CHAPTER 21

BACKTRACKING

BIRD'S-EYE VIEW

A surefire way to find the answer to a problem is to make a list of all candidate answers, examine each, and following the examination of all or some of the candidates, declare the identified answer. In theory, this approach should work whenever the candidate list is finite and when it is possible to identify the answer following the examination of all or some of the candidates. In practice, the approach isn't very useful because the number of candidates is often much too large (say, exponential, or even factorial in the instance size). As a result, even the fastest computers are able to complete the examination of the candidates in reasonable time only when the instance size is quite small.

Backtracking and branch and bound are two ways to make a systematic examination of the candidate list. Such a systematic examination of the candidate list often results in significant run-time savings in both the worst and expected cases. In fact, these methods often enable us to eliminate the explicit examination of a large subset of the candidates while still guaranteeing that the answer will be found if the algorithm is run to termination. As a result, these methods are often able to obtain solutions to large instances.

This chapter focuses on the backtracking method. This method is used to obtain algorithms for the container-loading, knapsack, max-clique, traveling-salesperson, and board-permutation problems. Each of these applications is an NP-hard problem. When we need an optimal solution (versus a solution that is close to optimal) for an NP-hard problem, a systematic search of the candidate list using either backtracking or branch and bound often results in the best algorithm.

21.1 THE METHOD

Backtracking is a systematic way to search for the solution to a problem. The solution provided in Section 9.5.6 for the rat-in-a-maze problem utilized this technique. In backtracking we begin by defining a **solution space** for the problem. This space must include at least one (optimal) solution to the problem. In the case of the rat-in-a-maze problem, we may define the solution space to consist of all paths from the entrance to the exit. For the case of the 0/1 knapsack problem (see Sections 18.3.2 and 20.2.1) with n objects, a reasonable choice for the solution space is the set of 2^n 0/1 vectors of size n. This set represents all possible ways to assign the values 0 and 1 to x. When $n = 3$, the solution space is {(0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1)}.

The next step is to organize the solution space so that it can be searched easily. The typical organization is either a graph or a tree. Figure 21.1 shows a graph organization for a 3 × 3 maze. All paths from the vertex labeled (1,1) to the vertex labeled (3,3) define an element of the solution space for a 3 × 3 maze. Depending on the placement of obstacles, some of these paths may be infeasible.

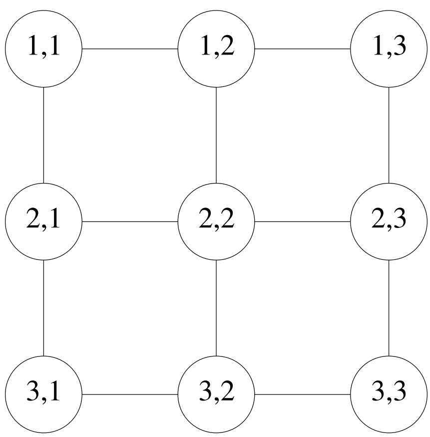

Figure 21.1 Solution space for a 3 × 3 maze

A tree organization for the three-object 0/1 knapsack solution space appears in Figure 21.2. The label on an edge from a level i node to a level $i + 1$ node gives the value of x_i. All paths from the root to a leaf define an element of the solution space. The path from the root to leaf H defines the solution $x = [1,1,1]$. Depending on the values w and c, some or all of the root-to-leaf paths may define infeasible solutions.

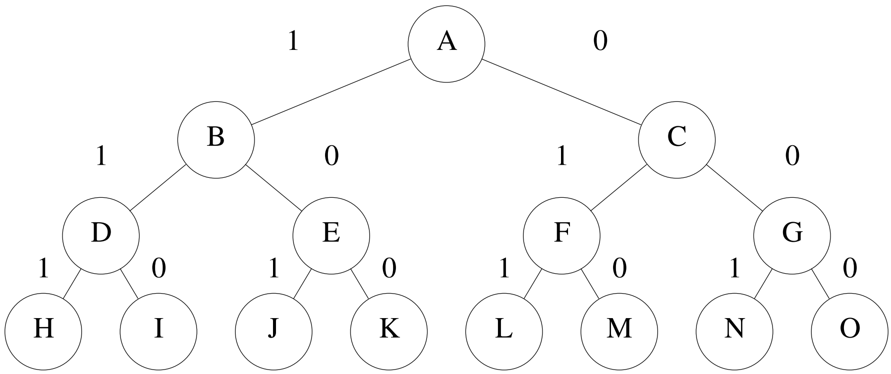

Figure 21.2 Solution space for a three-object knapsack

Once we have defined an organization for the solution space, this space is searched in a depth-first manner beginning at a start node—the entrance node (1,1) in the rat-in-a-maze problem or the root node in the case of the 0/1 knapsack problem. This start node is both a **live** node and the **E-node** (expansion node). From this E-node, we try to move to a new node. If we can move to a new node from the current E-node, then we do so. The new node becomes a live node and also becomes the new E-node. The old E-node remains a live node. If we cannot move to a new node, the current E-node dies (i.e., it is no longer a live node) and we move back (i.e., backtrack) to the most recently seen live node that remains. This live node becomes the new E-node. The search terminates when we have found the answer or when we run out of live nodes to back up to.

Example 21.1 [Rat in a Maze] Consider the 3×3 rat-in-a-maze instance given by the matrix of Figure 21.3(a). We will search this maze using the solution space graph of Figure 21.1.

$$\begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{matrix} \qquad \begin{matrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{matrix} \qquad \begin{matrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{matrix}$$

(a) (b) (c)

Figure 21.3 Mazes

Every path from the entrance of the maze to the exit corresponds to a path from vertex (1,1) to vertex (3,3) in the graph of Figure 21.1. However, some of the

(1,1) to (3,3) paths in this graph do not correspond to entrance-to-exit paths in the example.

The search begins at position (1,1), which is the only live node at this time. It is also the E-node. To avoid going through this position again, we set $maze(1, 1)$ to 1. From this position we can move to either (1,2) or (2,1). For the particular instance we are dealing with, both moves are feasible, as the maze has a 0 at each position. Suppose we choose to move to (1,2). $maze(1, 2)$ is set to 1 to avoid going through here again. The status of $maze$ is as in Figure 21.3(b). At this time we have two live nodes: (1,1) and (1,2). (1,2) becomes the E-node. From the current E-node three moves are possible in the graph of Figure 21.1. Two of these moves are infeasible as the maze has a 1 in these positions. The only feasible move is to (1,3). We move to this position and set $maze(1, 3)$ to 1 to avoid going through here again. The maze of Figure 21.3(c) is obtained, and (1,3) becomes the E-node. The graph of Figure 21.1 indicates two possible moves from the new E-node. Neither of these moves is feasible; so the E-node (1,3), dies and we back up to the most recently seen live node, which is (1,2). No feasible moves from here remain, and this node also dies. The only remaining live node is (1,1). This node becomes the E-node again, and we have an untried move that gets us to position (2,1). The live nodes now are (1,1) and (2,1). Continuing in this way, we reach position (3,3). At this time the list of live nodes is (1,1), (2,1), (3,1), (3,2), (3,3). This list also gives the path to the exit.

Program 9.15 is a backtracking algorithm to find a path in a maze. ■

Example 21.2 [0/1 Knapsack] Consider the knapsack instance $n = 3$, $w = [20, 15, 15]$, $p = [40, 25, 25]$, and $c = 30$. We search the tree of Figure 21.2, beginning at the root. The root is the only live node at this time. It is also the E-node. From here we can move to either B or C. Suppose we move to B. The live nodes now are A and B. B is the current E-node. At node B the remaining capacity r is 10, and the profit earned cp is 40. From B we can move to either D or E. The move to D is infeasible, as the capacity needed to move there is $w_2 = 15$. The move to E is feasible, as no capacity is used in this move. E becomes the new E-node. The live nodes at this time are A, B, and E. At node E, $r = 10$ and $cp = 40$. From E we have two possible moves (i.e., to nodes J and K). The move to node J is infeasible, while that to K is not. Node K becomes the new E-node. Since K is a leaf, we have a feasible solution. This solution has profit value $cp = 40$. The values of x are determined by the path from the root to K. This path (A, B, E, K) is also the live-node sequence at this time. Since we cannot expand K further, this node dies and we back up to E. Since we cannot expand E further, it dies too.

Next we back up to B, which also dies, and A becomes the E-node again. It can be expanded further, and node C is reached. Now $r = 30$ and $cp = 0$. From C we can move to either F or G. Suppose we move to F. F becomes the new E-node, and the live nodes are A, C, and F. At F, $r = 15$ and $cp = 25$. From F we can move to either L or M. Suppose we move to L. Now $r = 0$ and $cp = 50$. Since L is a leaf and it represents a better feasible solution than the best found so far (i.e., the one at node K), we remember this feasible solution as the best solution. Node L dies, and we back up to node F. Continuing in this way, we search the entire tree. The best solution found during the search is the optimal one. ■

Example 21.3 [Traveling Salesperson] In this problem we are given an n vertex network (either directed or undirected) and are to find a cycle of minimum cost that includes all n vertices. Any cycle that includes all n vertices of a network is called a **tour**. In the **traveling-salesperson problem**, we are to find a least-cost tour.

