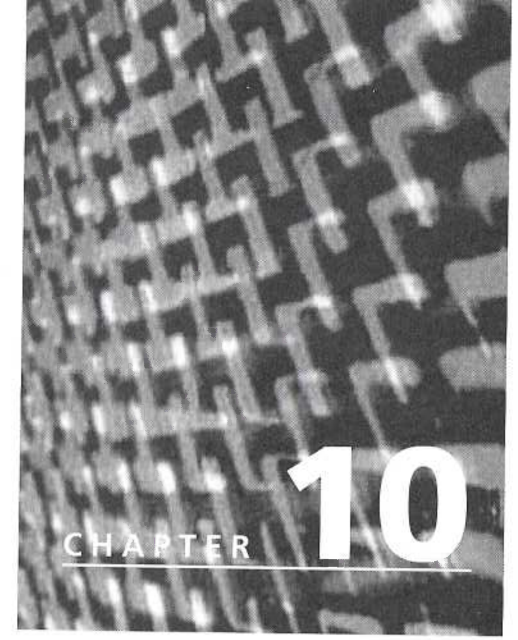


BACKTRACKING AND BRANCH-AND-BOUND

Suppose you enter a maze of garden hedges, like that shown in Figure 10.1. To find the exit of the maze (or to determine that there is no exit), you need a guaranteed strategy. Does such a strategy exist? Yes. You simply walk along the trail with your left hand always touching the hedge. If you hit a dead end, you turn around and backtrack, still keeping your left hand touching the hedge. This left-hand backtracking strategy will always lead you to the exit, or return you to the entrance if there is no exit. It even works when you're blindfolded.

Of course, by symmetry, an algorithm for generating a path through a maze can also be based on the right-hand backtracking strategy. For the maze shown in Figure 10.1, the right-hand backtracking strategy generates a slightly shorter path. In general, however, neither backtracking strategy generates the shortest path. (In fact, Figure 10.1 provides an example.) Thus, although backtracking always gets you through the maze, some good heuristics might yield a shorter path (see Figure 10.2).

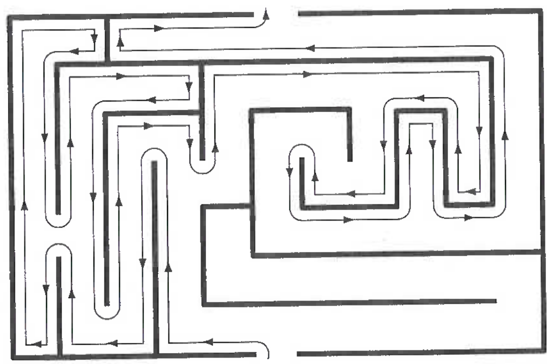


FIGURE 10.1 Garden maze and path to exit generated by the left-hand backtracking algorithm.

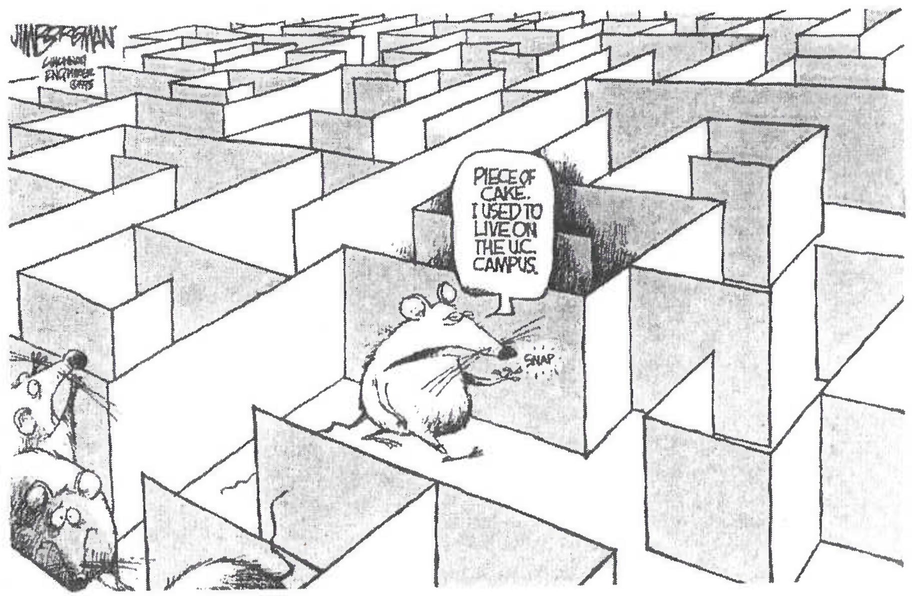


FIGURE 10.2 The left-hand backtracking algorithm versus heuristic search.

We can associate an implicit state-space T tree with the maze problem, where a node (or state) in T corresponds to a *sequence of decisions* made leading to a junction point in the maze. The children of a node correspond to the various branches from that junction point, where we restrict the choice of branches to those not leading to a junction already visited (this avoids cycles, yielding a finite tree). The left-hand backtracking algorithm is based on performing a depth-first search of the state-space tree.

10.1 State-Space Trees

The backtracking and branch-and-bound design strategies are applicable to any problem whose solution can be expressed as a sequence of decisions. Both backtracking and branch-and-bound are based on a search of an associated state-space tree that models all possible sequences of decisions. There may be several different ways to model decision sequences for a given problem, with each model leading to a different state-space tree. We assume in our model for the decision-making process that the decision x_k at stage k must be drawn from a finite set of choices. For each $k > 1$, the choices available for decision x_k may be limited by the choices that have already been made for $x_1, \ldots, x_{k-1}$.

For a given problem instance, suppose n is the maximum number of decision stages that can occur. For $k \leq n$, we let P_k denote the set of all possible sequences of k decisions, represented by k-tuples $(x_1, x_2, \ldots, x_k)$. Elements of P_k are called *problem states*, and problem states that correspond to solutions to the problem are called *goal states*.

Given a problem state $(x_1, \ldots, x_{k-1}) \in P_{k-1}$, we let $D_k(x_1, \ldots, x_{k-1})$ denote the *decision set* consisting of the set of all possible choices for decision x_k. Letting $\varnothing$ denote the null tuple ( ), note that $D_1(\varnothing)$ is the set of choices for x_1.

The decision sets $D_k(x_1, \ldots, x_{k-1})$, $k = 1, \ldots, n$, determine a decision tree T of depth n, called the *state-space tree*. The nodes of T at level k, $0 \leq k \leq n$, are the problem states $(x_1, \ldots, x_k) \in P_k$ (P_0 consists of the null tuple). For $1 \leq k < n$, the children of $(x_1, \ldots, x_{k-1})$ are the problem states $\{(x_1, \ldots, x_k) \mid x_k \in D_k(x_1, \ldots, x_{k-1})\}$.

A state-space tree that models a problem whose set of decision choices $D_k(x_1, \ldots, x_{k-1})$ depends only on the input size is called *static*. A state-space tree in which D_k depends on not only the input size but also the particular input is called *dynamic*. For example, if we are solving the knapsack problem with objects $b_0, \ldots, b_{n-1}$, a static state-space tree might be one in which the first decision is whether to include b_0, the second decision is whether to include b_1, and so forth (we are describing what we will call the static *fixed-tuple* state-space tree for the knapsack problem). On the other hand, based on some heuristic dependent on $b_0, \ldots, b_{n-1}$, we might decide that our first decision is whether to include b_m, where m could be different from zero. Similar comments hold for other levels in the tree, leading to a dynamic state-space tree modeling the knapsack problem. In this chapter, we always use static state-space trees, but we do introduce a dynamic state-space tree in connection with the backtracking solution to the conjunctive normal form (CNF) satisfiability problem developed in the exercises.

10.1.1 An Example

Our first illustration of state-space trees is for the *sum of subsets problem*. An input to the sum of subsets problem is a multiset $A = \{a_0, \dots, a_{n-1}\}$ of n positive integers, together with a positive integer *Sum*. A solution to the sum of subsets problem is a subset of elements $a_{i_1}, \dots, a_{i_k}$ of A, $i_1 < \dots < i_k$, such that $a_{i_1} + \dots + a_{i_k} = Sum$.

The sum of subsets problem can be interpreted as the problem of making correct change, where a_i represents the denomination of the $(i + 1)^{st}$ coin, $i = 0, \dots, n - 1$, and *Sum* represents the desired change. This differs from the version of the coin-changing problem discussed in Chapter 7, because here a limited number of coins of each denomination are available. For example, consider the multiset $A = \{25, 1, 1, 1, 5, 10, 1, 10, 25\}$. The denominations are 1, 5, 10, 25, which occur with multiplicities 4, 1, 2, 2, respectively.

There are two natural ways to model a decision sequence leading to a solution to the sum of subsets problem. In the first model, a problem state consists of choosing k elements $a_{i_1}, \dots, a_{i_k}$ of A, $i_1 < \dots < i_k$, in succession, for some $k \in \{1, \dots, n\}$. The decision sequence can be represented by the k-tuple $(x_1, \dots, x_k) = (i_1, \dots, i_k)$, where x_j corresponds to the decision to choose element a_i at stage j, $1 \le j \le k$. For example, consider the instance $n = 5$, and suppose that we have decided to choose the second, fourth, and fifth elements. Then $x_1 = 1$, $x_2 = 3$, and $x_3 = 4$, so that the problem state associated with this decision sequence is the 3-tuple (1, 3, 4).

Given that problem state $(x_1, \dots, x_{k-1})$ has occurred (that is, the decision has been made to choose elements $a_{x_1}, \dots, a_{x_{k-1}}$), then the available choices for decision x_k are $a_{x_{k-1}+1}, \dots, a_n$, yielding

$$D_k(x_1, \dots, x_{k-1}) = \{x_{k-1} + 1, x_{k-1} + 2, \dots, n - 1\}, \quad 1 \le k \le n. \quad \textbf{(10.1.1)}$$

For example, suppose $n = 5$ and that problem state (0, 2) has occurred. The only elements available for the third decision are a_3 and a_4, so that $D_3(0, 2) = \{3, 4\}$. Figure 10.3 illustrates the state-space tree T determined by $D_k(x_1, \dots, x_{k-1})$ for the sum of subsets problem, where $n = 5$. The goal states (nodes) are not determined until a particular instance of the problem is specified. For example, for the instance $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$, the goal states are (0, 2, 4), (1, 2), and (2, 4). On the other hand, for the same set A, if $Sum = 10$, then the goal states are (0, 1, 2), (3), and (0, 2, 4). State-space trees like this, in which the size of the goal states can vary for the same input size, are called *variable-tuple* state-space trees.

