13.1.1 Defining the Complexity Classes P and NP

By Lemma 13.1, we know that, for the problems discussed in this book, such as graph problems, text processing, or sorting, our previous polynomial-time algorithms translate into polynomial-time algorithms in the bit model. Even the repeated squaring algorithm (Section 10.1.4) for computing powers of an integer x runs in a polynomial number of bit operations if we apply it to raise x to a number that is represented using $O(\log n)$ bits. Thus, the notion of polynomial time is quite useful as a measure for tractability.

Moreover, the class of polynomials is closed under addition, multiplication, and composition. That is, if $p(n)$ and $q(n)$ are polynomials, then so are $p(n) + q(n)$, $p(n) \cdot q(n)$, and $p(q(n))$. Thus, we can combine or compose polynomial-time algorithms to construct new polynomial-time algorithms.

Decision Problems

To simplify our discussion, let us restrict our attention for the time being to ***decision problems***, that is, to computational problems for which the intended output is either "yes" or "no." In other words, a decision problem's output is a single bit, which is either 0 or 1. For example, each of the following are decision problems:

- Given a string T and a string P, does P appear as a substring of T?
- Given two sets S and T, do S and T contain the same set of elements?
- Given a graph G with integer weights on its edges, and an integer k, does G have a minimum spanning tree of weight at most k?

In fact, the last problem illustrates how we can often turn an ***optimization problem***, where we are trying to minimize or maximize some value, into a decision problem. Namely, we can introduce a parameter k and ask if the optimal value for the optimization problem is at most or at least k. Note that if we can show that a decision problem is hard, then its related optimization version must also be hard.

Problems and Languages

We say that an algorithm A ***accepts*** an input string x if A outputs "yes" on input x. Thus, we can view a ***decision problem*** as actually being just a set L of strings—the strings that should be accepted by an algorithm that correctly solves the problem. Indeed, we used the letter "L" to denote a decision problem, because a set of strings is often referred to as a ***language***. We can extend this language-based viewpoint further to say that an algorithm A ***accepts*** a language L if A outputs "yes" for each x in L and outputs "no" otherwise. Throughout this chapter, we assume that if x is in an improper syntax, then an algorithm given x will output "no." (Note: Some texts also allow for the possibility of A going into an infinite loop and never outputting anything on some inputs, but we are restricting our attention in this book to algorithms, that is, computations that terminate after a finite number of steps.)

The Complexity Class P

The ***complexity class P*** is the set of all decision problems (or languages) L that can be accepted in worst-case polynomial time. That is, there is an algorithm A that, if $x \in L$, then on input x, A outputs "yes" in $p(n)$ time, where n is the size of x and $p(n)$ is a polynomial. Note that the definition of ***P*** doesn't say anything about the running time for rejecting an input—when an algorithm A outputs "no." Such cases refer to the ***complement*** of a language L, which consists of all binary strings that are not in L. Still, given an algorithm A that accepts a language L in polynomial time, $p(n)$, we can easily construct a polynomial-time algorithm that accepts the complement of L. In particular, given an input x, we can construct a complement algorithm B that simply runs A for $p(n)$ steps, where n is the size of x, terminating A if it attempts to run more than $p(n)$ steps. If A outputs "yes," then B outputs "no." Likewise, if A outputs "no" or if A runs for at least $p(n)$ steps without outputting anything, then B outputs "yes." In either case, the complement algorithm B runs in polynomial time. Therefore, if a language L, representing some decision problem, is in ***P***, then the complement of L is also in ***P***.

The Complexity Class NP

The ***complexity class NP*** is defined to include the complexity class ***P*** but allow for the inclusion of languages that may not be in ***P***. Specifically, with ***NP*** problems, we allow algorithms to perform an additional operation:

- choose(b): this operation chooses in a nondeterministic way a bit (that is, a value that is either 0 or 1) and assigns it to b.

When an algorithm A uses the choose primitive operation, then we say A is ***nondeterministic***. We state that an algorithm A ***nondeterministically accepts*** a string x if there exists a set of outcomes to the choose calls that A could make on input x such that A would ultimately output "yes." In other words, it is as if we consider all possible outcomes to choose calls and only select those that lead to acceptance if there is such a set of outcomes. Note this is not the same as random choices.

The complexity class ***NP*** is the set of decision problems (or languages) L that can be nondeterministically accepted in polynomial time. That is, there is a nondeterministic algorithm A that, if $x \in L$, then, on input x, there is a set of outcomes to the choose calls in A so that it outputs "yes" in $p(n)$ time, where n is the size of x and $p(n)$ is a polynomial. Note that the definition of ***NP*** does not address the running time for a rejection. Indeed, we allow for an algorithm A accepting a language L in polynomial time $p(n)$ to take much more than $p(n)$ steps when A outputs "no." Moreover, because nondeterministic acceptance could involve a polynomial number of calls to the choose method, if a language L is in ***NP***, the complement of L is not necessarily also in ***NP***. Indeed, there is a complexity class, called **co-*NP***, that consists of all languages whose complement is in ***NP***, and many researchers believe **co-*NP*** $\neq$ ***NP***.

An Alternate Definition of NP

There is actually another way to define the complexity class *NP*, which might be more intuitive for some readers. This alternate definition of *NP* is based on deterministic verification, instead of nondeterministic acceptance. We say that a language L can be ***verified*** by an algorithm A if, given any string x in L as input, there is another string y such that A outputs "yes" on input $z = x + y$, where we use the symbol "+" to denote concatenation. The string y is called a ***certificate*** for membership in L, for it helps us certify that x is indeed in L. Note that we make no claims about verifying when a string is not in L.

This notion of verification allows us to give an alternate definition of the complexity class *NP*. Namely, we can define *NP* to be the set of all languages L, defining decision problems, such that L can be verified in polynomial time. That is, there is a (deterministic) algorithm A that, for any x in L, verifies using some certificate y that x is indeed in L in polynomial time, $p(n)$, including the time A takes to read its input $z = x + y$, where n is the size of x. Note that this definition implies that the size of y is less than $p(n)$. As the following theorem shows, this verification-based definition of *NP* is equivalent to the nondeterminism-based definition given above.

Theorem 13.2: *A language L can be (deterministically) verified in polynomial time if and only if L can be nondeterministically accepted in polynomial time.*

Proof: Let us consider each possibility. Suppose first that L can be verified in polynomial time. That is, there is a deterministic algorithm A (making no use of choose calls) that can verify in polynomial time $p(n)$ that a string x is in L when given a polynomial-length certificate y. Therefore, we can construct a nondeterministic algorithm B that takes the string x as input and calls the choose method to assign the value of each bit in y. After B has constructed a string $z = x + y$, it then calls A to verify that $x \in L$ given the certificate y. If there exists a certificate y such that A accepts z, then there is clearly a set of nondeterministic choices for B that result in B outputting "yes" itself. In addition, B will run in $O(p(n))$ steps.

Next, suppose that L can be nondeterministically accepted in polynomial time. That is, there is a nondeterministic algorithm A that, given a string x in L, performs $p(n)$ steps, which may include choose steps, such that, for some sequence of outcomes to these choose steps, A will output "yes." There is a deterministic verification algorithm B that, given x in L, uses as its certificate y the ordered concatenation of all the outcomes to choose calls that A makes on input x in order to ultimately output "yes." Since A runs in $p(n)$ steps, where n is the size of x, the algorithm B will also run in $O(p(n))$ steps given input $z = x + y$. ∎

The practical implication of this theorem is that, since both definitions of *NP* are equivalent, we can use either one for showing that a problem is in *NP*.

