3-conjunctive normal form, and } w \text{ is satisfiable}\}.$

3-SAT is a somewhat contrived language. It is significant primarily because doing reductions from 3-SAT is often substantially easier than doing them from SAT. 3-SAT's restricted form limits the number of conditions that must be considered, as we saw in the reduction we did, in Theorem 28.15, from 3-SAT to INDEPENDENT-SET.

Theorem 28.18 3-SAT is NP-Complete

Theorem: 3-SAT is NP-complete.

Proof: We showed, in Theorem 28.13, that 3-SAT is in NP. So all that remains is to show that it is NP-hard (i.e., that every other language in NP is deterministic, polynomial-time reducible to it).

We could show that 3-SAT is NP-hard if we could show a polynomial-time reduction from SAT to it. Define:

$R(w: \text{wff of Boolean logic}) =$

1. Use *conjunctiveBoolean* (as defined in the proof of Theorem B 1) to construct w' , where w' is in conjunctive normal form and w' is equivalent to w .
2. Use *3-conjunctiveBoolean* (as defined in the proof of Theorem B 2) to construct w'' , where w'' is in 3-conjunctive normal form and w'' is satisfiable iff w' is.
3. Return w'' .

If R ran in polynomial time, it would be the reduction that we need. In Exercise 28.4, we show that step two does run in polynomial time. Unfortunately, step one does not. The length of w' (and thus the time required to construct it) can grow exponentially with the length of w . There are two approaches that we could take to solving this problem:

- We can retain the idea of reducing SAT to 3-SAT. We observe that, for R to be a reduction from SAT to 3-SAT, it is not necessary that w' be equivalent to w . It is sufficient to assure that w' is satisfiable iff w is. There exists a polynomial-time algorithm (described in [Hopcroft, Motwani and Ullman 2001]) that constructs, from any wff w , a w' that meets that requirement. If we replace step one of R with that algorithm, R is a polynomial-time reduction from SAT to 3-SAT, so 3-SAT is NP-hard.
- We can prove that 3-SAT is NP-hard directly, using a variant of the proof we offered for the Cook-Levin Theorem. It is possible to modify the reduction R that proves the Cook-Levin Theorem so that it constructs a formula in conjunctive normal form. R will still run in polynomial time. We leave the proof of this claim as Exercise 28.13. Once R has constructed a conjunctive normal form formula w , we can use *3-conjunctiveBoolean* to construct w' , where w' is in 3-conjunctive normal form and w' is satisfiable iff w is. This composition of *3-conjunctiveBoolean* with R shows that any NP language can be reduced to 3-SAT. So 3-SAT is NP-hard. ■

28.6.4 INDEPENDENT-SET

Recall that, given a graph G , an independent set is a set of vertices of G , no two of which are adjacent (i.e., connected by a single edge). Using that definition, we defined the following language, which we can now show is NP-complete:

- $\text{INDEPENDENT-SET} = \{\langle G, k \rangle : G \text{ is an undirected graph and } G \text{ contains an independent set of at least } k \text{ vertices}\}.$

Theorem 28.19 INDEPENDENT-SET is NP-Complete

Theorem: INDEPENDENT-SET is NP-complete.

Proof: We must prove that INDEPENDENT-SET is in NP and that it is NP-hard (i.e., that every other language in NP is deterministic, polynomial-time reducible to it).

INDEPENDENT-SET is in NP: We describe Ver , a deterministic, polynomial-time verifier for it: Let G be a graph with vertices V and edges E . Let c be a certificate for $\langle G, k \rangle$; c will be a list of vertices. On input $\langle G, k, c \rangle$, Ver checks that the number of vertices in c is at least k and no more than $|V|$. If it is not, it rejects. Next it considers each vertex in c one at a time. For each such vertex v , it finds all edges in E that have v as one endpoint. It then checks that the other endpoint of each of those edges is not in c . $Timereq(Ver) \in \mathcal{O}(|c| \cdot |E| \cdot |c|)$. Both $|c|$ and $|E|$ are polynomial in $|\langle G, k \rangle|$. So Ver runs in polynomial time.

INDEPENDENT-SET is NP-hard because Theorem 28.15 tells us that $3\text{-SAT} \leq_p \text{INDEPENDENT-SET}$. ■

28.6.5 VERTEX-COVER

A vertex cover C of a graph G with vertices V and edges E is a subset of V with the property that every edge in E touches at least one of the vertices in C . Obviously V is a vertex cover of G . But we are typically interested in finding a smaller one. So we define the following language, which we will show is NP-complete:

- VERTEX-COVER = $\{\langle G, k \rangle: G \text{ is an undirected graph and there exists a vertex cover of } G \text{ that contains at most } k \text{ vertices}\}$.

To be able to test every link in a network, it suffices to place monitors at a set of vertices that form a vertex cover of the network. $\text{€ } 701$.

We will show that VERTEX-COVER (also called NODE-COVER) is NP-complete by reducing 3-SAT to it. The proof will provide another example of the use of carefully constructed gadgets that map the literals and clauses that occur in strings in 3-SAT to the vertices and edges described by strings in VERTEX-COVER. Alternatively, we could prove that VERTEX-COVER is NP-complete with a very simple reduction from INDEPENDENT-SET (since, if S is an independent set in some graph G with vertices V and edges E , then $V - S$ is a vertex cover of G). We leave that alternative proof as an exercise.

Theorem 28.20 VERTEX-COVER is NP-Complete

Theorem: VERTEX-COVER is NP-complete.

Proof: We must prove that VERTEX-COVER is in NP and that it is NP-hard.

VERTEX-COVER is in NP: We describe Ver , a deterministic, polynomial-time verifier for it: Let G be a graph with vertices V and edges E . Let c be a certificate for $\langle G, k \rangle$; c will be a list of vertices. On input $\langle G, k, c \rangle$, Ver checks that the number of vertices in c is at most $\min(k, |V|)$. If it is not, it rejects. Next it considers each vertex in c one at a time. For each such vertex v , it finds all edges in E that have v as one endpoint and it marks each such edge. Finally, it makes one pass through E and checks whether every edge is marked. If all of them are, it accepts; otherwise it rejects. $Timereq(Ver) \in \mathcal{O}(|c| \cdot |E|)$. Both $|c|$ and $|E|$ are polynomial in $|\langle G, k \rangle|$. So Ver runs in polynomial time.

VERTEX-COVER is NP-hard: We prove this by demonstrating a reduction R that shows that:

$$3\text{-SAT} \leq_p \text{VERTEX-COVER}.$$

R 's job is to map a Boolean formula f (in 3-conjunctive normal form) to a graph. It will exploit two kinds of gadgets:

- A variable gadget: For each variable x in f , R will build a simple graph with two vertices and one edge between them. Label one of the vertices x and the other one $\neg x$.
- A clause gadget: For each clause c in f , R will build a graph with three vertices, one for each literal in c . There will be an edge between each pair of vertices in this graph.

The variable and clause gadgets must then be connected to correspond to the structure of f . R will build an edge from every vertex in a clause gadget to the vertex of the variable gadget with the same label.

So, for example, given the Boolean formula $(P \vee \neg Q \vee T) \wedge (\neg P \vee Q \vee S)$, R will build the graph shown in Figure 28.7.

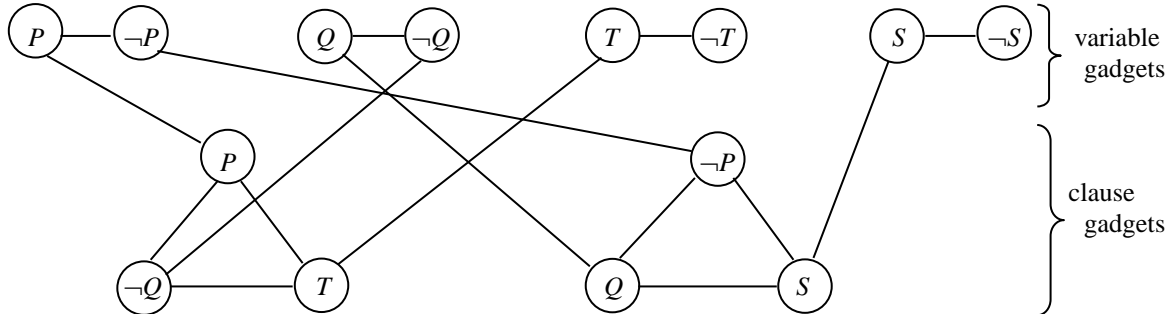


Figure 28.7 Reducing 3-SAT to VERTEX-COVER

Let f be a Boolean formula with c clauses and v variables. Then we can define R as follows:

- $R(\langle f \rangle) =$
1. Build a graph G as described above.
 2. Let $k = v + 2c$.
 3. Return $\langle G, k \rangle$.

R runs in polynomial time. To show that it is correct we must show that $\langle f \rangle \in 3\text{-SAT}$ iff $R(\langle f \rangle) \in \text{VERTEX-COVER}$.

We first show that $\langle f \rangle \in 3\text{-SAT} \rightarrow R(\langle f \rangle) \in \text{VERTEX-COVER}$. If $\langle f \rangle \in 3\text{-SAT}$ then there exists a satisfying assignment A of values to the variables in f . We can use that assignment to show that G , the graph that R builds, contains a vertex cover C of size at most (in fact, exactly) k . We can construct C by doing the following:

1. From each variable gadget, select the vertex that corresponds to the literal that is true in A . Add each of those vertices to C .
2. Since A is a satisfying assignment, there must exist at least one true literal in each clause. Pick one and put the vertices corresponding to the other two into C .

C contains exactly k vertices. And it is a cover of G because:

- One vertex from every variable gadget is in C so all the edges that are internal to the variable gadgets are covered.
- Two vertices from every clause gadget are in C so all the edges that are internal to the clause gadgets are covered.
- All the vertices that connect variable gadgets to clause gadgets are covered because, for each clause gadget:
 - Two of the three emerging edges are covered by the two clause gadget vertices in C .
 - The other one must be connected to a variable gadget vertex that corresponds to a true literal, so that vertex is in C .

Next we show that $R(\langle f \rangle) \in \text{VERTEX-COVER} \rightarrow \langle f \rangle \in 3\text{-SAT}$. If $R(\langle f \rangle) \in \text{VERTEX-COVER}$ then the graph G that R builds contains a vertex cover C of size at most (again, in fact, exactly) k . Notice that C must:

- Contain at least one vertex from each variable gadget in order to cover the internal edge in the variable gadget.
- Contain at least two vertices from each clause gadget in order to cover all three internal edges in the clause gadget.

Satisfying those two requirements uses up all k vertices, so the vertices we have just described are the only vertices in C . We can use C to show that there exists some satisfying assignment A for f . Building A is simple: assign the value

True to each literal that is the label for one of the vertices that comes from a variable gadget. We note that A is a satisfying assignment for f iff it assigns the value *True* to at least one literal in each of f 's clauses.

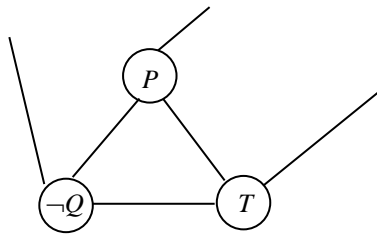


Figure 28.8 A clause gadget

To see why it is certain that A does this, consider an arbitrary clause gadget in G , as shown in Figure 28.8. Since C is a cover for G , all six of the edges that connect to vertices in this gadget must be covered. But we know that only two of the vertices in the gadget are in C . They can cover the three internal edges. But the three edges that connect to the variable gadgets must also be covered. Only two can be covered by a vertex in the clause gadget. The other one must be covered by its other endpoint, which is in some variable gadget. So each clause is connected to some literal whose corresponding vertex is in C . We made each such literal *True* in A . So A assigns the value *True* to at least one literal in each clause. Thus it is a satisfying assignment for f . ■

28.6.6 HAMILTONIAN-CIRCUIT and the Traveling Salesman Problem

We started our discussion of complexity, at the beginning of Chapter 27, by considering the traveling salesman problem. We observed then that, while there exists an obvious exponential algorithm for solving the problem, there does not exist an *obvious* polynomial algorithm for solving it exactly. While it remains an open question whether *any* polynomial algorithm for the traveling salesman problem does in fact exist, we can now prove a result that suggests that it is relatively unlikely that one does. TSP-DECIDE is NP-complete.

We have already shown that TSP-DECIDE is in NP. But we must also show that it is NP-hard, which we will do by reducing 3-SAT to it. It turns out to be easier to map 3-SAT to appropriate graph structures if the graph edges are directed. So we will introduce a new language:

- DIRECTED-HAMILTONIAN-CIRCUIT = $\{ \langle G \rangle : G \text{ is a directed graph and } G \text{ contains a Hamiltonian circuit} \}$.

Then we will prove that:

$$3\text{-SAT} \geq_P \text{DIRECTED-HAMILTONIAN-CIRCUIT} \geq_P \text{HAMILTONIAN-CIRCUIT} \geq_P \text{TSP-DECIDE}.$$

Theorem 28.21 DIRECTED-HAMILTONIAN-CIRCUIT is NP-Complete

Theorem: DIRECTED-HAMILTONIAN-CIRCUIT is NP-complete.

Proof: We must prove that DIRECTED-HAMILTONIAN-CIRCUIT is in NP and that it is NP-hard.

DIRECTED-HAMILTONIAN-CIRCUIT is in NP: We describe Ver , a deterministic, polynomial-time verifier for it: Let G be a graph with vertices V and edges E . Let c be a certificate for $\langle G \rangle$; c will be a list of vertices. On input $\langle G, c \rangle$, Ver checks that the number of vertices in c is $|V|+1$. If it is not, it rejects. It also rejects if the first and last vertices are not identical. Next it considers each vertex v in c , except the last, one at a time. It marks v in V and rejects if it had previously been marked. It also checks that the required edge to v exists and rejects if it does not. If it finishes without rejecting, it accepts. $Timereq(Ver) \in \mathcal{O}(|c| \cdot (|V| + |E|))$. All of $|c|$, $|V|$, and $|E|$ are polynomial in $|\langle G, k \rangle|$. So Ver runs in polynomial time.

DIRECTED-HAMILTONIAN-CIRCUIT is NP-hard: We prove this by demonstrating a reduction R that shows that:

3-SAT \leq_p DIRECTED-HAMILTONIAN-CIRCUIT.

R 's job is to map a Boolean formula Bf (in 3-conjunctive normal form) to a graph. R will exploit two kinds of gadgets, one to correspond to the variables of Bf and the other to correspond to the clauses.

We'll describe the variable gadgets first. Let n be the number of variables in the Boolean formula Bf . If v is the i^{th} such variable, let m be the larger of the number of occurrences of v or of $\neg v$ in Bf . The gadget that corresponds to v will have the structure shown in Figure 28.9. We'll call this gadget V_i .

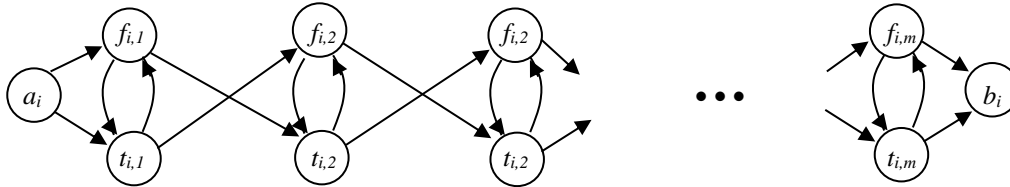


Figure 28.9 A variable gadget in the reduction from 3-SAT to DIRECTED-HAMILTONIAN-CIRCUIT

Now imagine a Hamiltonian path (not a circuit) through V_i . It must enter V_i from the left at a_i and leave it on the right at b_i . There are only two ways to do that. If the path begins by going down to a t vertex, then it must next go straight up to the matching f vertex, then crosswise to the next t vertex, up to the matching f vertex, and so forth. Similarly, if the path begins by going up to an f vertex, it must next go straight down to the matching t vertex, then crosswise to the next f vertex, and so forth. A path that did anything else would not be Hamiltonian since it would not visit all the vertices. So there are two paths through V :

- The one that begins by going down to a t vertex. We will use this one to correspond to assigning to the variable v the value *True*
- The one that begins by going up to an f vertex. We will use this one to correspond to assigning to the variable v the value *False*.

R will build the variable gadgets V_1 through V_n and then combine them into a single structure V , as shown in Figure 28.10. Suppose that H is a Hamiltonian circuit through V . Then H must enter each of the variable gadgets exactly once (through its a vertex), choose one of the two paths through that gadget (thus effectively choosing to make the corresponding variable either *True* or *False*), leave that variable gadget (through its b vertex), and then enter the next one.

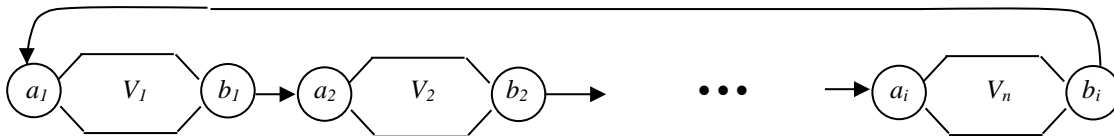


Figure 28.10 Stringing the variable gadgets together

Next we must describe the clause gadgets. The gadget that corresponds to the i^{th} clause in the formula Bf will have the structure shown in Figure 28.11. We'll call this gadget C_i .

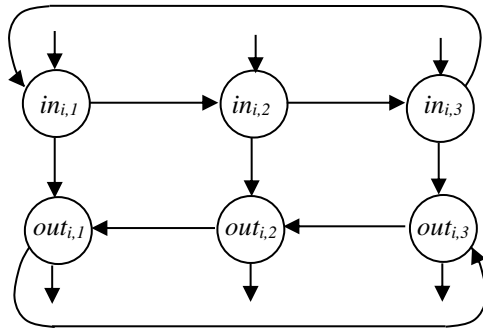


Figure 28.11 A clause gadget in the reduction from 3-SAT to DIRECTED-HAMILTONIAN-CIRCUIT

Suppose that C_i is part of a graph G that contains some Hamiltonian cycle H . H must enter through one of C_i 's in vertices. Further, note that if it enters in column j it must also leave through column j . To see why this is so, we consider all the paths it could take. From $in_{i,j}$, H can:

- Go straight down to $out_{i,j}$ and exit.
- Proceed across to the next in vertex and then down to the matching out one. From there it can go to the next out vertex (which will be $out_{i,j}$) and exit. It cannot simply exit right away because, if it does, there is no way for H to reach $out_{i,j}$. The two vertices that could precede it are already in H and neither of them went to it. Without $out_{i,j}$, H can't be Hamiltonian.
- Proceed across to the next in vertex and then the next one. From there it can go down to the matching out vertex, then across to the next and then to the next (which will be $out_{i,j}$) and then exit. It cannot exit at either of the other out vertices since, if it did, there would again be no way for H to reach $out_{i,j}$.

R 's final job is to connect the variable gadgets and the clause gadgets to form a single graph G that corresponds to the initial Boolean formula Bf . Its goal is to do so in such a way that there will be a Hamiltonian circuit through G iff Bf is satisfiable. The idea is that, if such a circuit exists, it will primarily correspond to a circuit through V , the variable gadget graph. As V has been defined, such a circuit exists. In fact, several exist since there are two paths through each of the individual variable gadgets. So what R must do now is to connect V to the clause gadgets so that there will still be a Hamiltonian circuit if Bf is satisfiable. If, on the other hand, Bf is not satisfiable, the introduction of the clause gadgets will produce a graph through which no Hamiltonian circuit exists. What R is going to do is to use the clause gadgets to introduce detours through V so that this is true.

In each clause gadget C , think of the first column as corresponding to its corresponding clause's first literal, the second column as corresponding to the second literal, and the third column as corresponding to the third literal. R will create three detours from V into C and back, one for each of those literals. So R will consider each of C 's three columns in turn. The literal that corresponds to that column is either some variable v or its negation $\neg v$:

- Suppose R is working on column i and the corresponding literal is v . Then R will go to the gadget for v and choose the first of its columns whose t vertex has not yet been chosen. (Remember that the number of columns in v 's gadget is equal to the larger of the number of instances of v or of $\neg v$, so such a column will always be able to be chosen.) Suppose that the vertex labeled $t_{v,j}$ is chosen. R will create a detour from $t_{v,j}$ to C and then back into V to whatever vertex $t_{v,j}$ previously linked to. If we end up choosing the path through v 's gadget that corresponds to assigning v the value *True*, then that successor vertex is $f_{v,j}$. So, when working on column i , R will create a detour by adding a vertex from $t_{v,j}$ to $in_{C,i}$ and from $out_{C,i}$ to $f_{v,j}$.
- Suppose, on the other hand, that the corresponding literal is $\neg v$. Then R will go to the gadget for v and choose the first of its columns whose f vertex has not yet been chosen. Suppose that the vertex labeled $f_{v,j}$ is chosen. Just as above, R will create a detour from the chosen vertex into C and then back. But this time it will assume that we will end up choosing the path through v 's gadget that corresponds to assigning v the value *False*. In that case, the

successor vertex of $f_{v,j}$ is $t_{v,j}$. So, when working on column i , R will create a detour by adding a vertex from $f_{v,j}$ to $in_{C,i}$ and from $out_{C,i}$ vertex to $t_{v,j}$.

To see how these detours work, consider the simple example shown in Figure 28.12. We show the gadget for the variable P (which we've assumed needs just two columns). We also show the gadget for the clause $(P \vee Q \vee S)$. When R considers that gadget's first column, it goes to the gadget for P and finds the first available t vertex. Assume it's the first one. Then it adds to G the two dashed edges.

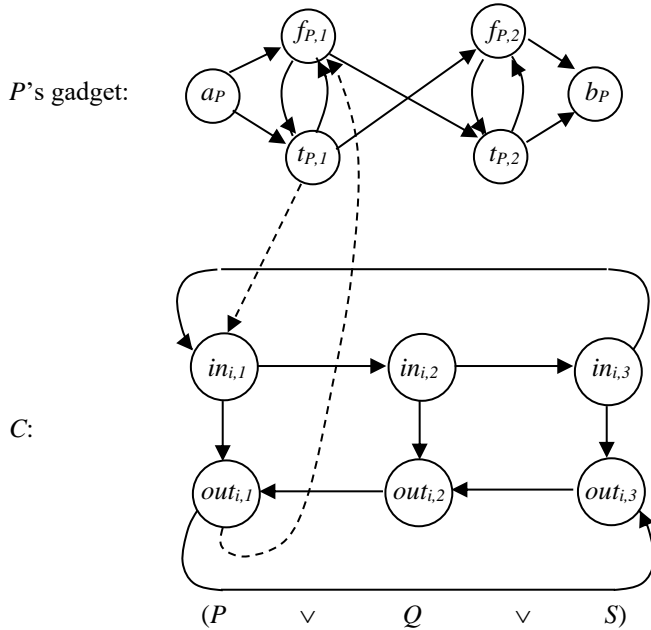


Figure 28.12 Combining the variable and the clause gadgets

Notice what effect these two new edges have on our ability to find a Hamiltonian circuit through G . If such a circuit is traversing P 's gadget in the *True* direction (i.e., it starts by going from a_P down to $t_{P,1}$), then it can now pass through all the vertices of C , leave C , and then continue through the rest of P 's gadget. If, on the other hand, such a circuit is traversing P 's gadget in the *False* direction, it cannot. It can enter C , but when it leaves it would have to return to a vertex ($f_{P,1}$) that it has already visited.

Let Bf be a Boolean formula. We can define a reduction R from 3-SAT to DIRECTED-HAMILTONIAN-CIRCUIT as follows:

- $R(\langle Bf \rangle) =$
1. Build the graph G as described above.
 2. Return $\langle G \rangle$.

R runs in polynomial time. To show that it is correct we must show that $\langle Bf \rangle \in 3\text{-SAT}$ iff $R(\langle Bf \rangle) \in \text{DIRECTED-HAMILTONIAN-CIRCUIT}$.

We first show that $\langle Bf \rangle \in 3\text{-SAT} \rightarrow R(\langle Bf \rangle) \in \text{DIRECTED-HAMILTONIAN-CIRCUIT}$. If $\langle Bf \rangle \in 3\text{-SAT}$, then there exists a satisfying assignment A of values to the variables in Bf . We can use that assignment to show that G , the graph that R builds, contains a Hamiltonian circuit. We can construct such a circuit H as follows: Begin by letting H be just a Hamiltonian circuit through V . We have a choice, for each variable gadget, of two paths through it. If A assigns the variable v the value *True*, then choose the path that begins by going to the first t vertex in v 's gadget. If,

on the other hand, A assigns v the value *False*, then choose the path that begins by going to the first f vertex in v 's gadget.

But now we must add to H the vertices in all the clause gadgets. Since A is a satisfying assignment, each clause c must contain at least one literal to which A assigns the value *True*. Pick one. If it is the (unnegated) variable v , look at v 's gadget. There will be an edge from one of v 's t vertices (call it $t_{v,k}$) into c 's clause gadget at some *in* vertex $in_{c,i}$, and then back out again (from the same column) to the f vertex ($f_{v,k}$) immediately above $t_{v,k}$. H currently includes an edge from $t_{v,k}$ to $f_{v,k}$. Remove that edge and insert, in its place, the edge from $t_{v,k}$ to $in_{c,i}$. Then add the edges that visit the other two vertices in the top row of c 's gadget, followed by the three vertices in its bottom row. Finally add the edge that leaves c 's gadget at $out_{c,i}$ and returns to v at $f_{v,k}$.

To see how this works, consider the simple case shown in Figure 28.13. Assume that Bf contains the clause $c = (\neg P \vee P \vee \neg S)$ and that the only variables in Bf are P and S . Then the graph G that R builds will contain the two fragments shown in the figure: V , the variable gadget structure, and C , the gadget for c . Notice that there are three paths into and out of C , one corresponding to $\neg P$, one corresponding to P , and one corresponding to R . (Ignore the distinction between solid and dashed lines for the moment.)

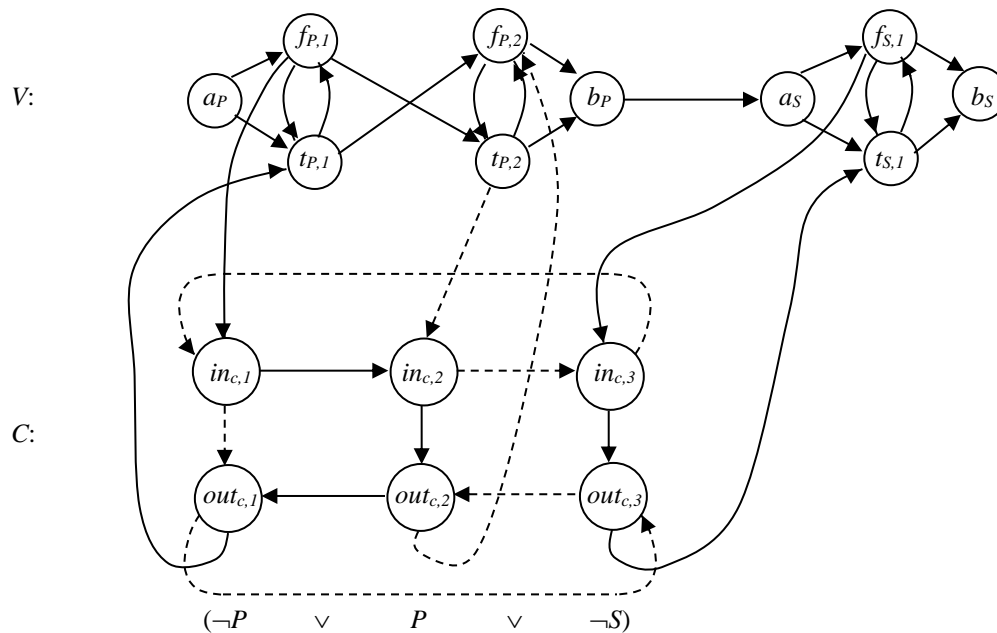


Figure 28.13 How the detours work

Suppose that P is assigned the value *True* by A and that P is the *True* literal that we pick as we are building H . Because A assigns P the value *True*, H 's path through P 's gadget will be a_P then $t_{P,1}$ then $f_{P,1}$ then $t_{P,2}$ and so forth. Initially H contains all the edges in V . But now we remove from it the edge from $t_{P,2}$ to $f_{P,2}$ and replace it by the set of edges shown above as dashed lines. H can still continue its path through V . But now it also detours and visits every vertex in C . And it visits each of them only once because we apply this operation to exactly one of $(\neg P \vee P \vee \neg R)$'s *True* literals.

