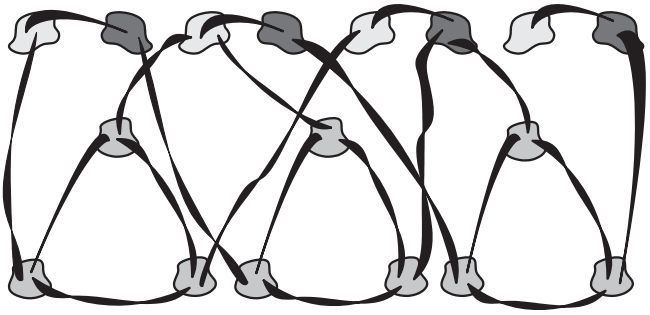
Chapter

13 NP-Completeness



Contents

Some computational problems are hard. We rack our brains to find efficient algorithms for solving them, but time and time again we fail. It would be nice if we could prove that finding an efficient algorithm is impossible in such cases. Such a proof would be a great relief when an efficient algorithm evades us, for then we could take comfort from the fact that no efficient algorithm exists for this problem. Unfortunately, such proofs are typically even harder to come by than efficient algorithms.

Still, all is not frustration, for the topics we discuss in this chapter let us show that certain problems are indeed computationally hard. The proofs involve a concept known as ***NP-completeness***. This concept allows us to rigorously show that finding an efficient algorithm for a certain problem is at least as hard as finding efficient algorithms for ***all*** the problems in a large class of problems called "***NP***." The formal notion of "efficient" we use here is that a problem has an algorithm running in time proportional to a polynomial function of its input size, n. (Recall that this notion of efficiency was already mentioned in Section 1.2.2.) That is, we consider an algorithm "efficient" if it runs in time $O(n^k)$ on any input of size n, for some constant $k > 0$. Even so, the class ***NP*** contains some extremely difficult problems, for which polynomial-time solutions have eluded researchers for decades. Therefore, while showing that a problem is ***NP***-complete is admittedly not the same as proving that an efficient algorithm for the problem is impossible, it is nevertheless a powerful statement. Basically, showing that a problem L is ***NP***-complete says that, although we have been unable to find an efficient algorithm for L, neither has any computer scientist who has ever lived! Indeed, most computer scientists strongly believe it is impossible to solve any ***NP***-complete problem in polynomial time.

In this chapter, we formally define the class ***NP*** and its related class ***P***, and we show how to prove that some problems are ***NP***-complete. We also discuss some of the best known of the ***NP***-complete problems, showing that each one is at least as hard as every other problem in ***NP***. These problems include satisfiability, vertex cover, knapsack, and traveling salesperson problems.

We do not stop there, however, for many of these problems are quite important, in that they are related to optimization problems whose solution in the real world can oftentimes save money, time, or other resources. Thus, we also discuss some ways of dealing with ***NP***-completeness in this chapter. One of the most effective methods is to construct polynomial-time approximation algorithms for ***NP***-complete problems. Although such algorithms do not usually produce optimal solutions, they oftentimes come close to being optimal. In fact, in some cases we can provide a guarantee of how close an approximation algorithm will come to an optimal solution. We explore several such situations in this chapter.

We conclude this chapter by covering techniques that often work well for dealing with ***NP***-complete problems in practice. We present, in particular, ***backtracking*** and ***branch-and-bound***, which construct algorithms that run in exponential time in the worst case, but nevertheless take advantage of situations where faster time is possible. We give Java examples of both techniques.

13.1 P and NP

In order to study ***NP***-completeness, we need to be more precise about running time. Namely, instead of the informal notion of input size as the number of "items" that form the input (see Chapter 1), we define the ***input size***, n, of a problem to be the number of bits used to encode an input instance. We also assume that characters and numbers in the input are encoded using a reasonable binary encoding scheme, so that each character uses a constant number of bits and each integer $M > 0$ is represented with at most $c \log M$ bits, for some constant $c > 0$. In particular, we disallow ***unary encoding***, where an integer M is represented with M 1's.

Recall that we have, for the rest of this book, defined the input size n to be the number of "items" in an input. Let us for the moment, however, refer to the number of items in an input as N and the number of bits to encode an input as n. Thus, if M is the largest integer in an input, then $N + \log M \leq n \leq cN \log M$, for some constant $c > 0$. Formally, we define the worst-case ***running time*** of an algorithm A to be the worst-case time taken by A as a function of n, taken over all possible inputs having an encoding with n bits. Fortunately, as we show in the following lemma, most algorithms running in polynomial time in terms of N still result in polynomial-time algorithms in terms of n. We define an algorithm to be ***c-incremental*** if any primitive operation involving one or two objects represented with b bits results in an object represented with at most $b + c$ bits, for $c \geq 0$. For example, an algorithm using multiplication as a primitive operation may not be c-incremental for any constant c. Of course, we can include a routine in a c-incremental algorithm to perform multiplication, but we should not count this routine as a primitive operation here.

Lemma 13.1: *If a c-incremental algorithm A has a worst-case running time $t(N)$ in the RAM model, as a function of the number of input items, N, for some constant $c > 0$, then A has running time $O(n^2 t(n))$, in terms of the number, n, of bits in a standard nonunary encoding of the input.*

Proof: Note that $N \leq n$. Thus, $t(N) \leq t(n)$. Likewise, each primitive operation in the algorithm A, involving one or two objects represented with $b \geq 1$ bits, can be performed using at most db^2 bitwise operations, for some constant $d \geq 1$, since c is a constant. Such primitive operations include all comparison, control flow, and basic non-multiplicative arithmetic operations. Moreover, in N steps of a c-incremental algorithm, the largest any object's representation can become is $cN + b$, where b is the maximum size of any input object. But, $cN + b \leq (c+1)n$. Thus, every step in A will take at most $O(n^2)$ bit steps to complete. ∎

Therefore, any "reasonable" algorithm that runs in polynomial time in terms of the number of input items will also run in polynomial time in terms of the number of input bits. Thus, for the remainder of this chapter, we may revert to using n as input size and number of "items" with the understanding that any "polynomial-time" algorithm must run in polynomial time in terms of the number of input bits.

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 $\boldsymbol{P} = \boldsymbol{NP}$ or not. Indeed, researchers don't even know for sure whether or not $\boldsymbol{P} = \boldsymbol{NP} \cap$ **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.

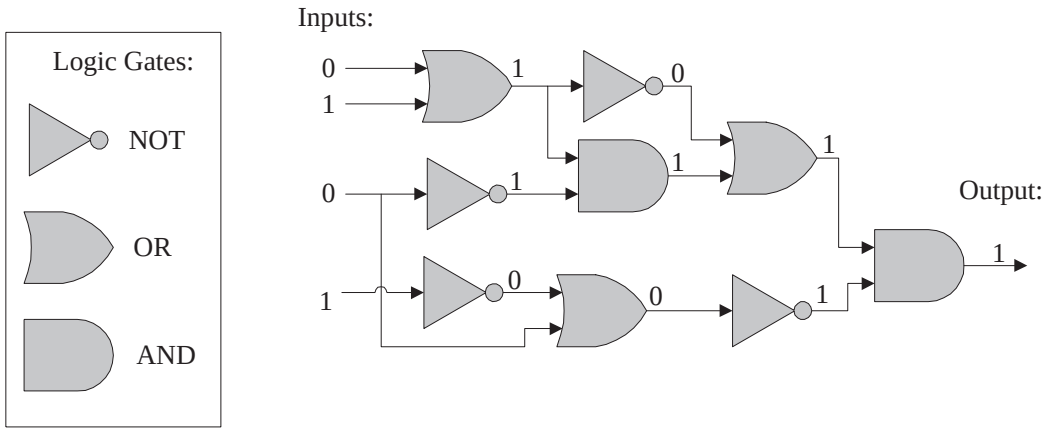


Figure 13.1: An example Boolean circuit.

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 $L \in \boldsymbol{NP}$, $L \xrightarrow{\text{poly}} M$. 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 $\boldsymbol{P} = \boldsymbol{NP}$.