The second natural way to model the sum of subsets problem is an example of a *fixed-tuple* model, in which goal states can be considered as n-tuples. In this

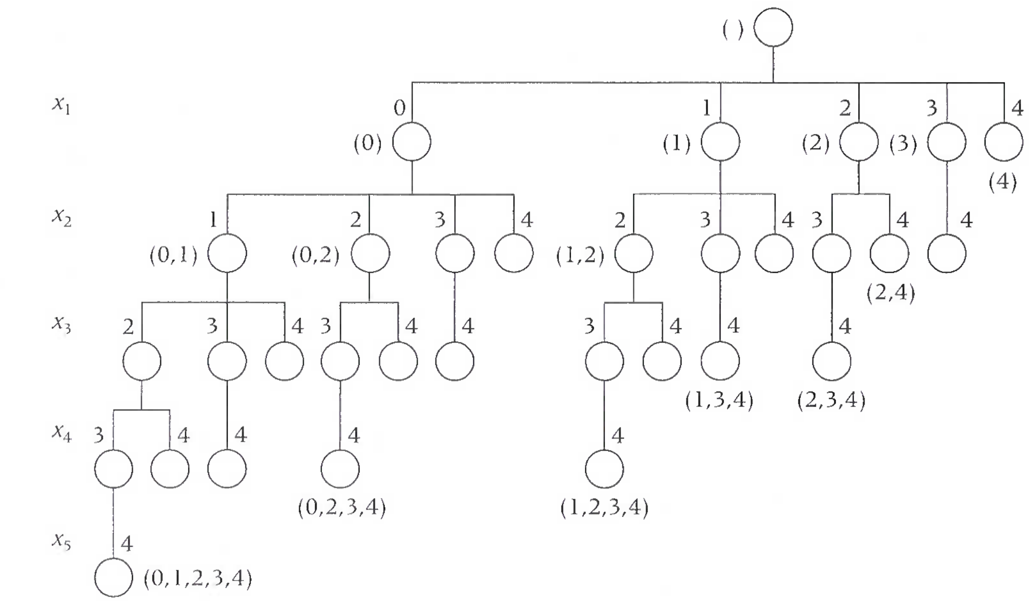


FIGURE 10.3

Variable-tuple state-space tree T modeling the decision set D_k given by Formula (10.1.1) for the sum of subsets problem with $n = 5$. Edges are labeled with the indices of the chosen elements. Index values of the problem states are shown outside some sample nodes.

model, the decision at stage k is whether to choose element a_{k-1}, $1 \le k \le n$. Thus, $D_k = \{0, 1\}$, where $x_k = 1$ if element a_{k-1} is chosen, and $x_k = 0$ otherwise. Thus, the state-space tree T associated with the decision sets $D_k(x_1, \dots, x_{k-1})$ is the full binary tree on $2^{n+1} - 1$ nodes, with a left child of a node at level $k - 1$ corresponding to choosing a_{k-1} ($x_k = 1$) and a right child corresponding to omitting a_{k-1} ($x_k = 0$), so that

$$D_k(x_1, \dots, x_{k-1}) = \{0, 1\}, \quad 1 \le k \le n. \quad \textbf{(10.1.2)}$$

Figure 10.4 shows the fixed-tuple state-space tree for the same sum of subsets problem illustrated in Figure 10.3. For the same instance $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$ considered earlier, the goal states are now represented by the 5-tuples (1, 1, 0, 0, 1), (0, 1, 1, 0, 0), and (0, 0, 1, 0, 1).

10.1.2 Searching State-space Trees

The state-space tree for most problems is large (exponential or worse in the input size). Thus, while a brute-force search of the entire state-space tree has the advantage of always finding a goal state if one exists, the search might not end in

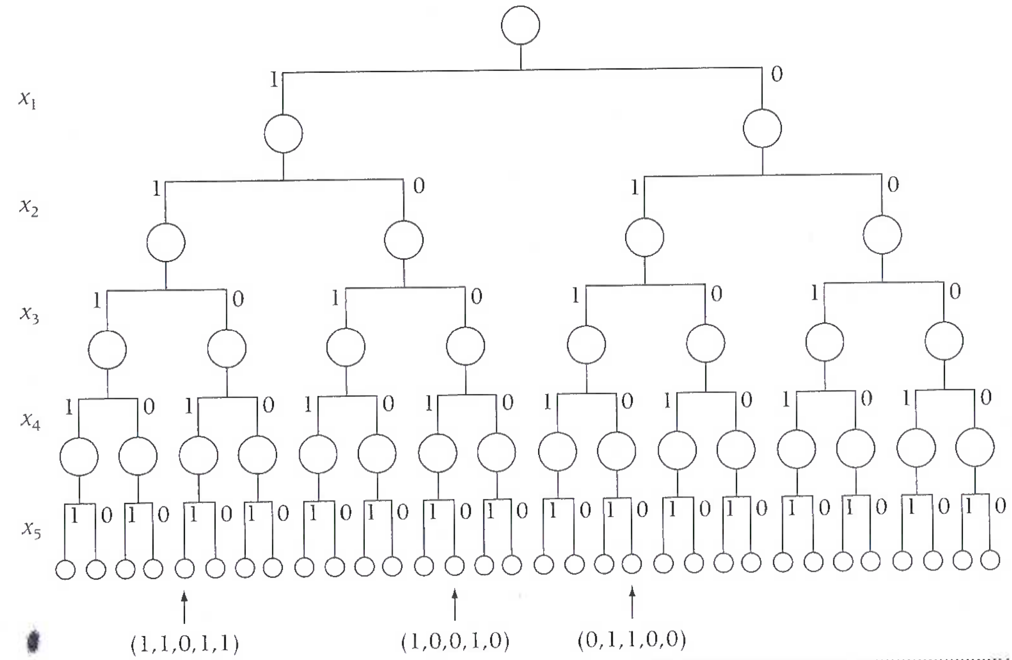


FIGURE 10.4

Fixed-tuple state-space tree T modeling the decision set D_k given by Formula (10.1.2) for the sum of subsets problem with $n = 5$. Edges ending at level k are labeled 1 or 0, depending on whether a_k was chosen or not. Labels of the path from the root to some sample leaf nodes shown.

a single lifetime, even for relatively small input sizes. However, we can often determine that there is no goal node in the subtree rooted at a given node X in the state-space tree. In this case, we say that X is *bounded*, and we can prune the state-space tree by eliminating the descendants of node X. Thus, when searching state-space trees, we look for good bounding functions. *Bounded* is a Boolean function such that if *Bounded*(X) is **.true.**, then there is no descendant of X that is a goal node. Good bounding functions can possibly limit the search to relatively small portions of the state-space tree.

Key Fact

An algorithm that performs a potentially complete search of a state-space tree modeling a given problem will always find a goal state if one exists. However, the state-space tree usually grows exponentially with the input size to the problem, so unless good bounding functions can be found to limit the search, such an algorithm will usually be too inefficient in the worst case to be practical.

In this chapter, we discuss two general-purpose design strategies based on searching the state-space tree associated with a given problem: *backtracking* and *branch-and-bound*. The state-space tree is usually implicit to backtracking algorithms, whereas branch-and-bound algorithms usually require the state-space tree to be explicitly implemented. Backtracking is based on a depth-first search of the state-space tree. When a node is accessed during a backtracking search, it becomes the current node being expanded (called the *E*-node), but immediately, its first child not yet visited becomes the new *E*-node. On the other hand, branch-and-bound algorithms are based on breadth-first searches of the state-space tree that generate all the children of the *E*-node when the node is first accessed. Thus, a node can be the *E*-node many times during a backtracking search, but a node is the *E*-node at most once during a branch-and-bound algorithm. There are various versions of branch-and-bound, differing only in the manner in which the next *E*-node is chosen. For example, the children might be placed on a queue (FIFO branch-and-bound), a stack (LIFO branch-and-bound), or a priority queue (least-cost branch-and-bound).

Key Fact

Backtracking is based on a depth-first search of the state-space tree *T*, and only needs to explicitly maintain the current path (or problem state) at any given point in the search. Branch-and-bound is based on a breadth-first search and normally needs to explicitly maintain the entire portion already reached in the search (except for nodes that are bounded).

As mentioned earlier, unless good bounding functions can be found, backtracking and branch-and-bound tend to be inefficient in the worst case. However, they can be applied in a wider variety of settings than the other major design strategies that we have discussed. Moreover, there are many practical and important problems for which the best solutions known are based on backtracking or branch-and-bound together with clever heuristics to bound the search. This is especially true for the NP-complete problems, such as the fundamental problem of determining the satisfiability of CNF Boolean expressions. The best-known solutions to CNF satisfiability are based on backtracking searches of dynamic state-space trees modeling the input as determined by clever heuristics and bounding strategies.

10.2 Backtracking

Before stating the general backtracking design strategy, we illustrate the method by applying it to the sum of subsets problem discussed in the previous section.

10.2.1 A Backtracking Algorithm for the Sum of Subsets Problem

Initially, we will not assume that the set $A = \{a_0, \ldots, a_{n-1}\}$ is ordered. (Later, we show that by sorting A in increasing order, we can obtain an improved bounding function.) We use the decision sequence formulation corresponding to the variable-tuple state-space tree, so that the decision sets $D_k(x_1, \ldots, x_{k-1})$ are given by Formula (10.1.1).

To motivate the definition of a bounding function for the problem states, we consider the instance of the sum of subsets problem where $n = 5$, $A = \{1, 4, 5, 10, 4\}$, and $Sum = 9$. The state-space tree for this instance, and the three goal states $(x_1, x_2, x_3) = (0, 1, 4)$, $(x_1, x_2) = (1, 2)$, and $(x_1, x_2) = (2, 4)$, are shown in Figure 10.5.

For example, consider the problem state (0, 1, 2) in the state-space tree in Figure 10.5 corresponding to the choice of elements a_0, a_1, a_2. Note that $a_0 + a_1 + a_2 = 10 > Sum = 9$. Further, any extension of (0, 1, 2) corresponds to a set of elements whose sum is even greater than 10. Thus, there is no path in the state-space tree from (0, 1, 2) to a goal state. In other words, *"You can't get there from here!"* (See Figure 10.6.) We *bound* node (0, 1, 2) because there is no need to examine any of its descendants.

