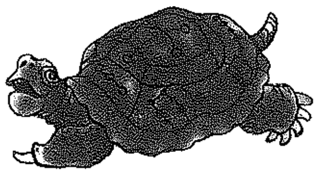
Introduction to The Design & Analysis of Algorithms

2ND EDITION



Anany Levitin

Villanova University

PEARSON
Addison Wesley

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

12

Coping with the Limitations of Algorithm Power

Keep on the lookout for novel ideas that others have used successfully. Your idea has to be original only in its adaptation to the problem you're working on.

—Thomas Edison (1847–1931)

As we saw in the previous chapter, there are problems that are difficult to solve algorithmically. At the same time, some of them are so important that we cannot just sigh in resignation and do nothing. This chapter outlines several ways of dealing with such difficult problems.

Sections 12.1 and 12.2 introduce two algorithm design techniques—***back-tracking*** and ***branch-and-bound***—that often make it possible to solve at least some large instances of difficult combinatorial problems. Both strategies can be considered an improvement over exhaustive search, discussed in Section 3.4. Unlike exhaustive search, they construct candidate solutions one component at a time and evaluate the partially constructed solutions: if no potential values of the remaining components can lead to a solution, the remaining components are not generated at all. This approach makes it possible to solve some large instances of difficult combinatorial problems, though, in the worst case, we still face the same curse of exponential explosion encountered in exhaustive search.

Both backtracking and branch-and-bound are based on the construction of a ***state-space tree*** whose nodes reflect specific choices made for a solution's components. Both techniques terminate a node as soon as it can be guaranteed that no solution to the problem can be obtained by considering choices that correspond to the node's descendants. The techniques differ in the nature of problems they can be applied to. Branch-and-bound is applicable only to optimization problems because it is based on computing a bound on possible values of the problem's objective function. Backtracking is not constrained by this demand, but more often than not, it applies to nonoptimization problems. The other distinction between backtracking and branch-and-bound lies in the order in which nodes of the 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-by-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.

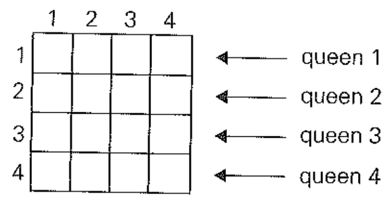


FIGURE 12.1 Board for the four-queens problem

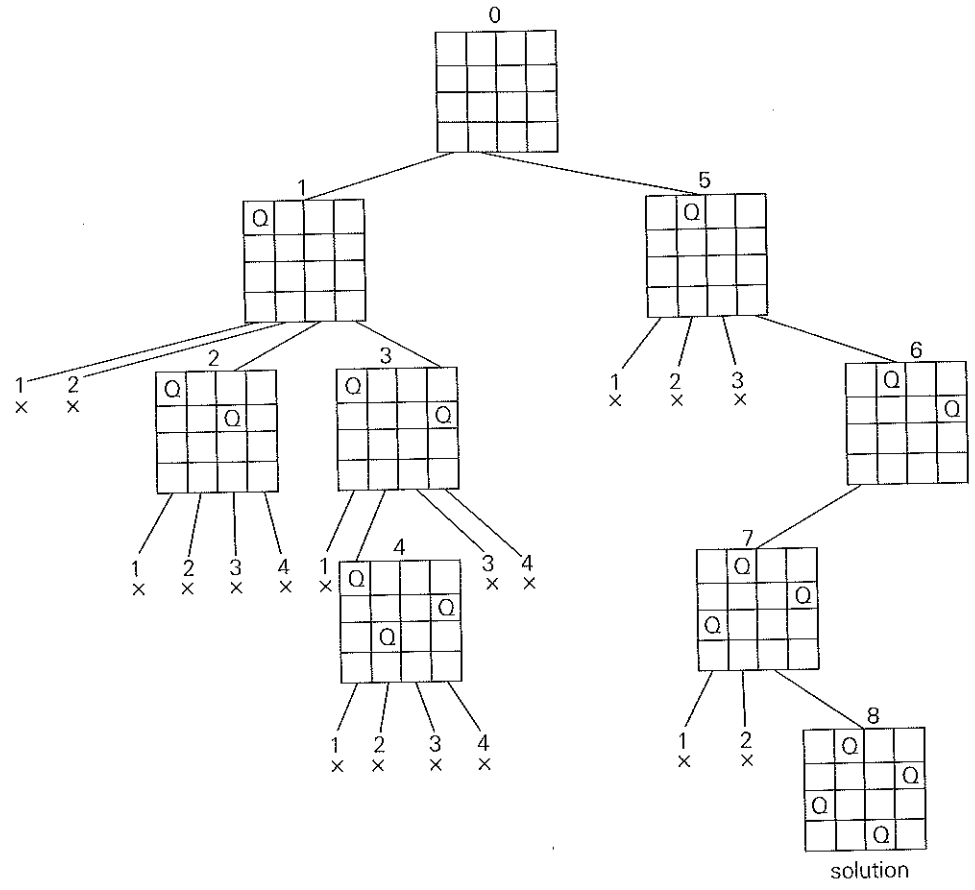


FIGURE 12.2 State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

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.

Hamiltonian Circuit Problem

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

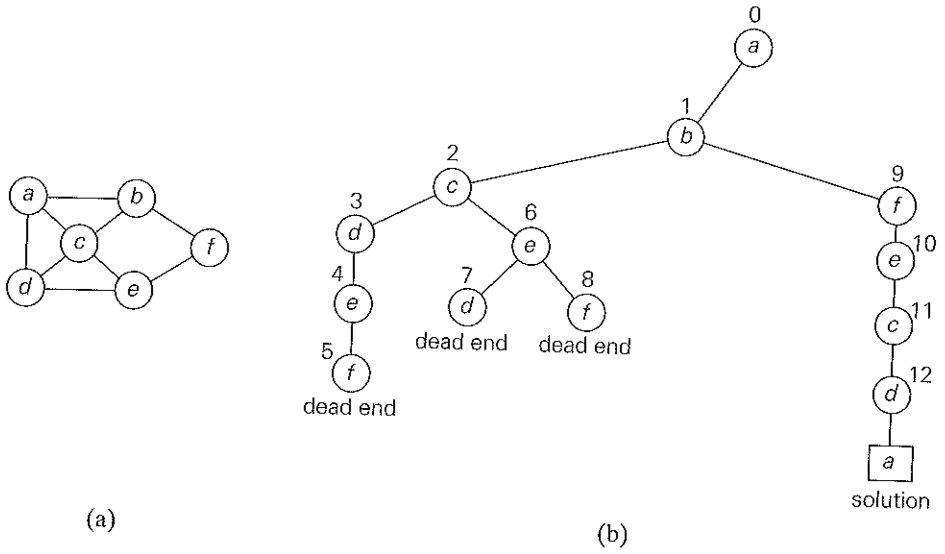


FIGURE 12.3 (a) Graph. (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order in which the nodes are generated.

