MODULE-5

11.2 Decision Trees

Many important algorithms, especially those for sorting and searching, work by comparing items of their inputs. We can study the performance of such algorithms with a device called a ***decision tree***. As an example, Figure 11.1 presents a decision tree of an algorithm for finding a minimum of three numbers. Each internal node of a binary decision tree represents a key comparison indicated in the node, e.g., $k < k'$. The node's left subtree contains the information about subsequent comparisons made if $k < k'$, and its right subtree does the same for the case of $k > k'$. (For the sake of simplicity, we assume throughout this section that all input items are distinct.) Each leaf represents a possible outcome of the algorithm's run on some input of size n. Note that the number of leaves can be greater than the number of outcomes because, for some algorithms, the same outcome can be arrived at through a different chain of comparisons. (This happens to be the case for the decision tree in Figure 11.1.) An important point is that the number of leaves must be at least as large as the number of possible outcomes. The algorithm's work on a particular input of size n can be traced by a path from the root to a leaf in its decision tree, and the number of comparisons made by the algorithm on such

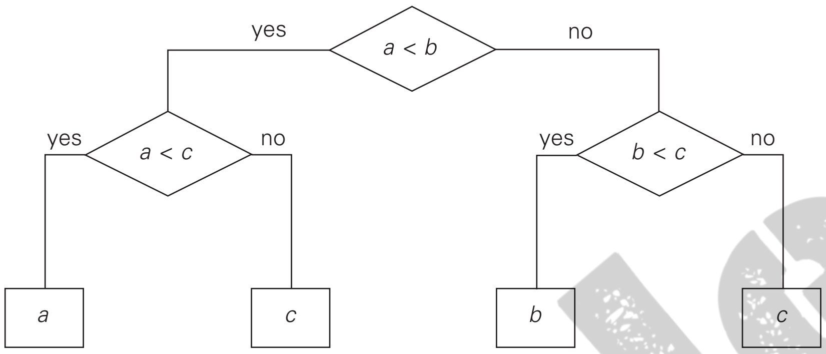

FIGURE 11.1 Decision tree for finding a minimum of three numbers.

a run is equal to the length of this path. Hence, the number of comparisons in the worst case is equal to the height of the algorithm's decision tree.

The central idea behind this model lies in the observation that a tree with a given number of leaves, which is dictated by the number of possible outcomes, has to be tall enough to have that many leaves. Specifically, it is not difficult to prove that for any binary tree with l leaves and height h,

$$h \geq \lceil \log_2 l \rceil. \tag{11.1}$$

Indeed, a binary tree of height h with the largest number of leaves has all its leaves on the last level (why?). Hence, the largest number of leaves in such a tree is 2^h. In other words, $2^h \geq l$, which immediately implies (11.1).

Inequality (11.1) puts a lower bound on the heights of binary decision trees and hence the worst-case number of comparisons made by any comparison-based algorithm for the problem in question. Such a bound is called the ***information-theoretic lower bound*** (see Section 11.1). We illustrate this technique below on two important problems: sorting and searching in a sorted array.

Decision Trees for Sorting

Most sorting algorithms are comparison based, i.e., they work by comparing elements in a list to be sorted. By studying properties of decision trees for such algorithms, we can derive important lower bounds on their time efficiencies.

We can interpret an outcome of a sorting algorithm as finding a permutation of the element indices of an input list that puts the list's elements in ascending order. Consider, as an example, a three-element list a, b, c of orderable items such as real numbers or strings. For the outcome $a < c < b$ obtained by sorting this list (see Figure 11.2), the permutation in question is 1, 3, 2. In general, the number of possible outcomes for sorting an arbitrary n-element list is equal to $n!$.

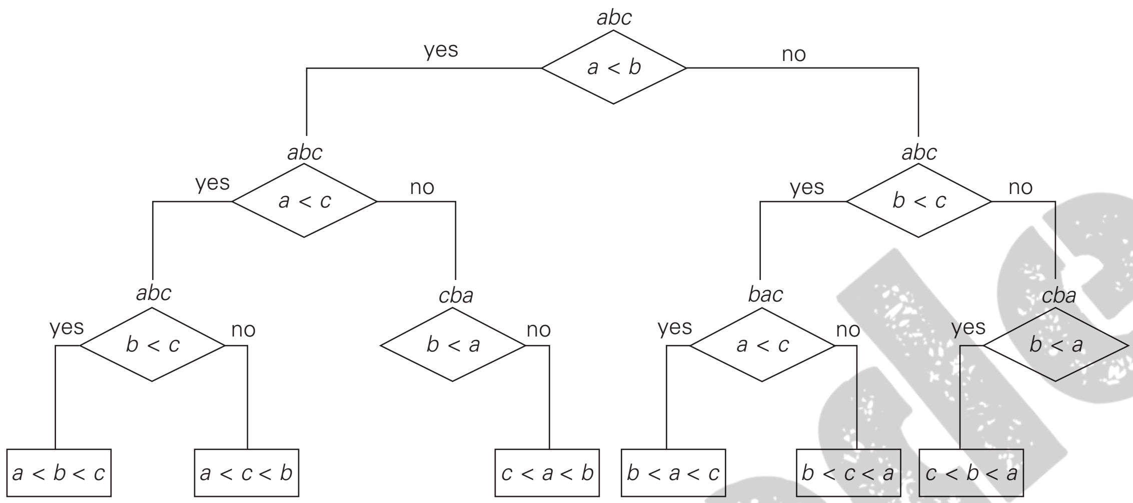

FIGURE 11.2 Decision tree for the tree-element selection sort. A triple above a node indicates the state of the array being sorted. Note two redundant comparisons $b < a$ with a single possible outcome because of the results of some previously made comparisons.

Inequality (11.1) implies that the height of a binary decision tree for any comparison-based sorting algorithm and hence the worst-case number of comparisons made by such an algorithm cannot be less than $\lceil \log_2 n! \rceil$:

$$C_{worst}(n) \geq \lceil \log_2 n! \rceil. \qquad \textbf{(11.2)}$$

Using Stirling's formula for $n!$, we get

$$\lceil \log_2 n! \rceil \approx \log_2 \sqrt{2\pi n}(n/e)^n = n \log_2 n - n \log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\pi}{2} \approx n \log_2 n.$$

In other words, about $n \log_2 n$ comparisons are necessary in the worst case to sort an arbitrary n-element list by any comparison-based sorting algorithm. Note that mergesort makes about this number of comparisons in its worst case and hence is asymptotically optimal. This also implies that the asymptotic lower bound $n \log_2 n$ is tight and therefore cannot be substantially improved. We should point out, however, that the lower bound of $\lceil \log_2 n! \rceil$ can be improved for some values of n. For example, $\lceil \log_2 12! \rceil = 29$, but it has been proved that 30 comparisons are necessary (and sufficient) to sort an array of 12 elements in the worst case.

We can also use decision trees for analyzing the average-case efficiencies of comparison-based sorting algorithms. We can compute the average number of comparisons for a particular algorithm as the average depth of its decision tree's leaves, i.e., as the average path length from the root to the leaves. For example, for

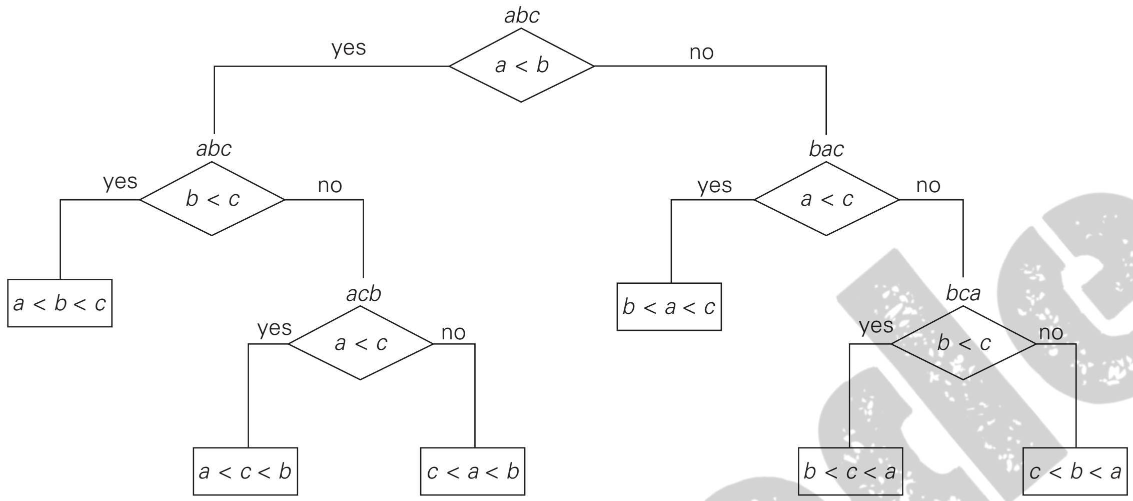

FIGURE 11.3 Decision tree for the three-element insertion sort.

the three-element insertion sort whose decision tree is given in Figure 11.3, this number is $(2 + 3 + 3 + 2 + 3 + 3)/6 = 2\frac{2}{3}$.

Under the standard assumption that all $n!$ outcomes of sorting are equally likely, the following lower bound on the average number of comparisons C_{avg} made by any comparison-based algorithm in sorting an n-element list has been proved:

$$C_{avg}(n) \geq \log_2 n!. \qquad \textbf{(11.3)}$$

As we saw earlier, this lower bound is about $n \log_2 n$. You might be surprised that the lower bounds for the average and worst cases are almost identical. Remember, however, that these bounds are obtained by maximizing the number of comparisons made in the average and worst cases, respectively. For a particular sorting algorithm, the average-case efficiency can, of course, be significantly better than their worst-case efficiency.

Decision Trees for Searching a Sorted Array

In this section, we shall see how decision trees can be used for establishing lower bounds on the number of key comparisons in searching a sorted array of n keys: $A[0] < A[1] < \cdots < A[n-1]$. The principal algorithm for this problem is binary search. As we saw in Section 4.4, the number of comparisons made by binary search in the worst case, $C^{bs}_{worst}(n)$, is given by the formula

$$C^{bs}_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n + 1) \rceil. \qquad \textbf{(11.4)}$$

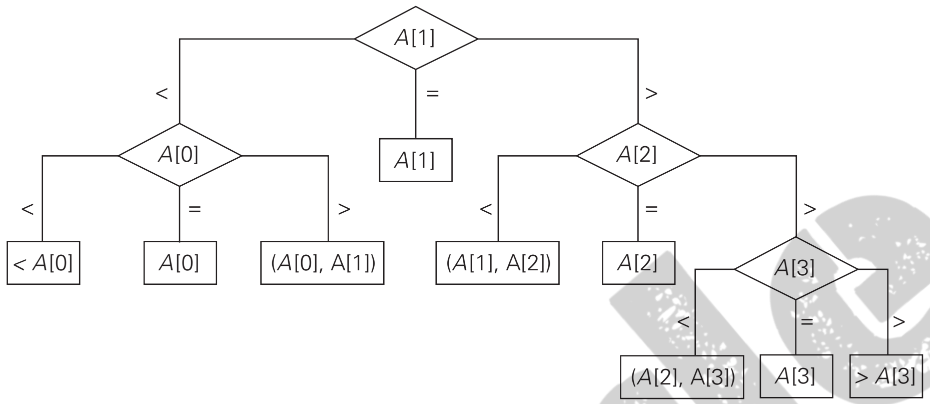

FIGURE 11.4 Ternary decision tree for binary search in a four-element array.

We will use decision trees to determine whether this is the smallest possible number of comparisons.

Since we are dealing here with three-way comparisons in which search key K is compared with some element $A[i]$ to see whether $K < A[i]$, $K = A[i]$, or $K > A[i]$, it is natural to try using ternary decision trees. Figure 11.4 presents such a tree for the case of $n = 4$. The internal nodes of that tree indicate the array's elements being compared with the search key. The leaves indicate either a matching element in the case of a successful search or a found interval that the search key belongs to in the case of an unsuccessful search.