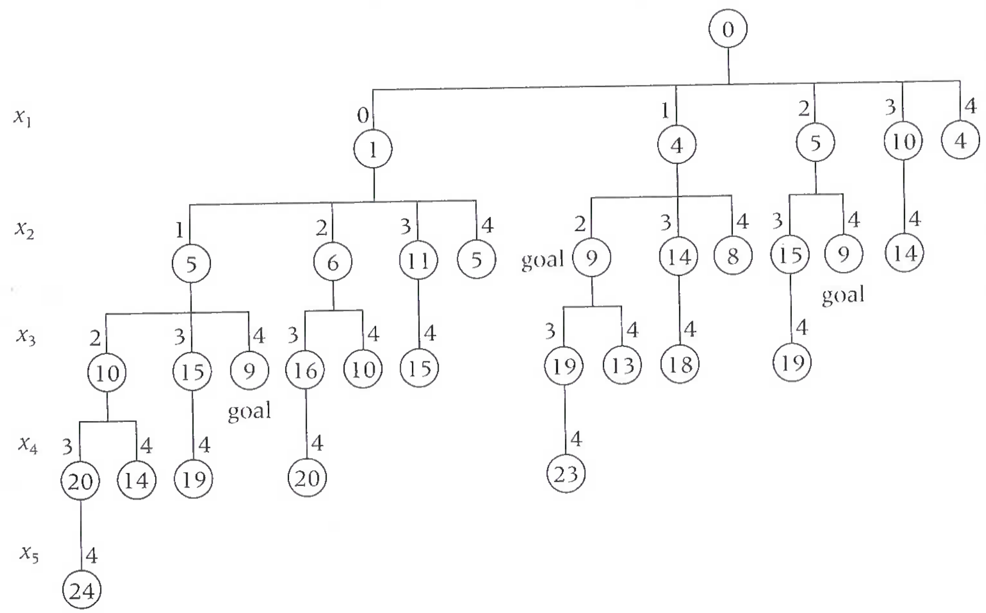


FIGURE 10.5

Variable-tuple state-space tree for sum of subsets problem with $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$. The value of the sum of the elements chosen is shown inside each node. Edges are labeled with the indices of the chosen elements.

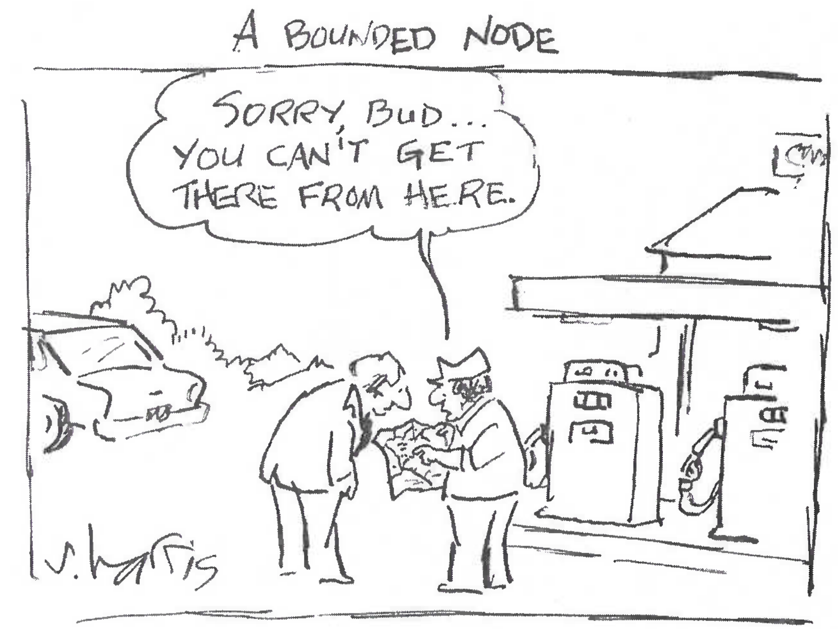


FIGURE 10.6

A bounded node

Source: © 1996 by Sidney Harris. Altered and reprinted with permission. All rights reserved.

For the general problem state $(x_1, \ldots, x_k) \in P_k$, we define the bounding function $Bounded(x_1, \ldots, x_k)$ by

$$Bounded(x_1, \ldots, x_k) = \begin{cases} \textbf{.true.} & \text{if } s_{x_1} + \cdots + s_{x_k} \geq Sum, \\ \textbf{.false.} & \text{otherwise.} \end{cases} \qquad \textbf{(10.2.1)}$$

Clearly, if the elements corresponding to $(x_1, \ldots, x_k)$ have a sum greater than or equal to *Sum*, then any extension of $(x_1, \ldots, x_k)$ corresponds to a set of elements whose sum is strictly greater than *Sum*. Thus, if $Bounded(x_1, \ldots, x_k)$ = **.true.**, then no descendant of $(x_1, \ldots, x_k)$ can be a goal state. For the problem instance $A = (1, 4, 5, 10, 4)$, Figure 10.7 shows the state-space tree T (from Figure 10.5) after it has been pruned at all the nodes bounded by Formula (10.2.1). The bounded nodes are labeled B, except the bounded nodes that are also goal nodes, which are so labeled. The unlabeled leaves correspond to nodes that were leaves in the original state-space T, before pruning.

The backtracking strategy performs a depth-first search of the state-space tree T, using an appropriate bounding function. By convention, when moving from an E-node to the next level of the state-space tree, we select the leftmost child not already visited. If no such child exists, or if the E-node is bounded, then we backtrack to the previous level. If only one solution to the problem is desired,

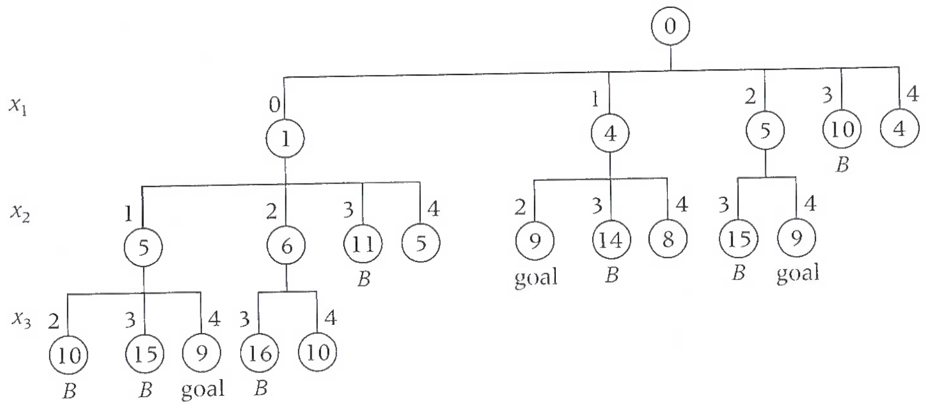


FIGURE 10.7

Pruned state-space tree for the sum of subsets problem with $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$. Edges are labeled with the indices of the chosen elements. Bounded nodes are labeled *B*.

then the backtracking algorithm terminates once a goal state is found. Otherwise, the algorithm continues until all the nodes have been exhausted, outputting each goal state when it is reached.

We now give pseudocode for nonrecursive and recursive backtracking procedures for solving the sum of subsets problem. These procedures perform a depth-first search of the variable-tuple state-space tree *T*, using the bounding function given in Formula (10.2.1). The procedures output all goal nodes but can be trivially modified to terminate once the first goal state is found.

```
procedure SumOfSubsets(A[0:n − 1], Sum, X[0:n])
Input:   A[0:n − 1] (an array of positive integers)
         Sum (a positive integer)
         X[0:n] (an array of integers where X[1:n] stores variable-tuple problem states,
            and X[0] = −1 for convenience of pseudocode)
Output:  print all goal states; that is, print all (X[1], ... , X[k]) such that
            A[X[1]] + ··· + A[X[k]] = Sum
   for i ← 0 to n do
      X[i] ← −1
   endfor
   PathSum ← 0
   k ← 1
   while k ≥ 1 do                       //E-node is (X[1], ... , X[k − 1]). Initially
                                        // E-node = ( ) corresponding to root
      ChildSearch ← .true.
      while ChildSearch do              // searching for unbounded child of E-node
         if X[k] = −1 then
            X[k] ← X[k − 1] + 1         // visit first child of E-node
         else
            X[k] ← X[k] + 1             //visit next child of E-node
         endif
         if X[k] > n − 1 then           // no more children of E-node
            ChildSearch ← .false.
         else
            PathSum ← PathSum + A[X[k]]
            if PathSum ≥ Sum then       //(X[1], ... , X[k]) is bounded
               if PathSum = Sum then
                  Print(X[1], ... , X[k])    //print goal state
                  PathSum ← PathSum − A[X[k]]
               endif
            else                        //(X[1], ... , X[k]) is not bounded
               ChildSearch ← .false.
            endif
         endif
      endwhile
      if X[k] > n − 1 then
         X[k] = 0         //backtrack to previous level; no more children of E-node
         k ← k − 1
      else
         k ← k + 1        //go one more level deep in state-space tree
      endif
   endwhile
end SumOfSubsets
```


The following recursive backtracking algorithm *SumOfSubsetsRec*(k) for the sum of subsets problem is called initially with $k = 0$. We assume that $A[0{:}n-1]$, *Sum*, and $X[0{:}n-1]$ are global variables. When calling *SumOfSubsetsRec* with input parameter k, it is assumed that $X[1], \ldots, X[k]$ have already been assigned values, so that $k = 0$ on the initial call.

```
procedure SumOfSubsetsRec(k) recursive
Input:   k (a nonnegative integer, 0 on initial call)
         A[0:n − 1] (global array of positive integers)
         Sum (global positive integer)
         X[0:n] (an array of integers, where X[1:n] stores variable-tuple problem states,
         X[1], ... , X[k] have already been assigned, and where X[0] = −1 for
         convenience of pseudocode)
         PathSum (global variable = A[X[1]] + ··· + A[X[k]])
Output:  print all descendant goal states of (X[1], ··· , X[k]); that is, print all (X[1], ... ,
         X[k], X[k + 1], ... , X[q]) such that A[X[1]] + ··· + A[X[k]] + A[X[k + 1]] +
         ··· + A[X[q]] = Sum
   k ← k + 1                            // move on to next level
```

```
        for Child ← X[k − 1] + 1 to n − 1 do
            X[k] ← Child
                PathSum ← PathSum + A[X[k]]
            if PathSum ≥ Sum then                  //(X[1], … , X[k]) is bounded
                if PathSum = Sum then
                    Print(X[1], … , X[k])          // print goal state
                        PathSum ← PathSum − A[X[k]]
                endif
            else                                   //(X[1], … , X[k]) is not bounded
                SumOfSubsetsRec(k)
            endif
        endfor
    end SumOfSubsetsRec
```

If the elements $a_0, \ldots, a_{n-1}$ are first sorted in increasing order (the reverse of the ordering used by the greedy algorithm for the coin-changing problem), then the following bounding function can be used for the problem states $(x_1, \ldots, x_k)$. The bounding function (10.2.2) is stronger than that given by (10.2.1).

$$Bounded(x_1, \ldots, x_k) = \begin{cases} \textbf{.true.} & \text{if } a_{x_1} + \cdots + a_{x_k} + a_{x_k+1} > Sum, \\ \textbf{.false.} & \text{otherwise} \end{cases} \qquad \textbf{(10.2.2)}$$

Because $a_0 \le a_1 \le \cdots \le a_{n-1}$, whenever the elements corresponding to problem state $(x_1, \ldots, x_k, x_k + 1)$ have a sum strictly greater than *Sum*, then the elements corresponding to any problem state $(x_1, \ldots, x_k, x_{k+1})$ also have a sum strictly greater than *Sum*. Thus, (10.2.2) is a valid bounding function. *SumOfSubsets* and *SumOfSubsetsRec* can be easily modified to use the bounding function given in 10.2.2