The P = NP Question

Computer scientists do not know for certain whether $P = NP$ or not. Indeed, researchers don't even know for sure whether or not $P = NP \cap$ co-*NP*. Still, the vast majority of researchers believe that *P* is different than both *NP* and co-*NP*, as well as their intersection. In fact, the problems we discuss next are examples of problems in *NP* that many believe are not in *P*.

13.1.2 Some Interesting Problems in NP

Another way of interpreting Theorem 13.2 is that it implies we can always structure a nondeterministic algorithm so that all of its choose steps are performed first and the rest of the algorithm is just a verification. We illustrate with several examples in this subsection this approach of showing interesting decision problems to be in *NP*. Our first example is for a graph problem.

HAMILTONIAN-CYCLE is the problem that takes a graph G as input and asks whether there is a simple cycle in G that visits each vertex of G exactly once and then returns to its starting vertex. Such a cycle is called an Hamiltonian cycle of G.

Lemma 13.3: HAMILTONIAN-CYCLE *is in NP.*

Proof: Let us define a nondeterministic algorithm A that takes, as input, a graph G encoded as an adjacency list in binary notation, with the vertices numbered 1 to N. We define A to first iteratively call the choose method to determine a sequence S of $N + 1$ numbers from 1 to N. Then, we have A check that each number from 1 to N appears exactly once in S (for example, by sorting S), except for the first and last numbers in S, which should be the same. Then, we verify that the sequence S defines a cycle of vertices and edges in G. A binary encoding of the sequence S is clearly of size at most n, where n is the size of the input. Moreover, both of the checks made on the sequence S can be done in polynomial time in n.

Observe that if there is a cycle in G that visits each vertex of G exactly once, returning to its starting vertex, then there is a sequence S for which A will output "yes." Likewise, if A outputs "yes," then it has found a cycle in G that visits each vertex of G exactly once, returning to its starting point. That is, A nondeterministically accepts the language HAMILTONIAN-CYCLE. In other words, HAMILTONIAN-CYCLE is in *NP*. ∎

Our next example is a problem related to circuit design testing. A ***Boolean circuit*** is a directed graph where each node, called a ***logic gate***, corresponds to a simple Boolean function, AND, OR, or NOT. The incoming edges for a logic gate correspond to inputs for its Boolean function and the outgoing edges correspond to outputs, which will all be the same value, of course, for that gate. (See Figure 13.1.) Vertices with no incoming edges are ***input*** nodes and a vertex with no outgoing edges is an ***output*** node.

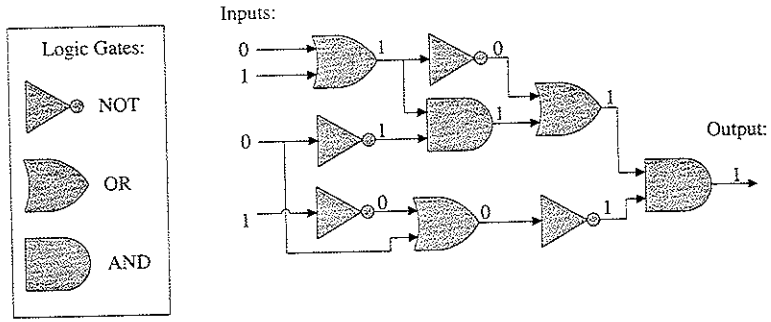


Figure 13.1: An example Boolean circuit.

CIRCUIT-SAT is the problem that takes as input a Boolean circuit with a single output node, and asks whether there is an assignment of values to the circuit's inputs so that its output value is "1." Such an assignment of values is called a ***satisfying assignment***.

Lemma 13.4: CIRCUIT-SAT *is in NP.*

Proof: We construct a nondeterministic algorithm for accepting CIRCUIT-SAT in polynomial time. We first use the choose method to "guess" the values of the input nodes as well as the output value of each logic gate. Then, we simply visit each logic gate g in C, that is, each vertex with at least one incoming edge. We then check that the "guessed" value for the output of g is in fact the correct value for g's Boolean function, be it an AND, OR, or NOT, based on the given values for the inputs for g. This evaluation process can easily be performed in polynomial time. If any check for a gate fails, or if the "guessed" value for the output is 0, then we output "no." If, on the other hand, the check for every gate succeeds and the output is "1," the algorithm outputs "yes." Thus, if there is indeed a satisfying assignment of input values for C, then there is a possible collection of outcomes to the choose statements so that the algorithm will output "yes" in polynomial time. Likewise, if there is a collection of outcomes to the choose statements so that the algorithm outputs "yes" in polynomial time algorithm, there must be a satisfying assignment of input values for C. Therefore, CIRCUIT-SAT is in *NP*. ∎

The next example illustrates how a decision version of an optimization problem can be shown to be in *NP*. Given a graph G, a ***vertex cover*** for G is a subset C of vertices such that, for every edge (v,w) of G, $v \in C$ or $w \in C$ (possibly both). The optimization goal is to find as small a vertex cover for G as possible.

VERTEX-COVER is the decision problem that takes a graph G and an integer k as input, and asks whether there is a vertex cover for G containing at most k vertices.

Lemma 13.5: VERTEX-COVER *is in NP.*

Proof: Suppose we are given an integer k and a graph G, with the vertices of G numbered from 1 to N. We can use repeated calls to the choose method to construct a collection C of k numbers that range from 1 to N. As a verification, we insert all the numbers of C into a dictionary and then we examine each of the edges in G to make sure that, for each edge (v,w) in G, v is in C or w is in C. If we ever find an edge with neither of its end-vertices in G, then we output "no." If we run through all the edges of G so that each has an end-vertex in C, then we output "yes." Such a computation clearly runs in polynomial time.

Note that if G has a vertex cover of size at most k, then there is an assignment of numbers to define the collection C so that each edge of G passes our test and our algorithm outputs "yes." Likewise, if our algorithm outputs "yes," then there must be a subset C of the vertices of size at most k, such that C is a vertex cover. Thus, VERTEX-COVER is in *NP*. ∎

Having given some interesting examples of problems in *NP*, let us now turn to the definition of the concept of *NP*-completeness.

13.2 NP-Completeness

The notion of nondeterministic acceptance of a decision problem (or language) is admittedly strange. There is, after all, no conventional computer that can efficiently perform a nondeterministic algorithm with many calls to the choose method. Indeed, to date no one has shown how even an unconventional computer, such as a quantum computer or DNA computer, can efficiently simulate any nondeterministic polynomial-time algorithm using a polynomial amount of resources. Certainly, we can deterministically simulate a nondeterministic algorithm by trying out, one by one, all possible outcomes to the choose statements that the algorithm makes. But this simulation would become an exponential-time computation for any nondeterministic algorithm that makes at least n^ε calls to the choose method, for any fixed constant $\varepsilon > 0$. Indeed, there are hundreds of problems in the complexity class *NP* for which most computer scientists strongly believe there is no conventional deterministic method for solving them in polynomial time.

The usefulness of the complexity class *NP*, therefore, is that it formally captures a host of problems that many believe to be computationally difficult. In fact, there are some problems that are provably at least as hard as every other problem in *NP*, as far as polynomial-time solutions are concerned. This notion of hardness is based on the concept of polynomial-time reducibility, which we now discuss.

13.2.1 Polynomial-Time Reducibility and NP-Hardness

We say that a language L, defining some decision problem, is ***polynomial-time reducible*** to a language M, if there is a function f computable in polynomial time, that takes an input x to L, and transforms it to an input $f(x)$ of M, such that $x \in L$ if and only if $f(x) \in M$. In addition, we use a shorthand notation, saying $L \xrightarrow{\text{poly}} M$ to signify that language L is polynomial-time reducible to language M.