We can represent any algorithm for searching a sorted array by three-way comparisons with a ternary decision tree similar to that in Figure 11.4. For an array of n elements, all such decision trees will have $2n + 1$ leaves (n for successful searches and $n + 1$ for unsuccessful ones). Since the minimum height h of a ternary tree with l leaves is $\lceil \log_3 l \rceil$, we get the following lower bound on the number of worst-case comparisons:

$$C_{worst}(n) \geq \lceil \log_3(2n + 1) \rceil.$$

This lower bound is smaller than $\lceil \log_2(n + 1) \rceil$, the number of worst-case comparisons for binary search, at least for large values of n (and smaller than or equal to $\lceil \log_2(n + 1) \rceil$ for every positive integer n—see Problem 7 in this section's exercises). Can we prove a better lower bound, or is binary search far from being optimal? The answer turns out to be the former. To obtain a better lower bound, we should consider binary rather than ternary decision trees, such as the one in Figure 11.5. Internal nodes in such a tree correspond to the same three-way comparisons as before, but they also serve as terminal nodes for successful searches. Leaves therefore represent only unsuccessful searches, and there are $n + 1$ of them for searching an n-element array.

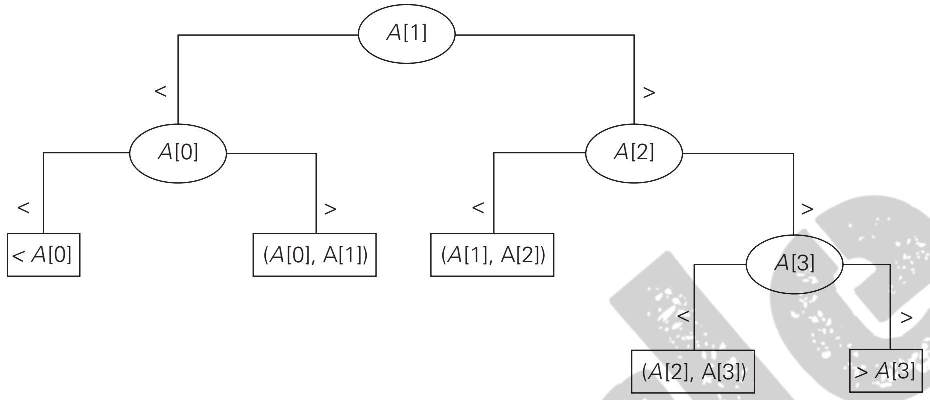

FIGURE 11.5 Binary decision tree for binary search in a four-element array.

As comparison of the decision trees in Figures 11.4 and 11.5 illustrates, the binary decision tree is simply the ternary decision tree with all the middle subtrees eliminated. Applying inequality (11.1) to such binary decision trees immediately yields

$$C_{worst}(n) \geq \lceil \log_2(n+1) \rceil. \tag{11.5}$$

This inequality closes the gap between the lower bound and the number of worst-case comparisons made by binary search, which is also $\lceil \log_2(n+1) \rceil$. A much more sophisticated analysis (see, e.g., [KnuIII, Section 6.2.1]) shows that under the standard assumptions about searches, binary search makes the smallest number of comparisons on the average, as well. The average number of comparisons made by this algorithm turns out to be about $\log_2 n - 1$ and $\log_2(n+1)$ for successful and unsuccessful searches, respectively.

11.3 *P*, *NP*, and *NP*-Complete Problems

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

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

There are several reasons for drawing the intractability line in this way. First, the entries of Table 2.1 and their discussion in Section 2.1 imply that we cannot solve arbitrary instances of intractable problems in a reasonable amount of time unless such instances are very small. Second, although there might be a huge difference between the running times in $O(p(n))$ for polynomials of drastically different degrees, there are very few useful polynomial-time algorithms with the degree of a polynomial higher than three. In addition, polynomials that bound running times of algorithms do not usually have extremely large coefficients. Third, polynomial functions possess many convenient properties; in particular, both the sum and composition of two polynomials are always polynomials too. Fourth, the choice of this class has led to a development of an extensive theory called ***computational complexity***, which seeks to classify problems according to their inherent difficulty. And according to this theory, a problem's intractability remains the same for all principal models of computations and all reasonable input-encoding schemes for the problem under consideration.

We just touch on some basic notions and ideas of complexity theory in this section. If you are interested in a more formal treatment of this theory, you will have no trouble finding a wealth of textbooks devoted to the subject (e.g., [Sip05], [Aro09]).

P and *NP* Problems

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

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

The restriction of P to decision problems can be justified by the following reasons. First, it is sensible to exclude problems not solvable in polynomial time because of their exponentially large output. Such problems do arise naturally—e.g., generating subsets of a given set or all the permutations of n distinct items—but it is apparent from the outset that they cannot be solved in polynomial time. Second, many important problems that are not decision problems in their most natural formulation can be reduced to a series of decision problems that are easier to study. For example, instead of asking about the minimum number of colors needed to color the vertices of a graph so that no two adjacent vertices are colored the same color, we can ask whether there exists such a coloring of the graph's vertices with no more than m colors for $m = 1, 2, \ldots$. (The latter is called the ***m-coloring problem***.) The first value of m in this series for which the decision problem of m-coloring has a solution solves the optimization version of the graph-coloring problem as well.

It is natural to wonder whether *every* decision problem can be solved in polynomial time. The answer to this question turns out to be no. In fact, some decision problems cannot be solved at all by any algorithm. Such problems are called ***undecidable***, as opposed to ***decidable*** problems that can be solved by an algorithm. A famous example of an undecidable problem was given by Alan

Turing in 1936.[1] The problem in question is called the ***halting problem***: given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

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

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

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

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

Then on substituting Q for P, we obtain

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

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

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

There are many important problems, however, for which no polynomial-time algorithm has been found, nor has the impossibility of such an algorithm been proved. The classic monograph by M. Garey and D. Johnson [Gar79] contains a list of several hundred such problems from different areas of computer science, mathematics, and operations research. Here is just a small sample of some of the best-known problems that fall into this category:

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

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

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

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

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

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

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

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

Another common feature of a vast majority of decision problems is the fact that although solving such problems can be computationally difficult, checking whether a proposed solution actually solves the problem is computationally easy, i.e., it can be done in polynomial time. (We can think of such a proposed solution as being randomly generated by somebody leaving us with the task of verifying its validity.) For example, it is easy to check whether a proposed list of vertices is a Hamiltonian circuit for a given graph with n vertices. All we need to check is that the list contains $n + 1$ vertices of the graph in question, that the first n vertices are distinct whereas the last one is the same as the first, and that every consecutive pair of the list's vertices is connected by an edge. This general observation about decision problems has led computer scientists to the notion of a nondeterministic algorithm.

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

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

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

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

Now we can define the class of NP problems.

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

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

$$P \subseteq NP.$$

This is true because, if a problem is in P, we can use the deterministic polynomial-time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simply ignores string S generated in its nondeterministic ("guessing") stage. But NP also contains the Hamiltonian circuit problem, the partition problem, decision versions of the traveling salesman, the knapsack, graph coloring, and many hundreds of other difficult combinatorial optimization problems cataloged in [Gar79]. The halting problem, on the other hand, is among the rare examples of decision problems that are known not to be in NP.

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

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

Note that $P = NP$ would imply that each of many hundreds of difficult combinatorial decision problems can be solved by a polynomial-time algorithm, although computer scientists have failed to find such algorithms despite their persistent efforts over many years. Moreover, many well-known decision problems are known to be "NP-complete" (see below), which seems to cast more doubts on the possibility that $P = NP$.

NP-Complete Problems

Informally, an *NP*-complete problem is a problem in *NP* that is as difficult as any other problem in this class because, by definition, any other problem in *NP* can be reduced to it in polynomial time (shown symbolically in Figure 11.6).

Here are more formal definitions of these concepts.

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

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

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

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

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

The fact that closely related decision problems are polynomially reducible to each other is not very surprising. For example, let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling

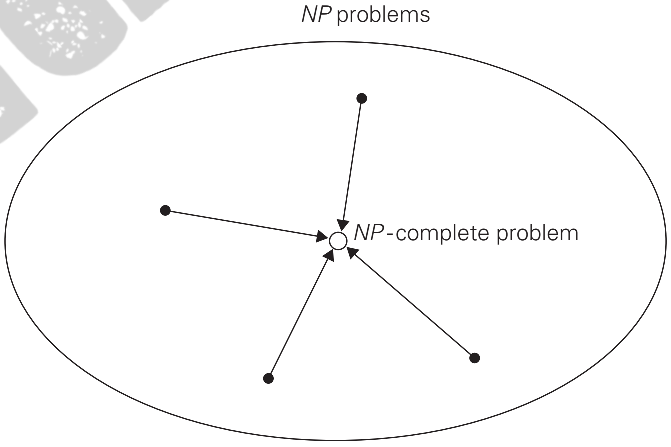

FIGURE 11.6 Notion of an *NP*-complete problem. Polynomial-time reductions of *NP* problems to an *NP*-complete problem are shown by arrows.

salesman problem. The latter can be stated as the existence problem of a Hamiltonian circuit not longer than a given positive integer m in a given complete graph with positive integer weights. We can map a graph G of a given instance of the Hamiltonian circuit problem to a complete weighted graph G' representing an instance of the traveling salesman problem by assigning 1 as the weight to each edge in G and adding an edge of weight 2 between any pair of nonadjacent vertices in G. As the upper bound m on the Hamiltonian circuit length, we take $m = n$, where n is the number of vertices in G (and G'). Obviously, this transformation can be done in polynomial time.

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

The notion of *NP*-completeness requires, however, polynomial reducibility of *all* problems in *NP*, both known and unknown, to the problem in question. Given the bewildering variety of decision problems, it is nothing short of amazing that specific examples of *NP*-complete problems have been actually found. Nevertheless, this mathematical feat was accomplished independently by Stephen Cook in the United States and Leonid Levin in the former Soviet Union.[2] In his 1971 paper, Cook [Coo71] showed that the so-called ***CNF-satisfiability problem*** is *NP*-complete. The CNF-satisfiability problem deals with boolean expressions. Each boolean expression can be represented in conjunctive normal form, such as the following expression involving three boolean variables x_1, x_2, and x_3 and their negations denoted $\bar{x}_1$, $\bar{x}_2$, and $\bar{x}_3$, respectively:

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

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

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

For a while, the leading candidate to be such an example was the problem of determining whether a given integer is prime or composite. But in an important theoretical breakthrough, Professor Manindra Agrawal and his students Neeraj Kayal and Nitin Saxena of the Indian Institute of Technology in Kanpur announced in 2002 a discovery of a deterministic polynomial-time algorithm for primality testing [Agr04]. Their algorithm does not solve, however, the related problem of factoring large composite integers, which lies at the heart of the widely used encryption method called the ***RSA algorithm*** [Riv78].

Showing that a decision problem is *NP*-complete can be done in two steps. First, one needs to show that the problem in question is in *NP*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents a solution to the problem. Typically, this step is easy. The second step is to show that every problem in *NP* is reducible to the problem in question in polynomial time. Because of the transitivity of polynomial reduction, this step can be done by showing that a known *NP*-complete problem can be transformed to the problem in question in polynomial time (see Figure 11.7). Although such a transformation may need to be quite ingenious, it is incomparably simpler than proving the existence of a transformation for every problem in *NP*. For example, if we already know that the Hamiltonian circuit problem is *NP*-complete, its polynomial reducibility to the decision traveling salesman problem implies that the latter is also *NP*-complete (after an easy check that the decision traveling salesman problem is in class *NP*).