A four-vertex undirected network appears in Figure 21.4. Some of the tours in this network are 1,2,4,3,1; 1,3,2,4,1; and 1,4,3,2,1. The tours 2,4,3,1,2; 4,3,1,2,4; and 3,1,2,4,3 are the same as the tour 1,2,4,3,1, whereas the tour 1,3,4,2,1 is the reverse of the tour 1,2,4,3,1. The cost of the tour 1,2,4,3,1 is 66; that of 1,3,2,4,1 is 25; and that of 1,4,3,2,1 is 59. 1,3,2,4,1 is the least-cost tour in the network.

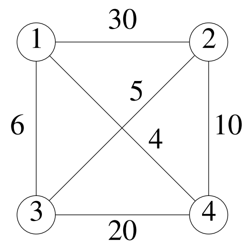

Figure 21.4 A four-vertex network

As the name suggests, the traveling-salesperson problem may be used to model the territory covered by a salesperson. The vertices represent the cities (including the home base) in the salesperson's territory. The edge costs give the travel time (or cost) required to go between two cities. A tour represents the path taken by the salesperson when visiting all cities in his/her territory and then returning home.

We can use the traveling-salesperson problem to model other problems. Suppose we are to drill a number of holes on a sheet of metal or on a printed circuit board. The location of the holes is known. The holes are drilled by a robot drill that begins at its base position, travels to each hole location, drills, and then returns to its base. The total time is that to drill all holes plus the drill travel time. The time to drill all holes is independent of the order in which they are drilled. However, the drill travel time is a function of the length of the tour used by the drill. Therefore, we wish to find a tour of minimum length.

As another example, consider a manufacturing environment in which a particular machine is to be used to manufacture n different items. The items are manufactured repeatedly by using a manufacturing cycle. In one cycle all n items are manufactured in sequence, and then we are ready to begin the next cycle. The next cycle uses the same manufacturing sequence. For example, if the machine is used to paint red, white, and blue cars in this sequence, then following the painting of the blue cars, we begin the sequence again with the red cars. The cost of one cycle includes the actual cost of manufacturing the items plus the cost of changing over from one item to the next. Although the actual cost of manufacturing the items is independent of the sequence in which the items are manufactured, the changeover cost depends on the sequence. To minimize the changeover cost, we may define a directed graph in which the vertices represent the items, and the edge (i, j) has a cost equal to that of changing from the manufacture of item i to that of item j. A minimum-cost tour defines a least-cost manufacturing cycle.

Since a tour is a cycle that includes all vertices, we may pick any vertex as the start (and hence the end). Let us arbitrarily select vertex 1 as the start and end vertex. Each tour is then described by the vertex sequence 1, v_2, $\cdots$, v_n, 1 when v_2, $\cdots$, v_n is a permutation of (2, 3, $\cdots$, n). The possible tours may be described by a tree in which each root-to-leaf path defines a tour. Figure 21.5 shows such a tree for the case of a four-vertex network. The edge labels on the path from the root to a leaf define a tour (when 1 is appended). For example, the path to node L represents the tour 1,2,3,4,1, while the path to node O represents the tour 1,3,4,2,1. Every tour in the network is represented by exactly one root-to-leaf path in the tree. As a result, the number of leaves in the tree is $(n-1)!$.

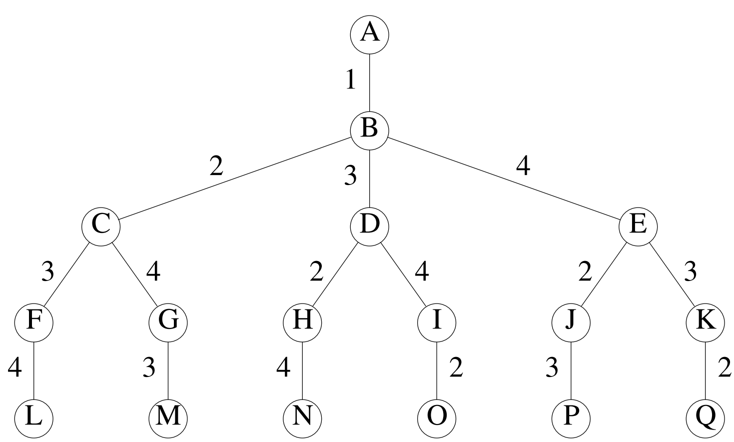

Figure 21.5 Solution space tree for a four-vertex network

A backtracking algorithm will find a minimum-cost tour by searching the solution space tree in a depth-first manner, beginning at the root. A possible search using Figure 21.5 would move from node A to B to C to F to L. At L the tour

1,2,3,4,1 is recorded as the best tour seen so far. Its cost is 59. From L we backtrack to the live node F. As F has no unexamined children, it is killed and we backtrack to node C. C becomes the E-node, and we move forward to G and then to M. We have now constructed the tour 1,2,4,3,1 whose cost is 66. Since this tour isn't superior to the best tour we have, we discard the new tour and backtrack to G, then C, and then B. From B the search moves forward to D and then to H and N. The tour 1,3,2,4,1 defined at N has cost 25 and is better than the previous best tour. We save 1,3,2,4,1 as the best tour seen so far. From N the search backtracks to H and then to D. At D we can again move forward. We reach node O. Continuing in this way, we search the entire tree; 1,3,2,4,1 is the least-cost tour. ■

When the problem we are to solve asks for a subset of n elements that optimizes some function, the solution space tree is called a **subset tree**. So the solution space tree for an n-object 0/1 knapsack instance is a subset tree. Such a tree has 2^n leaf nodes and $2^{n+1} - 1$ total nodes. As a result, every algorithm that moves through all nodes in the tree must spend $\Omega(2^n)$ time. When the problem asks for an n-element permutation that optimizes some function, the solution space tree is a **permutation tree**. Such a tree has $n!$ leaves, and so every algorithm that moves through all nodes of the tree must spend $\Omega(n!)$ time. Note that the tree of Figure 21.5 is for the case when we are looking for the best permutation of the vertices $\{2, 3, 4\}$. Vertex 1 is the first and last vertex of the tour.

We can speed the search for an optimal solution by determining whether or not a newly reached node can possibly lead to a solution better than the best found so far. If it cannot, then there is no point moving into any of its subtrees and the node may be immediately killed. Strategies that are used to kill live nodes are called **bounding functions**. In Example 21.2 we used the following bounding function: Kill nodes that represent infeasible solutions. For the traveling-salesperson problem, we could use this bounding function: If the cost of the partial tour built so far isn't less than that of the best tour found to this point, kill the current node. If we use this bounding function on the example of Figure 21.4, then by the time we reach node I of Figure 21.5, we have found the tour 1,3,2,4,1 with cost 25. At node I the partial tour is 1,3,4 whose cost is 26. By completing this partial tour into a full tour, we cannot get a tour with cost less than 25. There is no point in searching the subtree with root I.

Summary

The steps involved in the backtracking method are

1. Define a solution space that includes the answer to the problem instance.
2. Organize this space in a manner suitable for search.
3. Search the space in a depth-first manner using bounding functions to avoid moving into subspaces that cannot possibly lead to the answer.

An interesting feature of backtracking implementations is that the solution space is generated while the search is conducted. At any time during the search, only the path from the start node to the current E-node is saved. As a result, the space needs of a backtracking algorithm are typically O(length of longest path from the start node). This feature is important because the size of the solution space organization is usually exponential or factorial in length of the longest path. So the solution space organization needs excessive memory if stored in its entirety.

EXERCISES

1. Consider the 0/1 knapsack instance: $n = 4$, $w = [20, 25, 15, 35]$, $p = [40, 49, 25, 60]$, and $c = 62$.
   (a) Draw the solution space tree for 0/1 knapsack instances with $n = 4$.
   (b) Trace the working of a backtracking algorithm on this tree (use the ps, ws, and c values given in this exercise). Clearly label the nodes in the order in which the backtrack algorithm first reaches them. Identify the nodes that do not get reached.
2. (a) Draw the solution space tree for traveling-salesperson instances with $n = 5$.
   (b) Trace the working of a backtracking algorithm on this tree (use the instance of Figure 21.6). Clearly label the nodes in the order in which the backtrack algorithm first reaches them. Identify the nodes that do not get reached.

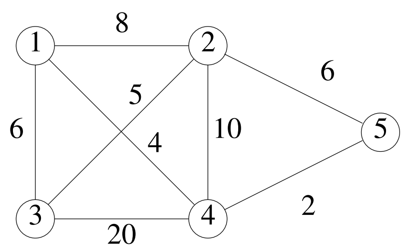

Figure 21.6 Instance for Exercise 2.