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 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 $S = \{s_1, \ldots, s_n\}$ of n positive integers whose sum is equal to a given positive integer d. For example, for $S = \{1, 2, 5, 6, 8\}$ 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

$$s_1 \le s_2 \le \ldots \le s_n.$$

The state-space tree can be constructed as a binary tree like that in Figure 12.4 for the instance $S = \{3, 5, 6, 7\}$ 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 s_1 in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of s_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:

$$s' + s_{i+1} > d \text{ (the sum } s' \text{ is too large)}$$

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

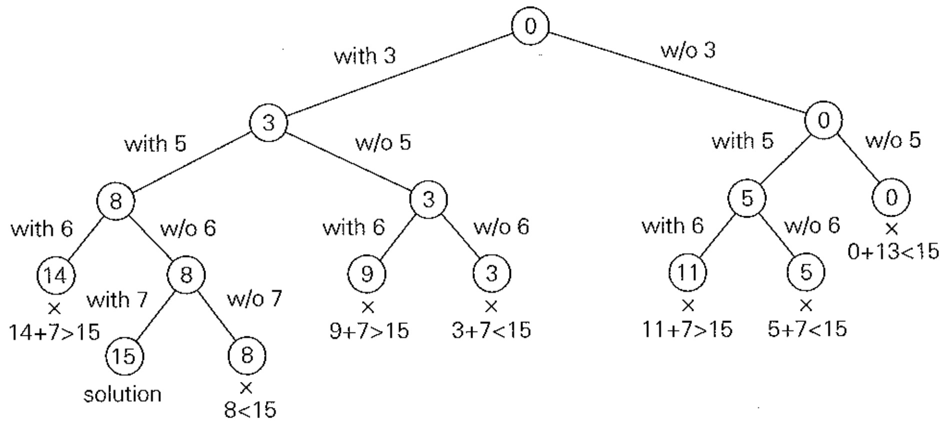


FIGURE 12.4 Complete state-space tree of the backtracking algorithm applied to the instance $S = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in subsets represented by the node. The inequality below a leaf indicates the reason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

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) or 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 $(x_1, x_2, \ldots, x_i)$ is not a solution, the algorithm finds the next element in S_{i+1} that is consistent with the values of $(x_1, x_2, \ldots, x_i)$ 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 8 in the 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 has c_i children on average, we estimate the number of nodes in the tree as

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

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.

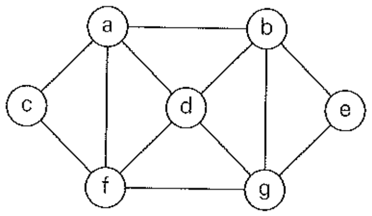
2. **a.** Which is the *last* solution to the five-queens problem found by the backtracking algorithm?

   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.

   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. Apply backtracking to the problem of finding a Hamiltonian circuit in the following graph.



5. Apply backtracking to solve the 3-coloring problem for the graph in Figure 12.3a.

6. Generate all permutations of {1, 2, 3, 4} by backtracking.

7. **a.** Apply backtracking to solve the following instance of the subset-sum problem: $S = \{1, 3, 4, 5\}$ 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?

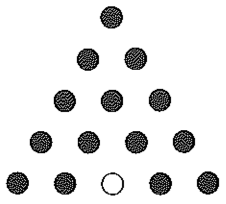
8. The general template for backtracking algorithms, which was given in Section 11.1, works correctly only if no solution is a prefix to another solution to the problem. Change the pseudocode to work correctly for such problems as well.

9. Write a program implementing a backtracking algorithm for

   a. the Hamiltonian circuit problem.

   b. the m-coloring problem.

10. *Puzzle pegs* This puzzle-like game is played on a triangular 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 one 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, one that seeks to minimize or maximize an objective function, usually subject to some constraints (a tour's length, the value of items selected, the cost of an assignment, and the like). 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), while 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

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

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 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") because 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-by-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 we investigated in Section 3.4:

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

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's 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***.

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

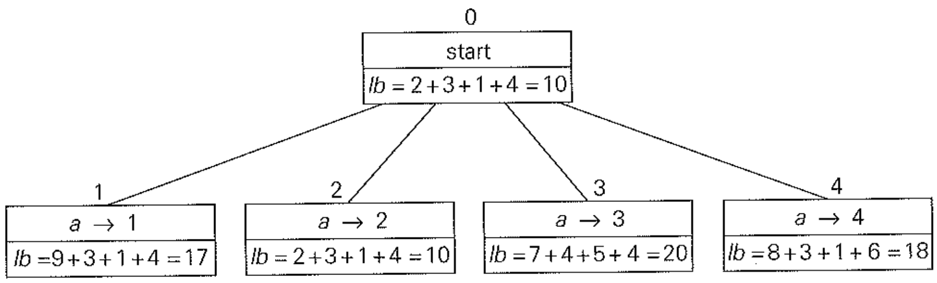


FIGURE 12.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower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

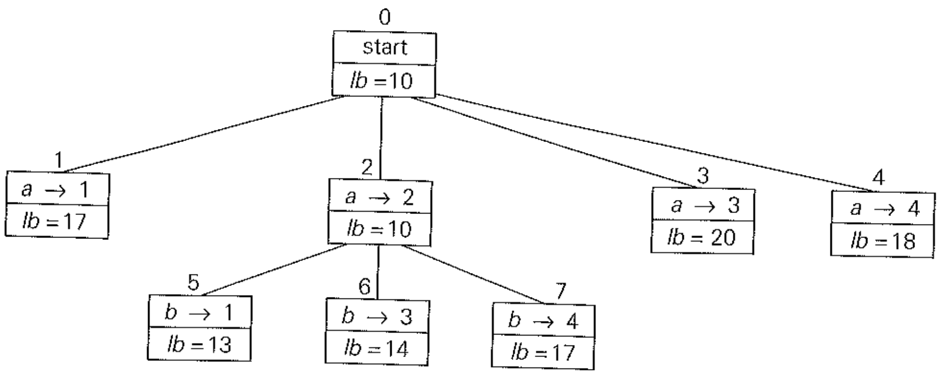


FIGURE 12.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

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

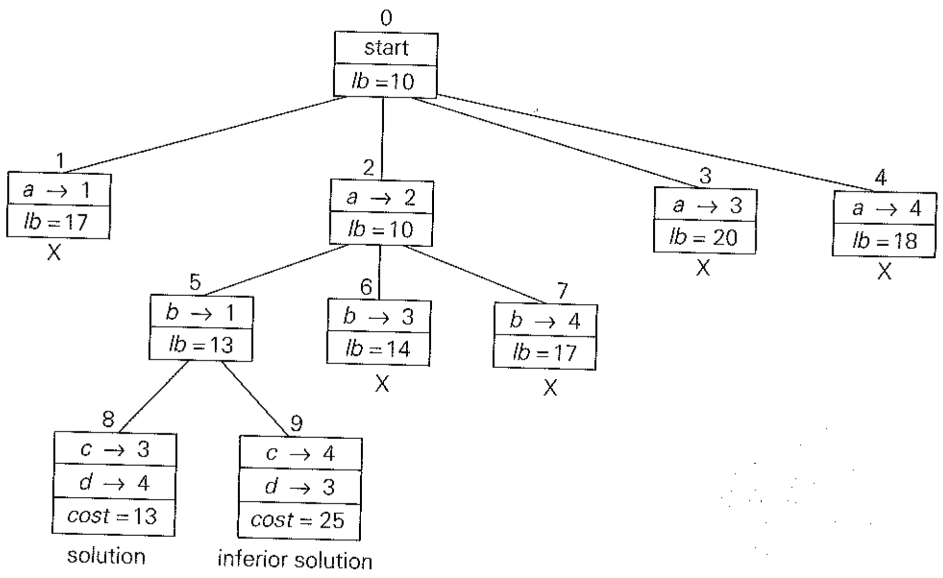


FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

of the solution represented by leaf 8, node 9 is simply terminated. (Note that if 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:

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

It is natural to structure the state-space tree for this problem as a binary tree constructed as follows (Figure 12.8). 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, while 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.