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

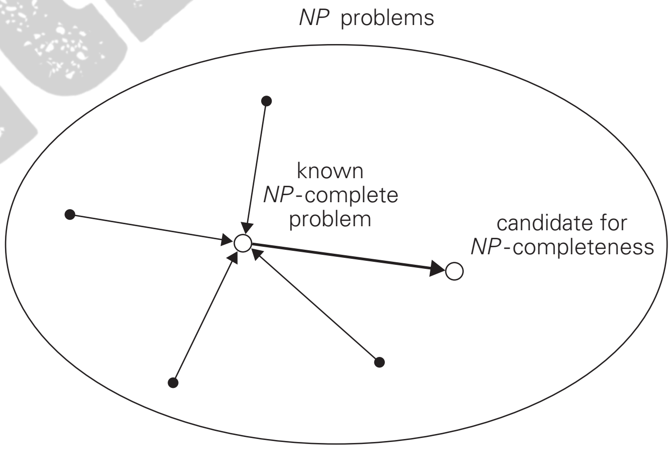

FIGURE 11.7 Proving *NP*-completeness by reduction.

for one *NP*-complete problem would mean that there is no qualitative difference between the complexity of checking a proposed solution and finding it in polynomial time for the vast majority of decision problems of all kinds. Such implications make most computer scientists believe that $P \neq NP$, although nobody has been successful so far in finding a mathematical proof of this intriguing conjecture. Surprisingly, in interviews with the authors of a book about the lives and discoveries of 15 prominent computer scientists [Sha98], Cook seemed to be uncertain about the eventual resolution of this dilemma whereas Levin contended that we should expect the $P = NP$ outcome.

Whatever the eventual answer to the $P \overset{?}{=} NP$ question proves to be, knowing that a problem is *NP*-complete has important practical implications for today. It means that faced with a problem known to be *NP*-complete, we should probably not aim at gaining fame and fortune[3] by designing a polynomial-time algorithm for solving all its instances. Rather, we should concentrate on several approaches that seek to alleviate the intractability of such problems. These approaches are outlined in the next chapter of the book.

12.1 Backtracking

Throughout the book (see in particular Sections 3.4 and 11.3), we have encountered problems that require finding an element with a special property in a domain that grows exponentially fast (or faster) with the size of the problem's input: a Hamiltonian circuit among all permutations of a graph's vertices, the most valuable subset of items for an instance of the knapsack problem, and the like. We addressed in Section 11.3 the reasons for believing that many such problems might not be solvable in polynomial time. Also recall that we discussed in Section 3.4 how such problems can be solved, at least in principle, by exhaustive search. The exhaustive-search technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property.

Backtracking is a more intelligent variation of this approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for *any* remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

It is convenient to implement this kind of processing by constructing a tree of choices being made, called the ***state-space tree***. Its root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution, the nodes of the second level represent the choices for the second component, and so on. A node in a state-space tree is said to be ***promising*** if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called ***nonpromising***. Leaves represent either nonpromising dead ends or complete solutions found by the algorithm. In the majority of cases, a state-space tree for a backtracking algorithm is constructed in the manner of depth-first search. If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child. If the current node turns out to be nonpromising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree, and so on. Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

n-Queens Problem

As our first example, we use a perennial favorite of textbook writers: the ***n*-queens problem**. The problem is to place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. For $n = 1$, the problem has a trivial solution, and it is easy to see that there is no solution for $n = 2$ and $n = 3$. So let us consider the four-queens problem and solve it by the backtracking technique. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board presented in Figure 12.1.

We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem. The state-space tree of this search is shown in Figure 12.2.

If other solutions need to be found (how many of them are there for the four-queens problem?), the algorithm can simply resume its operations at the leaf at which it stopped. Alternatively, we can use the board's symmetry for this purpose.

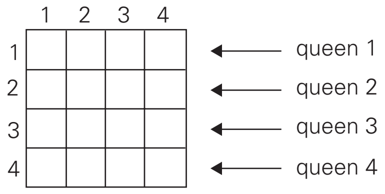

FIGURE 12.1 Board for the four-queens problem.

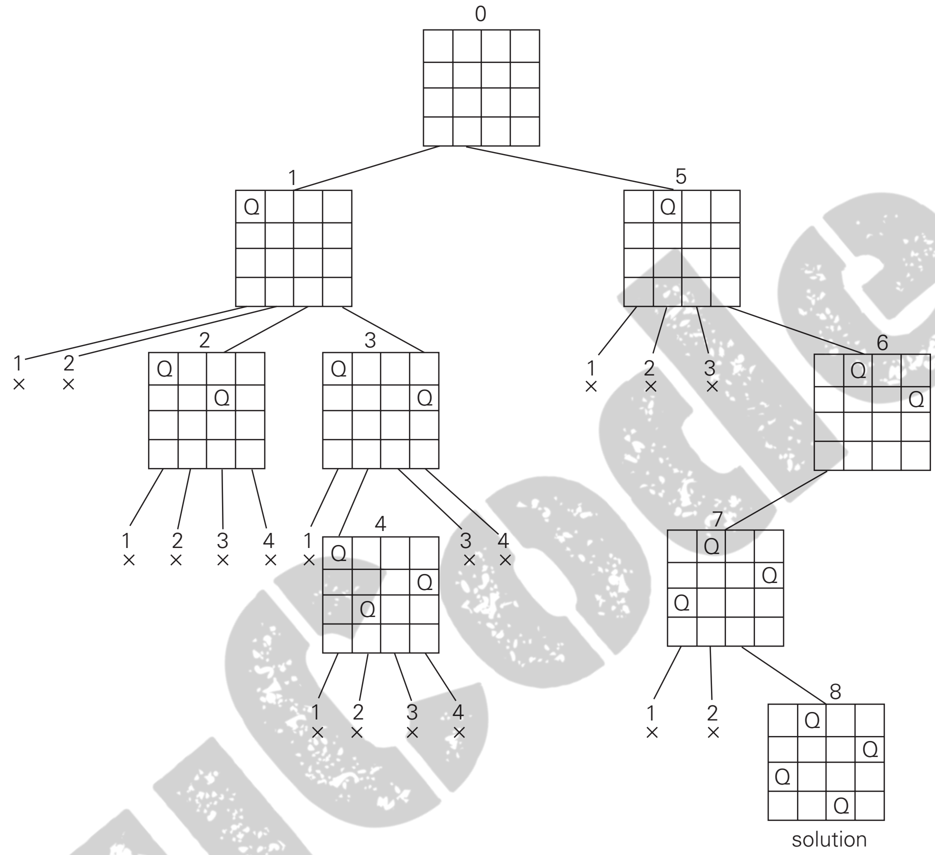

FIGURE 12.2 State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

Finally, it should be pointed out that a single solution to the n-queens problem for any $n \geq 4$ can be found in linear time. In fact, over the last 150 years mathematicians have discovered several alternative formulas for nonattacking positions of n queens [Bel09]. Such positions can also be found by applying some general algorithm design strategies (Problem 4 in this section's exercises).

Hamiltonian Circuit Problem

As our next example, let us consider the problem of finding a Hamiltonian circuit in the graph in Figure 12.3a.

Without loss of generality, we can assume that if a Hamiltonian circuit exists, it starts at vertex a. Accordingly, we make vertex a the root of the state-space

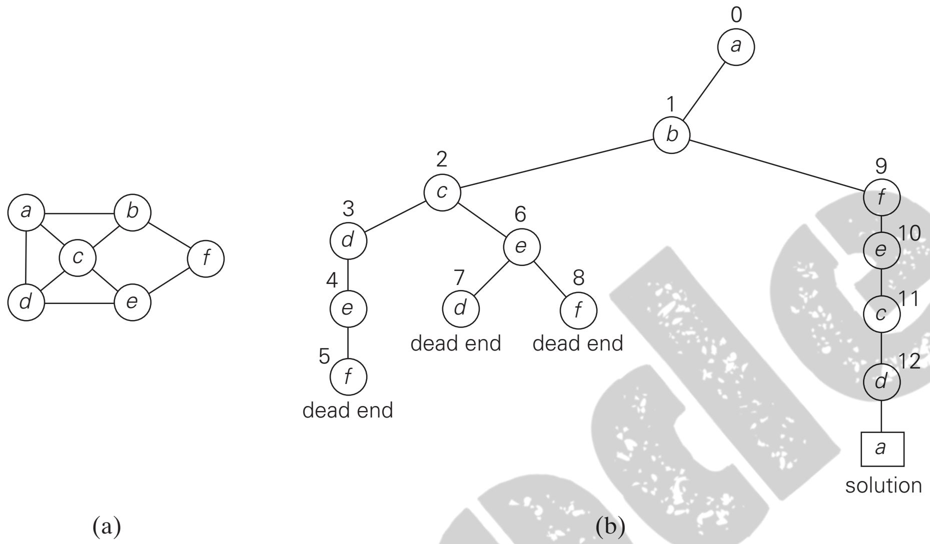

FIGURE 12.3 (a) Graph. (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order in which the nodes are generated.

tree (Figure 12.3b). The first component of our future solution, if it exists, is a first intermediate vertex of a Hamiltonian circuit to be constructed. Using the alphabet order to break the three-way tie among the vertices adjacent to a, we select vertex b. From b, the algorithm proceeds to c, then to d, then to e, and finally to f, which proves to be a dead end. So the algorithm backtracks from f to e, then to d, and then to c, which provides the first alternative for the algorithm to pursue. Going from c to e eventually proves useless, and the algorithm has to backtrack from e to c and then to b. From there, it goes to the vertices f, e, c, and d, from which it can legitimately return to a, yielding the Hamiltonian circuit a, b, f, e, c, d, a. If we wanted to find another Hamiltonian circuit, we could continue this process by backtracking from the leaf of the solution found.

Subset-Sum Problem

As our last example, we consider the ***subset-sum problem***: find a subset of a given set $A = \{a_1, \ldots, a_n\}$ of n positive integers whose sum is equal to a given positive integer d. For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: $\{1, 2, 6\}$ and $\{1, 8\}$. Of course, some instances of this problem may have no solutions.

It is convenient to sort the set's elements in increasing order. So, we will assume that

$$a_1 < a_2 < \cdots < a_n.$$

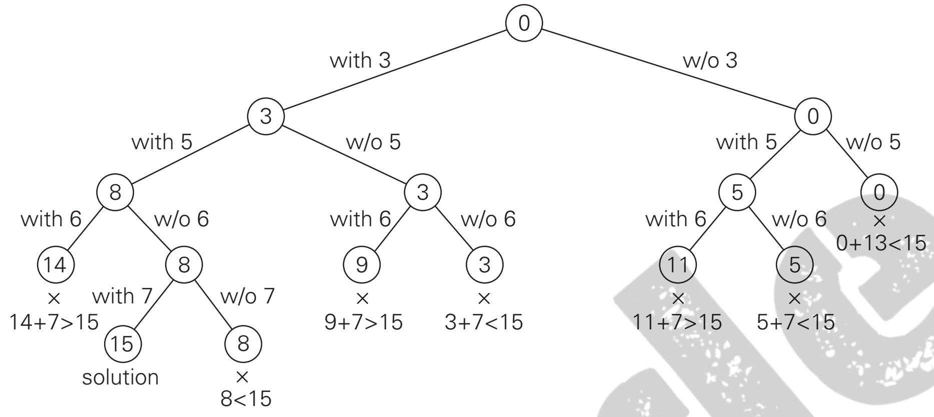

FIGURE 12.4 Complete state-space tree of the backtracking algorithm applied to the instance $A = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

The state-space tree can be constructed as a binary tree like that in Figure 12.4 for the instance $A = \{3, 5, 6, 7\}$ and $d = 15$. The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left and right children represent, respectively, inclusion and exclusion of a_1 in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of a_2 while going to the right corresponds to its exclusion, and so on. Thus, a path from the root to a node on the ith level of the tree indicates which of the first i numbers have been included in the subsets represented by that node.

We record the value of s, the sum of these numbers, in the node. If s is equal to d, we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent. If s is not equal to d, we can terminate the node as nonpromising if either of the following two inequalities holds:

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large)},$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small)}.$$

General Remarks

From a more general perspective, most backtracking algorithms fit the following description. An output of a backtracking algorithm can be thought of as an n-tuple $(x_1, x_2, \ldots, x_n)$ where each coordinate x_i is an element of some finite linearly ordered set S_i. For example, for the n-queens problem, each S_i is the set of integers (column numbers) 1 through n. The tuple may need to satisfy some additional constraints (e.g., the nonattacking requirements in the n-queens problem). Depending on the problem, all solution tuples can be of the same length (the n-queens and the Hamiltonian circuit problem) and of different lengths (the subset-sum problem). A backtracking algorithm generates, explicitly or implicitly, a state-space tree; its nodes represent partially constructed tuples with the first i coordinates defined by the earlier actions of the algorithm. If such a tuple $(x_1, x_2, \dots, x_i)$ is not a solution, the algorithm finds the next element in S_{i+1} that is consistent with the values of $(x_1, x_2, \dots, x_i)$ and the problem's constraints, and adds it to the tuple as its $(i+1)$st coordinate. If such an element does not exist, the algorithm backtracks to consider the next value of x_i, and so on.

