Chapter 8

BRANCH-AND-BOUND

8.1 THE METHOD

This chapter makes extensive use of terminology defined in Section 7.1. The reader is urged to review this section before proceeding.

The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node. We have already seen (in Section 7.1) two graph search strategies, BFS and D-search, in which the exploration of a new node cannot begin until the node currently being explored is fully explored. Both of these generalize to branch-and-bound strategies. In branch-and-bound terminology, a BFS-like state space search will be called FIFO (**F**irst **I**n **F**irst **O**ut) search as the list of live nodes is a first-in-first-out list (or queue). A D-search-like state space search will be called LIFO (**L**ast **I**n **F**irst **O**ut) search as the list of live nodes is a last-in-first-out list (or stack). As in the case of backtracking, bounding functions are used to help avoid the generation of subtrees that do not contain an answer node.

Example 8.1 [4-queens] Let us see how a FIFO branch-and-bound algorithm would search the state space tree (Figure 7.2) for the 4-queens problem. Initially, there is only one live node, node 1. This represents the case in which no queen has been placed on the chessboard. This node becomes the E-node. It is expanded and its children, nodes 2, 18, 34, and 50, are generated. These nodes represent a chessboard with queen 1 in row 1 and columns 1, 2, 3, and 4 respectively. The only live nodes now are nodes 2, 18, 34, and 50. If the nodes are generated in this order, then the next E-node is node 2. It is expanded and nodes 3, 8, and 13 are generated. Node 3 is immediately killed using the bounding function of Example 7.5. Nodes 8 and 13 are added to the queue of live nodes. Node 18 becomes the next E-node. Nodes 19, 24, and 29 are generated. Nodes 19 and 24 are killed as a result of the bounding functions. Node 29 is added to the queue of live nodes. The E-node is node 34. Figure 8.1 shows the portion of the tree of Figure 7.2 that is generated by a FIFO branch-and-bound search. Nodes that are killed as a result of the bounding functions have a "B" under them. Numbers inside the nodes correspond to the numbers in Figure 7.2. Numbers outside the nodes give the order in which the nodes are generated by FIFO branch-and-bound. At the time the answer node, node 31, is reached, the only live nodes remaining are nodes 38 and 54. A comparison of Figures 7.6 and 8.1 indicates that backtracking is a superior search method for this problem. □

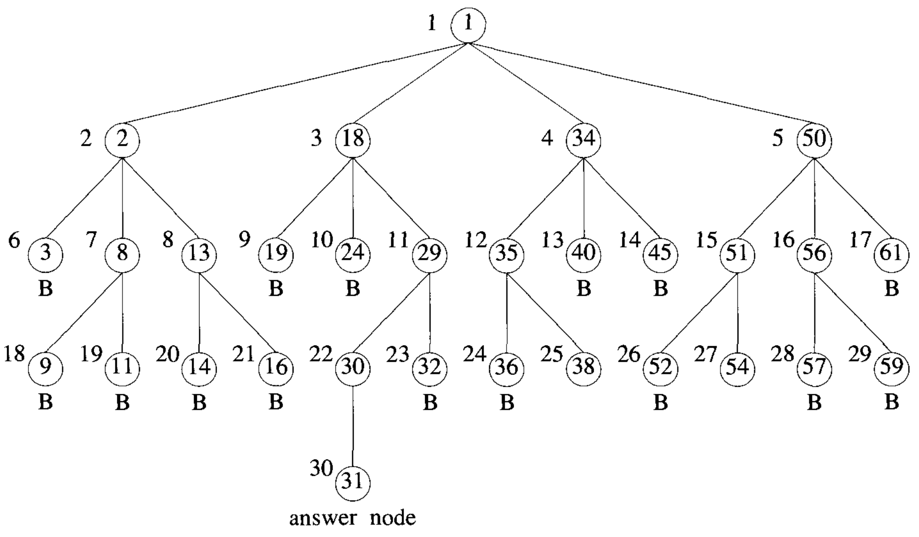


Figure 8.1 Portion of 4-queens state space tree generated by FIFO branch-and-bound

8.1.1 Least Cost (LC) Search

In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rather rigid and in a sense blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly. Thus, in Example 8.1, when node 30 is generated, it should have become obvious to the search algorithm that this node will lead to an answer node in one move. However, the rigid FIFO rule first requires the expansion of all live nodes generated before node 30 was expanded.

