CHAPTER 22

BRANCH AND BOUND

BIRD'S-EYE VIEW

All good things must come to an end. We are at the last chapter of this book. Fortunately, most of the concepts used in this chapter have been developed in earlier ones. Like backtracking, branch and bound searches a solution space that is often organized as a tree. The common tree organizations are the subset and permutation trees introduced in Chapter 21. However, unlike backtracking, which searches these tree organizations in a depth-first manner, branch and bound usually searches these trees in either a breadth-first or least-cost manner. The applications considered in this chapter are the same as those of Chapter 21. Consequently, it should be easy for you to see the similarities and differences between the backtracking and branch-and-bound methods.

Since the space requirements of branch-and-bound algorithms are often considerably more than those of their backtracking counterparts, backtracking is often more successful at finding the answer in memory-limited situations.

22.1 THE METHOD

Branch and bound is another way to systematically search a solution space. It differs from backtracking primarily in the way an E-node is expanded. Each live node becomes an E-node exactly once. When a node becomes an E-node, all new nodes that can be reached using a single move are generated. Generated nodes that cannot possibly lead to a (optimal) feasible solution are discarded (i.e., the node dies). The remaining nodes are added to the list of live nodes, and then one node from this list is selected to become the next E-node. The selected node is extracted from the list of live nodes and expanded. This expansion process is continued until either the answer is found or the list of live nodes becomes empty.

There are two common ways to select the next E-node (though other possibities exist):

- **First In, First Out (FIFO)**
  This scheme extracts nodes from the list of live nodes in the same order as they are put into it. The live node list behaves as a queue.

- **Least Cost or Max Profit**
  This scheme associates a cost or profit with each node. If we are searching for a solution with least cost, then the list of live nodes can be set up as a min heap. The next E-node is the live node with least cost. If we want a solution with maximum profit, the live node list can be set up as a max heap. The next E-node is the live node with maximum profit.

Example 22.1 [Rat in a Maze] Consider the rat-in-a-maze instance of Figure 21.3(a) and the solution space organization of Figure 21.1. In a FIFO branch and bound, we begin with (1,1) as the E-node and an empty live node list. The maze position (1,1) is set to 1 to prevent a return to this position. (1,1) is expanded, and its neighbor nodes (1,2) and (2,1) are added to the queue (i.e., the list of live nodes). Positions (1,2) and (2,1) are set to 1 in the maze to prevent moving to these positions again. The maze now is as shown in Figure 22.1(a), and the E-node (1,1) is discarded.

$$\begin{array}{ccc} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{array} \qquad \begin{array}{ccc} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{array} \qquad \begin{array}{ccc} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{array}$$

(a) (b) (c)

Figure 22.1 FIFO branch and bound in a maze

Node (1,2) is removed from the queue and expanded. Its three neighbors (see the solution space of Figure 21.1) are examined. Only (1,3) represents a feasible move (the remaining two nodes represent moves to blocked positions), and it is added to the queue. This maze position is set to 1, and the status of $maze$ is as shown in Figure 22.1(b). Node (1,2) is discarded. The next E-node is extracted from the queue. It is (2,1). When this E-node is expanded, node (3,1) is added to the queue, $maze(3,1)$ is set to 1, and node (2,1) is discarded. $maze$ is as shown in Figure 22.1(c), and the queue has the nodes (1,3) and (3,1) on it. (1,3) becomes the next E-node. Since this E-node does not get us to any new nodes, it is discarded and (3,1) becomes the new E-node. At this time the queue is empty. Node (3,1) gets us to node (3,2), which is now added to the queue, and (3,1) is discarded. (3,2) is the next E-node. Expanding this node, we reach the exit (3,3), and the search terminates.

A FIFO search of a maze has the desirable property that the path found (if any) is a shortest path from the entrance to the maze. Observe that the path found by backtracking may not be a shortest path. Interestingly, we have already seen the code for a FIFO branch-and-bound search of a maze. The wire-routing code of Program 10.8 when run with the start position (1,1) and finish position (n, n) performs a FIFO branch-and-bound search of the maze and determines the shortest start-to-finish path. ■

Example 22.2 [0/1 Knapsack] We will carry out both a FIFO and a max-profit branch-and-bound search on the knapsack instance $n = 3$, $w = [20, 15, 15]$, $p = [40, 25, 25]$, and $c = 30$. The FIFO version uses a queue to keep track of live nodes, as these nodes are to be extracted in FIFO order. The max-profit version uses a max heap, as E-nodes are selected from among the live nodes in decreasing order of profit earned at the live node or in decreasing order of an estimate of the maximum profit earned at any leaf in the live node's subtree. The instance we are using is the same as that used in Example 21.2, and the solution space tree is that of Figure 21.2.

The FIFO branch-and-bound search begins with the root A as the E-node. At this time the live node queue is empty. When node A is expanded, nodes B and C are generated. As both are feasible, they are added to the live node queue, and node A is discarded. The next E-node is node B. It is expanded to get nodes D and E. D is infeasible and discarded, while E is added to the queue. Next C becomes the E-node. When expanded, it leads to nodes F and G. Both are feasible and added to the queue. The next E-node, E, gets us to J and K. J is infeasible and discarded. K is a feasible leaf and represents a possible solution to the instance. Its profit value is 40.

The next E-node is node F. Its children L and M are generated. L represents a feasible packing with profit value 50, while M represents a feasible packing with value 15. G is the last node to become the E-node. Its children N and O are both feasible. The search now terminates because the live node queue is empty. The best solution found has value 50.

Notice that a FIFO branch and bound working on a solution space tree is very much like a breadth-first search of the tree with the root as the start vertex. The major difference is that FIFO branch and bound does not search subtrees of infeasible nodes.

The max-profit branch-and-bound algorithm begins with node A of the solution space tree as the initial E-node. The max heap of live nodes is initially empty. Expanding the initial E-node yields the nodes B and C. Both are feasible and are inserted into the heap. The profit earned at node B is 40 (as $x_1 = 1$ here), while that earned at C is 0. A is discarded, and B becomes the next E-node, as its profit value is larger than that of C. When B is expanded, the nodes D and E are generated. D is infeasible and discarded. E is added to the heap. E becomes the next E-node, as its profit value is 40, while that of C is 0. When E is expanded, the nodes J and K are generated. J is infeasible and discarded. K represents a feasible solution. This solution is recorded as the best found so far, and K discarded. Only one live node, node C, remains. This live node becomes the new E-node. Nodes F and G are generated and inserted into the max heap. F has a profit of 25 and becomes the next E-node. Nodes L and M are generated. Both are discarded, as they are leaf nodes. The solution corresponding to L is recorded as the best found so far. Finally, G becomes the E-node, and the nodes N and O generated. Both are leaves and are discarded. Neither represents a solution that is better than the best found so far, so no solution update takes place. The heap is empty, and there is no next E-node. The search terminates with L representing the optimal solution.