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. ■

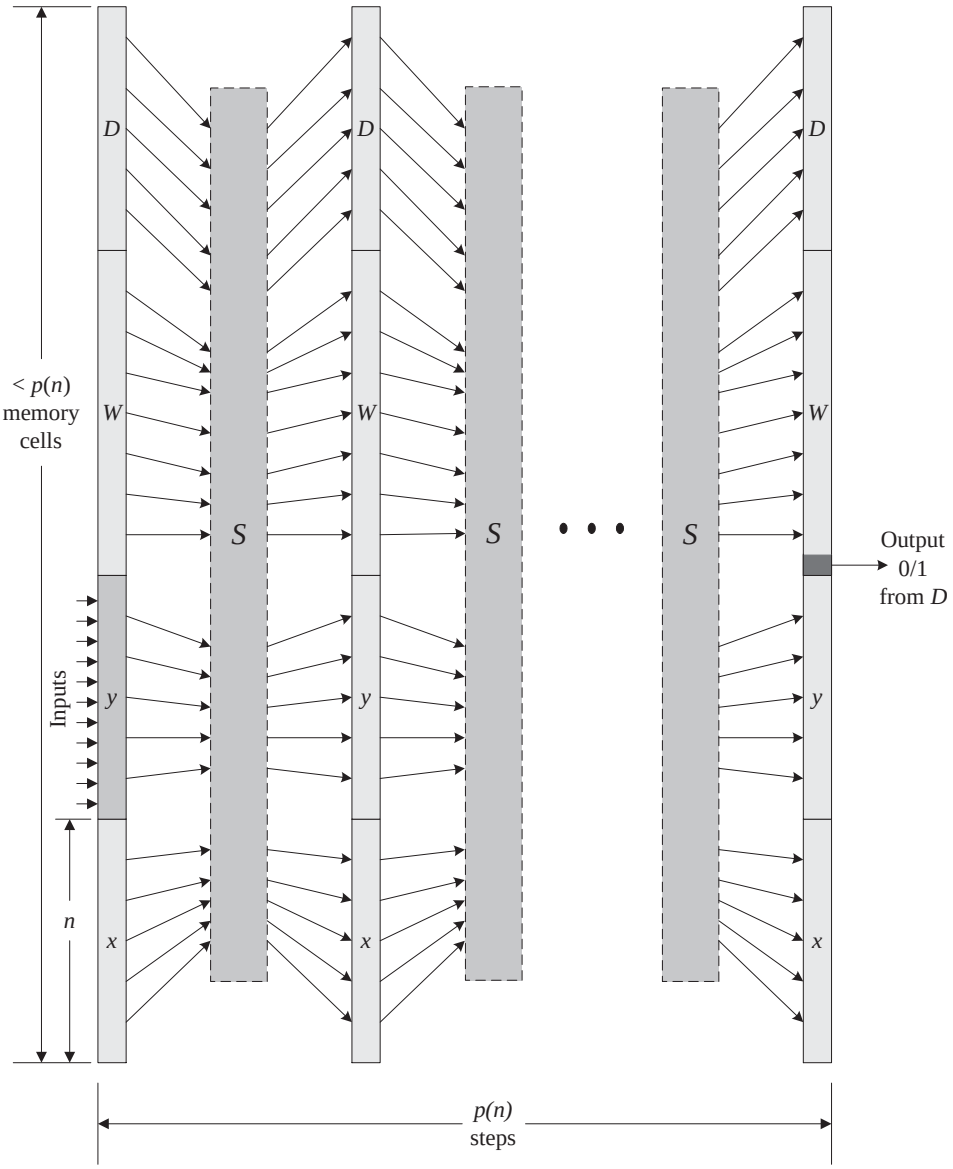


Figure 13.2: An illustration of the circuit used to prove that CIRCUIT-SAT is ***NP***-hard. The only true inputs correspond to the certificate, y. The problem instance, x, the working storage, W, and the program code, D, are initially "hard wired" values. The only output is the bit that determines if 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 $L_1 \xrightarrow{poly} L_2$ and $L_2 \xrightarrow{poly} L_3$, then $L_1 \xrightarrow{poly} L_3$.*

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.

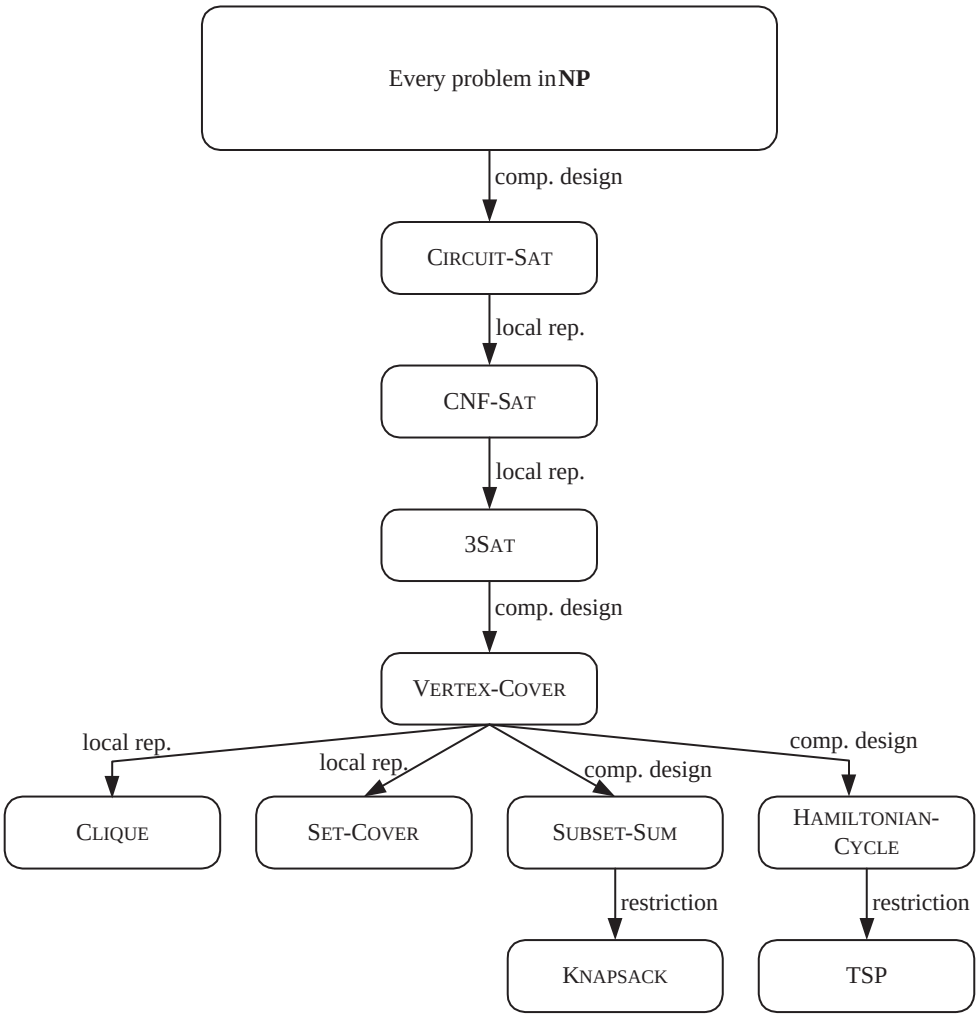


Figure 13.3: Illustration of the reductions used in some fundamental ***NP***-completeness proofs. Each directed edge denotes a polynomial-time reduction, with the label on the edge indicating the primary form of that reduction. The topmost reduction is the Cook-Levin Theorem.

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.

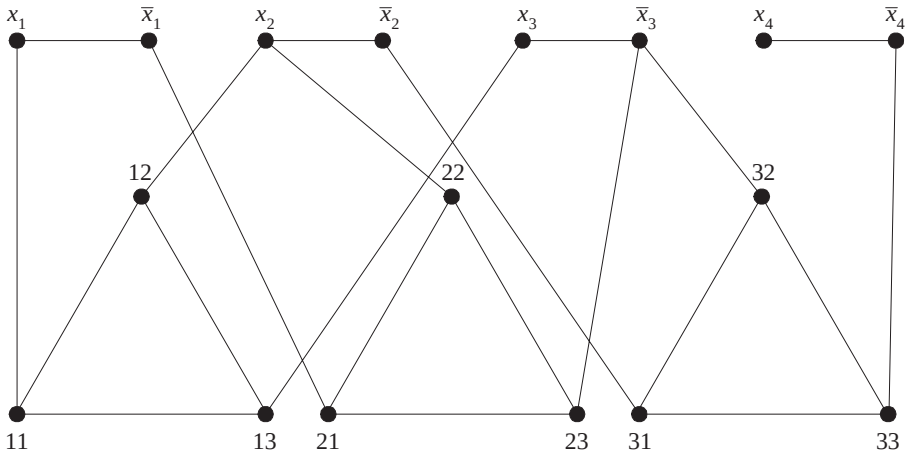


Figure 13.5: Example graph G as an instance of the VERTEX-COVER problem constructed from the formula $S = (x_1 + x_2 + x_3) \cdot (\overline{x_1} + x_2 + \overline{x_3}) \cdot (\overline{x_2} + \overline{x_3} + \overline{x_4})$.

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.)

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.

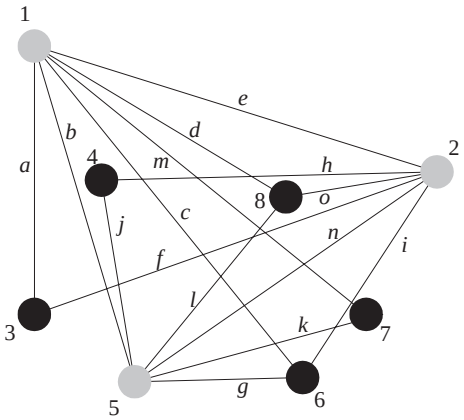
SET-COVER

Problem SET-COVER takes a collection of m sets S_1, S_2, ..., 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.)



(a)

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

(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.

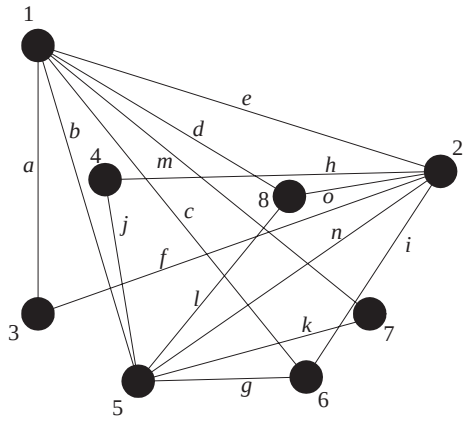
Example 13.14: *Suppose we have an Internet web server, and we are presented with a collection of download requests. For each each download request we can easily determine the size of the requested file. Thus, we can abstract each web request simply as an integer—the size of the requested file. Given 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.*

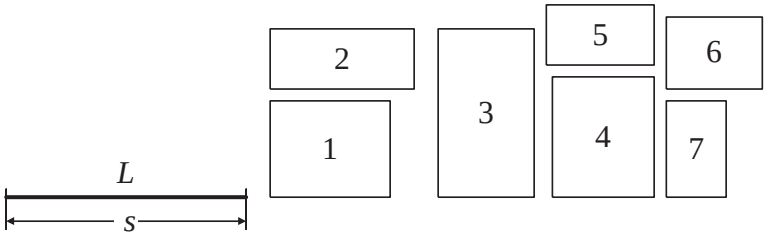


Figure 13.9: A geometric view of the KNAPSACK problem. Given a line L of length s, and a collection of n rectangles, can we translate a subset of the rectangles to have their bottom edge on L so that the total area of the rectangles touching L is at least w? Thus, the width of rectangle i is s_i and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

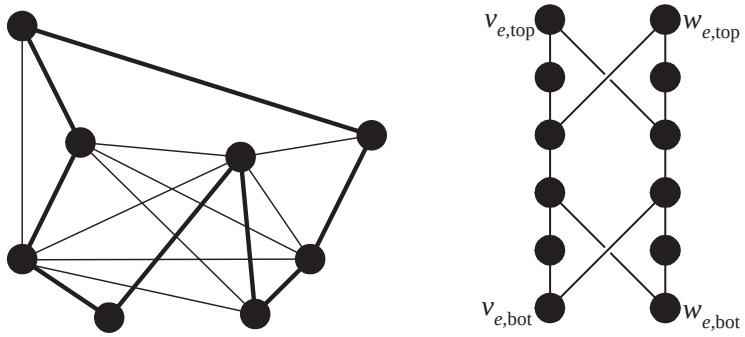


Figure 13.10: Illustrating the HAMILTONIAN-CYCLE problem and its ***NP***-completeness proof. (a) Shows an example graph with a Hamiltonian cycle shown in bold. (b) Illustrates a cover-enforcer subgraph H_e used to show that HAMILTONIAN-CYCLE is ***NP***-hard.

