

state-space tree are generated. For backtracking, this tree is usually developed depth-first (i.e., similar to DFS). Branch-and-bound can generate nodes according to several rules: the most natural one is the so-called best-first rule explained in Section 12.2.

Section 12.3 takes a break from the idea of solving a problem exactly. The algorithms presented there solve problems approximately but fast. Specifically, we consider a few approximation algorithms for the traveling salesman and knapsack problems. For the traveling salesman problem, we discuss basic theoretical results and pertinent empirical data for several well-known approximation algorithms. For the knapsack problem, we first introduce a greedy algorithm and then a parametric family of polynomial-time algorithms that yield arbitrarily good approximations.

Section 12.4 is devoted to algorithms for solving nonlinear equations. After a brief discussion of this very important problem, we examine three classic methods for approximate root finding: the bisection method, the method of false position, and Newton's method.

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.

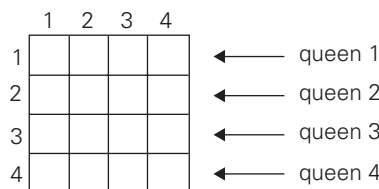


FIGURE 12.1 Board for the four-queens problem.

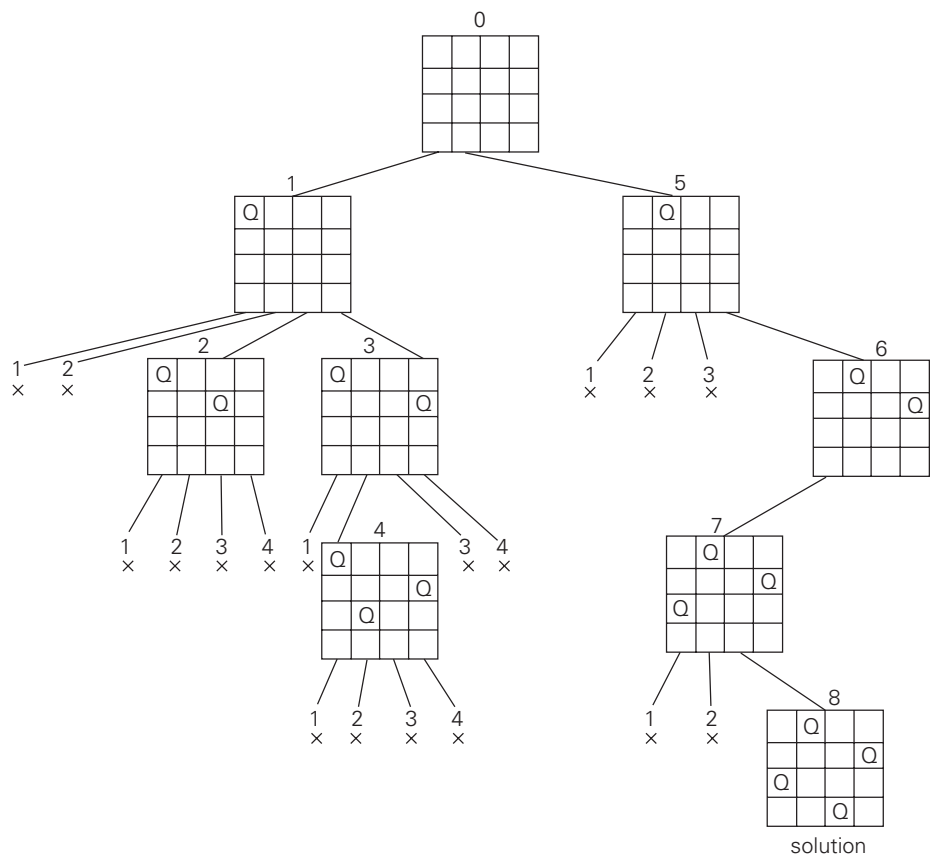


FIGURE 12.2 State-space tree of solving the four-queens problem by backtracking. \times denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