As in the case of backtracking, *the search for an optimal solution can be speeded by using a bounding function.* This function places an upper bound on the maximum profit that can possibly be obtained by expanding a particular node. If a node's bound isn't larger than the profit of the best solution found so far, it may be discarded without expansion. Further, in the case of a max-profit branch and bound, nodes may be extracted from the max heap in nonincreasing order of the profit bound, rather than by the actual profit for the node. This strategy to extract nodes gives preference to live nodes that are likely to lead to good leaves, rather than to nodes that have already earned large profit. ■

Example 22.3 [Traveling Salesperson] Consider the four-city traveling-salesperson instance of Figure 21.4. The corresponding solution space organization is the permutation tree of Figure 21.5. A FIFO branch and bound would begin with node B as the initial E-node and an empty queue of live nodes. When B is expanded, the nodes C, D, and E are generated. As there is an edge from vertex 1 to each of the vertices 2, 3, and 4, all three of these nodes are feasible and all three are added to the queue. The E-node B is discarded, and the next E-node is the first live node on the queue. Node C is the next E-node. When this node is expanded, nodes F and G are generated. Both are added to the queue because the graph of Figure 21.4 has an edge from vertex 2 to both vertex 3 and vertex 4. Next D becomes the E-node, and then E becomes the E-node. Now the live node queue contains the nodes F through K.

The next E-node is F. It is expanded to obtain node L, which is a leaf. A tour has been found. Its cost is 59. The next E-node, G, gets us to leaf M, which defines a tour whose cost is 66. When node H becomes the E-node, the leaf N that represents a tour of cost 25 is reached. The next E-node is I. It represents the partial tour 1,3,4 whose cost, 26, is more than that of the best tour found so far. So I is not expanded. Finally, J and K become E-nodes and get expanded. Following this expansion, the queue is empty, and the algorithm terminates with node N identifying the best tour.

Instead of searching the solution space tree in a FIFO manner, we could search in a least-cost manner, using a min heap to store the live nodes. This search also begins with node B as the E-node and an empty live node list. When B is expanded, the nodes C, D, and E are generated and added to the min heap. Of the nodes in the min heap, E has least cost (the partial tour 1,4 has cost 4) and becomes the new E-node. E is expanded, and the nodes J and K are added to the min heap. These nodes have a cost of 14 and 24, respectively. The least-cost node in the min heap is now D. It becomes the E-node, and H and I are generated. The min heap now contains the nodes C, H, I, J, and K. Of these nodes, H has least cost. H is the next E-node. It is expanded, and the tour 1,3,2,4,1 of cost 25 is completed. Node J is the next E-node. When it is expanded, we reach node P, which represents a tour of cost 25. Nodes K and I are the next two E-nodes. As the cost of I exceeds that of the best solution found so far, the search terminates; none of the remaining live nodes can get us to a better solution.

As in the case of the knapsack example (Example 22.2), we can use a bounding function to reduce the number of nodes generated and expanded. Such a function will determine a bound on the minimum-cost tour lower than can possibly be obtained by expanding a particular node. If a node's bound isn't smaller than the cost of the best tour found so far, that node may be discarded without expansion. Further, in the case of a least-cost branch and bound, nodes may be extracted from the min heap in nondecreasing order of the cost bound. ■

As mentioned in the preceding examples, we can use bounding functions to reduce the number of nodes of the solution space tree that are generated. When developing a bounding function, we should keep in mind that our primary objective is to solve the instance using the least amount of time and using no more memory than is available to us. *Solving the problem by generating the least number of nodes is not the primary objective.* As a result, we need a bounding function that pays for its computation time by a corresponding reduction in the number of nodes generated.

Backtracking generally has a memory advantage over branch and bound. The memory needed by backtracking is O(length of longest path in the solution space organization), while that needed by branch and bound is O(size of solution space organization). For a subset space backtracking requires $\Theta(n)$ memory, while the branch-and-bound methods considered require $O(2^n)$ memory. For a permutation space backtracking requires $\Theta(n)$ memory, while branch and bound needs $O(n!)$.

Although a max-profit or least-cost branch and bound has intuitive appeal over backtracking and might be expected to examine fewer nodes on many inputs, the space needs might exceed what is available sooner than the time needs of backtracking exceed the length of time we are willing to wait for the answer.

EXERCISES

1. In a **last-in-first-out (LIFO)** branch-and-bound search, the list of live nodes behaves as a stack. Describe the progress of such a method on the knapsack instance of Example 22.2. How does LIFO branch and bound differ from backtracking?

2. Consider the 0/1 knapsack instance with $n = 4$, $p = [4, 3, 2, 1]$, $w = [1, 2, 3, 4]$, and $c = 6$.

   (a) Draw the solution space tree for a four-object knapsack instance.

   (b) Trace through the working of a FIFO branch-and-bound search, as was done in Example 22.2.

   (c) Use the method `bound` (Program 21.9) to determine the maximum profit obtainable at any leaf in a subtree. Use this bound together with the value of the best solution determined so far to decide whether or not to add a node to the live node list. Which nodes of the solution space tree are generated by a FIFO branch and bound that uses this mechanism?

   (d) Trace through the working of a max-profit branch-and-bound search, as was done in Example 22.2.

   (e) Which nodes of the solution space tree are generated during a max-profit branch and bound when the bounding function of (c) is used?

22.2 APPLICATIONS

22.2.1 Container Loading

FIFO Branch and Bound

The container-loading problem of Section 21.2.1 essentially requires us to find a maximum loading of the first ship. This problem is a subset-selection problem, and the solution space organization is a subset tree. The FIFO branch-and-bound analog of Program 21.1 is Program 22.1. Like Program 21.1, Program 22.1 finds only the weight of a maximum loading.

The method `maxLoading` does a FIFO branch-and-bound search of the solution space tree using the queue `liveNodeQueue` to store the weight associated with each live node. The queue also stores the weight -1 to mark the end of a level of live nodes. The method `addLiveNode` is used to add nodes (i.e., their weights) to the

```
public static int maxLoading(int [] weight, int capacity)
{
   // set class data members
   numberOfContainers = weight.length - 1;
   maxWeightSoFar = 0;
   liveNodeQueue = new ArrayQueue();
   liveNodeQueue.put(new Integer(-1));  // end-of-level marker

   // initialize for level 1 E-node
   int eNodeLevel = 1;
   int eNodeWeight = 0;

   // search subset space tree
   while (true)
   {
      // check left child of E-node
      if (eNodeWeight + weight[eNodeLevel] <= capacity)
         // left child
         addLiveNode(eNodeWeight + weight[eNodeLevel], eNodeLevel);

      // right child is always feasible
      addLiveNode(eNodeWeight, eNodeLevel);

      // get next E-node
      eNodeWeight = ((Integer) liveNodeQueue.remove()).intValue();
      if (eNodeWeight == -1)
      {// end of level
         if (liveNodeQueue.isEmpty())            // no more live nodes
            return maxWeightSoFar;
         liveNodeQueue.put(new Integer(-1));     // end-of-level marker
         // get next E-node
         eNodeWeight = ((Integer) liveNodeQueue.remove()).intValue();
         eNodeLevel++;
      }
   }
}
```

Program 22.1 FIFO branch-and-bound search for container loading (continues)

```
private static void addLiveNode(int theWeight, int theLevel)
{
   if (theLevel == numberOfContainers)
   {// feasible leaf
      if (theWeight > maxWeightSoFar)  // better leaf reached
         maxWeightSoFar = theWeight;
    }
   else  // not a leaf
      liveNodeQueue.put(new Integer(theWeight));
}
```

Program 22.1 FIFO branch-and-bound search for container loading (concluded)

live node queue. This method begins by checking whether the level of the node to be added equals the number of containers. If so, we are at a leaf. Leaves are not added to the queue, as these nodes cannot be expanded. Leaves that are reached define feasible solutions, and each is checked for being better than the best found so far. The weight of a nonleaf node is added to the queue.