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.

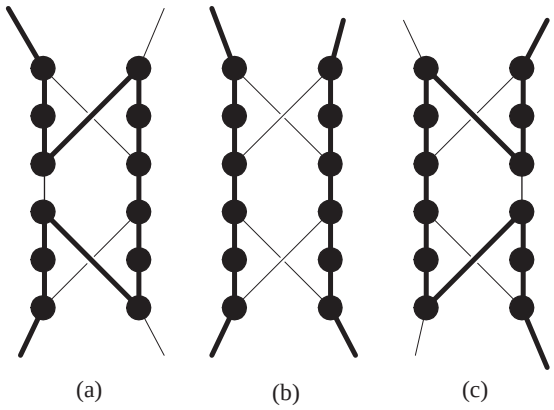


Figure 13.11: The three possible ways that a Hamiltonian cycle can visit the edges in a cover-enforcer H_e.

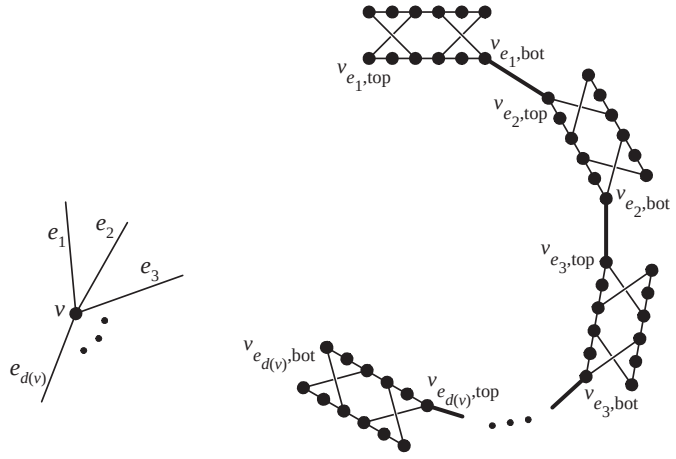
Thus, a Hamiltonian cycle can visit the nodes of H_e in only one of three possible ways, as shown in Figure 13.11.

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



(a) (b)

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.*

13.4 Approximation Algorithms

One way of dealing with ***NP***-completeness for optimization problems is to use an ***approximation algorithm***. Such an algorithm typically runs much faster than an algorithm that strives for an exact solution, but it is not guaranteed to find the best solution. In this section, we study methods for constructing and analyzing approximation algorithms for hard optimization problems.

The general situation is that we have some problem instance x, which could be an encoding of a set of numbers, a graph, etc., as discussed above. In addition, for the problem we are interested in solving for x, there will often be a large number of ***feasible*** solutions for x, which define a set $\mathcal{F}$ of such feasible solutions.

We also have a cost function, c, that determines a numeric cost $c(S)$ for any solution $S \in \mathcal{F}$. In the general optimization problem, we are interested in finding a solution S in $\mathcal{F}$, such that

$$c(S) = OPT = \min\{c(T) : T \in \mathcal{F}\}.$$

That is, we want a solution with minimum cost. We could also formulate a maximization version of the optimization problem, as well, which would simply involve replacing the above "min" with "max." To keep the discussion in this section simple, however, we will typically take the view that, unless otherwise stated, our optimization goal is a minimization.

The goal of an approximation algorithm is to come as close to the optimum value as possible in a reasonable amount of time. As we have been doing for this entire chapter, we take the view in this section that a reasonable amount of time is at most polynomial time.

Ideally, we would like to provide a guarantee of how close an approximation algorithm comes to the optimal value, OPT. We say that a δ-***approximation*** algorithm for a particular optimization problem is an algorithm that returns a feasible solution S (that is, $S \in \mathcal{F}$), such that

$$c(S) \leq \delta OPT,$$

for a minimization problem. For a maximization problem, a δ-approximation algorithm would guarantee $OPT \leq \delta c(S)$. Or, in general, we have

$$\delta \leq \max\{c(S)/OPT, OPT/c(S)\}.$$

In the remainder of this section, we study problems for which we can construct δ-approximation algorithms for various values of δ. We begin with the ideal situation as far as approximation factors are concerned.

13.4.1 Polynomial-Time Approximation Schemes

There are some problems for which we can construct δ-approximation algorithms that run in polynomial-time with $\delta = 1+\varepsilon$, for any fixed value $\varepsilon > 0$. The running time of such a collection of algorithms depends both on n, the size of the input, and also on the fixed value ε. We refer to such a collection of algorithms as a ***polynomial-time approximation scheme***, or ***PTAS***. When we have a polynomial-time approximation scheme for a given optimization problem, we can tune our performance guarantee based on how much time we can afford to spend. Ideally, the running time is polynomial in both n and $1/\varepsilon$, in which case we have a ***fully polynomial-time approximation scheme***.

Polynomial-time approximation schemes take advantage of a property that some hard problems possess, namely, that they are rescalable. A problem is said to be ***rescalable*** if an instance x of the problem can be transformed into an equivalent instance x' (that is, one with the same optimal solution) by scaling the cost function, c. For example, TSP is rescalable. Given an instance G of TSP, we can construct an equivalent instance G' by multiplying the distance between every pair of vertices by a scaling factor s. The traveling salesperson tour in G' will be the same as in G, although its cost will now be multiplied by s.

A Fully Polynomial-Time Approximation Scheme for KNAPSACK

To be more concrete, let us give a fully polynomial approximation scheme for the optimization version of a well-known problem, KNAPSACK (Sections 5.1.1 and 13.3.4). In the optimization version of this problem, we are given a set S of items, numbered 1 to n, together with a size constraint, s. Each item i in S is given an integer size, s_i, and worth, w_i, and we are asked to find a subset, T, of S, such that T maximizes the worth

$$w = \sum_{i \in T} w_i \quad \text{while satisfying} \quad \sum_{i \in T} s_i \leq s.$$

We desire a PTAS that produces a $(1+\varepsilon)$-approximation, for any given fixed constant ε. That is, such an algorithm should find a subset T' satisfying the size constraint such that if we define $w' = \sum_{i \in T'} w_i$, then

$$OPT \leq (1+\varepsilon)w',$$

where OPT is the optimal worth summation, taken over all possible subsets satisfying the total size constraint. To prove that this inequality holds, we will actually prove that

$$w' \geq (1-\varepsilon/2)OPT,$$

for $0 < \varepsilon < 1$. This will be sufficient, however, since, for any fixed $0 < \varepsilon < 1$,

$$\frac{1}{1-\varepsilon/2} < 1+\varepsilon.$$

To derive a PTAS for KNAPSACK, we take advantage of the fact that this problem is rescalable. Suppose we are given a value of ε with $0 < \varepsilon < 1$. Let $w_{\max}$ denote the maximum worth of any item in S. Without loss of generality, we assume that the size of each item is at most s (for an item larger than this could not fit in the knapsack). Thus, the item with worth $w_{\max}$ defines a lower bound for the optimal value. That is, $w_{\max} \leq OPT$. Likewise, we can define an upper bound on the optimal solution by noting that the knapsack can at most contain all n items in S, each of which has worth at most $w_{\max}$. Thus, $OPT \leq n w_{\max}$. To take advantage of the rescalability of KNAPSACK, we round each worth value w_i to w_i', the nearest smaller multiple of $M = \varepsilon w_{\max}/2n$. Let us denote the rounded version of S as S', and let us also use OPT' to denote the solution for this rounded version S'. Note that, by simple substitution, $OPT \leq 2n^2M/\varepsilon$. Moreover, $OPT' \leq OPT$, since we rounded every worth value in S down to form S'. Thus, $OPT' \leq 2n^2M/\varepsilon$.

Therefore, let us turn to our solution for the rounded version S' of the KNAPSACK problem for S. Since every worth value in S' is a multiple of M, any achievable worth of a collection of items taken from S' is also a multiple of M. Moreover, there are just $N = \lceil 2n^2/\varepsilon \rceil$ such multiples that need to be considered, because of the upper bound on OPT'. We can use dynamic programming (Section 5.3) to construct an efficient algorithm for finding the optimal worth for S'. In particular, let us define the parameter,

$$s[i,j] = \text{the size of the smallest set of items in } \{1,2,\ldots,j\} \text{ with worth } iM.$$

The key insight to the design of a dynamic programming algorithm for solving the rounded KNAPSACK problem is the observation that we can write

$$s[i,j] = \min\{s[i,j-1],\ s_j + s[i-(w_j'/M), j-1]\},$$

for $i = 1,2,\ldots,N$, and $j = 1,2,\ldots,n$. (See Figure 13.13.)

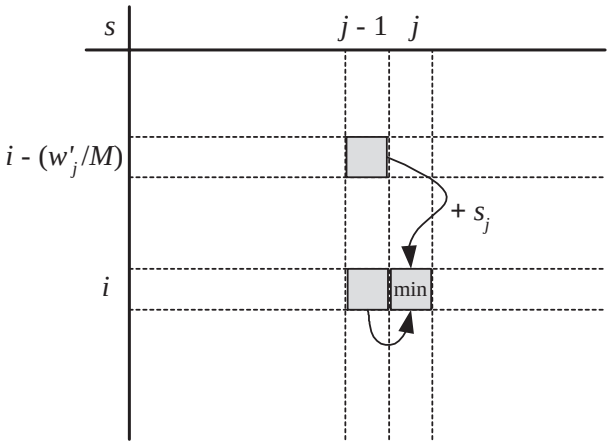


Figure 13.13: Illustration of the equation for $s[i,j]$ used in the dynamic program for the scaled version of KNAPSACK.