To start a backtracking algorithm, the following pseudocode can be called for $i = 0$; $X[1..0]$ represents the empty tuple.

```
ALGORITHM  Backtrack(X[1..i])
    //Gives a template of a generic backtracking algorithm
    //Input: X[1..i] specifies first i promising components of a solution
    //Output: All the tuples representing the problem's solutions
    if X[1..i] is a solution write X[1..i]
    else    //see Problem 9 in this section's exercises
        for each element $x \in S_{i+1}$ consistent with X[1..i] and the constraints do
            $X[i+1] \leftarrow x$
            Backtrack(X[1..i+1])
```

Our success in solving small instances of three difficult problems earlier in this section should not lead you to the false conclusion that backtracking is a very efficient technique. In the worst case, it may have to generate all possible candidates in an exponentially (or faster) growing state space of the problem at hand. The hope, of course, is that a backtracking algorithm will be able to prune enough branches of its state-space tree before running out of time or memory or both. The success of this strategy is known to vary widely, not only from problem to problem but also from one instance to another of the same problem.

There are several tricks that might help reduce the size of a state-space tree. One is to exploit the symmetry often present in combinatorial problems. For example, the board of the n-queens problem has several symmetries so that some solutions can be obtained from others by reflection or rotation. This implies, in particular, that we need not consider placements of the first queen in the last $\lfloor n/2 \rfloor$ columns, because any solution with the first queen in square $(1, i)$, $\lceil n/2 \rceil \le i \le n$, can be obtained by reflection (which?) from a solution with the first queen in square $(1, n - i + 1)$. This observation cuts the size of the tree by about half. Another trick is to preassign values to one or more components of a solution, as we did in the Hamiltonian circuit example. Data presorting in the subset-sum example demonstrates potential benefits of yet another opportunity: rearrange data of an instance given.

It would be highly desirable to be able to estimate the size of the state-space tree of a backtracking algorithm. As a rule, this is too difficult to do analytically, however. Knuth [Knu75] suggested generating a random path from the root to a leaf and using the information about the number of choices available during the path generation for estimating the size of the tree. Specifically, let c_1 be the number of values of the first component x_1 that are consistent with the problem's constraints. We randomly select one of these values (with equal probability $1/c_1$) to move to one of the root's c_1 children. Repeating this operation for c_2 possible values for x_2 that are consistent with x_1 and the other constraints, we move to one of the c_2 children of that node. We continue this process until a leaf is reached after randomly selecting values for $x_1, x_2, \ldots, x_n$. By assuming that the nodes on level i have c_i children on average, we estimate the number of nodes in the tree as

$$1 + c_1 + c_1c_2 + \cdots + c_1c_2 \cdots c_n.$$

Generating several such estimates and computing their average yields a useful estimation of the actual size of the tree, although the standard deviation of this random variable can be large.

In conclusion, three things on behalf of backtracking need to be said. First, it is typically applied to difficult combinatorial problems for which no efficient algorithms for finding exact solutions possibly exist. Second, unlike the exhaustive-search approach, which is doomed to be extremely slow for all instances of a problem, backtracking at least holds a hope for solving some instances of nontrivial sizes in an acceptable amount of time. This is especially true for optimization problems, for which the idea of backtracking can be further enhanced by evaluating the quality of partially constructed solutions. How this can be done is explained in the next section. Third, even if backtracking does not eliminate any elements of a problem's state space and ends up generating all its elements, it provides a specific technique for doing so, which can be of value in its own right.

12.2 Branch-and-Bound

Recall that the central idea of backtracking, discussed in the previous section, is to cut off a branch of the problem's state-space tree as soon as we can deduce that it cannot lead to a solution. This idea can be strengthened further if we deal with an optimization problem. An optimization problem seeks to minimize or maximize some objective function (a tour length, the value of items selected, the cost of an assignment, and the like), usually subject to some constraints. Note that in the standard terminology of optimization problems, a ***feasible solution*** is a point in the problem's search space that satisfies all the problem's constraints (e.g., a Hamiltonian circuit in the traveling salesman problem or a subset of items whose total weight does not exceed the knapsack's capacity in the knapsack problem), whereas an ***optimal solution*** is a feasible solution with the best value of the objective function (e.g., the shortest Hamiltonian circuit or the most valuable subset of items that fit the knapsack).

Compared to backtracking, branch-and-bound requires two additional items:

- a way to provide, for every node of a state-space tree, a bound on the best value of the objective function[1] on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
- the value of the best solution seen so far

If this information is available, we can compare a node's bound value with the value of the best solution seen so far. If the bound value is not better than the value of the best solution seen so far—i.e., not smaller for a minimization problem and not larger for a maximization problem—the node is nonpromising and can be terminated (some people say the branch is "pruned"). Indeed, no solution obtained from it can yield a better solution than the one already available. This is the principal idea of the branch-and-bound technique.

In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

- The value of the node's bound is not better than the value of the best solution seen so far.
- The node represents no feasible solutions because the constraints of the problem are already violated.
- The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Assignment Problem

Let us illustrate the branch-and-bound approach by applying it to the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible. We introduced this problem in Section 3.4, where we solved it by exhaustive search. Recall that an instance of the assignment problem is specified by an $n \times n$ cost matrix C so that we can state the problem as follows: select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible. We will demonstrate how this problem can be solved using the branch-and-bound technique by considering the same small instance of the problem that we investigated in Section 3.4:

$$C = \begin{array}{c} \\ \\ \\ \\ \end{array}\begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \\ \end{array}$$

$$\begin{array}{c} \\ C = \end{array} \begin{array}{c} \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{array}{l} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$

How can we find a lower bound on the cost of an optimal selection without actually solving the problem? We can do this by several methods. For example, it is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows. For the instance here, this sum is $2 + 3 + 1 + 4 = 10$. It is important to stress that this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix); it is just a lower bound on the cost of any legitimate selection. We can and will apply the same thinking to partially constructed solutions. For example, for any legitimate selection that selects 9 from the first row, the lower bound will be $9 + 3 + 1 + 4 = 17$.

One more comment is in order before we embark on constructing the problem's state-space tree. It deals with the order in which the tree nodes will be generated. Rather than generating a single child of the last promising node as we did in backtracking, we will generate all the children of the most promising node among nonterminated leaves in the current tree. (Nonterminated, i.e., still promising, leaves are also called ***live***.) How can we tell which of the nodes is most promising? We can do this by comparing the lower bounds of the live nodes. It is sensible to consider a node with the best bound as most promising, although this does not, of course, preclude the possibility that an optimal solution will ultimately belong to a different branch of the state-space tree. This variation of the strategy is called the ***best-first branch-and-bound***.

So, returning to the instance of the assignment problem given earlier, we start with the root that corresponds to no elements selected from the cost matrix. As we already discussed, the lower-bound value for the root, denoted lb, is 10. The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person a (Figure 12.5).

So we have four live leaves—nodes 1 through 4—that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lower-bound value. Following our best-first search strategy, we branch out from that node first by considering the three different ways of selecting an element from the second row and not in the second column—the three different jobs that can be assigned to person b (Figure 12.6).

Of the six live leaves—nodes 1, 3, 4, 5, 6, and 7—that may contain an optimal solution, we again choose the one with the smallest lower bound, node 5. First, we consider selecting the third column's element from c's row (i.e., assigning person c to job 3); this leaves us with no choice but to select the element from the fourth column of d's row (assigning person d to job 4). This yields leaf 8 (Figure 12.7), which corresponds to the feasible solution $\{a \to 2, b \to 1, c \to 3, d \to 4\}$ with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution $\{a \to 2, b \to 1, c \to 4, d \to 3\}$ with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (Of course, if

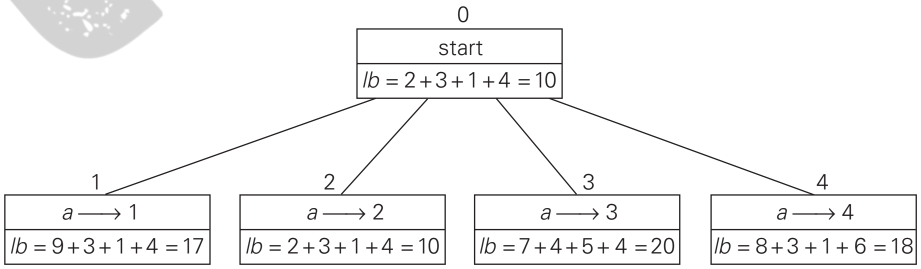

FIGURE 12.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

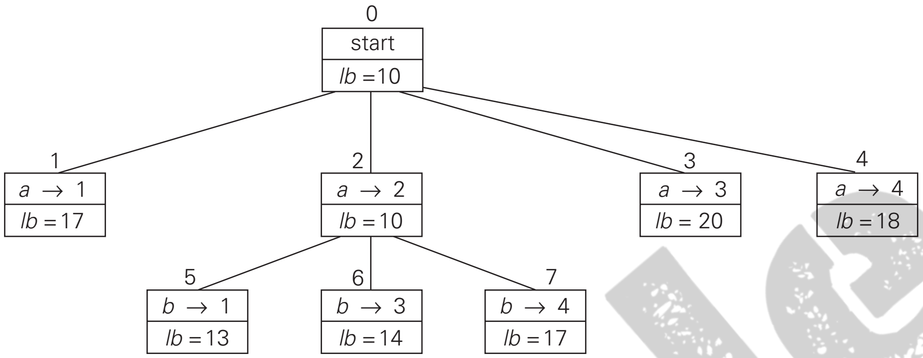

FIGURE 12.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

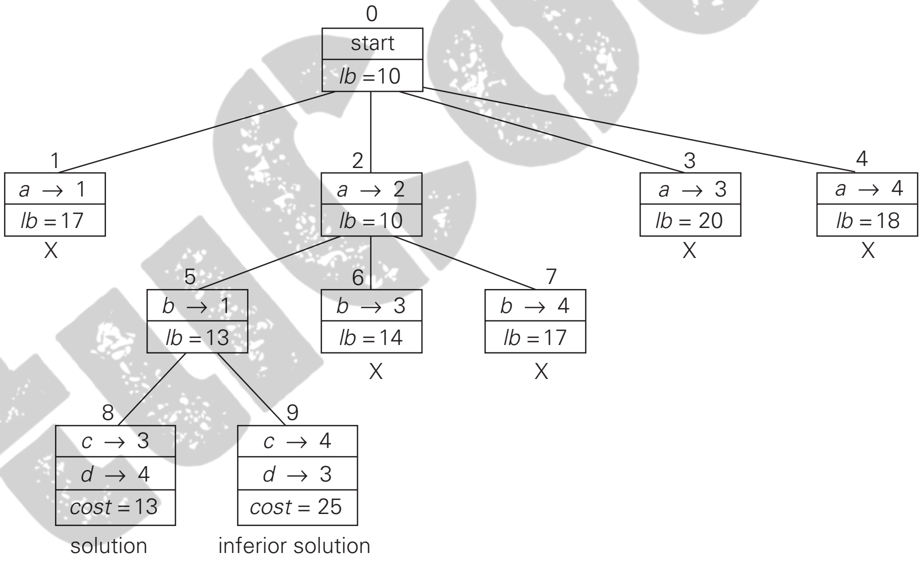

FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.)

Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 in Figure 12.7—we discover that their lower-bound values are not smaller than 13, the value of the best selection seen so far (leaf 8). Hence, we terminate all of them and recognize the solution represented by leaf 8 as the optimal solution to the problem.

Before we leave the assignment problem, we have to remind ourselves again that, unlike for our next examples, there is a polynomial-time algorithm for this problem called the Hungarian method (e.g., [Pap82]). In the light of this efficient algorithm, solving the assignment problem by branch-and-bound should be considered a convenient educational device rather than a practical recommendation.