`maxLoading` begins by initializing `eNodeLevel` $= 1$ (current E-node is the root) and `maxWeightSoFar` $= 0$ (value of best loading found so far). At this time no live nodes are in the queue. A -1 is added to the queue to indicate that we are at the end of level 1. In the `while` loop we first see whether the left child of the E-node is feasible. If so, `addLiveNode` is invoked. Then the right child is added. (This child is guaranteed to be feasible.)

When both children of the E-node have been generated, the E-node dies and we extract the next E-node from the queue. The queue cannot be empty at this time because it must contain at least the end-of-level marker -1. If we have reached the end of a level, then we see whether any live nodes from the next level are present. These nodes are present iff the queue is not empty. When live nodes from the next level are present, we add an end-of-level marker to the queue and begin to process the live nodes at the next level.

The time and space requirements of `maxLoading` are $O(2^n)$.

An Improvement

We may attempt the refinement used in Program 21.2. In this refinement a right child was pursued only if the weight associated with it plus (`remainingWeight`) exceeds `maxWeightSoFar`. In Program 22.1 `maxWeightSoFar` doesn't get updated until `eNodeLevel` equals `numberOfContainers`. Prior to this time the right-child test always succeeds, as `maxWeightSoFar` $= 0$ and `remainingWeight` > 0. When `eNodeLevel` equals `numberOfContainers`, no more nodes are added to the queue. So the right-child test is of no use at this time.

To make the right-child test effective, we need to update `maxWeightSoFar` earlier. We know that the weight of the best loading is the maximum of the weights associated with the feasible nodes in the subset tree. Since these associated weights increase only when a move is made to a left child, we may update `maxWeightSoFar` at all such moves. This observation results in the code of Program 22.2. When a live node is added to the queue, `theWeight` cannot exceed `maxWeightSoFar` and so `maxWeightSoFar` is not updated. A single statement, inserted directly into `maxLoading`, now replaces the method `addLiveNode`.

Finding the Best Subset

To be able to find the best subset, we need to store paths from the live nodes to the tree root. Then when we have determined which leaf gives the best loading, we can traverse the path to the root setting the `x` values. We need to change the data type of the elements in the queue of live nodes from `Integer` to `QNode`, where `QNode` has the instance data members `parent` (pointer to parent node in the solution space tree), `leftChild` (true iff node is the left child of its parent), and `weight` (weight of partial loading at this node). The new branch-and-bound code appears in Program 22.3.

Max-Profit Branch and Bound

In a max-profit branch-and-bound search of the subset tree, the list of live nodes is a max-priority queue. Each live node `x` in the queue has an upper weight (or max profit) associated with it. This upper weight is the weight associated with the node `x` plus the weight of the remaining containers. Live nodes become E-nodes in decreasing order of their upper weight. Notice that if `x` is a node with upper weight `x.upperWeight`, then no node in its subtree has weight more than `x.upperWeight`. From this observation and the observation that the weight associated with a leaf node equals its upper weight, we conclude that when a leaf becomes the E-node in a max-cost branch and bound, no remaining live node can lead to a leaf with more weight. Therefore, we may terminate the search for the best loading.

This strategy may be implemented in one of two ways. In the first each live node resides in the max-priority queue alone. In this case each node must contain the path from the root of the subset tree to the node. This information is needed to determine the `x` values once we have identified the leaf that yields the best loading. In the second strategy, in addition to placing each live node into the max-priority queue, the node is entered into a separate tree structure that represents the portion of the subset tree generated. When the best leaf is identified, the corresponding `x` values are determined by following the path from the leaf to the root. We will use this second implementation method. Exercise 5 explores the first method.

The solution space tree is represented using nodes of the type `BBnode`. Each node of this type has the fields `parent` (pointer to parent node in the solution space tree) and `leftChild` (true iff the node is a left child of its parent).

```
public static int maxLoading(int [] weight, int capacity)
{
   // set class data members
   numberOfContainers = weight.length - 1;
   maxWeightSoFar = 0;
   liveNodeQueue = new ArrayQueue();
   liveNodeQueue.put(new Integer(-1));  // end-of-level marker

   // initialize for level 1 E-node
   int eNodeLevel = 1;
   int eNodeWeight = 0;
   int remainingWeight = 0;
   for (int j = 2; j <= numberOfContainers; j++)
      remainingWeight += weight[j];

   // search subset space tree
   while (true)
   {
      // check left child of E-node
      int leftChildWeight = eNodeWeight + weight[eNodeLevel];
      if (leftChildWeight <= capacity)
      {// feasible left child
         if (leftChildWeight > maxWeightSoFar)
            maxWeightSoFar = leftChildWeight;
         // add to queue unless leaf
         if (eNodeLevel < numberOfContainers)
            liveNodeQueue.put(new Integer(leftChildWeight));
      }

      // check right child
      if (eNodeWeight + remainingWeight > maxWeightSoFar
          && eNodeLevel < numberOfContainers)
          // right child may lead to better leaf
          liveNodeQueue.put(new Integer(eNodeWeight));

      // get next E-node
      eNodeWeight = ((Integer) liveNodeQueue.remove()).intValue();
      if (eNodeWeight == -1)
```

Program 22.2 Improved version of Program 22.1 (continues)

```
      {// end of level
         if (liveNodeQueue.isEmpty())        // no more live nodes
            return maxWeightSoFar;
         liveNodeQueue.put(new Integer(-1));  // end-of-level marker
         // get next E-node
         eNodeWeight = ((Integer) liveNodeQueue.remove()).intValue();
         eNodeLevel++;
         remainingWeight -= weight[eNodeLevel];
      }
   }
}
```

Program 22.2 Improved version of Program 22.1 (concluded)

The max-priority queue may be represented as a max heap. The elements of this max heap are of type `HeapNode` where each instance of `HeapNode` has the fields `liveNode` (a pointer to the node p of the solution space tree represented by this heap node), `upperWeight` (upper bound on the weight at p), and `level` (the level of p). The class `HeapNode` implements the interface `Comparable` by using its `upperWeight` field.

The method `addLiveNode` (Program 22.4), which is a member of the class `MaxProfitLoading`, adds a new live node to the subset tree, using a node of type `BBnode`, and also inserts a corresponding node into the max heap, using a node of type `HeapNode`.

The method `maxLoading` (Program 22.5) performs a max-profit branch-and-bound search beginning at the root of the solution space tree. The `while` loop generates the left and right children of the current E-node. If the left child is feasible (i.e., its weight does not exceed the capacity), it is added to the subset tree and to the max heap as a level `eNodeLevel+1` node. The right child of a feasible node is guaranteed to be feasible and so is always added to the set subtree and max heap. Following this addition, the next E-node is extracted from the max heap. If the next E-node is a leaf, it represents the optimal loading. This loading is determined by following the path from this leaf to the root.