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

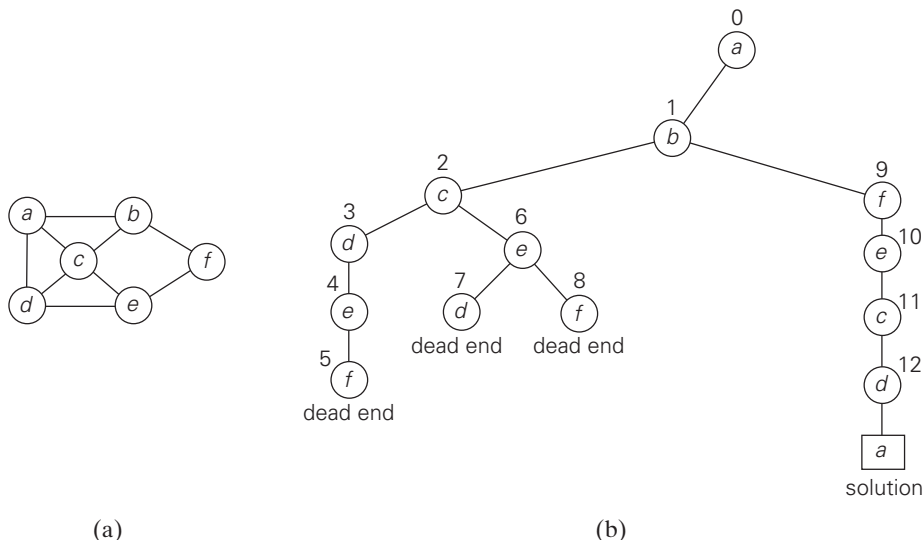


FIGURE 12.3 (a) Graph. (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order in which the nodes are generated.

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 $A = \{a_1, \dots, a_n\}$ 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

$$a_1 < a_2 < \dots < a_n.$$

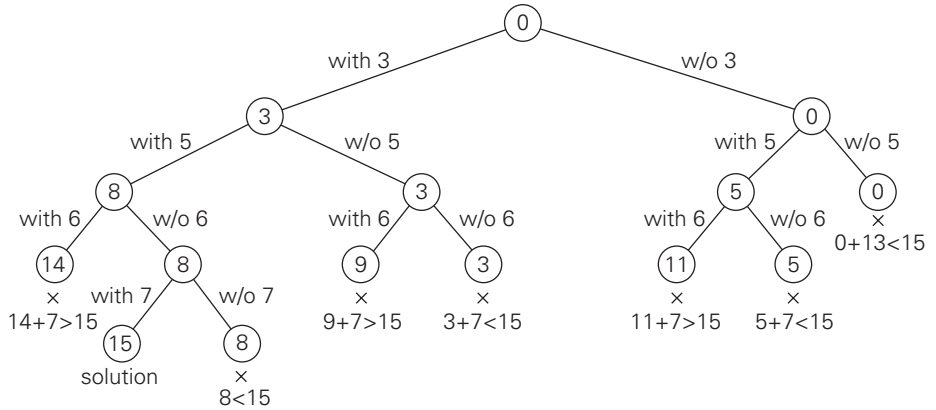


FIGURE 12.4 Complete state-space tree of the backtracking algorithm applied to the instance $A = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the 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 i th 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, \dots, x_n) where each coordinate x_i is an element of some finite lin-

early 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 $(1, i)$, $\lfloor n/2 \rfloor \leq i \leq n$, 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, \dots, 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

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

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.

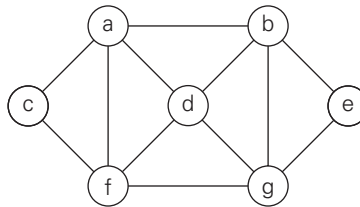
Exercises 12.1

1.
 - a. Continue the backtracking search for a solution to the four-queens problem, which was started in this section, to find the second solution to the problem.
 - b. Explain how the board's symmetry can be used to find the second solution to the four-queens problem.
2.
 - a. Which is the *last* solution to the five-queens problem found by the backtracking algorithm?
 - b. Use the board's symmetry to find at least four other solutions to the problem.

3. **a.** Implement the backtracking algorithm for the n -queens problem in the language of your choice. Run your program for a sample of n values to get the numbers of nodes in the algorithm's state-space trees. Compare these numbers with the numbers of candidate solutions generated by the exhaustive-search algorithm for this problem (see Problem 9 in Exercises 3.4).
- b.** For each value of n for which you run your program in part (a), estimate the size of the state-space tree by the method described in Section 12.1 and compare the estimate with the actual number of nodes you obtained.



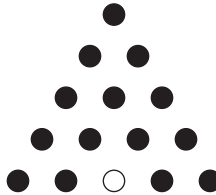
4. Design a linear-time algorithm that finds a solution to the n -queens problem for any $n \geq 4$.
5. Apply backtracking to the problem of finding a Hamiltonian circuit in the following graph.