3. Mary and Joe practice tennis together every Saturday. They begin with a basket of 120 balls each and continue until both baskets are empty. Then they need to pick up 240 balls from around the tennis court. Mary and Joe pick up the balls by retrieving the empty baskets, filling them with balls, and returning the full baskets to their original positions. Mary picks up the balls

on her side of the net, while Joe picks up the remaining balls. Describe how the traveling-salesperson problem can help Mary and Joe determine the order in which they should pick up balls so that they walk the minimum distance.

21.2 APPLICATIONS

21.2.1 Container Loading

The Problem

In Section 18.3.1 we considered the problem of loading a ship with the maximum number of containers. Now we will consider a variant of this problem in which we have two ships and n containers. The capacity of the first ship is c_1, and that of the second c_2. w_i is the weight of container i, and $\sum_{i=1}^{n} w_i \le c_1 + c_2$. We wish to determine whether there is a way to load all n containers. In case there is, then such a loading is to be determined.

Example 21.4 When $n = 3$, $c_1 = c_2 = 50$, and $w = [10, 40, 40]$, we can load containers 1 and 2 onto the first ship and container 3 onto the second ship. If the weights are [20, 40, 40], then we cannot load the containers onto the ships. ■

When $\sum_{i=1}^{n} w_i = c_1 + c_2$, the two-ship-loading problem is equivalent to the **sum-of-subset** problem in which we are given n numbers and asked to find a subset (if it exists) that sums to c_1. When $c_1 = c_2$ and $\sum_{i=1}^{n} w_i = 2c_1$, the two-ship-loading problem is equivalent to the **partition problem**. In this latter problem, we are given n numbers a_i, $1 \le i \le n$ and asked to find a subset (if it exists) that sums to $(\sum_{i=1}^{n} a_i)/2$. Both the partition and sum-of-subset problems are NP-hard problems and remain NP-hard even if the instances are limited to integer numbers. So we do not expect to solve the two-ship-loading problem in polynomial time.

You may verify that the following strategy to load the two ships succeeds whenever there is a way to load all n containers: (1) load the first ship as close to its capacity as possible and (2) put the remaining containers into the second ship. To load the first ship as close to capacity as possible, we need to select a subset of containers with total weight as close to c_1 as possible. This selection can be made by solving the 0/1 knapsack problem

$$\text{maximize} \sum_{i=1}^{n} w_i x_i$$

subject to the constraints:

$$\sum_{i=1}^{n} w_i x_i \le c_1 \text{ and } x_i \in \{0, 1\},\ 1 \le i \le n$$

When the weights are integer, we can use the dynamic-programming solution of Section 20.2.1 to determine the best loading of the first ship. The time needed is $O(\min\{c_1, 2^n\})$ with the tuple method. We can use the backtracking method to develop an $O(2^n)$ algorithm that can outperform the dynamic-programming algorithm on some (though not all) instances.

First Backtracking Solution

Since we are to find a subset of the weights with sum as close to c_1 as possible, we use a subset space that may be organized into a binary tree as in Figure 21.2. The solution space will be searched in a depth-first manner for the best solution. We use a bounding function to prevent the expansion of nodes that cannot possibly lead to the answer. If Z is a node on level $j+1$ of the tree, then the path from the root to Z defines values for x_i, $1 \leq i \leq j$. Using these values, define cw (current weight) to be $\sum_{i=1}^{j} w_i x_i$. If $cw > c_1$, then the subtree with root Z cannot contain a feasible solution. We can use this test as our bounding function. (We could augment this test with a test to see whether $cw = c_1$. If it does, then we can terminate the search for the subset with the sum closest to c_1. We do not do this augmentation in our discussion and codes.) Define a node to be **infeasible** iff its cw value exceeds c_1.

Example 21.5 Suppose that $n = 4$, $w = [8, 6, 2, 3]$, and $c_1 = 12$. The solution space tree is the tree of Figure 21.2 with one more level of nodes. The search begins at the root A, and $cw = 0$. If we move to the left child B, then $cw = 8$, which is $\leq c_1 = 12$. The subtree with root B contains a feasible node, and we move to node B. From node B we do not move to node D because $cw + w_2 > c_1$; therefore, no leaves in this subtree represent feasible solutions. Instead, we move directly to node E. This move doesn't change cw. The next move is to node J as the cw value here is 10. The left child of J has a cw value of 13, which exceeds c_1, so the search does not move there. Instead, we make a move to the right child of J, which is a leaf. At this point we have found a subset with weight $cw = 10$. The x_i values are obtained from the path from A to the right child of J. These x_i values are [1, 0, 1, 0].

The backtrack algorithm now backs up to J and then to E. From E we can again move down the tree to node K where $cw = 8$. Its left subtree has cw value 11, and we move there. Since we have reached a leaf, we check whether the cw value exceeds the best found so far. It does, so this leaf represents a better solution than [1, 0, 1, 0]. The path to this leaf has x values [1, 0, 0, 1].

From this leaf we back up to node K. Now we can move to K's right child, which is a leaf with $cw = 8$. This leaf doesn't represent a better solution than the best found so far, so we back up to K, E, B, and A. The root is the first node reached from which we can move down again. The algorithm moves to C and searches this subtree. ■

Program 21.1 is the backtracking algorithm that results when we use the preceding bounding function. The public method `maxLoading` returns the maximum subset sum that is $\leq$ `c`, but does not find the subset that has this weight. We will later

```
public class RecursiveBTLoading1
{
   // class data members
   static int numberOfContainers;
   static int [] weight;
   static int capacity;
   static int weightOfCurrentLoading;
   static int maxWeightSoFar;

   public static int maxLoading(int [] theWeight, int theCapacity)
   {
      // set class data members
      numberOfContainers = theWeight.length - 1;
      weight = theWeight;
      capacity = theCapacity;
      weightOfCurrentLoading = 0;
      maxWeightSoFar = 0;

      // compute weight of max loading
      rLoad(1);
      return maxWeightSoFar;
   }
   /** actual method to find max loading */
   private static void rLoad(int currentLevel)
   {// search from a node at currentLevel
      if (currentLevel > numberOfContainers)
      {// at a leaf
         if (weightOfCurrentLoading > maxWeightSoFar)
            maxWeightSoFar = weightOfCurrentLoading;
         return;
      }
      // not at a leaf, check subtrees
      if (weightOfCurrentLoading + weight[currentLevel] <= capacity)
      {// try left subtree; i.e., x[currentLevel] = 1
         weightOfCurrentLoading += weight[currentLevel];
         rLoad(currentLevel + 1);
         weightOfCurrentLoading -= weight[currentLevel];
      }
      rLoad(currentLevel + 1);  // try right subtree
   }
}
```

Program 21.1 First backtracking code for the loading problem refine the code so as to find this subset. The public method maxLoading sets the class data members and then invokes the private recursive method rLoad. The invocation rLoad(1) actually does the search of the solution space. rLoad(currentLevel) explores the subtree rooted at an implicitly specified node that is at the level currentLevel.

If currentLevel $>$ numberOfContainers, we have reached a leaf node. The solution defined by this leaf has weight weightOfCurrentLoading, which is guaranteed to be $\leq$ capacity, as the search does not move to infeasible nodes. If weightOfCurrentLoading $>$ maxWeightSoFar, the value of the best solution found so far is updated. When currentLevel $\leq$ numberOfContainers, we are at a node Z that has two children. The left child represents the case x[currentLevel] $=$ 1. We can move here only if weightOfCurrentLoading+weight[currentLevel] $\leq$ capacity. When we move to the left child, weightOfCurrentLoading increases by weight[currentLevel] and we reach a level currentLevel+1 node. The subtree of which this node is the root is searched recursively. Upon completion of this search, we return to node Z. To get its weightOfCurrentLoading value, we need to decrease the current weightOfCurrentLoading by weight[currentLevel]. The right subtree of Z has not been searched. Since this subtree represents the case x[currentLevel] $= 0$, the search may move there without a feasibility check because the right child of a feasible node is always feasible.

Notice that the solution space tree is not constructed explicitly by rLoad. The method rLoad spends $\Theta(1)$ time at each node that it reaches. The number of nodes reached is $O(2^n)$, so its complexity is $O(2^n)$. The function also uses $\Theta(n)$ space for the recursion stack.

Second Backtracking Solution

We can improve the expected performance of method rLoad by not moving into right subtrees that cannot possibly contain better solutions than the best found so far. Let Z be a node at level i of the solution space tree. No leaf in the subtree with root Z has weight more than weightOfCurrentLoading+remainingWeight where remainingWeight $= \sum_{j=i+1}^{n}$weight$[j]$ is the weight of the remaining containers (n is the number of containers). Therefore, when

$$\texttt{weightOfCurrentLoading} + \texttt{remainingWeight} \leq \texttt{maxWeightSoFar}$$

there is no need to search the right subtree of Z.

Example 21.6 Let n, w, and c_1 be as in Example 21.5. With the new bounding function, the search proceeds as with the old one until we reach the first leaf (which is the right child of J). maxWeightSoFar is set to 10; we back up to E and then move down to the left child of K where maxWeightSoFar is updated to 11. We do not move to the right child of K, because at this right child weightOfCurrentLoading $= 8$, remainingWeight $= 0$, and weightOfCurrentLoading+remainingWeight $\leq$

`maxWeightSoFar`. Instead, we back up to node A. Again, we do not move to the right child C, because at C `weightOfCurrentLoading` = 0, `remainingWeight` = 11, and `weightOfCurrentLoading+remainingWeight` ≤ `maxWeightSoFar`.