We say that a language M, defining some decision problem, is ***NP-hard*** if every other language L in NP is polynomial-time reducible to M. In more mathematical notation, M is NP-hard, if, for every $L \in NP$, $L \xrightarrow{\text{poly}} M$. If a language M is NP-hard and it is also in the class NP itself, then M is ***NP-complete***. Thus, an NP-complete problem is, in a very formal sense, one of the hardest problems in NP, as far as polynomial-time computability is concerned. For, if anyone ever shows that an NP-complete problem L is solvable in polynomial time, then that immediately implies that every other problem in the entire class NP is solvable in polynomial time. For, in this case, we could accept any other NP language M by reducing it to L and then running the algorithm for L. In other words, if anyone finds a deterministic polynomial-time algorithm for even one NP-complete problem, then $P = NP$.

13.2.2 The Cook-Levin Theorem

At first, it might appear that the definition of NP-completeness is too strong. Still, as the following theorem shows, there is at least one NP-complete problem.

Theorem 13.6 (The Cook-Levin Theorem): CIRCUIT-SAT *is NP-complete.*

Proof: Lemma 13.4 shows that CIRCUIT-SAT is in NP. Thus, we have yet to show this problem is NP-hard. That is, we need to show that every problem in NP is polynomial-time reducible to CIRCUIT-SAT. So, consider a language L, representing some decision problem that is in NP. Since L is in NP, there is a deterministic algorithm D that accepts any x in L in polynomial-time $p(n)$, given a polynomial-sized certificate y, where n is the size of x. The main idea of the proof is to build a large, but polynomial-sized, circuit C that simulates the algorithm D on an input x in such a way that C is satisfiable if and only if there is a certificate y such that D outputs "yes" on input $z = x + y$.

Recall (from Section 1.1.2) that any deterministic algorithm, such as D, can be implemented on a simple computational model (called the Random Access Machine, or RAM) that consists of a CPU and a bank M of addressable memory cells. In our case, the memory M contains the input, x, the certificate, y, the working storage, W, that D needs to perform its computations, and the code for the algorithm D itself. The working storage W for D includes all the registers used for temporary calculations and the stack frames for the procedures that D calls during its execution. The topmost such stack frame in W contains the program counter (PC) that identifies where D currently is in its program execution. Thus, there are no memory cells in the CPU itself. In performing each step of D, the CPU reads the next instruction i, which is pointed to by the PC, and performs the calculation indicated by i, be it a comparison, arithmetic operation, a conditional jump, a step in procedure call, etc., and then updates the PC to point to the next instruction to be performed. Thus, the current state of D is completely characterized by the contents of its memory cells. Moreover, since D accepts an x in L in a polynomial $p(n)$ number of steps, where n is the size of x, then the entire effective collection of its memory cells can be assumed to consist of just $p(n)$ bits. For in $p(n)$ steps, D can access at most $p(n)$ memory cells. Note also that the size of D's code is constant with respect to the sizes of x, y, and even W. We refer to the $p(n)$-sized collection M of memory cells for an execution of D as the ***configuration*** of the algorithm D.

The heart of the reduction of L to CIRCUIT-SAT depends on our constructing a Boolean circuit that simulates the workings of the CPU in our computational model. The details of such a construction are beyond the scope of this book, but it is well known that a CPU can be designed as a Boolean circuit consisting of AND, OR, and NOT gates. Moreover, let us further take for granted that this circuit, including its address unit for connecting to a memory of $p(n)$ bits, can be designed so as to take a configuration of D as input and provide as output the configuration resulting from processing the next computational step. In addition, this simulation circuit, which we will call S, can be constructed so as to consist of at most $cp(n)^2$ AND, OR, and NOT gates, for some constant $c > 0$.

To then simulate the entire $p(n)$ steps of D, we make $p(n)$ copies of S, with the output from one copy serving as the input for the next. (See Figure 13.2.) Part of the input for the first copy of S consists of "hard wired" values for the program for D, the value of x, the initial stack frame (complete with PC pointing to the first instruction of D), and the remaining working storage (initialized to all 0's). The only unspecified true inputs to the first copy of S are the cells of D's configuration for the certificate y. These are the true inputs to our circuit. Likewise, we ignore all the outputs from the final copy of S, except the single output that indicates the answer from D, with "1" for "yes" and "0" for "no." The total size of the circuit C is $O(p(n)^3)$, which of course is still polynomial in the size of x.

Consider an input x that D accepts for some certificate y after $p(n)$ steps. Then there is an assignment of values to the input to C corresponding to y, such that, by having C simulate D on this input and the hard-wired values for x, we will ultimately have C output a "1." Thus, C is satisfiable in this case. Conversely, consider a case when C is satisfiable. Then there is a set of inputs, which correspond to the certificate y, such that C outputs a "1." But, since C exactly simulates the algorithm D, this implies that there is an assignment of values to the certificate y, such that D outputs "yes." Thus, D will verify x in this case. Therefore, D accepts x with certificate y if and only if C is satisfiable. ■

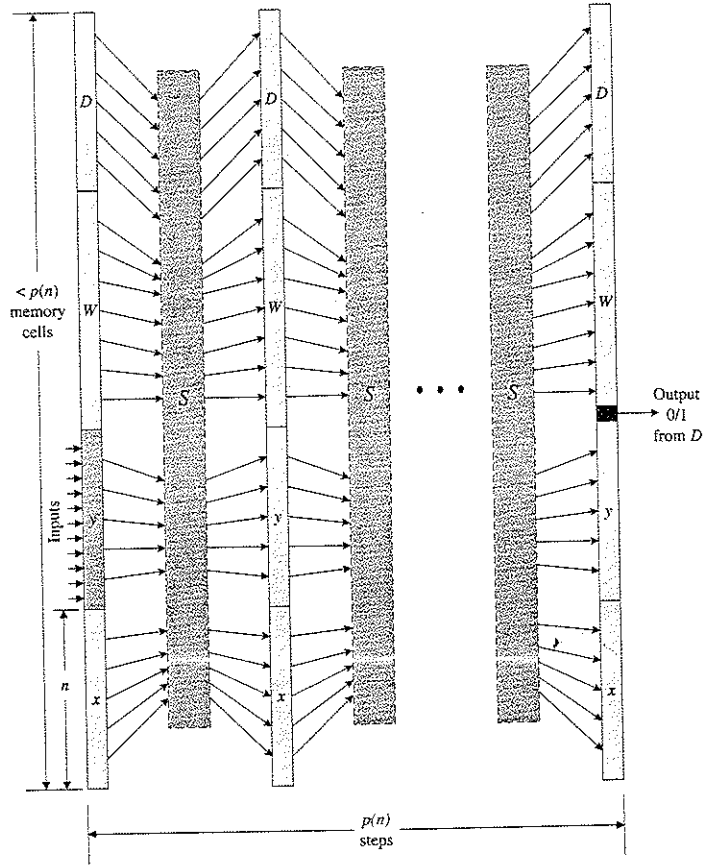


Figure 13.2: An illustration of the circuit used to prove that CIRCUIT-SAT is *NP*-hard. The only true inputs correspond to the certificate, y. The problem instance, x, the working storage, W, and the program code, D, are initially "hard wired" values. The only output is the bit that determines if the algorithm accepts x or not.

13.3 Important NP-Complete Problems

So there is indeed an *NP*-complete problem. But proving this fact was admittedly a tiring exercise, even taking into account the major shortcut we took in assuming the existence of the simulation circuit S. Fortunately, now that we are armed with one problem that is proven to be *NP*-complete "from scratch," we can prove other problems are *NP*-complete using simple polynomial-time reductions. We explore a number of such reductions in this section.