The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function. If in the 4-queens example we use a ranking function that assigns node 30 a better rank than all other live nodes, then node 30 will become the E-node following node 29. The remaining live nodes will never become E-nodes as the expansion of node 30 results in the generation of an answer node (node 31).

The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node. For any node x, this cost could be (1) the number of nodes in the subtree x that need to be generated before an answer node is generated or, more simply, (2) the number of levels the nearest answer node (in the subtree x) is from x. Using cost measure 2, the cost of the root of the tree of Figure 8.1 is 4 (node 31 is four levels from node 1). The costs of nodes 18 and 34, 29 and 35, and 30 and 38 are respectively 3, 2, and 1. The costs of all remaining nodes on levels 2, 3, and 4 are respectively greater than 3, 2, and 1. Using these costs as a basis to select the next E-node, the E-nodes are nodes 1, 18, 29, and 30 (in that order). The only other nodes to get generated are nodes 2, 34, 50, 19, 24, 32, and 31. It should be easy to see that if cost measure 1 is used, then the search would always generate the minimum number of nodes every branch-and-bound type algorithm must generate. If cost measure 2 is used, then the only nodes to become E-nodes are the nodes on the path from the root to the nearest answer node. The difficulty with using either of these ideal cost functions is that computing the cost of a node usually involves a search of the subtree x for an answer node. Hence, by the time the cost of a node is determined, that subtree has been searched and there is no need to explore x again. For this reason, search algorithms usually rank nodes only on the basis of an estimate $\hat{g}(\cdot)$ of their cost.

Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x. Node x is assigned a rank using a function $\hat{c}(\cdot)$ such that $\hat{c}(x) = f(h(x)) + \hat{g}(x)$, where $h(x)$ is the cost of reaching x from the root and $f(\cdot)$ is any nondecreasing function. At first, we may doubt the usefulness of using an $f(\cdot)$ other than $f(h(x)) = 0$ for all $h(x)$. We can justify such an $f(\cdot)$ on the grounds that the effort already expended in reaching the live nodes cannot be reduced and all we are concerned with now is minimizing the additional effort we spend to find an answer node. Hence, the effort already expended need not be considered.

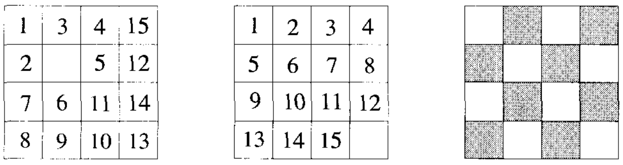
Using $f(\cdot) \equiv 0$ usually biases the search algorithm to make deep probes into the search tree. To see this, note that we would normally expect $\hat{g}(y) \leq \hat{g}(x)$ for y, a child of x. Hence, following x, y will become the E-node, then one of y's children will become the E-node, next one of y's grandchildren will become the E-node, and so on. Nodes in subtrees other than the subtree x will not get generated until the subtree x is fully searched. This would not be a cause for concern if $\hat{g}(x)$ were the true cost of x. Then, we would not wish to explore the remaining subtrees in any case (as x is guaranteed to get us to an answer node quicker than any other existing live node). However, $\hat{g}(x)$ is only an estimate of the true cost. So, it is quite possible that for two nodes w and z, $\hat{g}(w) < \hat{g}(z)$ and z is much closer to an answer node than w. It is therefore desirable not to overbias the search algorithm in favor of deep probes. By using $f(\cdot) \not\equiv 0$, we can force the search algorithm to favor a node z close to the root over a node w which is many levels below z. This would reduce the possibility of deep and fruitless searches into the tree.

A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$. Hence, such a search strategy is called an LC-search (**L**east **C**ost search). It is interesting to note that BFS and D-search are special cases of LC-search. If we use $\hat{g}(x) \equiv 0$ and $f(h(x)) =$ level of node x, then a LC-search generates nodes by levels. This is essentially the same as a BFS. If $f(h(x)) \equiv 0$ and $\hat{g}(x) \geq \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search. An LC-search coupled with bounding functions is called an LC branch-and-bound search.