The strengthened bounding function has avoided the search of the right subtree of A as well as the right subtree of K. ■

When we use the strengthened bounding function, we get the code of Program 21.2. This code assumes that a static data member `remainingWeight` has been added to the class. The new class is called `RecursiveBTLoading2`. Notice that the new code does not check whether a reached leaf has more weight than the previous best. Such a check is unneccessary because the strengthened bounding function disallows moves to nodes that cannot yield a better solution. As a result, each leaf that is reached represents a better solution than all previously reached leaves. Although the complexity of the new code remains $O(2^n)$, it is expected to examine fewer nodes than Program 21.1 examines.

Finding the Best Subset

To determine the subset of containers that has weight closest to `capacity`, it is necessary to add code to remember the best subset found so far. To remember this subset, we add the parameter `bestLoading` to `maxLoading`. `bestLoading` is an integer array of zeros and ones so that container i is in the best subset iff `bestLoading`[i] = 1. The new code appears in Programs 21.3 and 21.4.

This code assumes that two static data members `currentLoading` and `bestLoadingSoFar` are added to the class. Both of these data members are one-dimensional arrays of type `int` (Boolean arrays could be used instead). The array `currentLoading` is used to record the path from the search tree root to the current node (i.e., it saves the x_i values on this path), and `bestLoadingSoFar` records the best solution found so far. Whenever a leaf with a better value is reached, `bestLoadingSoFar` is updated to represent the path from the root to this leaf. The ones on this path identify the containers to be loaded. Space for the array `currentLoading` is allocated by `maxLoading`.

Since `bestLoadingSoFar` is updated $O(2^n)$ times, the complexity of `rLoad` is $O(n2^n)$. This complexity can be reduced to $O(2^n)$ with one of the following strategies:

1. First run the code of Program 21.2 to determine the weight of the best loading. Let this weight be `maxWeight`. Then run a modified version of Programs 21.3 and 21.4. The modified version begins with `maxWeightSoFar` = `maxWeight`, enters right subtrees so long as `weightOfCurrentLoading+remainingWeight` ≥ `maxWeightSoFar`, and terminates the first time a leaf is reached (i.e., the first time `currentLevel` > `numberOfContainers`).

2. Modify the code of Program 21.4 so that the path from the root to the best leaf encountered so far is saved incrementally. Specifically, if we are at a

```
public static int maxLoading(int [] theWeight, int theCapacity)
{
   // set class data members
   numberOfContainers = theWeight.length - 1;
   weight = theWeight;
   capacity = theCapacity;
   weightOfCurrentLoading = 0;
   maxWeightSoFar = 0;

   // initialize remainingWeight to sum of all weights
   for (int i = 1; i <= numberOfContainers; i++)
      remainingWeight += weight[i];

   // compute weight of max loading
   rLoad(1);
   return maxWeightSoFar;
}

/** actual method to find max loading */
private static void rLoad(int currentLevel)
{// search from a node at currentLevel
   if (currentLevel > numberOfContainers)
   {// at a leaf
      maxWeightSoFar = weightOfCurrentLoading;
      return;
   }

   // not at a leaf, check subtrees
   remainingWeight -= weight[currentLevel];
   if (weightOfCurrentLoading + weight[currentLevel] <= capacity)
   {// try left subtree
      weightOfCurrentLoading += weight[currentLevel];
      rLoad(currentLevel + 1);
      weightOfCurrentLoading -= weight[currentLevel];
   }
   if (weightOfCurrentLoading + remainingWeight > maxWeightSoFar)
      // try right subtree
      rLoad(currentLevel+1);
   remainingWeight += weight[currentLevel];
}
```

Program 21.2 Refinement of Program 21.1

```
public static int maxLoading(int [] theWeight, int theCapacity,
                             int [] bestLoading)
{
   // set class data members
   numberOfContainers = theWeight.length - 1;
   weight = theWeight;
   capacity = theCapacity;
   weightOfCurrentLoading = 0;
   maxWeightSoFar = 0;
   currentLoading = new int [numberOfContainers + 1];
   bestLoadingSoFar = bestLoading;

   // initialize r to sum of all weights
   for (int i = 1; i <= numberOfContainers; i++)
      remainingWeight += weight[i];

   // compute weight of best loading
   rLoad(1);
   return maxWeightSoFar;
}
```

Program 21.3 Preprocessor for code to report best loading

level i node, then the path to the best leaf is given by `currentLoading`$[j]$, $1 \leq j < i$, and `bestLoadingSoFar`$[j]$, $j \leq i \leq$ `numberOfContainers`. This way, each time the algorithm backs up by one level, one x_i is stored in `bestLoadingSoFar`. Since the number of times the algorithm backs up is $O(2^n)$, the additional cost is $O(2^n)$.

An Improved Iterative Version

The code of Programs 21.3 and 21.4 can be improved to reduce its space requirements. We can eliminate the recursion-stack space, which is $\Theta(n)$, as the array `currentLoading` retains all the information needed to move around in the tree. As illustrated in Example 21.5, from any node in the solution space tree, our algorithm makes a series of left-child moves until no more can be made. Then if a leaf has been reached, the best solution is updated. Otherwise, it tries to move to a right child. When either a leaf is reached or a right-child move is not worthwhile, the algorithm moves back up the tree to a node from which a possibly fruitful right child move can be made. This node has the property that it is the nearest node on the path from the root that has `currentLoading[i]` $= 1$. If a move to the right child is fruitful,

```
/** actual method to find best loading */
private static void rLoad(int currentLevel)
{// search from a node at currentLevel
   if (currentLevel > numberOfContainers)
   {// at a leaf, save better solution
      for (int j = 1; j <= numberOfContainers; j++)
         bestLoadingSoFar[j] = currentLoading[j];
      maxWeightSoFar = weightOfCurrentLoading;
      return;
   }

   // not at a leaf, check subtrees
   remainingWeight -= weight[currentLevel];
   if (weightOfCurrentLoading + weight[currentLevel] <= capacity)
   {// try left subtree
      currentLoading[currentLevel] = 1;
      weightOfCurrentLoading += weight[currentLevel];
      rLoad(currentLevel + 1);
      weightOfCurrentLoading -= weight[currentLevel];
   }
   if (weightOfCurrentLoading + remainingWeight > maxWeightSoFar)
   {
      currentLoading[currentLevel] = 0; // try right subtree
      rLoad(currentLevel + 1);
   }
   remainingWeight += weight[currentLevel];
}
```

Program 21.4 Backtracking code to report best loading

it is made and we again attempt to make a series of left-child moves. If the move to the right child is not fruitful, we back up to the next node with `currentLoading[i] = 1`. This motion of the algorithm through the tree can be coded as an iterative algorithm as in Program 21.5. Unlike the recursive code, this code moves to a right child before checking whether it should. If the move should not have been made, the code backs up. The time complexity of the iterative code is the same as that of Programs 21.3 and 21.4.

```
public static int maxLoading(int [] weight, int capacity,
                             int [] bestLoading)
{
   // initialize for root
   int currentLevel = 1;
   int numberOfContainers = weight.length - 1;
   int [] currentLoading = new int [numberOfContainers + 1];
          // currentLoading[1:i-1] is path to current node
   int maxWeightSoFar = 0;
   int weightOfCurrentLoading = 0;
   int remainingWeight = 0;
   for (int j = 1; j <= numberOfContainers; j++)
      remainingWeight += weight[j];

   // search the tree
   while (true)
   {// move down and left as far as possible
      while (currentLevel <= numberOfContainers &&
             weightOfCurrentLoading + weight[currentLevel] <= capacity)
      {// move to left child
         remainingWeight -= weight[currentLevel];
         weightOfCurrentLoading += weight[currentLevel];
         currentLoading[currentLevel] = 1;
         currentLevel++;
      }
      if (currentLevel > numberOfContainers)
      {// leaf reached
         for (int j = 1; j <= numberOfContainers; j++)
            bestLoading[j] = currentLoading[j];
         maxWeightSoFar = weightOfCurrentLoading;
      }
      else
      {// move to right child
         remainingWeight -= weight[currentLevel];
         currentLoading[currentLevel] = 0;
         currentLevel++;
      }
```

Program 21.5 Iterative loading code (continues)

```
      // back up if necessary
      while (weightOfCurrentLoading + remainingWeight <= maxWeightSoFar)
      {// this subtree does not have a better leaf, back up
         currentLevel--;
         while (currentLevel > 0 && currentLoading[currentLevel] == 0)
         {// back up from a right child
            remainingWeight += weight[currentLevel];
            currentLevel--;
         }
         if (currentLevel == 0) return maxWeightSoFar;

         currentLoading[currentLevel] = 0;   // move to right subtree
         weightOfCurrentLoading -= weight[currentLevel];
         currentLevel++;
      }
   }
}
```

Program 21.5 Iterative loading code (concluded)

21.2.2 0/1 Knapsack Problem

Backtracking Solution

The 0/1 knapsack problem is an NP-hard problem for which we considered greedy heuristics in Section 18.3.2 and developed dynamic-programming algorithms in Section 20.2.1. In this section we develop a backtracking algorithm for this problem. Since we are to select a subset of objects for inclusion into the knapsack such that the profit obtained is maximum, the solution space is organized as a subset tree (Figure 21.2). The backtracking algorithm is very similar to that for the loading problem of Section 21.2.1. As in the development of Section 21.2.1, let us initially develop a recursive algorithm that finds the maximum profit obtainable. Later, this algorithm can be refined to code that finds the subset of objects to be included in the knapsack so as to earn this much profit.