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

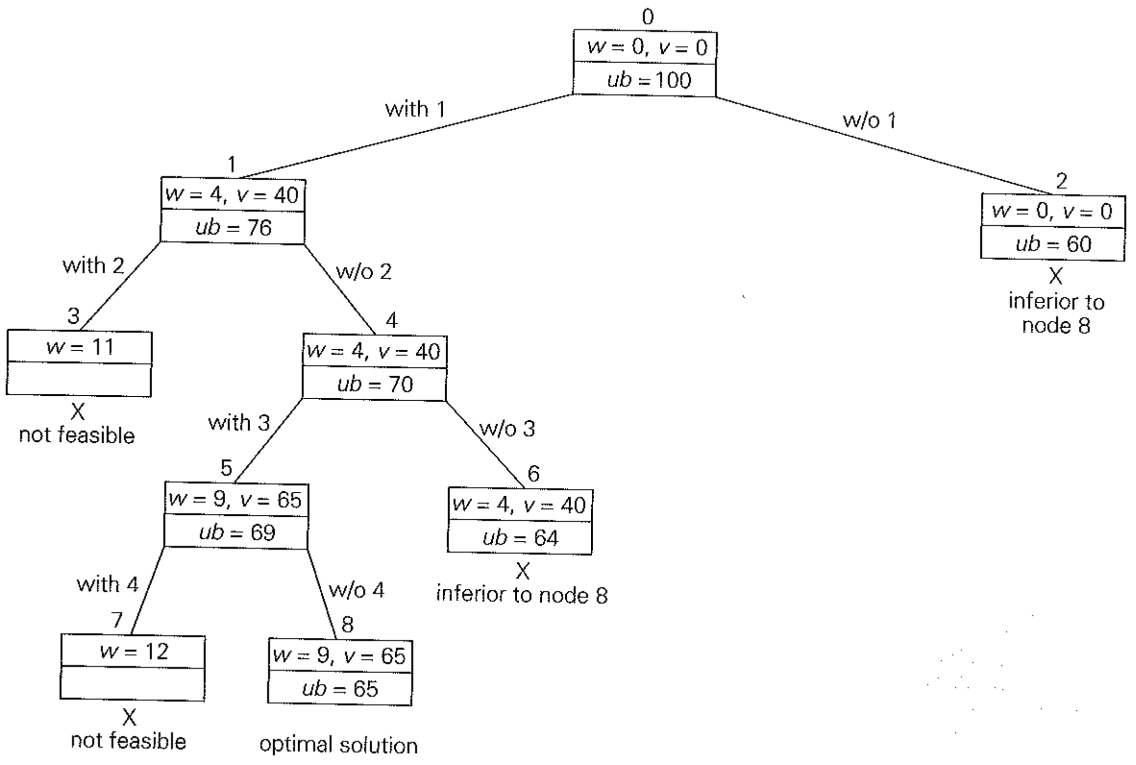


FIGURE 12.8 State-space tree of the branch-and-bound algorithm for the instance of the knapsack problem

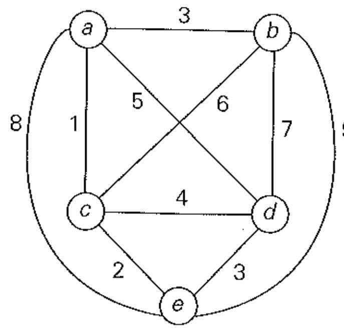
and \$40, respectively; the value of the upper bound is $40 + (10 - 4) * 6 = \$76$. Node 2 represents the subsets that do not include item 1. Accordingly, $w = 0$, $v = \$0$, and $ub = 0 + (10 - 0) * 6 = \$60$. Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising 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}. (As there are no additional items to consider, the upper bound for node 7 is simply equal to the total value of these two items.) 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.

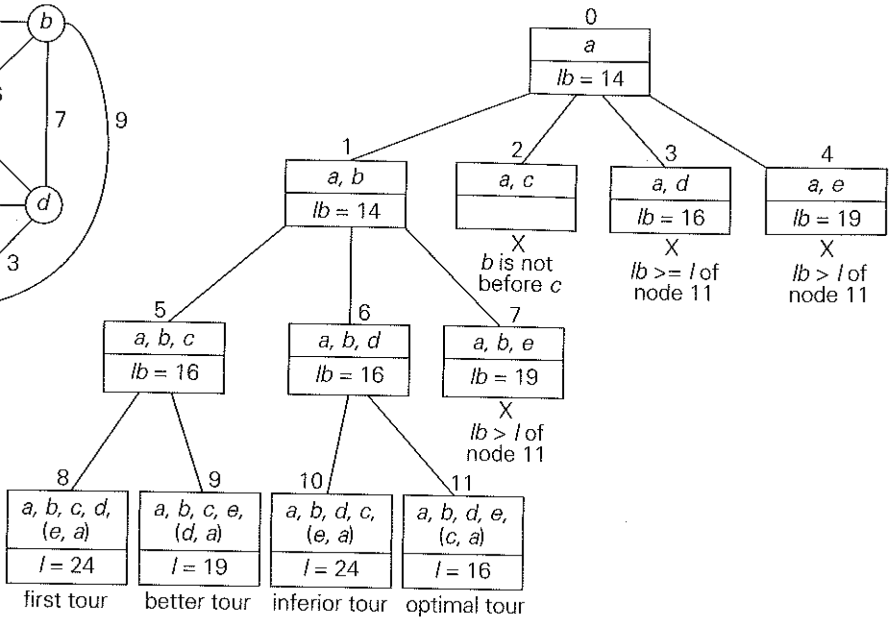
Traveling Salesman Problem

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, which does not require a lot of work to compute. It is not difficult to show (Problem 8 in Exercises 12.2) 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:

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



(a)



(b)

FIGURE 12.9 (a) Weighted graph. (b) State-space tree of the the branch-and-bound algorithm to find the 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.

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 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 12.9a. 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 an 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. (The branch of computer science called artificial intelligence (AI) is 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 under one minute on your computer.

5. Solve the following instance of the knapsack problem by the branch-and-bound algorithm.

item	weight	value	
1	10	\$100	
2	7	\$63	$W = 16$
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.

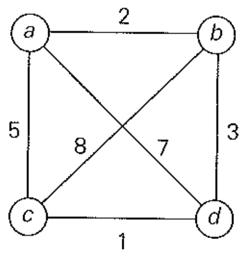
b. Use your bounding function in the branch-and-bound algorithm applied to the instance of Problem 5.

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. The optimization versions of such difficult combinatorial problems 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 as demonstrated in Section 8.4. 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 get solutions to many large instances of difficult problems of combinatorial optimization 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

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).

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 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 often 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 definition.

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 quesiton:

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

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 approximations 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 problems of combinatorial optimization 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 approximate solutions with a reasonable level of accuracy is much easier for some of these problems than for the 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], [Joh97], and [Gut02].)