In discussing LC-searches, we sometimes make reference to a cost function $c(\cdot)$ defined as follows: if x is an answer node, then $c(x)$ is the cost (level, computational difficulty, etc.) of reaching x from the root of the state space tree. If x is not an answer node, then $c(x) = \infty$ providing the subtree x contains no answer node; otherwise $c(x)$ equals the cost of a minimum-cost answer node in the subtree x. It should be easy to see that $\hat{c}(\cdot)$ with $f(h(x)) = h(x)$ is an approximation to $c(\cdot)$. From now on $c(x)$ is referred to as the cost of x.

8.1.2 The 15-puzzle: An Example

The 15-puzzle (invented by Sam Loyd in 1878) consists of 15 numbered tiles on a square frame with a capacity of 16 tiles (Figure 8.2). We are given an initial arrangement of the tiles, and the objective is to transform this arrangement into the goal arrangement of Figure 8.2(b) through a series of legal moves. The only legal moves are ones in which a tile adjacent to the empty spot (ES) is moved to ES. Thus from the initial arrangement of Figure 8.2(a), four moves are possible. We can move any one of the tiles numbered 2, 3, 5, or 6 to the empty spot. Following this move, other moves can be made. Each move creates a new arrangement of the tiles. These arrangements are called the *states* of the puzzle. The initial and goal arrangements are called the initial and goal states. A state is reachable from the initial state iff there is a sequence of legal moves from the initial state to this state. The state space of an initial state consists of all states that can be reached from the initial state. The most straightforward way to solve the puzzle would be to search the state space for the goal state and use the path from the initial state to the goal state as the answer. It is easy to see that there are 16! ($16! \approx 20.9 \times 10^{12}$) different arrangements of the tiles on the frame. Of these only one-half are reachable from any given initial state. Indeed, the state space for the problem is very large. Before attempting to search this state space for the goal state, it would be worthwhile to determine whether the goal state is reachable from the initial state. There is a very simple way to do this. Let us number the frame positions 1 to 16. Position i is the frame position containing tile numbered i in the goal arrangement of Figure 8.2(b). Position 16 is the empty spot. Let $position(i)$ be the position number in the initial state of the tile numbered i. Then $position(16)$ will denote the position of the empty spot.



(a) An arrangement (b) Goal arrangement (c)

Figure 8.2 15-puzzle arrangements

For any state let $less(i)$ be the number of tiles j such that $j < i$ and $position(j) > position(i)$. For the state of Figure 8.2(a) we have, for example, $less(1) = 0$, $less(4) = 1$, and $less(12) = 6$. Let x =1 if in the initial state the empty spot is at one of the shaded positions of Figure 8.2(c) and $x = 0$ if it is at one of the remaining positions. Then, we have the following theorem:

Theorem 8.1 The goal state of Figure 8.2(b) is reachable from the initial state iff $\sum_{i=1}^{16} less(i) + x$ is even.

Proof: Left as an exercise. □

Theorem 8.1 can be used to determine whether the goal state is in the state space of the initial state. If it is, then we can proceed to determine a sequence of moves leading to the goal state. To carry out this search, the state space can be organized into a tree. The children of each node x in this tree represent the states reachable from state x by one legal move. It is convenient to think of a move as involving a move of the empty space rather than a move of a tile. The empty space, on each move, moves either up, right, down, or left. Figure 8.3 shows the first three levels of the state space tree of the 15-puzzle beginning with the initial state shown in the root. Parts of levels 4 and 5 of the tree are also shown. The tree has been pruned a little. *No node p has a child state that is the same as p's parent.* The subtree eliminated in this way is already present in the tree and has root *parent(p)*. As can be seen, there is an answer node at level 4.