As in the case of Program 21.2, left branches are taken whenever the left child represents a feasible node; right branches are taken when there is a possiblity that the right subtree contains a better solution than the best found so far. A simple way to decide whether or not to move into the right subtree is to check whether the sum of the profit earned at the current node and the profits of the objects yet to be considered exceeds the value of the best solution found so far. If not, the right subtree need not be searched. A more effective way is to order the remaining objects by profit density (p_i/w_i), fill the remaining capacity by putting in objects in decreasing order of density, and use a fraction of the first such object that doesn't fit.

Example 21.7 Consider the instance $n = 4$, $c = 7$, $p = [9, 10, 7, 4]$, and $w = [3, 5, 2, 1]$. The profit densities of these objects are [3, 2, 3.5, 4]. When the knapsack is packed in decreasing order of density, object 4 is packed first, then object 3 is packed, and then object 1. Following the packing of these three objects, the available capacity is 1. This capacity is adequate for 0.2 of object 2. Putting in 0.2 of this object yields a profit of 2. The solution constructed is $x = [1, 0.2, 1, 1]$, and the corresponding profit is 22. Although this solution is infeasible (x_2 is 0.2 while it should be either 0 or 1), its value 22 can be shown to be no less than the best feasible solution. Therefore, we know that the 0/1 knapsack instance has no solution with value more than 22.

The solution space tree is that of Figure 21.2 with one additional level of nodes. When we are at node B of the solution space tree, $x_1 = 1$ and the profit earned so far is $cp = 9$. The capacity used at this node is $cw = 3$. The best additional profit we can earn is by filling the remaining capacity $cleft = c - cw = 4$ in order of denisty. That is, first put in object 4, then object 3, and then 0.2 of object 2. Therefore, the value of the best solution in the subtree A is at most 22.

When we are at node C, $cp = cw = 0$ and $cleft = c = 7$. Loading the remaining capacity by density, objects 4 and 3 are packed and then 0.8 of object 2 is packed. This packing yields a profit of 19. No node in subtree C can yield greater profit.

At node E, $cp = 9$, $cw = 3$, and $cleft = 4$. Only objects 3 and 4 remain to be considered. When these objects are considered in order of density, object 4 is packed first and then object 3. So no node in subtree E has value more than $cp + 4 + 7 = 20$. If we have already found a solution with value 20 or more, there is no point in searching subtree E. ■

This bounding function is easy to implement if the objects are in decreasing order of profit density.

Java Implementation

The backtracking solution for the knapsack problem is developed as the class `RecursiveBTKnapsack`. This class includes a top-level nested class `Element`. The data members of `Element` are `id` (element identifier, an `int`) and `profitDensity`, (a `double`). The class `Element` implements the interface `Comparable` by comparing the `profitDensity` value of instances. Therefore, sorting a collection of `Elements` results in the collection being ordered in ascending order of profit density. The class data members of `RecursiveBTKnapsack` are given in Program 21.6.

The public method `RecursiveBTKnapsack.knapsack` (Program 21.7) returns the value of the best filling for the knapsack. This method first creates an array `q` of type `Element` that contains the profit density of all objects. Next we sort the array `q` into ascending order of density using `mergeSort` (Program 19.3). Then, using this

```
static double capacity;
static int numberOfObjects;
static double [] weight;
static double [] profit;
static double weightOfCurrentPacking;
static double profitFromCurrentPacking;
static double maxProfitSoFar;
```

Program 21.6 Class data members of `RecursiveBTKnapsack`

ordering information, the class data members `profit` and `weight` are set so that `profit[i]/weight[i]` $\geq$ `profit[i-1]/weight[i-1]`. With all this work done, the recursive method `rKnap` (Program 21.8), which implements the backtracking method, is invoked.

Notice the similarity between the method `rknap` and the method `rLoad` (Program 21.2). Program 21.9 gives the code for the bounding function. Notice that Program 21.8 computes the bounding function only for right-child moves. For left-child moves, the bounding function value at the left child is the same as at its parent.

Complexity Analysis

The complexity of `rKnap` is $O(2^n)$, even though the bounding function whose complexity is $O(n)$ is computed at $O(2^n)$ right children. To arrive at this complexity note that if, in the computation of the bounding function, the `while` loop of `profitBound` is entered q times, then this computation of the bounding function is followed by q left-child moves. So over the entire run of `rKnap`, the `while` loop of `profitBound` cannot be entered more times than the number of left-child moves that are made. This number is $O(2^n)$. Therefore, the total time spent computing the bounding function is $O(2^n)$.

21.2.3 Max Clique

Problem Description

A subset U of the vertices of an undirected graph G defines a **complete subgraph** iff for every u and v in U, (u, v) is an edge of G. The **size** of a subgraph is the number of vertices in it. A complete subgraph is a **clique** of G iff it is not contained in a larger complete subgraph of G. A **max clique** is a clique of maximum size.

Example 21.8 In the graph of Figure 21.7(a), the subset {1,2} defines a complete subgraph of size 2. This subgraph is not a clique, as it is contained in a larger

```
public static double knapsack(double [] theProfit, double [] theWeight,
                              double theCapacity)
{
   // set class data members
   capacity = theCapacity;
   numberOfObjects = theProfit.length - 1;
   weightOfCurrentPacking = 0.0;
   profitFromCurrentPacking = 0.0;
   maxProfitSoFar = 0.0;

   // define an Element array for profit densities
   Element [] q  = new Element [numberOfObjects];

   // set up densities in q[0:n-1]
   for (int i = 1; i <= numberOfObjects; i++)
      q[i - 1] = new Element(i, theProfit[i] / theWeight[i]);

   // sort into increasing density order
   MergeSort.mergeSort(q);

   // initialize class members
   profit = new double [numberOfObjects + 1];
   weight = new double [numberOfObjects + 1];
   for (int i = 1; i <= numberOfObjects; i++)
   {// profits and weights in decreasing density order
      profit[i] = theProfit[q[numberOfObjects - i].id];
      weight[i] = theWeight[q[numberOfObjects - i].id];
   }

   rKnap(1);  // compute max profit
   return maxProfitSoFar;
}
```

Program 21.7 The method `RecursiveBTKnapsack.knapsack`

complete subgraph (i.e., the one defined by {1,2,5}). {1,2,5} defines a max clique of the graph. The vertex sets {1,4,5} and {2,3,5} define other max cliques. ■

A subset U of vertices of G defines an empty subgraph iff for every u and v in U, (u, v) is *not* an edge of G. The subset is an **independent set** of G iff the subset is not contained in a larger subset of vertices that also defines an empty subgraph of G. A **max-independent set** is an independent set of maximum size. For any

```
private static void rKnap(int currentLevel)
{// search from a node at currentLevel
   if (currentLevel > numberOfObjects)
   {// at a leaf
      maxProfitSoFar = profitFromCurrentPacking;
      return;
   }

   // not at a leaf, check subtrees
   if (weightOfCurrentPacking + weight[currentLevel] <= capacity)
   {// try left subtree
      weightOfCurrentPacking += weight[currentLevel];
      profitFromCurrentPacking += profit[currentLevel];
      rKnap(currentLevel + 1);
      weightOfCurrentPacking -= weight[currentLevel];
      profitFromCurrentPacking -= profit[currentLevel];
   }

   if (profitBound(currentLevel + 1) > maxProfitSoFar)
      rKnap(currentLevel + 1);   // try right subtree
}
```

Program 21.8 Recursive backtracking method for 0/1 knapsack problem

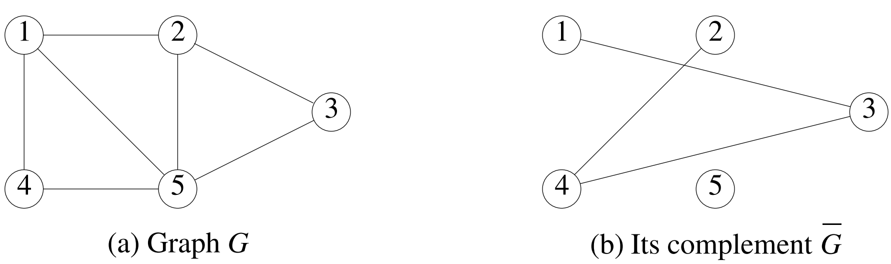

Figure 21.7 A graph and its complement

graph G, its **complement** $\overline{G}$ is a graph that has the same vertex set. Further, (u, v) is an edge of $\overline{G}$ iff (u, v) is not an edge of G.