Given just a single *NP*-complete problem, we can now use polynomial-time reducibility to show other problems to be *NP*-complete. In addition, we will make repeated use of the following important lemma about polynomial-time reducibility.

Lemma 13.7: *If $L_1 \xrightarrow{\text{poly}} L_2$ and $L_2 \xrightarrow{\text{poly}} L_3$, then $L_1 \xrightarrow{\text{poly}} L_3$.*

Proof: Since $L_1 \xrightarrow{\text{poly}} L_2$, any instance x for L_1 can be converted in polynomial-time $p(n)$ into an instance $f(x)$ for L_2, such that $x \in L_1$ if and only if $f(x) \in L_2$, where n is the size of x. Likewise, since $L_2 \xrightarrow{\text{poly}} L_3$, any instance y for L_2 can be converted in polynomial-time $q(m)$ into an instance $g(y)$ for L_3, such that $y \in L_2$ if and only if $g(y) \in L_3$, where m is the size of y. Combining these two constructions, any instance x for L_1 can be converted in time $q(k)$ into an instance $g(f(x))$ for L_3, such that $x \in L_1$ if and only if $g(f(x)) \in L_3$, where k is the size of $f(x)$. But, $k \leq p(n)$, since $f(x)$ is constructed in $p(n)$ steps. Thus, $q(k) \leq q(p(n))$. Since the composition of two polynomials always results in another polynomial, this inequality implies that $L_1 \xrightarrow{\text{poly}} L_3$. ■

In this section we establish several important problems to be *NP*-complete, using this lemma. All of the proofs have the same general structure. Given a new problem L, we first prove that L is in *NP*. Then, we reduce a known *NP*-complete problem to L in polynomial time, showing L to be *NP*-hard. Thus, we show L to be in *NP* and also *NP*-hard; hence, L has been shown to be *NP*-complete. (Why not do the reduction in the other direction?) These reductions generally take one of three forms:

- ***Restriction***: This form shows a problem L is *NP*-hard by noting that a known *NP*-complete problem M is actually just a special case of L.
- ***Local replacement***: This forms reduces a known *NP*-complete problem M to L by dividing instances of M and L into "basic units," and then showing how each basic unit of M can be locally converted into a basic unit of L.
- ***Component design***: This form reduces a known *NP*-complete problem M to L by building components for an instance of L that will enforce important structural functions for instances of M. For example, some components might enforce a "choice" while others enforce an "evaluation" function.

The latter of the three above forms tends to be the most difficult to construct; it is the form used, for example, by the proof of the Cook-Levin Theorem (13.6).

In Figure 13.3, we illustrate the problems we prove are *NP*-complete, together with the problems they are reduced from and the technique used in each polynomial-time reduction.

In the remainder of this section we study some important *NP*-complete problems. We treat most of them in pairs, with each pair addressing an important class of problems, including problems involving Boolean formulas, graphs, sets, and numbers. We begin with two problems involving Boolean formulas.

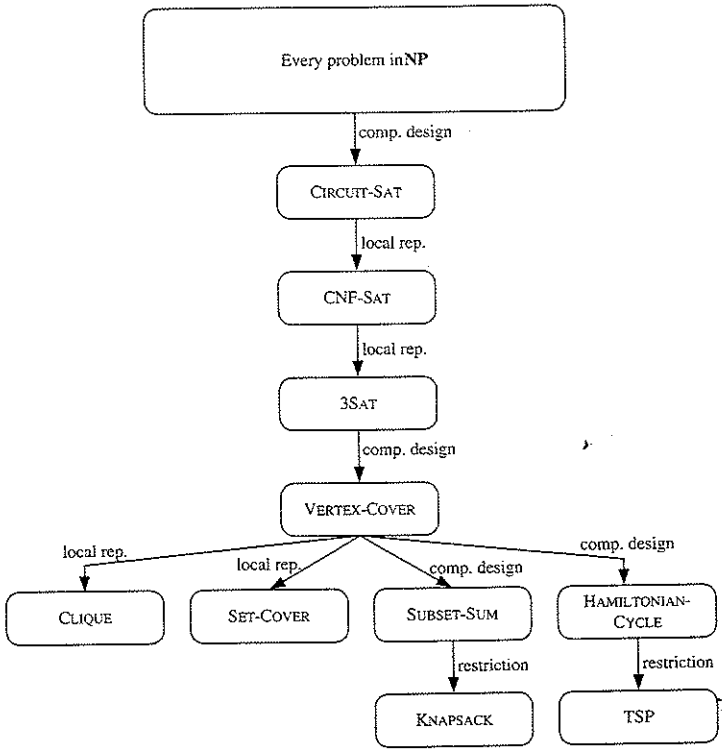


Figure 13.3: Illustration of the reductions used in some fundamental *NP*-completeness proofs. Each directed edge denotes a polynomial-time reduction, with the label on the edge indicating the primary form of that reduction. The topmost reduction is the Cook-Levin Theorem.

13.3.1 CNF-SAT and 3SAT

The first reductions we present are for problems involving Boolean formulas. A Boolean formula is a parenthesized expression that is formed from Boolean variables using Boolean operations, such as OR ($+$), AND ($\cdot$), NOT (drawn as a bar over the negated subexpression), IMPLIES ($\rightarrow$), and IF-AND-ONLY-IF ($\leftrightarrow$). A Boolean formula is in ***conjunctive normal form*** (CNF) if it is formed as a collection of subexpressions, called ***clauses***, that are combined using AND, with each clause formed as the OR of Boolean variables or their negation, called ***literals***. For example, the following Boolean formula is in CNF:

$$(\overline{x_1} + x_2 + x_4 + \overline{x_7})(x_3 + \overline{x_5})(\overline{x_2} + x_4 + \overline{x_6} + x_8)(x_1 + x_3 + x_5 + \overline{x_8}).$$

This formula evaluates to 1 if x_2, x_3, and x_4 are 1, where we use 0 for **false** and 1 for **true**. CNF is called a "normal" form, because any Boolean formula can be converted into this form.

CNF-SAT

Problem CNF-SAT takes a Boolean formula in CNF form as input and asks if there is an assignment of Boolean values to its variables so that the formula evaluates to 1.

It is easy to show that CNF-SAT is in *NP*, for, given a Boolean formula S, we can construct a simple nondeterministic algorithm that first "guesses" an assignment of Boolean values for the variables in S and then evaluates each clause of S in turn. If all the clauses of S evaluate to 1, then S is satisfied; otherwise, it is not.

To show that CNF-SAT is *NP*-hard, we will reduce the CIRCUIT-SAT problem to it in polynomial time. So, suppose we are given a Boolean circuit, C. Without loss of generality, we assume that each AND and OR gate has two inputs and each NOT gate has one input. To begin the construction of a formula S equivalent to C, we create a variable x_i for each input for the entire circuit C. One might be tempted to limit the set of variables to just these x_i's and immediately start constructing a formula for C by combining subexpressions for inputs, but in general this approach will not run in polynomial time. (See Exercise C-13.3.) Instead, we create a variable y_i for each output of a gate in C. Then, we create a short formula B_g that corresponds to each gate g in C as follows:

- If g is an AND gate with inputs a and b (which could be either x_i's or y_i's) and output c, then $B_g = (c \leftrightarrow (a \cdot b))$.
- If g is an OR gate with inputs a and b and output c, then $B_g = (c \leftrightarrow (a + b))$.
- If g is a NOT gate with input a and output b, then $B_g = (b \leftrightarrow \overline{a})$.