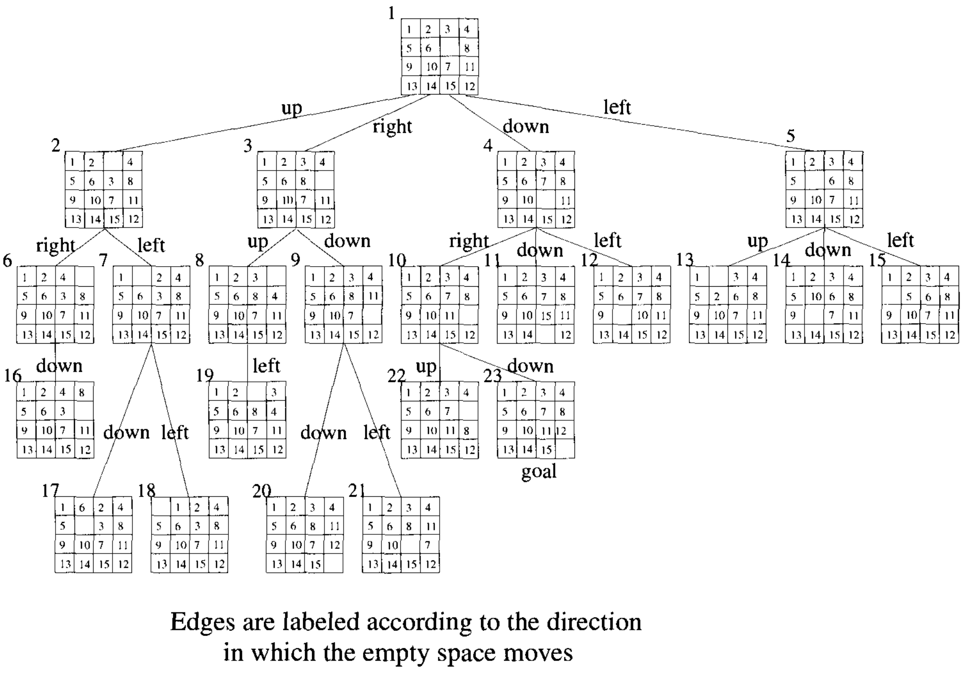

up
right
down
left
goal
Edges are labeled according to the direction
in which the empty space moves

Figure 8.3 Part of the state space tree for the 15-puzzle

A depth first state space tree generation will result in the subtree of Figure 8.4 when the next moves are attempted in the order: move the empty space up, right, down, and left. Successive board configurations reveal that each move gets us farther from the goal rather than closer. The search of the state space tree is blind. It will take the leftmost path from the root regardless of the starting configuration. As a result, an answer node may never be found (unless the leftmost path ends in such a node). In a FIFO search of the tree of Figure 8.3, the nodes will be generated in the order numbered. A breadth first search will always find a goal node nearest to the root. However, such a search is also blind in the sense that no matter what the initial configuration, the algorithm attempts to make the same sequence of moves. A FIFO search always generates the state space tree by levels.

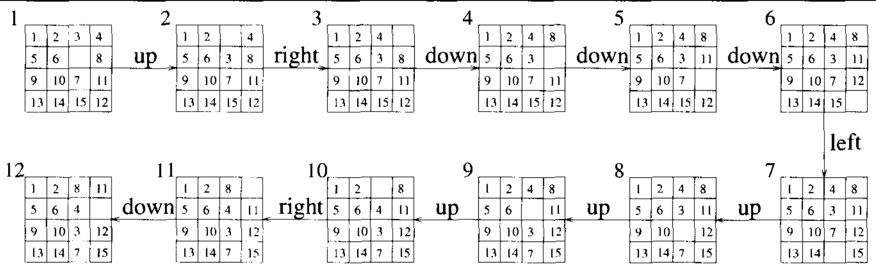


Figure 8.4 First ten steps in a depth first search

What we would like, is a more "intelligent" search method, one that seeks out an answer node and adapts the path it takes through the state space tree to the specific problem instance being solved. We can associate a cost $c(x)$ with each node x in the state space tree. The cost $c(x)$ is the length of a path from the root to a nearest goal node (if any) in the subtree with root x. Thus, in Figure 8.3, $c(1) = c(4) = c(10) = c(23) = 3$. When such a cost function is available, a very efficient search can be carried out. We begin with the root as the E-node and generate a child node with $c()$-value the same as the root. Thus children nodes 2, 3, and 5 are eliminated and only node 4 becomes a live node. This becomes the next E-node. Its first child, node 10, has $c(10) = c(4) = 3$. The remaining children are not generated. Node 4 dies and node 10 becomes the E-node. In generating node 10's children, node 22 is killed immediately as $c(22) > 3$. Node 23 is generated next. It is a goal node and the search terminates. In this search strategy, the only nodes to become E-nodes are nodes on the path from the root to a nearest goal node. Unfortunately, this is an impractical strategy as it is not possible to easily compute the function $c(\cdot)$ specified above.