Example 21.9 The graph of Figure 21.7(b) is the complement of the graph of Figure 21.7(a), and vice versa. {2,4} defines an empty subgraph of Figure 21.7(a)

```
/** bounding function
  * @return upper bound on value of best leaf in subtree */
private static double profitBound(int currentLevel)
{
   double remainingCapacity = capacity - weightOfCurrentPacking;
   double profitBound = profitFromCurrentPacking;
   // fill remaining capacity in order of profit density
   while (currentLevel <= numberOfObjects &&
          weight[currentLevel] <= remainingCapacity)
   {
      remainingCapacity -= weight[currentLevel];
      profitBound += profit[currentLevel];
      currentLevel++;
   }

   // take fraction of next object
   if (currentLevel <= numberOfObjects)
      profitBound += profit[currentLevel] / weight[currentLevel]
                        * remainingCapacity;
   return profitBound;
}
```

Program 21.9 Knapsack bounding function

and is also a max-independent set of this graph. Although {1,2} defines an empty subgraph of Figure 21.7(b), {1,2} is not an independent set because {1,2} is contained in {1,2,5}, which also defines an empty subgraph. {1,2,5} is one of the max-independent sets of Figure 21.7(b). ■

Notice that if U defines a complete subgraph of G, then U also defines an empty subgraph of $\overline{G}$, and vice versa. So there is a correspondence between the cliques of G and the independent sets of $\overline{G}$. In particular, a max clique of G defines a max-independent set of $\overline{G}$.

The **max-clique** problem is to find a max clique of the graph G. Similarly, the **max-independent-set** problem is to find a max-independent set of G. Both problems are NP-hard. We can solve one using an algorithm for the other. For example, if we have an algorithm that solves the max-clique problem, we can solve the max-independent-set problem by first computing the complement of the given graph and then finding a max clique in the complement graph.

Example 21.10 Suppose we have a collection of n animals. We may define a **compatibility graph** G that has n vertices. (u, v) is an edge of G iff animals u and v are compatible. A max clique of G defines a largest subset of mutually compatible animals.

In Section 20.2.5 we considered the problem of finding a maximum size noncrossing subset of nets. We can also formulate this problem as a max-independent-set problem. Define a graph in which each vertex represents a net. There is an edge between two vertices iff the corresponding nets cross. So a max-independent set of the graph corresponds to a maximum size subset of noncrossing nets. When the nets have one endpoint at the top of the routing channel and the other at the bottom, a maximum size subset of noncrossing nets can be found in polynomial time (actually in $\Theta(n^2)$ time) by using dynamic programming. When the endpoints of a net may lie anywhere in the plane, no polynomial-time algorithm to find a maximum size subset of noncrossing nets is known. ■

Backtracking Solution

The max-clique and max-independent-set problems may be solved by $O(n2^n)$-time backtracking algorithms. In both a subset solution space tree (Figure 21.2) may be used. Consider the max-clique problem. The recursive backtracking algorithm is very similar to Program 21.3. When attempting to move to the left child of a level i node Z of the space tree, we need to verify that there is an edge from vertex i to every other vertex, j, for which $x_j = 1$ on the path from the root to Z. When attempting to move to the right child of Z, we need to verify that enough vertices remain so that there is a possibility of finding a larger clique in the right subtree.

Java Implementation

The backtracking algorithm may be implemented as a member of the class `AdjacencyGraph` (Section 17.7) by first adding the `static` members `currentClique` (integer array, used to store path to current node), `maxCliqueFoundSoFar` (integer array, used to save best solution found so far), `sizeOfMaxCliqueFoundSoFar` (number of vertices in `maxCliqueFoundSoFar`), and `sizeOfCurrentClique` (number of vertices in `currentClique`) to this class.

The public method `btMaxClique` (Program 21.10) initializes the necessary class data members and then invokes the private method `rClique` (Program 21.11). The method `rClique` uses the backtracking methodology to do the actual search of the solution space.

```
/** solve max-clique problem using backtracking
  * @return size of max clique
  * @param maxClique set maxClique[i] = 1 iff i is in max clique */
public int btMaxClique(int [] maxClique)
{
   // initialize for rClique
   currentClique = new int [n + 1];
   sizeOfCurrentClique = 0;
   sizeOfMaxCliqueSoFar = 0;
   maxCliqueSoFar = maxClique;

   // find max clique
   rClique(1);
   return sizeOfMaxCliqueSoFar;
}
```

Program 21.10 The method `AdjacencyGraph.btMaxClique`

21.2.4 Traveling Salesperson

Backtracking Solution

The solution space for the traveling-salesperson problem (Example 21.3) is a permutation tree. Such a tree may be searched by using the method `perm` (Program 1.27), which generates all permutations of a list of elements. If we begin with $x = [1, 2, \cdots, n]$, then we can generate the solution space for the n-vertex traveling-salesperson problem by generating all permutations of x_2 through x_n. It is easy to modify `perm` so that it does not generate permutations that have an invalid prefix (i.e., the prefix does not define a path) or a prefix that cannot be completed into better tours than the best found so far. Notice that in a permutation space tree the permutations defined by the leaves in any subtree have the same prefix (see Figure 21.5). Therefore, eliminating certain prefixes from consideration is equivalent to not entering certain subtrees during the search.

Java Implementation

The backtracking algorithm for the traveling-salesperson problem is best implemented as a member of the class `AdjacencyWDigraph` (Program 17.2). Our implementation requires us to add the class data members `partialTour` (an integer array that gives the partial tour to the current node), `bestTourSoFar`, `costOfBestTourSoFar`, and `costOfPartialTour`. As in our other examples, we will have two methods `btSalesperson` and `rTSP`. The former is a public method, and the latter a private method. `btSalesperson` is essentially a preprocessor for `rTSP`, which does a recursive backtrack search in the permutation space tree. The preprocessor

```
private void rClique(int currentLevel)
{// search from a node at currentLevel
   if (currentLevel > n)
   {// at leaf, found a larger clique
    // update maxCliqueSoFar and sizeOfMaxCliqueSoFar
      for (int j = 1; j <= n; j++)
         maxCliqueSoFar[j] = currentClique[j];
      sizeOfMaxCliqueSoFar = sizeOfCurrentClique;
      return;
   }

   // not at leaf; see whether vertex currentLevel
   // is connected to others in current clique
   boolean connected = true;
   for (int j = 1; j < currentLevel; j++)
      if (currentClique[j] == 1 && !a[currentLevel][j])
      {// vertex currentLevel not connected to j
         connected = false;
         break;
      }

   if (connected)
   {// try left subtree
      currentClique[currentLevel] = 1;  // add to clique
      sizeOfCurrentClique++;
      rClique(currentLevel + 1);
      sizeOfCurrentClique--;
   }

   if (sizeOfCurrentClique + n - currentLevel > sizeOfMaxCliqueSoFar)
   {// try right subtree
      currentClique[currentLevel] = 0;
      rClique(currentLevel + 1);
   }
}
```

Program 21.11 Recursive backtrack method for max clique

`btSalesperson` appears in Program 21.12. The invocation `rTSP(2)` searches a tree that contains all permutations of `partialTour[2:n]`.

```
/** traveling salesperson by backtracking
  * @param theZero zero weight
  * @param bestTour bestTour[1:n] is set to best tour
  * @return cost of best tour */
public Object btSalesperson(int [] bestTour, Operable theZero)
{
   // set partialTour to identity permutation
   partialTour = new int [n + 1];
   for (int i = 1; i <= n; i++)
      partialTour[i] = i;

   costOfBestTourSoFar = null;
   bestTourSoFar = bestTour;
   costOfPartialTour = theZero;

   // search permutations of partialTour[2:n]
   rTSP(2);

   return costOfBestTourSoFar;
}
```

Program 21.12 Preprocessor for traveling-salesperson backtracking

Program 21.13 gives the method `rTSP`. The structure of this method is the same as that of method `perm` (Program 1.27). When `currentLevel` equals `n`, we are at the parent of a leaf of the permutation tree and need to first verify that there is an edge from vertex `partialTour[n-1]` to `partialTour[n]`, as well as one from `partialTour[n]` back to the start vertex 1. If both edges exist, we have found a new tour. In this case we need to see whether this tour is the best found so far. If it is, we record the tour and its cost in `bestTourSoFar` and `costOfBestTourSofar`, respectively.