The above equation for $s[i,j]$ follows from the fact that item j will either contribute or not contribute to the smallest way of achieving worth iM from the items in $\{1,2,\ldots,j\}$. In addition, note that for the base case, $j = 0$, when no items at all are included in the knapsack, then

$$s[i,0] = +\infty,$$

for $i = 1,2,\ldots,N$. That is, such a size is undefined. In addition,

$$s[0,j] = 0,$$

for $j = 1,2,\ldots,n$, since we can always achieve worth 0 by including no items in the knapsack. The optimal value is defined by

$$OPT' = \max\{iM : s[i,n] \leq s\}.$$

This is the value that is output by our PTAS algorithm.

Analysis of the PTAS for KNAPSACK

We can easily convert the above description into a dynamic programming algorithm that computes OPT' in $O(n^3/\varepsilon)$ time. Such an algorithm gives us the value of an optimal solution, but we can easily translate the dynamic programming algorithm for computing the size into one for the actual set of items.

Let us consider, then, how good an approximation OPT' is for OPT. Recall that we reduced the worth w_i of each item i by at most $M = \varepsilon w_{\max}/2n$. Thus,

$$OPT' \geq OPT - \varepsilon w_{\max}/2,$$

since the optimal solution can contain at most n items. Since $OPT \geq w_{\max}$, this in turn implies that

$$OPT' \geq OPT - \varepsilon OPT/2 = (1 - \varepsilon/2)OPT.$$

Thus, $OPT \leq (1+\varepsilon)OPT'$, which was what we wished to prove. The running time of our approximation algorithm is $O(n^3/\varepsilon)$. Our scheme of designing an efficient algorithm for any given $\varepsilon > 0$ gives rise to a fully polynomial approximation scheme, since the running time is polynomial in both n and $1/\varepsilon$. This fact gives us the following.

Theorem 13.20: *The* KNAPSACK *optimization problem has a fully polynomial approximation scheme that achieves a $(1+\varepsilon)$-approximation factor in $O(n^3/\varepsilon)$ time, where n is the number of items in the* KNAPSACK *instance and $0 < \varepsilon < 1$ is a given fixed constant.*

13.4.2 A 2-Approximation for VERTEX-COVER

It is not always possible to design a polynomial-time approximation scheme for a hard problem, let alone a fully polynomial-time approximation scheme. In such cases, we would still like good approximations, of course, but we will have to settle for an approximation factor that is not arbitrarily close to 1, as we had in the previous subsection. In this subsection, we describe how we can achieve a 2-approximation algorithm for a well-known ***NP***-complete problem, the VERTEX-COVER problem (Section 13.3.2). In the optimization version of this problem, we are given a graph G and we are asked to produce the smallest set C that is a vertex cover for G, that is, such that every edge in G is incident on some vertex in C.

Our approximation algorithm is based on the greedy method, and is rather simple. It involves picking an edge in the graph, adding both its endpoints to the cover, and then deleting this edge and its incident edges from the graph. The algorithm repeats this process until no edges are left. We give the details for this approach in Algorithm 13.14.

Algorithm VertexCoverApprox(G):
Input: A graph G
Output: A small vertex cover C for G
$C \leftarrow \emptyset$
while G still has edges **do**
  select an edge $e = (v,w)$ of G
  add vertices v and w to C
  for each edge f incident to v or w **do**
    remove f from G
return C

Algorithm 13.14: A 2-approximation algorithm for VERTEX-COVER.

We leave the details of how to implement this algorithm in $O(n+m)$ time as a simple exercise (R-13.18). Let us consider, then, why this algorithm is a 2-approximation. First, observe that each edge $e = (v,w)$ selected by the algorithm, and used to add v and w to C, must be covered in any vertex cover. That is, any vertex cover for G must contain v or w (possibly both). The approximation algorithm adds both v and w to C in such a case. When the approximation algorithm completes, there are no uncovered edges left in G, for we remove all the edges covered by the vertices v and w when we add them to C. Thus, C forms a vertex cover of G. Moreover, the size of C is at most twice that of an optimal vertex cover for G, since, for every two vertices we add to C, one of these vertices must belong to the optimal cover. Therefore, we have the following.

Theorem 13.21: *Given a graph with n vertices and m edges, the optimization version of* VERTEX-COVER *has a 2-approximation algorithm taking $O(n+m)$ time.*

13.4.3 A 2-Approximation for a Special Case of TSP

In the optimization version of the traveling salesperson problem, or TSP, we are given a weighted graph G, such that each edge e in G has an integer weight $c(e)$, and we are asked to find a minimum-weight cycle in G that visits all vertices in G. In this section we describe a simple 2-approximation algorithm for a special case of the TSP optimization problem.

The Triangle Inequality

Consider an instance of TSP such that the edge weights satisfy the ***triangle inequality***. That is, for any three edges (u,v), (v,w), and (u,w) in G, we have

$$c((u,v)) + c((v,w)) \geq c((u,w)),$$

Also, suppose that every pair of vertices in G is connected by an edge, that is, G is a complete graph. These properties, which hold for any distance metric, imply that the optimal tour of G will visit each vertex exactly once; hence, let us consider only Hamiltonian cycles as possible solutions of TSP.

The Approximation Algorithm

Our approximation algorithm takes advantage of the above properties of G to design a very simple TSP approximation algorithm, which has just three steps. In the first step we construct a minimum-spanning tree, M, of G (Section 7.3). In the second step we construct an Euler-tour traversal, E, of M, that is, a traversal of M that starts and ends at the same vertex and traverses each edge of M exactly once in each direction (Section 2.3.3). In the third step we construct a tour T from E by marching through the edges of E, but each time we have two edges (u,v) and (v,w) in E, such that v has already been visited, we replace these two edges by the edge (u,w) and continue. In essence, we are constructing T as a a preorder traversal of M. This three-step algorithm clearly runs in polynomial-time. (See Figure 13.15.)

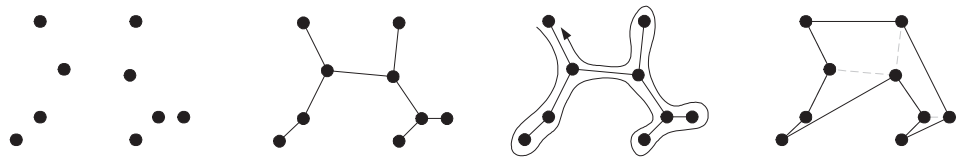


Figure 13.15: Example run of the approximation algorithm for TSP for a graph satisfying the triangle inequality: (a) a set S of points in the plane, with Euclidean distance defining the costs of the edges (not shown); (b) the minimum-spanning tree, M, for S, (c) an Euler tour, E, of M; (d) the approximate TSP tour, T.

Analysis of the TSP Approximation Algorithm

The analysis of why this algorithm achieves an approximation factor of 2 is also simple. Let us extend our notation so that $c(H)$ denotes the total weight of the edges in a subgraph H of G. Let T' be the optimal tour for the graph G. If we delete any edge from T' we get a path, which is, of course, also a spanning tree. Thus,

$$c(M) \leq c(T').$$

We can also easily relate the cost of E to that of M, as

$$c(E) = 2c(M),$$

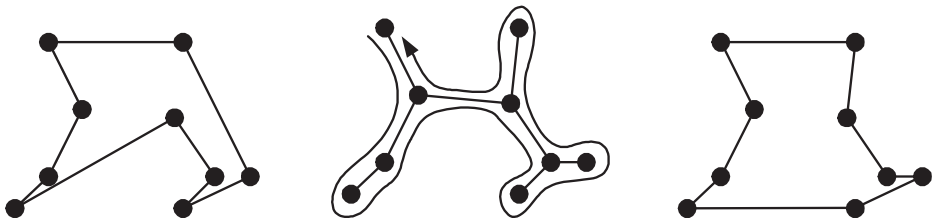
since the Euler tour E visits each edge of M exactly once in each direction. Finally, note that, by the triangle inequality, when we construct our tour T, each time we replace two edges (u,v) and (v,w) with the edge (u,w), we do not increase the cost of the tour. That is,

$$c(T) \leq c(E).$$

Therefore, we have

$$c(T) \leq 2c(T').$$

(See Figure 13.16.)



Output tour T (at most the cost of E) | Euler tour E of MST M (twice the cost of M) | Optimal tour T' (at least the cost of MST M)

Figure 13.16: Illustrating the proof that MST-based algorithm is a 2-approximation for the TSP optimization problem.

We may summarize this discussion as follows.

Theorem 13.22: *If a weighted graph G is complete and has edge weights satisfying the triangle inequality, then there is a 2-approximation algorithm for the* TSP *optimization problem for G that runs in polynomial time.*

This theorem depends heavily on the fact that the cost function on the graph G satisfies the triangle inequality. In fact, without this assumption, no constant-factor approximation algorithm for the optimization version of TSP exists that runs in polynomial time, unless $\boldsymbol{P} = \boldsymbol{NP}$. (See Exercise C-13.14.)

13.4.4 A Logarithmic Approximation for SET-COVER

There are some cases when achieving even a constant-factor approximation in polynomial time is difficult. In this section, we study one of the best known of such problems, the SET-COVER problem (Section 13.3.3). In the optimization version of this problem, we are given a collection of sets $S_1, S_2, \ldots, S_m$, whose union is a universe U of size n, and we are asked to find the smallest integer k, such that there is a subcollection of k sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ with

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

Although it is difficult to find a constant-factor approximation algorithm that runs in polynomial time for this problem, we can design an efficient algorithm that has an approximation factor of $O(\log n)$. As with several other approximation algorithms for hard problems, this algorithm is based on the greedy method (Section 5.1).