6. Apply backtracking to solve the 3-coloring problem for the graph in Figure 12.3a.
7. Generate all permutations of $\{1, 2, 3, 4\}$ by backtracking.
8. **a.** Apply backtracking to solve the following instance of the subset sum problem: $A = \{1, 3, 4, 5\}$ and $d = 11$.
- b.** Will the backtracking algorithm work correctly if we use just one of the two inequalities to terminate a node as nonpromising?
9. The general template for backtracking algorithms, which is given in the section, works correctly only if no solution is a prefix to another solution to the problem. Change the template's pseudocode to work correctly without this restriction.
10. Write a program implementing a backtracking algorithm for
 - a.** the Hamiltonian circuit problem.
 - b.** the m -coloring problem.



11. *Puzzle pegs* This puzzle-like game is played on a board with 15 small holes arranged in an equilateral triangle. In an initial position, all but one of the holes are occupied by pegs, as in the example shown below. A legal move is a jump of a peg over its immediate neighbor into an empty square opposite; the jump removes the jumped-over neighbor from the board.



Design and implement a backtracking algorithm for solving the following versions of this puzzle.

- a. Starting with a given location of the empty hole, find a shortest sequence of moves that eliminates 14 pegs with no limitations on the final position of the remaining peg.
- b. Starting with a given location of the empty hole, find a shortest sequence of moves that eliminates 14 pegs with the remaining peg at the empty hole of the initial board.

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 function¹ 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

1. This bound should be a lower bound for a minimization problem and an upper bound for a maximization 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:

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

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 $\{a \rightarrow 2, b \rightarrow 1, c \rightarrow 3, d \rightarrow 4\}$ with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution $\{a \rightarrow 2, b \rightarrow 1, c \rightarrow 4, d \rightarrow 3\}$ 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

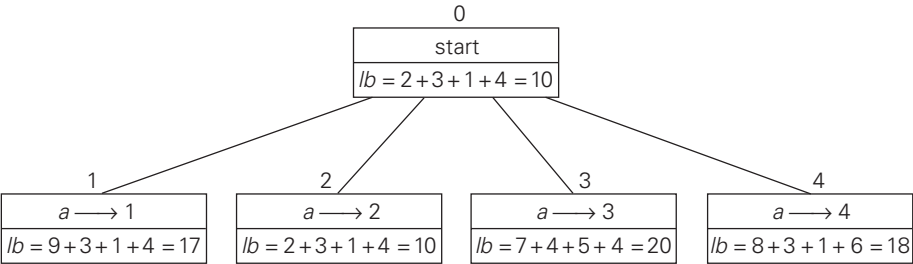


FIGURE 12.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node’s fields indicate the job number assigned to person *a* and the lower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