We wish to create our formula S by taking the AND of all of these B_g's, but such a formula would not be in CNF. So our method is to first convert each B_g to be in CNF, and then combine all of these transformed B_g's by AND operations to define the CNF formula S.

a	b	c	$B = (c \leftrightarrow (a \cdot b))$
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	1	1
0	0	1	0
0	0	0	1

DNF formula for $\overline{B} = a \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot c + \overline{a} \cdot b \cdot c + \overline{a} \cdot \overline{b} \cdot c$

CNF formula for $B = (\overline{a} + \overline{b} + c) \cdot (\overline{a} + b + \overline{c}) \cdot (a + \overline{b} + \overline{c}) \cdot (a + b + \overline{c})$.

Figure 13.4: A truth table for a Boolean formula B over variables a, b, and c. The equivalent formula for $\overline{B}$ in DNF, and equivalent formula for B in CNF.

To convert a Boolean formula B into CNF, we construct a truth table for B, as shown in Figure 13.4. We then construct a short formula D_i for each table row that evaluates to 0. Each D_i consists of the AND of the variables for the table, with the variable negated if and only if its value in that row is 0. We create a formula D by taking the OR of all the D_i's. Such a formula, which is the OR of formulas that are the AND of variables or their negation, is said to be in ***disjunctive normal form***, or ***DNF***. In this case, we have a DNF formula D that is equivalent to $\overline{B}$, since it evaluates to 1 if and only if B evaluates to 0. To convert D into a CNF formula for B, we apply, to each D_i, De Morgan's Laws, which establish that

$$\overline{(a+b)} = \overline{a} \cdot \overline{b} \quad \text{and} \quad \overline{(a \cdot b)} = \overline{a} + \overline{b}.$$

From Figure 13.4, we can replace each B_g that is of the form $(c \leftrightarrow (a \cdot b))$, by

$$(\overline{a} + \overline{b} + c)(\overline{a} + b + \overline{c})(a + \overline{b} + \overline{c})(a + b + \overline{c}),$$

which is in CNF. Likewise, for each B_g that is of the form $(b \leftrightarrow \overline{a})$, we can replace B_g by the equivalent CNF formula

$$(\overline{a} + \overline{b})(a + b).$$

We leave the CNF substitution for a B_g of the form $(c \leftrightarrow (a+b))$ as an exercise (R-13.2). Substituting each B_g in this way results in a CNF formula S' that corresponds exactly to each input and logic gate of the circuit, C. To construct the final Boolean formula S, then, we define $S = S' \cdot y$, where y is the variable that is associated with the output of the gate that defines the value of C itself. Thus, C is satisfiable if and only if S is satisfiable. Moreover, the construction from C to S builds a constant-sized subexpression for each input and gate of C; hence, this construction runs in polynomial time. Therefore, this local-replacement reduction gives us the following.

Theorem 13.8: CNF-SAT *is NP-complete.*

3SAT

Consider the 3SAT problem, which takes a Boolean formula S that is in conjunctive normal form (CNF) with each clause in S having exactly three literals, and asks if S is satisfiable. Recall that a Boolean formula is in CNF if it is formed by the AND of a collection of clauses, each of which is the OR of a set of literals. For example, the following formula could be an instance of 3SAT:

$$(\overline{x_1} + x_2 + \overline{x_7})(x_3 + \overline{x_5} + x_6)(\overline{x_2} + x_4 + \overline{x_6})(x_1 + x_5 + \overline{x_8}).$$

Thus, the 3SAT problem is a restricted version of the CNF-SAT problem. (Note that we cannot use the restriction form of *NP*-hardness proof, however, for this proof form only works for reducing a restricted version to its more general form.) In this subsection, we show that 3SAT is *NP*-complete, using the local replacement form of proof. Interestingly, the 2SAT problem, where every clause has exactly two literals, can be solved in polynomial time. (See Exercises C-13.4 and C-13.5.)

Note that 3SAT is in *NP*, for we can construct a nondeterministic polynomial-time algorithm that takes a CNF formula S with 3-literals per clause, guesses an assignment of Boolean values for S, and then evaluates S to see if it is equal to 1.

To prove that 3SAT is *NP*-hard, we reduce the CNF-SAT problem to it in polynomial time. Let C be a given Boolean formula in CNF. We perform the following local replacement for each clause C_i in C:

- If $C_i = (a)$, that is, it has one term, which may be a negated variable, then we replace C_i with $S_i = (a+b+c) \cdot (a+\overline{b}+c) \cdot (a+b+\overline{c}) \cdot (a+\overline{b}+\overline{c})$, where b and c are new variables not used anywhere else.
- If $C_i = (a+b)$, that is, it has two terms, then we replace C_i with the subformula $S_i = (a+b+c) \cdot (a+b+\overline{c})$, where c is a new variable not used anywhere else.
- If $C_i = (a+b+c)$, that is, it has three terms, then we set $S_i = C_i$.
- If $C_i = (a_1 + a_2 + a_3 + \cdots + a_k)$, that is, it has $k > 3$ terms, then we replace C_i with $S_i = (a_1 + a_2 + b_1) \cdot (\overline{b_1} + a_3 + b_2) \cdot (\overline{b_2} + a_4 + b_3) \cdots (\overline{b_{k-3}} + a_{k-1} + a_k)$, where $b_1, b_2, \ldots, b_{k-1}$ are new variables not used anywhere else.

Notice that the value assigned to the newly introduced variables is completely irrelevant. No matter what we assign them, the clause C_i is 1 if and only if the small formula S_i is also 1. Thus, the original clause C is 1 if and only if S is 1. Moreover, note that each clause increases in size by at most a constant factor and that the computations involved are simple substitutions. Therefore, we have shown how to reduce an instance of the CNF-SAT problem to an equivalent instance of the 3SAT problem in polynomial time. This, together with the earlier observation about 3SAT belonging to *NP*, gives us the following theorem.

Theorem 13.9: 3SAT *is NP-complete.*

13.3.2 VERTEX-COVER

Recall from Lemma 13.5 that VERTEX-COVER takes a graph G and an integer k and asks if there is a vertex cover for G containing at most k vertices. Formally, VERTEX-COVER asks if there is a subset C of vertices of size at most k, such that for each edge (v,w), we have $v \in C$ or $w \in C$. We showed, in Lemma 13.5, that VERTEX-COVER is in *NP*. The following example motivates this problem.

Example 13.10: *Suppose we are given a graph G representing a computer network where vertices represent routers and edges represent physical connections. Suppose further that we wish to upgrade some of the routers in our network with special new, but expensive, routers that can perform sophisticated monitoring operations for incident connections. If we would like to determine if k new routers are sufficient to monitor every connection in our network, then we have an instance of* VERTEX-COVER *on our hands.*

Let us now show that VERTEX-COVER is *NP*-hard, by reducing the 3SAT problem to it in polynomial time. This reduction is interesting in two respects. First, it shows an example of reducing a logic problem to a graph problem. Second, it illustrates an application of the component design proof technique.

Let S be a given instance of the 3SAT problem, that is, a CNF formula such that each clause has exactly three literals. We construct a graph G and an integer k such that G has a vertex cover of size at most k if and only if S is satisfiable. We begin our construction by adding the following:

- For each variable x_i used in the formula S, we add two vertices in G, one that we label with x_i and the other we label with $\overline{x_i}$. We also add the edge $(x_i,\overline{x_i})$ to G. (Note: These labels are for our own benefit; after we construct the graph G we can always relabel vertices with integers if that is what an instance of the VERTEX-COVER problem should look like.)

Each edge $(x_i,\overline{x_i})$ is a "truth-setting" component, for, with this edge in G, a vertex cover must include at least one of x_i or $\overline{x_i}$. In addition, we add the following:

- For each clause $C_i = (a+b+c)$ in S, we form a triangle consisting of three vertices, $i1$, $i2$, and $i3$, and three edges, $(i1,i2)$, $(i2,i3)$, and $(i3,i1)$.