When `currentLevel` < `n`, we move to one of the children of the current node only if there is (1) an edge from `partialTour[currentLevel-1]` to `partialTour[currentLevel]` (if so, `partialTour[1:currentLevel]` defines a path in the network) and (2) the cost of the path `partialTour[1:currentLevel]` is less than the cost of the best tour found so far (if not, the path cannot be completed into a better tour).

```
private void rTSP(int currentLevel)
{// search from a node at currentLevel
   if (currentLevel == n)
   {// at parent of a leaf
      // complete tour by adding last two edges
      if (a[partialTour[n - 1]][partialTour[n]] != null &&
          a[partialTour[n]][1] != null &&
          (costOfBestTourSoFar == null ||
          ((Operable) ((Operable) costOfPartialTour.
                        add(a[partialTour[n - 1]][partialTour[n]])).
                        add(a[partialTour[n]][1])).
                        compareTo(costOfBestTourSoFar) < 0))
      {// better tour found
         for (int j = 1; j <= n; j++)
            bestTourSoFar[j] = partialTour[j];
         costOfBestTourSoFar = ((Operable) costOfPartialTour.
                        add(a[partialTour[n - 1]][partialTour[n]])).
                        add(a[partialTour[n]][1]);
      }
   }
   else
   {// try out subtrees
      for (int j = currentLevel; j <= n; j++)
         // is move to subtree labeled partialTour[j] possible?
         if (a[partialTour[currentLevel - 1]][partialTour[j]] != null &&
             (costOfBestTourSoFar == null ||
              ((Operable) costOfPartialTour.
                add(a[partialTour[currentLevel - 1]][partialTour[j]])).
                compareTo(costOfBestTourSoFar) < 0))
         {// search this subtree
            MyMath.swap(partialTour, currentLevel, j);
            costOfPartialTour.increment(a[partialTour[currentLevel - 1]]
                                         [partialTour[currentLevel]]);
            rTSP(currentLevel + 1);
            costOfPartialTour.decrement(a[partialTour[currentLevel - 1]]
                                         [partialTour[currentLevel]]);
            MyMath.swap(partialTour, currentLevel, j);
         }
   }
}
```

Program 21.13 Recursive backtracking for traveling salesperson

Complexity Analysis

Excluding the cost of updating `bestTourSoFar` whenever a better tour is found, `rTSP` takes $O((n-1)!)$ time. The updating time is $O(n*(n-1)!)$, as $O((n-1)!)$ updates take place and each costs $\Theta(n)$ time. So the overall complexity is $O(n!)$. We can reduce the number of search tree nodes visited by `rTSP` by using stronger conditions for the cost of a path (see Exercise 16).

21.2.5 Board Permutation

Problem Description

The board-permutation problem arises in the design of large electronic systems. The classical form of this problem has n circuit boards that are to be placed into slots in a cage (Figure 21.8). Each permutation of the n boards defines a placement of the boards into the cage. Let $B = \{b_1, \cdots, b_n\}$ denote the n boards. A set $L = \{N_1, \cdots, N_m\}$ of m nets is defined on the boards. Each N_i is a subset of B. These subsets need not be disjoint. Each net is realized by running a wire through the boards that constitute the net.

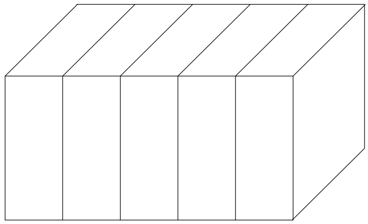

Figure 21.8 Cage with slots

Example 21.11 Let $n = 8$ and $m = 5$. Let the boards and nets be as given below.

$$
\begin{aligned}
B &= \{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8\} \\
L &= \{N_1, N_2, N_3, N_4, N_5\} \\
N_1 &= \{b_4, b_5, b_6\} \\
N_2 &= \{b_2, b_3\} \\
N_3 &= \{b_1, b_3\} \\
N_4 &= \{b_3, b_6\} \\
N_5 &= \{b_7, b_8\}
\end{aligned}
$$

Figure 21.9 shows a possible permutation for the boards. The edges denote the wires that have to be run between the boards. ■

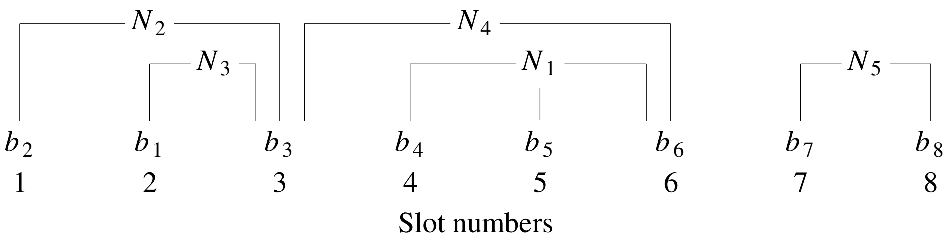

Figure 21.9 Board wiring

Let x denote a board permutation. Board x_i is placed into slot i of the cage when the placement is done using permutation x_i. *density*(x) is the maximum number of wires that cross the gap between any pair of adjacent slots in the cage. For the permutation of Figure 21.9, the *density* is 2. Two wires cross the gaps between slots 2 and 3, slots 4 and 5, and slots 5 and 6. The gap between slots 6 and 7 has no wires, and the remaining gaps have one wire each.

Card cages are designed with a uniform gap size (i.e., the space between adjacent slots is the same). This gap size therefore determines the size of the cage. The gap size itself must be adequate to accommodate the number of wires that must pass through it. Hence the gap size (and in turn the cage size) is determined by *density*(x).

Backtracking Solution

The objective of the board-permutation problem is to find a permutation of the boards that has least *density*. Since this problem is an NP-hard problem, it is unlikely it can be solved by a polynomial-time algorithm, and a search method such as backtracking is an attractive way to solve it. The backtracking algorithm will search a permutation space for the best board permutation.

Java Implementation

We may represent the input as an integer array `board` such that `board[i][j]` is 1 iff net N_j includes board b_i (a Boolean array may be used if we wish to conserve space). Let `boardsWithNet[j]` be the number of boards that include net N_j. For any partial board permutation `partial[1:i]`, let `boardsInPartialWithNet[j]` be the number of boards in `partial[1:i]` that include net N_j. Net N_j crosses the gap between slots `i` and `i+1` iff `boardsInPartialWithNet[j]` > 0 and `boardsInPartialWithNet[j]` $\neq$ `boardsWithNet[j]`. The wire density between slots `i` and `i+1` may be computed by using this test to determine which wires cross the gap between the two slots. The maximum of the wire densities between slots `k` and `k+1` for $1 \leq$ `k` $\leq$ `i` gives the density of the partial permutation.

Our backtracking algorithm for the board-permutation problem is implemented as a class method of the class `RecursiveBTBoard`. Program 21.14 gives the public method `arrangeBoards`, which is essentially a preprocessor for the recursive method `rBoard` (Program 21.15). The method `arrangeBoards` returns the density of the best board arrangement; the best arrangement is returned in the array `bestPermutation`. All variables not explicitly declared are class data members.

The method `arrangeBoards` sets the class data members of `RecursiveBTBoard`. In particular, `boardsWithNet` is initialized so that `boardsWithNet[j]` equals the number of boards with net `j`. The elements `boardsInPartialWithNet[1:n]` have their default initial value of 0. This setting corresponds to a null partial permutation. The invocation `rBoard(1,0)` searches the permutation tree of `partial[1:numberOfBoards]` for a best completion of the null permutation whose density is zero. In general, `rBoard(currentLevel,densityOfPartial)` finds the best completion of the partial permutation `partial[1:currentLevel-1]`. This partial permutation has density `densityOfPartial`.

The method `rBoard` (Program 21.15) has the same structure as Program 21.13, which also searches a permutation space. In Program 21.15, however, when `i` equals `numberOfBoards`, all boards have been placed and `densityOfPartial` is the density of the complete permutation. Since the algorithm completes only those permutations that are better than the best found so far, we need not verify that `densityOfPartial` is less than `leastDensitySoFar`. When `currentLevel` $<$ `numberOfBoards`, the permutation is not complete. `partial[1:currentLevel-1]` defines the partial permutation at the current tree node, and `densityOfPartial` is its density. Each child of this node expands this partial permutation by adding one board at the end. For each such expansion, the new density `density` is computed, and only those nodes for which `density` $<$ `leastDensitySoFar` are searched. Other nodes and their subtrees are not searched.

Complexity Analysis

At each node of the permutation tree, function `rBoard` spends $\Theta(m)$ time computing the density at each child. So the total time spent computing these densities is $O(mn!)$. In addition, $O(n!)$ time is spent generating permutations and $O(mn)$ time is spent updating the best. Note that each update reduces `bestDensitySoFar` by at least one, and on termination `bestDensitySoFar` ≥ 0. So the number of updates is $O(m)$. The overall complexity of `rBoard` is $O(mn!)$.

```
/** preprocessor for recursive backtracking method
  * @return density of best arrangement */
public static int arrangeBoards(int [][] theBoard,
                                int theNumberOfNets,
                                int [] bestPermutation)
{
   // set class data members
   numberOfBoards = theBoard.length - 1;
   numberOfNets = theNumberOfNets;
   partial = new int [numberOfBoards + 1];
   bestPermutationSoFar = bestPermutation;
   boardsWithNet = new int [numberOfNets + 1];
   boardsInPartialWithNet = new int [numberOfNets + 1];
   leastDensitySoFar = numberOfNets + 1;
   board = theBoard;

   // initialize partial to identity permutation
   // and compute boardsWithNet[]
   for (int i = 1; i <= numberOfBoards; i++)
   {
      partial[i] = i;
      for (int j = 1; j <= numberOfNets; j++)
         boardsWithNet[j] += board[i][j];
   }

   // find best arrangement
   rBoard(1, 0);
   return leastDensitySoFar;
}
```

Program 21.14 Preprocessor for `rBoard` (Program 21.15)

EXERCISES

4. Prove that the two-ship-loading strategy in which the first is loaded as close to its capacity as possible finds a feasible loading whenever there is a way to load all containers.