Knapsack Problem

Let us now discuss how we can apply the branch-and-bound technique to solving the knapsack problem. This problem was introduced in Section 3.4: given n items of known weights w_i and values v_i, $i = 1, 2, \ldots, n$, and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack. It is convenient to order the items of a given instance in descending order by their value-to-weight ratios. Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n.$$

It is natural to structure the state-space tree for this problem as a binary tree constructed as follows (see Figure 12.8 for an example). Each node on the ith level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items. This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion. We record the total weight w and the total value v of this selection in the node, along with some upper bound ub on the value of any subset that can be obtained by adding zero or more items to this selection.

A simple way to compute the upper bound ub is to add to v, the total value of the items already selected, the product of the remaining capacity of the knapsack $W - w$ and the best per unit payoff among the remaining items, which is v_{i+1}/w_{i+1}:

$$ub = v + (W - w)(v_{i+1}/w_{i+1}). \qquad \textbf{(12.1)}$$

As a specific example, let us apply the branch-and-bound algorithm to the same instance of the knapsack problem we solved in Section 3.4 by exhaustive search. (We reorder the items in descending order of their value-to-weight ratios, though.)

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.

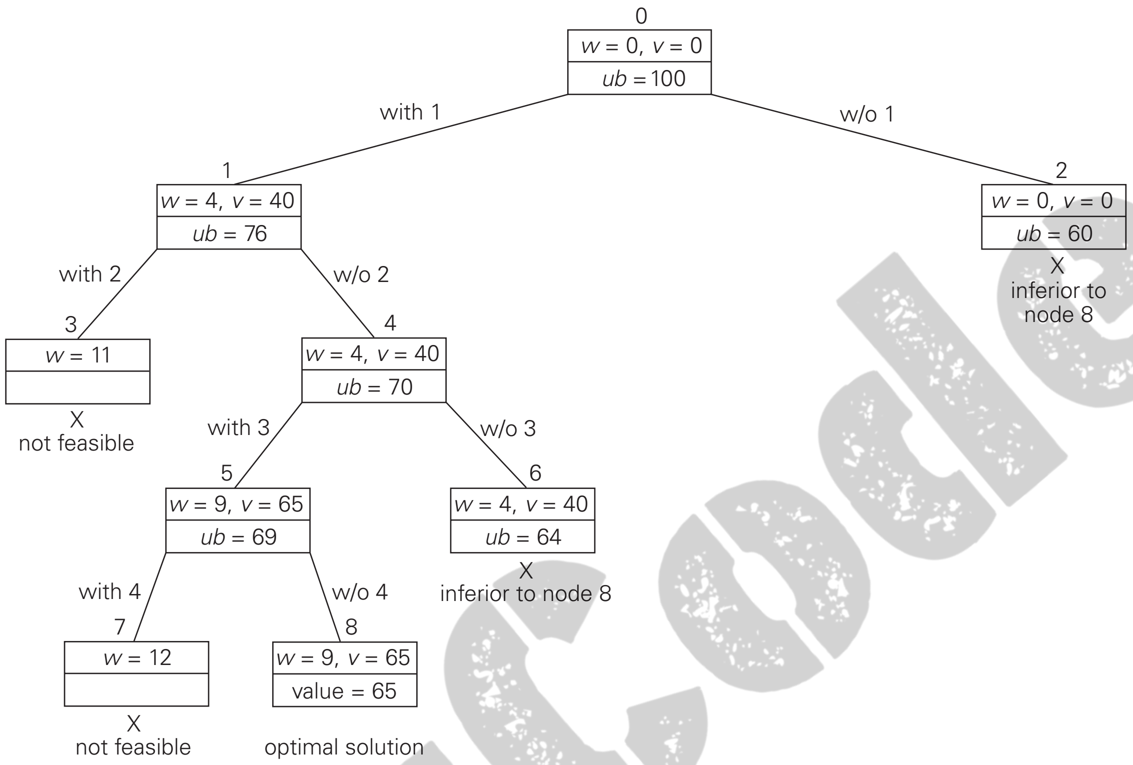

FIGURE 12.8 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

At the root of the state-space tree (see Figure 12.8), no items have been selected as yet. Hence, both the total weight of the items already selected w and their total value v are equal to 0. The value of the upper bound computed by formula (12.1) is \$100. Node 1, the left child of the root, represents the subsets that include item 1. The total weight and value of the items already included are 4 and \$40, respectively; the value of the upper bound is $40 + (10 - 4) * 6 = \$76$. Node 2 represents the subsets that do not include item 1. Accordingly, $w = 0$, $v = \$0$, and $ub = 0 + (10 - 0) * 6 = \$60$. Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem, and we branch from node 1 first. Its children—nodes 3 and 4—represent subsets with item 1 and with and without item 2, respectively. Since the total weight w of every subset represented by node 3 exceeds the knapsack's capacity, node 3 can be terminated immediately. Node 4 has the same values of w and v as its parent; the upper bound ub is equal to $40 + (10 - 4) * 5 = \$70$. Selecting node 4 over node 2 for the next branching (why?), we get nodes 5 and 6 by respectively including and excluding item 3. The total weights and values as well as the upper bounds for these nodes are computed in the same way as for the preceding nodes. Branching from node 5 yields node 7, which represents no feasible solutions, and node 8, which represents just a single subset {1, 3} of value \$65. The remaining live nodes 2 and 6 have smaller upper-bound values than the value of the solution represented by node 8. Hence, both can be terminated making the subset {1, 3} of node 8 the optimal solution to the problem.

Solving the knapsack problem by a branch-and-bound algorithm has a rather unusual characteristic. Typically, internal nodes of a state-space tree do not define a point of the problem's search space, because some of the solution's components remain undefined. (See, for example, the branch-and-bound tree for the assignment problem discussed in the preceding subsection.) For the knapsack problem, however, every node of the tree represents a subset of the items given. We can use this fact to update the information about the best subset seen so far after generating each new node in the tree. If we had done this for the instance investigated above, we could have terminated nodes 2 and 6 before node 8 was generated because they both are inferior to the subset of value \$65 of node 5.

Traveling Salesman Problem

We will be able to apply the branch-and-bound technique to instances of the traveling salesman problem if we come up with a reasonable lower bound on tour lengths. One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix D and multiplying it by the number of cities n. But there is a less obvious and more informative lower bound for instances with symmetric matrix D, which does not require a lot of work to compute. It is not difficult to show (Problem 8 in this section's exercises) that we can compute a lower bound on the length l of any tour as follows. For each city i, $1 \le i \le n$, find the sum s_i of the distances from city i to the two nearest cities; compute the sum s of these n numbers, divide the result by 2, and, if all the distances are integers, round up the result to the nearest integer:

$$lb = \lceil s/2 \rceil. \tag{12.2}$$

For example, for the instance in Figure 12.9a, formula (12.2) yields

$$lb = \lceil[(1+3) + (3+6) + (1+2) + (3+4) + (2+3)]/2\rceil = 14.$$

Moreover, for any subset of tours that must include particular edges of a given graph, we can modify lower bound (12.2) accordingly. For example, for all the Hamiltonian circuits of the graph in Figure 12.9a that must include edge (a, d), we get the following lower bound by summing up the lengths of the two shortest edges incident with each of the vertices, with the required inclusion of edges (a, d) and (d, a):

$$\lceil[(1+5) + (3+6) + (1+2) + (3+5) + (2+3)]/2\rceil = 16.$$

We now apply the branch-and-bound algorithm, with the bounding function given by formula (12.2), to find the shortest Hamiltonian circuit for the graph in

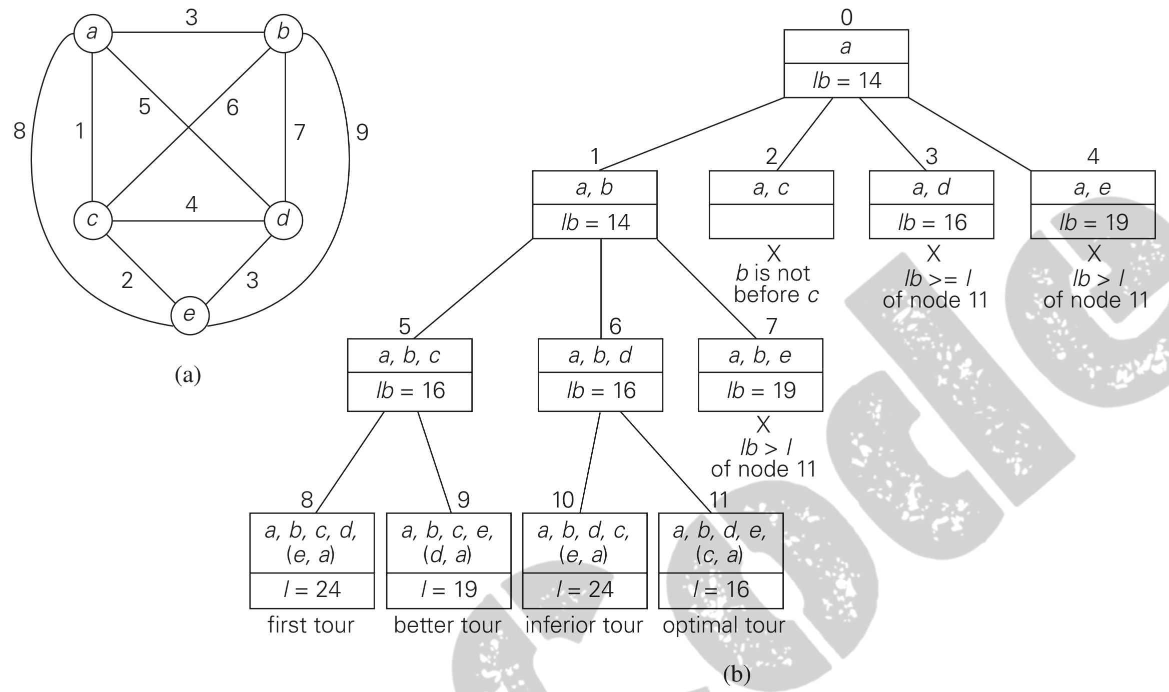

FIGURE 12.9 (a) Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

Figure 12.9a. To reduce the amount of potential work, we take advantage of two observations made in Section 3.4. First, without loss of generality, we can consider only tours that start at a. Second, because our graph is undirected, we can generate only tours in which b is visited before c. In addition, after visiting $n - 1 = 4$ cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one. The state-space tree tracing the algorithm's application is given in Figure 12.9b.

The comments we made at the end of the preceding section about the strengths and weaknesses of backtracking are applicable to branch-and-bound as well. To reiterate the main point: these state-space tree techniques enable us to solve many large instances of difficult combinatorial problems. As a rule, however, it is virtually impossible to predict which instances will be solvable in a realistic amount of time and which will not.

Incorporation of additional information, such as a symmetry of a game's board, can widen the range of solvable instances. Along this line, a branch-and-bound algorithm can be sometimes accelerated by a knowledge of the objective function's value of some nontrivial feasible solution. The information might be obtainable—say, by exploiting specifics of the data or even, for some problems, generated randomly—before we start developing a state-space tree. Then we can use such a solution immediately as the best one seen so far rather than waiting for the branch-and-bound processing to lead us to the first feasible solution.

In contrast to backtracking, solving a problem by branch-and-bound has both the challenge and opportunity of choosing the order of node generation and finding a good bounding function. Though the best-first rule we used above is a sensible approach, it may or may not lead to a solution faster than other strategies. (Artificial intelligence researchers are particularly interested in different strategies for developing state-space trees.)

Finding a good bounding function is usually not a simple task. On the one hand, we want this function to be easy to compute. On the other hand, it cannot be too simplistic—otherwise, it would fail in its principal task to prune as many branches of a state-space tree as soon as possible. Striking a proper balance between these two competing requirements may require intensive experimentation with a wide variety of instances of the problem in question.

12.3 Approximation Algorithms for *NP*-Hard Problems

In this section, we discuss a different approach to handling difficult problems of combinatorial optimization, such as the traveling salesman problem and the knapsack problem. As we pointed out in Section 11.3, the decision versions of these problems are *NP*-complete. Their optimization versions fall in the class of ***NP*-hard problems**—problems that are at least as hard as *NP*-complete problems.[2] Hence, there are no known polynomial-time algorithms for these problems, and there are serious theoretical reasons to believe that such algorithms do not exist. What then are our options for handling such problems, many of which are of significant practical importance?