We can arrive at an easy to compute estimate $\hat{c}(x)$ of $c(x)$. We can write $\hat{c}(x) = f(x) + \hat{g}(x)$, where $f(x)$ is the length of the path from the root to node x and $\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x. One possible choice for $\hat{g}(x)$ is

$$\hat{g}(x) = \text{number of nonblank tiles not in their goal position}$$

Clearly, at least $\hat{g}(x)$ moves have to be made to transform state x to a goal state. More than $\hat{g}(x)$ moves may be needed to achieve this. To see this, examine the problem state of Figure 8.5. There $\hat{g}(x) = 1$ as only tile 7 is not in its final spot (the count for $\hat{g}(x)$ excludes the blank tile). However, the number of moves needed to reach the goal state is many more than $\hat{g}(x)$. So $\hat{c}(x)$ is a *lower bound* on the value of $c(x)$.

An LC-search of Figure 8.3 using $\hat{c}(x)$ will begin by using node 1 as the E-node. All its children are generated. Node 1 dies and leaves behind the live nodes 2, 3, 4, and 5. The next node to become the E-node is a live node with least $\hat{c}(x)$. Then $\hat{c}(2) = 1+4$, $\hat{c}(3) = 1+4$, $\hat{c}(4) = 1+2$, and $\hat{c}(5) = 1+4$. Node 4 becomes the E-node. Its children are generated. The live nodes at this time are 2, 3, 5, 10, 11, and 12. So $\hat{c}(10) = 2 + 1$, $\hat{c}(11) = 2 + 3$, and $\hat{c}(12) = 2 + 3$. The live node with least $\hat{c}$ is node 10. This becomes the next E-node. Nodes 22 and 23 are generated next. Node 23 is determined to be a goal node and the search terminates. In this case LC-search was almost as efficient as using the exact function $c()$. It should be noted that with a suitable choice for $\hat{c}()$, an LC-search will be far more selective than any of the other search methods we have discussed.

1	2	3	4
5	6		8
9	10	11	12
13	14	15	7

Figure 8.5 Problem state

8.1.3 Control Abstractions for LC-Search

Let t be a state space tree and $c()$ a cost function for the nodes in t. If x is a node in t, then $c(x)$ is the minimum cost of any answer node in the subtree with root x. Thus, $c(t)$ is the cost of a minimum-cost answer node in t. As remarked earlier, it is usually not possible to find an easily computable function $c()$ as defined above. Instead, a heuristic $\hat{c}$ that estimates $c()$ is used. This heuristic should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then $c(x) = \hat{c}(x)$. LCSearch (Algorithm 8.1) uses $\hat{c}$ to find an answer node. The algorithm uses two functions Least() and Add(x) to delete and add a live node from or to the list of live nodes, respectively. Least() finds a live node with least $\hat{c}()$. This node is deleted from the list of live nodes and returned. Add(x) adds the new live node x to the list of live nodes. The list of live nodes will usually be implemented as a min-heap (Section 2.4). Algorithm LCSearch outputs the path from the answer node it finds to the root node t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When an answer node g is found, the path from

g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

```
        listnode = record {
            listnode *next, *parent; float cost;
        }

1       Algorithm LCSearch(t)
2       // Search t for an answer node.
3       {
4           if *t is an answer node then output *t and return;
5           E := t; // E-node.
6           Initialize the list of live nodes to be empty;
7           repeat
8           {
9               for each child x of E do
10              {
11                  if x is an answer node then output the path
12                      from x to t and return;
13                  Add(x); // x is a new live node.
14                  (x → parent) := E; // Pointer for path to root.
15              }
16              if there are no more live nodes then
17              {
18                  write ("No answer node"); return;
19              }
20              E := Least();
21          } until (false);
22      }
```