Comment on Implementation

Define `maxWeightSoFar` to be the maximum weight associated with any of the feasible nodes generated so far. The priority queue of live nodes may contain several nodes whose `upperWeight` value does not exceed `maxWeightSoFar`. These nodes cannot possibly lead to the best leaf. Their presence in the priority queue is taking valuable queue space and also contributing to the time needed to insert/delete. We should eliminate them. One elimination strategy is to test `upperWeight` >

```
public static int maxLoading(int [] weight, int capacity,
                             int [] theBestLoading)
{
   // set class data members
   numberOfContainers = weight.length - 1;
   maxWeightSoFar = 0;
   liveNodeQueue = new ArrayQueue();
   liveNodeQueue.put(null);      // end-of-level marker
   QNode eNode = null;
   bestENodeSoFar = null;
   bestLoading = theBestLoading;

   // initialize for level 1 E-node
   int eNodeLevel = 1;
   int eNodeWeight = 0;
   int remainingWeight = 0;
   for (int j = 2; j <= numberOfContainers; j++)
      remainingWeight += weight[j];

   // search subset space tree
   while (true)
   {
      // check left child of E-node
      int leftChildWeight = eNodeWeight + weight[eNodeLevel];
      if (leftChildWeight <= capacity)
      {// feasible left child
         if (leftChildWeight > maxWeightSoFar)
            maxWeightSoFar = leftChildWeight;
         addLiveNode(leftChildWeight, eNodeLevel, eNode, true);
      }
      // check right child
      if (eNodeWeight + remainingWeight > maxWeightSoFar)
         addLiveNode(eNodeWeight, eNodeLevel, eNode, false);

      eNode = (QNode) liveNodeQueue.remove();
      if (eNode == null)
      {// end of level
         if (liveNodeQueue.isEmpty()) break;  // no more live nodes
         liveNodeQueue.put(null);             // end-of-level pointer
```

Program 22.3 Branch-and-bound code that also computes the best subset (continues)

```
         eNode = (QNode) liveNodeQueue.remove();
         eNodeLevel++;
         remainingWeight -= weight[eNodeLevel];
      }
      eNodeWeight = eNode.weight;
   }
   // construct bestLoading[] by following path from
   // bestENodeSoFar to root, bestLoading[numberOfContainers]
   // is set by addLiveNode
   for (int j = numberOfContainers - 1; j > 0; j--)
   {
      bestLoading[j] = (bestENodeSoFar.leftChild) ? 1 : 0;
      bestENodeSoFar = bestENodeSoFar.parent;
   }
   return maxWeightSoFar;
}

/** add a live node at level theLevel and having weight theWeight
  * to liveNodeQueue if not a leaf
  * if feasible leaf, set bestLoading[numberOfContainers] = 1
  * iff leftChild is true
  * @param theParent parent of new node
  * @param leftChild true iff new node is left child of theParent */
private static void addLiveNode(int theWeight, int theLevel,
                                QNode theParent, boolean leftChild)
{
   if (theLevel == numberOfContainers)
   {// feasible leaf
      if (theWeight == maxWeightSoFar)
      {// best leaf so far
        bestENodeSoFar = theParent;
        bestLoading[numberOfContainers] = (leftChild) ? 1 : 0;
      }
      return;
  }
   // not a leaf, add to queue
   QNode b = new QNode(theParent, leftChild, theWeight);
   liveNodeQueue.put(b);
}
```

Program 22.3 Branch-and-bound code that also computes the best subset (concluded)

```
/** add a new live node to the live node max heap
  * also add the live node to the solution space tree
  * @param theParent is the parent of the new live node
  * @param leftChild is true iff the new live node is
  * the left child of theParent */
private static void addLiveNode(int upperWeight, int level,
                                BBnode theParent, boolean leftChild)
{
   // create the new node of the solution space tree
   BBnode b = new BBnode(theParent, leftChild);

   // create corresponding node for max heap
   HeapNode hNode = new HeapNode(b, upperWeight, level);

   // put into max heap
   liveNodeMaxHeap.put(hNode);
}
```

Program 22.4 MaxProfitBBLoading.addLiveNode

`maxWeightSoFar` before inserting a node into the priority queue. However, since `maxWeightSoFar` increases as the algorithm progresses, nodes that pass this test at the time of insertion may fail it later on. A more aggressive strategy is to also apply the test whenever `maxWeightSoFar` increases and delete from the priority queue all nodes with `upperWeight` $<$ `maxWeightSoFar`. This strategy requires us to delete nodes with least `upperWeight`. Hence we need a priority queue that supports the operations insert, delete max, and delete min. Such a priority queue is called a **double-ended** priority queue. Data structures for double-ended priority queues appear on the Web site.

22.2.2 0/1 Knapsack Problem

A max-profit branch-and-bound algorithm for the 0/1 knapsack problem may be developed by using the method `profitBound` of Program 21.9 to compute for each live node N an upper profit `maxPossibleProfitInSubtree` such that no node in the subtree with root N has profit value more than `maxPossibleProfitInSubtree`.

The max-profit branch-and-bound code is similar to Program 22.5. We use a max heap for the live nodes and construct portions of the solution space tree as needed. The elements in the max heap are of type `HeapNode` where `HeapNode` has the data members `upperProfit` (upper bound on profit at any leaf in subtree with this root), `profit` (profit of partial solution at this node), `weight` (weight of

```
public static int maxLoading(int [] weight, int capacity,
                             int [] bestLoading)
{
   // set class data member
   liveNodeMaxHeap = new MaxHeap();

   // initialize for level 1 E-node
   int numberOfContainers = weight.length - 1;
   BBnode eNode = null;
   int eNodeLevel = 1;
   int eNodeWeight = 0;

   // remainingWeight[j] will be sum of weight[j+1:n]
   // default initial value is 0
   int [] remainingWeight = new int [numberOfContainers + 1];
   for (int j = numberOfContainers - 1; j > 0; j--)
      remainingWeight[j] = remainingWeight[j + 1] + weight[j + 1];

   // search subset space tree
   while (eNodeLevel != numberOfContainers + 1)
   {// not at a leaf
      // check children of E-node
      if (eNodeWeight + weight[eNodeLevel] <= capacity)
         // feasible left child
         addLiveNode(eNodeWeight + weight[eNodeLevel] +
                     remainingWeight[eNodeLevel], eNodeLevel + 1,
                     eNode, true);
      // right child is always feasible
      addLiveNode(eNodeWeight + remainingWeight[eNodeLevel],
                  eNodeLevel + 1, eNode, false);

      // get next E-node, heap cannot be empty
      HeapNode nextENode = (HeapNode) liveNodeMaxHeap.removeMax();
      eNodeLevel = nextENode.level;
      eNode = nextENode.liveNode;
      eNodeWeight = nextENode.upperWeight
                    - remainingWeight[eNodeLevel - 1];
   }
```

Program 22.5 Max-profit branch and bound for loading problem (continues)

```
   // construct bestLoading[] by following path
   // from eNode to the root
   for (int j = numberOfContainers; j > 0; j--)
   {
      bestLoading[j] = (eNode.leftChild) ? 1 : 0;
      eNode = eNode.parent;
   }

   return eNodeWeight;
}
```

Program 22.5 Max-profit branch and bound for loading problem (concluded)

partial solution at this node), `level` (level of this node in the solution space tree), and `liveNode` (pointer to corresponding node in solution space tree). Nodes are extracted from the max heap using their `upperProfit` value. The nodes in the solution space tree are of type `BBnode` where `BBNode` has the data members `parent` (pointer to parent in the tree) and `leftChild` (true iff node is the left child of its parent).