A Greedy Approach

Our algorithm selects sets S_{i_j} one at a time, each time selecting the set that has the most uncovered elements. When every element in U is covered, we are done. We give a simple pseudo-code description in Algorithm 13.17.

Algorithm SetCoverApprox(S):
Input: A collection S of sets $S_1, S_2, \ldots, S_m$ whose union is U
Output: A small set cover C for S
$C \leftarrow \emptyset$ {The set cover we are building}
$E \leftarrow \emptyset$ {The set of covered elements from U}
while $E \neq U$ **do**
  select a set S_i that has the maximum number of uncovered elements
  add S_i to C
  $E \leftarrow E \cup S_i$
Return C.

Algorithm 13.17: An approximation algorithm for SET-COVER.

This algorithm runs in polynomial time. (See Exercise R-13.19.)

Analyzing the Greedy SET-COVER Algorithm

To analyze the approximation factor of the above greedy SET-COVER algorithm, we will use an amortization argument based on a charging scheme (Section 1.5). Namely, each time our approximation algorithm selects a set S_j we will charge the elements of S_j for its selection.

Specifically, consider the moment in our algorithm when a set S_j is added to C, and let k be the number of previously uncovered elements in S_j. We must pay a total charge of 1 to add this set to C, so we charge each previously uncovered element i of S_j a charge of

$$c(i) = 1/k.$$

Thus, the total size of our cover is equal to the total charges made by our algorithm. That is,

$$|C| = \sum_{i \in U} c(i).$$

To prove an approximation bound, we will consider the charges made to the elements in each subset S_j that belongs to an optimal cover, C'. So, suppose that S_j belongs to C'. Let us write $S_j = \{x_1, x_2, \ldots, x_{n_j}\}$ so that S_j's elements are listed in the order in which they are covered by our algorithm (we break ties arbitrarily). Now, consider the iteration in which x_1 is first covered. At that moment, S_j has not yet been selected; hence, whichever set is selected must have at least n_j uncovered elements. Thus, x_1 is charged at most $1/n_j$. So let us consider, then, the moment our algorithm charges an element x_l of S_j. In the worst case, we will have not yet chosen S_j (indeed, our algorithm may never choose this S_j). Whichever set is chosen in this iteration has, in the worst case, at least $n_j - l + 1$ uncovered elements; hence, x_l is charged at most $1/(n_j - l + 1)$. Therefore, the total amount charged to all the elements of S_j is at most

$$\sum_{l=1}^{n_j} \frac{1}{n_l - l + 1} = \sum_{l=1}^{n_j} \frac{1}{l},$$

which is the familiar Harmonic number, H_{n_i}. It is well known (for example, see the Appendix) that H_{n_j} is $O(\log n_j)$. Let $c(S_j)$ denote the total charges given to all the elements of a set S_j that belongs to the optimal cover C'. Our charging scheme implies that $c(S_j)$ is $O(\log n_j)$. Thus, summing over the sets of C', we obtain

$$\begin{aligned} \sum_{S_j \in C'} c(S_j) &\leq \sum_{S_j \in C'} b \log n_j \\ &\leq b|C'| \log n, \end{aligned}$$

for some constant $b \geq 1$. But, since C' is a set cover,

$$\sum_{i \in U} c(i) \leq \sum_{S_j \in C'} c(S_j)$$

Therefore,

$$|C| \leq b|C'| \log n.$$

This fact gives us the following result.

Theorem 13.23: *The optimization version of the* SET-COVER *problem has an $O(\log n)$-approximation polynomial-time algorithm for finding a cover of a collection of sets whose union is a universe of size n.*

13.5 Backtracking and Branch-and-Bound

In the above sections, we showed many problems to be ***NP***-complete. Thus, unless $\boldsymbol{P} = \boldsymbol{NP}$, which the vast majority of computer scientists believes is not true, it is impossible to solve any of these problems in polynomial time. Nevertheless, many of these problems arise in real-life applications where solutions to them need to be found, even if finding these solutions may take a long time. Thus, in this section, we address techniques for dealing with ***NP***-completeness that have shown much promise in practice. These techniques allow us to design algorithms that can find solutions to hard problems, often in a reasonable amount of time. In this section, we study the methods of ***backtracking*** and ***branch-and-bound***.

13.5.1 Backtracking

The backtracking design pattern is a way to build an algorithm for some hard problem L. Such an algorithm searches through a large, possibly even exponential-size, set of possibilities in a systematic way. The search strategy is typically optimized to avoid symmetries in problem instances for L and to traverse the search space so as to find an "easy" solution for L if such a solution exists.

The ***backtracking*** technique takes advantage of the inherent structure that many ***NP***-complete problems possess. Recall that acceptance for an instance x in a problem in ***NP*** can be verified in polynomial time given a polynomial-sized certificate. Oftentimes, this certificate consists of a set of "choices," such as the values assigned to a collection of Boolean variables, a subset of vertices in a graph to include in a special set, or a set of objects to include in a knapsack. Likewise, the verification for a certificate often involves a simple test of whether or not the certificate demonstrates a successful configuration for x, such as satisfying a formula, covering all the edges in a graph, or conforming to certain performance criteria. In such cases, we can use the ***backtracking*** algorithm, given in Algorithm 13.18, to systematically search for a solution to our problem, if such a problem exists.

The backtracking algorithm traverses through possible "search paths" to locate solutions or "dead ends." The configuration at the end of such a path consists of a pair (x,y), where x is the remaining subproblem to be solved and y is the set of choices that have been made to get to this subproblem from the original problem instance. Initially, we give the backtracking algorithm the pair $(x,\emptyset)$, where x is our original problem instance. Anytime the backtracking algorithm discovers that a configuration (x,y) cannot lead to a valid solution no matter how additional choices are made, then it cuts off all future searches from this configuration and "backtracks" to another configuration. In fact, this approach gives the backtracking algorithm its name.

Algorithm Backtrack(x):
Input: A problem instance x for a hard problem
Output: A solution for x or "no solution" if none exists
$F \leftarrow \{(x,\emptyset)\}$. {$F$ is the "frontier" set of subproblem configurations}
while $F \neq \emptyset$ **do**
select from F the most "promising" configuration (x,y)
expand (x,y) by making a small set of additional choices
let (x_1,y_1), (x_2,y_2), ..., (x_k,y_k) be the set of new configurations.
for each new configuration (x_i,y_i) **do**
perform a simple consistency check on (x_i,y_i)
if the check returns "solution found" **then**
return the solution derived from (x_i,y_i)
if the check returns "dead end" **then**
discard the configuration (x_i,y_i) {Backtrack}
else
$F \leftarrow F \cup \{(x_i,y_i)\}$ {(x_i,y_i) starts a promising search path}
return "no solution"

Algorithm 13.18: The template for a backtracking algorithm.

Filling in the Details

In order to turn the backtracking strategy into an actual algorithm, we need only fill in the following details:

1. Define a way of selecting the most "promising" candidate configuration from the frontier set F.
2. Specify the way of expanding a configuration (x,y) into subproblem configurations. This expansion process should, in principle, be able to generate all feasible configurations, starting from the initial configuration, $(x,\emptyset)$.
3. Describe how to perform a simple consistency check for a configuration (x,y) that returns "solution found," "dead end," or "continue."

If F is a stack, then we get a depth-first search of the configuration space. In fact, in this case we could even use recursion to implement F automatically as a stack. Alternatively, if F is a queue, then we get a breadth-first search of the configuration space. We can also imagine other data structures to implement F, but as long as we have an intuitive notion of how to select the most "promising" configuration from F with each iteration, then we have a backtracking algorithm.

So as to make this approach more concrete, let us work through an application of the backtracking technique to the CNF-SAT problem.

A Backtracking Algorithm for CNF-SAT

Recall that in the CNF-SAT problem we are given a Boolean formula S in conjunctive normal form (CNF) and are asked if S is satisfiable. To design a backtracking algorithm for CNF-SAT, we will systematically make tentative assignments to the variables in S and see if such assignments make S evaluate immediately to 1 or 0, or yield a new formula S' for which we could continue making tentative value assignments. Thus, a configuration in our algorithm will consist of a pair (S', y), where S' is a Boolean formula in CNF, and y is an assignment of values to Boolean variables not in S' such that making these assignments in S results in the formula S'.

To formulate our backtracking algorithm, then, we need to give the details of each of the three components to the backtracking algorithm. Given a frontier F of configurations, we make our most "promising" choice, which is the subformula S' with the smallest clause. Such a formula is the most constrained of all the formulas in F; hence, we would expect it to hit a dead end most quickly if that is indeed its destiny.

Let us consider, then, how to generate subproblems from a subformula S'. We do this by locating a smallest clause C in S', and picking a variable x_i that appears in C. We then create two new subproblems that are associated with our assigning $x_i = 1$ and $x_i = 0$ respectively.

Finally, we must say how to process S' to perform a consistency check for an assignment of a variable x_i in S'. We begin this processing by reducing any clauses containing x_i based on the assigned 0 or 1 value of x_i (depending on the choice we made). If this reduction results in a new clause with a single literal, x_j or $\overline{x_j}$, we also perform the appropriate value assignment to x_j to make this single-literal clause satisfied. We then process the resulting formula to propagate the assigned value of x_j. If this new assignment in turn results in a new single-literal clause, we repeat this process until we have no more single-literal clauses. If at any point we discover a contradiction (that is, clauses x_i and $\overline{x_i}$, or an empty clause), then we return "dead end." If we reduce the subformula S' all the way to the constant 1, then we return "solution found," along with all the variable assignments we made to reach this point. Otherwise, we derive a new subformula, S'', such that each clause has at least two literals, along with the value assignments that lead from the original formula S to S''. We call this operation the ***reduce*** operation for propagating an assignment of value to x_i in S'.

Fitting all of these pieces into the template for the backtracking algorithm results in an algorithm that solves the CNF-SAT problem in about as fast a time as we can expect. In general, the worst-case running time for this algorithm is still exponential, but the backtracking can often speed things up. Indeed, if every clause in the given formula has at most two literals, then this algorithm runs in polynomial time. (See Exercise C-13.4.)