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

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

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

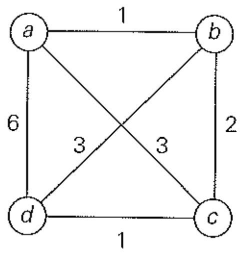


FIGURE 12.10 Instance of the traveling salesman problem for illustrating the nearest-neighbor algorithm.

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 $P = NP$. ▨

Nearest-neighbor algorithm

The following simple greedy algorithm is based on the ***nearest-neighbor*** heuristic: the idea of always going to the nearest unvisited city next.

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.

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 $4 + w$, while 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).

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].

Multifragment-heuristic algorithm

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 until a tour of length n is obtained, 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 mutifragment-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-heuristics algorithms remain unbounded on Euclidean instances, their accuracy ratios satisfy the following inequality for any such instance with $n \geq 2$ cities:

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

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) (Fig. 12.11b). A twice-around-the-tree walk that starts and ends at a is

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

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

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

of length 39. ▨

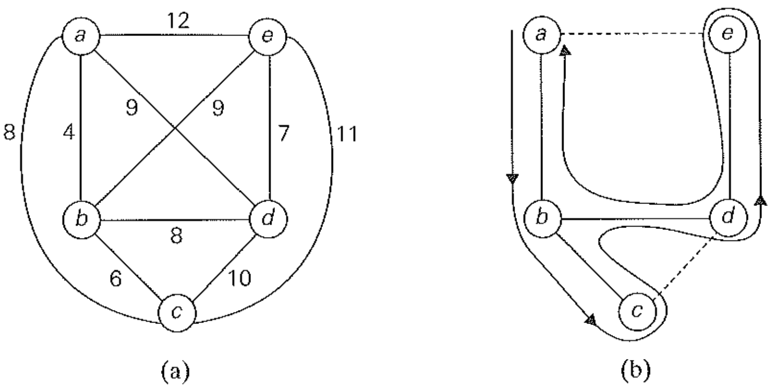


FIGURE 12.11 Illustration of the twice-around-the-tree algorithm. (a) Graph. (b) Walk around the minimum spanning tree with the shortcuts.

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^*; that is,

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

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

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

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; that is,

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

Combining the last two inequalities, we get the inequality

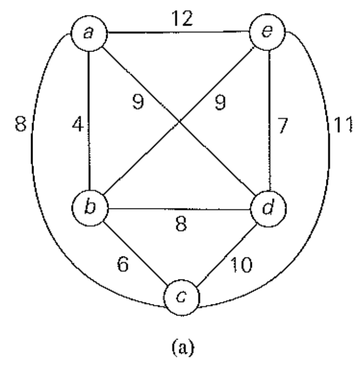
$$2f(s^*) > f(s_a),$$

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 enhancement would not have improved the tour $a - b - c - e - d - a$ obtained in



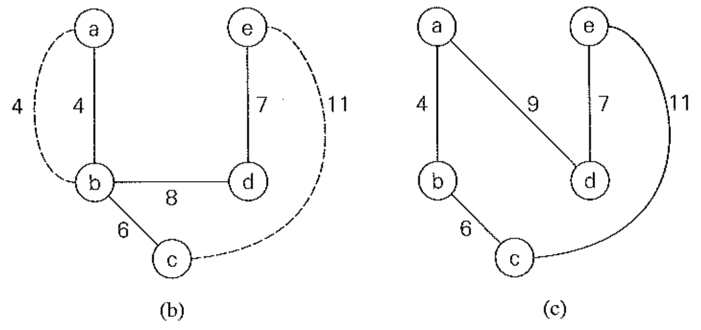


FIGURE 12.12 Application of the Christofides algorithm. (a) Graph. (b) Minimum spanning tree with added edges (in dash) of a minimum-weight matching of all odd-degree vertices. (c) Hamiltonian circuit obtained.

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 [Joh02].

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 of 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 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.

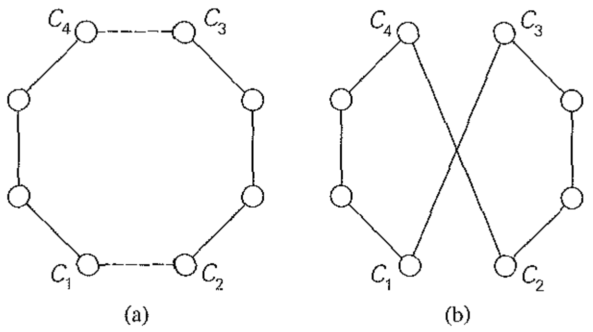


FIGURE 12.13 2-change: (a) Original tour. (b) New tour.

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 in 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 surveys by Johnson and McGeoch ([Joh97], [Joh02]) contain an outline of the algorithm and its modern extensions, as well as methods for its efficient implementation. These surveys 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 fifty years. This interest was driven by a combination of pure theoretical interest and serious practical needs stemming from such newer

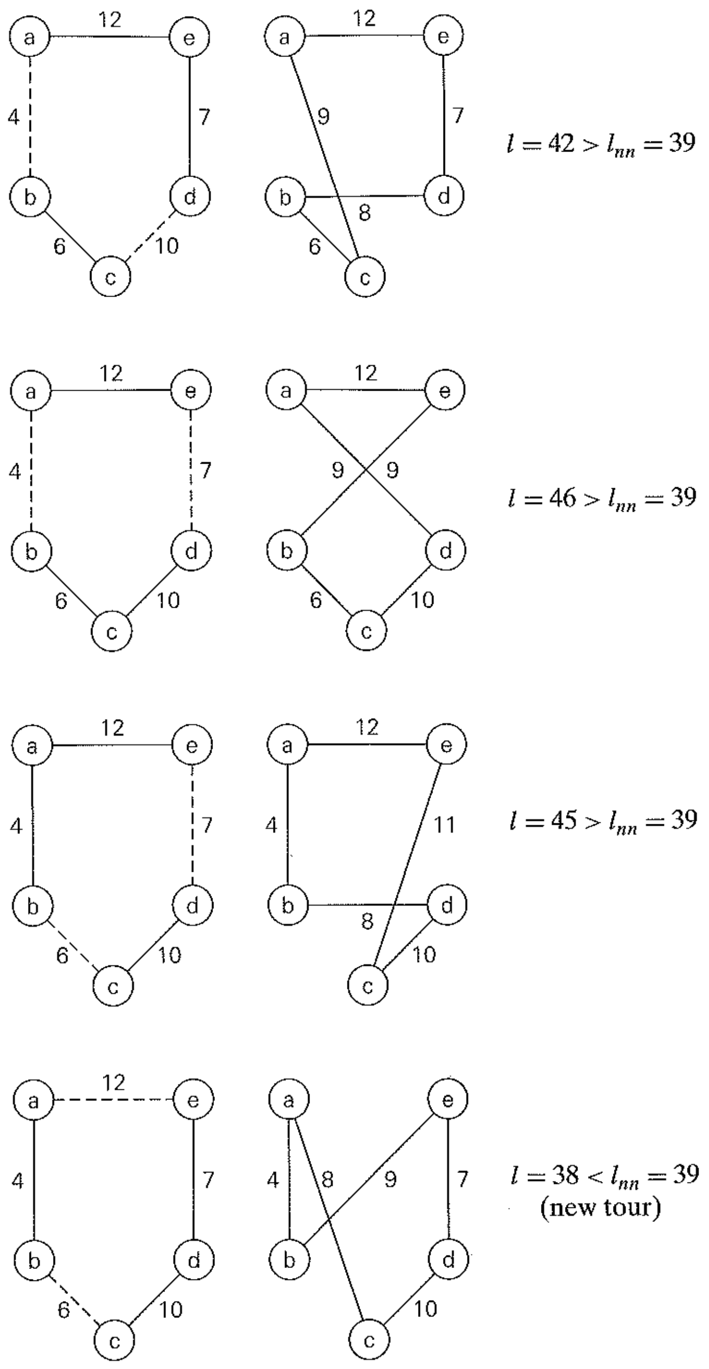


FIGURE 12.14 2-changes from the nearest-neighbor tour of the graph in Figure 12.11