The code of Program 22.6 assumes that the knapsack objects have been sorted into ascending order of density, using the same technique as we used in Program 21.7. The method `addLiveNode` adds a new live node to both the solution space tree, using a node of type `BBnode`, and to the max heap, using a node of type `HeapNode`. This method is very similar to the corresponding function used for the loading problem (Program 22.4). Therefore, the code is omitted.

The method `maxProfitBBKnapsack` performs the max-profit branch-and-bound search on the subset tree. The `while` loop is iterated until a leaf becomes the E-node. Since no node remaining in the max heap has an upper profit that is more than the profit at this leaf, this leaf defines an optimal packing. This packing is determined by following the path from the leaf to the root.

The structure of the `while` loop of `maxProfitKnapsack` is very similar to that of the `while` loop of Program 22.4. First we check the feasiblity of the left child of the E-node. If this child is feasible, it is added to the subset tree as well as to the live node list (i.e., the max heap). The right child is added only if its `maxPossibleProfitInSubtree` value indicates that it might lead us to the best packing.

22.2.3 Max Clique

The solution space tree for the clique problem (Section 21.2.3) is also a subset tree. Let us use the same max-profit branch-and-bound implementation strategy as we used for the loading and knapsack problems. The nodes in the portion of the solution space tree constructed are of type `BBnode`, while the max-priority queue elements

```
private static double maxProfitBBKnapsack()
{
   // initialize for level 1 start
   BBnode eNode = null;
   int eNodeLevel = 1;
   double maxProfitSoFar = 0.0;
   double maxPossibleProfitInSubtree = profitBound(1);

   // search subset space tree
   while (eNodeLevel != numberOfObjects + 1)
   {// not at leaf
      // check left child
      double weightOfLeftChild = weightOfCurrentPacking
                                 + weight[eNodeLevel];
      if (weightOfLeftChild <= capacity)
      {// feasible left child
         if (profitFromCurrentPacking + profit[eNodeLevel]
             > maxProfitSoFar)
            maxProfitSoFar = profitFromCurrentPacking
                             + profit[eNodeLevel];
         addLiveNode(maxPossibleProfitInSubtree,
                     profitFromCurrentPacking + profit[eNodeLevel],
                     weightOfCurrentPacking + weight[eNodeLevel],
                     eNodeLevel + 1, eNode, true);
      }
      maxPossibleProfitInSubtree = profitBound(eNodeLevel + 1);

      // check right child
      if (maxPossibleProfitInSubtree >= maxProfitSoFar)
         // right child has prospects
         addLiveNode(maxPossibleProfitInSubtree,
                     profitFromCurrentPacking,
                     weightOfCurrentPacking,
                     eNodeLevel + 1, eNode, false);

      // get next E-node, heap cannot be empty
      HeapNode nextENode = (HeapNode) liveNodeMaxHeap.removeMax();
      eNode = nextENode.liveNode;
      weightOfCurrentPacking = nextENode.weight;
```

Program 22.6 Max-profit branch and bound for the 0/1 knapsack problem (continues)

```
      profitFromCurrentPacking = nextENode.profit;
      maxPossibleProfitInSubtree = nextENode.upperProfit;
      eNodeLevel = nextENode.level;
   }

   // construct bestPackingSoFar[] by following path
   // from eNode to the root
   for (int j = numberOfObjects; j > 0; j--)
   {
      bestPackingSoFar[j] = (eNode.leftChild) ? 1 : 0;
      eNode = eNode.parent;
   }

   return profitFromCurrentPacking;
}
```

Program 22.6 Max-profit branch and bound for the 0/1 knapsack problem (concluded)

are of type `HeapNode`. This time, `HeapNode` has the data members `cliqueSize` (number of vertices in the clique represented by this node), `upperSize` (maximum possible clique size for any leaf in this node's subtree), `level` (level of the node in the solution space tree), and `liveNode` (pointer to coresponding node in the solution space tree). For `upperSize` we simply use the value `cliqueSize + n - level + 1`. As a result, we can eliminate either the `cliqueSize` or the `level` field because from `upperSize` and either `cliqueSize` or `level`, the other can be computed. When an element is to be extracted from the max-priority queue, we select an element with maximum `upperSize`. In our implementation of `HeapNode`, we include all three of the fields `cliqueSize`, `upperSize`, and `level`. The inclusion of these fields makes it easier to experiment with alternative definitions of `upperSize`.

The method `addLiveNode` adds a live node to the subset tree being constructed and also to the max heap. The code is very similar to the code for the corresponding method for the loading and knapsack problems and is omitted.

Program 22.7 gives the method `maxProfitBBMaxClique`. This method performs a max-profit branch-and-bound search of the subset solution space tree. The root of this tree is the initial E-node. This node is not explicitly represented in the constructed tree. For this E-node `sizeOfCliqueAtENode` is 0 because no vertices have been selected for inclusion into the clique. The level of the E-node is designated by the variable `eNodeLevel`. This initial value of `eNodeLevel` is 1 because the initial E-node is the root of the subset tree.

```
/** max-profit branch-and-bound code to find a max clique
  * @param maxClique maxClique[i] set to 1 iff i is in max clique
  * @return size of max clique */
public int maxProfitBBMaxClique(int [] maxClique)
{
   liveNodeMaxHeap = new MaxHeap();

   // initialize for level 1 start
   BBnode eNode = null;
   int eNodeLevel = 1;
   int sizeOfCliqueAtENode = 0;
   int sizeOfMaxCliqueSoFar = 0;

   // search subset space tree
   while (eNodeLevel != n + 1)
   {// while not at leaf
      // see if vertex eNodeLevel is connected to all vertices
      // in current clique
      boolean connected = true;
      BBnode currentNode = eNode;
      for (int j = eNodeLevel - 1; j > 0;
           currentNode = currentNode.parent, j--)
         if (currentNode.leftChild && !a[eNodeLevel][j])
         {// j is in the clique but no edge between eNodeLevel and j
            connected = false;
            break;
         }

      if (connected)
      {// left child is feasible
         if (sizeOfCliqueAtENode + 1 > sizeOfMaxCliqueSoFar)
            sizeOfMaxCliqueSoFar = sizeOfCliqueAtENode + 1;
         addLiveNode(sizeOfCliqueAtENode + n - eNodeLevel + 1,
            sizeOfCliqueAtENode + 1, eNodeLevel + 1, eNode, true);
      }

      if (sizeOfCliqueAtENode + n - eNodeLevel >= sizeOfMaxCliqueSoFar)
         // right child has prospects
         addLiveNode(sizeOfCliqueAtENode + n - eNodeLevel,
            sizeOfCliqueAtENode, eNodeLevel + 1, eNode, false);
```

Program 22.7 Max-profit branch-and-bound max-clique code (continues)

```
      // get next E-node, heap cannot be empty
      HeapNode nextENode = (HeapNode) liveNodeMaxHeap.removeMax();
      eNode = nextENode.liveNode;
      sizeOfCliqueAtENode = nextENode.cliqueSize;
      eNodeLevel = nextENode.level;
   }

   // construct maxClique[] by following path from eNode to the root
   for (int j = n; j > 0; j--)
   {
      maxClique[j] = (eNode.leftChild) ? 1 : 0;
      eNode = eNode.parent;
   }

   return sizeOfMaxCliqueSoFar;
}
```

Program 22.7 Max-profit branch-and-bound max-clique code (concluded)

In the `while` loop E-nodes are expanded until a leaf (i.e., a level `n+1` node) becomes the E-node. For a leaf node `upperSize` = `sizeOfCliqueAtENode`. Since all remaining nodes have an `upperSize` value $\leq$ that of the current E-node, they cannot lead to a larger clique than the clique represented by this E-node. Therefore, the max clique has been found. The clique itself is constructed by following the path from the E-node leaf to the root of the constructed subset tree.