Java Example: A Backtracking Solution for SUBSET-SUM

To be even more concrete, let us consider a Java backtracking solution for the ***NP***-hard SUBSET-SUM problem. Recall from Section 13.3.4 that 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.

To make the decision easier for determining whether a configuration is a dead end or not, let us assume that the integers in S are given in nondecreasing order as $S = \{a_0, a_1, \ldots, a_{n-1}\}$. Given this choice, we then define the three main components of our backtracking algorithm as follows:

1. For our way of selecting the most "promising" candidate configuration, we make a choice common in many backtracking algorithms. Namely, we will use recursion to perform a depth-first search of the configuration space. Thus, our method stack will automatically keep track of unexplored configurations for us.
2. For our procedure for specifying the way of expanding a configuration (x,y) into subproblem configurations, let us simply "march" down the sequence of integers in S in order. Thus, having a configuration that has already considered the subset $S_i = \{a_0, \ldots, a_i\}$, we simply need to generate the two possible configurations determined by whether we use a_{i+1} or not.
3. The final major component of our backtracking algorithm is a way to perform a consistency check that returns "solution found," "dead end," or "continue." To aid in this test, we will again use the fact that the integers in S are sorted. Suppose we have already considered integers in the subset $S_i = \{a_0, \ldots, a_i\}$ and are now considering a_{i+1}. There is a simple two-fold consistency test we can perform at this point. Let k_i denote the sum of the elements in S_i that we have tentatively chosen to belong to our selection, and let the sum of the remaining integers be

$$r_{i+1} = \sum_{j=i+1}^{n-1} a_j.$$

The first part of our test is to confirm that

$$k \geq k_i + a_{i+1}.$$

If this condition fails, then we are "overshooting" k. Moreover, since S is sorted, any a_j with $j > i+1$ will also overshoot k; hence, we are at a dead end in this case. For the second part of our test, we confirm that

$$k \leq k_i + r_{i+1}.$$

If this condition fails, then we are "undershooting" k, and there is no hope for us to reach k with the remaining set of integers in S; hence, we are also at a dead end in this case. If both tests succeed, then we consider a_{i+1} and proceed.

We give the Java code for this method in Code Fragment 13.19.

```
/**
 * Method to find a subset of an array of integers summing to k, assuming:
 * - the array a is sorted in nondecreasing order,
 * - we have already considered the elements up to index i,
 * - the ones we have chosen add up to sum,
 * - the set that are left sum to reamin.
 * The function returns "true" and prints the subset if it is found.
 * Should be first called as findSubset(a,k,-1,0,t), where t is total.
 */
public static boolean findSubset(int[] a, int k, int i, int sum, int remain) {
  /* Test conditions for expanding this configuration */
  if (i+1 >= a.length) return false; // safety check that integers remain
  if (sum + remain < k) return false; // we're undershooting k
  int next = a[i+1];                  // the next candidate integer
  if (sum + next > k) return false; // we're overshooting k
  if (sum + next == k) {            // we've found a solution!
    System.out.print(k + "=" + next); // begin printing solution
    return true;
    }
  if (findSubset(a, k, i+1, sum+next, remain−next)) {
    System.out.print("+" + next);     // solution includes a[i+1]
    return true;
    }
  else    // backtracking - solution doesn't include a[i+1]
    return findSubset(a, k, i+1, sum, remain);
}
```

Code Fragment 13.19: A Java implementation of a backtracking algorithm for the SUBSET-SUM problem.

Note that we have defined the searching method, findSubset, recursively, and we use that recursion both for the backtracking and also for printing a solution when it is found. Also note that each recursive call takes $O(1)$ time, since an array is passed as a reference to the base address. Thus, the worst-case running time of this method is $O(2^n)$, where n is the number of integers we are considering. We hope, of course, that for typical problem instances the backtracking will avoid this worst case, either by quickly finding a solution or by using the conditions for dead ends to prune away many useless configurations.

There is a minor improvement we can make to the findSubset method if we can faithfully trust that it is always called correctly. Namely, we can drop the safety test for integers that are left to be considered. For, if there are no more integers to consider, then remain $= 0$. Thus, sum $+$ remain $<$ k, since we would have terminated this path of searching for a solution earlier had we overshot k or hit k exactly. Still, although we could, in theory, drop this test for remaining integers, we keep this test in our code for safety reasons, just in case someone calls our method incorrectly.

13.5.2 Branch-and-Bound

The backtracking algorithm is effective for decision problems, but it is not designed for optimization problems, where, in addition to having some feasibility condition be satisfied for a certificate y associated with an instance x, we also have a cost function $f(x)$ that we wish to minimize or maximize (without loss of generality, let us assume the cost function should be minimized). Nevertheless, we can extend the backtracking algorithm to work for such optimization problems, and in so doing derive the algorithmic design pattern known as ***branch-and-bound***.

The branch-and-bound design pattern, given in Algorithm 13.20, has all the elements of backtracking, except that rather than simply stopping the entire search process any time a solution is found, we continue processing until the best solution is found. In addition, the algorithm has a scoring mechanism to always choose the most promising configuration to explore in each iteration. Because of this approach, branch-and-bound is sometimes called a ***best-first search*** strategy.

Algorithm Branch-and-Bound(x):
  Input: A problem instance x for a hard optimization (minimization) problem
  Output: An optimal solution for x or "no solution" if none exists
  $F \leftarrow \{(x,\emptyset)\}$ {Frontier set of subproblem configurations}
  $b \leftarrow (+\infty,\emptyset)$ {Cost and configuration of current best solution}
  while $F \neq \emptyset$ **do**
    select from F the most "promising" configuration (x,y)
    expand (x,y), yielding new configurations (x_1,y_1), ..., (x_k,y_k)
    for each new configuration (x_i,y_i) **do**
      perform a simple consistency check on (x_i,y_i)
      if the check returns "solution found" **then**
        if the cost c of the solution for (x_i,y_i) beats b **then**
          $b \leftarrow (c,(x_i,y_i))$
        else
          discard the configuration (x_i,y_i)
      if the check returns "dead end" **then**
        discard the configuration (x_i,y_i) {Backtrack}
      else
        if $lb(x_i,y_i)$ is less than the cost of b **then**
          $F \leftarrow F \cup \{(x_i,y_i)\}$ {(x_i,y_i) starts a promising search path}
        else
          discard the configuration (x_i,y_i) {A "bound" prune}
  return b

Algorithm 13.20: The template for a branch-and-bound algorithm. This algorithm assumes the existence of a function, $lb(x_i,y_i)$, that returns a lower bound on the cost of any solution derived from the configuration (x_i,y_i).

To provide for the optimization criterion of always selecting the "most promising" configuration, we extend the three assumptions for a backtracking algorithm to add one more condition:

- For any configuration (x,y), we assume we have a function, $lb(x,y)$, that returns a lower bound on the cost of any solution that is derived from this configuration.

The only strict requirement for $lb(x,y)$, then, is that it must be less than or equal to the cost of any derived solution. But, as should be clear from the branch-and-bound description, if this lower bound is more accurate, the algorithm's efficiency improves.

A Branch-and-Bound Algorithm for TSP

To make the branch-and-bound approach more concrete, let us consider how it can be applied to solve the optimization version of the traveling salesperson (TSP) problem. In the optimization version of this problem, we are given a graph G with a cost function $c(e)$ defined for each edge e in G, and we wish to find the smallest total-cost tour that visits every vertex in G, returning back to its starting vertex.

We can design an algorithm for TSP by computing for each edge $e = (v,w)$, the minimum-cost path that begins at v and ends at w while visiting all other vertices in G along the way. To find such a path, we apply the branch-and-bound technique. We generate the path from v to w in $G - \{e\}$ by augmenting a current path by one vertex in each loop of the branch-and-bound algorithm.

- After we have built a partial path P, starting, say, at v, we only consider augmenting P with vertices in not in P.
- We can classify a partial path P as a "dead end" if the vertices not in P are disconnected in $G - \{e\}$.
- To define the lower bound function, lb, we can use the total cost of the edges in P plus $c(e)$. This will certainly be a lower bound for any tour that will be built from e and P.

In addition, after we have run the algorithm to completion for one edge e in G, we can use the best path found so far over all tested edges, rather than restarting the current best solution b at $+\infty$. The running time of the resulting algorithm will still be exponential in the worst-case, but it will avoid a considerable amount of unnecessary computation in practice. The TSP problem is of considerable interest, as it has many applications, so such a solution could be of use in practice if the number of vertices in the input graph is not too high. In addition, there are a number of other heuristics that can be added to the search for an optimal TSP tour, but these are beyond the scope of this book.

Java Example: A Branch-and-Bound Solution for KNAPSACK

To be even more concrete, let us describe a Java implementation of a branch-and-bound solution to the KNAPSACK problem. In the optimization version of this problem, we are given a set S of items, numbered 0 to $n-1$. Each item i is given an integer size, s_i, and worth, w_i. We are also given an integer parameter, *size*, and asked for a subset, T, of S such that $\sum_{i \in T} s_i \leq size$ and the total worth of items in T, $worth = \sum_{i \in T} w_i$, is maximized.

Let us begin by borrowing some ideas from the greedy algorithm for solving the fractional KNAPSACK problem (Section 5.1.1). Namely, let us assume that the items in S are sorted in nonincreasing order by w_i/s_i values. We will process them in this order, so that we are considering items by decreasing gain, starting with the element with maximum gain, that is, the item with maximum worth for its size. Our configuration, then, will be defined by a subset S_i of the first i items in S based on this order. So the indices of the items in S_i are in the range 0 to $i-1$ (and let us define S_0 to be the empty configuration having index -1).

We begin by placing the configuration for S_0 into a priority queue, P. Then, in each iteration of the branch-and-bound algorithm, we select the most promising configuration c in P. If i is the index of the last item considered for c, then we expand c into two new configurations, one that includes item $i+1$ and one that excludes it. Note that every configuration that satisfies the size constraint is a valid solution to the KNAPSACK problem. Thus, if either of the two new configurations is valid and better than the best solution found so far, we update our current best to be the better of the two, and continue the loop.