Now suppose that, for some clause c , the *True* literal that we pick is $\neg v$. Then we do almost the same thing except that now there will be an edge from one of v 's f vertices (call it $f_{v,k}$) into c 's clause gadget and then back out again to the t vertex (call it $t_{v,k}$) immediately below $f_{v,k}$. H currently includes an edge from $f_{v,k}$ to $t_{v,k}$. Remove that edge and insert, in its place, the edge from $f_{v,k}$ into c 's gadget. Then, just as above, add the edges that visit the other two vertices

in the top row of c 's gadget, followed by the three vertices in its bottom row. Finally add the edge that leaves c 's gadget and returns to v at $t_{v,k}$.

H is a Hamiltonian circuit through the graph G that R builds. It includes every vertex in V exactly once. It contains exactly one detour into each clause gadget and that detour visits all six of the vertices in that gadget. So every vertex in G is contained in H exactly once.

It remains to show that $R(\langle Bf \rangle) \in \text{DIRECTED-HAMILTONIAN-CIRCUIT} \rightarrow \langle Bf \rangle \in 3\text{-SAT}$. If $R(\langle Bf \rangle) \in \text{DIRECTED-HAMILTONIAN-CIRCUIT}$ then the graph G that R builds contains a Hamiltonian circuit we can call H . We use H to construct A , a satisfying assignment of values to the variables of Bf . Building A is simple: Examine each variable gadget in G . If H follows the *True* path through the gadget corresponding to variable v (i.e., it begins by going from a_v to $t_{v,l}$), then assign v the value *True*. If, on the other hand, H follows the *False* path through v 's gadget (i.e., it begins by going from a_v to $f_{v,l}$), then assign v the value *False*. Since H is Hamiltonian, it goes through each clause gadget exactly once. And, since it is Hamiltonian, one of the following two things must be true, for each clause gadget, given the way G was constructed:

- H connects to the clause gadget in a column that corresponds to a positive literal v and it does so by a detour from a *True* path. In this case, A assigns v the value *True* and so the clause is satisfied.
- H connects to the clause gadget in a column that corresponds to a negated literal $\neg v$ and it does so by a detour from a *False* path. In this case, A assigns v the value *False*. So $\neg v$ is *True* and so the clause is satisfied.

Since each of its clauses is satisfied, Bf is also satisfied. ■

The reduction R that we just described, from 3-SAT to DIRECTED-HAMILTONIAN-CIRCUIT, only worked because the edges in the graph that R built were directed. But the fundamental question, “Does a Hamiltonian circuit exist?”, is just as hard to answer for undirected graphs. We prove that result next, using a very simple reduction from DIRECTED-HAMILTONIAN-CIRCUIT.

Theorem 28.22 HAMILTONIAN-CIRCUIT is NP-Complete

Theorem: HAMILTONIAN-CIRCUIT = $\{\langle G \rangle : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian circuit}\}$ is NP-complete.

Proof: We must prove that HAMILTONIAN-CIRCUIT is in NP and that it is NP-hard.

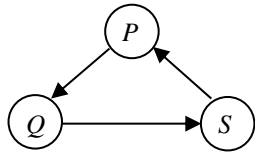
HAMILTONIAN-CIRCUIT is in NP: *Ver*, the verifier that we just described in the proof of Theorem 28.21, also works here. It will simply consider undirected edges instead of requiring directed ones.

HAMILTONIAN-CIRCUIT is NP-hard: we prove this by demonstrating a reduction R that shows that:

$$\text{DIRECTED-HAMILTONIAN-CIRCUIT} \leq_p \text{HAMILTONIAN-CIRCUIT}.$$

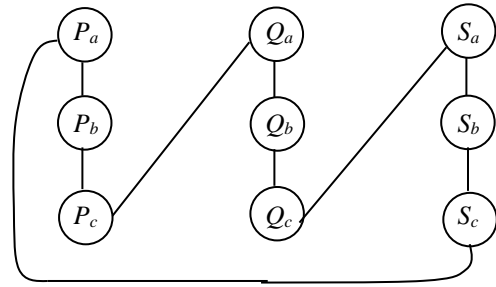
Given a directed graph G , R will build an undirected graph G' . Each of G 's vertices will be represented in G' by a gadget that contains three vertices connected by two edges. Further, if there is a directed edge in G from v to w , then G' will contain an (undirected) edge from the last of the vertices in v 's gadget to the first of the vertices in w 's gadget. Figure 28.14 shows a simple example.

Given G (which does contain a Hamiltonian circuit):

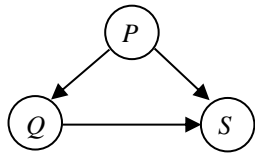


R will build:

G' :



Given G (which does not contain a Hamiltonian circuit):



R will build:

G' :

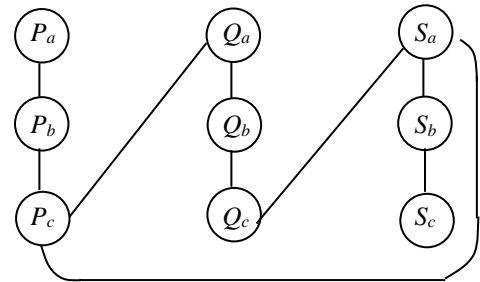


Figure 28.14 Reductions from DIRECTED-HAMILTONIAN-CIRCUIT to HAMILTONIAN-CIRCUIT

Let G be a directed graph. We can define a reduction R from DIRECTED-HAMILTONIAN-CIRCUIT to HAMILTONIAN-CIRCUIT as follows:

$R(\langle G \rangle) =$

1. Build the graph G' as described above.
2. Return $\langle G' \rangle$.

R runs in polynomial time. To show that it is correct we must show that $\langle G \rangle \in \text{DIRECTED-HAMILTONIAN-CIRCUIT}$ iff $R(\langle G \rangle) \in \text{HAMILTONIAN-CIRCUIT}$.

We first show that $\langle G \rangle \in \text{DIRECTED-HAMILTONIAN-CIRCUIT} \rightarrow R(\langle G \rangle) \in \text{HAMILTONIAN-CIRCUIT}$. G must contain at least one Hamiltonian circuit, which we will call H . Assume that $H = (v_1, v_2, \dots, v_k, v_1)$. Then we can describe H' , a Hamiltonian circuit through G' . It starts at the top of v_1 's gadget, walks down through it, then goes to the top of v_2 's gadget, walks down through it, and so forth until it has visited the last vertex of v_k 's gadget. It ends by returning to the first vertex of v_1 's gadget. In other words,

$$H' = (v_{1a}, v_{1b}, v_{1c}, v_{2a}, v_{2b}, v_{2c}, \dots, v_{ka}, v_{kb}, v_{kc}, v_{1a}).$$

It remains to show that $(R(\langle G \rangle) \in \text{HAMILTONIAN-CIRCUIT}) \rightarrow (\langle G \rangle \in \text{DIRECTED-HAMILTONIAN-CIRCUIT})$. Notice that, in any graph that R builds, each b vertex is attached to exactly two edges. So any Hamiltonian circuit through such a vertex comes either down from the top, or up from the bottom, of the corresponding gadget. Pick a gadget. If a Hamiltonian circuit through it goes down from the top, then it must continue to the top of some

other gadget. So it must go down through that one as well. And it must continue through all the gadgets, in each case going down from the top. Alternatively, it can move bottom to top through all the gadgets. The key is that it must move in the same direction through all the vertex gadgets. If $R(\langle G \rangle) \in \text{HAMILTONIAN-CIRCUIT}$, then the graph G' that R builds must contain at least one Hamiltonian circuit. Pick one and call it H . Assume that H traverses G' 's gadgets top to bottom. (If it goes in the other direction, then, since G' is undirected, there is another Hamiltonian circuit through G' that is identical to H except that it moves in the other direction. Choose it instead.) Note that H can only traverse the gadget for v and then the gadget for w in case there was a directed edge from v to w in the original graph G . So, suppose H visits the gadgets for the vertices $(v_1, v_2, \dots, v_k, v_1)$, in that order. Then $(v_1, v_2, \dots, v_k, v_1)$ is a Hamiltonian circuit through G . ■

We are now in a position to return to the traveling salesman problem, with which we began the previous chapter.

Theorem 28.23 TSP-DECIDE is NP-Complete

Theorem: $\text{TSP-DECIDE} = \{\langle G, cost \rangle : \langle G \rangle \text{ encodes an undirected graph with a positive distance attached to each of its edges and } G \text{ contains a Hamiltonian circuit whose total cost is less than } cost\}$ is NP-complete.

Proof: We have already shown (in Theorem 28.10) that TSP-DECIDE is in NP. It remains to prove that it is NP-hard, which we do with a straightforward reduction R that shows that:

$$\text{HAMILTONIAN-CIRCUIT} \leq_p \text{TSP-DECIDE}.$$

Let G be an unweighted, undirected graph with vertices V . R must map G into a weighted, undirected graph plus a cost. We observe that, if there is a Hamiltonian circuit through G , it must contain exactly $|V|$ edges. So suppose that we augment G with edge costs by assigning to every edge a cost of 1. Then, if there is a Hamiltonian circuit in G , its total cost must be equal to $|V|$. Because this is true, we can define R as follows:

$$R(\langle G \rangle) =$$

1. From G construct G' , a weighted graph. G' will be identical to G except that each edge will be assigned the cost 1.
2. Return $\langle G', |V| \rangle$.

R runs in polynomial time. And it is correct since G has a Hamiltonian circuit iff G' has one with cost equal to $|V|$. ■

28.7 The Relationship between P and NP-Complete

So far, every NP language that we have considered has turned out also either to be in P or to be NP-complete. Is it necessarily true that every NP language has that property? The answer is no. In fact, unless $P = NP$, there must exist languages that don't.

28.7.1 The Gap between P and NP-Complete

Call the class of NP-complete languages NPC. Let $\text{NPL} = \text{NP} - (P \cup \text{NPC})$. In other words, NPL is the limbo area between P and NP-complete. Trivially, if $P = \text{NP}$ then $\text{NPL} = \emptyset$. But what if (as seems more likely) $P \neq \text{NP}$? We can prove the following theorem that tells us that, in that case, NPL is not empty.

Theorem 28.24 Ladner's Theorem

Theorem: If $P \neq \text{NP}$, then $\text{NPL} \neq \emptyset$.

Proof: The proof relies on the following more general claim that is proved in [Ladner 1975]:

Claim: Let B be any decidable language that is not in P. There exists a language D that is in P and that has the following property: Let $A = D \cap B$. Then $A \notin P$, $A \leq_p B$, but it is not true that $B \leq_p A$.

Suppose that B is any NP-complete language. Unless $P = NP$, B is not in P . So there must exist a language D that is in P , and from which we can compute $A = D \cap B$. A must be in NP since membership in D can be decided in polynomial time and membership in B can be verified in polynomial time. So the claim that Ladner proved tells us that:

- $A \notin P$, but
- It is not true that $B \leq_P A$. Since B is in NP but is not deterministic, polynomial-time reducible to A , A is not NP-complete.

So A is an example of an NP language that is neither in P nor NP-complete. Thus $NPL \neq \emptyset$. ■

It is possible, using diagonalization techniques, to construct languages that are in NPL. But it remains true that few “natural” languages are in that class. A comprehensive catalogue of NP problems, [Garey and Johnson 1979], lists three candidates for membership in NPL:

- COMPOSITES = $\{w : w \text{ is the binary encoding of a composite number}\}$. Recall that a composite number is a natural number greater than 1 that is not prime.
- LINEAR-PROGRAMMING, which we will describe in Section 28.7.7.
- GRAPH-ISOMORPHISM = $\{\langle G_1, G_2 \rangle : G_1 \text{ is isomorphic to } G_2\}$. Recall that two graphs G and H are isomorphic to each other iff there exists a way to rename the vertices of G so that the result is equal to H .

It is now known that COMPOSITES (see Section 28.1.7) and LINEAR-PROGRAMMING (see Section 28.7.7) are in P .

The jury is still out on GRAPH-ISOMORPHISM. It is easy to show that GRAPH-ISOMORPHISM is in NP. A proposed renaming of the vertices of G_1 so that it matches G_2 is a certificate, which can easily be checked in polynomial time. Recall that the *subgraph* isomorphism language, SUBGRAPH-ISOMORPHISM, which asks whether G_1 is isomorphic to *some subgraph of* G_2 is NP-complete. It appears that the graph isomorphism problem is easier, perhaps because we must compare only G_1 and G_2 , not G_1 and all of G_2 's subgraphs. But graph isomorphism has not been shown to be in P , nor has it been shown not to be NP-hard (and thus NP-complete).

Problems like GRAPH-ISOMORPHISM are rare, though. So, most of the time, an NP problem will turn out either to be NP-complete or to be in P . The question then is, “Which?” It is interesting to note that sometimes what appears to be a slight change in a problem definition makes the difference between a language that is in P and one that is NP-complete. We’ll next consider several examples of this phenomenon.

28.7.2 Two Similar Circuit Problems

Consider the two circuit problems:

- EULERIAN-CIRCUIT, in which we check that there is a circuit that visits every *edge* exactly once.
- HAMILTONIAN-CIRCUIT, in which we check that there is a circuit that visits every *vertex* exactly once.

We have already seen that EULERIAN-CIRCUIT is in P , but HAMILTONIAN-CIRCUIT is NP-complete.

28.7.3 Two Similar SAT Problems

Define 2-conjunctive normal form (2-CNF) analogously to 3-conjunctive normal form (3-CNF) except that each clause must contain exactly two literals. So, for example, $(\neg P \vee R) \wedge (S \vee \neg T)$ is in 2-conjunctive normal form. Despite the use of the term “normal form” here, note that it is not true that every Boolean formula can be rewritten as an equivalent one in 2-CNF. Now consider:

- 2-SAT = { $\langle w \rangle$: w is a wff in Boolean logic, w is in 2-conjunctive normal form and w is satisfiable}.
- 3-SAT = { $\langle w \rangle$: w is a wff in Boolean logic, w is in 3-conjunctive normal form and w is satisfiable}.

2-SAT is in P (which we prove in Exercise 28.5a)). But 3-SAT is NP-complete.

28.7.4 Two Similar Path Problems:

Consider the problem of finding the *shortest* path with no repeated edges through an unweighted graph G . We can convert this to a decision problem by defining the language:

- SHORTEST-PATH = { $\langle G, u, v, k \rangle$: G is an unweighted, undirected graph, u , and v are vertices in G , $k \geq 0$, and there exists a path from u to v whose length is at most k }.

SHORTEST-PATH is in P because the following simple marking algorithm decides it in $\mathcal{O}(|G|^3)$ time:

shortest-path(G : graph with vertices V and edges E , u : vertex, v : vertex, k : integer) =

1. Mark u .
2. For $i = 1$ to $\min(k, |E|)$ do:
 - For each currently marked vertex n do:
 - For each edge from n to some other vertex m do:
 - Mark m .
3. If v is marked then accept; else reject.

We should note here that the simple algorithm *shortest-path* works because we are considering only unweighted graphs. So it suffices simply to count the number of edges that are traversed. If, on the other hand, we want to solve the analogous problem for weighted graphs, the problem is more difficult. But even this problem can also be solved efficiently, for example by using Dijkstra's algorithm \square .

Finding the shortest path through a weighted graph is important in many applications. The obvious ones include finding routes on a map or routing packets through a network. $\text{C } 701$.

But there are many less obvious ones as well, particularly if we allow weighted edges. For example, consider one problem that an optical character recognition (OCR) system must solve: find the boundaries between letters. One way to think about doing this is that the goal is to find as straight as possible a path that cuts between the regions occupied by two characters and that touches as few black pixels as possible. To solve this problem, we model the boundaries between pixels as vertices and we add edges that cut through the pixels from one boundary to another. We assign a weight of one to every edge that cuts through a white pixel and we assign a very large weight to every edge that cuts through a black pixel. Then the lowest-cost path between two regions is the most direct path that cuts through the fewest black pixels.

But now consider the problem of finding the *longest* path with no repeated vertices through an unweighted graph G . We can convert this to a decision problem by defining the language:

- LONGEST-PATH = { $\langle G, u, v, k \rangle$: G is an unweighted, undirected graph, u , and v are vertices in G , $k \geq 0$, and there exists a path with no repeated vertices from u to v whose length is at least k }.

LONGEST-PATH is in NP (since a candidate path can be checked in polynomial time). And it can be shown to be NP-complete.

28.7.5 Two Similar Covering Problems:

Recall that a **vertex cover** (also called a **node cover**) C of a graph G is a subset of the vertices of G with the property that every edge of G touches at least one of the vertices in C . Now define an **edge cover** C of a graph G to be a subset of the edges of G with the property that every vertex of G is an endpoint of at least one of the edges in C . Consider the graph G shown in Figure 28.15. The set of heavy edges is an edge cover of G . The set of circled vertices is a vertex cover of it.

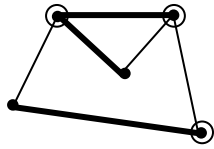


Figure 28.15 An edge cover and a vertex cover

Consider the problem of finding the smallest *edge* cover of a graph. We can convert this to a decision problem by defining the following language:

- EDGE-COVER = $\{ \langle G, k \rangle : G \text{ is an undirected graph and there exists an edge cover of } G \text{ that contains at most } k \text{ edges} \}$.

EDGE-COVER can be shown to be in P. (We leave it as an exercise.) But we have proven that the corresponding vertex-cover language is NP-complete:

- VERTEX-COVER = $\{ \langle G, k \rangle : G \text{ is an undirected graph and there exists a vertex cover of } G \text{ that contains at most } k \text{ vertices} \}$.

28.7.6 Three Similar Map (Graph) Coloring Problems

Consider the problem of coloring a planar map in such a way that no two adjacent regions (countries, states, or whatever) have the same color. We will allow two regions that share only a single common point to have the same color. So all of the map colorings shown in Figure 28.16 are allowed.

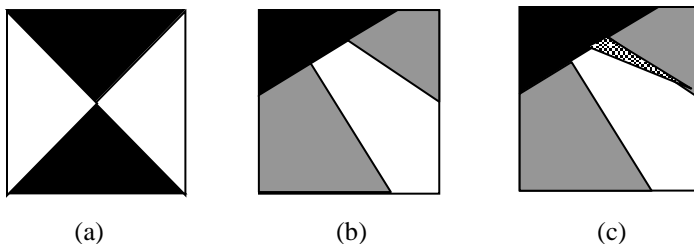


Figure 28.16 Legal map colorings

We'll say that a map is ***n*-colorable** or that it can be colored using n colors iff it can be colored, according to the rule given above, using no more than n distinct colors. Now define the following three languages:

- 2-COLORABLE = $\{ \langle m \rangle : m \text{ is a 2-colorable map} \}$.
- 3-COLORABLE = $\{ \langle m \rangle : m \text{ is a 3-colorable map} \}$.
- 4-COLORABLE = $\{ \langle m \rangle : m \text{ is a 4-colorable map} \}$.

What is the complexity of each of these three languages?

2-COLORABLE is easy. A map is 2-colorable iff it does not contain any point that is the junction of an odd number of regions. We leave the proof of this claim as Exercise 28.21). (The proof of a related claim is given as Exercise 32.22.) Map (a) above is 2-colorable. Maps (b) and (c) are not. There is a simple, polynomial-time algorithm to check this requirement. So 2-COLORABLE is in P.

3-COLORABLE \square is harder. It can be shown to be NP-complete. We leave the proof of this claim as an exercise.

What about 4-COLORABLE? It turns out that 4-COLORABLE is in P. It can be decided by the trivial algorithm that simply accepts any map that it is given. To see why, we'll sketch the history of the **4-Color Problem** \square .

In 1852, Francis Guthrie noticed that he could color all the maps he was working with using only four colors. He asked the question, "Can all planar maps be colored (following the rules described above) using at most four colors?" For over a hundred years, the answer to this question eluded children and mathematicians alike. All attempts to find uncolorable maps failed. Yet neither was there a proof of the **4-Color Theorem**: the claim, articulated by Guthrie, that no such map exists. A few "proofs" were published, but all were shown to contain flaws.

Then, in 1976, a proof that has stood the test of time was announced by Kenneth Appel and Wolfgang Haken. Interestingly (since we are discussing computation), a computer program played a key role in the development of that proof. Appel and Haken showed that the question of whether all maps are four-colorable could be reduced to a set of about 1700 special cases. So it remained to check all of them and show that the maps in each case were four-colorable. Appel and Haken used a computer to do that. When their proof was published, there was some concern about the use of a program as part of a proof. What if, for example, the program were incorrect? In the years since the Appel and Haken proof was published, no programming errors have been discovered. Newer, simpler proofs have also been found.

One reason that the four-color problem is important is that the coloring question applies not just to maps. It applies to a wide range of problems that can be described as graphs. To see why, notice first that a map can be described as an undirected graph in which the vertices correspond to regions and the edges correspond to the adjacency relationships between regions. So there will be an edge between vertices v_1 and v_2 iff the regions that correspond to v_1 and v_2 share a common boundary in the graph. Then the map coloring problem becomes the following graph coloring problem: given a graph G , assign colors to the vertices of G in such a way that no pair of adjacent vertices are assigned the same color. We can define graph equivalents of the three coloring languages that we defined above.

We will define the **chromatic number** of a graph to be the smallest number of colors required to color its vertices, subject to the constraint that no two adjacent vertices may be assigned the same color. In the specific case in which a graph has a chromatic number of two, we'll say that the graph is **bipartite**.

The 4-Color Theorem tells us that the chromatic number of any *planar* graph (i.e., one that corresponds to a map on a plane) must be less than five. (More precisely, a graph is **planar** iff it can be drawn in such a way that no edges cross.) But, if we do not require planarity, there are graphs of arbitrary chromatic numbers. In particular, any complete graph (i.e., one in which there is an edge between every pair of vertices) with k vertices has the chromatic number k . Define the following language:

- CHROMATIC-NUMBER = $\{ \langle G, k \rangle : G \text{ is an undirected graph whose chromatic number is no more than } k \}$.

CHROMATIC-NUMBER is NP-complete.

Many optimization problems can be described as graph-coloring problems. We mention two here:

Consider the problem of scheduling final exams in such a way that no two classes that have any common students share an exam time. We can represent the problem as a graph in which there is a vertex for each class. There is an edge between every pair of classes

that share at least one student. Then the number of required exam slots is the chromatic number of that graph.

Consider the problem of assigning trains to platforms. Clearly no two trains can be assigned to the same platform at the same time. We can represent the problem as a graph in which there is a vertex for each train. There is an edge between every pair of trains that are scheduled to be in the station at the same time. Then the number of required platforms is the chromatic number of that graph.

Note that CHROMATIC-NUMBER and INDEPENDENT-SET are related. CHROMATIC-NUMBER relates a graph G to the number of distinct colors that are required to color it. INDEPENDENT-SET relates G to the largest number of vertices that can be colored with a single color. So, for example, if the exam scheduling problem were described as an instance of INDEPENDENT-SET, we'd be asking about the maximum number of classes that could share a single exam time.

28.7.7 Two Similar Linear Programming Problems:

Linear programming problems \square are optimization problems in which both the objective function and the constraints that must be satisfied are linear. We can cast the linear programming problem as a language to be decided by defining:

- LINEAR-PROGRAMMING = {<a set of linear inequalities $Ax \leq b$ > : there exists a vector X of rational numbers that satisfies all of the inequalities}.

Linear programming is used routinely to solve industrial resource allocation problems.

The simplex algorithm, invented by George Dantzig in 1947, solves linear programming problems (by finding the vector X if it exists). In the worst case, it may require exponential time. But, in practice, it is highly effective and substantial work over the years since its invention has further improved its performance. For example, we mentioned in the introduction to Chapter 27 that it can be used to solve large instances of the traveling salesman problem. Without a decision procedure that could be guaranteed to halt in polynomial time, however, the question of whether LINEAR-PROGRAMMING was in P remained open. In 1979, Leonid Khachian answered the question by exhibiting a new, polynomial time, linear-programming algorithm. Unfortunately, his algorithm performed worse in practice than did the simplex algorithm, so it remained of only theoretical interest. Then, in 1984, Narendra Karmarkar described a polynomial-time, linear-programming algorithm [Karmarkar 1984] that works well in practice. Both the simplex algorithm and a variety of techniques based on Karmarkar's algorithm are commonly used today.

But now consider a slightly different problem in which we require that a solution be a vector of integers (as opposed to arbitrary rationals). We can describe this problem as the language:

- INTEGER-PROGRAMMING = {<a set of linear inequalities $Ax \leq b$ > : there exists an integer vector X that satisfies all of the inequalities}.

INTEGER-PROGRAMMING is known to be NP-complete.

28.7.8 A Hierarchy of Diophantine Equation Problems

A Diophantine equation is a polynomial equation in any number of variables, all with integer coefficients. A Diophantine problem then is, "Given a system of Diophantine equations, does it have an integer solution?" Depending on the restrictions that are imposed on the form of a particular problem, it may be undecidable, decidable but intractable, or tractable (i.e., decidable in polynomial time).

- The general Diophantine problem is undecidable, as we saw in Section 22.1.
- If the problem is restricted to equations of the form $ax^2 + by = c$, where a , b , and c are positive integers and we ask whether there exist integer values of x and y that satisfy the equation, then the problem becomes decidable. But it is NP-complete.

- If the problem is restricted to systems in which all the variables are of degree (exponent) 1 or to equations of a single variable of the form $ax^k = c$, and again we ask for integer values of the variable(s), then it is in P.

28.8 The Language Class co-NP

Given a language L that is in NP, can we say anything about whether $\neg L$ is also in NP? Recall that we are defining the complement of a language to be taken with respect to the universe of strings with the correct syntax whenever it is possible to determine that in polynomial time. So, for example, $\neg\text{TSP-DECIDE} = \{w \text{ of the form: } \langle G, cost \rangle, \text{ where } \langle G \rangle \text{ encodes an undirected graph with a positive distance attached to each of its edges and } \langle G \rangle \text{ does not contain a Hamiltonian circuit whose total cost is less than } cost\}$. Is $\neg\text{TSP-DECIDE} \in \text{NP}$? It is not obviously so. For example, the simple technique we used to prove Theorem 28.1 (that the class P is closed under complement) won't work here. We cannot simply swap accepting and nonaccepting states since, if there were some accepting paths and some rejecting paths, there would then still be some accepting paths and some rejecting ones. So the new machine would accept some strings that are also accepted by the original one. Because the decidable languages are closed under complement, we know that we can build a Turing machine to decide $\neg\text{TSP-DECIDE}$. But the obvious way to do so requires that we explore *all* candidate paths in order to verify that none of them is acceptable. Since the number of candidate paths is $\mathcal{O}(|\langle G \rangle|!)$, we cannot do that in polynomial time. No alternative approach is known to do significantly better. In other words, no nondeterministic polynomial time algorithm to decide $\neg\text{TSP-DECIDE}$ is known.