To expand a nonleaf E-node, we first consider its left child. At the left child, a new vertex v is included into the clique being constructed. This inclusion is possible only if an edge exists between vertex v and each of the vertices already included at the E-node. To determine the feasiblity of the left child, we follow the path from the E-node to the root, determining which vertices are included and also verifying that each included vertex is connected to vertex v by an edge. If the left child is feasible, we add it to the max-priority queue as well as to the subset tree being constructed. Next we add the right child provided that its subtree could contain a leaf that represents a max clique.

Since every graph has a max clique, we do not need to test for an empty heap when deleting from the max heap. The `while` loop is exited only when we reach a feasible leaf.

22.2.4 Traveling Salesperson

The traveling-salesperson problem was introduced in Section 21.2.4. The solution space for this problem is a permutation tree. As in the case of max-profit and least-cost branch-and-bound searches of subset trees, there are two possibilities for the implementation. In one we use only a priority queue in which each element contains the path to the root. In the other we maintain the portion of the solution space tree that is generated and a priority queue of live nodes. In the latter case the priority queue elements do not contain the path to the root. The implementation in this section uses the former approach, though the latter could also have been used.

Since we are looking for a least-cost traveling-salesperson route, we will employ a least-cost branch and bound. The implementation uses a min-priority queue of live nodes. The nodes in this queue are of type `HeapNode`. Each node of this type has the fields `partialTour` (a permutation of the numbers 1 through `n` with `partialTour[0]` being 1); `sizeOfPartialTour` (an integer such that the path from the root of the permutation tree to this node defines the tour prefix `partialTour[0:sizeOfPartialTour]` and the vertices yet to be visited by the tour are `partialTour[sizeOfPartialTour+1:n-1]`; also equals number of edges in partial tour); `costOfPartialTour` (cost of tour prefix represented by the path from the solution space tree root to this node); `lowerCost` (least possible cost of any leaf in this node's subtree); and `minAdditionalCost` (sum of costs of least-cost outbound edges from vertices `partialTour[sizeOfPartialTour:n-1]`). Extractions from the min heap are done by `lowerCost` value. The branch-and-bound code appears in Program 22.8.

Program 22.8 begins by creating a min heap that represents the min-priority queue of live nodes. Next we compute the cost of the cheapest outbound edge from each vertex in the digraph. If some vertex has no outbound edge, the digraph has no tour and we terminate. If each vertex has an outbound edge, a least-cost branch and bound is initiated. We begin with the child of the root (node B in Figure 21.5) as the first E-node. At this node the tour prefix constructed is just the single vertex 1. Therefore, `sizeOfPartialTour` = 0, `partialTour[0]` = 1, and `partialTour[1:n-1]` are the remaining vertices (2, 3, $\cdots$, `n`). The tour prefix 1 has cost 0, so `costOfPartialTour` = 0. Also, `minAdditionalCost` = $\sum_{i=1}^{\mathtt{n}}$`costOfMinOutEdge`$[i]$. Initially, no tour has been found, so `costOfBestTourSoFar` is set to `null`.

The `while` loop expands E-nodes until we reach one that is a leaf or we run out of E-nodes to expand. A leaf is detected by noticing that when `sizeOfPartialTour` = `n-1`, the tour prefix is `partialTour[0:n-1]`; this prefix includes all `n` vertices of the digraph. Hence a live node with `sizeOfPartialTour` = `n-1` represents a leaf. By the nature of the algorithm, a leaf has `costOfPartialTour` and `lowerCost` equal to the cost of the tour it represents. Since all remaining live nodes have a `lowerCost` value at least as much as that of the first leaf extracted from the min heap, none of these remaining nodes can lead to a better leaf. Therefore, the search for an optimal tour may terminate as soon as a leaf becomes the E-node. If we run out of E-nodes before a leaf is reached, the graph has no tour.

```
/** least-cost branch-and-bound code to find a shortest tour
  * @param theZero zero weight
  * @param bestTour bestTour[i] set to i'th vertex on shortest tour
  * @return cost of shortest tour */
public Object leastCostBBSalesperson(int bestTour[], Operable theZero)
{
   MinHeap liveNodeMinHeap = new MinHeap();

   // costOfMinOutEdge[i] = cost of least-cost edge leaving vertex i
   Operable [] costOfMinOutEdge = new Operable [n + 1];

   Operable sumOfMinCostOutEdges = (Operable) theZero.zero();
                    // use a new copy of zero

   for (int i = 1; i <= n; i++)
   {// compute costOfMinOutEdge[i] and sumOfMinCostOutEdges
      Operable minCost = null;
      for (int j = 1; j <= n; j++)
         if (a[i][j] != null && (minCost == null ||
             minCost.compareTo(a[i][j]) > 0))
            minCost = (Operable) a[i][j];

      if (minCost == null) return null; // no route
      costOfMinOutEdge[i] = minCost;
      sumOfMinCostOutEdges.increment(minCost);
   }

   // initial E-node is tree root
   HeapNode eNode = new HeapNode();
   eNode.partialTour = new int [n];
   for (int i = 0; i < n; i++)
      eNode.partialTour[i] = i + 1;
   eNode.sizeOfPartialTour = 0;              // partial tour is
                                             // partial[0:0]
   eNode.costOfPartialTour = theZero;        // its cost is zero
   eNode.minAdditionalCost = sumOfMinCostOutEdges;
   Operable costOfBestTourSoFar = null;      // no tour found so far
   int [] partialTour = eNode.partialTour;   // shorthand for
                                             // eNode.partialTour
```

Program 22.8 Least-cost branch and bound for traveling salesperson (continues)

```
// search permutation tree
while (eNode != null && eNode.sizeOfPartialTour < n - 1)
{// not at leaf
   partialTour = eNode.partialTour;
   if (eNode.sizeOfPartialTour == n - 2)
   {// parent of leaf
      // complete tour by adding two edges
      // see whether new tour is better
      if (a[partialTour[n - 2]][partialTour[n - 1]] != null
          && a[partialTour[n - 1]][1] != null
          && (costOfBestTourSoFar == null ||
             ((Operable) ((Operable) eNode.costOfPartialTour
            .add(a[partialTour[n - 2]][partialTour[n - 1]]))
            .add(a[partialTour[n - 1]][1]))
            .compareTo(costOfBestTourSoFar) < 0))
      {// better tour found
         costOfBestTourSoFar = (Operable) (((Operable) eNode
                .costOfPartialTour.add(a[partialTour[n - 2]]
                 [partialTour[n - 1]]))
                .add(a[partialTour[n - 1]][1]));
          eNode.costOfPartialTour = costOfBestTourSoFar;
          eNode.lowerCost = costOfBestTourSoFar;
          eNode.sizeOfPartialTour++;
          liveNodeMinHeap.put(eNode);
      }
   }
   else
   {// generate children
      for (int i = eNode.sizeOfPartialTour + 1; i < n; i++)
         if (a[partialTour[eNode.sizeOfPartialTour]]
             [partialTour[i]] != null)
         {
            // feasible child, bound path cost
            Operable costOfPartialTour = (Operable) eNode
                .costOfPartialTour
                .add(a[partialTour[eNode.sizeOfPartialTour]]
                      [partialTour[i]]);
```