Note that any vertex cover will have to include at least two of the vertices in $\{i1,i2,i3\}$ for each such triangle. Each such triangle is a "satisfaction-enforcing" component. We then connect these two types of components, by adding, for each clause $C_i = (a+b+c)$, the edges $(i1,a)$, $(i2,b)$, and $(i3,c)$. (See Figure 13.5.) Finally, we set the integer parameter $k = n+2m$, where n is the number of variables in S and m is the number of clauses. Thus, if there is a vertex cover of size at most k, it must have size exactly k. This completes the construction of an instance of the VERTEX-COVER problem.

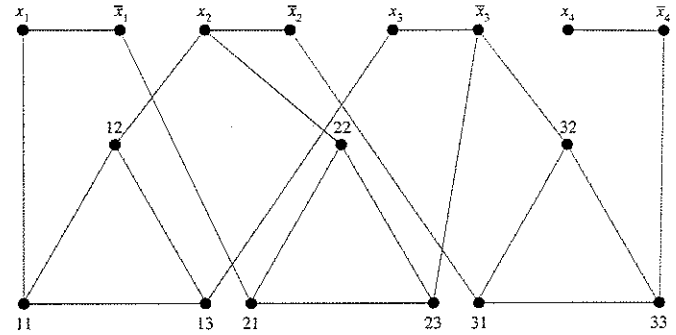


Figure 13.5: Example graph G as an instance of the VERTEX-COVER problem constructed from the formula $S = (x_1+x_2+x_3)\cdot(\overline{x_1}+x_2+\overline{x_3})\cdot(\overline{x_2}+\overline{x_3}+\overline{x_4})$.

This construction clearly runs in polynomial time, so let us consider its correctness. Suppose there is an assignment of Boolean values to variables in S so that S is satisfied. From the graph G constructed from S, we can build a subset of vertices C that contains each literal a (in a truth-setting component) that is assigned 1 by the satisfying assignment. Likewise, for each clause $C_i = (a+b+c)$, the satisfying assignment sets at least one of a, b, or c to 1. Whichever one of a, b, or c is 1 (picking arbitrarily if there are ties), we include the other two in our subset C. This C is of size $n+2m$. Moreover, notice that each edge in a truth-setting component and clause-satisfying component is covered, and two of every three edges incident on a clause-satisfying component are also covered. In addition, notice that an edge incident to a component associated clause C_i that is not covered by a vertex in the component must be covered by the node in C labeled with a literal, for the corresponding literal in C_i is 1.

Suppose then the converse, namely that there is a vertex cover C of size at most $n+2m$. By construction, this set must have size exactly $n+2m$, for it must contain one vertex from each truth-setting component and two vertices from each clause-satisfying component. This leaves one edge incident to a clause-satisfying component that is not covered by a vertex in the clause-satisfying component; hence, this edge must be covered by the other endpoint, which is labeled with a literal. Thus, we can assign the literal in S associated with this node 1 and each clause in S is satisfied; hence, all of S is satisfied. Therefore, S is satisfiable if and only if G has a vertex cover of size at most k. This gives us the following.

Theorem 13.11: VERTEX-COVER *is NP-complete.*

As mentioned before, the above reduction illustrates the component design technique. We constructed truth-setting and clause-satisfying components in our graph G to enforce important properties in the clause S.

13.3.3 CLIQUE and SET-COVER

As with the VERTEX-COVER problem, there are several problems that involve selecting a subset of objects from a larger set so as to optimize the size the subset can have while still satisfying an important property. In this subsection, we study two more such problems, CLIQUE and SET-COVER.

CLIQUE

A *clique* in a graph G is a subset C of vertices such that, for each v and w in C, with $v \neq w$, (v,w) is an edge. That is, there is an edge between every pair of distinct vertices in C. Problem CLIQUE takes a graph G and an integer k as input and asks whether there is a clique in G of size at least k.

We leave as a simple exercise (R-13.7) to show that CLIQUE is in *NP*. To show CLIQUE is *NP*-hard, we reduce the VERTEX-COVER problem to it. Therefore, let (G,k) be an instance of the VERTEX-COVER problem. For the CLIQUE problem, we construct the complement graph G^c, which has the same vertex set as G, but has the edge (v,w), with $v \neq w$, if and only if (v,w) is not in G. We define the integer parameter for CLIQUE as $n-k$, where k is the integer parameter for VERTEX-COVER. This construction runs in polynomial time and serves as a reduction, for G^c has a clique of size at least $n-k$ if and only if G has a vertex cover of size at most k. (See Figure 13.6.)

(a) (b)

Figure 13.6: A graph G illustrating the proof that CLIQUE is *NP*-hard. (a) Shows the graph G with the nodes of a clique of size 5 shaded in grey. (b) Shows the graph G^c with the nodes of a vertex cover of size 3 shaded in grey.

Therefore, we have the following.

Theorem 13.12: CLIQUE *is NP-complete.*

Note how simple the above proof by local replacement is. Interestingly, the next reduction, which is also based on the local replacement technique, is even simpler.

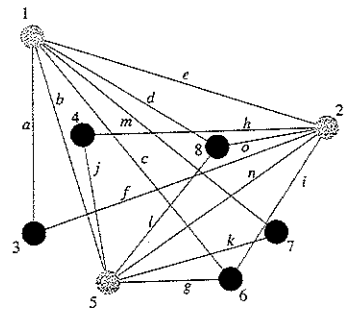
SET-COVER

Problem SET-COVER takes a collection of m sets $S_1, S_2, \ldots, S_m$ and an integer parameter k as input, and asks whether there is a subcollection of k sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$, such that

$$\bigcup_{i=1}^{m} S_i = \bigcup_{j=1}^{k} S_{i_j}.$$

That is, the union of the subcollection of k sets includes every element in the union of the original m sets.

We leave it to an exercise (R-13.14) to show SET-COVER is in *NP*. As to the reduction, we note that we can define an instance of SET-COVER from an instance G and k of VERTEX-COVER. Namely, for each each vertex v of G, there is set S_v, which contains the edges of G incident on v. Clearly, there is a set cover among these sets S_v's of size k if and only if there is a vertex cover of size k in G. (See Figure 13.7.)



$S_1 = \{a,b,c,m,d,e\}$
$S_2 = \{e,h,o,n,i\}$
$S_3 = \{a,f\}$
$S_4 = \{j,m\}$
$S_5 = \{b,j,l,n,k,g\}$
$S_6 = \{g,c,i\}$
$S_7 = \{k,m\}$
$S_8 = \{d,o,l\}$

(a) (b)

Figure 13.7: A graph G illustrating the proof that SET-COVER is *NP*-hard. The vertices are numbered 1 through 8 and the edges are given letter labels a through o. (a) Shows the graph G with the nodes of a vertex cover of size 3 shaded in grey. (b) Shows the sets associated with each vertex in G, with the subscript of each set identifying the associated vertex. Note that $S_1 \cup S_2 \cup S_5$ contains all the edges of G.

Thus, we have the following.

Theorem 13.13: SET-COVER *is NP-complete.*

This reduction illustrates how easily we can covert a graph problem into a set problem. In the next subsection we show how we can actually reduce graph problems to number problems.