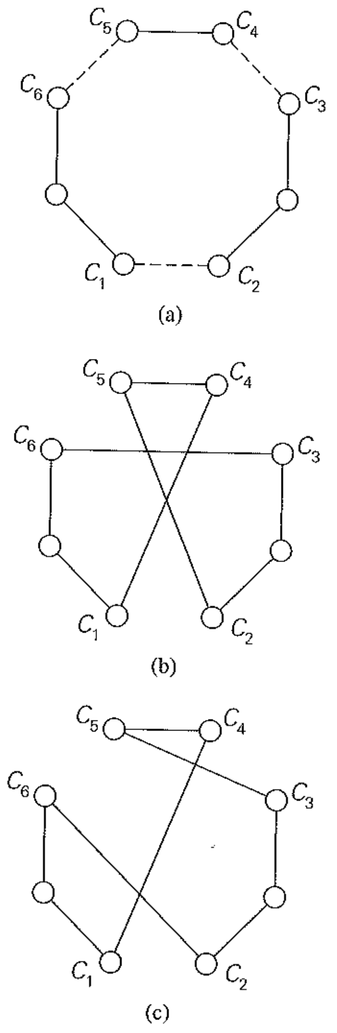


FIGURE 12.15 3-change: (a) Original tour. (b), (c) New tours.

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.

It used to be the case that instances with a few hundred cities were considered too large to be solved exactly. Nowadays, instances with up to 1,000 cities can be solved exactly in a quite reasonable amount of time—typically, in minutes or less 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 May 2004 was the shortest tour through all 24,978 cities in Sweden. There should be little doubt that this record will eventually be 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 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 [Joh02] 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.

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 sub-

3. We did not include the results for the twice-around-the-tree heuristic because of the inferior quality of its approximations with the average excess of about 40%. Nor did we quote the results for the most sophisticated local search heuristics with the average excess over optimum of less than a fraction of 1%.

TABLE 12.1 Average tour quality and running times for various heuristics on the 10,000-city random uniform Euclidean instances [Joh02]

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

set 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 8.4), 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, $i = 1, 2, \dots, n$, 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 $r_i = v_i/w_i$, $i = 1, \dots, n$, 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; otherwise, proceed to the next item.

EXAMPLE 5 Let us consider the instance of the knapsack problem with the knapsack's capacity equal to 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	$\frac{\text{value}}{\text{weight}}$
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	$\frac{\text{value}}{\text{weight}}$
1	1	2	2
2	W	W	1

The knapsack's 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 is 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 (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.

item	weight	value	$\frac{\text{value}}{\text{weight}}$
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.

Indeed, the total number of subsets the algorithm generates before adding extra elements is

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

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 while being 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] amd [Kel04] are especially recommended for their wealth of other material about the knapsack problem.

Exercises 12.3

1. **a.** Apply the nearest-neighbor algorithm to the instance defined by the distance matrix below. Start the algorithm at the first city, assuming that the cities are numbered from 1 to 5.

$$\begin{bmatrix} 0 & 14 & 4 & 10 & \infty \\ 14 & 0 & 5 & 8 & 7 \\ 4 & 5 & 0 & 9 & 16 \\ 10 & 8 & 9 & 0 & 32 \\ \infty & 7 & 16 & 32 & 0 \end{bmatrix}$$

   b. Compute the accuracy ratio of this approximate solution.

2. **a.** Write a pseudocode for the nearest-neighbor algorithm. Assume that its input is given by an n-by-n distance matrix.

   b. What is the time efficiency of the nearest-neighbor algorithm?

3. Apply the twice-around-the-tree algorithm to the graph in Figure 12.11a with a walk around the minimum spanning tree that starts at the same vertex a but differs from the walk in Figure 12.11b. Is the length of the obtained tour the same as the length of the tour in Figure 12.11b?

4. Prove that making a shortcut of the kind used by the twice-around-the-tree algorithm cannot increase the tour's length in an Euclidean graph.

5. What is the time efficiency class of the greedy algorithm for the knapsack problem?

6. Prove that the performance ratio R_A of the enhanced greedy algorithm for the knapsack problem is equal to 2.

7. Consider the greedy algorithm for the bin-packing problem, which is called the ***first-fit*** (*FF*) ***algorithm***: place each of the items in the order given into the

first bin the item fits in; when there are no such bins, place the item in a new bin and add this bin to the end of the bin list.

a. Apply *FF* to the instance

$$s_1 = 0.4,\ \ s_2 = 0.7,\ \ s_3 = 0.2,\ \ s_4 = 0.1,\ \ s_5 = 0.5$$

and determine whether the solution obtained is optimal.

b. Determine the worst-case time efficiency of *FF*.

c. Prove that *FF* is a 2-approximation algorithm.

8. The ***first-fit decreasing*** (*FFD*) approximation algorithm for the bin-packing problem starts by sorting the items in nonincreasing order of their sizes and then acts as the first-fit algorithm.

a. Apply *FFD* to the instance

$$s_1 = 0.4,\ \ s_2 = 0.7,\ \ s_3 = 0.2,\ \ s_4 = 0.1,\ \ s_5 = 0.5$$

and determine whether the solution obtained is optimal.

b. Does *FFD* always yield an optimal solution? Justify your answer.

c. Prove that *FFD* is a 1.5-approximation algorithm.

d. Run an experiment to determine which of the two algorithms—*FF* or *FFD*—yields more accurate approximations on a random sample of the problem's instances.

9. a. Design a simple 2-approximation algorithm for finding a ***minimum vertex cover*** (a vertex cover with the smallest number of vertices) in a given graph.

b. Consider the following approximation algorithm for finding a ***maximum independent set*** (an independent set with the largest number of vertices) in a given graph. Apply the 2-approximation algorithm of part a and output all the vertices that are not in the obtained vertex cover. Can we claim that this algorithm is a 2-approximation algorithm, too?

10. a. Design a polynomial-time greedy algorithm for the graph-coloring problem.

b. Show that the performance ratio of your approximation algorithm is infinitely large.

12.4 Algorithms for Solving Nonlinear Equations

In this section, we discuss several algorithms for solving nonlinear equations in one unknown,

$$f(x) = 0. \tag{12.4}$$

There are several reasons for this choice among subareas of numerical analysis. First of all, this is an extremely important problem from the practical and theoretical points of view. It arises as a mathematical model of numerous phenomena in the sciences and engineering, both directly and indirectly. (Recall, for example, that the standard calculus technique for finding extremum points of a function $f(x)$ is based on finding its critical points, which are the roots of the equation $f'(x) = 0$.) Second, it represents the most accessible topic in numerical analysis and, at the same time, exhibits its typical tools and concerns. Third, some methods for solving equations closely parallel algorithms for array searching and hence provide examples of applying general algorithm design techniques to problems of continuous mathematics.