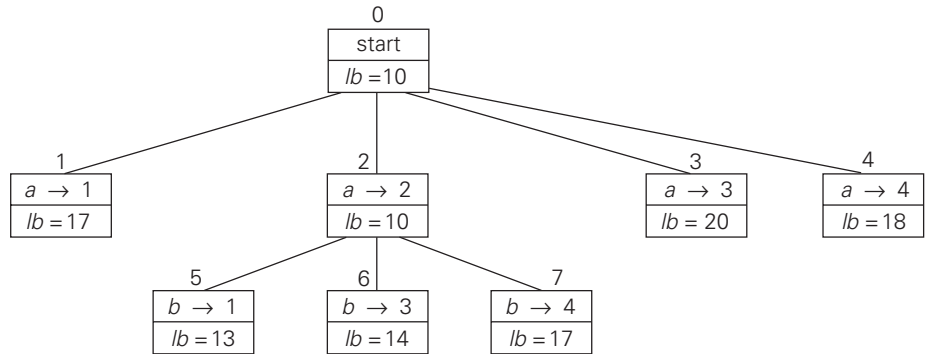


FIGURE 12.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

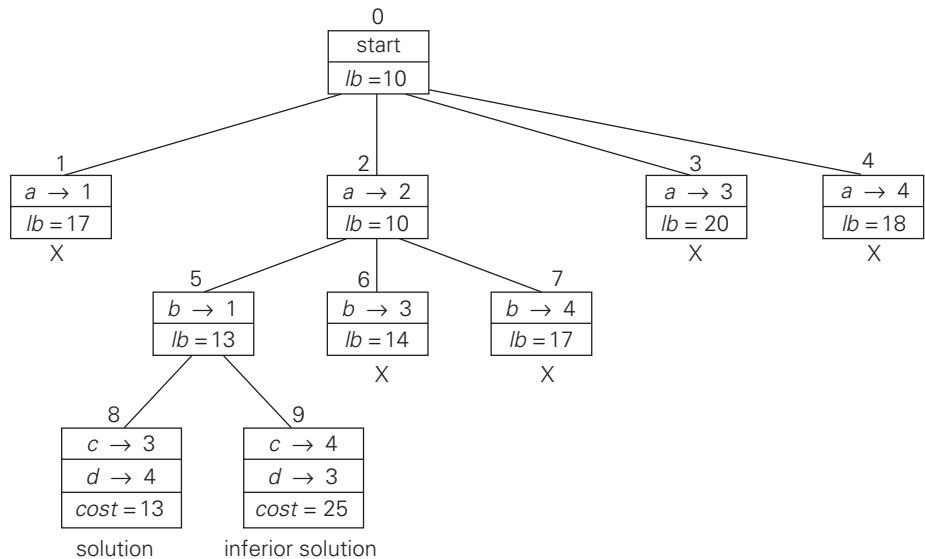


FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

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 $v_i, i = 1, 2, \dots, n$, 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:

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

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 i th 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} :

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

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	$\frac{\text{value}}{\text{weight}}$	The knapsack's capacity W is 10.
1	4	\$40	10	
2	7	\$42	6	
3	5	\$25	5	
4	3	\$12	4	

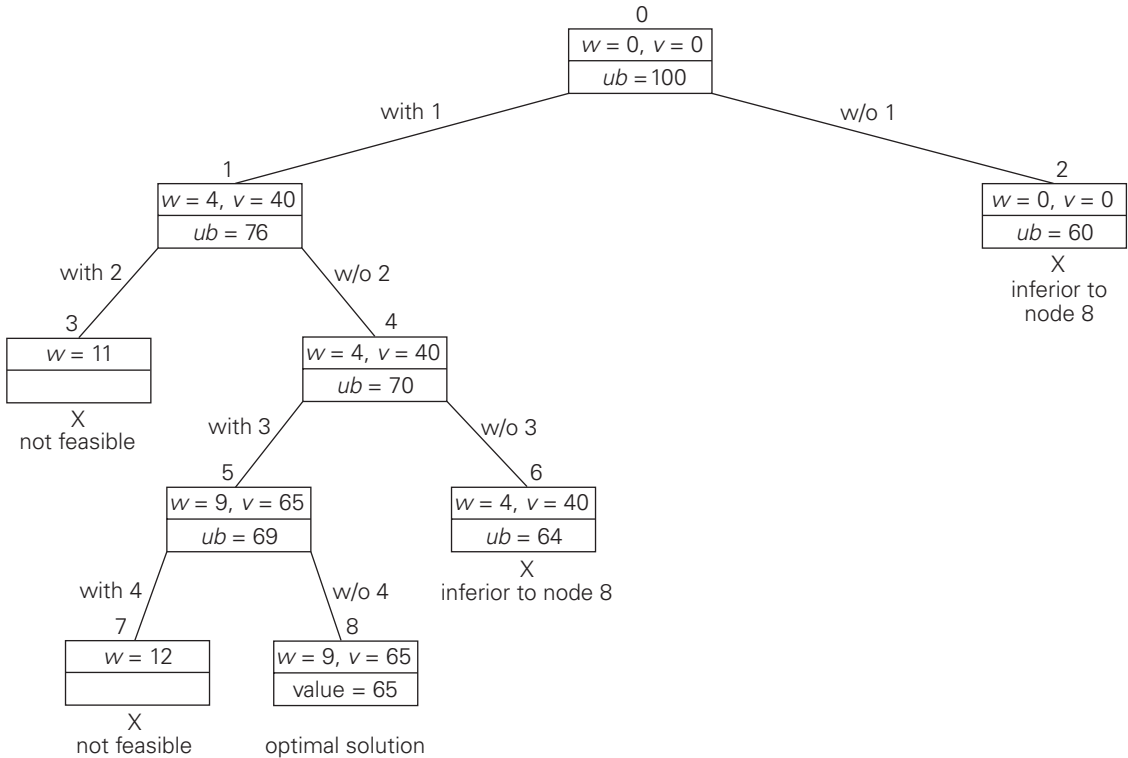


FIGURE 12.8 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

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 , $1 \leq i \leq n$, 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:

$$lb = \lceil s/2 \rceil. \quad (12.2)$$

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

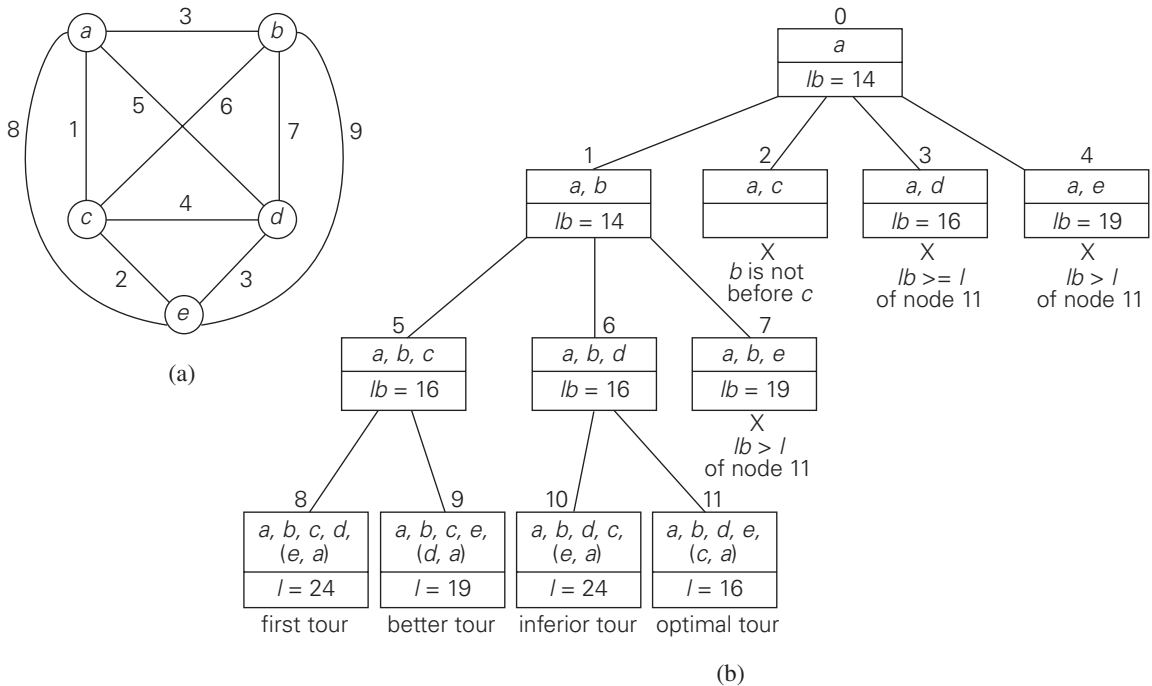


FIGURE 12.9 (a) Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

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.)

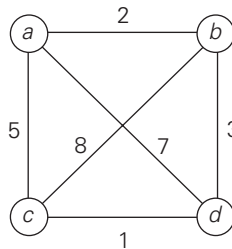
Finding a good bounding function is usually not a simple task. On the one hand, we want this function to be easy to compute. On the other hand, it cannot be too simplistic—otherwise, it would fail in its principal task to prune as many branches of a state-space tree as soon as possible. Striking a proper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

Exercises 12.2

1. What data structure would you use to keep track of live nodes in a best-first branch-and-bound algorithm?
2. Solve the same instance of the assignment problem as the one solved in the section by the best-first branch-and-bound algorithm with the bounding function based on matrix columns rather than rows.
3.
 - a. Give an example of the best-case input for the branch-and-bound algorithm for the assignment problem.
 - b. In the best case, how many nodes will be in the state-space tree of the branch-and-bound algorithm for the assignment problem?
4. Write a program for solving the assignment problem by the branch-and-bound algorithm. Experiment with your program to determine the average size of the cost matrices for which the problem is solved in a given amount of time, say, 1 minute on your computer.
5. Solve the following instance of the knapsack problem by the branch-and-bound algorithm:

item	weight	value	$W = 16$
1	10	\$100	
2	7	\$63	
3	8	\$56	
4	4	\$12	

6. a. Suggest a more sophisticated bounding function for solving the knapsack problem than the one used in the section.
b. Use your bounding function in the branch-and-bound algorithm applied to the instance of Problem 5.
7. Write a program to solve the knapsack problem with the branch-and-bound algorithm.
8. a. Prove the validity of the lower bound given by formula (12.2) for instances of the traveling salesman problem with symmetric matrices of integer intercity distances.
b. How would you modify lower bound (12.2) for nonsymmetric distance matrices?
9. Apply the branch-and-bound algorithm to solve the traveling salesman problem for the following graph:



(We solved this problem by exhaustive search in Section 3.4.)

10. As a research project, write a report on how state-space trees are used for programming such games as chess, checkers, and tic-tac-toe. The two principal algorithms you should read about are the minimax algorithm and alpha-beta pruning.

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?

2. The notion of an *NP*-hard problem can be defined more formally by extending the notion of polynomial reducibility to problems that are not necessarily in class *NP*, including optimization problems of the type discussed in this section (see [Gar79, Chapter 5]).