13.3.4 Subset-Sum and Knapsack

Some hard problems involve only numbers. In such cases, we must take extra care to use the size of the input in bits, for some numbers can be very large. To clarify the role that the size of numbers can make, researchers say that a problem L is ***strongly NP-hard*** if L remains *NP*-hard even when we restrict the value of each number in the input to be bounded by a polynomial in the size (in bits) of the input. An input x of size n would satisfy this condition, for example, if each number i in x was represented using $O(\log n)$ bits. Interestingly, the number problems we study in this section are not strongly *NP*-hard. (See Exercises C-13.12 and C-13.13.)

Subset-Sum

In the Subset-Sum problem, we are given a set S of n integers and an integer k, and we are asked if there is a subset of integers in S that sum to k. This problem could arise, for example, as in the following.

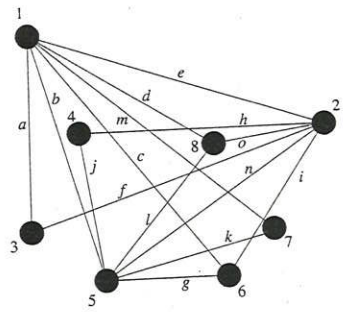
Example 13.14: *Suppose we have an Internet web server, and we are presented with a collection of download requests. For each each download request we can easily determine the size of the requested file. Thus, we can abstract each web request simply as an integer—the size of the requested file. Given this set of integers, we might be interested in determining a subset of them that exactly sums to the bandwidth our server can accommodate in one minute. Unfortunately, this problem is an instance of* Subset-Sum. *Moreover, because it is NP-complete, this problem will actually become harder to solve as our web server's bandwidth and request-handling ability improves.*

Subset-Sum might at first seem easy, and indeed showing it belongs to *NP* is straightforward. (See Exercise R-13.15.) Unfortunately, it is *NP*-complete, as we now show. Let G and k be given as an instance of the Vertex-Cover problem. Number the vertices of G from 1 to n and the edges G from 1 to m, and construct the ***incidence matrix*** H for G, defined so that $H[i,j] = 1$ if and only if the edge numbered j is incident on the vertex numbered i; otherwise, $H[i,j] = 0$. (See Figure 13.8.)

We use H to define some admittedly large (but still polynomial-sized) numbers to use as inputs to the Subset-Sum problem. Namely, for each row i of H, which encodes all the edges incident on vertex i, we construct the number

$$a_i = 4^{m+1} + \sum_{j=1}^{m} H[i,j]4^j.$$

Note that this number adds in a different power of 4 for each 1-entry in the ith row of $H[i,j]$, plus a larger power of 4 for good measure. The collection of a_i's defines an "incidence component" to our reduction, for each power of 4 in an a_i, except for the largest, corresponds to a possible incidence between vertex i and some edge.



(a)

H	1	2	3	4	5	6	7	8
a	1	0	1	0	0	0	0	0
b	1	0	0	0	1	0	0	0
c	1	0	0	0	0	1	0	0
d	1	0	0	0	0	0	0	1
e	1	1	0	0	0	0	0	0
f	0	1	1	0	0	0	0	0
g	0	0	0	0	1	1	0	0
h	0	1	0	1	0	0	0	0
i	0	1	0	0	0	1	0	0
j	0	0	0	1	1	0	0	0
k	0	0	0	0	1	0	1	0
l	0	0	0	0	1	0	0	1
m	0	1	0	1	0	0	0	0
n	0	1	0	0	1	0	0	0
o	0	1	0	0	0	0	0	1

(b)

Figure 13.8: A graph G illustrating the proof that Subset-Sum is *NP*-hard. The vertices are numbered 1 through 8 and the edges are given letter labels a through o. (a) Shows the graph G; (b) shows the incidence matrix H for G. Note that there is a 1 for each edge in one or more of the columns for vertices 1, 2, and 5.

In addition to the above incidence component, we also define an "edge-covering component," where, for each edge j, we define a number

$$b_j = 4^j.$$

We then set the sum we wish to attain with a subset of these numbers as

$$k' = k4^{m+1} + \sum_{j=1}^{m} 2 \cdot 4^j,$$

where k is the integer parameter for the Vertex-Cover instance.

Let us consider, then, how this reduction, which clearly runs in polynomial time, actually works. Suppose graph G has a vertex cover $C = \{i_1, i_2, \ldots, i_k\}$, of size k. Then we can construct a set of values adding to k' by taking every a_i with an index in C, that is, each a_{i_r} for $r = 1, 2, \ldots, k$. In addition, for each edge numbered j in G, if only one of j's endpoints is included in C, then we also include b_j in our subset. This set of numbers sums to k', for it includes k values of 4^{m+1} plus 2 values of each 4^j (either from two a_{i_r}'s such that this edge has both endpoints in C or from one a_{i_r} and one b_j if C contains just one endpoint of edge j).

Suppose there is a subset of numbers that sums to k'. Since k' contains k values of 4^{m+1}, it must include exactly k a_i's. Let us include vertex i in our cover for each such a_i. Such a set is a cover, for each edge j, which corresponds to a power 4^j, must contribute two values to this sum. Since only one value can come from a b_j, one must have come from at least one of the chosen a_i's. Thus, we have:

Theorem 13.15: Subset-Sum *is NP-complete.*

KNAPSACK

In the KNAPSACK problem, illustrated in Figure 13.9, we are given a set S of items, numbered 1 to n. Each item i has an integer size, s_i, and worth, w_i. We are also given two integer parameters, s, and w, and are asked if there is a subset, T, of S such that

$$\sum_{i \in T} s_i \leq s, \quad \text{and} \quad \sum_{i \in T} w_i \geq w.$$

Problem KNAPSACK defined above is the decision version of the optimization problem "0-1 knapsack" discussed in Section 5.3.3.

We can motivate the KNAPSACK problem with the following Internet application.

Example 13.16: *Suppose we have s widgets that we are interested in selling at an Internet auction web site. A prospective buyer i can bid on multiple lots by saying that he or she is interested in buying s_i widgets at a total price of w_i dollars. If multiple-lot requests, such as this, cannot be broken up (that is, buyer i wants exactly s_i widgets), then determining if we can earn w dollars from this auction gives rise to the* KNAPSACK *problem. (If lots can be broken up, then our auction optimization problem gives rise to the fractional knapsack problem, which can be solved efficiently using the greedy method of Section 5.1.1.)*

The KNAPSACK problem is in *NP*, for we can construct a nondeterministic polynomial-time algorithm that guesses the items to place in our subset T and then verifies that they do not violate the s and w constraints, respectively.

KNAPSACK is also *NP*-hard, as it actually contains the SUBSET-SUM problem as a special case. In particular, any instance of numbers given for the SUBSET-SUM problem can correspond to the items for an instance of KNAPSACK with each $w_i = s_i$ set to a value in the SUBSET-SUM instance and the targets for the size s and worth w both equal to k, where k is the integer we wish to sum to for the SUBSET-SUM problem. Thus, by the restriction proof technique, we have the following.

Theorem 13.17: KNAPSACK *is NP-complete.*

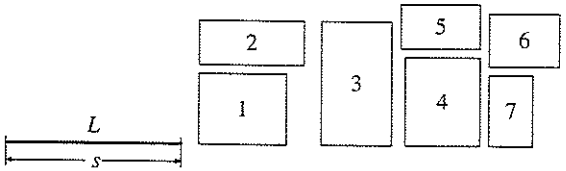


Figure 13.9: A geometric view of the KNAPSACK problem. Given a line L of length s, and a collection of n rectangles, can we translate a subset of the rectangles to have their bottom edge on L so that the total area of the rectangles touching L is at least w? Thus, the width of rectangle i is s_i and its area is w_i.

13.3.5 HAMILTONIAN-CYCLE and TSP

The last two *NP*-complete problems we consider involve the search for certain kinds of cycles in a graph. Such problems are useful for optimizing the travel of robots and printer-plotters, for example.