Algorithm 8.1 LC-search

The correctness of algorithm LCSearch is easy to establish. Variable E always points to the current E-node. By the definition of LC-search, the root node is the first E-node (line 5). Line 6 initializes the list of live nodes. At any time during the execution of LCSearch, this list contains all live nodes except the E-node. Thus, initially this list should be empty (line 6). The **for** loop of line 9 examines all the children of the E-node. If one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If a child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes (line 13) and its *parent* field set to

E (line 14). When all the children of E have been generated, E becomes a dead node and line 16 is reached. This happens only if none of E's children is an answer node. So, the search must continue further. If there are no live nodes left, then the entire state space tree has been searched and no answer nodes found. The algorithm terminates in line 18. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

From the preceding discussion, it is clear that LCSearch terminates only when either an answer node is found or the entire state space tree has been generated and searched. Thus, termination is guaranteed only for finite state space trees. Termination can also be guaranteed for infinite state space trees that have at least one answer node provided a "proper" choice for the cost function $\hat{c}()$ is made. This is the case, for example, when $\hat{c}(x) > \hat{c}(y)$ for every pair of nodes x and y such that the level number of x is "sufficiently" higher than that of y. For infinite state space trees with no answer nodes, LCSearch will not terminate. Thus, it is advisable to restrict the search to find answer nodes with a cost no more than a given bound C.

One should note the similarity between algorithm LCSearch and algorithms for a breadth first search and D-search of a state space tree. If the list of live nodes is implemented as a queue with Least() and Add(x) being algorithms to delete an element from and add an element to the queue, then LCSearch will be transformed to a FIFO search schema. If the list of live nodes is implemented as a stack with Least() and Add(x) being algorithms to delete and add elements to the stack, then LCSearch will carry out a LIFO search of the state space tree. Thus, the algorithms for LC, FIFO, and LIFO search are essentially the same. The only difference is in the implementation of the list of live nodes. This is to be expected as the three search methods differ only in the selection rule used to obtain the next E-node.

8.1.4 Bounding

A branch-and-bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost $c(x)$ associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. (Another method, heuristic search, is discussed in the exercises.) A cost function $\hat{c}(\cdot)$ such that $\hat{c}(x) \leq c(x)$ is used to provide lower bounds on solutions obtainable from any node x. If *upper* is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $\hat{c}(x) > upper$ may be killed as all answer nodes reachable from x have cost $c(x) \geq \hat{c}(x) > upper$. The starting value for *upper* can be obtained by some heuristic or can be set to ∞. Clearly, so long as the initial value for *upper* is no less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of *upper* can be updated.

Let us see how these ideas can be used to arrive at branch-and-bound algorithms for optimization problems. In this section we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function. We need to be able to formulate the search for an optimal solution as a search for a least-cost answer node in a state space tree. To do this, it is necessary to define the cost function $c(\cdot)$ such that $c(x)$ is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for $c(\cdot)$. For nodes representing feasible solutions, $c(x)$ is the value of the objective function for that feasible solution. For nodes representing infeasible solutions, $c(x) = \infty$. For nodes representing partial solutions, $c(x)$ is the cost of the minimum-cost node in the subtree with root x. Since $c(x)$ is in general as hard to compute as the original optimization problem is to solve, the branch-and-bound algorithm will use an estimate $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x. In general then, the $\hat{c}(\cdot)$ function used in a branch-and-bound solution to optimization functions will estimate the objective function value and not the computational difficulty of reaching an answer node. In addition, to be consistent with the terminology used in connection with the 15-puzzle, any node representing a feasible solution (a solution node) will be an answer node. However, only minimum-cost answer nodes will correspond to an optimal solution. Thus, answer nodes and solution nodes are indistinguishable.

As an example optimization problem, consider the job sequencing with deadlines problem introduced in Section 4.4. We generalize this problem to allow jobs with different processing times. We are given n jobs and one processor. Each job i has associated with it a three tuple (p_i, d_i, t_i). Job i requires t_i units of processing time. If its processing is not completed by the deadline d_i, then a penalty p_i is incurred. The objective is to select a subset J of the n jobs such that all jobs in J can be completed by their deadlines. Hence, a penalty can be incurred only on those jobs not in J. The subset J should be such that the penalty incurred is minimum among all possible subsets J. Such a J is optimal.