10.2.2 The General Backtracking Paradigm

The following general backtracking paradigm, *Backtrack*, follows the backtracking strategy we just described for the sum of subsets problem. *Backtrack* finds all solutions to a given problem by searching for all goal states in a state-space tree associated with the problem. *Backtrack* invokes a bounding function, *Bounded*, for the problem states. The definition of *Bounded* depends on the particular problem being solved. We assume that an implicit ordering exists for the elements of $D_k(x_1, \ldots, x_{k-1})$.


```
procedure Backtrack()
Input:    T (implicit state-space tree associated with the given problem)
          D_k (decision set, where D_k = ∅ for k ≥ n)
          Bounded (bounding function)
Output:   all goal states
    k ← 1
    while k ≥ 1 do                          //E-node is (X[1], … , X[k − 1]). Initially
                                              E-node = ( ) corresponding to root.
        Searching ← .true.
        while Searching do                  //searching for unbounded child
            X[k] ← first of the remaining untried values from D_k(X[1], … , X[k − 1]),
                    where this value is ∅ if all values in D_k(X[1], … , X[k − 1]) have
                    been tried
            if X[k] ← ∅ then
                Searching ← .false.
            else
                if (X[1], … , X[k]) is a goal state then
                    Print(X[1], … , X[k])
                endif
                if .not. Bounded(X[1], … , X[k]) then
                    Searching ← .false.
                endif
            endif
        endwhile
            if X[k] = ∅ then
                Arrange for all values in D_k to be considered as untried
                k ← k − 1                   //backtrack to previous level
            else
                k ← k + 1                   //move on to next level
            endif
    endwhile
end Backtrack
```

The procedure *BacktrackRec* is the recursive version of the procedure *Backtrack*. Since we are essentially performing a depth-first search of the state-space tree *T* starting at the root, *BacktrackRec*(k) is initially called with $k = 0$. Note how elegantly the recursion implements the backtracking process. We assume that $D_k(X[1], \ldots, X[k])$ is empty for $k \ge n$.

```
procedure BacktrackRec(k) recursive
Input:    T (implicit state-space tree associated with the given problem)
          k (a nonnegative integer, 0 in initial call)
          D_k (decision set, where D_k = ∅ for k ≥ n)
```

```
        X[0:n] (global array where X[1:n] maintains the problem states of T, and where
            the problem state (X[1], ... , X[k]) has already been generated)
            Bounded (bounding function)
Output: all goals that are descendants of (X[1], ... , X[k])
    k ← k + 1
    for each x_k ∈ D_k(X[1], ... , X[k − 1]) do
        X[k] ← x_k
        if (X[1], ... , X[k]) is a goal state then
            Print(X[1], ... , X[k])
        endif
        if .not. Bounded(X[1], ... , X[k]) then
            BacktrackRec(k)
        endif
    endfor
end BacktrackRec
```

REMARKS

1. Often, computing only one goal state is required. We can easily modify the procedures *Backtrack* and *BacktrackRec* to halt once the first goal state is reached.
2. Notice that in a slight variation from the pseudocode for procedures *Backtrack* and *BacktrackRec*, the *SumOfSubsets* and *SumOfSubsetsRec* pseudocode only checks for a goal state after determining that a problem state is bounded. Putting off this check for a goal state applies to any problem where all the goal states are bounded.

10.2.3 Tic-Tac-Toe

Consider the problem of finding a tie board (result of a "cat's game") in the familiar game of tic-tac-toe. Here we are trying to determine whether tie games are possible, not to devise a strategy for playing the game. (In Chapter 23, we consider the problem of designing strategies for various perfect-information games such as tic-tac-toe.)

The game of tic-tac-toe involves two players A and B, who alternately place Xs and Os into unoccupied positions on a board such as the one shown in Figure 10.8.

X	X	O
O	O	X
X	X	O

FIGURE 10.8
A 3 × 3 tic-tac-toe tie board

We assume that player A starts by placing an X in any position, and then player B places an O in any remaining position. The two players continue to alternately place Xs and Os on the board until either there are three Xs in a row (horizontally, vertically, or diagonally) so that A wins, there are three Os in a row so that B wins, or all nine positions are occupied with no three Os or three Xs in a row and the game is a tie (a cat's game).

We use backtracking to find a tie board — that is, a board corresponding to a tie game (see Figure 10.8). In fact, we solve the problem for an $n \times n$ board, where a *tie board* contains no "three in row" of either Xs or Os in any 3×3 subboard of contiguous positions in the $n \times n$ board. In our definition of a tie board, we do not assume that the number of Xs and the number of Os differ by at most 1. However, it is interesting that all tie boards in the $n \times n$ board have this property, so we don't need to check for it in our backtracking algorithm.

We refer to the cell in row i and column j of the $n \times n$ board as cell (i, j), $i, j \in \{1, \ldots, n\}$. We also refer to cell (i, j) as the cell labeled k, where $k = n(i - 1) + j$; that is, k is a *row-major* labeling (see Figure 10.9). Row-major labeling of the cells is useful in defining the problem states P_k. However, when defining the bounding function, it is more convenient to use row-column labeling. The problem of finding a board filled with Xs and Os, containing no three in a row in either Xs or Os, can be expressed as a sequence of decisions in which the decision at stage k is whether to place an X or O in the cell labeled k. We set $x_k = 1$ if an X is placed

(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)

(a)

1	2	3
4	5	6
7	8	9

(b)

X	X	O
X	O	O
X		

(c)

FIGURE 10.9
(a) Row-column labeling (i, j); (b) row-major labeling k; and (c) a 3×3 board configuration corresponding to the 7-tuple (1, 1, 0, 1, 0, 0, 1)

in cell k in the k^{th} stage; otherwise, $x_k = 0$. Thus, $D_k = \{0, 1\}$, and the problem states of size k, $1 \le k \le n$, are given by

$$P_k = \{(x_1, \dots, x_k) \mid x_1, \dots, x_k \in \{0, 1\}\}. \qquad \textbf{(10.2.3)}$$

The (fixed-tuple) state-space tree T for tic-tac-toe is identical to the fixed-tuple state-space tree for the sum of subsets problem; namely, T is the full binary tree on $2^{n+1} - 1$ nodes. Figure 10.9*c* shows the 3×3 board configuration corresponding to the 7-tuple (1, 1, 0, 1, 0, 0, 1).

An obvious bounding function for tic-tac-toe is given by

$$Bounded(x_1, \dots, x_k) = \begin{cases} \textbf{.true.} & \text{if board configuration corresponding} \\ & \text{to } (x_1, \dots, x_k) \text{ contains 3 in a row,} \\ \textbf{.false.} & \text{otherwise.} \end{cases} \qquad \textbf{(10.2.4)}$$

Note that *Bounded* = **.true.** for the problem state (1, 1, 0, 1, 0, 0, 1) in Figure 10.9. If the board configuration corresponding to the problem state $(x_1, \dots, x_k)$ already contains three in a row (in either Xs or Os), then it obviously cannot be extended to a board configuration not containing three in a row. In particular, it cannot be extended to a goal state.

Before giving pseudocode for the algorithm *TicTacToe* solving the problem of finding an $n \times n$ generalized tie board configurations, we give pseudocode for the Boolean function *BoundedBoard* based on Formula (10.2.4). The board configuration is represented by the two-dimensional array $B[-1{:}n + 2, -1{:}n + 2]$, in which, for convenient implementation of *BoundedBoard* (and by abuse of notation), we assume the existence of "border" rows and columns indexed by -1, 0, $n + 1$, $n + 2$. The cells corresponding to these rows and columns are always empty; that is, $B[i, j]$ = "E" if either $i \in \{-1, 0, n + 1, n + 2\}$ or $j \in \{-1, 0, n + 1, n + 2\}$. We illustrate a bordered 4×4 board in Figure 10.10, together with an assignment of Xs and Os to the positions $k = 1, \dots, 10$.

We assume that the two-dimensional array $B[-1{:}n + 2, -1{:}n + 2]$ is global to the function *BoundedBoard*. We refer to a set of three adjacent cells along a horizontal, vertical, or diagonal line as a *winning line*. Suppose the board configuration restricted to the cells labeled $1, 2, \dots, k - 1$ in the row-major labeling of the 4×4 board contains no three in a row in either Xs or Os. Then, to determine whether the board configuration restricted to the cells labeled $1, 2, \dots, k$ contains three in a row in either Xs or Os, we merely need to check all winning lines containing the cell labeled k. Four winning lines need to be checked: one horizontal, one vertical, and two diagonal. Figure 10.11 shows these lines for the cell in the board of Figure 10.10 whose row-major label is $k = 11$.

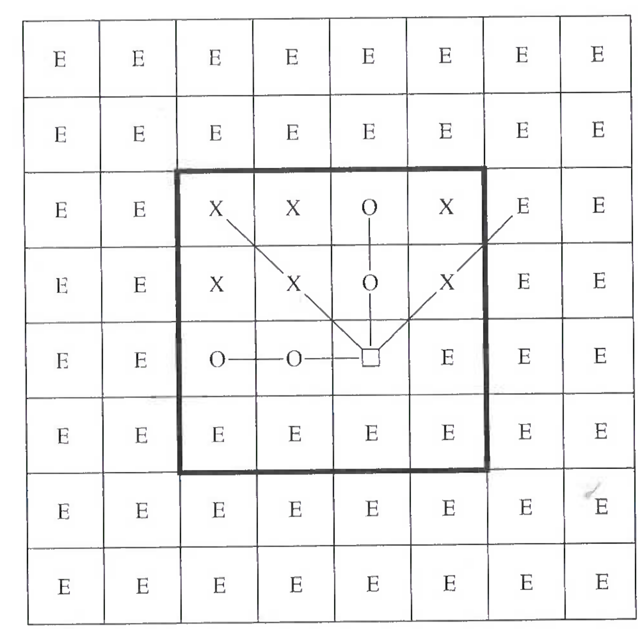
FIGURE 10.10

A bordered 4×4 tic-tac-toe board with first ten positions filled

E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E
E	E	X	X	O	X	E	E
E	E	X	X	O	X	E	E
E	E	O	O	E	E	E	E
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E

FIGURE 10.11

Winning lines that are checked for k = 11 in a $4 \times 4 \times 4$ tic-tac-toe board



The following Boolean function *BoundedBoard*(i, j) returns the value **.true.** if and only if the board configuration corresponding to $B[-1{:}n + 2, -1{:}n + 2]$ contains all *X*s or *O*s in one of the four winning lines previously described.

```
function BoundedBoard(i, j)
Input:   B[−1:n + 2, −1:n + 2]     (global array corresponding to board
                                    configuration)
         i, j                      (integers between 1 and n, inclusive)
Output:  returns .true. if the board configuration involving the cells labeled 1, ... ,
         k = n(i − 1) + j, contains three in a row in either Xs or Os along a line
         containing the cell labeled k.
    LineH ← (B[i, j] = B[i, j − 1]) .and. (B[i, j] = B[i, j − 2])
    LineV ← (B[i, j] = B[i − 1, j]) .and. (B[i, j] = B[i − 2, j])
    LineD1 ← (B[i, j] = B[i − 1, j − 1]) .and. (B[i, j] = B[i − 2, j − 2])
    LineD2 ← (B[i, j] = B[i − 1, j + 1]) .and. (B[i, j] = B[i − 2, j + 2])
    return(LineH .or. LineV .or. LineD1 .or. LineD2)
end BoundedBoard
```