In order to have a place to put $\neg\text{TSP-DECIDE}$, we define the class co-NP (i.e., the complement of some element of NP) as follows:

The Class co-NP: $L \in \text{co-NP}$ iff $\neg L \in \text{NP}$.

Another way to think about the relationship between NP and co-NP is the following:

- A language L is in NP iff a qualifying certificate, i.e., one that proves that an input string w is in L , can be checked efficiently.
- A language L is in co-NP iff a disqualifying certificate, i.e., one that proves that an input string w is not in L , can be checked efficiently. For example, a string of the form $\langle G, cost \rangle$ is not in $\neg\text{TSP-DECIDE}$ if there exists even one Hamiltonian circuit through G whose cost is less than $cost$. Checking such a proposed circuit can easily be done in polynomial time.

Example 28.2 Two Co-NP Languages: UNSAT and VALID

Two important languages based on properties of Boolean formulas are in co-NP:

- $\text{UNSAT} = \{\langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is not satisfiable}\}$. UNSAT is the complement of SAT (since we are taking complements with respect to the universe of well-formed expressions).
- $\text{VALID} = \{\langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is valid}\}$. Recall that a wff is valid (equivalently, is a tautology) iff it is true for all assignments of truth values to the variables it contains. So w is valid iff $\neg w$ is not satisfiable. Thus we can determine whether a string w is in VALID by constructing the string $\neg w$ (which can be done in constant time) and then checking whether $\neg w$ is in UNSAT.

No one knows whether NP is closed under complement. In other words, we do not know whether $\text{NP} = \text{co-NP}$. For a variety of reasons, it is generally believed that $\text{NP} \neq \text{co-NP}$. We state two such reasons in the next two theorems.

Theorem 28.25 If $\text{NP} \neq \text{co-NP}$ then $\text{P} \neq \text{NP}$

Theorem: If $\text{NP} \neq \text{co-NP}$ then $\text{P} \neq \text{NP}$.

Proof: From Theorem 28.1, we know that the class P is closed under complement. If $P = NP$, then NP must also be closed under complement. If $NP \neq \text{co-NP}$ then NP is not closed under complement. So it cannot equal P. ■

We do not know whether $NP = \text{co-NP}$ implies that $P = NP$. It is possible that $NP = \text{co-NP}$ but that that class is nevertheless larger than P.

Theorem 28.26 $NP = \text{co-NP}$ iff there is Some NP-Complete Language whose Complement is also in NP

Theorem: $NP = \text{co-NP}$ iff there exists some language L such that L is NP-complete and $\neg L$ is also in NP.

Proof: We prove the two directions of the claim separately:

If $NP = \text{co-NP}$ then there exists some language L such that L is NP-complete and $\neg L$ is also in NP: There exists at least one language L (for example, SAT) that is NP-complete. By definition, $\neg L$ is in co-NP. If $NP = \text{co-NP}$ then $\neg L$ must also be in NP.

If there exists some language L such that L is NP-complete and $\neg L$ is also in NP then $NP = \text{co-NP}$: Suppose that some language L is NP-complete and $\neg L$ is also in NP. Then we can show that $NP \subseteq \text{co-NP}$ and $\text{co-NP} \subseteq NP$:

- $NP \subseteq \text{co-NP}$: let L_1 be any language in NP. Since, by assumption, L is NP-complete, there exists a polynomial-time reduction R from L_1 to L . R is also a polynomial time reduction from $\neg L_1$ to $\neg L$. Since, by assumption, $\neg L$ is in NP, there exists a nondeterministic polynomial-time Turing machine M that decides it. So we can decide $\neg L_1$ in nondeterministic polynomial time by first running R and then running M . So $\neg L_1$ is in NP and its complement, L_1 , is in co-NP. Thus every language in NP is also in co-NP.
- $\text{co-NP} \subseteq NP$: let L_1 be any language in co-NP. Then $\neg L_1$ is in NP. Since, by assumption, L is NP-complete, there exists a polynomial-time reduction R from $\neg L_1$ to L . R is also a reduction from L_1 to $\neg L$. Since, by assumption, $\neg L$ is in NP, there exists a nondeterministic polynomial-time Turing machine M that decides it. So we can decide L_1 in nondeterministic polynomial time by first running R and then M . So L_1 is in NP. Thus every language in co-NP is also in NP. ■

Despite substantial effort, no one has yet found a single language that can be proven to be NP-complete and whose complement can be proven to be in NP.

28.9 The Time Hierarchy Theorems, EXPTIME, and Beyond

To prove that a language L has an efficient decision procedure, it suffices to exhibit such a procedure, prove its correctness, and analyze its complexity. In general, proving that *no* efficient decision procedure exists is much more difficult. We know however, that there exist some languages that are inherently hard. We know this for two reasons:

- There exists a set of hierarchy theorems that show that adding resources (in terms of either time or space) increases the set of languages that can be decided.
- There exist some specific decidable languages that can be shown to be hard in the sense that no efficient algorithm to decide them exists.

In the next section, we'll describe the hierarchy theorems and their implications. Then we will define one new (and larger) time-complexity class and consider one example of a naturally-occurring language that can be shown to be very hard.

28.9.1 Time Hierarchy Theorems ✪

There exist two time hierarchy theorems. They formalize the intuitive notion that, as we allow a Turing machine to use more and more time, the set of languages that can be decided grows. So, for any fixed time bound, there must be decidable languages that can be decided within the bound but that cannot be decided using “substantially less” time. One of the theorems applies to deterministic Turing machines; the other applies to nondeterministic ones. There is also a corresponding pair of space hierarchy theorems that make the same case for what happens as the amount of space that can be used grows.

The hierarchy theorems are important. In particular, they tell us that, while it is possible that particular pairs of complexity classes may collapse, it is not possible that all of them do. There are time complexity classes that properly contain other ones (and similarly for space complexity classes). Unfortunately, there are two kinds of important questions that the hierarchy theorems cannot answer:

- They do not tell us what languages lie where in the hierarchy. They are proved by diagonalization so they show only that some language must exist. They are not constructive.
- They do not relate deterministic complexity classes to nondeterministic ones. So, for example, they say nothing about whether $P = NP$. They also do not relate time complexity classes to space complexity classes (such as the ones we will define in the next chapter).

We would like to be able to show that *any* increase in the amount of time that is allowed increases the set of languages that can be decided. Unfortunately, we cannot prove that that is true. The strongest statement that we can prove is that increasing the amount of time by at least a logarithmic factor makes a difference.

We will state and prove the deterministic version of the time hierarchy theorems. The nondeterministic version is similar. The proof that we will do will be by construction of a Turing machine that can do the following two things:

- Compute the value of a *timereq* function, on a given input, and store that value, in binary, on its tape.
- Efficiently simulate another Turing machine for a specified number of steps.

Before we state the theorem and give its complete proof, we’ll discuss how to do each of those things.

Time-Constructible Functions

Our goal will be to show that, given a function $t(n)$, there exists some language $L_{t(n)hard}$ that can be decided in $t(n)$ time but not in “substantially less” time. (We’ll soon see that “substantially less” will mean by a factor of $1/\log t(n)$.) So we will want to be able to conduct a simulation for at most $t(n)/\log t(n)$ steps. We could do that if we could compute $t(n)$ and write it on the simulator’s tape. Then we could divide that number by $\log t(n)$ and use that number as a counter, decrementing it by one for each simulated step and quitting, even if the simulation hasn’t yet halted, if the counter ever reaches zero. We will need an efficient representation of $t(n)$ ’s value. We could choose to use any base other than one. We will choose to represent the value in binary. So what we need is the ability to compute $t(n)$ and store the result in binary. Since n is the length of some Turing machine’s input, we can think of that input as though all of its symbols were 1’s.

So we can compute $t(n)$ if we can map the string 1^n to the binary representation of $t(n)$. We need this computation not to dominate the simulation itself. So we will require that it be able to be done in $\mathcal{O}(t(n))$ time. So define a function $t(n)$ from the positive integers to the positive integers to be *time-constructible* iff:

- $t(n)$ is at least $\mathcal{O}(n \log n)$, and
- the function that maps the unary representation of n (i.e., 1^n) to the binary representation of $t(n)$ can be computed in $\mathcal{O}(t(n))$ time.

Most useful functions, as long as they are at least $\mathcal{O}(n \log n)$, are time-constructible. For example, all polynomial functions that are at least $\mathcal{O}(n \log n)$ are time-constructible. So are $n \log n$, $n\sqrt{n}$, 2^n , and $n!$.

Efficient Bounded Simulation

The proof that we are about to do depends critically on the ability to perform a bounded simulation of one Turing machine by another and to do so efficiently. Any overhead that occurs as part of the simulation will weaken the claim that we are going to be able to make about the impact of additional time on our ability to decide additional languages (because time that gets spent on simulation overhead doesn't get spent doing real work).

The universal Turing machine that we described in Section 17.7 simulates the computation of an arbitrary Turing machine M on an arbitrary input w . But it uses three tapes. If we simply convert that three-tape machine to a one-tape machine as described in Section 17.3.1, then a computation that took $t(n)$ steps on the three-tape machine will take $\mathcal{O}(t(n)^2)$ steps on the corresponding one-tape machine. We can do better. If we look again at the way that the construction of Section 17.3.1 works, we observe that the new, one-tape machine spends most of its time scanning the simulated tapes. First it scans to collect the values under all of the read/write heads. And then it scans again to update each tape in the neighborhood of its read/write head. The fact that the length of any of the tapes may grow as $\mathcal{O}(t(n))$ is what adds the $\mathcal{O}(t(n))$ factor to the time required by the simulation. We can avoid that overhead if we can describe a simulator that uses multiple tapes but that manages them in such a way that it is no longer necessary to scan the length of each tape at each step.

We are about to describe a simulator $BSim$ that does that. $BSim$ also differs from the universal Turing machine in that it takes a third parameter, a time bound b . It will simulate a machine M on input w for b steps or until M halts, whichever comes first. $BSim$ is otherwise like the universal Turing machine that we have already described. In particular, we will assume that the Turing machine that $BSim$ simulates is encoded as for the universal Turing machine. This assumption guarantees that $BSim$ can simulate any Turing machine, regardless of the size of its tape alphabet.

$BSim$ accepts as input a Turing machine M , an input string w , and a time bound b . It uses a single tape that is divided into three tracks. (As in the construction in Section 17.3.1, multiple tracks can be represented on a single tape by using a tape alphabet that contains one symbol for each possible ordered 3-tuple of track values.) The three tracks will be used as follows:

- Track 1 will hold the current value of M 's tape, along with an indication of where its read/write head is.
- Track 2 will hold M 's current state followed by M 's description (i.e., its transition function).
- Track 3 will hold a counter that is initially set to be the time bound b . As each step of M is simulated, the counter will be decremented by 1. The simulation will halt if the counter ever reaches 0 (or if M naturally halts).

The key to $BSim$'s efficiency is that it will keep the contents of its three tracks lined up so that it can find what it needs by examining only a small slice through the tracks. Suppose that the tracks are as shown in Figure 28.17. The position of M 's read/write head is shown as a character in bold.

Track 1:	a	b	a	a	b	b	b	b	a	a	a	a	a	a	b	b	b	a	a	a	a	b	b	b													
Track 2:								<i>state, <M></i>																													
Track 3:								<i>counter</i>																													

Figure 28.17 Lining up the tapes for efficiency

Each time it needs to make a move, $BSim$ needs to check one square on track 1. It also needs to check M 's state and it needs to examine M 's transition function in order to discover what to do. Because of the way that the tracks are lined up, it can do all of these things by scanning its tape starting in the position shown in bold (i.e., the square that corresponds to the current location of M 's read/write head). The number of squares that it must examine on track 2 is a function of the length of M 's description, not the length of its input w or its working tape. So $BSim$ can determine M 's next move in $\mathcal{O}(|<M>|)$ steps.

To make M 's next move, $BSim$ must then:

- Update track 1 as specified by M 's transition function. Doing this requires moving at most one square on track 1, so it takes constant time.
- Update M 's state on track 2. Doing this requires time that is a function of the length of the state description, which is bounded by $|<M>|$. So it takes $\mathcal{O}(|<M>|)$ time.
- Move the contents of track 2 one square to the right or to the left, depending on which way M 's read/write head moved. Doing this takes time that is a function only of M . So it also takes $\mathcal{O}(|<M>|)$ time.

All that remains is to describe how $BSim$ considers b , the bound it has been given. Track 3 contains a counter that has been initialized to a string that corresponds to the binary encoding of b . At each of M 's steps, $BSim$ must:

- Decrement the counter by 1 and check for 0. This can be done in constant time.
- Shift the counter left or right one square so that it remains lined up with M 's read/write head. The number of steps required to do this is a function of the length of the counter. The maximum value of the counter is the original bound, b . Since the counter is represented in binary, its maximum length is $\log b$. So this step takes $\log b$ time.

$BSim$ runs M for no more than b steps. Each step takes $\mathcal{O}(|<M>|)$ time to do the computation plus $\mathcal{O}(\log b)$ time to manage the counter. So $BSim$ can simulate b steps of M in $\mathcal{O}(b \cdot (|<M>| + \log b))$ time.

The Deterministic Time-Hierarchy Theorem

The Deterministic Time-Hierarchy Theorem tells us that changing the amount of available time by a logarithmic factor makes a difference in what can be done. As we'll see, the logarithmic factor comes from the fact that the best technique we have for bounded simulation (as described above) introduces a logarithmic overhead factor. We'll state the theorem precisely using both \mathcal{O} and \mathcal{o} notation. Recall that $f(n) \in \mathcal{o}(g(n))$ iff, for every positive c , there exists a positive integer k such that $\forall n \geq k (f(n) < c g(n))$. In other words, for all but some finite number k of small values, $f(n) < c g(n)$.

Theorem 28.27 Deterministic Time Hierarchy Theorem

Theorem: For any time-constructible function $t(n)$, there exists a language $L_{t(n)hard}$ that is deterministically decidable in $\mathcal{O}(t(n))$ time but that is not deterministically decidable in $\mathcal{o}(t(n)/\log t(n))$ time.

Proof: To prove this claim, we will present a technique that, given a function $t(n)$, finds a language $L_{t(n)hard}$ that has the properties that we seek. We'll define $L_{t(n)hard}$ by describing a Turing machine that decides it in $\mathcal{O}(t(n))$ time. So the first requirement will obviously be met. The only thing that remains is to design it so that any other Turing machine that decides it takes at least $t(n)/\log t(n)$ time. We'll use diagonalization to do that. In particular, we'll make sure that the Turing machine that decides $L_{t(n)hard}$ behaves differently, on at least one input, than any Turing machine that runs in $\mathcal{o}(t(n)/\log t(n))$ time.

$L_{t(n)hard}$ will be a language of Turing machine descriptions with a simple string consisting of a single 1 and then a string of 0's tacked on to the right. More specifically, every string in $L_{t(n)hard}$ will have the form $<M>10^*$. The job of the appendage is to guarantee that L contains some arbitrarily long strings.

The rest of the definition of $L_{t(n)hard}$ is difficult to state in words. Instead, we will define $L_{t(n)hard}$ by describing a Turing machine $M_{t(n)hard}$ that decides it:

$M_{t(n)hard}(w) =$

1. Let n be $|w|$. Compute $t(n)$. Store the result, in binary, on the tape.
2. Divide that number by $\log t(n)$. Store $\lceil t(n)/\log t(n) \rceil$, in binary, on the tape. Call this number b .
3. Check to see that w is of the form $<M>10^*$. If it is not, reject.
4. Check that $|<M>| < \log b$. If it is not, reject.
5. Reformat the tape into the three tracks required by $BSim$. To do this, leave w on track 1. Copy M 's start state and $<M>$ to track 2 starting at the left end of w . Copy b to track 3, also starting at the left end of w .
6. Run $BSim$. In other words, simulate M on w (which is of the form $<M>10^*$) for $\lceil t(n)/\log t(n) \rceil$ steps.

7. If M did not halt in that time, reject.
8. If M did halt and it accepted, reject.
9. If M did halt and it rejected, accept.

The key feature of the way that $M_{t(n)hard}$ is defined is the following: Whenever it runs a simulation to completion, it does exactly the opposite of what the machine it just simulated would have done.

We need to show that $L_{t(n)hard}$, the language accepted by $M_{t(n)hard}$, can be decided in $\mathcal{O}(t(n))$ time and that it cannot be decided (by some other Turing machine) in $\mathcal{O}(t(n)/\log t(n))$ time.

We'll first show that $M_{t(n)hard}$ runs in $\mathcal{O}(t(n))$ time. In a nutshell, on input $\langle M, 10^* \rangle$, $M_{t(n)hard}$ uses its time to simulate $t(n)/\log t(n)$ steps of M , using $\mathcal{O}(\log t(n))$ time for each one. We can analyze it in more detail as follows: Step 1 can be done in $\mathcal{O}(t(n))$ time since $t(n)$ is time-constructible. Step 2 can also be done in $\mathcal{O}(t(n))$ time. Step 3 can be done in linear time if we just check the most basic syntax. It isn't necessary, for example, to make sure that all the states in M are numbered sequentially, even though our description of our encoding scheme specifies that. Step 4 can be done in $\mathcal{O}(t(n))$ time. The point of this check is to make sure that the cost of running the simulation in step 6 is dominated by the total length of w , not by the length of $\langle M \rangle$. Step 5 can be done in linear time.

The core of $M_{t(n)hard}$ is step 6. On input (M, w, b) , $BSim$ requires $\mathcal{O}(b \cdot (|\langle M \rangle| + \log b))$ time. But we have guaranteed (in step 4) that $|\langle M \rangle| < \log b$. So $BSim$ requires $\mathcal{O}(b \log b)$ time. We set b , the number of steps to be simulated, to $t(n)/\log t(n)$. Each simulation step will take $\mathcal{O}(\log t(n))$ time. So the total simulation time will be $\mathcal{O}(t(n))$. Giving a bit more detail, notice that, since b is $t(n)/\log t(n)$, we have:

$$timereq(BSim) \in \mathcal{O}\left(\frac{t(n) \cdot \log(t(n)/\log t(n))}{\log t(n)}\right).$$

Since $t(n) > 1$, we have that $timereq(BSim) \in \mathcal{O}(t(n))$. Steps 7, 8, and 9 take constant time. So $M_{t(n)hard}$ runs in $\mathcal{O}(t(n))$ time.

Now we must show that there is no other Turing machine that decides $L_{t(n)hard}$ substantially more efficiently than $M_{t(n)hard}$ does. Specifically, we must show that no such machine does so in time that is $\mathcal{O}(t(n)/\log t(n))$. Suppose that there were such a machine. We'll call it $M_{t(n)easy}$. For any constant c , $M_{t(n)easy}$ must, on all inputs of length greater than some constant k , halt in no more than $c \cdot t(n)/\log t(n)$ steps. So, in particular, we can let c be 1. Then, on all inputs of length greater than some constant k , $M_{t(n)easy}$ must halt in fewer than $t(n)/\log t(n)$ steps.

What we are going to do is to show that $M_{t(n)easy}$ is not in fact a decider for $L_{t(n)hard}$ because it is not equivalent to $M_{t(n)hard}$. We can do that if we can show even one string on which the two machines return different results. That string will be $w = \langle M_{t(n)easy} \rangle 10^p$, for a particular value of p that we will choose so that:

- $|\langle M_{t(n)easy} \rangle|$ is short relative to the entire length of w . Let n be $|\langle M_{t(n)easy} \rangle 10^p|$. Then, more specifically, it must be the case that $|\langle M_{t(n)easy} \rangle| < \log(t(n)/\log t(n))$. We require this so that $M_{t(n)hard}$ will not reject in step 4. Remember that $M_{t(n)hard}$ checks for this condition in order to guarantee that, when $BSim$ runs, the overhead, at each step, of managing the counter dominates the overhead of scanning M 's description. Let m be $|\langle M_{t(n)easy} \rangle|$. Then this condition will be satisfied if p is at least 2^{2^m} . (We leave as an exercise the proof that this value works.)
- $|w| > k$. On input $w = \langle M_{t(n)easy} \rangle 10^p$, $M_{t(n)hard}$ will simulate $M_{t(n)easy}$ on w for $t(|w|)/\log t(|w|)$ steps. For inputs of length at least k , $M_{t(n)easy}$ is guaranteed to halt within that many steps. That means that $M_{t(n)hard}$ will do exactly the opposite of what $M_{t(n)easy}$ does. Thus the two machines are not identical. This condition is satisfied if p is at least k .

So let p be the larger of k and 2^{2^m} . On input $w = \langle M_{t(n)easy} \rangle 10^p$, the simulation of $M_{t(n)easy}$ on $\langle M_{t(n)easy} \rangle 10^p$ will run to completion. If $M_{t(n)easy}$ accepts, $M_{t(n)hard}$ rejects. And vice versa. This contradicts the assumption that $M_{t(n)easy}$ decides $L_{t(n)hard}$. ■

One consequence of the Deterministic Time Hierarchy Theorem is the claim that we made at the beginning of this chapter, namely that the polynomial time complexity classes do not collapse. There are languages that are deterministically decidable in $\mathcal{O}(n^2)$ time but not in linear time. And there are languages that are deterministically decidable in $\mathcal{O}(n^{2000})$ but not in $\mathcal{O}(n^{1999})$ time. So there are languages that are in P but that are not tractable in any useful sense.

Another consequence is that there are languages that are deterministically decidable in exponential time but not in polynomial time.

28.9.2 EXPTIME

In Section 28.5.1, we suggested that there are languages that are NP-hard but that cannot be shown to be NP-complete because they cannot be shown to be in NP. The example that we mentioned was:

- CHESS = { $\langle b \rangle$: b is a configuration of an $n \times n$ chess board and there is a guaranteed win for the current player}.

We can describe the complexity of CHESS, other “interesting” games like Go, and many other apparently very difficult languages, by defining the class EXPTIME as follows:

The Class EXPTIME: For any language L , $L \in \text{EXPTIME}$ iff there exists some deterministic Turing machine M that decides L and $\text{timereq}(M) \in \mathcal{O}(2^{(n^k)})$ for some positive integer k .

We show that a language is in EXPTIME by exhibiting an algorithm that decides it in exponential time. We sketch such an algorithm for chess (and other two person games) in § 785. In general, if we can describe an algorithm that decides L by exploring *all* of the paths in a tree whose size grows exponentially with the size of the input, then L is in EXPTIME.

As we did for the class NP, we can define a class of equally hard EXPTIME languages. So we consider two properties that a language L might possess:

1. L is in EXPTIME.
2. Every language in EXPTIME is deterministic, polynomial-time reducible to L .

We’ll say that a language is **EXPTIME-hard** iff it possesses property 2. If, in addition, it possesses property 1, we’ll say that it is **EXPTIME-complete**. In § 783, we’ll return to a discussion of the complexity of CHESS. If we make the assumption that, as we add rows and columns to the chess board, we also add pieces, then CHESS can be shown to be EXPTIME-complete.

In Section 29.2, we will define another important complexity class, this time based on space, rather than time, requirements. The class PSPACE contains exactly those languages that can be decided by a deterministic Turing machine whose space requirement grows as some polynomial function of its input. We can summarize what is known about the space complexity classes P, NP, and EXPTIME, as well as the space complexity class PSPACE as follows:

$$P \subseteq NP \subseteq PSPACE \subseteq \text{EXPTIME}.$$

It is not known which of these inclusions is proper. However, it follows from the Deterministic Time Hierarchy Theorem that $P \neq \text{EXPTIME}$. So at least one of them is. It is thought that all of them are.

A consequence of the fact that $P \neq EXPTIME$ is that we know that there are decidable problems for which no efficient (i.e., polynomial time) decision procedure exists. In particular, this must be true for every $EXPTIME$ -complete problem. So, for example, CHESS is provably intractable in the sense that no polynomial-time algorithm for it exists. Practical solutions for $EXPTIME$ -complete problems must exploit techniques like the approximation algorithms that we describe in Chapter 30.

28.9.3 Harder Than $EXPTIME$ Problems

Some problems are even harder than the $EXPTIME$ -complete problems, such as CHESS. We will mention one example.

Recall that, in Section 22.4.2, we proved that the language $FOL_{\text{theorem}} = \{\langle A, w \rangle : A \text{ is a decidable set of axioms in first-order logic, } w \text{ is a sentence in first-order logic, and } w \text{ is entailed by } A\}$ is not decidable. The proof relied on the fact that there exists at least one specific first-order theory that is not decidable. In particular, it relied on the theory of Peano arithmetic, which describes the natural numbers with the functions *plus* and *times*.

The fact that not all first-order theories are decidable does not mean that none of them is. In particular, we have mentioned the theory of Presburger arithmetic, a theory of the natural numbers with just the function *plus*. Presburger arithmetic is decidable. Unfortunately, it is intractable. [Fischer and Rabin 1974] showed that any algorithm that decides whether a sentence is a theorem of Presburger arithmetic must have time complexity at least $\mathcal{O}(2^{2^{cn}})$.

28.10 The Problem Classes FP and FNP ★

Recall that:

- A language L that corresponds to a decision problem Q is in P iff there is deterministic polynomial time algorithm that determines, given an arbitrary input x , whether $x \in L$.
- A language L that corresponds to a decision problem Q is in NP iff there is a deterministic polynomial time verifier that determines, given an arbitrary input x and a certificate c , whether c is a certificate for x . Equivalently, L is in NP iff there is a nondeterministic polynomial time algorithm that determines, given an arbitrary input x , whether there exists a certificate for x .

Now suppose that, instead of restricting our attention to decision problems, we wish to be able to characterize the complexity of functions whose result may of any type (for example, the integers). What we'll actually do is to go one step farther, and define the following complexity classes for arbitrary binary relations:

The Class FP : A binary relation Q is in FP iff there is deterministic polynomial time algorithm that, given an arbitrary input x , can find some y such that $(x, y) \in Q$.

The Class FNP : A binary relation Q is in FNP iff there is a deterministic polynomial time verifier that, given an arbitrary input pair (x, y) , determines whether $(x, y) \in Q$. Equivalently, Q is in FNP iff there is a nondeterministic polynomial time algorithm that, given an arbitrary input x , can find some y such that $(x, y) \in Q$.

FP is the functional/relational analog of P : if a relation Q is in FP then it is possible, in deterministic polynomial time, given a value x , to find a value y such that (x, y) is in Q . FNP is the functional/relational analog of NP : if a relation Q is in FNP then it is possible, in deterministic polynomial time, to determine whether a particular ordered pair (x, y) is in Q .

As before, checking all values is at least as hard as checking a single value. So we have:

$$FP \subseteq FNP.$$

But are they equal? The answer is that $FP = FNP$ iff $P = NP$.