Program 22.8 Least-cost branch and bound for traveling salesperson (continues)

```
              Operable minAdditionalCost =
                  (Operable) eNode.minAdditionalCost.subtract
                  (costOfMinOutEdge[partialTour
                   [eNode.sizeOfPartialTour]]);
              Operable leastCostPossible =
                (Operable) costOfPartialTour.add(minAdditionalCost);
              if (costOfBestTourSoFar == null ||
               leastCostPossible.compareTo(costOfBestTourSoFar) < 0)
              {// subtree may have better leaf, put root in min heap
                  HeapNode hNode = new HeapNode();
                  hNode.partialTour = new int [n];
                  for (int j = 0; j < n; j++)
                     hNode.partialTour[j] = partialTour[j];
                  hNode.partialTour[eNode.sizeOfPartialTour + 1] =
                           partialTour[i];
                  hNode.partialTour[i] =
                          partialTour[eNode.sizeOfPartialTour + 1];
                  hNode.costOfPartialTour = costOfPartialTour;
                  hNode.sizeOfPartialTour = eNode.sizeOfPartialTour
                                            + 1;
                  hNode.lowerCost = leastCostPossible;
                  hNode.minAdditionalCost = minAdditionalCost;
                  liveNodeMinHeap.put(hNode);
              }
           }
        }

      // get next E-node
      eNode = (HeapNode) liveNodeMinHeap.removeMin();
   }

   if (costOfBestTourSoFar == null)
      return null; // no route

   // copy best route into bestTour[1:n]
   for (int i = 0; i < n; i++)
      bestTour[i + 1] = partialTour[i];

   return costOfBestTourSoFar;
}
```

Program 22.8 Least-cost branch and bound for traveling salesperson (concluded)

The body of the `while` loop is split into two cases. The first is for E-nodes with `sizeOfPartialTour` = `n-2`. At this time the E-node is the parent of a single leaf. If this leaf defines a feasible tour and if the tour cost is less than that of the best tour found so far, the leaf is inserted into the min heap. Otherwise, the leaf is discarded, and we move on to the next E-node.

All other E-nodes fall into the second case handled in the body of the `while`. Now we generate each child of the E-node. Since the E-node represents the feasible path `partialTour[0:sizeOfPartialTour]`, the feasible children are those for which `(partialTour[s],partialTour[i])` is an edge of the digraph and `partialTour[i]` is one of `partialTour[sizeOfPartialTour+1:n-1]`. For each feasible child, we compute the cost `costOfPartialTour` of the prefix (`partialTour[0:sizeOfPartialTour]`, `partialTour[i]`) by adding the cost of the edge `(partialTour[sizeOfPartialTour], partialTour[i])` to `eNode.costOfPartialTour`. Since every tour that has this prefix must also contain an edge that leaves each of the remaining vertices, no leaf can have a cost less than `costofPartialTour` plus the sum of the costs of the cheapest edge that leaves each of the remaining vertices. We use this bound as the value of `lowerCost` of the child generated. We add this new child to the live node list (i.e., the min heap) if its `lowerCost` is less than the cost of the best tour found so far.

If the digraph contains no tour, Program 22.8 returns the value `null`. Otherwise, it returns the cost of the optimal tour. The vertex sequence corresponding to this tour is returned in the array `bestTour`.

22.2.5 Board Permutation

The solution space for the board-permutation problem (Section 21.2.5) is a permutation tree. We can perform a least-cost branch-and-bound search of this tree to find a least-density board arrangement. We use a min-priority queue, each element of which represents a live node and is of type `HeapNode`. Each object of type `HeapNode` has the fields `partial` (a board permutation); `sizeOfPartial` (boards `partial[1:sizeOfPartial]` are fixed in positions 1 through `sizeOfPartial`, respectively); `partialDensity` (density of the board arrangement `partial[1:sizeOfPartial]`, including wires going to the right of `partial[sizeOfPartial]`); and `boardsInPartialWithNet` (`boardsInPartialWithNet[j]` is the number of boards in `partial[1:sizeOfPartial]` that contain net `j`). Nodes are removed from the min heap in ascending order of their `partialDensity` value. Program 22.9 gives the branch-and-bound code.

Program 22.9 initializes the E-node to be the tree root. No board has been placed at this node. Therefore, `sizeOfPartial` = 0, `partialDensity` = 0, `boardsInPartialWithNet[i]` = 0 for $1 \leq$ `i` $\leq$ `numberOfBoards`, and `partial[1:numberOfBoards` is any permutation of the numbers 1 through `numberOfBoards`. The array `boardsWithNet` is initialized such that `boardsWithNet[i]` is the number of boards that contain net `i`. The best board permutation found so far is saved in the array `bestPermutationSoFar`, and the density is saved in `leastDensitySo-`

```
/** least-cost branch-and-bound code
  * @param board 2-D board array
  * @return density of best arrangement */
public static int leastCostBBBoards(int [][] board, int numberOfNets,
                                    int [] bestPermutation)
{
   int numberOfBoards = board.length - 1;
   MinHeap liveNodeMinHeap = new MinHeap();

   // initialize first E-node (partialDensity,
   // boardsInPartialWithNet, sizeOfPartial, partial)
   HeapNode eNode = new HeapNode(0, new int [numberOfNets + 1],
                                 0, new int [numberOfBoards + 1]);

   // set eNode.boardsInPartialWithNet[i] = number of boards
   // in partial[1:s] with net i
   // set eNode.partial[i] = i, initial permutation
   // set eNode.boardsWithNet[i] = number of boards with net i
   int [] boardsWithNet = new int [numberOfNets + 1];
   for (int i = 1; i <= numberOfBoards; i++)
   {
      eNode.partial[i] = i;
      for (int j = 1; j <= numberOfNets; j++)
         boardsWithNet[j] += board[i][j];
   }

   int leastDensitySoFar = numberOfNets + 1;
   int [] bestPermutationSoFar = null;

   do
   {// expand E-node
      if (eNode.sizeOfPartial == numberOfBoards - 1)
      {// one child only
         int localDensityAtLastBoard = 0;
         for (int j = 1; j <= numberOfNets; j++)
            localDensityAtLastBoard +=
                 board[eNode.partial[numberOfBoards]][j];
```