We now give pseudocode for the algorithm *TicTacToe*. *TicTacToe* calls the procedures *Previous*(i, j) and *Next*(i, j), which accomplish the operations of backtracking to the previous cell ($k = k - 1$) and moving forward to the next cell ($k = k + 1$), respectively, in the row-major labeling. Thus, *Previous*(i, j) executes the statement

```
if j > 1 then
    j ← j − 1
else
    i ← i − 1
    j ← n
endif
```

and *Next*(i, j) executes the statement

```
if j < n then
    j ← j + 1
else
    i ← i + 1
    j ← 1
endif
```



```
procedure TicTacToe(n)
Input:   n (a positive integer representing size of board)
Output:  all generalized tie board configurations; that is, all board configurations not
         containing three in a row in either Xs or Os
    for i ← −1 to n + 2 do
        for j ← −1 to n + 2 do
            B[i, j] ← 'E';                  //initialize all positions on board to empty
        endfor
    endfor
    i ← 1                                   //k = 1
    j ← 1
    while i > 0 do
        if B[i, j] = 'O' then               //backtrack: k = k − 1
            B[i, j] ← 'E'
            Previous(i, j)
        else
            if B[i, j] = 'E' then
                B[i, j] ← 'X'               //visit left child of E-node
            else
                B[i, j] ← 'O'               //visit right child of E-node
            endif
            if .not. BoundedBoard(i, j) then
                if (i = n) .and. (j = n) then
                    PrintBoard(Board[1:n, 1:n])     //print goal state
                else
                    Next(i, j)
                endif
            endif
        endif
    endwhile
end TicTacToe
```

We can write a recursive version, *TicTacToeRec*(i, j), of *TicTacToe* as follows. *TicTacToeRec*(i, j) is initially called with $i = 0$ and $j = n$ ($k = 0$). The augmented board *Board* is initialized to "E."

```
procedure TicTacToeRec(i, j) recursive
Input:   i, j   (integers between 1 and n, inclusive, called initially with i = 0 and j = n)
         B[−1:n + 2, −1:n + 2]          (global array corresponding to board
              configuration, initialized to "E," and B[1, 1], ... , B[i, j] filled with Xs and
              Os with no three in a row)
Output:  all extensions of B[1, 1], ... , B[i, j] to goal states; that is, board configurations
         not containing three in a row in either Xs or Os
```

```
      Next(i, j)                                   //k = k + 1
      for Child ← 1 to 2 do
         if Child = 1 then
            B[i, j] ← 'X'
         else
            B[i, j] ← 'O'
         endif
         if .not. BoundedBoard(i, j) then
            if (i = n) .and. (j = n) then
               PrintBoard(Board[1:n, 1:n])         //print goal state
            else
               TicTacToeRec(i, j)
            endif
         endif
      endfor
   end TicTacToeRec
```

10.2.5 Solving Optimization Problems Using Backtracking

Backtracking is frequently used to solve optimization problems—that is, to optimize (maximize or minimize) an objective function f over all goal states for a given problem. For example, for a sum of subsets problem interpreted as a coin-changing problem, we want to make correct change using the fewest coins (the objective function f is the number of coins). To do so, we use the following generic backtracking paradigm for solving the problem of minimizing the objective function (the paradigm is easily altered to solve maximization problems). Given an objective function f, let f^* denote the minimum of f over all solution states. A solution state X such that $f(X) = f^*$ is a goal state. Note that for *any* solution state $X = (x_1, \ldots, x_k)$, the value $f(X)$ is an upper bound for f^*. We maintain a variable UB, initialized to infinity. Additionally, at each stage of the backtracking algorithm, we maintain a solution state *CurrentBest* such that $UB = f(Current Best)$ is the minimum value of f over all solution states generated so far. For many problems, we can efficiently compute a function $LowerBound(x_1, \ldots, x_k)$ that is not larger than the value of f on any solution state belonging to the subtree of the state-space tree rooted at $(x_1, \ldots, x_k)$. We can then dynamically bound a problem state $(x_1, \ldots, x_k)$ if $LowerBound(x_1, \ldots, x_k) \geq UB$. For example, in the coin-changing problem modeled on the variable-tuple state-space tree, $LowerBound(x_1, \ldots, x_k) = k$ if $(x_1, \ldots, x_k)$ is a goal state; otherwise, $LowerBound(x_1, \ldots, x_k) = k + 1$.

The generic backtracking paradigm for minimizing an objective function is based on the strategy just outlined. We describe the recursive version *BacktrackMinRec* of the paradigm, and leave the iterative version *BacktrackMin* as an exercise. The following high-level recursive procedure *BacktrackMinRec* is called initially with $k = 0$. During its resolution, *BacktrackMinRec* calls a function *StaticBounded*, which plays the same role as the function *Bounded* used for nonoptimization problems. For example, in the optimization version of the sum of subsets problem with input parameter *Sum*, a problem state is statically bounded if its sum was not smaller than *Sum*, whereas it is dynamically bounded if the cardinality of the subset corresponding to the problem state is at least as large as a previously generated solution state. In general, a problem state is either bounded dynamically ($LowerBound(x_1, \ldots, x_k) \geq UB$), or bounded statically ($Static\ Bounded(x_1, \ldots, x_k) =$ **.true.**).


```
procedure BacktrackMinRec(k, CurrentBest) recursive
Input:   T (implicit state-space tree associated with the given problem)
         D_k (decision set, with D_k = ∅ for k ≥ n)
         f (objective function defined on problem states)
         k (a nonnegative integer, 0 on initial call)
         X[0:n] (global array where X[1:n] maintains the problem states of T, and where
            the problem state (X[1], … , X[k]) has already been generated)
         StaticBounded (a static bounding function on the problem states)
         LowerBound (a function defined on the problem states)
         CurrentBest (solution state extending (X[1], … , X[k]) with the current
            minimum value of f over all descendants of (X[1], … , X[k]) )
         UB (global variable, initialized to ∞)
Output:  CurrentBest (solution state extending (X[1], … , X[k]) with the minimum value
            of f over all descendants of (X[1], … , X[k]) )
   k ← k + 1
   for each X[k] ∈ D_k(X[1], … , X[k − 1]) do
      if (X[1], … , X[k]) is a solution state then
         if f(X[1], … , X[k]) < UB then
            UB ← f(X[1], … , X[k])
            CurrentBest ← (X[1], … , X[k])
         endif
      endif
      if LowerBound(X[1], … , X[k]) < UB .and.
         .not. StaticBounded(X[1], … , X[k]) then
         BacktrackMinRec(k, CurrentBest)
      endif
   endfor
end BacktrackMinRec
```

REMARK

The paradigm *BacktrackMinRec* ends up returning the first optimal goal state that was encountered. (Of course, unlike nonoptimization problems solved using backtracking, the algorithm has no way of checking that it was an optimal goal state until all goal states have been examined or eliminated.) If all optimal goal states are desired, then *CurrentBest* must maintain *all* the goal states that have the current minimum value of f. For example, the sum of subsets problem illustrated in Figure 10.12 has two optimal goal states, represented by the tuples (1, 2) and (2, 4). Procedure *SumOfSubsetsMinRec* only outputs the goal state (1, 2).

The algorithm *BacktrackMinRec* is easily altered to apply to optimization problems where we wish to maximize the objective function f. For such problems, LB is the current maximum value of f, and $UpperBound(x_1, \ldots, x_k)$ is an upper bound for the maximum value of f over all solution states in the subtree of the state-space tree rooted at $(x_1, \ldots, x_k)$. Alternatively, a problem involving maximizing an objective function f can be canonically transformed into an equivalent problem of minimizing the associated objective function $g = M - f$, where M is a suitable constant. By replacing f by $M - f$ in a maximization problem, *BacktrackMinRec* can be applied directly to solve both minimization and maximization problems. Of course, M can be taken as zero, but for a given problem a nonzero value of M might yield a natural interpretation for g. For example, for the 0/1 knapsack problem, if M is taken as the sum of the values of all of the input objects, then $M - f$ is the sum of the values of the objects left out of the knapsack.

As our first illustration of the use of *BacktrackMinRec*, we show how it can be directly translated into a solution for the optimization version of the sum of subsets problem, where goal states are solution states having minimum cardinality. Thus, UB is the smallest cardinality of a solution state currently generated. Note that a problem state $(x_1, \ldots, x_k)$ corresponding to a subset of cardinality k can be dynamically bounded if k is not smaller than the current value of UB. Interpreting dynamic bounding in terms of the generic paradigm *BacktrackMinRec*, we see that $LowerBound(x_1, \ldots, x_k) = k + 1$ if $x_1 + \cdots + x_k < Sum$; otherwise, $LowerBound(x_1, \ldots, x_k) = k$.

```
procedure SumOfSubsetsMinRec(k, CurrentBest) recursive
Input:  k (a nonnegative integer, 0 on initial call)
        A[0:n − 1] (global array of positive integers)
        Sum (global positive integer)
        X[0:n] (global array initialized to −1s. It is assumed that X[1], ... , X[k]
        representing a partial solution is already defined) CurrentBest (solution state
        (X[1], ... , X[m]) extending (X[1], ... , X[m]) such that m is minimum over all
        currently examined solution states that are descendants of (X[1], ... , X[k]) )
```



```
        PathSum (global variable = A[X[1]] + ··· + A[X[k]])
        UB (global variable, initialized to ∞)
Output: CurrentBest (solution state (X[1], ... , X[m]) such that m is a minimum over all
        solution states that are descendants of (X[1], ... , X[k]))
    k ← k + 1                          //go one level deeper in state-space tree
    for Child ← X[k − 1] + 1 to n do
        X[k] ← Child
        Temp ← PathSum +A [X[k]]
        if Temp ≥ Sum then             //(X[1], ... , X[k]) is statically bounded
            if Temp = Sum then         //(X[1], ... , X[k]) is a solution state
                if k < UB then
                    UB ← k
                    CurrentBest ← (X[1], ... , X[k])
                endif
            endif
        else                           //(X[1], ... , X[k]) is not statically bounded
            if k < UB then             //(X[1], ... , X[k]) is not dynamically bounded
                PathSum ← Temp
                SumOfSubsetsMinRec(k, CurrentBest)
            endif
        endif
    endfor
end SumOfSubsetsMinRec
```

The portion of the state-space tree generated by procedure *SumOfSubsetsMinRec* is illustrated in Figure 10.12 for a sample set $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$.