In order to select configurations that are most promising, of course, we need a way of scoring configurations by their potential value. Since in the KNAPSACK problem we are interested in maximizing a worth, rather than minimizing a cost, we score configurations by computing an upper bound on their potential worth. Specifically, given a configuration c, which has considered items with indices 0 to i, we compute an upper bound for c by starting with the total worth w_c for c and see how much more worth we can add to c if we were to augment c with a solution to a fractional KNAPSACK problem taken from the remaining items in S. Recalling that the items in S are sorted by nonincreasing w_i/s_i values, let k be the largest index such that $\sum_{j=i+1}^{k} s_j \leq size - s_c$, where s_c is the size of all the items already in configuration c. The items indexed $i+1$ to k are the best remaining items that completely fit in the knapsack. To compute our upper bound for c, then, we consider adding all these elements to c plus as much of item $k+1$ (if it exists) as will fit. Namely, our upper bound for c is defined as follows:

$$\text{upper}(c) = w_c + \sum_{j=i+1}^{k} w_j + \left(size - s_c - \sum_{j=i+1}^{k} s_j \right) \frac{w_{k+1}}{s_{k+1}}.$$

If $k = n-1$, then assume that $w_{k+1}/s_{k+1} = 0$.

We give code fragments from the Java branch-and-bound solution based on this approach in Code Fragments 13.21, 13.22, and 13.23.

```
/**
 * Method to find an optimal solution to KNAPSACK problem, given:
 * - s, indexed array of the sizes
 * - w, index array of element worth (profit)
 * - indexes of s and w are sorted by w[i]/s[i] values
 * - size, the total size constraint
 * It returns an external-node Configuration object for optimal solution.
 */
public static Configuration solve(int[] s, int[] w, long size) {
  /* Create priority queue for selecting current best configurations */
  PriorityQueue p = DoublePriorityQueue();
  /* Create root configuration */
  Configuration root = new Configuration(s,w,size);
  double upper = root.getUpperBound(); // upper bound for root
  Configuration curBest = root; // the current best solution
  p.insertItem(new Double(-upper), root); // add root configuration to p
  /* generate new configurations until all viable solutions are found */
  while (!p.isEmpty()) {
    double curBound = -((Double)p.minKey()).doubleValue(); // we want max
    Configuration curConfig = (Configuration) p.removeMin();
    if (curConfig.getIndex() >= s.length-1) continue; // nothing to expand
    /* Expand this configuration to include the next item */
    Configuration child = curConfig.expandWithNext();
    /* Test if new child has best valid worth seen so far */
    if ((child.getWorth() > curBest.getWorth()) && (child.getSize() <= size))
      curBest = child;
    /* Test if new child is worth expanding further */
    double newBound = child.getUpperBound();
    if (newBound > curBest.getWorth())
      p.insertItem( new Double(-newBound), child);
    /* Expand the current configuration to exclude the next item */
    child = curConfig.expandWithoutNext();
    /* Test if new child is worth expanding further */
    newBound = child.getUpperBound();
    if (newBound > curBest.getWorth())
      p.insertItem( new Double(-newBound), child);
  }
  return curBest;
}
```

Code Fragment 13.21: The engine method for the branch-and-bound solution to the KNAPSACK problem. The class DoublePriorityQueue, which is not shown, is a priority queue specialized to hold objects with Double keys. Note that the key values used in the priority queue are made negative, since we are interested in maximizing worth, not minimizing cost. Also note that when we expand a configuration without adding the next element, we don't check if this is a better solution, since its worth is the same as its parent.

```
class Configuration {
  protected int index; // index of the last element considered
  protected boolean in; // true iff the last element is in the tentative sol'n
  protected long worth; // total worth of all elements in this solution
  protected long size; // total size of all elements in this solution
  protected Configuration parent; // configuration deriving this one
  protected static int[] s;
  protected static int[] w;
  protected static long bagSize;
  /** The initial configuration - is only called for the root config. */
  Configuration(int[] sizes, int[] worths, long sizeConstraint) {
    /* Set static references to the constraints for all configurations */
    s = sizes;
    w = worths;
    bagSize = sizeConstraint;
    /* Set root configuration values */
    index = -1;
    in = false;
    worth = 0L;
    size = 0L;
    parent = null;
  }
  /** Default constructor */
  Configuration() { /* Assume default initial values */ }
  /** Expand this configuration to one that includes next item */
  public Configuration expandWithNext() {
    Configuration c = new Configuration();
    c.index = index + 1;
    c.in = true;
    c.worth = worth + w[c.index];
    c.size = size + s[c.index];
    c.parent = this;
    return c;
  }
  /** Expand this configuration to one that doesn't include next item */
  public Configuration expandWithoutNext() {
    Configuration c = new Configuration();
    c.index = index + 1;
    c.in = false;
    c.worth = worth;
    c.size = size;
    c.parent = this;
    return c;
  }
```

Code Fragment 13.22: The Configuration class and its methods for constructing and expanding configurations. These are used in the branch-and-bound solution to the KNAPSACK problem. This class is continued in Code Fragment 13.23.

```
/** Get this configuration's index */
public long getIndex() {
  return index;
}
/** Get this configuration's size */
public long getSize() {
  return size;
}
/** Get this configuration's worth */
public long getWorth() {
  return worth;
}
/** Get this configuration's upper bound on future potential worth */
public double getUpperBound() {
  int g;  // index for greedy solution
  double bound = worth; // start from current worth
  long curSize=0L;
  long sizeConstraint = bagSize - size;
  /* Greedily add items until remaining size is overflowed */
  for (g=index+1; (curSize <= sizeConstraint) && (g < s.length); g++) {
    curSize += s[g];
    bound += (double) w[g];
    }
  if (g < s.length) {
    bound -= w[g]; // roll back to worth that fit
    /* Add fractional component of the extra greedy item */
    bound += (double) (bagSize - size)*w[g]/s[g];
  }
  return bound;
}
/** Print a solution from this configuration */
public void printSolution() {
  Configuration c = this; // start with external-node Configuration
  System.out.println("(Size,Worth) = " + c.size + "," + c.worth);
  System.out.print("index-size-worth list = [");
  for (; c.parent != null; c = c.parent)  // march up to root
    if (c.in) { // print index, size, and worth of next included item
      System.out.print("(" + c.index);
      System.out.print("," + s[c.index]);
      System.out.print("," + w[c.index] + ")");
    }
  System.out.println("]");
}
```

Code Fragment 13.23: The support methods for the Configuration class, of Code Fragment 13.22, used in the branch-and-bound solution to the KNAPSACK problem. The method for computing upper bounds is particularly important.

13.6 Exercises

Reinforcement

R-13.1 Professor Amongus has shown that a decision problem L is polynomial-time reducible to an ***NP***-complete problem M. Moreover, after 80 pages of dense mathematics, he has also just proven that L can be solved in polynomial time. Has he just proven that $\boldsymbol{P} = \boldsymbol{NP}$? Why or why not?

R-13.2 Use a truth table to convert the Boolean formula $B = (a \leftrightarrow (b+c))$ into an equivalent formula in CNF. Show the truth table and the intermediate DNF formula for $\overline{B}$.

R-13.3 Show that the problem SAT, which takes an arbitrary Boolean formula S as input and asks if S is satisfiable, is ***NP***-complete.

R-13.4 Consider the problem DNF-SAT, which takes a Boolean formula S in disjunctive normal form (DNF) as input and asks if S is satisfiable. Describe a deterministic polynomial-time algorithm for DNF-SAT.

R-13.5 Consider the problem DNF-DISSAT, which takes a Boolean formula S in disjunctive normal form (DNF) as input and asks if S is dissatisfiable, that is, there is an assignment of Boolean values to the variables of S so that it evaluates to 0. Show that DNF-DISSAT is ***NP***-complete.

R-13.6 Convert the Boolean formula $B = (x_1 \leftrightarrow x_2) \cdot (\overline{x_3} + x_4 x_5) \cdot (\overline{x_1 x_2} + x_3 \overline{x_4})$ into CNF.

R-13.7 Show that the CLIQUE problem is in ***NP***.

R-13.8 Given the CNF formula $B = (x_1) \cdot (\overline{x_2} + x_3 + x_5 + \overline{x_6}) \cdot (x_1 + x_4) \cdot (x_3 + \overline{x_5})$, show the reduction of B into an equivalent input for the 3SAT problem.

R-13.9 Given $B = (x_1 + \overline{x_2} + x_3) \cdot (x_4 + x_5 + \overline{x_6}) \cdot (x_1 + \overline{x_4} + \overline{x_5}) \cdot (x_3 + x_4 + x_6)$, draw the instance of VERTEX-COVER that is constructed by the reduction from 3SAT of the Boolean formula B.

R-13.10 Draw an example of a graph with 10 vertices and 15 edges that has a vertex cover of size 2.

R-13.11 Draw an example of a graph with 10 vertices and 15 edges that has a clique of size 6.

R-13.12 Professor Amongus has just designed an algorithm that can take any graph G with n vertices and determine in $O(n^k)$ time whether or not G contains a clique of size k. Does Professor Amongus deserve the Turing Award for having just shown that $\boldsymbol{P} = \boldsymbol{NP}$? Why or why not?

R-13.13 Is there a subset of the numbers in $\{23, 59, 17, 47, 14, 40, 22, 8\}$ that sums to 100? What about 130? Show your work.

R-13.14 Show that the SET-COVER problem is in ***NP***.

R-13.15 Show that the SUBSET-SUM problem is in ***NP***.

R-13.16 Draw an example of a graph with 10 vertices and 20 edges that has a Hamiltonian cycle. Also, draw an example of a graph with 10 vertices and 20 edges that does not have a Hamiltonian cycle.

R-13.17 The ***Manhattan distance*** between two points (a,b) and (c,d) in the plane is $|a-c|+|b-d|$. Using Manhattan distance to define the cost between every pair of points, find an optimal traveling salesperson tour of the following set of points: $\{(1,1),(2,8),(1,5),(3,-4),(5,6),(-2,-6)\}$.