In Section 28.5, we said that a language is NP-hard iff all other languages in NP are deterministic, polynomial time reducible to it. It is also common to apply the term “NP-hard” to functions. In this case, we’ll say that a function is *NP-hard* iff its corresponding decision problem is NP-hard. So, for example:

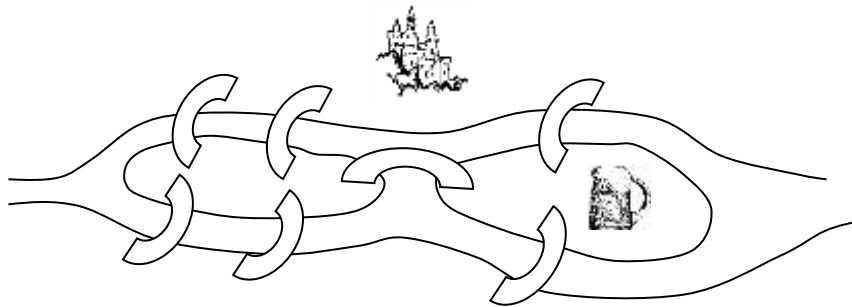
- The language TSP-DECIDE = $\{\langle G, cost \rangle : \langle G \rangle \text{ encodes an undirected graph with a positive distance attached to each of its edges and } G \text{ contains a Hamiltonian circuit whose total cost is less than } cost\}$ is NP-complete (and thus NP-hard). So the function that determines the cost of the lowest cost Hamiltonian circuit in G is NP-hard.
- Recall that the chromatic number of a graph is the smallest number of colors required to color its vertices, subject to the constraint that no two adjacent vertices may be assigned the same color. We defined the language CHROMATIC-NUMBER = $\{\langle G, k \rangle : G \text{ is an undirected graph whose chromatic number is no more than } k\}$. It is NP-complete. So the function that maps a graph to its chromatic number is NP-hard.

There are, however problems for which the decision version (i.e., a language to be decided) is easy, yet the function version remains hard. Probably the most important of these is the following:

- The language PRIMES = $\{w : w \text{ is the binary encoding of a prime number}\}$ is in P. But the problem of finding the factors of a composite number has no known polynomial time solution.

28.11 Exercises

- 1) Consider the following modification of the Seven Bridges of Königsberg problem that we described in Section 28.1.5:



The good prince lives in the castle. He wants to be able to return home from the pub (on one of the islands as shown above) and cross every bridge exactly once along the way. But he wants to make sure that his evil twin, who lives on the other river bank, is unable to cross every bridge exactly once on his way home from the pub. The good prince is willing to invest in building one new bridge in order to make his goal achievable. Where should he build his bridge?

- 2) Consider the language NONEULERIAN = $\{\langle G \rangle : G \text{ is an undirected graph and } G \text{ does not contain an Eulerian circuit}\}$.
 - a) Show an example of a connected graph with 8 vertices that is in NONEULERIAN.
 - b) Prove that NONEULERIAN is in P.
- 3) Show that each of the following languages is in P:
 - a) WWW = $\{www : w \in \{a, b\}^*\}$.
 - b) $\{\langle M, w \rangle : \text{Turing machine } M \text{ halts on } w \text{ within 3 steps}\}$.
 - c) EDGE-COVER = $\{\langle G, k \rangle : G \text{ is an undirected graph and there exists an edge cover of } G \text{ that contains at most } k \text{ edges}\}$.

- 4) In the proof of Theorem 33.2, we present the algorithm *3-conjunctiveBoolean*, which, given a Boolean wff, constructs a new wff w' , where w' is in 3-CNF.
- We claimed that w' is satisfiable iff w is. Prove that claim.
 - Prove that *3-conjunctiveBoolean* runs in polynomial time.
- 5) Consider the language $2\text{-SAT} = \{\langle w \rangle : w \text{ is a wff in Boolean logic, } w \text{ is in 2-conjunctive normal form and } w \text{ is satisfiable}\}$.
- Prove that 2-SAT is in P. (Hint: use resolution, as described in § 609.)
 - Why cannot your proof from part a) be extended to show that 3-SAT is in P?
 - Now consider a modification of 2-SAT that might, at first, seem even easier, since it may not require all of the clauses of w to be simultaneously satisfied. Let $2\text{-SAT-MAX} = \{\langle w, k \rangle : w \text{ is a wff in Boolean logic, } w \text{ is in 2-conjunctive normal form, } 1 \leq k \leq |C|, \text{ where } |C| \text{ is the number of clauses in } w, \text{ and there exists an assignment of values to the variables of } w \text{ that simultaneously satisfies at least } k \text{ of the clauses in } w\}$. Show that 2-SAT-MAX is NP-complete.
- 6) In Chapter 9, we showed that all of the questions that we posed about regular languages are decidable. We'll see, in Section 29.3.3, that while decidable, some straightforward questions about the regular languages appear to be hard. Some are easy however. Show that each of the following languages is in P:
- DFSM-ACCEPT = $\{\langle M, w \rangle : M \text{ is a DFSM and } w \in L(M)\}$.
 - FSM-EMPTY = $\{\langle M \rangle : M \text{ is a FSM and } L(M) = \emptyset\}$.
 - DFSM-ALL = $\{\langle M \rangle : M \text{ is a DFSM and } L(M) = \Sigma^*\}$.
- 7) We proved (in Theorem 28.1) that P is closed under complement. Prove that it is also closed under:
- Union.
 - Concatenation.
 - Kleene star.
- 8) It is not known whether NP is closed under complement. But prove that it is closed under:
- Union.
 - Concatenation.
 - Kleene star.
- 9) If L_1 and L_2 are in P and $L_1 \subseteq L \subseteq L_2$, must L be in P? Prove your answer.
- 10) Show that each of the following languages is NP-complete by first showing that it is in NP and then showing that it is NP-hard:
- CLIQUE = $\{\langle G, k \rangle : G \text{ is an undirected graph with vertices } V \text{ and edges } E, k \text{ is an integer, } 1 \leq k \leq |V|, \text{ and } G \text{ contains a } k\text{-clique}\}$.
 - SUBSET-SUM = $\{\langle S, k \rangle : S \text{ is a multiset (i.e., duplicates are allowed) of integers, } k \text{ is an integer, and there exists some subset of } S \text{ whose elements sum to } k\}$.
 - SET-PARTITION = $\{\langle S \rangle : S \text{ is a multiset (i.e., duplicates are allowed) of objects, each of which has an associated cost, and there exists a way to divide } S \text{ into two subsets, } A \text{ and } S - A, \text{ such that the sum of the costs of the elements in } A \text{ equals the sum of the costs of the elements in } S - A\}$.
 - KNAPSACK = $\{\langle S, v, c \rangle : S \text{ is a set of objects each of which has an associated cost and an associated value, } v \text{ and } c \text{ are integers, and there exists some way of choosing elements of } S \text{ (duplicates allowed) such that the total cost of the chosen objects is at most } c \text{ and their total value is at least } v\}$.
 - LONGEST-PATH = $\{\langle G, u, v, k \rangle : G \text{ is an unweighted, undirected graph, } u, \text{ and } v \text{ are vertices in } G, k \geq 0, \text{ and there exists a path with no repeated edges from } u \text{ to } v \text{ whose length is at least } k\}$.
 - BOUNDED-PCP = $\{\langle P, k \rangle : P \text{ is an instance of the Post Correspondence problem that has a solution of length less than or equal to } k\}$.

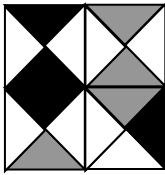
- 11) Let $USAT = \{ \langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ has exactly one satisfying assignment} \}$. Does the following nondeterministic, polynomial-time algorithm decide $USAT$? Explain your answer.

$decideUSAT(\langle w \rangle) =$

1. Nondeterministically select an assignment x of values to the variables in w .
 2. If x does not satisfy w , reject.
 3. Else nondeterministically select another assignment $y \neq x$.
 4. If y satisfies w , reject.
 5. Else accept.
- 12) Ordered binary decision diagrams (OBDDs) are useful in manipulating Boolean formulas such as the ones in the language SAT. They are described in \mathfrak{B} 612. Consider the Boolean function f_1 shown on page 612. Using the variable ordering $(x_3 < x_1 < x_2)$, build a decision tree for f . Show the (reduced) OBDD that *createOBDDfromtree* will create for that tree.
- 13) Complete the proof of Theorem 28.18 by showing how to modify the proof of Theorem 28.16 so that R constructs a formula in conjunctive normal form.
- 14) Show that, if $P = NP$, then there exists a deterministic, polynomial-time algorithm that finds a satisfying assignment for a Boolean formula if one exists.
- 15) Let R be the reduction from 3-SAT to VERTEX-COVER that we defined in the proof of Theorem 28.20. Show the graph that R builds when given the Boolean formula, $(\neg P \vee Q \vee T) \wedge (\neg P \vee Q \vee S) \wedge (T \vee \neg Q \vee S)$.
- 16) We'll say that an assignment of truth values to variables *almost satisfies* a conjunctive normal form (CNF) Boolean wff with k clauses iff it satisfies at least $k-1$ clauses. A CNF Boolean wff is *almost satisfiable* iff some assignment almost satisfies it. Show that the following language is NP-complete:
- $ALMOST-SAT = \{ \langle w \rangle : w \text{ is an almost satisfiable CNF Boolean formula} \}$.
- 17) Show that VERTEX-COVER is NP-complete by reduction from INDEPENDENT-SET.
- 18) In \mathfrak{C} 792, we describe the regular expression sublanguage in Perl. Show that regular expression matching in Perl (with variables allowed) is NP-hard.
- 19) In most route-planning problems the goal is to find the shortest route that meets some set of conditions. But consider the following problem (aptly named the taxicab rip-off problem in [Lewis and Papadimitriou 1998]): Given a directed graph G with positive costs attached to each edge, find the longest path from vertex i to vertex j that visits no vertex more than once.
- a) Convert this optimization problem to a language recognition problem.
 - b) Make the strongest statement you can about the complexity of the resulting language.
- 20) In Section 22.3, we introduced a family of tiling problems and defined the language TILES. In that discussion, we considered the question, "Given an infinite set T of tile designs and an infinite number of copies of each such design, is it possible to tile every finite surface in the plane?" As we saw, this unbounded version of the problem is undecidable. Now suppose that we are again given a set T of tile designs. But, this time, we are also given n^2 specific tiles drawn from that set. The question we now wish to answer is, "Given a particular stack of n^2 tiles, is it possible to tile an $n \times n$ surface in the plane?" As before, the rules are that tiles may not be rotated or flipped and the abutting regions of every pair of adjacent tiles must be the same color. So, for example, suppose that the tile set is:



Then a 2×2 grid can be tiled as:



- a) Formulate this problem as a language, FINITE-TILES.
- b) Show that FINITE-TILES is in NP.
- c) Show that FINITE-TILES is NP-complete (by showing that it is NP-hard).

21) In Section 28.7.6, we defined what we mean by a map coloring.

- a) Prove the claim, made there, that a map is 2-colorable iff it does not contain any point that is the junction of an odd number of regions. (Hint: use the pigeonhole principle.)
- b) Prove that THREE-COLORABLE = $\{ \langle m \rangle : m \text{ is a 3-colorable map} \}$ is in NP.
- c) Prove that THREE-COLORABLE = $\{ \langle m \rangle : m \text{ is a 3-colorable map} \}$ is NP-complete.

22) Define the following language:

- BIN-OVERSTUFFED = $\{ \langle S, c, k \rangle : S \text{ is a set of objects each of which has an associated size and it is not possible to divide the objects so that they fit into } k \text{ bins, each of which has size } c \}$.

Explain why it is generally believed that BIN-OVERSTUFFED is not NP-complete.

23) Let G be an undirected, weighted graph with vertices V , edges E , and a function $cost(e)$ that assigns a positive cost to each edge e in E . A **cut** of G is a subset S of the vertices in V . The cut divides the vertices in V into two subsets, S and $V - S$. Define the **size** of a cut to be the sum of the costs of all edges (u, v) such that one of u or v is in S and the other is not. We'll say that a cut is **nontrivial** iff it is neither \emptyset nor V . Recall that we saw, in Section 28.7.4, that finding shortest paths is easy (i.e., it can be done in polynomial time), but that finding longest paths is not. We'll observe a similar phenomenon with respect to cuts.

a) Sometimes we want to find the smallest cut in a graph. For example, it is possible to prove that the maximum flow between two nodes s and t is equal to the weight of the smallest cut that includes s but not t . Show that the following language is in P:

- MIN-CUT = $\{ \langle G, k \rangle : \text{there exists a nontrivial cut of } G \text{ with size at most } k \}$.

b) Sometimes we want to find the largest cut in a graph. Show that the following language is NP-complete:

- MAX-CUT = $\{ \langle G, k \rangle : \text{there exists a cut of } G \text{ with size at least } k \}$.

c) Sometimes, when we restrict the form of a problem we wish to consider, the problem becomes easier. So we might restrict the maximum-cut problem to graphs where all edge costs are 1. It turns out that, in this case, the "simpler" problem remains NP-complete. Show that the following language is NP-complete:

- SIMPLE-MAX-CUT = $\{ \langle G, k \rangle : \text{all edge costs in } G \text{ are 1 and there exists a cut of } G \text{ with size at least } k \}$.

d) Define the **bisection** of a graph G to be a cut where S contains exactly half of the vertices in V . Show that the following language is NP-complete: (Hint: the graph G does not have to be connected.)

- MAX-BISECTION = $\{ \langle G, k \rangle : G \text{ has a bisection of size at least } k \}$.

24) Show that each of the following functions is time-constructible:

- a) $n \log n$.
- b) $n\sqrt{n}$.
- c) n^3 .
- d) 2^n .
- e) $n!$.

25) In the proof of Theorem 28.27 (the Deterministic Time Hierarchy Theorem), we had to construct a string w of the form $\langle M_{t(n)\text{easy}} \rangle 1 0^p$. Let n be $|\langle M_{t(n)\text{easy}} \rangle 1 0^p|$. One of the constraints on our choice of p was that it be long enough that $|\langle M_{t(n)\text{easy}} \rangle| < \log(t(n)/\log t(n))$. Let m be $|\langle M_{t(n)\text{easy}} \rangle|$. Then we claimed that the condition would be satisfied if p is at least 2^{2^m} . Prove this claim.

26) Prove or disprove each of the following claims:

- a) If $A \leq_M B$ and $B \in P$, then $A \in P$.
- b) If $A \leq_P B$ and B and C are in NP, then $A \cup C \in NP$.
- c) Let $\text{NTIME}(f(n))$ be the set of languages that can be decided by some nondeterministic Turing machine in time $\mathcal{O}(f(n))$. Every language in $\text{NTIME}(2^n)$ is decidable.
- d) Define a language to be *co-finite* iff its complement is finite. Any co-finite language is in NP.
- e) Given an alphabet Σ , let A and B be nonempty proper subsets of Σ^* . If both A and B are in NP then $A \leq_M B$.
- f) Define the language:
 - $\text{MANY-CLAUSE-SAT} = \{\langle w \rangle : w \text{ is a Boolean wff in conjunctive normal form, } w \text{ has } m \text{ variables and } k \text{ clauses, and } k \geq 2^m\}$.

If $P \neq NP$, $\text{MANY-CLAUSE-SAT} \in P$.

29 Space Complexity Classes

In the last chapter, we analyzed problems with respect to the time required to decide them. In this chapter, we'll focus instead on space requirements.

29.1 Analyzing Space Complexity

Our analysis of space complexity begins with the function $spacereq(M)$ as described in Section 27.4.2:

- If M is a deterministic Turing machine that halts on all inputs, then the value of $spacereq(M)$ is the function $f(n)$ defined so that, for any natural number n , $f(n)$ is the maximum number of tape squares that M reads on any input of length n .
- If M is a nondeterministic Turing machine all of whose computational paths halt on all inputs, then the value of $spacereq(M)$ is the function $f(n)$ defined so that, for any natural number n , $f(n)$ is the maximum number of tape squares that M reads on any path that it executes on any input of length n .

So, just as $timereq(M)$ measures the worst-case time requirement of M as a function of the length of its input, $spacereq(M)$ measures the worst-case space requirement of M as a function of the length of its input.

29.1.1 Examples

To begin our discussion of space complexity, we'll return to three of the languages that we examined in the last chapter: CONNECTED, SAT, and TSP-DECIDE.

Example 29.1 CONNECTED

We begin by showing that $\text{CONNECTED} = \{ \langle G \rangle : G \text{ is an undirected graph and } G \text{ is connected} \}$ can be decided by a deterministic Turing machine that uses linear space. Recall that a graph is connected iff there exists a path from each vertex to each other vertex.

Theorem 28.4 tells us that CONNECTED is in P. The proof exploited an algorithm that we called *connected*. *Connected* works by starting at the first vertex and following edges, marking vertices as they are visited. If every vertex is eventually marked, then G is connected; otherwise it isn't. In addition to representing G , *connected* uses space for:

- Storing the marks on the vertices: This can be done by adding one extra bit to the representation of each vertex.
- Maintaining the list L of vertices that have been marked but whose successors have not yet been examined: We didn't describe how L is maintained. One easy way to do it is to add to the representation of each vertex one extra bit, which will be 1 if that vertex is a member of L and 0 otherwise.
- The number *marked-vertices-counter*: Since the value of the counter cannot exceed the number of vertices of G , it can be stored in binary in $\log(|\langle G \rangle|)$ bits.

So $spacereq(\text{connected})$ is $\mathcal{O}(|\langle G \rangle|)$.

CONNECTED is an "easy" language both from the perspective of time and the perspective of space since it can be decided in polynomial time and polynomial (in fact linear) space. Next we consider a language that appears to be harder if we measure time but is still easy if we measure only space.

Example 29.2 SAT

Consider $\text{SAT} = \{ \langle w \rangle : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable} \}$. SAT is in NP, so it can be decided in polynomial time by a nondeterministic Turing machine that, given a wff w , guesses at an assignment of values to its variables. Then it checks whether that assignment makes w *True*. The checking procedure (outlined in the proof of Theorem 28.12), requires no space beyond the space required to encode w . It can overwrite the variables of w with

their assigned values. Then it can evaluate subexpressions and replace each one with the value T or F . So SAT can be decided by a nondeterministic Turing machine that uses linear space.

SAT is believed not to be in P. No deterministic, polynomial-time algorithm is known for it. But it can be decided by a deterministic, polynomial-space algorithm that works as follows:

$decideSATdeterministically(\langle w \rangle) =$

1. Lexicographically enumerate the rows of the truth table for w . For each row do:
 - 1.1. Evaluate w (by replacing the variables with their values and applying the operators to those values, as described above).
 - 1.2. If w evaluates to *True*, accept.
2. If no row of the truth table caused w to be *True*, reject.

Each step of this procedure requires only linear space. But what about the space that may be required to control the loop? When we analyze algorithms to determine their space requirements, we must be careful to include whatever space is used by a loop index or, more significantly, a stack if one is used. For example, consider a recursive implementation of $decideSATdeterministically$ that, at each invocation, evaluates w if all variables have had values assigned. Otherwise, it picks an unassigned variable, assigns it a value, and then recurs. This implementation could require a stack whose depth equals the number of variables in w . Each stack entry would need a copy of w . Since the number of variables can grow linearly with w , we'd have that $space_{req}(decideSATdeterministically)$ is $\mathcal{O}(|\langle G \rangle|^2)$. That's polynomial, but not linear.

Fortunately, in the case of $decideSATdeterministically$, it is possible to control the loop using only an amount of space that is linear in the number of variables in w . Let 0 correspond to *False* and 1 correspond to *True*. Assign an order to the n variables in w . Then each row of w 's truth table is a binary string of length n . Begin by generating the string 0^n . At each step, use binary addition to increment the string by 1. Halt once the assignment that corresponds to 1^n has been evaluated.

Using this technique, we have that $space_{req}(decideSATdeterministically)$ is $\mathcal{O}(|\langle G \rangle|)$. So SAT can also be decided by a deterministic Turing machine that uses linear space.

Example 29.3 TSP-DECIDE

Consider $TSP-DECIDE = \{\langle G, cost \rangle : \langle G \rangle$ encodes an undirected graph with a positive distance attached to each of its edges and G contains a Hamiltonian circuit whose total cost is less than $cost\}$. We showed, in Theorem 28.10, that TSP-DECIDE is in NP. To prove the theorem, we described a nondeterministic Turing machine, $TSPdecide$, that decides the language TSP-DECIDE by nondeterministically attempting to construct a circuit one step at a time, checking, at each step, to see that the circuit's total cost is less than $cost$. $TSPdecide$ uses space to store the partial circuit and its cost. The length of any Hamiltonian circuit can't be longer than the list of edges in G , since no edge may appear twice. So the space required to store a partial circuit is a linear function of $|\langle G \rangle|$. The machine halts if the cost so far exceeds $cost$, so the space required to store the cost so far is bounded by $cost$.

Thus we have that $space_{req}(TSPdecide)$ is $\mathcal{O}(|\langle G \rangle|)$.

But $TSPdecide$ is nondeterministic. How much space would be required by a deterministic machine that decides TSP-DECIDE? We can define such a machine as follows:

$decideTSPdeterministically(\langle G, cost \rangle) =$

1. Set *circuit* to contain just vertex 1.
2. If $explore(G, 0, circuit)$ returns *True* then accept, else reject.

The bulk of the work is then done by the recursive procedure $explore$, which takes a partial circuit as input. It uses depth-first search to see whether it is possible to extend that circuit into one that is complete and whose cost is less than $cost$. Each call to $explore$ extends the circuit by one edge. $explore$ is defined as follows:

$explore(\langle G, cost, circuit \rangle) =$

1. If $circuit$ is complete and its cost is less than $cost$, return $True$.
2. If $circuit$ is complete and its cost is not less than $cost$, return $False$.
3. If $circuit$ is not complete then do: /* Try to extend it.
4. For each edge e that is incident on the last vertex of $circuit$, or until a return statement is executed, do:
 - 4.1. If the other vertex of e is not already part of $circuit$ or if it would complete $circuit$ then:
 - Call $explore(\langle G, cost + cost\ of\ e, circuit\ with\ e\ added \rangle)$.
 - If the value returned is $True$ then return $True$.
5. No alternative returned $True$. So return $False$.

$DecideTSPdeterministically$ works by recursively invoking $explore$. It needs space to store the stack that holds the individual invocations of $explore$, including their arguments. Some paths may end without considering a complete circuit, but the maximum depth of the stack is $|V| + 1$, since that is the number of vertices in any complete circuit. Each stack record needs space to store a cost and a complete circuit, whose length is $|V| + 1$.

So we have that $spacereq(TSPdecidedeterministically)$ is $\mathcal{O}(|G|^2)$. We can actually do better and decide TSP-DECIDE using only linear space by storing, at each invocation of $explore$, just a cost and the one new vertex that is added at that step. Thus, while we know of no deterministic Turing machine that can decide TSP-DECIDE in polynomial time, there does exist one that can decide it in polynomial space.

29.1.2 Relating Time and Space Complexity

The examples that we have just considered suggest that there is some relationship between the number of steps a Turing machine executes and the amount of space it uses. The most fundamental relationship between the two numbers arises from the fact that, at each step of its operation, a Turing machine can examine at most one tape square. So we have, for any Turing machine M , that $spacereq(M) \leq timereq(M)$.

But M 's time requirement cannot be arbitrarily larger than its space requirement. We are considering only Turing machines that halt on all inputs. If a Turing machine M halts, then it can never re-enter a configuration that it has been in before. (If it did, it would be in an infinite loop.) So the number of steps that M can execute is bounded by the number of distinct configurations that it can enter. We can compute the maximum number of such configurations as follows: Let K be the states of M and let Γ be its tape alphabet. M may be in any one of its $|K|$ states. Define M 's active tape to be the smallest tape fragment that contains all the nonblank symbols plus the read/write head. Assuming that $spacereq(M) \geq n$ (i.e., that M actually reads all its input), the number of squares in M 's active tape at any point during M 's computation is bounded by $spacereq(M)$. Each of those squares may hold any one of the $|\Gamma|$ tape symbols. So the maximum number of distinct tape snapshots is $|\Gamma|^{spacereq(M)}$. And M 's read/write head may be on any one of the $spacereq(M)$ tape squares. So the maximum number of distinct configurations that M can enter is:

$$MaxConfigs(M) = |K| \cdot |\Gamma|^{spacereq(M)} \cdot spacereq(M).$$

Let c be a constant such that $c > |\Gamma|$. Then:

$$MaxConfigs(M) \in \mathcal{O}(c^{spacereq(M)}).$$

(We leave the proof of this claim as an exercise.). Using the analysis we have just presented, we can prove the following theorem:

Theorem 29.1 Relating Time and Space Requirements

Theorem: Given a Turing machine $M = (K, \Sigma, \Gamma, \delta, s, H)$ and assuming that $spacereq(M) \geq n$, the following relationships hold between M 's time and space requirements:

$$spacereq(M) \leq timereq(M) \in \mathcal{O}(c^{spacereq(M)}).$$

Proof: $spacereq(M)$ is bounded by $timereq(M)$ since M must use at least one time step for every tape square it visits.

The upper bound on $timereq(M)$ follows from the fact, since M halts, the number of steps that it can execute is bounded by the number of distinct configurations that it can enter. That number is given by the function $MaxConfigs(M)$, as described above. Since $MaxConfigs(M) \in \mathcal{O}(c^{spacereq(M)})$, $timereq(M) \in \mathcal{O}(c^{spacereq(M)})$. ■

In a nutshell, space can be reused. Time cannot.

29.2 PSPACE, NPSPACE, and Savitch's Theorem

If our measure of complexity is time, it appears that nondeterminism adds power. So, for example, there are languages, such as SAT and TSP-DECIDE, that are in NP but that do not appear to be in P. When we change perspectives and measure complexity in terms of space requirements, the distinction between nondeterministic and deterministic machines turns out almost to disappear.

Recall that we defined the language class P to include exactly those languages that could be decided by a *deterministic* Turing machine in polynomial time. And we defined the class NP to include exactly those languages that could be decided by a *nondeterministic* Turing machine in polynomial time. We'll now define parallel classes based on space requirements:

The Class PSPACE: $L \in \text{PSPACE}$ iff there exists some deterministic Turing machine M that decides L and $spacereq(M) \in \mathcal{O}(n^k)$ for some constant k .

The Class NPSPACE: $L \in \text{NPSPACE}$ iff there exists some nondeterministic Turing machine M that decides L and $spacereq(M) \in \mathcal{O}(n^k)$ for some constant k .

Savitch's Theorem, which we'll state and prove next, tells us that we needn't have bothered to define the two classes. $\text{PSPACE} = \text{NPSPACE}$.

Note that, since every deterministic Turing machine is also a legal nondeterministic one, if a language L can be decided by some deterministic Turing machine that requires $f(n)$ space, then it can also be decided by some nondeterministic Turing machine that requires at most $f(n)$ space. The other direction does not follow. It may be possible to decide L with a nondeterministic Turing machine that uses just $f(n)$ space but there may exist no deterministic machine that can do it without using more than $\mathcal{O}(f(n))$ space. However, it turns out that L 's deterministic space complexity cannot be *much* worse than its nondeterministic space complexity. We are about to prove that, assuming one common condition is satisfied, there must exist a deterministic Turing machine that decides it using $\mathcal{O}(f(n)^2)$ space.

The proof that we will do is by construction. We'll show how to transform a nondeterministic Turing machine into an equivalent deterministic one that conducts a systematic search through the set of "guesses" that the nondeterministic machine could have made. That's exactly what we did for TSP-DECIDE above. In that case, we were able to construct a deterministic Turing machine that conducted a straightforward depth-first search and that required only $\mathcal{O}(n^2)$ space to store its stack. But we exploited a specific property of TSP-DECIDE to make that work: we knew that any Hamiltonian circuit through a graph with $|V|$ vertices must have exactly $|V|$ edges. So the depth of the stack was bounded by $|V|$ and thus by $|G|$.