Let us start with dispelling a misconception you might have about solving equations. Your experience with equation solving from middle school to calculus courses might have led you to believe that we can solve equations by "factoring" or by applying a readily available formula. Sorry to break it to you, but you have been deceived (with the best of educational intentions, of course): you were able to solve all those equations only because they had been carefully selected to make it possible. In general, we cannot solve equations exactly and need approximation algorithms to do so.

This is true even for solving the quadratic equation

$$ax^2 + bx + c = 0$$

because the standard formula for its roots

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

requires computing the square root, which can be done only approximately for most positive numbers. In addition, as we discussed in Section 11.4, this canonical formula needs to be modified to avoid the possibility of low-accuracy answers.

What about formulas for roots of polynomials of degrees higher than two? Such formulas for third- and fourth-degree polynomials exist, but they are too cumbersome to be of practical value. For polynomials of degrees higher than four, there can be no general formula for their roots that would involve only the polynomial's coefficients, arithmetical operations, and radicals (taking roots). This remarkable result was published first by the Italian mathematician and physician Paolo Ruffini (1765–1822) in 1799 and rediscovered a quarter century later by the Norwegian mathematician Niels Abel (1802–1829); it was developed further by the French mathematician Evariste Galois (1811–1832).[4]

4. Ruffini's discovery was completely ignored by almost all prominent mathematicians of that time. Abel died young after a difficult life of poverty. Galois was killed in a duel when he was only 21 years old. Their results on the solution of higher-degree equations are now considered to be among the crowning achievements in the history of mathematics.

The impossibility of such a formula can hardly be considered a great disappointment. As the great German mathematician Carl Friedrich Gauss (1777–1855) put it in his thesis of 1801, the algebraic solution of an equation was no better than devising a symbol for the root of the equation and then saying that the equation had a root equal to the symbol [OCo98].

We can interpret solutions to equation (12.4) as points at which the graph of the function $f(x)$ intersects with the x-axis. The three algorithms we discuss in this section take advantage of this interpretation. Of course, the graph of $f(x)$ may intersect the x-axis at a single point (e.g., $x^3 = 0$), at multiple or even infinitely many points ($\sin x = 0$), or at no point ($e^x + 1 = 0$). Equation (12.3) would then have a single root, several roots, and no roots, respectively. It is a good idea to sketch a graph of the function before starting to approximate its roots. It can help to determine the number of roots and their approximate locations. In general, it is a good idea to isolate roots, i.e., to identify intervals containing a single root of the equation in question.

Bisection Method

This algorithm is based on an observation that the graph of a continuous function must intersect with the x-axis between two points a and b at least once if the function's values have opposite signs at these two points (Figure 12.17).

The validity of this observation is proved as a theorem in calculus courses, and we take it for granted here. It serves as the basis of the following algorithm, called the ***bisection method,*** for solving equation (12.4). Starting with an interval $[a, b]$ at whose endpoints $f(x)$ has opposite signs, the algorithm computes the value of $f(x)$ at the middle point $x_{mid} = (a + b)/2$. If $f(x_{mid}) = 0$, a root was found and the algorithm stops. Otherwise, it continues the search for a root either on $[a, x_{mid}]$ or on $[x_{mid}, b]$, depending on which of the two halves the values of $f(x)$ have opposite signs at the endpoints of the new interval.

Since we cannot expect the bisection algorithm to stumble on the exact value of the equation's root and stop, we need a different criterion for stopping the algorithm. We can stop the algorithm after the interval $[a_n, b_n]$ bracketing some root x^* becomes so small that we can guarantee that the absolute error of approximating

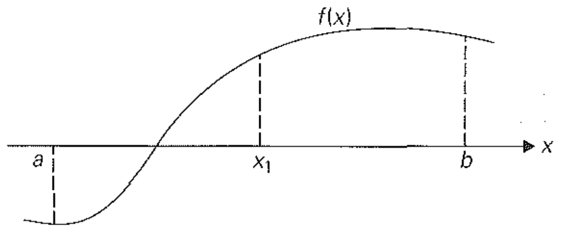


FIGURE 12.17 First iteration of the bisection method: x_1 is the middle point of interval $[a, b]$.

x^* by x_n, the middle point of this interval, is smaller than some small preselected number $\varepsilon > 0$. Since x_n is the middle point of $[a_n, b_n]$ and x^* lies within this interval as well, we have

$$|x_n - x^*| \le \frac{b_n - a_n}{2}. \tag{12.5}$$

Hence, we can stop the algorithm as soon as $(b_n - a_n)/2 < \varepsilon$ or, equivalently,

$$x_n - a_n < \varepsilon. \tag{12.6}$$

It is not difficult to prove that

$$|x_n - x^*| \le \frac{b_1 - a_1}{2^n} \quad \text{for } n = 1, 2, \ldots . \tag{12.7}$$

This inequality implies that the sequence of approximations $\{x_n\}$ can be made as close to root x^* as we wish by choosing n large enough. In other words, we can say that $\{x_n\}$ ***converges*** to root x^*. Note, however, that because any digital computer represents extremely small values by zero (Section 11.4), the convergence assertion is true in theory but not necessarily in practice. In fact, if we choose ε below a certain machine-dependent threshold, the algorithm may never stop! Another source of potential complications is round-off errors in evaluating values of the function in question. Therefore it is a good practice to include in a program implementing the bisection method a limit on the number of iterations the algorithm is allowed to run.

Here is a pseudocode for the bisection method.

ALGORITHM *Bisection*$(f(x), a, b, eps, N)$

```
//Implements the bisection method for finding a root of $f(x) = 0$
//Input: Two real numbers $a$ and $b$, $a < b$,
//        a continuous function $f(x)$ on $[a, b]$, $f(a)f(b) < 0$,
//        an upper bound on the absolute error $eps > 0$,
//        an upper bound on the number of iterations $N$
//Output: An approximate (or exact) value $x$ of a root in $(a, b)$
//or an interval bracketing the root if the iteration number limit is reached
$n \leftarrow 1$   //iteration count
while $n \le N$ do
    $x \leftarrow (a + b)/2$
    if $x - a < eps$ return $x$
    $fval \leftarrow f(x)$
    if $fval = 0$ return $x$
    if $fval * f(a) < 0$
        $b \leftarrow x$
    else $a \leftarrow x$
    $n \leftarrow n + 1$
return "iteration limit", $a$, $b$
```

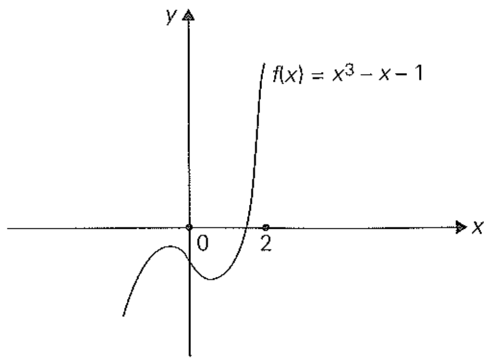


FIGURE 12.18 Graph of function $f(x) = x^3 - x - 1$

Note that we can use inequality (12.7) to find in advance the number of iterations that should suffice, at least in theory, to achieve a preselected accuracy level. Indeed, choosing the number of iterations n large enough to satisfy inequality $(b_1 - a_1)/2^n < \varepsilon$, i.e.,

$$n > \log_2 \frac{b_1 - a_1}{\varepsilon} \qquad \textbf{(12.8)}$$

does the trick.