SumOfSubsetsMinRec was written as a direct translation of *BacktrackMinRec*. However, since *BacktrackMinRec* is a generic paradigm, it is often the case that additional efficiencies might be possible when adapting it to a specific problem. Indeed, in the variable-tuple implementation of the optimization version of the sum of subsets problem, where a single optimal goal is to be output, when a goal state X is generated, there is no need to generate the remaining siblings of X because the subsets corresponding to these siblings cannot have smaller cardinality than the subset corresponding to X. To implement this improvement in the procedure *CoinChangingRec*, we need only break out of the **for** loop whenever a goal is generated (by simply adding a **break** statement after the assignment updating *CurrentBest*). For example, with this alteration, we would not generate the two siblings of the goal ($X[1] = 1$, $X[2] = 2$) in Figure 10.12.

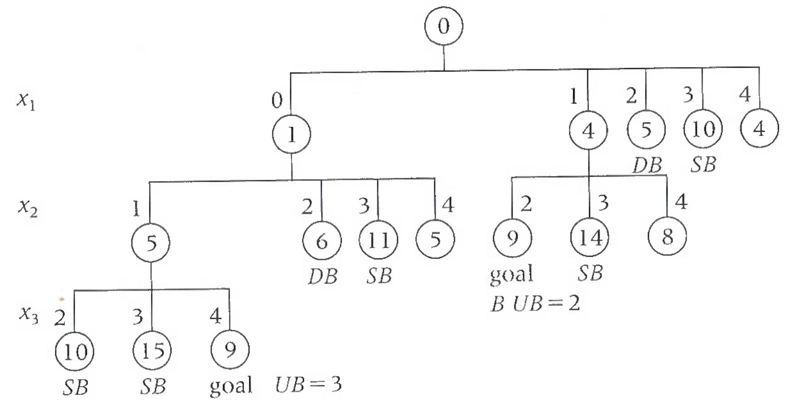


FIGURE 10.12

Portion of the state-space tree generated by procedure *SumOfSubsetsMinRec* for the set $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$. Statically bounded nodes are labeled *SB*. Nodes not statically bounded but dynamically bounded are labeled *DB*. Leaf nodes in the entire state-space tree are unlabeled, unless they are goals. Solution states resulting in updates to *UB* and *CurrentBest* are labeled with the updated value of *UB*. At termination, *SumOfSubsetsMinRec* outputs the final value *CurrentBest* = (1, 2).

Our next example, the 0/1 knapsack problem, illustrates the transformation of maximization problems into minimization problems. In Chapter 7, an efficient greedy algorithm was given for solving the knapsack problem. The greedy method does not necessarily yield an optimal solution to the 0/1 knapsack problem. In fact, there is no known worst-case polynomial algorithm for solving the 0/1 knapsack problem. However, the greedy algorithm for the knapsack problem helps us to define a useful function *LowerBound* for dynamic bounding in the transformed minimization problem.

Since the 0/1 knapsack problem involves looking at subsets of a set of size n, we may solve the problem using backtracking by searching the same state-space tree as in the sum of subsets problem. Consider the variable-tuple state-space tree determined by (10.1.1) (see Figure 10.3). An obvious static bounding function for the problem state $(x_1, \ldots, x_k)$ is given by

$$\text{Static Bounded}(x_1, \ldots, x_k) = \begin{cases} \textbf{.true.} & \text{if } w_{x_1} + \cdots + w_{x_k} \geq C, \\ \textbf{.false.} & \text{otherwise.} \end{cases} \qquad \textbf{(10.2.5)}$$

For $(x_1, \ldots, x_k)$ a problem state, let

$$Value(x_1, \ldots, x_k) = \sum_{i=1}^{k} v_{x_i},$$
$$Weight(x_1, \ldots, x_k) = \sum_{i=1}^{k} w_{x_i}. \qquad \textbf{(10.2.6)}$$

For the 0/1 knapsack problem, the solution states consist of all tuples not statically bounded — that is, all tuples $(x_1, \ldots, x_k)$ such that $Weight(x_1, \ldots, x_k) \leq C$

goal states maximize the objective function *Value* over all solution states. The maximization problem is transformed into a minimization problem by letting

$$M = \sum_{i=0}^{n-1} v_i,$$
$$LeftOutVal(x_1, \ldots, x_k) = M - Value(x_1, \ldots, x_k) \qquad \textbf{(10.2.7)}$$

In other words, $LeftOutVal(x_1, \ldots, x_k)$ is the total value of all the objects left out of the knapsack corresponding to solution state $(x_1, \ldots, x_k)$. In the transformed problem, which we now consider, the objective is to minimize the objective function *LeftOutVal*. To dynamically bound problem states, we maintain a variable *UB*, which at each stage of the backtracking algorithm keeps track of the minimum value of $LeftOutVal(x_1, \ldots, x_k)$ over all the solution states generated so far.

Consider a solution state $(x_1, \ldots, x_k)$ corresponding to the partial filling of the knapsack with the subset of objects $B_k = \{b_{x_1}, \ldots, b_{x_k}\}$. Suppose $(x_1, \ldots, x_k)$ is not bounded by Formula (10.2.6), and let $C' = C - Weight(x_1, \ldots, x_k)$ denote the remaining capacity of the knapsack. Let $LeftOutVal^*(x_1, \ldots, x_k)$ denote the minimum value of *LeftOutVal* over all solution states in the state-space subtree rooted at $(x_1, \ldots, x_k)$. In other words, $LeftOutVal^*(x_1, \ldots, x_k)$ is the smallest value of *LeftOutVal* that can be achieved by placing additional objects in the knapsack from the remaining set of objects $B' = B \backslash B_k$. Clearly, if $LeftOutVal^*(x_1, \ldots, x_k) \geq UB$, then $(x_1, \ldots, x_k)$ can be dynamically bounded. Unfortunately, there is no known efficient method to compute $LeftOutVal^*(x_1, \ldots, x_k)$. In fact, the general problem of computing $LeftOutVal^*(x_1, \ldots, x_k)$ is equivalent to the original 0/1 knapsack problem.

Fortunately, we can efficiently compute a useful lower bound for $LeftOutVal^*(x_1, \ldots, x_k)$ by applying the greedy algorithm *Knapsack*. For B, a given set of objects (with associated values and weights), and C, a given capacity for the knapsack, we let $Greedy(C, B)$ denote the value of the optimal placement of objects in the knapsack, where fractions of objects are permitted (that is, the value of the knapsack generated by *Knapsack*). We define $LowerBound(x_1, \ldots, x_k)$ by

$$LowerBound(x_1, \ldots, x_k) = LeftOutVal(x_1, \ldots, x_k) - Greedy(C', B'). \qquad \textbf{(10.2.8)}$$

Clearly, we have

$$LeftOutVal^*(x_1, \ldots, x_k) \geq LowerBound(x_1, \ldots, x_k).$$

Thus, we can dynamically bound a problem state $(x_1, \ldots, x_k)$ if $LowerBound(x_1, \ldots, x_k) \geq UB$. Figure 10.13 shows the portion of the state-space tree T generated by backtracking for a sample instance of the 0/1knapsack problem.

i	0	1	2	3	4
v_i	16	6	5	12	4
w_i	4	2	2	6	5
v_i/w_i	4	3	2.5	2	.8

$C = 11$, $M = 43$

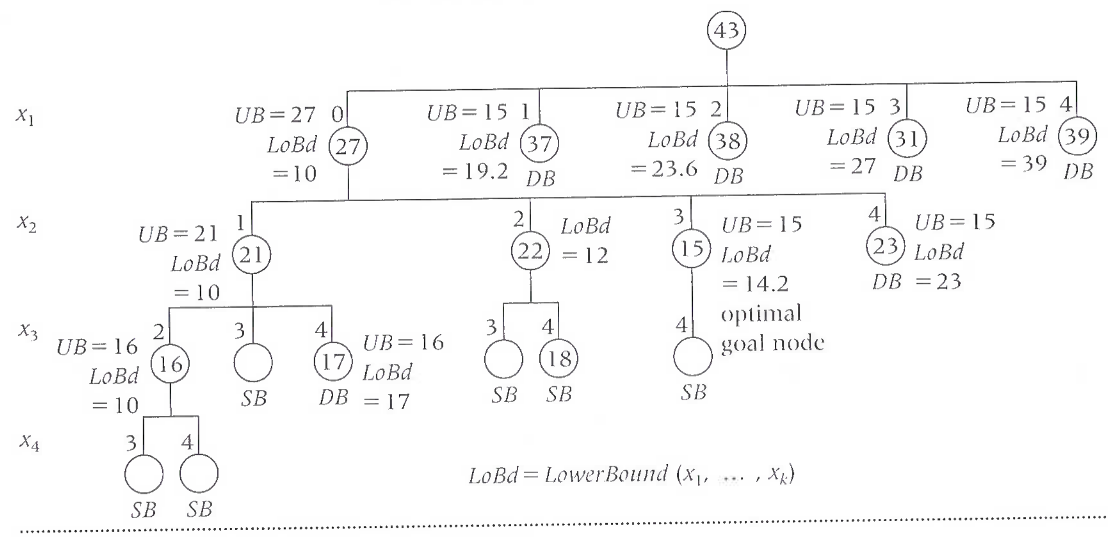


FIGURE 10.13

Portion of the variable-tuple state-space tree T generated by the procedure *BacktrackMin* for a sample input to the 0/1 knapsack problem. Problem states that are statically bounded ($Weight(x_1, \ldots, x_k) \geq C$) are labeled *SB*, whereas problem states not statically bounded but dynamically bounded ($LowerBound(x_1, \ldots, x_k) \geq UB$) are labeled *DB*. $LeftOutVal(x_1, \ldots, x_k)$ is shown inside each solution state. $LowerBound(x_1, \ldots, x_k)$ and the current value of *UB* are shown outside each problem state where *UB* is updated, or where the problem state is dynamically bounded.

10.3 Branch-and-Bound

Cowards die many times before their deaths;
The valiant never taste of death but once.

—Shakespeare, *Julius Caesar*, Act II, Scene II

As with backtracking algorithms, branch-and-bound algorithms are based on searches of an associated state-space tree for goal states. However, in a branch-and-bound algorithm, *all* the children of the *E*-node (the node currently being expanded) are generated before the next *E*-node is chosen. When the children are generated, they become *live* nodes and are stored in a suitable data structure, *LiveNodes*. *LiveNodes* is typically a queue, a stack, or a priority queue. Branch-and-bound algorithms using the latter three data structures are called *FIFO (first in, first out) branch-and-bound*, *LIFO (last in, first out) branch-and-bound*, and *least cost branch-and-bound*, respectively.

Immediately upon expansion, the current *E*-node becomes a *dead* node and a new *E*-node is selected from *LiveNodes*. Thus, branch-and-bound is quite different from backtracking, where we might backtrack to a given node many times, making it the *E*-node each time until all its children have finally been generated or the algorithm terminates. The nodes of the state-space tree at any given point in a branch-and-bound algorithm are therefore in one of the following four states: *E-node, live node, dead node,* or *not yet generated*.