In general, while there is a bound on the depth of the stack, it is much weaker. We can guarantee only that, if a nondeterministic Turing machine M uses $spacereq(M)$ space, then any one branch of a depth-first deterministic Turing machine that simulates M must halt in no more than $MaxConfigs(M)$ steps (since otherwise it is in a loop). But $MaxConfigs(M) \in \mathcal{O}(c^{spacereq(M)})$. We can't afford a stack that could grow that deep. There is, however, an alternative to depth-first search that can be guaranteed to require a stack whose depth is $\mathcal{O}(n)$. We'll use it in the proof of Savitch's Theorem, which we state next.

Theorem 29.2 Savitch's Theorem

Theorem: If L can be decided by some nondeterministic Turing machine M and $spacereq(M) \geq n$, then there exists a deterministic Turing machine M' that also decides L and $spacereq(M') \in \mathcal{O}(spacereq(M)^2)$.

Proof: We require that $spacereq(M) \geq n$, which means just that M must at least be able to read all its input. In Section 29.4, we'll introduce a way to talk about machines that use less than linear *working space*. Once we do that, this constraint can be weakened to $spacereq(M) \geq \log n$.

The proof is by construction. Suppose that L is decided by some nondeterministic Turing machine M . We will show an algorithm that builds a deterministic Turing machine M' that also decides L and that uses space that grows no faster than the square of the space required by M . M' will systematically examine the paths that M could pursue. Since our goal is to put a bound on the amount of space that M' uses, the key to its construction is a technique for guaranteeing that the stack that it uses to manage its search doesn't get "too deep". We'll use a divide-and-conquer technique in which we chop each problem into two halves in such a way that we can solve the first half and then reuse the same space to solve the second half.

To simplify the question that we must answer, we'll begin by changing M so that, whenever it is about to accept, it first blanks out its tape. Then it enters a new, unique, accepting state. Call this new machine M_{blank} . Note that M_{blank} accepts iff it ever reaches the configuration in which its tape is blank and it is in the new accepting state. Call this configuration c_{accept} . M_{blank} uses no additional space, so $spacereq(M_{blank}) = spacereq(M)$.

Now we must describe the construction of M' , which, on input w , must accept iff M_{blank} , on input w , can (via at least one of its computational paths) reach c_{accept} . Because we need to bound the depth of the stack that M' uses, we need to bound the number of steps it can execute (since it might have to make a choice at each step). We have already seen that simple approaches, such as depth-first search, cannot do that adequately. So we'll make use of the following function, *canreach*. Its job is to answer the more general question, "Given a Turing machine T running on input w , two configurations, c_1 and c_2 , and a number t , could T , if it managed to get to c_1 , go on and reach c_2 within t steps?" *Canreach* works by exploiting the following observation: if T can go from c_1 to c_2 within t steps, then at least one of the following must be true:

1. $t = 0$. In this case, $c_1 = c_2$.
2. $t = 1$. In this case, $c_1 \mid\text{-}_T c_2$. (Recall that $\mid\text{-}_T$ is the *yields-in-one-step* relation between configurations of machine T .) Whether the single required step exists can be determined just by examining the transitions of T .
3. $t > 1$. In this case, $c_1 \mid\text{-}_T \dots \mid\text{-}_T c_k \mid\text{-}_T \dots \mid\text{-}_T c_2$. In other words, there is some (at least one) configuration c_k that T goes through on the way from c_1 to c_2 . Furthermore, note that, however many configurations there are on the path from c_1 to c_2 , there is a "middle" one, i.e., one with the property that half of T 's work is done getting from c_1 to it and the other half is done getting from it to c_2 . (It won't matter that, if the length of the computation is not an even number, there may be one more configuration on one side of the middle one than there is on the other.)

So *canreach* operates as follows: If $t = 0$, all it needs to do is to check whether $c_1 = c_2$. If $t = 1$, it just checks whether the one required transition exists in T . If $t > 1$, then it considers, as a possible "middle" configuration, all configurations of T that use no more space than $spacereq(T)$ allows for inputs of length $|w|$. It will recursively invoke itself and ask whether T could both go from c_1 to *middle* in $t/2$ steps and from *middle* to c_2 in the remaining $t/2$ steps. (Since $t/2$ may not be an integer, we'll give each invocation $\lceil t/2 \rceil$ steps, where $\lceil t/2 \rceil$ is the ceiling of $t/2$, i.e., the smallest integer that is greater than or equal to $t/2$.) For this approach to work, we must be able to guarantee that there is only a finite number of configurations that T could enter while processing w . We are only going to invoke *canreach* on deciding Turing machines, so we know not only that the number of such configurations is finite but that it is bounded by $MaxConfigs(T)$, which is a function of the number of states in T , the size of its tape alphabet, and $spacereq(T)$, which bounds the number of tape squares that can be nonblank as a function of the length of w .

Canreach will take five arguments: a Turing machine T , an input string w , a pair of configurations, c_1 and c_2 , and a nonnegative integer that corresponds to the number of time steps that T may use in attempting to get from c_1 to c_2 . Note that T and w won't change as *canreach* recursively invokes itself. Also note that the only role w plays is that its length determines the number of tape squares that can be used. *Canreach* can be defined as follows:

- canreach*(T : Turing machine, w : string, c_1 : configuration, c_2 : configuration, t : nonnegative integer) =
1. If $c_1 = c_2$ then return *True*.
 2. If $t = 1$ then:

- 2.1. If $c_1 \vdash_T c_2$ then return *True*. /* c_1 yields c_2 in one step.
- 2.2. Else return *False*. /* In one step, c_1 cannot yield c_2 .
3. If $t > 1$, then let *Confs* be the set of all of T 's configurations whose tape is no longer than $\text{spacereq}(T)$ applied to $|w|$. For each configuration *middle* in *Confs* do:
 - 3.1. If $\text{canreach}(T, w, c_1, \text{middle}, \lceil t/2 \rceil)$ and $\text{canreach}(T, w, \text{middle}, c_2, \lceil t/2 \rceil)$ then return *True*.
4. Return *False*. /* None of the possible *middles* worked.

We can now return to our original problem: given a nondeterministic Turing machine M , construct a deterministic Turing machine M' such that $L(M') = L(M)$ and $\text{spacereq}(M') \in \mathcal{O}(\text{spacereq}(M)^2)$. The following algorithm solves the problem:

builddet(M : nondeterministic Turing machine) =

1. From M , build M_{blank} as described above.
2. From M_{blank} , build M' . To make it easy to describe M' , define:
 - c_{start} to be the start configuration of M_{blank} on input w .
 - $\text{max-on-}w$ to be the result of applying the function $\text{maxConfs}(M_{\text{blank}})$ to $|w|$. (So $\text{max-on-}w$ is the maximum number of distinct configurations that M_{blank} might enter when started on w . Thus it is also the maximum number of steps that M_{blank} might execute on input w , given that it eventually halts.)

Then $M'(w)$ operates as follows:

If $\text{canreach}(M_{\text{blank}}, w, c_{\text{start}}, c_{\text{accept}}, \text{max-on-}w)$ then accept, else reject.

Canreach will return *True* iff M_{blank} (and thus M) accepts w . So $L(M') = L(M)$. But it remains to show that $\text{spacereq}(M') \in \mathcal{O}(\text{spacereq}(M)^2)$.

Each invocation of *canreach* requires storing an activation record. It suffices to store M and w once. But each record must contain a new copy of two configurations and the integer that puts a bound on the number of steps to be executed. Each configuration requires $\mathcal{O}(\text{spacereq}(M))$ space, so each invocation record also requires $\mathcal{O}(\text{spacereq}(M))$ space. Now all we need to do is to determine the depth of the stack of invocation records. Notice that each invocation of *canreach* cuts the allotted number of steps in half. So the depth of the stack is bounded by $\log_2(\text{max-on-}w)$. But $\text{max-on-}w$ is $\text{maxConfs}(M)$ applied to $|w|$ and we have that:

$$\begin{aligned} \text{MaxConfs}(M) &\in \mathcal{O}(c^{\text{spacereq}(M)}). \\ \log_2(\text{MaxConfs}(M)) &\in \mathcal{O}(\text{spacereq}(M)). \end{aligned}$$

So the depth of the stack is $\mathcal{O}(\text{spacereq}(M))$ and the total space required is $\mathcal{O}(\text{spacereq}(M)^2)$. ■

Savitch's Theorem has an important corollary, which we state next:

Theorem 29.3 PSPACE = NPSPACE

Theorem: PSPACE = NPSPACE.

Proof: In one direction, the claim is trivial: If L is in PSPACE, then it must also be in NPSPACE because the deterministic Turing machine that decides it in polynomial time is also a nondeterministic Turing machine that decides it in polynomial time.

To prove the other direction, we note that Savitch's Theorem tells us that the price for going from a nondeterministic machine to a deterministic one is at most a squaring of the amount of space required. More precisely, if L is in NPSPACE then there is some nondeterministic Turing machine M such that M decides L and $\text{spacereq}(M) \in \mathcal{O}(n^k)$ for some k . If $k \geq 1$, then, by Savitch's Theorem, there exists a deterministic Turing machine M' such that M' decides L and $\text{spacereq}(M') \in \mathcal{O}(n^{2k})$. If, on the other hand, $k < 1$ then, using the same construction that we used in the proof

of Savitch's Theorem, we can show that there exists an M' such that M' decides L and $space_{req}(M') \in \mathcal{O}(n^2)$. In either case, $space_{req}(M')$ is a polynomial function of n . So L can be decided by a deterministic Turing machine whose space requirement is some polynomial function of the length of its input. Thus L is in PSPACE. ■

Another corollary of Savitch's Theorem is the following:

Theorem 29.4 $P \subseteq NP \subseteq PSPACE$

Theorem: $P \subseteq NP \subseteq PSPACE$.

Proof: Theorem 28.14 tells us that $P \subseteq NP$. It remains to show that $NP \subseteq PSPACE$. If a language L is in NP, then it is decided by some nondeterministic Turing machine M in polynomial time. In polynomial time, M cannot use more than polynomial space since it takes a least one time step to visit a tape square. Since M is a nondeterministic Turing machine that decides L in polynomial space, L is in NPSPACE. But, by Savitch's Theorem, $PSPACE = NPSPACE$. So L is also in PSPACE. ■

It is assumed that both subset relationships are proper (i.e., that $P \neq NP \neq PSPACE$), but no proof of either of those claims exists.

29.3 PSPACE-Completeness

Recall that, in our discussion of space complexity, we introduced two useful language families: we said that a language is NP-hard iff every language in NP is deterministic, polynomial-time reducible to it. And we said that a language is NP-complete iff it is NP-hard and it is also in NP. All NP-complete languages are equivalently hard in the sense that all of them can be decided in nondeterministic, polynomial time and, if any one of them can also be decided in deterministic polynomial time, then all of them can.

In our attempt to understand why some problems appear harder than others, it is useful to define corresponding classes based on space complexity. So we consider the following two properties that a language L might possess:

1. L is in PSPACE.
2. Every language in PSPACE is deterministic, polynomial-time reducible to L .

Using those properties, we will define:

The Class PSPACE-hard: L is PSPACE-hard iff it possesses property 2.

The Class PSPACE-complete: L is PSPACE-complete iff it possesses *both* property 1 and property 2. All PSPACE-complete languages can be viewed as being equivalently hard in the sense that all of them can be decided in polynomial space and:

- If any PSPACE-complete language is also in NP, then all of them are and $NP = PSPACE$.
- If any PSPACE-complete language is also in P, then all of them are and $P = NP = PSPACE$.

Note that we have defined PSPACE-hardness, just as we defined NP-hardness, with respect to polynomial-time reducibility. We could have defined it in terms of the space complexity of the reductions that we use. But the polynomial-time definition is more useful because it provides a stronger notion of a "computationally feasible" reduction. If all we knew about two languages L_1 and L_2 were that L_1 were polynomial-space reducible to L_2 , an efficient (i.e., polynomial-time) solution to L_2 would not guarantee an efficient solution to L_1 . The efficiency of the solution for L_2 might be swamped by a very inefficient reduction from L_1 to L_2 . By continuing to restrict our attention to deterministic, polynomial-time reductions, we guarantee that if L_1 is reducible to L_2 and an efficient solution to L_2 were to be found, we would also have an efficient solution for L_1 .

When we began our discussion of NP-completeness, we faced a serious problem at the outset: how could we find our first NP-complete language? Once we had that one, we could prove that other languages were also NP-complete by reduction from it. We face the same problem now as we begin to explore the class of PSPACE-complete languages. We need a first one.

Recall that, in the case of NP-completeness, the language that got us going and that provided the basis for the proof of the Cook-Levin Theorem, was SAT (the language of satisfiable Boolean formulas). The choice of SAT was not arbitrary. To prove that it was NP-complete, we exploited the expressive power of Boolean logic to describe computational paths. Since every NP language is, by definition, decided by some nondeterministic Turing machine each of whose paths halts in a finite (and polynomially-bounded) number of steps, we were able to define a reduction from an arbitrary NP language L to the specific NP language SAT by showing a way to build, given a deciding machine M for L and a string w , a Boolean formula whose length is bounded by a polynomial function of the length of w and that is satisfiable iff M accepts w .

Perhaps we can, similarly, seed the class of PSPACE-complete languages with a logical language. Because we believe that PSPACE includes languages that are not in NP, we wouldn't expect SAT to work. On the other hand, we can't jump all the way to a first-order logic language like $\text{FOL}_{\text{theorem}} = \{ \langle A, w \rangle : A \text{ is a decidable set of axioms in first-order logic, } w \text{ is a sentence in first-order logic, and } w \text{ is entailed by } A \}$, since it isn't decidable at all, much less is it decidable in polynomial space. In the next section, we will define a new language, QBF, that adds quantifiers to the language of Boolean logic but that stops short of the full power of first-order logic. Then we will show that QBF is PSPACE-complete. We'll do that using a construction that is similar to the one that was used to prove the Cook-Levin Theorem. We'll discover one wrinkle, however: in order to guarantee that, on input w , the length of the formula that we build is bounded by some polynomial function of w , we will need to use the divide-and-conquer technique that we exploited in the proof of Savitch's Theorem.

29.3.1 The Language QBF

Boolean formulas are evaluated with respect to the universe, $\{True, False\}$. A particular Boolean well-formed formula (wff), such as $((P \wedge Q) \vee \neg R) \rightarrow S$, is a function, stated in terms of some finite number of Boolean variables. Given a particular set of values as its input, it returns either *True* or *False*. We have defined Boolean-formula languages, like SAT, in terms of properties that the formulas that are in the language must possess. So, for example:

- A wff $w \in \text{SAT}$ iff it is satisfiable. In other words, $w \in \text{SAT}$ iff *there exists some set of values* for the variables of w such that w evaluates to *True*.
- A wff $w \in \text{VALID}$ iff it is a tautology. In other words, $w \in \text{VALID}$ iff *for all values* for the variables of w , w evaluates to *True*.

So, while Boolean formulas do not contain quantifiers, we have used quantification in our descriptions of Boolean-formula languages. Now suppose that we add explicit quantifiers to the logical language itself.

Define the language of *quantified Boolean expressions* as follows:

- The base case: all wffs are quantified Boolean expressions.
- Adding quantifiers: if w is a quantified Boolean expression that contains the unbound variable A , then the expressions $\exists A (w)$ and $\forall A (w)$ are quantified Boolean expressions. Exactly as we do in first-order logic, we'll then say that A is bound in w and that the scope of the new quantifier is w .

All of the following are quantified Boolean expressions:

- $(P \wedge \neg R) \rightarrow S$.
- $\exists P ((P \wedge \neg R) \rightarrow S)$.
- $\forall R (\exists P ((P \wedge \neg R) \rightarrow S))$.
- $\forall S (\forall R (\exists P ((P \wedge \neg R) \rightarrow S)))$.

Notice that, because of the way they are constructed, all quantified Boolean expressions are in prenex normal form, as defined in § 616. In other words, the expression is composed of a quantifier list followed by a quantifier-free matrix. We'll find this form useful below.

As in first-order logic, we'll say that a quantified Boolean expression is a *sentence* iff all of its variables are bound. So, for example $\forall S (\forall R (\exists P ((P \wedge \neg R) \rightarrow S)))$ is a sentence, but none of the other expression listed above is.

A *quantified Boolean formula* is a quantified Boolean expression that is also a sentence. Every quantified Boolean formula, just like every sentence in first-order logic, can be evaluated to produce either *True* or *False*. For example:

- $\exists P (\exists R (P \wedge \neg R))$ evaluates to *True*.
- $\exists P (\forall R (P \wedge \neg R))$ evaluates to *False*.

We can now define the language that will turn out to be our first PSPACE-complete language:

- $\text{QBF} = \{ \langle w \rangle : w \text{ is a true quantified Boolean formula} \}$.

29.3.2 QBF is PSPACE-Complete

QBF, unlike languages like $\text{FOL}_{\text{theorem}}$ that are defined with respect to full first-order logic, is decidable. The reason is that the universe with respect to which existential and universal quantification are defined is finite. In general, we cannot determine the validity of an arbitrary first-order logic formula, such as $\forall x (P(x))$, by actually evaluating P for all possible values of x . The domain of x might, for example, be the integers. But it is possible to decide whether an arbitrary quantified Boolean formula is true by exhaustively examining its (finite) truth table.

We'll show next that not only is it possible to decide QBF, it is possible to decide it in polynomial (in fact linear) space.

Theorem 29.5 QBF is in PSPACE

Theorem: $\text{QBF} = \{ \langle w \rangle : w \text{ is a true quantified Boolean formula} \}$ is in PSPACE.

Proof: We show that QBF is in PSPACE by exhibiting a deterministic, polynomial-space algorithm that decides it. The algorithm that we present exploits the fact that quantified Boolean formulas are in prenex normal form. So, if w is a quantified Boolean formula, it must have the following form, where each Q is a quantifier (either \forall or \exists) and f is a Boolean wff:

$$Q_{v_1} (Q_{v_2} (Q_{v_3} \dots (Q_{v_n} (f) \dots)).$$

The following procedure, *QBFdecide*, decides whether w is true. It peels off the quantifiers one at a time, left-to-right. Each time it peels off a quantifier that binds some variable v , it substitutes *True* for every instance of v and calls itself recursively. Then it substitutes *False* for every instance of v and calls itself recursively again. At some point, it will be called with a Boolean wff that contains only constants. When that happens, the wff can simply be evaluated.

QBFdecide($\langle w \rangle$) =

1. Invoke *QBFcheck*($\langle w \rangle$).
2. If it returns *True*, accept; else reject.

QBFcheck($\langle w \rangle$) =

1. If w contains no quantifiers, evaluate it by applying its Boolean operators to its constant values. The result will be either *True* or *False*. Return it.
2. If w is $\forall v (w')$, where w' is some quantified Boolean formula, then:
 - 2.1. Substitute *True* for every occurrence of v in w' and invoke *QBFcheck* on the result.
 - 2.2. Substitute *False* for every occurrence of v in w' and invoke *QBFcheck* on the result.
 - 2.3. If both of these branches accept, then w' is true for all values of v . So accept; else reject.
3. If w is $\exists v (w')$, where w' is some quantified Boolean formula, then:

- 3.1. Substitute *True* for every occurrence of v in w' and invoke *QBFcheck* on the result.
- 3.2. Substitute *False* for every occurrence of v in w' and invoke *QBFcheck* on the result.
- 3.3. If at least one of these branches accepts, then w' is true for some value of v . So accept; else reject.

We analyze the space requirement of *QBFdecide* as follows: The depth of *QBFcheck*'s stack is equal to the number of variables in w , which is $\mathcal{O}(|w|)$. At each recursive call, the only new information is the value of one new variable. So the amount of space for each stack entry is constant. The actual evaluation of a variable-free wff w can be done in $\mathcal{O}(|w|)$ space. Thus the total space used by *QBFdecide* is $\mathcal{O}(|w|)$. *QBFdecide* runs in linear (and thus obviously polynomial) space. ■

We can't prove that a more efficient algorithm for deciding QBF doesn't exist. But the result that we will prove next strongly suggest that none does. In particular, it tells us that a nondeterministic, polynomial-time algorithm exists only if $\text{NP} = \text{PSPACE}$ and a deterministic, polynomial-time algorithm exists only if $\text{P} = \text{NP} = \text{PSPACE}$.

Theorem 29.6 QBF is PSPACE-Complete

Theorem: $\text{QBF} = \{ \langle w \rangle : w \text{ is a true quantified Boolean formula} \}$ is PSPACE-complete.

Proof: We have already shown that QBF is in PSPACE. So all that remains is to show that it is PSPACE-hard. We'll do that by showing a polynomial-time reduction to it from any language in PSPACE. We'll use approximately the same technique that we used, in the proof of the Cook-Levin Theorem, where we showed that SAT is NP-hard. Let L be any language in PSPACE. L is decided by some deterministic Turing machine M with the property that $\text{spacereq}(M)$ is a polynomial. We'll describe a reduction from L to QBF that works by constructing a quantified Boolean formula that describes the computation of M on input w and that is true iff M accepts w .

Just as we did in the proof of the Cook-Levin Theorem, we'll use Boolean variables to describe each of the configurations that M enters while processing w . Our first idea might be simply to construct a Boolean formula exactly as we described in the Cook-Levin Theorem proof. Then we can convert that formula into a quantified Boolean formula by binding each of its variables by an existential quantifier. The resulting quantified Boolean formula will be true iff the original formula is satisfiable.

It remains to analyze the time complexity of the construction. The number of steps required by the construction is a polynomial function of the length of the formula that it constructs. The length of the formula is polynomial in the number of cells in the table that describes the computation of M on w . Each row of the table corresponds to one configuration of M , so the number of cells in a row is $\mathcal{O}(\text{spacereq}(M))$, which is polynomial in $|w|$.

But now we have a problem. In the proof of the Cook-Levin Theorem, we knew that the maximum length of any computational path of M was $\mathcal{O}(|w|^k)$. So the maximum number of configurations that would have to be described, and thus the number of rows in the table, was also $\mathcal{O}(|w|^k)$. The problem that we now face is that we no longer have a polynomial bound on the number of configurations that M may enter before it halts. All we have is a polynomial bound on the amount of space M uses. Using that space bound, we can construct a time bound, as we've done before, by taking advantage of the fact that M may not enter a loop. So the maximum number of steps it may execute is bounded by the maximum number of distinct configurations it may enter. That number is

$$\text{MaxConfigs}(M) = |K| \cdot |\Gamma|^{\text{spacereq}(M)} \cdot \text{spacereq}(M) \in \mathcal{O}(c^{\text{spacereq}(M)}).$$

So, if we used exactly the same technique we used in the proof of the Cook-Levin Theorem, we'd be forced to describe the computation of M on w with a formula whose length grows exponentially with $|w|$. A polynomial-time reduction cannot build an exponentially long formula.

The solution to this problem is to exploit quantifiers to "cluster" subexpressions so that a whole group of them can be described at once.

To do this, we'll begin by returning to the divide-and-conquer technique that we used in our proof of Savitch's Theorem. As we did there, we will again solve a more general problem. This time, we will describe a technique for constructing a quantified Boolean formula $f(c_1, c_2, t)$ that is true iff M can get from configuration c_1 to configuration c_2 in at most t steps. We again observe that, if M can get from configuration c_1 to configuration c_2 within t steps, then at least one of the following must be true:

1. $t = 0$. In this case, $c_1 = c_2$. Since each configuration can be described by a formula whose length is polynomial in $|w|$, this condition can also be described by such a formula.
2. $t = 1$. In this case, c_1 yields c_2 in a single step. Using the techniques we used to build $Conjunct_4$ in the proof of the Cook-Levin Theorem, this condition can also be described by a formula whose length is polynomial in $|w|$. Note that, in the proof of the Cook-Levin Theorem, we built a Boolean formula and then asked whether it was satisfiable. Now we build the same Boolean formula and then bind all the variables with existential quantifiers, so we again ask whether any values satisfy the formula.
3. $t > 1$. In this case, c_1 yields c_2 in more than one step. Then there is some configuration we'll call *middle* with the property that M can get from c_1 to *middle* within $\lceil t/2 \rceil$ steps and from *middle* to c_2 within another $\lceil t/2 \rceil$ steps. Of course, as in the proof of Savitch's Theorem, we don't know what *middle* is. But, when we build $f(c_1, c_2, t)$, we can use an existential quantifier to assert that it exists. The resulting formula will only be true if, in fact, *middle* does exist.

Now we just need a space-efficient way to represent $f(c_1, c_2, t)$ in the third case. Suppose that *middle* exists. Then some set of Boolean variables m_1, m_2, \dots describe it. So we could begin by writing:

$$f(c_1, c_2, t) = \exists m_1 (\exists m_2 \dots (f(c_1, middle, \lceil t/2 \rceil) \wedge f(middle, c_2, \lceil t/2 \rceil)) \dots).$$

We can simplify this by introducing the following shorthand:

- If c is a configuration that is described by the variables c_1, c_2, \dots , let $\exists c(p)$ stand for $\exists c_1 (\exists c_2 \dots (p) \dots)$ and let $\forall c(p)$ stand for $\forall c_1 (\forall c_2 \dots (p) \dots)$. Note that since the number of variables required to describe c is polynomial in $|w|$, the length of the expanded formula is a polynomial function of the length of the shorthand one.

This lets us rewrite our definition of f as:

$$f(c_1, c_2, t) = \exists middle (f(c_1, middle, \lceil t/2 \rceil) \wedge f(middle, c_2, \lceil t/2 \rceil)).$$

Then we could recursively expand $f(c_1, middle, \lceil t/2 \rceil)$ and $f(middle, c_2, \lceil t/2 \rceil)$, continuing until $\lceil t/2 \rceil$ becomes 0 or 1. We cut the number of computation steps that might have to be described in half each time we do this recursion. But we also replace a single formula by the conjunction of two formulas. So the total length of the formula that we'll build, if we take this approach, is $\mathcal{O}(t)$.

Unfortunately, it becomes obvious that this approach isn't efficient enough as soon as we return to the original, specific problem of describing the computation of M on w . As we did in the proof of Savitch's Theorem, we'll actually work with M_{blank} , a modification of M that accepts by entering a unique accepting configuration that we'll call c_{accept} . Let c_{start} be the starting configuration of M on w . We know that the number of steps that M may execute in getting from c_{start} to c_{accept} is $\mathcal{O}(c^{space_{req}(M)})$ and that $space_{req}(M)$ is polynomial in $|w|$. The formula that we must build is then:

$$f(c_{start}, c_{accept}, 2^{k \cdot space_{req}(M)}).$$

So its length grows exponentially with $|w|$.

To reduce its size, we'll exploit universal quantifiers to enable us to describe the two recursively generated subformulas of $f(c_1, c_2, t)$ as a single formula. To do this, we need to create a new, generic formula that describes the transition, within $\lceil t/2 \rceil$ steps, from an arbitrary configuration we'll call c_3 to another arbitrary configuration we'll call c_4 . The names don't matter as long as we describe the two configurations with variables that are distinct from all the variables that we will use to describe actual configurations of M . So the new formula is $f(c_3, c_4, \lceil t/2 \rceil)$. Then we'll want to say that the new formula must be true both when:

- $c_3 = c_1$ and $c_4 = middle$, and
- $c_3 = middle$ and $c_4 = c_2$.