If an instance of the problem in question is very small, we might be able to solve it by an exhaustive-search algorithm (Section 3.4). Some such problems can be solved by the dynamic programming technique we demonstrated in Section 8.2. But even when this approach works in principle, its practicality is limited by dependence on the instance parameters being relatively small. The discovery of the branch-and-bound technique has proved to be an important breakthrough, because this technique makes it possible to solve many large instances of difficult optimization problems in an acceptable amount of time. However, such good performance cannot usually be guaranteed.

There is a radically different way of dealing with difficult optimization problems: solve them approximately by a fast algorithm. This approach is particularly appealing for applications where a good but not necessarily optimal solution will suffice. Besides, in real-life applications, we often have to operate with inaccurate data to begin with. Under such circumstances, going for an approximate solution can be a particularly sensible choice.

Although approximation algorithms run a gamut in level of sophistication, most of them are based on some problem-specific heuristic. A ***heuristic*** is a common-sense rule drawn from experience rather than from a mathematically proved assertion. For example, going to the nearest unvisited city in the traveling salesman problem is a good illustration of this notion. We discuss an algorithm based on this heuristic later in this section.

Of course, if we use an algorithm whose output is just an approximation of the actual optimal solution, we would like to know how accurate this approximation is. We can quantify the accuracy of an approximate solution s_a to a problem of minimizing some function f by the size of the relative error of this approximation,

$$re(s_a) = \frac{f(s_a) - f(s^*)}{f(s^*)},$$

where s^* is an exact solution to the problem. Alternatively, since $re(s_a) = f(s_a)/f(s^*) - 1$, we can simply use the ***accuracy ratio***

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$

as a measure of accuracy of s_a. Note that for the sake of scale uniformity, the accuracy ratio of approximate solutions to maximization problems is usually computed as

$$r(s_a) = \frac{f(s^*)}{f(s_a)}$$

to make this ratio greater than or equal to 1, as it is for minimization problems.

Obviously, the closer $r(s_a)$ is to 1, the better the approximate solution is. For most instances, however, we cannot compute the accuracy ratio, because we typically do not know $f(s^*)$, the true optimal value of the objective function. Therefore, our hope should lie in obtaining a good upper bound on the values of $r(s_a)$. This leads to the following definitions.

DEFINITION A polynomial-time approximation algorithm is said to be a ***c-approximation algorithm***, where $c \geq 1$, if the accuracy ratio of the approximation it produces does not exceed c for any instance of the problem in question:

$$r(s_a) \leq c. \qquad \textbf{(12.3)}$$

The best (i.e., the smallest) value of c for which inequality (12.3) holds for all instances of the problem is called the ***performance ratio*** of the algorithm and denoted R_A.

The performance ratio serves as the principal metric indicating the quality of the approximation algorithm. We would like to have approximation algorithms with R_A as close to 1 as possible. Unfortunately, as we shall see, some approximation algorithms have infinitely large performance ratios ($R_A = \infty$). This does not necessarily rule out using such algorithms, but it does call for a cautious treatment of their outputs.

There are two important facts about difficult combinatorial optimization problems worth keeping in mind. First, although the difficulty level of solving most such problems exactly is the same to within a polynomial-time transformation of one problem to another, this equivalence does not translate into the realm of approximation algorithms. Finding good approximate solutions is much easier for some of these problems than for others. Second, some of the problems have special classes of instances that are both particularly important for real-life applications and easier to solve than their general counterparts. The traveling salesman problem is a prime example of this situation.

Approximation Algorithms for the Traveling Salesman Problem

We solved the traveling salesman problem by exhaustive search in Section 3.4, mentioned its decision version as one of the most well-known *NP*-complete problems in Section 11.3, and saw how its instances can be solved by a branch-and-bound algorithm in Section 12.2. Here, we consider several approximation algorithms, a small sample of dozens of such algorithms suggested over the years for this famous problem. (For a much more detailed discussion of the topic, see [Law85], [Hoc97], [App07], and [Gut07].)

But first let us answer the question of whether we should hope to find a polynomial-time approximation algorithm with a finite performance ratio on all instances of the traveling salesman problem. As the following theorem [Sah76] shows, the answer turns out to be no, unless $P = NP$.

THEOREM 1 If $P \neq NP$, there exists no c-approximation algorithm for the traveling salesman problem, i.e., there exists no polynomial-time approximation algorithm for this problem so that for all instances

$$f(s_a) \leq cf(s^*)$$

for some constant c.

PROOF By way of contradiction, suppose that such an approximation algorithm A and a constant c exist. (Without loss of generality, we can assume that c is a positive integer.) We will show that this algorithm could then be used for solving the Hamiltonian circuit problem in polynomial time. We will take advantage of a variation of the transformation used in Section 11.3 to reduce the Hamiltonian circuit problem to the traveling salesman problem. Let G be an arbitrary graph with n vertices. We map G to a complete weighted graph G' by assigning weight 1 to each edge in G and adding an edge of weight $cn + 1$ between each pair of vertices not adjacent in G. If G has a Hamiltonian circuit, its length in G' is n; hence, it is the exact solution s^* to the traveling salesman problem for G'. Note that if s_a is an approximate solution obtained for G' by algorithm A, then $f(s_a) \leq cn$ by the assumption. If G does not have a Hamiltonian circuit in G, the shortest tour in G' will contain at least one edge of weight $cn + 1$, and hence $f(s_a) \geq f(s^*) > cn$. Taking into account the two derived inequalities, we could solve the Hamiltonian circuit problem for graph G in polynomial time by mapping G to G', applying algorithm A to get tour s_a in G', and comparing its length with cn. Since the Hamiltonian circuit problem is NP-complete, we have a contradiction unless $P = NP$. ■

Greedy Algorithms for the TSP The simplest approximation algorithms for the traveling salesman problem are based on the greedy technique. We will discuss here two such algorithms.

Nearest-neighbor algorithm

The following well-known greedy algorithm is based on the ***nearest-neighbor*** heuristic: always go next to the nearest unvisited city.

Step 1 Choose an arbitrary city as the start.
Step 2 Repeat the following operation until all the cities have been visited: go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).
Step 3 Return to the starting city.

EXAMPLE 1 For the instance represented by the graph in Figure 12.10, with a as the starting vertex, the nearest-neighbor algorithm yields the tour (Hamiltonian circuit) s_a: $a - b - c - d - a$ of length 10.

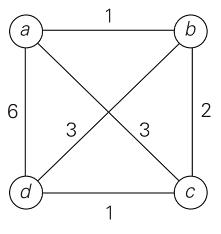

FIGURE 12.10 Instance of the traveling salesman problem.

The optimal solution, as can be easily checked by exhaustive search, is the tour s^*: $a - b - d - c - a$ of length 8. Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

(i.e., tour s_a is 25% longer than the optimal tour s^*). ■

Unfortunately, except for its simplicity, not many good things can be said about the nearest-neighbor algorithm. In particular, nothing can be said in general about the accuracy of solutions obtained by this algorithm because it can force us to traverse a very long edge on the last leg of the tour. Indeed, if we change the weight of edge (a, d) from 6 to an arbitrary large number $w \geq 6$ in Example 1, the algorithm will still yield the tour $a - b - c - d - a$ of length $4 + w$, and the optimal solution will still be $a - b - d - c - a$ of length 8. Hence,

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4 + w}{8},$$

which can be made as large as we wish by choosing an appropriately large value of w. Hence, $R_A = \infty$ for this algorithm (as it should be according to Theorem 1).

Multifragment-heuristic algorithm

Another natural greedy algorithm for the traveling salesman problem considers it as the problem of finding a minimum-weight collection of edges in a given complete weighted graph so that all the vertices have degree 2. (With this emphasis on edges rather than vertices, what other greedy algorithm does it remind you of?) An application of the greedy technique to this problem leads to the following algorithm [Ben90].

Step 1 Sort the edges in increasing order of their weights. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.

Step 2 Repeat this step n times, where n is the number of cities in the instance being solved: add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.

Step 3 Return the set of tour edges.

As an example, applying the algorithm to the graph in Figure 12.10 yields $\{(a, b), (c, d), (b, c), (a, d)\}$. This set of edges forms the same tour as the one produced by the nearest-neighbor algorithm. In general, the multifragment-heuristic algorithm tends to produce significantly better tours than the nearest-neighbor algorithm, as we are going to see from the experimental data quoted at the end of this section. But the performance ratio of the multifragment-heuristic algorithm is also unbounded, of course.

There is, however, a very important subset of instances, called ***Euclidean***, for which we can make a nontrivial assertion about the accuracy of both the nearest-neighbor and multifragment-heuristic algorithms. These are the instances in which intercity distances satisfy the following natural conditions:

- *triangle inequality* $d[i, j] \le d[i, k] + d[k, j]$ for any triple of cities i, j, and k (the distance between cities i and j cannot exceed the length of a two-leg path from i to some intermediate city k to j)
- *symmetry* $d[i, j] = d[j, i]$ for any pair of cities i and j (the distance from i to j is the same as the distance from j to i)

A substantial majority of practical applications of the traveling salesman problem are its Euclidean instances. They include, in particular, geometric ones, where cities correspond to points in the plane and distances are computed by the standard Euclidean formula. Although the performance ratios of the nearest-neighbor and multifragment-heuristic algorithms remain unbounded for Euclidean instances, their accuracy ratios satisfy the following inequality for any such instance with $n \ge 2$ cities:

$$\frac{f(s_a)}{f(s^*)} \le \frac{1}{2}(\lceil \log_2 n \rceil + 1),$$

where $f(s_a)$ and $f(s^*)$ are the lengths of the heuristic tour and shortest tour, respectively (see [Ros77] and [Ong84]).

Minimum-Spanning-Tree–Based Algorithms There are approximation algorithms for the traveling salesman problem that exploit a connection between Hamiltonian circuits and spanning trees of the same graph. Since removing an edge from a Hamiltonian circuit yields a spanning tree, we can expect that the structure of a minimum spanning tree provides a good basis for constructing a shortest tour approximation. Here is an algorithm that implements this idea in a rather straightforward fashion.

Twice-around-the-tree algorithm

Step 1 Construct a minimum spanning tree of the graph corresponding to a given instance of the traveling salesman problem.

Step 2 Starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by. (This can be done by a DFS traversal.)

Step 3 Scan the vertex list obtained in Step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list. (This step is equivalent to making shortcuts in the walk.) The vertices remaining on the list will form a Hamiltonian circuit, which is the output of the algorithm.

EXAMPLE 2 Let us apply this algorithm to the graph in Figure 12.11a. The minimum spanning tree of this graph is made up of edges (a, b), (b, c), (b, d), and (d, e) (Figure 12.11b). A twice-around-the-tree walk that starts and ends at a is

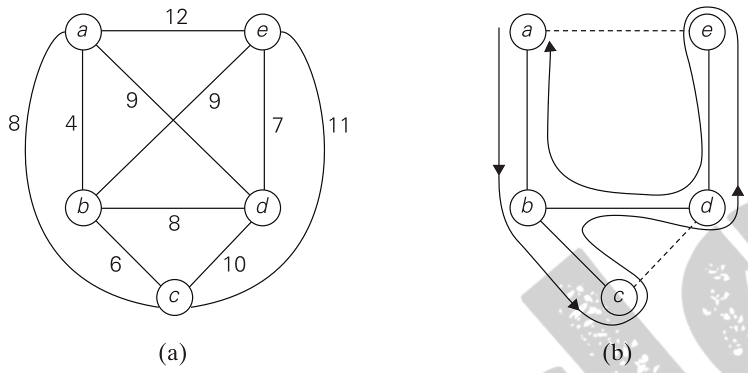

FIGURE 12.11 Illustration of the twice-around-the-tree algorithm. (a) Graph. (b) Walk around the minimum spanning tree with the shortcuts.

$$a,\ b,\ c,\ b,\ d,\ e,\ d,\ b,\ a.$$

Eliminating the second b (a shortcut from c to d), the second d, and the third b (a shortcut from e to a) yields the Hamiltonian circuit

$$a,\ b,\ c,\ d,\ e,\ a$$

of length 39. ■

The tour obtained in Example 2 is not optimal. Although that instance is small enough to find an optimal solution by either exhaustive search or branch-and-bound, we refrained from doing so to reiterate a general point. As a rule, we do not know what the length of an optimal tour actually is, and therefore we cannot compute the accuracy ratio $f(s_a)/f(s^*)$. For the twice-around-the-tree algorithm, we can at least estimate it above, provided the graph is Euclidean.