As with backtracking, the efficiency of branch-and-bound depends on the utilization of good bounding functions. Such functions are used in attempting to determine solutions by restricting attention to small portions of the entire state-space tree. When expanding a given *E*-node, a child can be bounded if it can be shown that it cannot lead to a goal node.

We illustrate branch-and-bound by revisiting the sum of subsets problem, where the data structure *LiveNodes* is a queue. Such a branch-and-bound, called FIFO branch-and-bound, involves performing a breadth-first search of the state-space tree. Initially the queue of live nodes is empty. The algorithm begins by generating the root node of the state-space tree and enqueuing it in the queue *LiveNodes*. At each stage of the algorithm, a node is dequeued from *LiveNodes* to become the new *E*-node. All the children of the *E*-node are then generated. The children that are not bounded are enqueued (as they are generated from left to right). If only one goal state is desired, then the algorithm terminates after the first goal state is found. Otherwise, the algorithm terminates when *LiveNodes* is empty. Because of the nature of FIFO branch-and-bound, the first goal state found for the sum of subsets problems automatically has the smallest cardinality; that is, it solves the coin-changing problem.

Figure 10.14 illustrates FIFO branch-and-bound for the sum of subsets problem for the instance $A = (1, 11, 6, 2, 6, 8, 5)$ and $Sum = 10$. The action of the queue *LiveNodes* and the portion of the state-space tree generated in reaching the first goal state are given. For this instance of the sum of subsets problem, FIFO branch-and-bound generates fewer nodes of the state-space tree before reaching a goal state than are generated by backtracking.

Figure 10.15 illustrates LIFO branch-and-bound, where *LiveNodes* is a stack, for the same instance of the sum of subsets problem given in Figure 10.14. LIFO branch-and-bound is similar to backtracking, except that a move is made to the rightmost child of a node first instead of the leftmost. However, unlike backtracking, all the children of a node are generated before moving on.

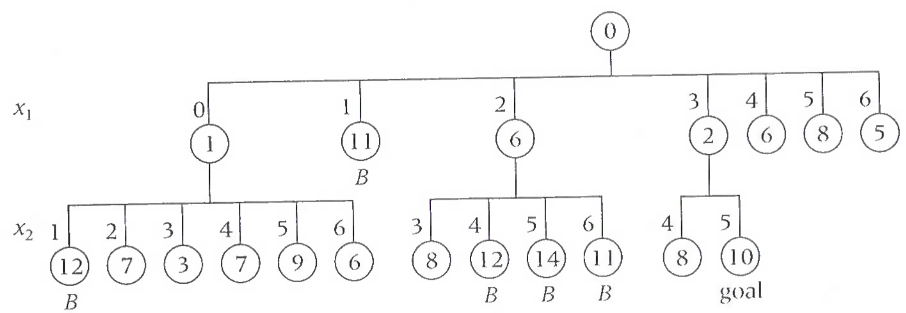
LIFO branch-and-bound generates fewer nodes than does FIFO branch-and-bound for the input considered in Figure 10.14, but for other inputs, the opposite can be true. In general, since FIFO branch-and-bound is based on a breadth-first search of the state-space tree, it is more efficient than LIFO branch-and-bound when goal nodes are not very deep in the state-space tree.

FIGURE 10.14

Action of queue *LiveNodes* and a portion of the variable-tuple state-space tree generated by FIFO branch-and-bound for the sum of subsets problem with $A = \{1, 11, 6, 2, 6, 8, 5\}$ and $Sum = 10$. The sum of the elements chosen is shown inside each node.

queue *LiveNodes*

generate ()	enqueue ()
dequeue	*E*-node = ()
generate (0)	enqueue (0)
generate (1)	bounded
generate (2)	enqueue (2)
generate (3)	enqueue (3)
generate (4)	enqueue (4)
generate (5)	enqueue (5)
generate (6)	enqueue (6)
dequeue	*E*-node = (0)
generate (0,1)	bounded
generate (0,2)	enqueue (0,2)
generate (0,3)	enqueue (0,3)
generate (0,4)	enqueue (0,4)
generate (0,5)	enqueue (0,5)
generate (0,6)	enqueue (0,6)
dequeue	*E*-node = (2)
generate (2,3)	enqueue (2,3)
generate (2,4)	bounded
generate (2,5)	bounded
generate (2,6)	bounded
dequeue	*E*-node = (3)
generate (3,4)	enqueue (3,4)
generate (3,5)	goal node



10.3.1 General Branch-and-Bound Paradigm

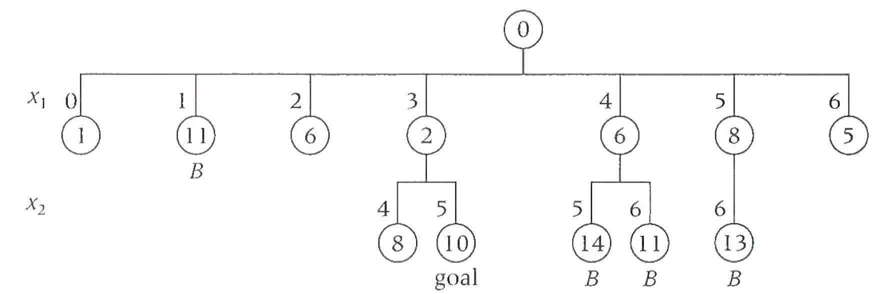
When we use backtracking, we do not explicitly implement the state-space tree. However, in branch-and-bound algorithms, we must explicitly implement the state-space tree and maintain the data structure storing the live nodes. In the general branch-and-bound paradigm *BranchAndBound*, the state-space tree *T* is implemented using the parent representation.

FIGURE 10.15

Action of queue *LiveNodes* and a portion of the variable-tuple state-space tree generated by LIFO branch-and-bound for the sum of subsets problem with $A = \{1, 11, 6, 2, 6, 8, 5\}$ and $Sum = 10$. The sum of the elements chosen is shown inside each node.

stack *LiveNodes*

generate ()	push ()
pop	*E*-node := ()
generate (0)	push (0)
generate (1)	bounded
generate (2)	push (2)
generate (3)	push (3)
generate (4)	push (4)
generate (5)	push (5)
generate (6)	push (6)
pop	*E*-node := (6)
pop	*E*-node := (5)
generate (5,6)	bounded
pop	*E*-node := 4
generate (4,5)	bounded
generate (4,6)	bounded
pop	*E*-node := (3)
generate (3,4)	push (3,4)
generate (3,5)	goal node



The nodes of *T* that are generated by paradigm *BranchAndBound* are represented as follows:

TreeNode = **record**
  Info: *InfoType*
  Parent: →*TreeNode*
end

Only the value x_k need be stored in the information field *Info* of the node *N* corresponding to problem state $(x_1, \ldots, x_k)$. Given a pointer *PtrNode* to *N*, the entire tuple $(x_1, \ldots, x_k)$ is recovered by following the path in *T* from *N* to the root. For convenience, we denote $D_k(x_1, \ldots, x_{k-1})$ by $D_k(PtrNode)$, where *PtrNode* is a pointer to the problem state $(x_1, \ldots, x_{k-1})$.

The next *E*-node is chosen from the elements of *LiveNodes* by calling the procedure *Select*(*LiveNodes*, *E-node*, *k*), where *E-node* is a pointer to the *E*-node and *k* is the size of the *E*-node. The definition of procedure *Select* is dependent on the type of branch-and-bound being implemented. For example, *Select* may choose the next *E*-node from a queue *LiveNodes* (FIFO branch-and-bound), a stack *LiveNodes* (LIFO branch-and-bound), a priority queue *LiveNodes* (least cost branch-and-bound), and so forth.

Paradigm *BranchAndBound* adds a node to *LiveNodes*, by calling the procedure *Add*(*LiveNodes*, *PtrNode*). *BranchAndBound* also invokes the Boolean functions *Answer*(*PtrNode*) and *Bound*(*PtrNode*). *Answer*(*PtrNode*) assumes the value **.true.** if the node pointed to by *PtrNode* is a goal state. *Bound*(*PtrNode*) returns the value **.true.** if the node pointed to by *PtrNode* is bounded. Similar to backtracking, the definition of the bounding function depends on the particular problem being solved.

```
procedure BranchAndBound
Input:   function D_k(x_1, ... , x_{k-1}) determining state-space tree T associated with the
           given problem)
         Bounding function Bounded
Output:  All goal states to the given problem
   LiveNodes is initialized to be empty
   AllocateTreeNode(Root)
   Root→Parent ← null
   Add(LiveNodes, Root)                    //add root to list of live nodes
   while LiveNodes is not empty do
      Select(LiveNodes, E-node, k)         //select next E-node from live nodes
      for each X[k] ∈ D_k(E-node) do       //for each child of the E-node do
         AllocateTreeNode(Child)
         Child→Info ← X[k]
         Child→Parent ← E-node
         if Answer(Child) then             //if child is a goal node then
            Path(Child)                    //output path from child to root
         endif
         if .not. Bounded(Child) then
            Add(LiveNodes, Child)          //add child to list of live nodes
         endif
      endfor
   endwhile
end BranchAndBound
```

As with the general paradigm *Backtrack* there is a corresponding version of the general paradigm *BranchAndBound* for problems involving minimizing or maximizing objective functions. For example, the paradigm *BranchAndBoundMin* maintains in *CurrentBest* the solution state with the current minimum value of the objective function *f*, and the value *f*(*CurrentBest*) is used to dynamically bound nodes. Again, this dynamic bounding is done using a suitable function, *LowerBound*, whose value at a given node X is a lower-bound estimate of the value of the objective function at all goal states in the subtree rooted at X. A node X can be (dynamically) bounded if $f(CurrentBest) \leq LowerBound(X)$. As with the 0/1 knapsack problem discussed earlier, the function $LowerBound(X)$ is often expressed in the form $f(X) + h(X)$, where $h(X)$ is a lower-bound estimate of the smallest incremental increase in f incurred in going from X to a descendant goal state. Sometimes $h(X)$ is a heuristic estimate that might not be provably a lower bound, but which nevertheless has been shown to work well in practice. *LiveNodes* is often maintained as a priority queue, where $LowerBound(X)$ is taken as the priority of a node X. The next *E*-node chosen by *Select* is the node in *LiveNodes* with the least value of *LowerBound*, and the strategy is called *least cost* branch-and-bound. A more detailed discussion of least cost branch-and-bound will be given in the Chapter 23, where it is shown to be a special case of a more general search strategy.

10.4 Closing Remarks

Both LIFO and FIFO branch-and-bound are blind searches of the state-space tree T in the sense that they search the nodes of T in the same order regardless of the input to the algorithm. Thus, they tend to be inefficient for searching the large state-space trees that often arise in practice. Using heuristics can help narrow the scope of otherwise blind searches. The least cost branch-and-bound strategy discussed above uses a heuristic cost function associated with the nodes of the state-space tree T, where the set of live nodes is maintained as a priority queue with respect to this cost function. In this way, the next node to become the *E*-node is the one that is the most promising to lead quickly to a goal.