EXAMPLE 1 Let us consider equation

$$x^3 - x - 1 = 0. \qquad \textbf{(12.9)}$$

It has one real root. (See Figure 12.18 for the graph of $f(x) = x^3 - x - 1$.) Since $f(0) < 0$ and $f(2) > 0$, the root must lie within interval (0, 2). If we choose the error tolerance level as $\varepsilon = 10^{-2}$, inequality (12.8) would require $n > \log_2(2/10^{-2})$ or $n \geq 8$ iterations.

Figure 12.19 contains a trace of the first eight iterations of the bisection algorithm applied to solving equation (12.9). Thus, we obtained $x_8 = 1.3203125$ as an approximate value for the root x^* of equation (12.9), and we can guarantee that

$$|1.3203125 - x^*| < 10^{-2}.$$

Moreover, if we take into account the signs of the left-hand side of equation (12.9) at a_8, b_8, and x_8, we can assert that the root lies between 1.3203125 and 1.328125. ▨

The principal weakness of the bisection method as a general algorithm for solving equations is its slow rate of convergence compared with other known methods. It is for this reason that the method is rarely used. Also, it cannot be

n	a_n	b_n	x_n	$f(x_n)$
1	0.0−	2.0+	1.0	−1.0
2	1.0−	2.0+	1.5	0.875
3	1.0−	1.5+	1.25	−0.296875
4	1.25−	1.5+	1.375	0.224609
5	1.25−	1.375+	1.3125	−0.051514
6	1.3125−	1.375+	1.34375	0.082611
7	1.3125−	1.34375+	1.328125	0.014576
8	1.3125−	1.328125+	1.3203125	−0.018711

FIGURE 12.19 Trace of the bisection method for solving equation (12.9). The signs after the numbers in the second and third columns indicate the sign of $f(x) = x^3 - x - 1$ at the corresponding endpoints of the intervals.

extended to solving more general equations and systems of equations. But it does have several strong points. It always converges to a root whenever we start with an interval whose properties are very easy to check. And it does not use derivatives of the function $f(x)$ as some faster methods do.

What important algorithm does the method of bisection remind you of? If you have found it to closely resemble binary search, you are correct. Both of them solve variations of the searching problem, and they are both divide-by-half algorithms. The principal difference lies in the problem's domain: discrete for binary search and continuous for the bisection method. Also note that while binary search requires its input array to be sorted, the bisection method does not require its function to be nondecreasing or nonincreasing. Finally, while binary search is very fast, the bisection method is slow.

Method of False Position

The ***method of false position*** (also known by its name in Latin, ***regula falsi***) is to interpolation search as the bisection method is to binary search. Like the bisection method, it has, on each iteration, some interval $[a_n, b_n]$ bracketing a root of a continuous function $f(x)$ that has opposite-sign values at a_n and b_n. Unlike the bisection method, however, it computes the next root approximation not as the middle of $[a_n, b_n]$ but as the x-intercept of the straight line through the points $(a_n, f(a_n))$ and $(b_n, f(b_n))$ (Figure 12.20).

You are asked in the exercises to show that the formula for this x-intercept can be written as

$$x_n = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}. \qquad \textbf{(12.10)}$$

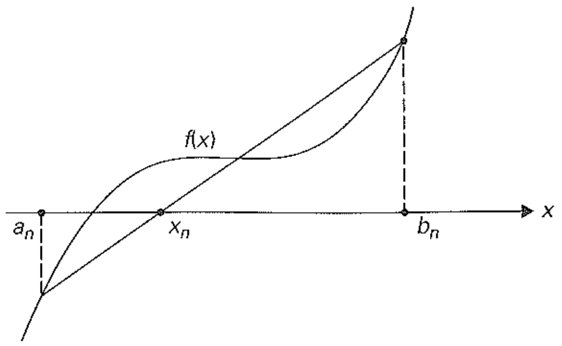


FIGURE 12.20 Iteration of the method of false position

	a_n	b_n	x_n	$f(x_n)$
1	0.0−	2.0+	0.333333	−1.296296
2	0.333333−	2.0+	0.676471	−1.366909
3	0.676471−	2.0+	0.960619	−1.074171
4	0.960619−	2.0+	1.144425	−0.645561
5	1.144425−	2.0+	1.242259	−0.325196
6	1.242259−	2.0+	1.288532	−0.149163
7	1.288532−	2.0+	1.309142	−0.065464
8	1.309142−	2.0+	1.318071	−0.028173

FIGURE 12.21 Trace of the method of false position for equation (12.9). The signs after the numbers in its second and third columns indicate the sign of $f(x) = x^3 - x - 1$ at the corresponding endpoints of the intervals.

EXAMPLE 2 Figure 12.21 contains the results of the first eight iterations of this method for solving equation (12.9).

Although for this example the method of false position does not perform as well as the bisection method, for many instances it yields a faster converging sequence. ■

Newton's Method

Newton's method, also called the ***Newton-Raphson method***, is one of the most important general algorithms for solving equations. When applied to equation (12.4) in one unknown, it can be illustrated by Figure 12.22: the next element

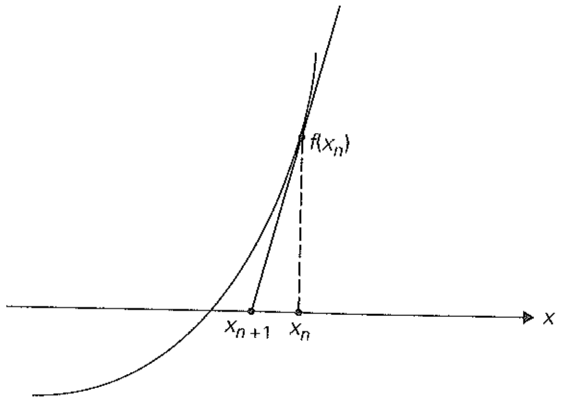


FIGURE 12.22 Iteration of Newton's method

x_{n+1} of the method's approximation sequence is obtained as the x-intercept of the tangent line to the graph of function $f(x)$ at x_n.

The analytical formula for the elements of the approximation sequence turns out to be

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{for } n = 0, 1, \ldots. \tag{12.11}$$

In most cases, Newton's algorithm guarantees convergence of sequence (12.11) if an initial approximation x_0 is chosen "close enough" to the root. (Precisely defined prescriptions for choosing x_0 can be found in numerical analysis textbooks.) It may converge for initial approximations far from the root as well, but this is not always true.