HAMILTONIAN-CYCLE

Recall, from Lemma 13.3, that HAMILTONIAN-CYCLE is the problem that takes a graph G and asks if there is a cycle in G that visits each vertex in G exactly once, returning to its starting vertex. (See Figure 13.10a.) Also recall, from Lemma 13.3, that HAMILTONIAN-CYCLE is in *NP*. To show that this problem is *NP*-complete, we will reduce VERTEX-COVER to it, using a component design type of reduction.

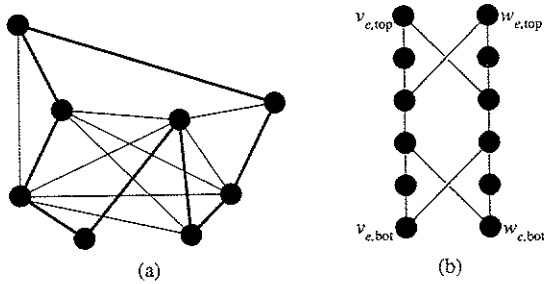


Figure 13.10: Illustrating the HAMILTONIAN-CYCLE problem and its *NP*-completeness proof. (a) Shows an example graph with a Hamiltonian cycle shown in bold. (b) Illustrates a cover-enforcer subgraph H_e used to show that HAMILTONIAN-CYCLE is *NP*-hard.

Let G and k be a given instance of the VERTEX-COVER problem. We will construct a graph H that has a Hamiltonian cycle if and only if G has a vertex cover of size k. We begin by including a set of k initially disconnected vertices $X = \{x_1, x_2, \ldots, x_k\}$ to H. This set of vertices will serve as a "cover-choosing" component, for they will serve to identify which nodes of G should be included in a vertex cover. In addition, for each edge $e = (v, w)$ in G we create a "cover-enforcer" subgraph H_e in H. This subgraph H_e has 12 vertices and 14 edges as shown in Figure 13.10b.

Six of the vertices in the cover-enforcer H_e for $e = (v, w)$ correspond to v and the other six correspond to w. Moreover, we label two vertices in cover-enforcer H_e corresponding to v as $v_{e,\text{top}}$ and $v_{e,\text{bot}}$, and we label two vertices in H_e corresponding to w as $w_{e,\text{top}}$ and $w_{e,\text{bot}}$. These are the only vertices in H_e that will be connected to any other vertices in H outside of H_e.

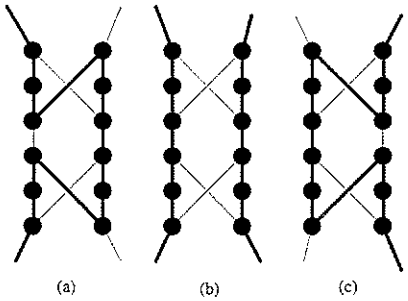


Figure 13.11: The three possible ways that a Hamiltonian cycle can visit the edges in a cover-enforcer H_e.

Thus, a Hamiltonian cycle can visit the nodes of H_e in only one of three possible ways, as shown in Figure 13.11.

We join the important vertices in each cover-enforcer H_e to other vertices in H in two ways, one that corresponds to the cover-choosing component and one that corresponds to the cover-enforcing component. For the cover-choosing component, we add an edge from each vertex in X to every vertex $v_{e,\text{top}}$ and every vertex $v_{e,\text{bot}}$. That is, we add $2kn$ edges to H, where n is the number of vertices in G.

For the cover-enforcing component, we consider each vertex v in G in turn. For each such v, let $\{e_1, e_2, \ldots, e_{d(v)}\}$ be a listing of the edges of G that are incident upon v. We use this listing to create edges in H by joining $v_{e_i,\text{bot}}$ in H_{e_i} to $v_{e_{i+1},\text{top}}$ in $H_{e_{i+1}}$, for $i = 1, 2, \ldots, d-1$. (See Figure 13.12.) We refer to the H_{e_i} components joined in this way as belonging to the ***covering thread*** for v. This completes the construction of the graph H. Note that this computation runs in polynomial time in the size of G.

We claim that G has a vertex cover of size k if and only if H has a Hamiltonian cycle. Suppose, first, that G has a vertex cover of size k. Let $C = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$ be such a cover. We construct a Hamiltonian cycle in H, by connecting a series of paths P_j, where each P_j starts at x_j and ends at x_{j+1}, for $j = 1, 2, \ldots, k-1$, except for the last path P_k, which starts at x_k and ends at x_1. We form such a path P_j as follows. Start with x_j, and then visit the entire covering thread for v_{i_j} in H, returning to x_{j+1} (or x_1 if $j = k$). For each cover-enforcer subgraph H_e in the covering thread for v_{i_j}, which is visited in this P_j, we write, without loss of generality, e as (v_{i_j}, w). If w is not also in C, then we visit this H_e as in Figure 13.11a or Figure 13.11c (with respect to v_{i_j}). Instead, if w is also in C, then we visit this H_e as in Figure 13.11b. In this way we will visit each vertex in H exactly once, since C is a vertex cover for G. Thus, this cycle we construct is in fact a Hamiltonian cycle.

Suppose, conversely, that H has a Hamiltonian cycle. Since this cycle must visit all the vertices in X, we break this cycle up into k paths, P_1, P_2, ..., P_k, each

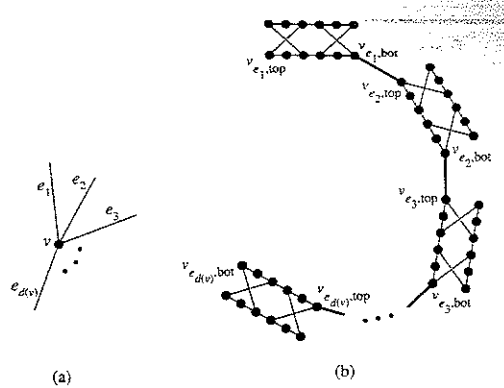


Figure 13.12: Connecting the cover-enforcers. (a) A vertex v in G and its set of incident edges $\{e_1, e_2, \ldots, e_{d(v)}\}$. (b) The connections made between the H_{e_i}'s in H for the edges incident upon v.

of which starts and ends at a vertex in X. Moreover, by the structure of the cover-enforcer subgraphs H_e and the way that we connected them, each P_j must traverse a portion (possibly all) of a covering thread for a vertex v in G. Let C be the set of all such vertices in G. Since the Hamiltonian cycle must include the vertices from every cover-enforcer H_e and every such subgraph must be traversed in a way that corresponds to one (or both) of e's endpoints, C must be a vertex cover in G.

Therefore, G has a vertex cover of size k if and only if H has a Hamiltonian cycle. This gives us the following.

Theorem 13.18: HAMILTONIAN-CYCLE *is NP-complete.*

TSP

In the ***traveling salesperson problem***, or language TSP, we are given an integer parameter k and a graph G, such that each edge e in G is assigned an integer cost $c(e)$, and we are asked if there is a cycle in G that visits all the vertices in G (possibly more than once) and has total cost at most k. Showing that TSP is in *NP* is as easy as guessing a sequence of vertices and then verifying that it forms a cycle of cost at most k in G. Showing that TSP is *NP*-complete is also easy, as it contains the HAMILTONIAN-CYCLE problem as a special case. Namely, given an instance G of the HAMILTONIAN-CYCLE problem, we can create an instance of TSP by assigning each edge in G the cost $c(e) = 1$ and setting the integer parameter $k = n$, where n is the number of vertices in G. Therefore, using the restriction form of reduction, we get the following.

Theorem 13.19: TSP *is NP-complete.*