Least cost branch-and-bound is closely related to the general heuristic search strategy called A*-search. A*-search can be applied to state-space digraphs (digraphs are discussed in Chapter 11), rather than just state-space trees. A*-search is one of the most commonly used search strategies in artificial intelligence. Both A*-search and least cost branch-and-bound are discussed in Chapter 23.

The backtracking and branch-and-bound strategies are well suited to parallelization because different portions of the state-space tree can be assigned to different processors for searching. In Chapter 18, we discuss a general parallel backtracking paradigm in the context of message-passing distributed computing; and in Appendix F, we give code for an MPI implementation for the optimization version of the sum of subsets problem.

References and Suggestions for Further Reading

Golumb, S., and L. Baumert. "Backtracking Programming." *Journal of the ACM* 12 (1965): 516–524. An early general description, and applications, of the backtracking method.

Walker, R. J. "An Enumerative Technique for a Class of Combinatorial Problems." *Proceedings of Symposia in Applied Mathematics*. Vol. X. Providence, RI: American Mathematical Society, 1960. The backtracking and branch-and-bound design strategies have been studied for a long time. (The name backtrack was coined by D. H. Lehmer in the 1950s.) Walker's article is one of the first accounts of the backtracking method.

For two early survey articles on the branch-and-bound paradigm, see:

Lawler, E. L., and D. W. Wood. "Branch-and-Bound Methods: A Survey." *Operations Research* 14 (1966): 699–719.

Mitten, L. "Branch-and-Bound Methods: General Formulation and Properties." *Operations Research* 18 (1970): 24–34.

EXERCISES

Section 10.1 State-Space Trees

10.1 Consider the backtracking solution to the following instance of the 0/1 knapsack problem. The capacity of knapsack = $C = 15$.

i	0	1	2	3	4	5	6
v_i	25	45	12	7	6	10	5
w_i	5	11	3	2	2	7	4

a. Give P_k, and $D_k(x_1, x_2, \dots, x_{k-1})$.

b. Draw the variable-tuple state-space tree (first three levels).

10.2 Repeat Exercise 10.1 for the fixed-tuple state-space tree.

10.3 Show that the number of nodes of both the fixed-tuple and variable-tuple state-space trees for the sum of subsets problem are exponential in n.

Section 10.2 Backtracking

10.4 Modify the pseudocode for the procedure *SumOfSubsets* to use the bounding function given in 10.2.2

10.5 a. Give pseudocode for a nonrecursive backtracking procedure for solving the sum of subsets problem, based on the state-space tree given in Figure 10.4.

b. Repeat part (a) for a recursive backtracking procedure.

10.6 a. Reformulate procedure *Backtrack* so it halts once the first goal is reached.

b. Repeat part (a) for the procedure *BacktrackRec*.

10.7 Give pseudocode for versions of the procedures *Backtrack* and *BacktrackRec* that only check for a goal state after determining that a problem state is bounded. These modified algorithms only apply to problems in which all the goal states are bounded.

10.8 Give pseudocode for a nonrecursive version of the paradigm *BacktrackMin* for minimizing an objective function.

10.9 Show the portion of the state-space tree generated during backtracking for the instance of the 0/1 knapsack problem given in Exercise 10.1 using the bounding functions (10.2.5) and (10.2.8).

10.10 a. Give pseudocode for a backtracking algorithm that solves the 0/1 knapsack problem using the bounding functions (10.2.5) and (10.2.8).

b. Write a program implementing your algorithm in part (a), and run the program for various inputs.

10.11 a. Write a program using backtracking that proves there are no tie boards for the $3 \times 3 \times 3$ tic-tac-toe game, even if we relax the condition that the number of Xs and the number of Os differ by one.

b. Write a program using backtracking that outputs all tie boards for the $3 \times 3 \times 3$ board minus the center position (when playing the game of tic-tac-toe, no X or O is placed in the center position), where we relax the condition that the number of Xs is equal to the number of Os. Is there a tie board where the number of Xs equals the number of Os?

10.12 Two queens in the ordinary chessboard are *nonattacking* if they are not in the same row, column, or diagonal. A classical problem known as the *8-queens problem* is to place eight queens on the board so that each pair of queens is nonattacking. One solution to the 8-queens problem is shown in Figure 10.16.

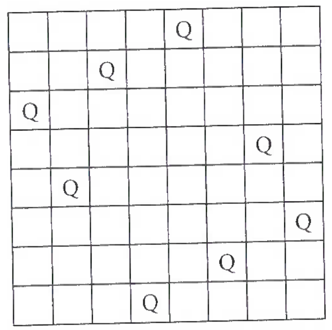


FIGURE 10.16 A solution to the 8-queens problem

The *n-queens problem* is to place n queens on the $n \times n$ chessboard so that each pair of queens is nonattacking.

a. Design a backtracking algorithm that generates all solutions to the n-queens problem.

b. Write a program implementing your algorithm in part (a), and run your program with various values of n.

10.13 Another classical problem associated with chess is the *knight's tour* problem. A knight can make up to eight moves, as shown in Figure 10.17. Starting at an arbitrary position in the $n \times n$ board, a knight's tour is a sequence of $n^2 - 1$ moves such that every square of the board is visited once.

a. Design a backtracking algorithm that either produces a knight's tour or determines that no such tour exits.

b. Write a program implementing your algorithm in part (a), and run your program with various values of n.

■ = possible move

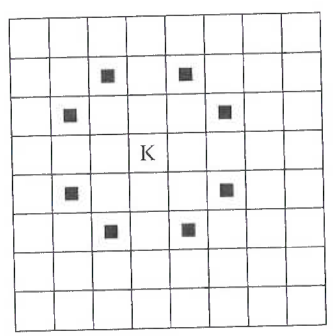


FIGURE 10.17 The eight possible moves for a knight in the given position

10.14 Let $Maze[0{:}n-1, 0{:}n-1]$ be a 0/1 two-dimensional array.

a. Design a backtracking algorithm that either finds a path from $Maze[0, 0]$ to $Maze[n-1, n-1]$ or determines that no such path exists. Adjacent vertices in the path correspond to adjacent cells in the matrix. You are not allowed to move to a cell that contains a 1.

b. Write a program implementing your algorithm in part (a), and run your program with various values of n.

10.15 Write a program that uses backtracking to solve the game of Hi-Q. Hi-Q is a popular game that can be found in many toy stores. Thirty-two pieces are arranged on a board as shown in Figure 10.18, with the center position left empty. The goal is to remove all the pieces but one by jumping and have the last piece end up in the middle position. A piece is allowed to jump a neighbor in either a horizontal or vertical direction (diagonal jumps are not permitted). When a piece is jumped, it is removed from the board. Output the 32 board configurations showing the solution: the initial board configuration and the board configuration after each jump is performed.

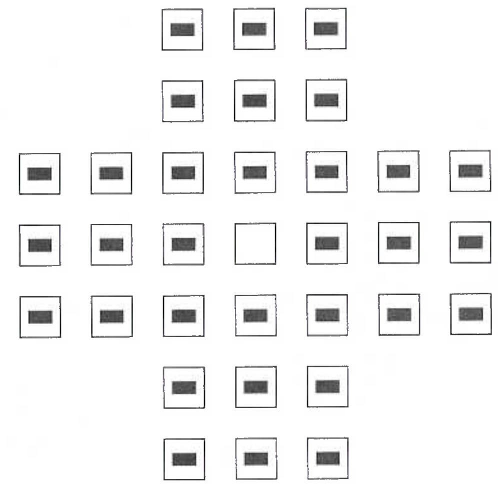


FIGURE 10.18 Initial board configuration for Hi-Q.

Section 10.3 Branch-and-Bound

10.16 Consider the optimization version of the sum of subsets problem for instance $\{a_0, \ldots, a_6\} = \{1, 11, 6, 2, 6, 8, 5\}$ and $Sum = 10$. Show that for this instance of the sum of subsets problem, FIFO branch-and-bound generates fewer nodes of the state-space tree before reaching a goal state than backtracking does.

10.17 Give pseudocode for a version of the general procedure *BranchAndBound* that terminates as soon as a goal is found.

10.18 Give pseudocode for the procedure *Path*(*PtrNode*).

10.19 Draw that portion of the variable-tuple state-space tree generated by FIFO branch-and-bound for the 0/1 knapsack problem given by the following chart. Label the nodes with appropriate values of *UB*, *LoBd*, *SB*, *DB*, and indicate the optimal goal node, as in Figure 10.13. Trace the action of the queue *LiveNodes*, as illustrated in Figure 10.14.

i	0	1	2	3	4	Capacity $C = 12$
v_i	28	15	18	5.5	1	
w_i	8	5	6	4	1	

10.20 Repeat Exercise 10.19 for least cost branch-and-bound.

10.21 Repeat Exercise 10.19 using the fixed-tuple state-space tree.

10.22 Repeat Exercise 10.20 using the fixed-tuple state-space tree.

10.23 Given a set of Boolean variables $y_1, y_2, \ldots, y_m$, a CNF expression involving these variables is a conjunction of the form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$, where each C_i is a disjunction of clauses of the form $z_{i,1} \vee z_{i,2} \vee \cdots \vee z_{i,n(i)}$, and where each $z_{i,j}$ (called a *literal*) is one of the Boolean variables $y_1, y_2, \ldots, y_m$ or its negation. The CNF SAT problem is to determine for a given CNF expression whether or not there is a truth assignment to the Boolean variables for which the CNF expression evaluates to **.true.** (that is, is *satisfied*). For a positive integer k, a k-CNF expression has the property that each clause contains exactly k literals. It turns out that the 2-CNF SAT problem has a polynomial solution (see Chapter 26), whereas the 3-CNF problem is already NP-complete. The best-known solutions to the CNF SAT problem involve fixed-tuple dynamic state-space trees and backtracking, together with such things as using clever heuristics to bound the search. In this exercise, we assume that each clause in a CNF expression is input as a string of integers, with positive integer i meaning that x_i occurs in the clause, and negative integer i meaning that the negative of x_i occurs in the clause. For example, 2, -5, 10 would represent the clause $y_2 \vee \overline{y_5} \vee y_{10}$.

a. Write a program that accepts a CNF expression as input and uses backtracking on a fixed-tuple static state-space tree to determine whether or not the CNF expression is satisfiable. The left (right) child of a node at level $k - 1$ corresponds to assigning **.true.** (**.false.**) to y_k, $k = 1, \ldots, n$.

b. Repeat part (a), but now use a dynamic state-space tree, where the k^{th} decision (level) in the tree is to give a truth assignment to the Boolean variable that occurs (either positively or negatively) k^{th} most often in the CNF expression (ties are decided using subscript ordering).

c. Repeat part (b), but now the decision at a given node in the tree is to assign a truth value to the variable that occurs most often in the clauses that have not already been satisfied by the previous assignments.

d. Run the programs written in parts (a), (b), and (c) with various input CNF expressions, and compare the results.

PART **THREE**

GRAPH AND NETWORK ALGORITHMS