EXAMPLE 3 Computing $\sqrt{a}$ for $a \geq 0$ can be done by finding a nonnegative root of equation $x^2 - a = 0$. If we use formula (12.11) for this case of $f(x) = x^2 - a$ and $f'(x) = 2x$, we obtain

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{x_n^2 + a}{2x_n} = \frac{1}{2}(x_n + \frac{a}{x_n}),$$

which is exactly the formula we used in Section 11.4 for computing approximate values of square roots. ■

EXAMPLE 4 Let us apply Newton's method to equation (12.9), which we previously solved with the bisection method and the method of false position. Formula (12.11) for this case becomes

$$x_{n+1} = x_n - \frac{x_n^3 - x_n - 1}{3x_n^2 - 1}.$$

n	x_n	x_{n+1}	$f(x_{n+1})$
0	2.0	1.545455	1.145755
1	1.545455	1.359615	0.153705
2	1.359615	1.325801	0.004625
3	1.325801	1.324719	$4.7 \cdot 10^{-6}$
4	1.324719	1.324718	$5 \cdot 10^{-12}$

FIGURE 12.23 Trace of Newton's method for equation (12.9)

As an initial element of the approximation sequence, we take, say, $x_0 = 2$. Figure 12.23 contains the results of the first five iterations of Newton's method. ■

You cannot fail to notice how much faster Newton's approximation sequence converges to the root than the approximation sequences of both the bisection method and the method of false position. This very fast convergence is typical of Newton's method if an initial approximation is close to the equation's root. Note, however, that on each iteration of this method we need to evaluate new values of the function and its derivative, whereas the previous two methods require only one new value of the function itself. Also, Newton's method does not bracket a root as these two methods do. Moreover, for an arbitrary function and arbitrarily chosen initial approximation, its approximation sequence may diverge. And, because formula (12.11) has the function's derivative in the denominator, the method may break down if it is equal to zero. In fact, Newton's method is most effective when $f'(x)$ is bounded away from zero near root x^*. In particular, if

$$|f'(x)| \geq m_1 > 0$$

on the interval between x_n and x^*, we can estimate the distance between x_n and x^* by using the Mean Value Theorem of calculus as follows:

$$f(x_n) - f(x^*) = f'(c)(x_n - x^*),$$

where c is some point between x_n and x^*. Since $f(x^*) = 0$ and $|f'(c)| \geq m_1$, we obtain

$$|x_n - x^*| \leq \frac{|f(x_n)|}{m_1}. \qquad \textbf{(12.12)}$$

Formula (12.12) can be used as a criterion for stopping Newton's algorithm when its right-hand side becomes smaller than a preselected accuracy level ε. Other possible stopping criteria are

$$|x_n - x_{n-1}| < \varepsilon$$

and

$$|f(x_n)| < \varepsilon,$$

where ε is a small positive number. Since the last two criteria do not necessarily imply closeness of x_n to root x^*, they should be considered inferior to the one based on (12.12).

The shortcomings of Newton's method should not overshadow its principal strengths: fast convergence for an appropriately chosen initial approximation and applicability to much more general types of equations and systems of equations.

Exercises 12.4

1. **a.** Find on the Internet or in your library a procedure for finding a real root of the general cubic equation $ax^3 + bx^2 + cx + d = 0$ with real coefficients.

   b. What general algorithm design technique is it based on?

2. Indicate how many roots each of the following equations has.

   a. $xe^x - 1 = 0$ **b.** $x - \ln x = 0$ **c.** $x \sin x - 1 = 0$

3. **a.** Prove that if $p(x)$ is a polynomial of an odd degree, then it must have at least one real root.

   b. Prove that if x_0 is a root of an n-degree polynomial $p(x)$, the polynomial can be factored into

$$p(x) = (x - x_0)q(x),$$

   where $q(x)$ is a polynomial of degree $n - 1$. Explain what significance this theorem has for finding roots of a polynomial.

   c. Prove that if x_0 is a root of an n-degree polynomial $p(x)$, then

$$p'(x_0) = q(x_0),$$

   where $q(x)$ is the quotient of the division of $p(x)$ by $x - x_0$.

4. Prove inequality (12.7).

5. Apply the bisection method to find the root of the equation

$$x^3 + x - 1 = 0$$

   with an absolute error smaller than 10^{-2}.

6. Derive formula (12.10) underlying the method of false position.

7. Apply the method of false position to find the root of the equation

$$x^3 + x - 1 = 0$$

   with an absolute error smaller than 10^{-2}.

8. Derive formula (12.11) underlying Newton's method.

9. Apply Newton's method to find the root of the equation

$$x^3 + x - 1 = 0$$

with an absolute error smaller than 10^{-2}.

10. Give an example that shows that the approximation sequence of Newton's method may diverge.

11. *Gobbling goat* There is a grassy field in the shape of a circle with a radius of 100 feet. A goat is attached by a rope to a hook at a fixed point on the field's border. How long should the rope be to let the goat reach only half of the grass in the field?

SUMMARY

- *Backtracking* and *branch-and-bound* are two algorithm design techniques for solving problems in which the number of choices grows at least exponentially with their instance size. Both techniques construct a solution one component at a time, trying to terminate the process as soon as one can ascertain that no solution can be obtained as a result of the choices already made. This approach makes it possible to solve many large instances of *NP*-hard problems in an acceptable amount of time.

- Both backtracking and branch-and-bound employ, as their principal mechanism, a *state-space tree*—a rooted tree whose nodes represent partially constructed solutions to the problem in question. Both techniques terminate a node as soon as it can be guaranteed that no solution to the problem can be obtained by considering choices that correspond to the node's descendants.

- *Backtracking* constructs its state-space tree in the depth-first search fashion in the majority of its applications. If the sequence of choices represented by a current node of the state-space tree can be developed further without violating the problem's constraints, it is done by considering the first remaining legitimate option for the next component. Otherwise, the method backtracks by undoing the last component of the partially built solution and replaces it by the next alternative.

- *Branch-and-bound* is an algorithm design technique that enhances the idea of generating a state-space tree with the idea of estimating the best value obtainable from a current node of the decision tree: if such an estimate is not superior to the best solution seen up to that point in the processing, the node is eliminated from further consideration.

- Approximation algorithms are often used to find approximate solutions to difficult problems of combinatorial optimization. The *performance ratio* is the principal metric for measuring the accuracy of such approximation algorithms.

- The *nearest-neighbor* and *multifragment heuristic* are two simple greedy algorithms for approximating a solution to the traveling salesman problem. The performance ratios of these algorithms are unbounded above, even for the important subset of *Euclidean graphs*.

- The *twice-around-the-tree* and *Christophides* algorithms exploit the graph's minimum spanning tree to construct an Eulerian circuit and then transform it into a Hamiltonian circuit (an approximate solution to the TSP) by shortcuts. For Euclidean graphs, the performance ratios of these algorithms are 2 and 1.5, respectively.

- *Local search heuristics—the 2-opt, 3-opt,* and *Lin-Kernighan* algorithms—work by replacing a few edges in the current tour to find a shorter one until no such replacement can be found. These algorithms are capable of finding in seconds a tour that is within a few percent of optimum for large Euclidean instances of the traveling salesman problem.

- A sensible greedy algorithm for the knapsack problem is based on processing an input's items in descending order of their value-to-weight ratios. For the continuous version of the problem, this algorithm always yields an exact optimal solution.

- *Polynomial approximation schemes* for the knapsack problem are polynomial-time parametric algorithms that approximate solutions with any predefined accuracy level.

- Solving nonlinear equations is one of the most important areas of numerical analysis. While there are no formulas for roots of nonlinear equations (with a few exceptions), several algorithms can solve them approximately.

- The *bisection method* and the *method of false position* are continuous analogues of binary search and interpolation search, respectively. Their principal advantage lies in bracketing a root on each iteration of the algorithm.

- *Newton's method* generates a sequence of root approximations that are x-intercepts of tangent lines to the function's graph. With a good initial approximation, it typically requires just a few iterations to obtain a high-accuracy approximation to the equation's root.