R-13.18 Describe in detail how to implement Algorithm 13.14 in $O(n+m)$ time on an n-vertex graph with m edges. You may use the traditional operation-count measure of running time in this case.

R-13.19 Describe the details of an efficient implementation of Algorithm 13.17 and analyze its running time.

R-13.20 Give an example of a graph G with at least 10 vertices such that the greedy 2-approximation algorithm for VERTEX-COVER given above is guaranteed to produce a suboptimal vertex cover.

R-13.21 Give a complete, weighted graph G, such that its edge weights satisfy the triangle inequality but the MST-based approximation algorithm for TSP does not find an optimal solution.

R-13.22 Give a pseudo-code description of the backtracking algorithm for CNF-SAT.

R-13.23 Give a recursive pseudo-code description of the backtracking algorithm, assuming the search strategy should visit configurations in a depth-first fashion.

R-13.24 Give a pseudo-code description of the branch-and-bound algorithm for TSP.

R-13.25 The branch-and-bound program in Section 13.5.2, for solving the KNAPSACK problem, uses a Boolean flag to determine when an item is included in a solution or not. Show that this flag is redundant. That is, even if we remove this field, there is a way (using no additional fields) to tell if an item is included in a solution or not.

Creativity

C-13.1 Show that we can deterministically simulate in polynomial time any nondeterministic algorithm A that runs in polynomial time and makes at most $O(\log n)$ calls to the choose method, where n is the size of the input to A.

C-13.2 Show that every language L in $\boldsymbol{P}$ is polynomial-time reducible to the language $M = \{5\}$, that is, the language that simply asks if the binary encoding of the input is equal to 5.

C-13.3 Show how to construct a Boolean circuit C such that, if we create variables only for the inputs of C and then try to build a Boolean formula that is equivalent to C, then we will create a formula exponentially larger than an encoding of C.

Hint: Use recursion to repeat subexpressions in a way that doubles their size each time they are used.

C-13.4 Show that the backtracking algorithm given in Section 13.5.1 for the CNF-SAT problem runs in polynomial time if every clause in the given Boolean formula has at most two literals. That is, it solves 2SAT in polynomial time.

C-13.5 Consider the 2SAT version of the CNF-SAT problem, in which every clause in the given formula S has exactly two literals. Note that any clause of the form $(a+b)$ can be thought of as two implications, $(\overline{a} \rightarrow b)$ and $(\overline{b} \rightarrow a)$. Consider a graph G from S, such that each vertex in G is associated with a variable, x, in S, or its negation, $\overline{x}$. Let there be a directed edge in G from $\overline{a}$ to b for each clause equivalent to $(\overline{a} \rightarrow b)$. Show that S is not satisfiable if and only if there is a variable x such that there is a path in G from x to $\overline{x}$ and a path from $\overline{x}$ to x. Derive from this rule a polynomial time algorithm for solving this special case of the CNF-SAT problem. What is the running time of your algorithm?

C-13.6 Suppose an oracle has given you a magic computer, C, that when given any Boolean formula B in CNF will tell you in one step if B is satisfiable or not. Show how to use C to construct an actual assignment of satisfying Boolean values to the variables in any satisfiable formula B. How many calls do you need to make to C in the worst case in order to do this?

C-13.7 Define SUBGRAPH-ISOMORPHISM as the problem that takes a graph G and another graph H and determines if H is a subgraph of G. That is, there is a mapping from each vertex v in H to a vertex $f(v)$ in G such that, if (v,w) is an edge in H, then $(f(v), f(w))$ is an edge in G. Show that SUBGRAPH-ISOMORPHISM is ***NP***-complete.

C-13.8 Define INDEPENDENT-SET as the problem that takes a graph G and an integer k and asks if G contains an independent set of vertices of size k. That is, G contains a set I of vertices of size k such that, for any v and w in I, there is no edge (v,w) in G. Show that INDEPENDENT-SET is ***NP***-complete.

C-13.9 Define HYPER-COMMUNITY to be the problem that takes a collection of n web pages and an integer k, and determines if there are k web pages that all contain hyperlinks to each other. Show that HYPER-COMMUNITY is ***NP***-complete.

C-13.10 Define PARTITION as the problem that takes a set $S = \{s_1, s_2, \ldots, s_n\}$ of numbers and asks if there is a subset T of S such that

$$\sum_{s_i \in T} s_i = \sum_{s_i \in S-T} s_i.$$

That is, it asks if there is a partition of the numbers into two groups that sum to the same value. Show that PARTITION is ***NP***-complete.

C-13.11 Show that the HAMILTONIAN-CYCLE problem on directed graphs is ***NP***-complete.

C-13.12 Show that the SUBSET-SUM problem is solvable in polynomial time if the input is given in a unary encoding. That is, show that SUBSET-SUM is not strongly ***NP***-hard. What is the running time of your algorithm?

C-13.13 Show that the KNAPSACK problem is solvable in polynomial time if the input is given in a unary encoding. That is, show that KNAPSACK is not strongly ***NP***-hard. What is the running time of your algorithm?

C-13.14 Consider the general optimization version of the TSP problem, where the underlying graph need not satisfy the triangle inequality. Show that, for any fixed value $\delta \geq 1$, there is no polynomial-time δ-approximation algorithm for the general TSP problem unless $\boldsymbol{P} = \boldsymbol{NP}$.

Hint: Reduce HAMILTONIAN-CYCLE to this problem by defining a cost function for a complete graph H for the n-vertex input graph G so that edges of H also in G have cost 1 but edges of H not in G have cost δn more than 1.

C-13.15 Consider the special case of TSP where the vertices correspond to points in the plane, with the cost defined on an edge for every pair (p,q) being the usual Euclidean distance between p and q. Show that an optimal tour will not have any pair of crossing edges.

C-13.16 Derive an efficient backtracking algorithm for the HAMILTONIAN-CYCLE problem.

C-13.17 Derive an efficient backtracking algorithm for the KNAPSACK decision problem.

C-13.18 Derive an efficient branch-and-bound algorithm for the KNAPSACK optimization problem.

C-13.19 Derive a new lower bound function, *lb*, for a branch-and-bound algorithm for solving the TSP optimization problem. Your function should always be greater than or equal to the *lb* function used in Section 13.5.2, but still be a valid lower bound function. Describe an example where your *lb* is strictly greater than the *lb* function used in Section 13.5.2.

Projects

P-13.1 Design and implement a backtracking algorithm for the CNF-SAT problem. Compare the running time of your algorithm on a rich set of instances of 2SAT and 3SAT.

P-13.2 Design and implement a branch-and-bound algorithm for the TSP problem. Use at least two different definitions for the lower bound function, *lb*, and test the effectiveness of each.

P-13.3 Possibly working in a group, design and implement a branch-and-bound algorithm for the TSP problem as well as a polynomial-time approximation algorithm for TSP. Test the efficiency and effectiveness of these two implementations for finding traveling salesperson tours for sets of points in the plane with Euclidean distance defining the costs between pairs.

P-13.4 Implement a backtracking algorithm for HAMILTONIAN-CYCLE. For various values of n, test its effectiveness for finding Hamiltonian cycles when the number of edges is $2n$, $\lceil n \log n \rceil$, and $10n$, and $20\lceil n^{1.5} \rceil$.

P-13.5 Do an empirical comparison of using dynamic programming and backtracking for the SUBSET-SUM problem.

P-13.6 Do an empirical comparison of using dynamic programming and branch-and-bound for the KNAPSACK problem.

Chapter Notes

Computing models are discussed in the textbooks by Lewis and Papadimitriou [133], Savage [177] and Sipser [187].

The proof sketch of the Cook-Levin Theorem (13.6) given in this chapter is an adaptation of a proof sketch of Cormen, Leiserson, and Rivest [55]. Cook's original theorem [53] showed that CNF-SAT was ***NP***-complete, and Levin's original theorem [131] was for a tiling problem. We refer to Theorem 13.6 as the "Cook-Levin" theorem in honor of these two seminal papers, for their proofs were along the same lines as the proof sketch given for Theorem 13.6. Karp [113] demonstrated several more problems to be ***NP***-complete, and subsequently hundreds of other problems have been shown to be ***NP***-complete. Garey and Johnson [76] give a very nice discussion of ***NP***-completeness as well as a catalog of many important ***NP***-complete and ***NP***-hard problems.

The reductions given in this chapter that use local replacement and restriction are well-known in the computer science literature; for example, see Garey and Johnson [76] or Aho, Hopcroft, and Ullman [7]. The component design proof that VERTEX-COVER is ***NP***-complete is an adaptation of a proof of Garey and Johnson [76], as is the component design proof that HAMILTONIAN-CYCLE is ***NP***-complete, which itself is a combination of two reductions by Karp [113]. The component design proof that SUBSET-SUM is ***NP***-complete is an adaptation of a proof of Cormen, Leiserson, and Rivest [55].

The discussion of backtracking and branch-and-bound is modeled after discussions by Lewis and Papadimitriou [133] and Brassard and Bratley [38], where backtracking is intended for decision problems and branch-and-bound is for optimization problems. Nevertheless, our discussion is also influenced by Neapolitan and Naimipour [159], who alternatively view backtracking as a heuristic search that uses a depth-first search of a configuration space and branch-and-bound as a heuristic search that uses breadth-first or best-first search with a lower bound function to perform pruning. The technique of backtracking itself dates to early work of Golomb and Baumert [80].

General discussions of approximation algorithms can be found in several other books, including those by Hochbaum [97] and Papadimitriou and Steiglitz [165], as well as the chapter by Klein and Young [116]. The PTAS for KNAPSACK is modeled after a result of Ibarra and Kim [106], as presented by Klein and Young [116]. Papadimitriou and Steiglitz attribute the 2-approximation for VERTEX-COVER to Gavril and Yannakakis. The 2-approximation algorithm for the special case of TSP is due to Rosenkrantz, Stearns, and Lewis [174]. The $O(\log n)$-approximation for SET-COVER, and its proof, follow from work of Chvátal [46], Johnson [109], and Lovász [136].