Program 22.9 Least-cost branch and bound for the board-permutation problem (continues)

```
      if (localDensityAtLastBoard < leastDensitySoFar)
      {// better permutation
         bestPermutationSoFar = eNode.partial;
         leastDensitySoFar = Math.max(localDensityAtLastBoard,
                                      eNode.partialDensity);
      }
   }
   else
   {// generate children of E-node
      for (int i = eNode.sizeOfPartial + 1;
           i <= numberOfBoards; i++)
      {
         HeapNode hNode = new HeapNode(0, new int
            [numberOfNets + 1], 0, new int [numberOfBoards + 1]);
         for (int j = 1; j <= numberOfNets; j++)
            // acccount for nets in new board
            hNode.boardsInPartialWithNet[j] =
                              eNode.boardsInPartialWithNet[j]
                              + board[eNode.partial[i]][j];

         int localDensityAtNewBoard = 0;
         for (int j = 1; j <= numberOfNets; j++)
            if (hNode.boardsInPartialWithNet[j] > 0 &&
              boardsWithNet[j] != hNode.boardsInPartialWithNet[j])
               localDensityAtNewBoard++;

         hNode.partialDensity = Math.max(localDensityAtNewBoard,
                                         eNode.partialDensity);
         if (hNode.partialDensity < leastDensitySoFar)
         {// may lead to better leaf
            hNode.sizeOfPartial = eNode.sizeOfPartial + 1;
            for (int j = 1; j <= numberOfBoards; j++)
               hNode.partial[j] = eNode.partial[j];
            hNode.partial[hNode.sizeOfPartial] = eNode.partial[i];
            hNode.partial[i] = eNode.partial[hNode.sizeOfPartial];
            liveNodeMinHeap.put(hNode);
         }
      }
   }
```

Program 22.9 Least-cost branch and bound for the board-permutation problem (continues)

```
      // next E-node
      eNode = (HeapNode) liveNodeMinHeap.removeMin();
   } while (eNode != null &&
            eNode.partialDensity < leastDensitySoFar);

   for (int i = 1; i <= numberOfBoards; i++)
      bestPermutation[i] = bestPermutationSoFar[i];
   return leastDensitySoFar;
}
```

Program 22.9 Least-cost branch and bound for the board-permutation problem (concluded)

`Far`. A `do-while` loop examines the E-nodes one at a time. At the end of each iteration of this loop, the next E-node is selected by extracting, from the min heap of live nodes, a node with least `partialDensity`. If this node's `partialDensity` value is $\geq$ `leastDensitySoFar`, then none of the remaining live nodes can lead to board permutations with density less than `leastDensitySoFar` and the algorithm terminates.

The `do-while` loop considers two cases for the E-node. The first arises when `sizeOfPartial` = `numberOfBoards-1`. At this time `numberOfBoards-1` boards have been placed, and the E-node is the parent of a leaf of the solution space tree. The permutation corresponding to this leaf is `partial`. Its density is computed, and `leastDensitySoFar` and `bestPermutationSoFar` are updated if necessary.

In the second case the E-node has two or more children. Each child `N` is generated, and the density `N.partialDensity` of the partial permutation (`partial[1:sizeOfPartial+1]`) corresponding to the child is computed. The child `N` is saved in the min-priority queue only if `N.partialDensity` < `leastDensitySoFar`. Notice that when `N.partialDensity` $\geq$ `leastDensitySoFar`, all leaves in its subtree have density $\geq$ `leastDensitySoFar` and do not represent board permutations better than `bestPermutationSoFar`.

EXERCISES

3. In the context of Program 22.4, define `maxWeightSoFar` to be the maximum of the weights associated with the feasible nodes generated so far. Modify Program 22.4 so that a new live node is added to the subset tree and max heap iff the live node's `upperWeight` is greater than or equal to `bestw`. You will also need to add code to initialize and update `bestw`.

4. Write a max-profit branch-and-bound code for the loading problem, using only a max-priority queue. That is, do not maintain the portion of the solution

space tree generated (as is done in Program 22.4). Each priority queue node will now contain the path to the tree root.

5. Write a max-profit branch-and-bound code for the 0/1 knapsack problem using only a max-priority queue. That is, do not maintain the portion of the solution space tree generated. Each priority queue node will now contain the path to the tree root.

6. (a) In Program 22.7 right children with `upperSize` value $\geq$ `bestn` are added to the max heap. Will the program still work correctly if only right children with `upperSize` $>$ `sizeOfMaxCliqueSoFar` are added? Why?

   (b) Does the program add left children with `upperSize` $\geq$ `sizeOfMaxClique-SoFar` to the max heap?

   (c) Modify the program so that only nodes with `upperSize` $>$ `sizeOfMax-CliqueSoFar` are added to the max heap and to the solution space subtree being constructed.

7. Consider the subset space tree for the max-clique problem. For any level `i` node `x` of the subset tree, let `minDegree(x)` be the minimum of the degrees of the vertices included at `x`.

   (a) Show that no leaf in the subtree with root `x` can represent a clique of size more than `x.upperSize` = min`x.sizeOfCliqueAtENode + n - i + 1, minDegree(x) + 1`.

   (b) Rewrite `maxProfitBBMaxClique` using this definition of `x.upperSize`.

   (c) Compare the run times as well as the number of solution space tree nodes generated by the two versions of `maxProfitBBMaxClique`.

8. Write a max-profit branch-and-bound code for the max-clique problem, using only a max-priority queue. That is, do not maintain the portion of the solution space tree generated. Each priority queue node will now contain the path to the tree root.

9. Modify Program 22.8 so that nodes with `sizeOfPartialTour` = `n-2` are not entered into the priority queue. Rather, the best permutation found so far is saved in an array `bestPermutationSofar`. The algorithm terminates when the next E-node has `lowerCost` $\geq$ `costOfBestTourSoFar`.

10. Write a version of Program 22.8 in which we use parent pointers to explicitly retain the portion of the solution space tree examined by the algorithm (as in Program 22.6) and the priority queue entries contain the fields `lowerCost`, `costOfPartialTour`, `minAdditionalCost`, and `liveNode` (pointer to corresponding node in solution space tree) only.

11. Write a FIFO branch-and-bound code for the board-permutation problem. Your code must output both the best board arrangement and its density. Use suitable test data to test the correctness of your code.

12. Write a FIFO branch-and-bound algorithm to find a board arrangement that minimizes the length of the longest net (see Exercise 17 in Chapter 21).

13. Do Exercise 12 using a least-cost branch and bound.

14. Write a least-cost branch-and-bound algorithm for the vertex-cover problem of Exercise 18 in Chapter 21.

15. Write a max-cost branch-and-bound algorithm for the simple max-cut problem of Exercise 19 in Chapter 21.

16. Write a least-cost branch-and-bound algorithm for the machine-design problem of Exercise 20 in Chapter 21.

17. Write a least-cost branch-and-bound algorithm for the network-design problem of Exercise 21 in Chapter 21.

18. Write a FIFO branch-and-bound algorithm for the n-queens-placement problem of Exercise 22 in Chapter 21.

19. Do Exercise 23 in Chapter 21 for FIFO branch and bound.

20. Do Exercise 24 in Chapter 21 for FIFO branch and bound.

21. Do Exercise 25 in Chapter 21 for FIFO branch and bound.

22. Do Exercise 23 in Chapter 21 for least-cost branch and bound.

23. Do Exercise 24 in Chapter 21 for least-cost branch and bound.

24. Do Exercise 25 in Chapter 21 for least-cost branch and bound.

25. Do Exercise 25 in Chapter 21 for arbitrary branch and bound. For this exercise you will need to pass functions to add live nodes and select the next E-node as parameters.