We'll do that by saying that it must be true for all (i.e., both) of those assignments of values to the variables of c_3 and c_4 . To do that, we need the following additional shorthands:

- Let $\forall(x, y) (p)$ stand for $\forall x (\forall y (p))$.
- Let $\forall x \in \{s, t\} (p)$ stand for $\forall x ((x = s \vee x = t) \rightarrow p)$.
- Combining these, let $\forall(x, y) \in \{(s_1, s_2), (t_1, t_2)\} (p)$ say that $((x = s_1 \wedge y = s_2) \rightarrow p) \wedge ((x = t_1 \wedge y = t_2) \rightarrow p)$.

Note that the length of the expanded versions of one of these shorthands grows at most polynomially in the length of the shortened form. With these conventions in hand, we can now offer a new way to begin to define f :

$$f(c_1, c_2, t) = \exists middle (\forall(c_3, c_4) \in \{(c_1, middle), (middle, c_2)\} (f(c_3, c_4, \lceil t/2 \rceil))).$$

We're still using the convention that a configuration name stands for the entire collection of variables that describe it. So this formula asserts that there is some configuration $middle$ such that $f(c_3, c_4, \lceil t/2 \rceil)$ is true both when:

- the variables in c_3 take on the values of the variables in c_1 and the variables in c_4 take on the values of the variables in $middle$, and
- the variables in c_3 take on the values of the variables in $middle$ and the variables in c_4 take on the values of the variables in c_2 .

Now we must recursively define $f(c_3, c_4, \lceil t/2 \rceil)$ and we must continue the recursive definition process until $\lceil t/2 \rceil = 0$ or 1.

If we do that, how long is $f(c_{start}, c_{accept}, 2^{k \cdot spacereq(M)})$? The answer is that the number of recursive steps is $\log_2(2^{k \cdot spacereq(M)})$, which is $\mathcal{O}(spacereq(M))$. And now the length of the subformula that is added at each step is also $\mathcal{O}(spacereq(M))$. So the total length of the formula, and thus the amount of time required to construct it, is $\mathcal{O}(spacereq^2(M))$. So we have described a technique that can reduce any language in PSPACE to QBF in polynomial time. ■

29.3.3 Other PSPACE-Hard and PSPACE-Complete Problems

QBF is not the only PSPACE-complete language. There are others, and many of them exploit the quantifier structure that QBF provides. We mention here some significant problems that are PSPACE-hard, many of which are also PSPACE-complete.

Two-Person Games

A quantified Boolean formula may exploit the quantifiers \forall and \exists in any order. But now consider the specific case in which they alternate. For example, we might write $\exists A (\forall B (\exists C (\forall D (P))))$, where P is a Boolean formula over the variables A, B, C , and D . This alternation naturally describes the way a player in a two-player game evaluates moves. So, for example, I could reason that a current game configuration is a guaranteed win for me if there exists *some* move that I can make and then be guaranteed a win. But then, to evaluate what will happen at the next move, I must consider that I don't get to choose the move. My opponent does. So I can only conclude that the next configuration is a win for me if *all* of the possible second moves lead to a win. At the next level, it is again my turn to choose, so I'm interested in the existence of some winning move, and so forth.

The theory of asymptotic complexity that we have developed doesn't tell us anything about solving a single problem of fixed size. So it can't be applied directly to games of fixed size. But it can be applied if we generalize the games to configurations of arbitrary size. When we do that, we discover that many popular games are PSPACE-hard. Some of them are also in PSPACE, and so are PSPACE-complete. Some appear to be harder. In particular:

- If the length of a game (i.e., the number of moves that occur before the game is over) is bounded by some polynomial function of the size of the game, then the game is likely to be PSPACE-complete.
- If the length of the game may grow exponentially with the size of the game, then the game is likely not be solvable in polynomial space. But it is likely to be solvable in exponential time and thus to be EXPTIME-complete.

Many real games are interesting precisely because they are too hard to be practically solvable by brute force search. We briefly discuss a few of them, including Sudoku, chess, and Go in § 780.

Questions about Languages and Automata

In Parts II, III, and IV, we described a variety of decision procedures for regular, context-free, and context-sensitive languages. During most of those discussions, we focused simply on decidability and ignored issues of complexity. We can now observe that several quite straightforward questions, while decidable, are hard. These include the following. After each claim, is a source for more information.

Finite state machine inequivalence: We showed, in Chapter 9, that it is decidable whether two FSMs are equivalent. Now define:

- $\text{NeqNDFSMs} = \{ \langle M_1, M_2 \rangle : M_1 \text{ and } M_2 \text{ are nondeterministic FSMs and } L(M_1) \neq L(M_2) \}$.

NeqNDFSMs is PSPACE-complete [Garey and Johnson 1979].

Finite state machine intersection: We showed, in Chapter 8, that the regular languages are closed under intersection. And we showed, in Chapter 9, that it is decidable whether the language accepted by an FSM is empty. So we know that the following language is decidable:

- $2\text{FSMs-INTERSECT} = \{ \langle M_1, M_2 \rangle : M_1 \text{ and } M_2 \text{ are deterministic FSMs and } L(M_1) \cap L(M_2) \neq \emptyset \}$.

2FSMs-INTERSECT is in P. So it is tractable. But now consider a generalization to an arbitrary number of FSMs:

- $\text{FSMs-INTERSECT} = \{ \langle M_1, M_2, \dots, M_n \rangle : M_1 \text{ through } M_n \text{ are deterministic FSMs and there exists some string accepted by all of them} \}$.

FSMs-INTERSECT is PSPACE-complete [Garey and Johnson 1979].

Regular expression inequivalence: We showed, in Chapter 6, that there exists an algorithm that can convert any regular expression into an equivalent FSM. So any question that is decidable for FSMs must also be decidable for regular expressions. So we know that the following language is decidable:

- $\text{NeqREGEX} = \{ \langle E_1, E_2 \rangle : E_1 \text{ and } E_2 \text{ are regular expressions and } L(M_1) \neq L(M_2) \}$.

NeqREGEX is PSPACE-complete [Garey and Johnson 1979].

Regular expression incompleteness: We showed, in Chapter 9, that it is decidable whether a regular language (described either as an FSM or as a regular expression) is equivalent to Σ^* . So we know that the following language is decidable:

- $\text{NOT-SIGMA-STAR} = \{ \langle E \rangle : E \text{ is a regular expression and } L(E) \neq \Sigma_E^* \}$.

NOT-SIGMA-STAR is PSPACE-complete [Sudkamp 2006].

Regular expression with squaring incompleteness: Define the language of regular expressions with squaring to be exactly the same as the language of regular expressions with the addition of one new operator defined as follows:

If α is a regular expression with squaring, then so is α^2 . $L(\alpha^2) = L(\alpha)L(\alpha)$.

Notice that the squaring operator does not introduce any descriptive power to the language of regular expressions. It does, however, make it possible to write shorter equivalents for some regular expressions. In particular, consider the regular expression that is composed of 2^n copies of α concatenated together (for some value of n). Its length is $\mathcal{O}(2^n)$. Using squaring, we can write an equivalent regular expression, $(\dots(((\alpha^2)^2)\dots))^2$, with the squaring operator applied n times, whose length is $\mathcal{O}(n)$. Since the complexity of any problem that requires reasoning about regular expressions is defined in terms of the length of the expression, this exponential compression of the size of an input string can be expected to make a difference. And it does. Define the language:

- NOT-SIGMA-STAR-SQUARING = $\{\langle E \rangle : E \text{ is a regular expression with squaring and } L(E) \neq \Sigma_E^*\}$.

While NOT-SIGMA-STAR (for standard regular expressions) is PSPACE-complete, NOT-SIGMA-STAR-SQUARING is provably not in PSPACE [Sudkamp 2006]. So we know that, since $P \subseteq PSPACE$, no polynomial-time algorithm can exist for NOT-SIGMA-STAR-SQUARING.

The membership question for context-sensitive languages: In Section 24.1, we described two techniques for answering the question, given a context-sensitive language L and a string w , is $w \in L$? One approach simulated the computation of a linear bounded automaton; the other simulated the generation of strings by a context-sensitive grammar. Unfortunately, neither of those techniques is efficient and it seems unlikely that better ones exist. Define the language:

- CONTEXT-SENSITIVE-MEMBERSHIP = $\{\langle G, w \rangle : w \in L(G)\}$.

CONTEXT-SENSITIVE-MEMBERSHIP is PSPACE-complete [Garey and Johnson 1979].

29.4 Sublinear Space Complexity

It doesn't make much sense to talk about algorithms whose time complexity is $\mathcal{O}(n)$, i.e., algorithms whose time complexity is less than linear. Such algorithms do not have time to read their entire input. But when we turn our attention to space complexity, it does make sense to consider algorithms that use a sublinear amount of working space (in addition to the space required to hold the original input). For example, consider a program P that is fed a stream of input events, eventually followed by an $\langle \text{end} \rangle$ symbol. P 's job is to count the number of events that occur before the $\langle \text{end} \rangle$. It doesn't need to remember the input stream. So the only working memory that is required is a single counter, which can be represented in binary. Thus, ignoring the space required by the input stream, $\text{spacereq}(P) \in \mathcal{O}(\log n)$.

To make it easy to talk about the space complexity of programs like P and the problems that they solve, we will make the following modification to our computational model: We will consider Turing machines with two tapes:

- A read-only input tape, and
- A read-write working tape.

While the input tape is read-only, it is not identical to the input stream of a finite state machine or of the simple counting example that we just described. The machine may move back and forth on the input tape, thus examining it any number of times.

Now we will define $\text{spacereq}(M)$ by counting only the number of visited cells of the read-write (working) tape. Notice that, if $\text{spacereq}(M)$ is at least linear, then this measure is equivalent to our original one since M 's input can be copied from the input tape to the read-write tape in $\mathcal{O}(n)$ space.

Using this new notion of space complexity, we can define two new and important space complexity classes: But first we must resolve a naming conflict. We have been using the variable L to refer to an arbitrary language. By convention,

one of the complexity classes that we are about to define is named L. To avoid confusion, we'll use the variable $L\#$ for languages when necessary. Now we can state the following definitions:

The Class L: $L\# \in L$ iff there exists some deterministic Turing machine M that decides $L\#$ and $space_{req}(M) \in \mathcal{O}(\log n)$.

The Class NL: $L\# \in NL$ iff there exists some nondeterministic Turing machine M that decides $L\#$ and $space_{req}(M) \in \mathcal{O}(\log n)$.

We have chosen to focus on $\mathcal{O}(\log n)$ because:

- Many useful problems can be solved in $\mathcal{O}(\log n)$ space. For example:
 - It is enough to remember the length of an input.
 - It is enough to remember a constant number of pointers into the input.
 - It is enough to remember a logarithmic number of Boolean values.
- It is unaffected by some reasonable changes in the way inputs are encoded. For example, it continues not to matter what base, greater than 1, is used for representing numbers.
- Savitch's Theorem can be extended to cases where $space_{req}(M) \geq \log n$.

Example 29.4 **Balanced Delimiters is in L**

Recall the balanced parentheses language $Bal = \{w \in \{\}, \{\}^* : \text{the parentheses are balanced}\}$. We have seen that Bal is not regular but it is context-free. It is also in L. It can be decided by a deterministic Turing machine M that uses its working tape to store a count, in binary, of the number of left parentheses that have not yet been matched. M will make one pass through its input. Each time it sees a left parenthesis, it will increment the count by one. Each time it sees a right parenthesis, it will decrement the count by one if it was positive. If, on the other hand, the count was zero, M will immediately reject. If, when M reaches the end of the input, the count is zero, it will accept. Otherwise it will reject. The amount of space required to store the counter grows as $\mathcal{O}(\log |w|)$, so $space_{req}(M) \in \mathcal{O}(\log n)$.

Example 29.5 **USTCON: Finding Paths in Undirected Graphs is also in L**

Let $USTCON = \{\langle G, s, t \rangle : G \text{ is an undirected graph and there exists an undirected path in } G \text{ from } s \text{ to } t\}$. In our discussion of finite state machines, we exploited an algorithm to find the states that are reachable from the start state. We used the same idea in analyzing context-free grammars to find nonterminals that are useless (because they aren't reachable from the start symbol). The obvious way to solve USTCON is the same way we solved those earlier problems: We start at s and mark the vertices that are connected to it via a single edge. Then we take each marked vertex and follow edges from it. We halt whenever we have marked the destination vertex t or we have made a complete pass through the vertices and marked no new ones. To decide USTCON then, we accept if t was marked and reject otherwise.

The simple decision procedure that we just described shows that USTCON is in P. But it fails to show that USTCON can be decided in logarithmic space because it requires (to store the marks) one bit of working storage for each vertex in G . An alternative approach shows that USTCON is in NL. Define a nondeterministic Turing machine M that searches for a path from s to t but only remembers the most recent vertex on the path. M begins by counting the vertices in G and recording the count (in binary) on its working tape. Then it starts at s and looks for a path. At each step, it nondeterministically chooses an edge from the most recent vertex it has visited to some new vertex. It stores on its working tape the index (in binary) of the new vertex. And it decrements its count by 1. If it ever selects vertex t , it halts and accepts. If, on the other hand, its count reaches 0, it halts and rejects. If there is a path from s to t , there must be one whose length is no more than the total number of vertices in G . So M will find it. And it uses only logarithmic space since both the step counter and the vertex index can be stored in space that grows logarithmically with the size of G .

It turns out that USTCON is also in L. In other words, there is a deterministic, logarithmic-space algorithm that decides it. That algorithm is described in [Reingold 2005].

What, if anything, can we say about the relationship between L, NL, and the other complexity classes that we have already considered? First, we note that trivially (since every deterministic Turing machine is also a nondeterministic one and since $\log n \in \mathcal{O}(n)$):

$$L \subseteq NL \subseteq PSPACE.$$

But what about the relationship between L and NL in the other direction? We know of no languages that are in NL and that can be proven not to be in L. But neither can we prove that $L = NL$. The $L = NL$ question exists in an epistemological limbo analogous to the $P = NP$ question. In both cases, it is widely assumed, but unproven, that the answer to the question is no.

As in the case of the $P = NP$ question, one way to increase our understanding of the $L = NL$ question is to define a class of languages that are at least as hard as every language in NL. As before, we will do this by defining a technique for reducing one language to another. In all the cases we have considered so far, we have used polynomial-time reduction. But, as we will see below, $NL \subseteq P$. So a polynomial-time reduction would dominate a logarithmic space computation. To be informative, we need to define a weaker notion of reduction. The one we will use is called logarithmic-space (or simply log-space) reduction. We will say that a language L_1 is *log-space reducible* to L_2 iff there is a deterministic two-tape Turing machine (as described above) that reduces L_1 to L_2 and whose working tape uses no more than $\mathcal{O}(\log n)$ space. Now we can define:

The Class NL-hard: A language $L\#$ is NL-hard iff every language in NL is log-space reducible to it.

The Class NL-complete: A language $L\#$ is NL-complete iff it is NL-hard and it is in NL.

Analogously to the case of NP-completeness, if we could find a single NL-complete language that is also in L, we would know that $L = NL$. So far, none has been found. But there are NL-complete languages. We mention one next.

Example 29.6 STCON: Finding Paths in Directed Graphs

Let $STCON = \{ \langle G, s, t \rangle : G \text{ is a directed graph and there exists a directed path in } G \text{ from } s \text{ to } t \}$. Note that STCON is like USTCON except that it asks for a path in a directed (rather than an undirected) graph. STCON is in NL because it can be decided by almost the same nondeterministic, log-space Turing machine that we described as a way to decide USTCON. The only difference is that now we must consider the direction of the edges that we follow.

Unlike in the case of USTCON, we know of no algorithm that shows that STCON is in L. Instead, it is possible to prove that STCON is NL-complete.

So we don't know whether $L = NL$. We also don't know the exact relationship among L, NL, and P. But it is straightforward to prove the following result about the relationship between L and P:

Theorem 29.7 $L \subseteq P$

Theorem: $L \subseteq P$.

Proof: Any language in L can be decided by a deterministic Turing machine M , where $space_{req}(M) \in \mathcal{O}(\log n)$. We can show that M must run in polynomial time by showing a bound on the number of distinct configurations it can enter. Since it halts, it can never enter the same configuration a second time. So we have a bound on the number of steps it can execute. Although M has two tapes, the contents of the first one remain the same in all configurations. So the number of distinct configurations of M on input of length n is the product of:

- The number of possible positions for the read head on the input tape. This is simply n .
- The number of different values for the working tape. Each square of the working tape can take on any element of Γ (M 's tape alphabet). So the maximum number of different values of the working tape is $|\Gamma|^{space_{req}(M)}$.

- The number of positions of the working tape's read/write head. This is bounded by its length, $space_{req}(M)$.
- The number of states of M . Call that number k .

Then, on an input of length n , the maximum number of distinct configurations of M is:

$$n \cdot |\Gamma|^{space_{req}(M)} \cdot space_{req}(M) \cdot k.$$

Since M is deciding a language in L , $space_{req}(M) \in \mathcal{O}(\log n)$. The number k is independent of n . So the maximum number of distinct configurations of M is $\mathcal{O}(n \cdot |\Gamma|^{\log n} \cdot \log n)$ or, simplifying, $\mathcal{O}(n^{1+\log |\Gamma|} \cdot \log n)$. Thus $time_{req}(M)$ is also $\mathcal{O}(n^{1+\log |\Gamma|} \cdot \log n)$ and thus $\mathcal{O}(n^{2+\log |\Gamma|})$, which is polynomial in n . So the language that M decides is in P . ■

It is also possible to prove the following theorem, which makes the stronger claim that $NL \subseteq P$:

Theorem 29.8 $NL \subseteq P$

Theorem: $NL \subseteq P$.

Proof: The proof relies on facts about $STCON = \{ \langle G, s, t \rangle : G \text{ is a directed graph and there exists a directed path in } G \text{ from } s \text{ to } t \}$. $STCON$ is in P because it can be decided by the polynomial-time marking algorithm that we described in our discussion of $USTCON$ in Example 29.5. $STCON$ is also NL -complete, which means that any other language in NL can be reduced to it in deterministic logarithmic space. But any deterministic log-space Turing machine also runs in polynomial time because the number of distinct configurations that it can enter is bounded by a polynomial, as we saw above in the proof that $L \subseteq P$. So any language in NL can be decided by the composition of two deterministic, polynomial-time Turing machines and thus is in P . ■

We can summarize what we know as:

$$L \subseteq NL \subseteq P \subseteq PSPACE.$$

Just as we have done for the other complexity classes that we have considered, we can define classes that contain the complements of languages in L and in NL . So we have:

The Class $co-L$: $L\# \in co-L$ iff $\neg L\# \in L$.

The Class $co-NL$: $L\# \in co-NL$ iff $\neg L\# \in NL$.

It is easy to show that the class L is closed under complement and thus that $L = co-L$: if a language $L\#$ is decided by a deterministic Turing machine M , then there exists another deterministic Turing machine M' such that M' decides $\neg L\#$. M' is simply M with the y and n states reversed. $space_{req}(M') = space_{req}(M)$. So, if $L\#$ is in the class L , so is its complement.

It is less obvious, but true, that $NL = co-NL$. The proof follows from the more general claim that we will state as Theorem 29.10 (the Immerman-Szelepcsényi Theorem) in the next section. There we will see that all nondeterministic space complexity classes whose space requirement is at least $\log n$ are closed under complement.

29.5 The Closure of Space Complexity Classes Under Complement

Recall that the class P is closed under complement. On the other hand, it is believed that $NP \neq co-NP$, although no proof of that exists. When we switch from considering time complexity to considering space complexity, the situation is clearer. Both deterministic and nondeterministic space-complexity classes are closed under complement. The fact that deterministic ones are is obvious (since a deciding machine for $\neg L$ is simply the deciding machine for L with its accepting and rejecting states reversed). The fact that nondeterministic ones are is not obvious.

To make it easy to state the next group of theorems, we will define the following families of languages:

- $dSPACE(f(n))$ = the set of languages that can be decided by some deterministic Turing machine M , where $space(M) \in \mathcal{O}(f(n))$.
- $NSPACE(f(n))$ = the set of languages that can be decided by some nondeterministic Turing machine M , where $space(M) \in \mathcal{O}(f(n))$.
- $co-DSPACE(f(n))$ = the set of languages whose complements can be decided by some deterministic Turing machine M , where $space(M) \in \mathcal{O}(f(n))$.
- $co-NSPACE(f(n))$ = the set of languages whose complements can be decided by some nondeterministic Turing machine M , where $space(M) \in \mathcal{O}(f(n))$.

Theorem 29.9 Deterministic Space-Complexity Classes are Closed under Complement

Theorem: For every function $f(n)$, $dSPACE(f(n)) = co-DSPACE(f(n))$.

Proof: If L is a language that is decided by some deterministic Turing machine M , then the deterministic Turing machine M' that is identical to M except that the halting states y and n are reversed decides $\neg L$. $space(M') = space(M)$. So, if $L \in dSPACE(f(n))$, so is $\neg L$. ■

Theorem 29.10 The Immerman-Szelepcsényi Theorem and the Closure of Nondeterministic Space-Complexity Classes under Complement

Theorem: For every function $f(n) \geq \log n$, $NSPACE(f(n)) = co-NSPACE(f(n))$.

Proof: The proof of this claim, that the nondeterministic space complexity classes are closed under complement, was given independently in [Immerman 1988] and [Szelepcsényi 1988]. ■

One application of the Immerman-Szelepcsényi Theorem is to a question we asked in Section 24.1, “Are the context-sensitive languages closed under complement?” We are now in a position to sketch a proof of Theorem 24.11, which we restate here as Theorem 29.11:

Theorem 29.11 Closure of the Context-Sensitive Languages under Complement

Theorem: The context-sensitive languages are closed under complement.

Proof: Recall that a language is context-sensitive iff it is accepted by some linear bounded automaton (LBA). An LBA is a nondeterministic Turing machine whose space is bounded by the length of its input. So the class of context-sensitive languages is exactly $NSPACE(n)$. By the Immerman-Szelepcsényi Theorem, $NSPACE(n) = co-NSPACE(n)$. So the complement of every context-sensitive language can also be decided by a nondeterministic Turing machine that uses linear space (i.e., an LBA). Thus it too is context-sensitive. ■

29.6 Space Hierarchy Theorems

We saw, in Section 28.9.1, that giving a Turing machine more time increases the class of languages that can be decided. The same is true of increases in space. We can prove a pair of space hierarchy theorems that are similar to the time hierarchy theorems that we have already described. The main difference is that the space hierarchy theorems that we can prove are stronger than the corresponding time hierarchy ones because running space-bounded simulations does not require the overhead that appears to be required in the time-bounded case.

Before we can define the theorems, we must define the class of space-requirement functions to which they will apply. So, analogously (but not identically) to the way we defined time constructibility, we will define space-constructibility:

A function $s(n)$ from the positive integers to the positive integers is *space-constructible* iff:

- $s(n) \geq \log n$, and
- the function that maps the unary representation of n (i.e., 1^n) to the binary representation of $s(n)$ can be computed in $\mathcal{O}(s(n))$ space.

Most useful functions, as long as they are at least $\log n$, are space-constructible.

Whenever we say that, for some Turing machine M , $\text{spacereq}(M) \in \mathcal{O}(n)$, we are using, as our definition of a Turing machine, the two-tape machine that we described in Section 29.4. In that case, we will take $\text{spacereq}(M)$ to be the size of M 's working tape.

Theorem 29.12 Deterministic Space Hierarchy Theorem

Theorem: For any space-constructible function $s(n)$, there exists a language $L_{s(n)\text{hard}}$ that is decidable in $\mathcal{O}(s(n))$ space but that is not decidable in $\mathcal{O}(s(n))$ space.

Proof: The proof is by diagonalization and is similar to the proof we gave for Theorem 28.27 (the Deterministic Space Hierarchy Theorem). The tighter bound in this theorem comes from the fact that it is possible to describe an efficient space-bounded simulator. The details of the proof, and in particular, the design of the simulator, are left as an exercise. ■

29.7 Exercises

- 1) In Section 29.1.2, we defined $\text{MaxConfigs}(M)$ to be $|K| \cdot |\Gamma|^{\text{spacereq}(M)} \cdot \text{spacereq}(M)$. We then claimed that, if c is a constant greater than $|\Gamma|$, then $\text{MaxConfigs}(M) \in \mathcal{O}(c^{\text{spacereq}(M)})$. Prove this claim by proving the following more general claim:

Given: f is a function from the natural numbers to the positive reals,
 f is monotonically increasing and unbounded,
 a and c are positive reals, and
 $1 < a < c$.

Then: $f(n) \cdot a^{f(n)} \in \mathcal{O}(c^{f(n)})$.

- 2) Prove that PSPACE is closed under:

- a) Complement.
- b) Union.
- c) Concatenation.
- d) Kleene star.

- 3) Define the language:

- $U = \{ \langle M, w, 1^s \rangle : M \text{ is a Turing machine that accepts } w \text{ within space } s \}$.

Prove that U is PSPACE-complete

- 4) In Section 28.7.3, we defined the language $2\text{-SAT} = \{ \langle w \rangle : w \text{ is a wff in Boolean logic, } w \text{ is in 2-conjunctive normal form and } w \text{ is satisfiable} \}$ and saw that it is in P. Show that 2-SAT is NL-complete.
- 5) Prove that $A^n B^n = \{ a^n b^n : n \geq 0 \}$ is in L.

- 6) In Example 21.5, we described the game of Nim. We also showed an efficient technique for deciding whether or not the current player has a guaranteed win. Define the language:
- $\text{NIM} = \{ \langle b \rangle : b \text{ is a Nim configuration (i.e., a set of piles of sticks) and there is a guaranteed win for the current player} \}$.

Prove that $\text{NIM} \in \text{L}$.

- 7) Prove Theorem 29.12 (The Deterministic Space Hierarchy Theorem).

30 Practical Solutions for Hard Problems

It appears unlikely that $P = NP$. It appears even more unlikely that, even if it does, a proof that it does will lead us to efficient algorithms to solve the hard problems that we have been discussing. (We base this second claim on at least two observations. The first is that people have looked long and hard for such algorithms and have failed to find them. The second is that just being polynomial is not sufficient to assure efficiency in any practical sense.) And things are worse. Some problems, for example those with a structure like generalized chess, are provably outside of P , whatever the verdict on NP is. Yet important applications depend on algorithms to solve these problems. So what can we do?

30.1 Approaches

In our discussion of the traveling salesman problem at the beginning of Chapter 27, we suggested two strategies for developing an efficient algorithm to solve a hard problem:

Compromise on generality: Design an algorithm that finds an optimal solution and that runs efficiently on most (although not necessarily all) problem instances. This approach is particularly useful if the problems that we actually care about solving possess particular kinds of structures and we can find an algorithm that is tuned to work well on those structures. We have already considered some examples of this approach:

- Very large real instances of the traveling salesman problem can be solved efficiently by iteratively solving a linear programming problem that is a relaxed instance of the exact problem. Although, in principle, it could happen that each such iteration removes only a single tour from consideration, when the graph corresponds to a real problem, large numbers of tours can almost always be eliminated at each step.
- Some very large Boolean formulas can be represented efficiently using ordered binary decision diagrams (OBDDs), as described in § 612. That efficient representation makes it possible to solve the satisfiability (SAT) problem efficiently. The OBDD representation of a randomly constructed Boolean formula may not be compact. But OBDDs exploit exactly the kinds of structures that typically appear in formulas that have been derived from natural problems, such as digital circuits.