5. Experiment with the codes of Programs 21.3 and 21.5 to determine their relative run-time performance.

```
private static void rBoard(int currentLevel, int densityOfPartial)
{// search from a node at level currentLevel
   if (currentLevel == numberOfBoards)
   {// all boards placed, we are at a better permutation
      for (int j = 1; j <= numberOfBoards; j++)
         bestPermutationSoFar[j] = partial[j];
      leastDensitySoFar = densityOfPartial;
   }
   else // try out subtrees
      for (int j = currentLevel; j <= numberOfBoards; j++)
      {// try child with board partial[j] as next one

         // update boardsInPartialWithNet[]
         // and compute density at last slot
         int density = 0;
         for (int k = 1; k <= numberOfNets; k++)
         {
            boardsInPartialWithNet[k] += board[partial[j]][k];
            if (boardsInPartialWithNet[k] > 0 &&
                boardsWithNet[k] != boardsInPartialWithNet[k])
               density++;
         }

         // update density to be overall density of partial arrangement
         if (densityOfPartial > density)
            density = densityOfPartial;

         // search subtree only if it may contain a better arrangement
         if (density < leastDensitySoFar)
         {// move to child
            MyMath.swap(partial, currentLevel, j);
            rBoard(currentLevel + 1, density);
            MyMath.swap(partial, currentLevel, j);
         }

         // reset boardsInPartialWithNet[]
         for (int k = 1; k <= numberOfNets; k++)
            boardsInPartialWithNet[k] -= board[partial[j]][k];
      }
}
```

Program 21.15 Search the permutation tree

6. Write a new version of Program 21.3 that uses strategy 1 to achieve a time complexity of $O(2^n)$.

7. Modify Program 21.3 using strategy 2 to reduce its time to $O(2^n)$.

8. Write a recursive backtracking algorithm for the sum-of-subset problem. In this problem you are given an array of integer weights and are to find a subset of the weights that sums to a target weight c. Notice that in this case we can terminate as soon as a subset with sum c is found. There is no need to remember the best solution found so far. Your code should not use an array such as the array x that is used in Program 21.3. Rather, the solution should be reconstructed as the recursion unfolds after finding a subset with sum c.

9. Refine Program 21.7 so that it also computes a 0/1 array `x` that corresponds to the best packing of the knapsack.

10. Develop an iterative backtracking algorithm for the 0/1 knapsack problem. Your algorithm should be similar to Program 21.5. Notice that the method `bound` can be modified so that following the computation of a bound, you can avoid remaking the left moves made by `bound` and move directly to the left-most node moved to by `bound`.

11. Write an iterative version of Program 21.11 that corresponds to Program 21.5. What can you say about the relative merits of the two versions?

12. Write a version of Program 21.10 that begins by sorting the vertices into decreasing order of degree. Do you expect this version to work any better than Program 21.10 works?

13. Write a backtracking algorithm for the max-independent-set problem.

14. Write a version of the max-clique code (Programs 21.10 and 21.11) that is implementation independent. The new code is to be a member of the abstract class `Graph` (Program 17.1), and the same code should work on instances of the classes `AdjacencyGraph`, `AdjacencyWGraph`, `LinkedGraph`, and `LinkedWGraph` (see Section 17.7).

15. Let G be a directed graph with n vertices. Let Max_i be the cost of the most expensive edge that leaves vertex i.

    (a) Show that every traveling-salesperson tour has a cost less than $\sum_{i=1}^{n} Max_i$ $+1$.

    (b) Use this bound as the initial value of `costOfBestTourSoFar`. Rewrite `btSalesperson` and `rTSP`, simplifying the code where possible.

16. Let G be a directed graph with n vertices. Let $MinOut_i$ be the cost of the least expensive edge that leaves vertex i.

(a) Show that all traveling-salesperson tours with the prefix x_1 through x_i cost at least $\sum_{j=2}^{i} A(x_{j-1}, x_j) + \sum_{j=i}^{n} MinOut_{x_j}$ where $A(u, v)$ is the cost of the edge (u, v).

(b) Use the result of (a) to obtain a condition stronger than

```
if (a[partialTour[currentLevel - 1]][partialTour[j]] != null
    && (costOfBestTourSoFar == null ||
     ((Operable) costOfPartialTour.
       add(a[partialTour[currentLevel - 1]][partialTour[j]])).
       compareTo(costOfBestTourSoFar) < 0))
```

which is used in Program 21.13 to determine when to move to a child node. The first sum is easily computed from `costOfPartialTour`, and the second is easily computed by maintaining a new variable `minAdditionalCost` that equals the sum of the $MinOut_i$ values of the vertices not included in the path so far constructed.

(c) Test the new version of `rTSP` to see how many nodes of the permutation tree it visits compared to the number visited by Program 21.13.

17. Consider any board arrangement. The length of a net is the distance between the first and last boards that include this net. For the board arrangement of Figure 21.9, the first board that includes N_4 is in slot 3, and the last in slot 6. The net length is therefore 3. The length of net N_2 is 2 because its first board is in slot 1 and its last in slot 3. The length of the longest net in the arrangement of Figure 21.9 is 3. Write a backtracking code to find the board arrangement that has the smallest maximum length. Test the correctness of your code.

18. [Vertex Cover] Let G be an undirected graph. A subset U of its vertices is a **vertex cover** iff for every edge (u, v) of G either u or v or both are in U. The number of vertices in U is the **size** of the cover. {1,2,5} is a vertex cover of size three in the graph of Figure 21.7(a). Write a backtracking algorithm to find a vertex cover of least size. What is its complexity?

19. [Simple Max Cut] Let G be an undirected graph and let U be any subset of its vertices. Let V be the remaining vertices of G. The number of edges with one endpoint in U and the other in V is the size of the **cut** defined by U. Write a backtracking algorithm to find the size of the maximum cut as well as the corresponding U. What is its complexity?

20. [Machine Design] A certain machine consists of n components. Each component is available from three vendors. Let w_{ij} be the weight of component i available from vendor j and let c_{ij} be its cost. Write a backtracking algorithm to determine the least-weight machine that can be constructed at a cost no more than c. What is its complexity?

21. [Network Design] A petroleum delivery network has been represented as a directed weighted acyclic graph G. G has a vertex s called the **source** vertex. This vertex is the source from which the petroleum flows to the remaining vertices. The in-degree of s is 0. Each edge weight gives the distance between the two vertices it connects. The pressure loss as petroleum flows through the network is a function of the distance traveled. To ensure proper operation of the network, it is necessary to maintain a certain minimum pressure $P_{\min}$ throughout the network. To maintain this minimum pressure, pressure boosters may be placed at some or all of the vertices of G. A pressure booster restores the pressure to the maximum allowable level $P_{\max}$. Let d be the distance petroleum can flow before its pressure drops from $P_{\max}$ to $P_{\min}$. In the **booster-placement problem**, we need to place the minimum number of boosters so that petroleum flows a distance no more than d before encountering a booster. Write a backtracking algorithm for the booster-placement problem. What is its complexity?

22. [n Queens] In the n-queens problem, we wish to find a placement of n queens on an $n \times n$ chessboard such that no two queens attack. Two queens are said to **attack** iff they are in the same row, column, diagonal, or antidiagonal of the chessboard. Hence we may assume that in any feasible solution, queen i is placed in row i of the chessboard. So we are interested only in determining the column placement of each queen. Let c_i denote the column that queen i is placed in. If no two queens attack, then $[c_1, \cdots, c_n]$ is a permutation of $[1, 2, \cdots, n]$. The solution space for the n-queens problem can therefore be limited to all permutations of $[1, 2, \cdots, n]$.

    (a) Organize the n-queens solution space as a tree.

    (b) Write a backtracking procedure to search this tree for a feasible placement of the n queens.

23. Write a method to search a subset space tree, which is a binary tree, by backtracking. The parameters to your method should include functions to determine whether a node is feasible, compute a bound at this node, determine if this bound is better than another value, and so on. Test your code by using it on the loading and 0/1 knapsack problems.

24. Do Exercise 23 for permutation space trees.

25. Write a method to search a solution space by backtracking. The parameters to your method should include functions to generate the next child of a node, determine whether this next child is feasible, compute a bound at this node, determine whether this bound is better than another value, and so on. Test your code by using it on the loading and 0/1 knapsack problems.