THEOREM 2 The twice-around-the-tree algorithm is a 2-approximation algorithm for the traveling salesman problem with Euclidean distances.

PROOF Obviously, the twice-around-the-tree algorithm is polynomial time if we use a reasonable algorithm such as Prim's or Kruskal's in Step 1. We need to show that for any Euclidean instance of the traveling salesman problem, the length of a tour s_a obtained by the twice-around-the-tree algorithm is at most twice the length of the optimal tour s^*, i.e.,

$$f(s_a) \le 2f(s^*).$$

Since removing any edge from s^* yields a spanning tree T of weight $w(T)$, which must be greater than or equal to the weight of the graph's minimum spanning tree $w(T^*)$, we get the inequality

$$f(s^*) > w(T) \ge w(T^*).$$

This inequality implies that

$$2f(s^*) > 2w(T^*) = \text{the length of the walk obtained in Step 2 of the algorithm.}$$

The possible shortcuts outlined in Step 3 of the algorithm to obtain s_a cannot increase the total length of the walk in a Euclidean graph, i.e.,

$$\text{the length of the walk obtained in Step 2} \geq \text{the length of the tour } s_a.$$

Combining the last two inequalities, we get the inequality

$$2f(s^*) > f(s_a),$$

which is, in fact, a slightly stronger assertion than the one we needed to prove. ■

Christofides Algorithm There is an approximation algorithm with a better performance ratio for the Euclidean traveling salesman problem—the well-known ***Christofides algorithm*** [Chr76]. It also uses a minimum spanning tree but does this in a more sophisticated way than the twice-around-the-tree algorithm. Note that a twice-around-the-tree walk generated by the latter algorithm is an Eulerian circuit in the multigraph obtained by doubling every edge in the graph given. Recall that an Eulerian circuit exists in a connected multigraph if and only if all its vertices have even degrees. The Christofides algorithm obtains such a multigraph by adding to the graph the edges of a minimum-weight matching of all the odd-degree vertices in its minimum spanning tree. (The number of such vertices is always even and hence this can always be done.) Then the algorithm finds an Eulerian circuit in the multigraph and transforms it into a Hamiltonian circuit by shortcuts, exactly the same way it is done in the last step of the twice-around-the-tree algorithm.

EXAMPLE 3 Let us trace the Christofides algorithm in Figure 12.12 on the same instance (Figure 12.12a) used for tracing the twice-around-the-tree algorithm in Figure 12.11. The graph's minimum spanning tree is shown in Figure 12.12b. It has four odd-degree vertices: a, b, c, and e. The minimum-weight matching of these four vertices consists of edges (a, b) and (c, e). (For this tiny instance, it can be found easily by comparing the total weights of just three alternatives: (a, b) and (c, e), (a, c) and (b, e), (a, e) and (b, c).) The traversal of the multigraph, starting at vertex a, produces the Eulerian circuit $a - b - c - e - d - b - a$, which, after one shortcut, yields the tour $a - b - c - e - d - a$ of length 37. ■

The performance ratio of the Christofides algorithm on Euclidean instances is 1.5 (see, e.g., [Pap82]). It tends to produce significantly better approximations to optimal tours than the twice-around-the-tree algorithm does in empirical tests. (We quote some results of such tests at the end of this subsection.) The quality of a tour obtained by this heuristic can be further improved by optimizing shortcuts made on the last step of the algorithm as follows: examine the multiply-visited cities in some arbitrary order and for each make the best possible shortcut. This

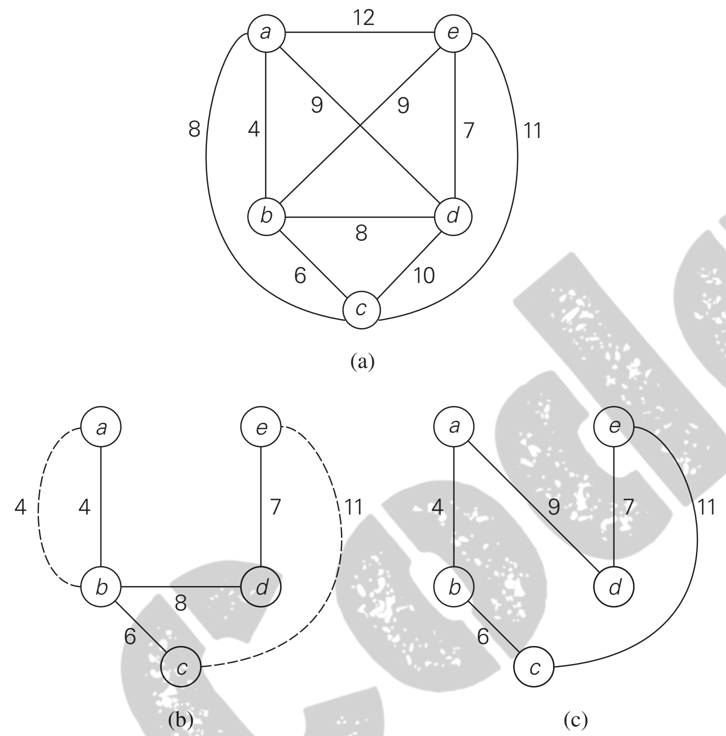

FIGURE 12.12 Application of the Christofides algorithm. (a) Graph. (b) Minimum spanning tree with added edges (in dash) of a minimum-weight matching of all odd-degree vertices. (c) Hamiltonian circuit obtained.

enhancement would have not improved the tour $a - b - c - e - d - a$ obtained in Example 3 from $a - b - c - e - d - b - a$ because shortcutting the second occurrence of b happens to be better than shortcutting its first occurrence. In general, however, this enhancement tends to decrease the gap between the heuristic and optimal tour lengths from about 15% to about 10%, at least for randomly generated Euclidean instances [Joh07a].

Local Search Heuristics For Euclidean instances, surprisingly good approximations to optimal tours can be obtained by iterative-improvement algorithms, which are also called ***local search*** heuristics. The best-known of these are the ***2-opt***, ***3-opt***, and ***Lin-Kernighan*** algorithms. These algorithms start with some initial tour, e.g., constructed randomly or by some simpler approximation algorithm such as the nearest-neighbor. On each iteration, the algorithm explores a neighborhood around the current tour by replacing a few edges in the current tour by other edges. If the changes produce a shorter tour, the algorithm makes it the current

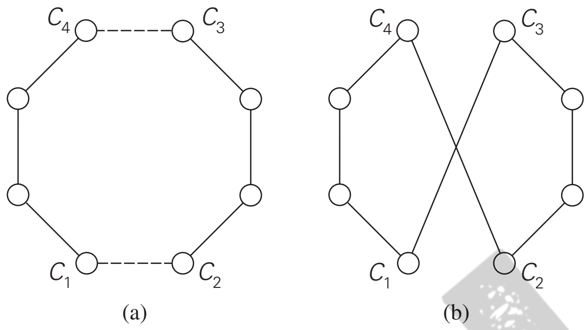

FIGURE 12.13 2-change: (a) Original tour. (b) New tour.

tour and continues by exploring its neighborhood in the same manner; otherwise, the current tour is returned as the algorithm's output and the algorithm stops.

The 2-opt algorithm works by deleting a pair of nonadjacent edges in a tour and reconnecting their endpoints by the different pair of edges to obtain another tour (see Figure 12.13). This operation is called the ***2-change***. Note that there is only one way to reconnect the endpoints because the alternative produces two disjoint fragments.

EXAMPLE 4 If we start with the nearest-neighbor tour $a - b - c - d - e - a$ in the graph of Figure 12.11, whose length l_{nn} is equal to 39, the 2-opt algorithm will move to the next tour as shown in Figure 12.14. ■

To generalize the notion of the 2-change, one can consider the ***k-change*** for any $k \geq 2$. This operation replaces up to k edges in a current tour. In addition to 2-changes, only the 3-changes have proved to be of practical interest. The two principal possibilities of 3-changes are shown in Figure 12.15.

There are several other local search algorithms for the traveling salesman problem. The most prominent of them is the ***Lin-Kernighan*** algorithm [Lin73], which for two decades after its publication in 1973 was considered the best algorithm to obtain high-quality approximations of optimal tours. The Lin-Kernighan algorithm is a variable-opt algorithm: its move can be viewed as a 3-opt move followed by a sequence of 2-opt moves. Because of its complexity, we have to refrain from discussing this algorithm here. The excellent survey by Johnson and McGeoch [Joh07a] contains an outline of the algorithm and its modern extensions as well as methods for its efficient implementation. This survey also contain results from the important empirical studies about performance of many heuristics for the traveling salesman problem, including of course, the Lin-Kernighan algorithm. We conclude our discussion by quoting some of these data.

Empirical Results The traveling salesman problem has been the subject of intense study for the last 50 years. This interest was driven by a combination of pure

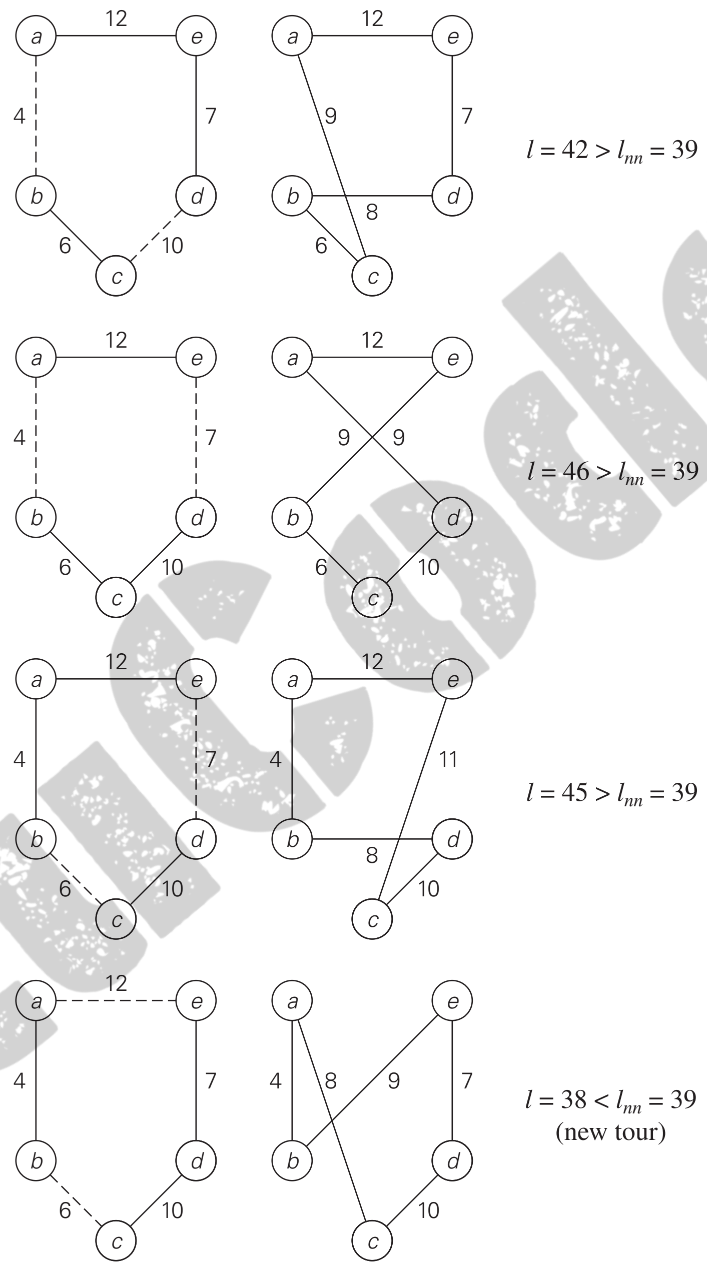

FIGURE 12.14 2-changes from the nearest-neighbor tour of the graph in Figure 12.11.