Compromise on optimality: Design an approximation algorithm that is guaranteed to find a good (although not necessarily optimal) solution and to do so efficiently. This approach is particularly attractive if the error that may be introduced in finding the solution is relatively small in comparison with errors that may have been introduced in the process of defining the problem itself. For example, in any real instance of the traveling salesman problem, we must start by measuring the physical world and no such measurement can be exact. Or consider the large class of problems in which we seek to maximize (or minimize) the value of some objective function that combines, into a single number, multiple numbers that measure the utility of a proposed solution along two or more dimensions. For example, we might define a cost function for a proposed stretch of new divided highway to be something like:

$$\text{cost}(s) = 4\text{-dollar-cost}(s) - 2\text{-number-of-lives-saved-by}(s) - 1.5\text{-commuting-hours-saved-per-week}(s).$$

Since the objective function is only an approximate measure of the utility of a new road, an approximately optimal solution to a highway system design problem may be perfectly acceptable.

Compromise on both: For some problems, it turns out that if we make some assumptions about problem structure then we can find very good, but not necessarily optimum solutions very quickly. For example, suppose that we limit the traveling salesman problem to graphs that satisfy the triangle inequality (as described in Section 27.1). Real world maps meet that constraint. Then there exist algorithms for finding very good solutions very quickly.

A fourth approach, useful in some kinds of problems is:

Compromise on total automation: Design an algorithm that works interactively with a human user who guides it into the most promising regions of its search space.

When applied to many practical problems, including verifying the correctness of both hardware and software systems, automatic theorem provers face exponential growth in the number of paths that must be considered. One way to focus such systems on paths that are likely to lead to the desired proofs is to let a human user guide the system. © 679.

In most of these approaches, we are admitting that we have no efficient and “direct” algorithm for finding the answer that we seek. Instead, we conduct a search through a space that is defined by the structure of the problem we are trying to solve. In the next two sections we will sketch two quite different approaches to conducting that search. In particular, we’ll consider:

- **Approach 1:** The space is structured randomly. Exploit that randomness.
- **Approach 2:** The space isn’t structured randomly and we have some knowledge about the structure that exists. Exploit that knowledge.

30.2 Randomized Algorithms and the Language Classes BPP, RP, co-RP and ZPP

For some kinds of problems, it is possible to avoid the expensive behavior of an exhaustive search algorithm by making a sequence of random guesses that, almost all of the time, converge efficiently to an answer that is correct.

Example 30.1 Quicksort

We’ll illustrate the idea of random guessing with a common algorithm that reduces the expected time required to sort a list from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. Given a list of n elements, define *quicksort* as follows:

quicksort(list: a list of n elements) =

1. If n is 0 or 1, return *list*. Otherwise:
2. Choose an element from *list*. Call it the *pivot*.
3. Reorder the elements in *list* so that every element that is less than *pivot* occurs ahead of it and every element that is greater than *pivot* occurs after it. If there are equal elements, they may be left in any order.
4. Recursively call *quicksort* with the fragment of *list* that includes the elements up to, but not including *pivot*.
5. Recursively call *quicksort* with the fragment of *list* that includes all the elements after *pivot*.

Quicksort always halts with its input list correctly sorted. At issue is the time required to do so. Step 3 can be done in $\mathcal{O}(n)$ steps. In fact, it can usually be implemented very efficiently. When step 3 is complete, *pivot* is in the correct place in *list*.

In the worst case, *quicksort* runs in $\mathcal{O}(n^2)$ time. This happens if, at each step, the reordering places all the elements on the same side of *pivot*. Then the length of *list* is reduced by only 1 each time *quicksort* is called. In the best case, however, the length of *list* is cut in half each time. When this happens, *quicksort* runs in $\mathcal{O}(n \log n)$ time.

The key to *quicksort*’s performance is a judicious choice of *pivot*. One particularly bad strategy is to choose the first element of *list*. In the not uncommon case in which *list* is already sorted, or nearly sorted, this choice will force worst-case performance. Any other systematic choice may also be bad if *list* is constructed by a malicious attacker with the goal of forcing worst-case behavior. The solution to this problem is to choose *pivot* randomly. When that is done, *quicksort*’s expected running time, like its best case running time, is $\mathcal{O}(n \log n)$.

In the next section, we’ll take the idea of random guessing and use it to build Turing machines that decide languages.

30.2.2 Randomized Algorithms

A *randomized algorithm* (sometimes called a *probabilistic algorithm*) is one that exploits the random guessing strategy that we have just described. Randomized algorithms are used when:

- The problem at hand can usually be solved without exhaustively considering all paths to a solution.
- A systematic way of choosing paths would be vulnerable to common kinds of bad luck (for example, being asked to sort a list that was already sorted) or to a malicious attacker that would explicitly construct worst-case instances if it knew how to do so.

Randomized algorithms are routinely exploited in cryptographic applications. © 722.

We can describe randomized algorithms as Turing machines. Call every step at which a nondeterministic Turing machine must choose from among competing moves a *choice point*. Define a *randomized Turing machine* to be a nondeterministic Turing machine M with the following properties:

- At every choice point, there are exactly two moves from which to choose.
- At every choice point, M (figuratively) flips a fair coin and uses the result of the coin toss to decide which of its two branches to pursue.

Note that the constraint of exactly two moves at each choice point is only significant in the sense that it will simplify our analysis of the behavior of these machines. Any nondeterministic Turing machine can be converted into one with a branching factor of two by replacing an n -way branch with several 2-way ones.

Since the coin flips are independent of each other, we have that, if b is a single path in a randomized Turing machine M and the number of choice points along b is k , then the probability that M will take b is:

$$\Pr(b) = 2^{-k}.$$

Note that every deterministic Turing machine is a randomized Turing machine that happens, on every input, to have zero choice points and thus a single branch whose probability is 1.

Now consider the specific case in which the job of M is to decide a language. A standard (nonrandomized) nondeterministic Turing machine accepts its input w iff there is at least one path that accepts. A randomized Turing machine only follows one path. It accepts iff that path accepts. It rejects iff that path rejects. So the probability that M accepts w is the sum of the probabilities of all of M 's accepting paths. The probability that M rejects w is the sum of the probabilities of all of M 's rejecting paths. Alternatively, it is $1 - \Pr(M \text{ accepts})$.

If the job of a randomized Turing machine M is to accept the language L , then there are two kinds of mistakes it could make: it could erroneously accept a string that is not in L , or it could erroneously reject one that is. We would like to be able to place a bound on the likelihood of both kinds of errors. So:

- We'll say that M accepts L with a *false positive* probability, ϵ_P , iff $(w \notin L) \rightarrow (\Pr(M \text{ accepts } w) \leq \epsilon_P)$.
- We'll say that M accepts L with a *false negative* probability, ϵ_N , iff $(w \in L) \rightarrow (\Pr(M \text{ rejects } w) \leq \epsilon_N)$.

If M is a randomized Turing machine, we define $timereq(M)$ and $spacereq(M)$ as for standard Turing machines. In both cases, we measure the complexity of the worst case of M 's performance on an input of size n .

We're now in a position to define a set of complexity classes based on acceptance by randomized Turing machines. In the next section, we'll define four such classes, all of them focused on accepting in polynomial time. It is possible to define other classes as well. For example, we could talk about languages that can be accepted by randomized Turing machines that use logarithmic space.

30.2.3 The Language Classes BPP, RP, co-RP, and ZPP

Our goal is to recognize a language with reasonable accuracy in a reasonable amount of time. When we use randomization to do that, there are two kinds of failure modes that we must consider:

- The algorithm always runs efficiently but it may (with small probability) deliver an incorrect answer. Algorithms with this property are called *Monte Carlo algorithms*.

- The algorithm never returns an incorrect answer but it may (with small probability) be very expensive to run. Algorithms with this property are called *Las Vegas algorithms*.

We can define complexity classes based on imposing constraints on both kinds of failures.

We begin with the first. Define:

The Class BPP: $L \in \text{BPP}$ iff there exists some probabilistic Turing machine M that runs in polynomial time and that decides L with a false positive probability, ϵ_P , and a false negative probability, ϵ_N , both less than $1/2$. The name BPP stands for **B**ounded-error, **P**robabilistic, **P**olynomial time.

A randomized Turing machine that decides a language in BPP implements a Monte Carlo algorithm. It is allowed to make both kinds of errors (i.e., false positives and false negatives) as long as the probability of making either of them is less than $1/2$. We can characterize such a machine in terms of a single error rate $\epsilon = \max(\epsilon_P, \epsilon_N)$. The requirement that ϵ be less than or equal to $1/2$ may seem too weak. It's hard to imagine saying that M decides L if it only gets it right about half the time. But it is possible to prove the following theorem:

Theorem 30.1 Reducing the Error Rate

Theorem: Let M be a randomized, polynomial-time Turing machine with error rate ϵ that is a constant equal to $\max(\epsilon_P, \epsilon_N)$. If $0 < \epsilon < 1/2$ and $f(n)$ is any polynomial function, then there exists an equivalent randomized, polynomial-time Turing machine M' with error rate $2^{-f(n)}$.

Proof: The idea is that M' will run M some polynomial number of times and return the answer that appeared more often. If the runs are independent, then the probability of error decreases exponentially as the number of runs of M increases. For a detailed analysis that shows that the desired error bound can be achieved with a polynomial number of runs of M , see [Sipser 2006].

■

So, for example, the definition of the class BPP wouldn't change if we required ϵ to be less than $1/10$ or $1/3000$ or $\frac{1}{2^{3000}}$. Note that the latter is substantially less than the probability that any computer on which M runs will experience a hardware failure that would cause it to return an erroneous result.

The class BPP is closed under complement. In other words, $\text{BPP} = \text{co-BPP}$ since false positives and false negatives are treated identically.

Sometimes it is possible to build a machine to accept a language L and to guarantee that only one kind of error will occur. It may be possible to examine a string w and to detect efficiently some property that proves that w is in L . Or it may be possible to detect efficiently some way in which w violates the membership requirement for L . So define:

The Class RP: $L \in \text{RP}$ iff there exists some randomized Turing machine M that runs in polynomial time and that decides L and where:

- If $w \in L$ then M accepts w with probability $1 - \epsilon_N$, where $\epsilon_N < 1/2$, and
- If $w \notin L$ then M rejects w with probability 1 (i.e., with false positive probability $\epsilon_P = 0$).

The name RP stands for **R**andomized, **P**olynomial time.

If L is in RP, then it can be decided by a randomized Turing machine that may reject when it shouldn't. But it will never accept when it shouldn't. Of course, it may also be possible to build a machine that does the opposite. So define the complement of RP:

The Class co-RP: $L \in \text{co-RP}$ iff there exists some randomized Turing machine M that runs in polynomial time and that decides L and where:

- If $w \in L$ then M accepts w with probability 1 (i.e., with false negative probability $\epsilon_N = 0$), and
- If $w \notin L$ then M rejects w with probability $1 - \epsilon_P$, where $\epsilon_P < 1/2$.

Note that, as in the definition of BPP, the error probabilities required for either RP or co-RP can be anything strictly between 0 and $1/2$ without changing the set of languages that can be accepted.

In the next section, we will present a randomized algorithm for primality testing. An obvious way to decide whether a number is prime would be to look for the existence of a factor that proves that the number isn't prime. The algorithm we will present doesn't do that, but it does look for the existence of a certificate that proves that its input isn't prime. If it finds such a certificate, it can report, with probability 1, that the input is composite. If it fails to find such a certificate, then it reports that the input is prime. That report has high probability of being the correct answer. We will use our algorithm to show that the language $\text{PRIMES} = \{w : w \text{ is the binary encoding of a prime number}\}$ is in co-RP and the language $\text{COMPOSITES} = \{w : w \text{ is the binary encoding of a composite number}\}$ is in RP.

But first let's consider what appears to be a different approach to the use of randomness. Suppose that we want to require an error rate of 0 and, in exchange, we are willing to accept a nonzero probability of a long run time. We call algorithms that satisfy this requirement Las Vegas algorithms. To describe the languages that can be accepted by machines that implement algorithms with this property, define:

The Class ZPP: $L \in \text{ZPP}$ iff there exists some randomized Turing machine M such that:

- If $w \in L$ then M accepts w with probability 1,
- If $w \notin L$ then M rejects w with probability 1, and
- There exists a polynomial function $f(n)$ such that, for all inputs w of length n , the expected running time of M on w is less than $f(n)$. It is nevertheless possible that M may run longer than $f(n)$ for some sequences of random events.

The name ZPP stands for **Z**ero-error, **P**robabilistic, **P**olynomial time.

There are two other, but equivalent ways to define ZPP:

- ZPP is the class of languages that can be recognized by some randomized Turing machine M that runs in polynomial time and that outputs one of three possible values: *Accept*, *Reject*, and *Don't Know*. M must never accept when it should reject nor reject when it should accept. Its probability of saying *Don't Know* must be less than $1/2$. This definition is equivalent to our original one because it says that, if M runs out of time before determining an answer, it can quit and say *Don't Know*.
- $\text{ZPP} = \text{RP} \cap \text{co-RP}$. To prove that this definition is equivalent to our original one, we show that each implies the other:

$(L \in \text{ZPP}) \rightarrow (L \in \text{RP} \cap \text{co-RP})$: If L is in ZPP, then there is a Las Vegas-style Turing machine M that accepts it. We can construct Monte Carlo-style Turing machines M_1 and M_2 that show that L is also in RP and in co-RP, respectively. On any input w , M_1 will run M on w for its expected running time or until it halts. If M halts naturally in that time, then M_1 will accept or reject as M would have done. Otherwise, it will reject. The probability that M will have halted is at least $1/2$, so the probability that M_1 will falsely reject a string that is in L is less than $1/2$. Since M_1 runs in polynomial time, it shows that L is in RP. Similarly, construct M_2 that shows that L is in co-RP except that, if the simulation of M does not halt, M_2 will accept.

$(L \in \text{RP} \cap \text{co-RP}) \rightarrow (L \in \text{ZPP})$: If L is in RP, then there is a Monte Carlo-style Turing machine M_1 that decides it and that never accepts when it shouldn't. If L is in co-RP, then there is another Monte Carlo-style Turing machine M_2 that decides it and that never rejects when it shouldn't. From these two, we can construct a Las

Vegas-style Turing machine M that shows that L is in ZPP. On any input w , M will first run M_1 on w . If it accepts, M will halt and accept. Otherwise M will run M_2 on w . If it rejects, M will halt and reject. If neither of these things happens, it will try again.

Randomization appears to be a useful tool for solving some kinds of problems. But what can we say about the relationships among BPP, RP, co-RP, ZPP and the other complexity classes that we have considered? The class P must be a subset of all four of the randomized classes since a standard, deterministic, polynomial-time Turing machine that doesn't happen to have any choice points satisfies the requirements for a machine that accepts languages in all of those classes. Further, we have already shown that ZPP is a subset of both RP and co-RP. So all of the following relationships are known:

$$\begin{aligned} P &\subseteq \text{BPP}. \\ P &\subseteq \text{ZPP} \subseteq \text{RP} \subseteq \text{NP}. \\ P &\subseteq \text{ZPP} \subseteq \text{co-RP} \subseteq \text{co-NP}. \\ \text{RP} \cup \text{co-RP} &\subseteq \text{BPP}. \end{aligned}$$

There are two big unknowns. One is the relationship between BPP and NP. Neither is known to be a subset of the other. The other is whether P is a proper subset of BPP. It is widely conjectured, but unproven, that $\text{BPP} = P$. If this is true, then randomization is a useful tool for constructing practical algorithms for some problems but it is not a technique that will make it possible to construct polynomial-time solutions for NP-complete problems unless $P = \text{NP}$.

30.2.4 Primality Testing

One of the most important applications of randomized algorithms is the problem of primality checking. We mentioned above that $\text{PRIMES} = \{w : w \text{ is the binary encoding of a prime number}\}$ is in co-RP and $\text{COMPOSITES} = \{w : w \text{ is the binary encoding of a composite number}\}$ is in RP. In this section, we will see why.

Recall that the obvious way to decide whether an integer p is prime is to consider all of the integers between 2 and \sqrt{p} , checking each to see whether it divides evenly into p . If any of them does, then p isn't prime. If none does, then p is prime. The time required to implement this approach is $\mathcal{O}(\sqrt{p})$. But n , the length of the string that encodes p , is $\log p$. So this simple algorithm is $\mathcal{O}(2^{n/2})$. It has recently been shown that PRIMES is in P, so there exists a polynomial-time algorithm that solves this problem exactly.

But, well before that result was announced, randomized algorithms were being used successfully in applications, such as cryptography, that require the ability to perform primality checking quickly. One idea for a randomized algorithm that would check the primality of p is to pick randomly some proposed factors of p and check them. If any of them is a factor, then p is composite. Otherwise, claim that p is prime. The problem with this idea is that, if p is large, most numbers may fail to be factors, even if p is composite. So it would be necessary to try a very large number of possible factors in order to be able to assert with high probability that p is prime. There is a better way.

The randomized algorithm that we are about to present is similar in its overall structure to the factor-testing method that we just rejected. It will randomly choose some numbers and check each to see whether it proves that p is not prime. If none of them does, it will report that p is (highly likely to be) prime. Its effectiveness relies on a few fundamental facts about modular arithmetic. To simplify the rest of this discussion, let $x \equiv_p y$, read “ x is equivalent to $y \pmod{p}$ ” mean that x and y have the same remainder when divided by p .

The first result that we will use is known as **Fermat's Little Theorem** \square . It tells us the following:

$$\text{If } p \text{ is prime, then, for any positive integer } a, \text{ if } \gcd(a, p) = 1, \text{ then } a^{p-1} \equiv_p 1.$$

Recall that the greatest common divisor (gcd) of two integers is the largest integer that is a factor of both of them. We'll say that p passes the Fermat test at a iff $a^{p-1} \equiv_p 1$. For example, let $p = 5$ and $a = 3$. Then $3^{(5-1)} = 81 \equiv_5 1$. So 5 passes the Fermat test at 3, which it must do since 5 is prime and 3 and 5 are relatively prime. But now let $p = 8$ and $a = 3$. Then $3^{(8-1)} = 2187 \equiv_8 3$. So 8 fails the Fermat test at 3, which is consistent with the theorem, since 8 is not prime. Whenever p fails the Fermat test at a , we'll say that a is a **Fermat witness** that p is composite.

Fermat's Little Theorem tells us that if p is prime, then it must pass the Fermat test at every appropriately chosen value of a . Can we turn this around? If p passes the Fermat test at some value a , do we know that p is prime? The answer to this question is no. If p is composite and yet it passes the Fermat test at a , we will say that a is a **Fermat liar** that p is prime.

Fermat's Little Theorem is the basis for a simple randomized algorithm for deciding the primality of p . We'll randomly choose values for a , looking for a witness that p is composite. We'll only consider values that are less than p . So, if p is prime, $\gcd(a, p)$ will always be 1. Thus our algorithm will not have to evaluate \gcd . If we fail to find a witness that shows that p is composite, we'll report that p is probably prime. Because liars exist, we can increase the likelihood of finding such a witness, if one exists, by increasing the number of candidate witnesses that we test. So we'll present an algorithm that takes two inputs, a value to be tested and the number of possible witnesses that should be checked. The output will be one of two values: *composite* and *probably prime*:

```
simpleFermat(p: integer, k: integer) =
1. Do k times:
  1.1. Randomly select a value a in the range [2: p-1].
  1.2. If it is not true that  $a^{p-1} \equiv_p 1$ , then return composite.
2. All tests have passed. Return probably prime.
```

Modular exponentiation can be implemented efficiently using the technique of successive squaring that we describe in Example 41.1. So *simpleFermat* runs in polynomial time. All that remains is to determine its error rate as a function of k . With the exception of a small class of special composite numbers that we will describe below, if p is composite, then the chance that any a is a Fermat liar for it is less than $1/2$. So, again with the exception we are about to describe, the error rate of *simpleFermat* is less than $1/2^k$.

But now we must consider the existence of composite numbers that pass the Fermat test at all values. Call such numbers **Carmichael numbers** \square . Every value of a is a Fermat liar for every Carmichael number, so no value of k will enable *simpleFermat* to realize that a Carmichael number isn't prime.

However, there is a separate randomized test that we can use to detect Carmichael numbers. It is based on the following fact: if p is prime, then 1 has exactly two square roots (mod p): 1 and -1. If, on the other hand, p is composite, it is possible that 1 has three or more square roots (mod p). For example, let $p = 8$. Then we have:

- $1^2 = 1$.
- $3^2 = 9 \equiv_8 1$.
- $5^2 = 25 \equiv_8 1$. Note that $5 \equiv_8 -3$.
- $7^2 = 49 \equiv_8 1$. Note that $7 \equiv_8 -1$.

So we can write the four square roots of 1 (mod 8) as 1, -1, 3, -3. Every Carmichael number has more than two square roots. For example, the smallest Carmichael number is 561. The square roots of 1 (mod 561) are 1, -1 (560), 67, -67 (494), 188 (-373), 254, and -254 (-307).

While *simpleFermat* cannot distinguish between primes and Carmichael numbers, a randomized test based on finding square roots can. We could design such a test that just chooses random values and checks to see whether they are square roots of 1 (mod p). If any is, then p isn't prime. And, unlike with *simpleFermat*, there exist witnesses even for Carmichael composite numbers. But there's a more efficient way to find additional square roots if they exist. Suppose that we have done the *simpleFermat* test at a and a has passed. Then we know that $a^{p-1} \equiv_p 1$. Taking the square root of both sides, we get that $a^{(p-1)/2}$ is a square root of 1 (mod p). If $a^{(p-1)/2} \equiv_p 1$, we haven't learned anything. But then we can again take the square root of both sides and continue until one of the following things happens:

- 1) We get a root that is -1. We're not interested in finding square roots of -1. So we give up, having failed to show that any additional roots exist. It is possible that p is prime or that it is composite.
- 2) We get a noninteger. Again, we simply stop as in case 1.
- 3) We get a root that is neither 1 nor -1. We have shown that p is composite.

So (taking all results (mod p)) we check:

- a^{p-1} (and, as in *simpleFermat*, assert composite if we get any value other than 1), then
- $a^{(p-1)/2}$ (and assert composite if we get any value other than 1 or -1), then
- $a^{(p-1)/4}$ (and assert composite if we get any value other than 1 or -1), and so forth, quitting as described above.

The most efficient way to generate this set of tests is in the opposite order. But, to do that, we need to know where to start. In particular, we need to know when the result (if we were going in the order we described above) would no longer be an integer. Suppose that $p-1$ is represented in binary. Then it can be rewritten as $d \cdot 2^s$, where d is odd. (The number s is the number of trailing 0's and the number d is what is left after the trailing 0's are removed.) The number of times that we would be able to take the square root of a^{p-1} and still get an integer is s . So we compute (mod p) the reverse of the sequence we described above:

$$a^{d \cdot 2^0}, a^{d \cdot 2^1}, \dots, a^{d \cdot 2^s}.$$

Then we check the sequence right to left. If the last element is not 1, then a fails the simple Fermat test and we can report that p is composite. Otherwise, as long as the values are 1, we continue. If we encounter -1, we must quit and report that we found no evidence that p is composite. If, on the other hand, we find some value other than 1 or -1, we can report that p is composite.

Using this idea, we can state the following algorithm, which is generally known as the Miller-Rabin test :

Miller-Rabin(p : integer, k : integer) =

1. If $p = 2$, return *prime*. Else, if p is even, return *composite*.
2. Rewrite $p-1$ as $d \cdot 2^s$, where d is odd.
3. Do k times:
 - 3.1. Randomly select a value a in the range $[2: p-1]$.
 - 3.2. Compute the following sequence (mod p):

$$a^{d \cdot 2^0}, a^{d \cdot 2^1}, \dots, a^{d \cdot 2^s}.$$

- 3.3. If the last element of the sequence is not 1, then a fails the simple Fermat test. Return *composite*.
- 3.4. For $i = s-1$ down to 0 do:

If $a^{d \cdot 2^i} = -1$, then exit this loop. Otherwise, if it is not 1, then return *composite*.
4. All tests have passed. Return *probably prime*.

Miller-Rabin runs in polynomial time and can be shown \square to have an error rate that is less than $1/4^k$. So it proves the claim, made above, that the language COMPOSITES is in RP. The efficiency of the algorithm can be improved in various ways. One is to check the elements of the sequence as they are generated. It's harder to see how to do that correctly, but it can cut out some tests. In fact, the algorithm is generally stated in that form.

While randomized algorithms provide a practical way to check for primality and thus to find large prime numbers, they do not tell us how to factor a large number this is known not to be prime. Modern cryptographic techniques, such as the RSA algorithm, rely on two important facts: generating primes can be done efficiently, but no efficient technique for factoring composites is known. $\text{C } 722$.

30.3 Heuristic Search

For some problems, randomized search works well. But suppose that we have some useful information about the shape of the space that we are attempting to search. Then it may make sense not to behave randomly but instead to exploit our knowledge each time a choice needs to be made.

30.3.1 An Introduction to Heuristic Search

A large class of important problems can be described generically as:

- A space of states that correspond to configurations of the problem situation.
- A start state.
- One or more goal states. If there are multiple goal states, then the set of them must be efficiently decidable.
- A set of operators (with associated costs) that describe how it is possible to move from one state to another.

Many puzzles are easy to state in this way. For example:

- In the 15-puzzle, which we described in Example 4.8, the states correspond to arrangements of tiles on the board. An instance of the puzzle specifies a particular arrangement of tiles as the start state. There is a single goal state in which the tiles are arranged in numeric order. And there is a set of legal moves. Specifically, it is possible to move from one state to another by sliding any tile that is adjacent to the empty square into the empty square.
- In the game Instant Insanity®, which we describe in § 782, the states correspond to the arrangement of blocks to form a stack. The start state describes a set of blocks, none of which is in the stack. There is a set of goal states (since a goal state is any state in which there is a stack that contains all the blocks and the colors are lined up as required). And there is a set of operators that correspond to adding and removing blocks from the stack.

More significantly, many real problems can also be described as state space search. For example, an airline scheduling problem can be described as a search through a space in which the states correspond to partial assignments of planes and crews to routes. The start state contains no assignments. Any state that assigns a plane and a crew to every flight and that meets some prescribed set of constraints is a goal state. The operators move from one state to the next by making (or unmaking) plane and crew assignments.

Now suppose that we are given a state space search problem and asked to find the shortest (cheapest) path from the start state to a goal. One approach is to conduct a systematic search through the state space by applying operators, starting from the start state. The problem with this technique is that, for many problems, the number of possible states and thus the number of paths that might have to be examined grows exponentially with the size of the problem being considered. For example, if we generalize the 15-puzzle to the n -puzzle (where $n=k^2-1$ for some positive integer k), then the number of distinct puzzle states is $(n+1)!$ It can be shown that the problem of finding the shortest sequence of moves that transforms a given start state into the goal state is NP-hard.

In Section 28.7.4, we showed, by exhibiting a $\mathcal{O}(n^3)$ algorithm to decide it, that the language SHORTEST-PATH = $\{ \langle G, u, v, k \rangle : G \text{ is an unweighted, undirected graph, } u, \text{ and } v \text{ are vertices in } G, k \geq 0, \text{ and there exists a path from } u \text{ to } v \text{ whose length is at most } k \}$ is in P. And we pointed out that the extension of SHORTEST-PATH to the case of weighted graphs is also in P. So why can't an instance of the n -puzzle (and other problems like it) be solved in polynomial time? The problem is to find the shortest path from the start state to a goal state and we appear to know how to do that efficiently.

The answer is simple. An instance of SHORTEST-PATH must contain an *explicit* description of the entire space that is to be searched. The vertices of G correspond to the states and the edges of G correspond to the operators that describe the legal moves from one state to the next. When we say that SHORTEST-PATH is in P, we mean that the amount of time that is required to search the space and find the shortest path is a polynomial function of the length of that *complete* state description.


But now consider an instance of the n -puzzle. It can be described much more succinctly. Instead of explicitly enumerating all the board configurations and the moves between them, we can describe just the start and goal

configurations, along with a function that defines the operators. This function, when given a state, returns a set of successor states and associated costs. What we do not have to do is to list explicitly the $(n+1)!$ states that describe configurations of the puzzle. An exhaustive search that requires considering that list would require time that is exponential in the length of the succinct description that we want to use, even though it would have been polynomial in the length of the explicit description that is required for an instance of SHORTEST-PATH.

Simple problems like the n -puzzle and Instant Insanity, as well as real problems, like airline scheduling, are only solvable in practice when:

- there exists a succinct problem description, and
- there exists a search technique that can find acceptable solutions without expanding the entire implicitly defined space.

We've already described a way to construct succinct problem descriptions as state space search. It remains to find efficient algorithms that can search the spaces that are defined in that way. For many problems, if we want an optimal solution and we have no additional information about how to find one, we are stuck. But for many problems, additional information is available. All we have to do is to find a way to exploit it.

A **heuristic**  is a rule of thumb. It is a technique that, while not necessarily guaranteed to work exactly all of the time, is useful as a problem-solving tool. The word "heuristic" comes from the Greek word εὐρίσκ-ειν (*heuriskein*), meaning "to find" or "to discover", which is also the root of the word "eureka", derived from Archimedes' reputed exclamation, *heurika* (meaning "I have found"), spoken when he had just discovered a method for determining the purity of gold. Heuristics typically work because they exploit relevant knowledge about the problem that they are being used to solve.

A **heuristic search algorithm** is a search algorithm that exploits knowledge of its problem space to help it find an acceptable solution efficiently. One way to encode that knowledge is in the operators that are supplied to the program. For example, instead of defining operators that correspond to all the legal moves in a problem space, we might define only operators that correspond to generally "sensible" moves. Another very useful way is to define a **heuristic function** whose job is to examine a state and return a measure of how "desirable" it is. That score can then be used by the search algorithm as it chooses which states to explore next. It is sometimes useful to define heuristic functions that assign high scores to states that merit further exploration. In other cases, it is useful to define heuristic functions that measure cost. For example, we might assign to a state a score that estimates the cost of getting from that state to a goal. When we do this, we assign low scores to the states that most merit further consideration.

30.3.2 The A* Algorithm

In this section, we will describe one very general and effective heuristic search algorithm. The **A* algorithm** finds the cheapest path from a start state to a goal state in a succinctly described state space. It exploits a version of **best-first search** in which a heuristic function that evaluates states as they are generated guides the algorithm so that it looks first in the part of the space that is most likely to contain the desired solution.

The A* algorithm is widely used to plan routes for the agents in video games. © 790.

Because what we are trying to do is to find a cheapest path, the score we would like to be able to compute, for any state n is:

$$f^*(n) = \text{cost of getting from the start state to a goal state via a path that goes through } n.$$

We can break $f^*(n)$ into two components:

$$f^*(n) = g^*(n) + h^*(n), \text{ where:}$$

- $g^*(n)$ is the cost of getting from the start state to n , and
- $h^*(n)$ is the cost of getting the rest of way, i.e., the cost of getting from n to a goal.

If we have generated the state n , then we know the cost of at least one way of getting to it. So we have an estimate of $g^*(n)$. But we don't know $h^*(n)$. If, however, we have information about the problem that allows us to estimate $h^*(n)$, we can use it. We'll denote an estimate of a function by omitting the $*$ symbol. So we have:

$$f(n) = g(n) + h(n).$$

The function $f(n)$ will be used to guide the search process. The function $h(n)$ will evaluate a state and return an estimate of the cost of getting from it to a goal.

In the rest of this discussion, we will assume two things:

- There is some positive number c such that all operator costs are at least c . We make this assumption because, if negative costs are allowed, there may be no cheapest path from the start state to a goal. It is possible, in that case, that any path could be made cheaper by repeating some negative cost operator one more time. And if costs can keep getting smaller and smaller, then the cheapest path to a goal might be one with an infinite number of steps. No procedure that halts will ever be able to output such a path.
- Every state has a finite number of successor states.

The most straightforward version of the A^* algorithm conducts a search through a tree of possible paths. In this version, we ignore the possibility that the same state might be generated along several paths. If it is, it will be explored several times. We'll present this version first. Then we'll consider a graph-based version of the technique. In this second algorithm, we will check, when a state is generated, to see if it has been generated before. If it has, we will collapse the paths. The second version is more complex to state, but it may be substantially more efficient at solving problems in which it is likely that the same state could be reached in many different ways.

To see the difference, consider the partial search shown in Figure 30.1. Suppose that, given the search as shown in (a), the next thing that happens is that state C is considered, its successors are generated, and one of its successors is the state labeled E . The tree-search version of A^* won't notice that E has already been generated another way. It will simply build a new search tree node, as shown in (b), that happens to correspond to the same state as E . If it decides that E is a good state to continue working from, it may explore the entire subtree under E twice, once for E and once for E' . On the other hand, the graph-search version of A^* will notice that it has generated E before. It will build the search graph shown as (c).

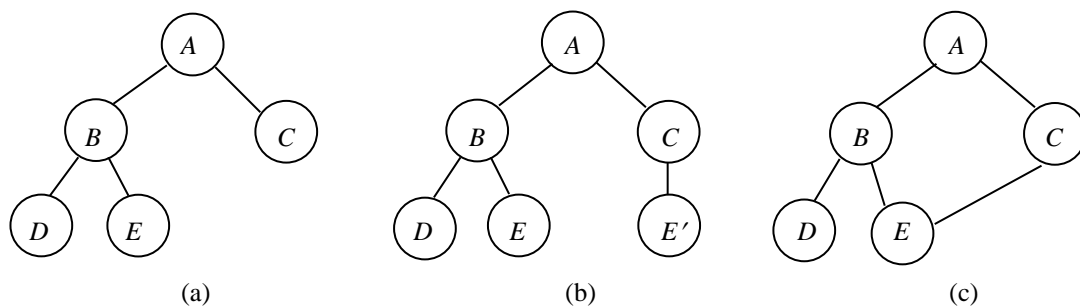


Figure 30.1 Tree search versus graph search

A^* is a best-first search algorithm. So it proceeds, at each step, by generating the successors of the state that looks most promising as a way to get cheaply to a goal. To see how it works, consider the search shown in Figure 30.2. State A is the start state. It is expanded (i.e., its successors are generated) first. In this example, shown in (a), it has two successors: B , which costs 1 to generate, and C , which costs 3 to generate. Let's say that the value of $h(B)$ is 3, so $f(B) = g(B) + h(B) = 1+3 = 4$. Similarly, if $h(C)$ is 2, then $f(C) = 3+2 = 5$. The expression $(g + h)$ for each state is shown directly under it.

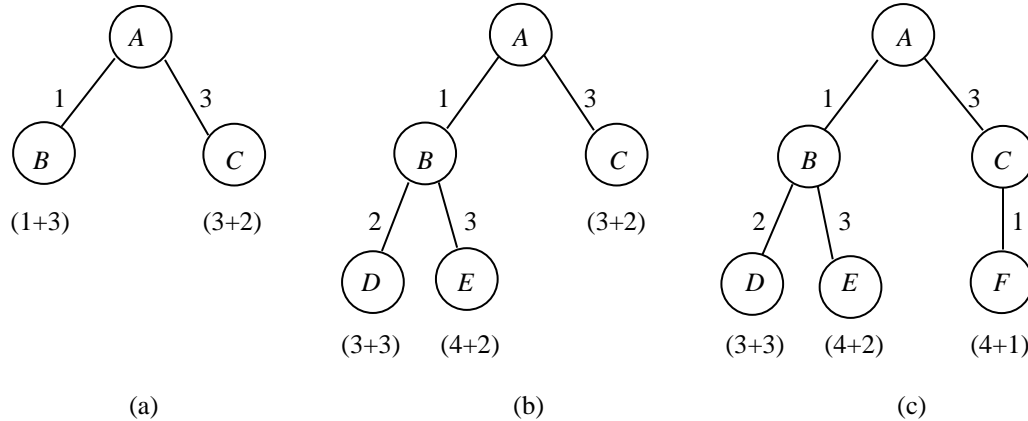


Figure 30.2 Best-first search

A^* maintains a set, called OPEN, of the nodes (corresponding to states) that have been generated but not yet expanded. So OPEN is now $\{B, C\}$. A^* chooses to expand next the element of OPEN that has the lowest f value. That element is B . So B 's successors are generated, producing the search tree shown in (b). Notice that the cost of getting to D is $1 + 2$; the cost of getting to E is $1 + 3$. OPEN is now $\{C, D, E\}$. The state with the lowest f value is C , so it is expanded next. Suppose it has one successor, F . Then the search tree is as shown in (c). F will be expanded next, and so forth, until a goal (a state with an h value of 0) is expanded.

Note two things about the process that we have just described:

- If the subtree under B had remained promising, none of the subtree under C would have been generated. If the subtree under C remains promising, no more of the subtree under B will be generated. If C does turn out to be on the shortest path to a goal, we wasted some time exploring the subtree under B because $h(B)$ underestimated the cost of getting to a goal from B . In so doing, it made B look more promising than it was. The better h is at estimating the true cost of getting to a goal, the more efficient A^* will be.
- The search process cannot stop as soon as a goal state is generated. Goal states have h values of 0. But a goal state may have a high value of $f = g + h$ if the path to it was expensive. If we want to guarantee to find the shortest path to a goal, the search process must continue until a goal state is chosen for expansion (on the basis of having the lowest total f value). To see why this is so, return to the situation shown above as (c). Suppose that F has a single successor G , it costs 8 to go from F to G , and G is a goal state. Then we have $f(G) = g(G) + h(G) = 12 + 0 = 12$. If the search process quits now, it has found a path of cost 12. But, given what we know, it is possible that either D or E could lead to a cheaper path. To see whether or not one of them does, we must expand them until all of their successors have f values of 12 or more.

The algorithm A^* -tree, which we state next, implements the process that we just described. We'll state the algorithm in terms of nodes in a search tree. Each node corresponds to a state in the problem space.

A^* -tree(P : state space search problem) =

1. Start with OPEN containing only the node corresponding to P 's start state. Set that node's g value to 0, its h value to whatever it is, and its f value to $0 + h = h$.
2. Until an answer is found or there are no nodes left in OPEN do:
 - 2.1. If there are no nodes left in OPEN, return Failure. There is no path from the start state to a goal state.
 - 2.2. Choose from OPEN a node such that no other node has a lower f value. Call the chosen node BESTNODE. Remove it from OPEN.

- 2.3. If *BESTNODE* is a goal node, halt and return the path from the initial node to *BESTNODE*.
- 2.4. Generate the successors of *BESTNODE*. For each of them do:
 - Compute f , g , and h , and add the node to *OPEN*.

If there exist any paths from the start state to a goal state, *A*-tree* will find one of them. So we'll say that *A*-tree* is *complete*.

But we can make an even stronger claim: We'll say that $h(n)$ is *admissible* iff it never overestimates the true cost $h^*(n)$ of getting to a goal from n . If h is admissible, then *A*-tree* finds an optimal (i.e., cheapest path). To see why this is so, consider the role of h :

- If h always returns 0, it offers no information. *A*-tree* will choose *BESTNODE* based only on the computed cost of reaching it. So it is guaranteed to find a path with a lowest cost. If, in addition to h being 0, all operator costs are the same, then *A*-tree* becomes breadth-first search.
- If $h(n)$ always returns $h^*(n)$, i.e., the exactly correct cost of getting to a goal from n , then *A*-tree* will walk directly down an optimal path and return it.
- If $h(n)$ overestimates $h^*(n)$, then it effectively “hides” a path that might turn out to be the cheapest. To see how this could happen, consider the search trees shown in Figure 30.3. After reaching the situation shown in (b), *A*-tree* will halt and return the path (with cost 5) from *A* to *B* to *D*. But suppose that there is an operator with a cost of 1 that can be applied to *C* to produce a goal. That would produce a path of cost 4. *A*-tree* will never find that path because the h estimate of 12 blocked it from being considered.

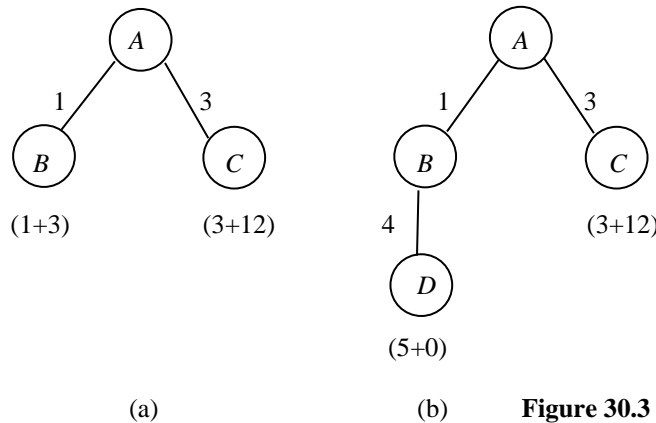


Figure 30.3 What happens if h overestimates h^*

- But if $h(n)$ errs in the other direction and underestimates the true cost of getting to a goal from n , its error will be discovered when the observed cost of the path exceeds the estimated cost without a goal being found. When that happens, *A*-tree* will switch to a cheaper path if there is one.

So the only way that *A*-tree* can find and return a path that is more expensive than some other path it could have found is the case in which h overestimates the true cost h^* .

Some simple heuristic functions are always admissible. For example, if the true cost of getting between points *A* and *B* is the distance between them along roads in a plane, then Euclidean distance (the length of a straight line between two points) is admissible. And, of course, the heuristic function that simply returns 0 is always admissible. But, for some problems, it may be hard to find a heuristic function that is informative but never runs the risk of overestimating true costs. In those cases, the following further observation is important. We'll call it the *graceful decay of admissibility*: If h rarely overestimates h^* by more than δ , then *A*-tree* will rarely find a solution whose cost is more

than δ greater than the cost of the optimal solution. So, as a practical matter, A^* -tree will find very good paths unless h makes large errors of overestimation.

So we have that A^* -tree is optimal in one sense: it finds the best solutions. Search algorithms that are optimal in this sense are called *admissible*. So A^* -tree is admissible. But is its own performance optimal or might it be possible to find cheapest paths by exploring a smaller number of nodes? The answer is that A^* -tree is not optimal in this sense. The reason is that it may explore identical subtrees more than once. As we suggested above, the way to fix this problem is to let it search a state graph rather than a state tree.

We'll give the name A^* to the version of A^* -tree that searches a graph. We present it next. A^* differs from A^* -tree in the following ways:

- A^* exploits two sets of nodes: *OPEN*, which functions as in A^* -tree and contains those nodes that have been generated but not expanded, and *CLOSED*, which contains those nodes that have already been expanded.
- Both A^* -tree and A^* must be able to trace backward from a goal so that they can return the path that they find. A^* -tree can do that trivially by simply storing bi-directional pointers as it builds its search tree. But A^* searches a graph. So it must explicitly record, at each node, the best way of getting to that node from the start node. Whenever a new path to node n is found, its backward pointer may change.
- Suppose that, in A^* , a new and cheaper path to node n is found after node n has been expanded. Clearly n 's g value changes. But the cheaper path to n may also mean a cheaper path to n 's successors. So it may be necessary to revisit them and update their backward pointers and their g values.

$A^*(P: \text{state space search problem}) =$

1. Start with *OPEN* containing only the node corresponding to P 's start state. Set that node's g value to 0, its h value to whatever it is, and its f value to $0 + h = h$. Set *CLOSED* to the empty set.
2. Until an answer is found or there are no nodes left in *OPEN* do:
 - 2.1. If there are no nodes left in *OPEN*, return *Failure*. There is no path from the start state to a goal state.
 - 2.2. Choose from *OPEN* a node such that no other node has a lower f value. Call the chosen node *BESTNODE*. Remove it from *OPEN*. Place it in *CLOSED*.
 - 2.3. If *BESTNODE* is a goal node, halt and return the path from the initial node to *BESTNODE*.
 - 2.4. Generate the successors of *BESTNODE*. But do not add them to the search graph until we have checked to see if any of them correspond to states that have already been generated. For each *SUCCESSOR* do:
 - 2.4.1. Set *SUCCESSOR* to point back to *BESTNODE*.
 - 2.4.2. Compute $g(\text{SUCCESSOR}) = g(\text{BESTNODE}) +$ the cost of getting from *BESTNODE* to *SUCCESSOR*.
 - 2.4.3. See if *SUCCESSOR* corresponds to the same state as any node in *OPEN*. If so, call that node *OLD*. Since this node already exists in the graph, we can throw *SUCCESSOR* away and add *OLD* to the list of *BESTNODE*'s successors. But first we must decide whether *OLD*'s backward pointer should be reset to point to *BESTNODE*. It should be if the path we have just found to *SUCCESSOR* is cheaper than the current best path to *OLD* (since *SUCCESSOR* and *OLD* are really the same node). So compare the g values of *OLD* and *SUCCESSOR*. If *OLD* is cheaper (or just as cheap), then we need do nothing. If *SUCCESSOR* is cheaper, then reset *OLD*'s backward point to *BESTNODE*, record the new cheaper path in $g(\text{OLD})$, and update $f(\text{OLD})$.
 - 2.4.4. If *SUCCESSOR* was not in *OPEN*, see if it is in *CLOSED*. If so, call the node in *CLOSED* *OLD* and add *OLD* to the list of *BESTNODE*'s successors. Check to see if the new path or the old path is better just as in step 2.4.3, and set the backward pointer and g and f values appropriately. If we have just found a better path to *OLD*, we must propagate the improvement to *OLD*'s successors. This is a bit tricky. *OLD* points to its successors. Each successor in turn points to its successors, and so forth, until each branch terminates with a node that either is still in *OPEN* or has no successors. So, to propagate the new cost downward, do a depth-first traversal of the search graph, starting at *OLD*, and changing each node's g value (and thus also its f value), terminating each branch when it reaches either a node with no successors or a node to which an equivalent or better path had already been

found. Note that this condition doesn't just allow the propagation to stop as soon as the new path ceases to make a difference to any further node's cost. It also guarantees that the algorithm will terminate even if there are cycles in the graph. If there is a cycle, then the second time that a given node is visited, the path will be no better than the first time and so propagation will stop.

2.4.5. If *SUCCESSOR* was not already in either *OPEN* or *CLOSED*, then put it in *OPEN*, and add it to the list of *BESTNODE*'s successors. Compute:

$$f(\text{SUCCESSOR}) = g(\text{SUCCESSOR}) + h(\text{SUCCESSOR}).$$

A^* , like A^* -tree, is complete; it will find a path if one exists. If h is admissible, then A^* will find a shortest path. And the graceful decay of admissibility principle applies to A^* just as it does to A^* -tree.

In addition, we can now say something about the efficiency with which A^* finds a shortest path. Let $c(n_1, n_2)$ be the cost of getting from n_1 to n_2 . We'll say that $h(n)$ is **monotonic** iff, whenever n_2 is a successor of n_1 (meaning that it can be derived from n_1 in exactly one move), $h(n_1) \leq c(n_1, n_2) + h(n_2)$. If f is monotonic, then A^* is optimal in the sense that no other search algorithm that uses the same heuristic function and that is guaranteed to find a cheapest path will do so by examining fewer nodes than A^* does. In particular, in this case it can be shown that A^* will never need to reexamine a node once it goes on *CLOSED*. So it is possible to skip step 2.4.4.

Unfortunately, even with these claims, A^* may not be good enough. Depending on the shape of the state space and the accuracy of h , it may still be necessary to examine a number of nodes that grows exponentially in the length of the cheapest path. However, if the maximum error that h may make is small, the number of nodes that must be examined grows only polynomially in the length of the cheapest path. More specifically, polynomial growth is assured if:

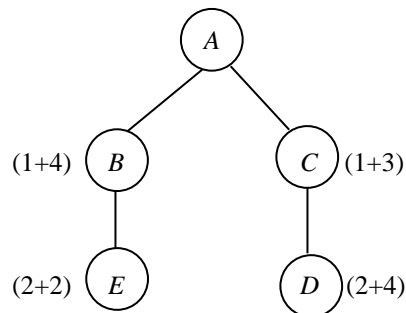
$$|h^*(n) - h(n)| \in \mathcal{O}(\log h(n)).$$

A^* is just one member of a large family of heuristic search algorithms. See [Pearl 1984] or [Russell and Norvig 2002] for a discussion of others. For example, A^* , like its cousin, breadth-first search, uses a lot of space. There exist other algorithms use less.


Generalized chess is provably intractable (since it is EXPTIME-complete). Even the standard chess board is large enough that it isn't possible to search a complete game tree to find a winning move. Yet champion chess programs exist. They, along with programs that play other classic games like checkers and Go, exploit a heuristic search algorithm called *minimax*, which we describe in § 785.

30.4 Exercises

- 1) In Exercise 28.23) we defined a cut in a graph, the size of a cut and a bisection. Let G be a graph with $2v$ vertices and m edges. Describe a randomized, polynomial-time algorithm that, on input G , outputs a cut of G with expected size at least $mv/(2v-1)$. (Hint: analyze the algorithm that takes a random bisection as its cut.)
- 2) Suppose that the A^* algorithm has generated the following tree so far:



Assume that the nodes were generated in the order, A, B, C, D, E . The expression (g, h) associated with each node gives the values of the functions g and h at that node.

- a) What node will be expanded at the next step?
 - b) Can it be guaranteed that A^* , using the heuristic function h that it is using, will find an optimal solution? Why or why not?
- 3) Simple puzzles offer a way to explore the behavior of search algorithms such as A^* , as well as to experiment with a variety of heuristic functions. Pick one (for example the 15-puzzle of Example 4.8 or see ) and use A^* to solve it. Can you find an admissible heuristic function that is effective at pruning the search space?

31 Summary and References

In Part IV, we saw that some problems are uncomputable in principle: for example, no effort on the part of the engineers of the world can make the halting problem solvable. In Part V, we've considered only problems that *are* computable in principle. But we've seen that while some are computable in practice, others aren't, at least with the techniques available to us today. In the years since the theory of NP-completeness was first described, a substantial body of work has increased our understanding of the ways in which some problems appear to require more computational resources than others. But that work has left many questions unanswered. While it is known that not all of the complexity classes that we have considered can collapse, it is unknown whether some of the most important of them can. In particular, we are left with the Millennium Problem \square , "Does $P = NP$?"

References

The traveling salesman problem, along with other related combinatorial problems, has been studied by mathematicians since the nineteenth century. The problem has been given a variety of names. For example, Karl Menger [Menger 1932] called it "Das Botenproblem" or the Messenger Problem, so named, he said "since this problem is encountered by every postal messenger, as well as by many travelers". The application to a survey of Bengal farmers was described in [Mahalanobis 1940]. Julia Robinson appears to have been the first to publish a discussion of the problem with the name traveling salesman problem [Robinson 1949]. The use of linear programming to solve the TSP was introduced in [Dantzig, Fulkerson, and Johnson 1954]. See [Cormen, Leiserson, Rivest and Stein 2001] for a description of a straightforward minimum spanning tree-based algorithm that, when the triangle inequality holds, finds a solution to the TSP whose distance is no more than twice the distance of an optimal solution. The existence of an algorithm that tightens that bound to 1.5 was proved in [Christofides 1976]. For a comprehensive discussion of the TSP, see [Lawler, Lenstra, Rinnooy Kan and Shmoys 1985].

The Complexity Zoo \square was created and is maintained by Scott Aaronson.

Strassen's algorithm for matrix multiplication was presented in [Strassen 1969]. The Coppersmith-Winograd algorithm was presented in [Coppersmith and Winograd 1990].

The Knuth-Morris-Pratt string search algorithm was discovered by Don Knuth and Vaughan Pratt and, independently, by Jim Morris. They published it jointly as [Knuth, Morris and Pratt 1977].

Kruskal's algorithm was originally described in [Kruskal 1956]. For a good discussion of it, along with Prim's algorithm, an alternative for finding minimum spanning trees, see [Cormen, Leiserson, Rivest and Stein 2001].

Shor's algorithm for factoring using quantum computing was described in [Shor 1994].

[Hartmanis and Stearns 1965] introduced the idea of defining language complexity classes based on the running time, stated in terms of the length of the input, of deciding Turing machines. In 1993, Hartmanis and Stearns won the Turing Award for this work. The citation read, "In recognition of their seminal paper which established the foundations for the field of computational complexity theory."

The notion of NP-completeness and the proof of Theorem 28.16 (now commonly called the Cook-Levin Theorem) were introduced in [Cook 1971] and [Levin 1973]. In 1982, Cook won the Turing Award for this work. The citation read, "For his advancement of our understanding of the complexity of computation in a significant and profound way. His seminal paper, "The Complexity of Theorem Proving Procedures," presented at the 1971 ACM SIGACT Symposium on the Theory of Computing, laid the foundations for the theory of NP-Completeness. The ensuing exploration of the boundaries and nature of NP-complete class of problems has been one of the most active and important research activities in computer science for the last decade."

[Karp 1972] showed that SAT was not the only NP-complete language. That paper presents a landmark list \square of 21 other NP-complete problems. Karp won the 1985 Turing Award, "For his continuing contributions to the theory of algorithms including the development of efficient algorithms for network flow and other combinatorial optimization

problems, the identification of polynomial-time computability with the intuitive notion of algorithmic efficiency, and, most notably, contributions to the theory of NP-completeness. Karp introduced the now standard methodology for proving problems to be NP-complete which has led to the identification of many theoretical and practical problems as being computationally difficult.”

For a comprehensive discussion of NP-completeness, see [Garey and Johnson 1979]. Also there you will find a well-organized list of NP-hard problems, along with what is known of their complexity class and the references to the appropriate results. The list includes many problems that are known to be PSPACE-complete or EXPTIME-complete. The proof we present of Theorem 28.24 (Ladner’s Theorem) is based on the one presented in [Garey and Johnson 1979]. Any NP-completeness claims that are made in this book and for which other references are not provided are discussed in [Garey and Johnson 1979]; further references are given there.

The literature on individual NP-complete languages includes:

- HAMILTONIAN-CIRCUIT: The proof we gave of Theorem 28.22 is patterned after the one given in [Hopcroft, Motwani and Ullman 2001].
- SUDOKU: A proof that SUDOKU is NP-complete was given in [Yato and Seta 2002].

For a description of the first proof of the 4-Color Theorem, see [Appel and Haken 1977].

The proof we present for Theorem 28.26 ($NP = co-NP$ iff there exists some language L such that L is NP-complete and $\neg L$ is also in NP) was taken from [Hopcroft, Motwani and Ullman 2001].

The proof we present of Theorem 28.27 (the Deterministic Time Hierarchy Theorem) was modeled closely after the one in [Sipser2006].

The statement of the Linear Speedup Theorem that we give as Theorem 37.3, as well as its proof, are taken from [Sudkamp 1998].

Savitch’s Theorem (Theorem 29.2) was stated and proved in [Savitch 1970]. The proof that we present of Theorem 29.6 is modeled on the one presented in [Sipser 2006].

Quicksort was first described in [Hoare 1961]. An early version of the Miller-Rabin primality test was described in [Miller 1976] and later modified in [Rabin 1980].

For a more comprehensive discussion of heuristic search, including the A* algorithm and its cousins, see [Russell and Norvig 2002]. The A* algorithm was introduced in [Hart, Nilsson and Raphael 1968]. Its description was amended slightly in [Hart, Nilsson and Raphael 1972].