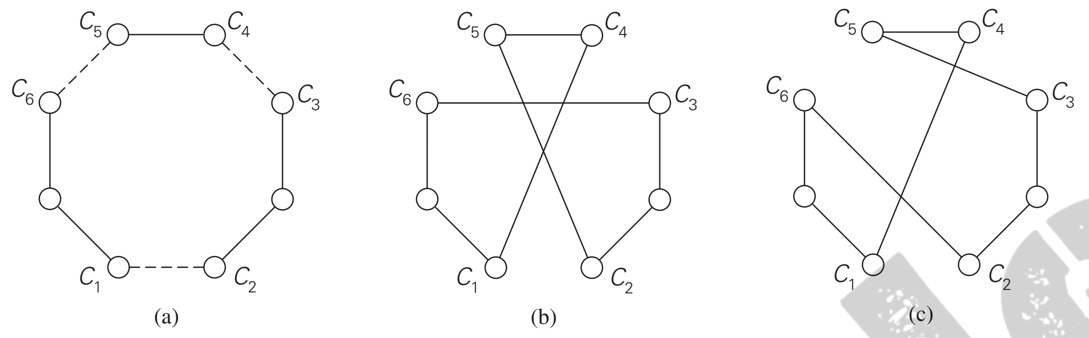

FIGURE 12.15 3-change: (a) Original tour. (b), (c) New tours.

theoretical interest and serious practical needs stemming from such newer applications as circuit-board and VLSI-chip fabrication, X-ray crystallography, and genetic engineering. Progress in developing effective heuristics, their efficient implementation by using sophisticated data structures, and the ever-increasing power of computers have led to a situation that differs drastically from a pessimistic picture painted by the worst-case theoretical results. This is especially true for the most important applications class of instances of the traveling salesman problem: points in the two-dimensional plane with the standard Euclidean distances between them.

Nowadays, Euclidean instances with up to 1000 cities can be solved exactly in quite a reasonable amount of time—typically, in minutes or faster on a good workstation—by such optimization packages as *Concord* [App]. In fact, according to the information on the Web site maintained by the authors of that package, the largest instance of the traveling salesman problem solved exactly as of January 2010 was a tour through 85,900 points in a VLSI application. It significantly exceeded the previous record of the shortest tour through all 24,978 cities in Sweden. There should be little doubt that the latest record will also be eventually superseded and our ability to solve ever larger instances exactly will continue to expand. This remarkable progress does not eliminate the usefulness of approximation algorithms for such problems, however. First, some applications lead to instances that are still too large to be solved exactly in a reasonable amount of time. Second, one may well prefer spending seconds to find a tour that is within a few percent of optimum than to spend many hours or even days of computing time to find the shortest tour exactly.

But how can one tell how good or bad the approximate solution is if we do not know the length of an optimal tour? A convenient way to overcome this difficulty is to solve the linear programming problem describing the instance in question by ignoring the integrality constraints. This provides a lower bound—called the ***Held-Karp bound***—on the length of the shortest tour. The Held-Karp bound is typically very close (less than 1%) to the length of an optimal tour, and this bound can be computed in seconds or minutes unless the instance is truly huge. Thus, for a tour

TABLE 12.1 Average tour quality and running times for various heuristics on the 10,000-city random uniform Euclidean instances [Joh07a]

Heuristic	% excess over the Held-Karp bound	Running time (seconds)
nearest neighbor	24.79	0.28
multifragment	16.42	0.20
Christofides	9.81	1.04
2-opt	4.70	1.41
3-opt	2.88	1.50
Lin-Kernighan	2.00	2.06

s_a obtained by some heuristic, we estimate the accuracy ratio $r(s_a) = f(s_a)/f(s^*)$ from *above* by the ratio $f(s_a)/HK(s^*)$, where $f(s_a)$ is the length of the heuristic tour s_a and $HK(s^*)$ is the Held-Karp lower bound on the shortest-tour length.

The results (see Table 12.1) from a large empirical study [Joh07a] indicate the average tour quality and running times for the discussed heuristics.[3] The instances in the reported sample have 10,000 cities generated randomly and uniformly as integral-coordinate points in the plane, with the Euclidean distances rounded to the nearest integer. The quality of tours generated by the heuristics remain about the same for much larger instances (up to a million cities) as long as they belong to the same type of instances. The running times quoted are for expert implementations run on a Compaq ES40 with 500 Mhz Alpha processors and 2 gigabytes of main memory or its equivalents.

Asymmetric instances of the traveling salesman problem—i.e., those with a nonsymmetic matrix of intercity distances—have proved to be significantly harder to solve, both exactly and approximately, than Euclidean instances. In particular, exact optimal solutions for many 316-city asymmetric instances remained unknown at the time of the state-of-the-art survey by Johnson et al. [Joh07b].

Approximation Algorithms for the Knapsack Problem

The knapsack problem, another well-known *NP*-hard problem, was also introduced in Section 3.4: given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of weight capacity W, find the most valuable subset of the items that fits into the knapsack. We saw how this problem can be solved by exhaustive search (Section 3.4), dynamic programming (Section 8.2), and branch-and-bound (Section 12.2). Now we will solve this problem by approximation algorithms.

Greedy Algorithms for the Knapsack Problem We can think of several greedy approaches to this problem. One is to select the items in decreasing order of their weights; however, heavier items may not be the most valuable in the set. Alternatively, if we pick up the items in decreasing order of their value, there is no guarantee that the knapsack's capacity will be used efficiently. Can we find a greedy strategy that takes into account both the weights and values? Yes, we can, by computing the value-to-weight ratios v_i/w_i, $i = 1, 2, \ldots, n$, and selecting the items in decreasing order of these ratios. (In fact, we already used this approach in designing the branch-and-bound algorithm for the problem in Section 12.2.) Here is the algorithm based on this greedy heuristic.

Greedy algorithm for the discrete knapsack problem

Step 1 Compute the value-to-weight ratios $r_i = v_i/w_i$, $i = 1, \ldots, n$, for the items given.

Step 2 Sort the items in nonincreasing order of the ratios computed in Step 1. (Ties can be broken arbitrarily.)

Step 3 Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack and proceed to the next item; otherwise, just proceed to the next item.

EXAMPLE 5 Let us consider the instance of the knapsack problem with the knapsack capacity 10 and the item information as follows:

item	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

Computing the value-to-weight ratios and sorting the items in nonincreasing order of these efficiency ratios yields

item	weight	value	value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The greedy algorithm will select the first item of weight 4, skip the next item of weight 7, select the next item of weight 5, and skip the last item of weight 3. The solution obtained happens to be optimal for this instance (see Section 12.2, where we solved the same instance by the branch-and-bound algorithm). ■

Does this greedy algorithm always yield an optimal solution? The answer, of course, is no: if it did, we would have a polynomial-time algorithm for the *NP*-hard problem. In fact, the following example shows that no finite upper bound on the accuracy of its approximate solutions can be given either.

EXAMPLE 6

item	weight	value	value/weight
1	1	2	2
2	W	W	1

The knapsack capacity is $W > 2$.

Since the items are already ordered as required, the algorithm takes the first item and skips the second one; the value of this subset is 2. The optimal selection consists of item 2 whose value is W. Hence, the accuracy ratio $r(s_a)$ of this approximate solution is $W/2$, which is unbounded above. ■

It is surprisingly easy to tweak this greedy algorithm to get an approximation algorithm with a finite performance ratio. All it takes is to choose the better of two alternatives: the one obtained by the greedy algorithm or the one consisting of a single item of the largest value that fits into the knapsack. (Note that for the instance of the preceding example, the second alternative is better than the first one.) It is not difficult to prove that the performance ratio of this ***enhanced greedy algorithm*** is 2. That is, the value of an optimal subset s^* will never be more than twice as large as the value of the subset s_a obtained by this enhanced greedy algorithm, and 2 is the smallest multiple for which such an assertion can be made.

It is instructive to consider the continuous version of the knapsack problem as well. In this version, we are permitted to take arbitrary fractions of the items given. For this version of the problem, it is natural to modify the greedy algorithm as follows.

Greedy algorithm for the continuous knapsack problem

Step 1 Compute the value-to-weight ratios v_i/w_i, $i = 1, \ldots, n$, for the items given.

Step 2 Sort the items in nonincreasing order of the ratios computed in Step 1. (Ties can be broken arbitrarily.)

Step 3 Repeat the following operation until the knapsack is filled to its full capacity or no item is left in the sorted list: if the current item on the list fits into the knapsack in its entirety, take it and proceed to the next item; otherwise, take its largest fraction to fill the knapsack to its full capacity and stop.

For example, for the four-item instance used in Example 5 to illustrate the greedy algorithm for the discrete version, the algorithm will take the first item of weight 4 and then 6/7 of the next item on the sorted list to fill the knapsack to its full capacity.

It should come as no surprise that this algorithm always yields an optimal solution to the continuous knapsack problem. Indeed, the items are ordered according to their efficiency in using the knapsack's capacity. If the first item on the sorted list has weight w_1 and value v_1, no solution can use w_1 units of capacity with a higher payoff than v_1. If we cannot fill the knapsack with the first item or its fraction, we should continue by taking as much as we can of the second-most efficient item, and so on. A formal rendering of this proof idea is somewhat involved, and we will leave it for the exercises.

Note also that the optimal value of the solution to an instance of the continuous knapsack problem can serve as an upper bound on the optimal value of the discrete version of the same instance. This observation provides a more sophisticated way of computing upper bounds for solving the discrete knapsack problem by the branch-and-bound method than the one used in Section 12.2.

Approximation Schemes We now return to the discrete version of the knapsack problem. For this problem, unlike the traveling salesman problem, there exist polynomial-time ***approximation schemes***, which are parametric families of algorithms that allow us to get approximations $s_a^{(k)}$ with any predefined accuracy level:

$$\frac{f(s^*)}{f(s_a^{(k)})} \le 1 + 1/k \quad \text{for any instance of size } n,$$

where k is an integer parameter in the range $0 \le k < n$. The first approximation scheme was suggested by S. Sahni in 1975 [Sah75]. This algorithm generates all subsets of k items or less, and for each one that fits into the knapsack it adds the remaining items as the greedy algorithm would do (i.e., in nonincreasing order of their value-to-weight ratios). The subset of the highest value obtained in this fashion is returned as the algorithm's output.

EXAMPLE 7 A small example of an approximation scheme with $k = 2$ is provided in Figure 12.16. The algorithm yields {1, 3, 4}, which is the optimal solution for this instance. ■

You can be excused for not being overly impressed by this example. And, indeed, the importance of this scheme is mostly theoretical rather than practical. It lies in the fact that, in addition to approximating the optimal solution with any predefined accuracy level, the time efficiency of this algorithm is polynomial in n. Indeed, the total number of subsets the algorithm generates before adding extra elements is

$$\sum_{j=0}^{k} \binom{n}{j} = \sum_{j=0}^{k} \frac{n(n-1)\cdots(n-j+1)}{j!} \le \sum_{j=0}^{k} n^j \le \sum_{j=0}^{k} n^k = (k+1)n^k.$$

item	weight	value	value/weight
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	1	\$ 4	4

capacity $W = 10$

(a)

subset	added items	value
∅	1, 3, 4	\$69
{1}	3, 4	\$69
{2}	4	\$46
{3}	1, 4	\$69
{4}	1, 3	\$69
{1, 2}	not feasible	
{1, 3}	4	\$69
{1, 4}	3	\$69
{2, 3}	not feasible	
{2, 4}		\$46
{3, 4}	1	\$69

(b)

FIGURE 12.16 Example of applying Sahni's approximation scheme for $k = 2$. (a) Instance. (b) Subsets generated by the algorithm.

For each of those subsets, it needs $O(n)$ time to determine the subset's possible extension. Thus, the algorithm's efficiency is in $O(kn^{k+1})$. Note that although it is polynomial in n, the time efficiency of Sahni's scheme is exponential in k. More sophisticated approximation schemes, called ***fully polynomial schemes***, do not have this shortcoming. Among several books that discuss such algorithms, the monographs [Mar90] and [Kel04] are especially recommended for their wealth of other material about the knapsack problem.