Consider the following instance: $n = 4$, $(p_1, d_1, t_1) = (5, 1, 1)$, $(p_2, d_2, t_2) = (10, 3, 2)$, $(p_3, d_3, t_3) = (6,\ 2,\ 1)$, and $(p_4, d_4, t_4) = (3,\ 1,\ 1)$. The solution space for this instance consists of all possible subsets of the job index set $\{1, 2, 3, 4\}$. The solution space can be organized into a tree by means of either of the two formulations used for the sum of subsets problem (Example 7.2). Figure 8.6 corresponds to the variable tuple size formulation while Figure 8.7 corresponds to the fixed tuple size formulation. In both figures square nodes represent infeasible subsets. In Figure 8.6 all nonsquare nodes are answer nodes. Node 9 represents an optimal solution and is the only minimum-cost answer node. For this node $J = \{2, 3\}$ and the penalty (cost) is 8. In Figure 8.7 only nonsquare leaf nodes are answer nodes. Node 25 represents the optimal solution and is also a minimum-cost answer node. This node corresponds to $J = \{2, 3\}$ and a penalty of 8. The costs of the answer nodes of Figure 8.7 are given below the nodes.

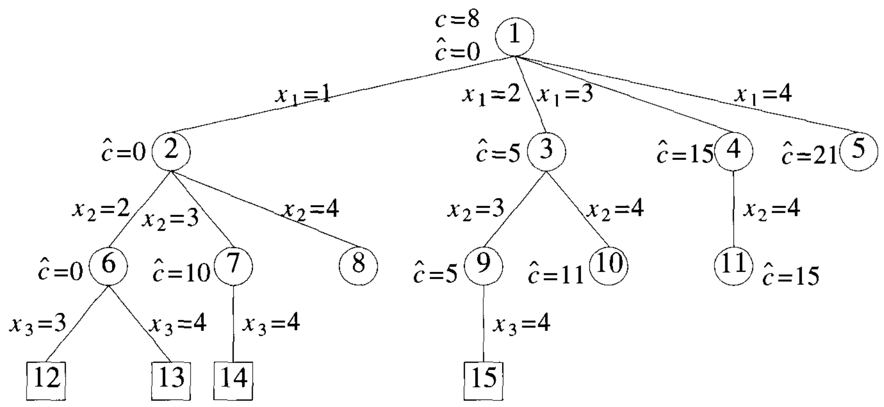


Figure 8.6 State space tree corresponding to variable tuple size formulation

We can define a cost function $c()$ for the state space formulations of Figures 8.6 and 8.7. For any circular node x, $c(x)$ is the minimum penalty corresponding to any node in the subtree with root x. The value of $c(x) = \infty$ for a square node. In the tree of Figure 8.6, $c(3) = 8$, $c(2) = 9$, and $c(1) = 8$. In the tree of Figure 8.7, $c(1) = 8$, $c(2) = 9$, $c(5) = 13$, and $c(6) = 8$. Clearly, $c(1)$ is the penalty corresponding to an optimal selection J.

A bound $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x is easy to obtain. Let S_x be the subset of jobs selected for J at node x. If $m = \max\ \{i | i \in S_x\}$, then $\hat{c}(x) = \sum_{\substack{i<m \\ i \notin S_x}} p_i$ is an estimate for $c(x)$ with the property $\hat{c}(x) \leq c(x)$. For each circular node x in Figures 8.6 and 8.7, the value of $\hat{c}(x)$ is the number outside node x. For a square node, $\hat{c}(x) = \infty$. For example, in Figure 8.6 for node 6, $S_6 = \{1, 2\}$ and hence $m = 2$. Also, $\sum_{\substack{i<2 \\ i \notin S_2}} p_i = 0$. For node 7, $S_7 = \{1, 3\}$ and $m = 3$. Therefore, $\sum_{\substack{i<2 \\ i \notin S_2}} p_i = p_2 = 10$. And so on. In Figure 8.7, node 12 corresponds to the omission of job 1 and hence a penalty of 5; node 13 corresponds to the omission of jobs 1 and 3 and hence a penalty of 11; and so on.

A simple upper bound $u(x)$ on the cost of a minimum-cost answer node in the subtree x is $u(x) = \sum_{i \notin S_x} p_i$. Note that $u(x)$ is the cost of the solution S_x corresponding to node x.

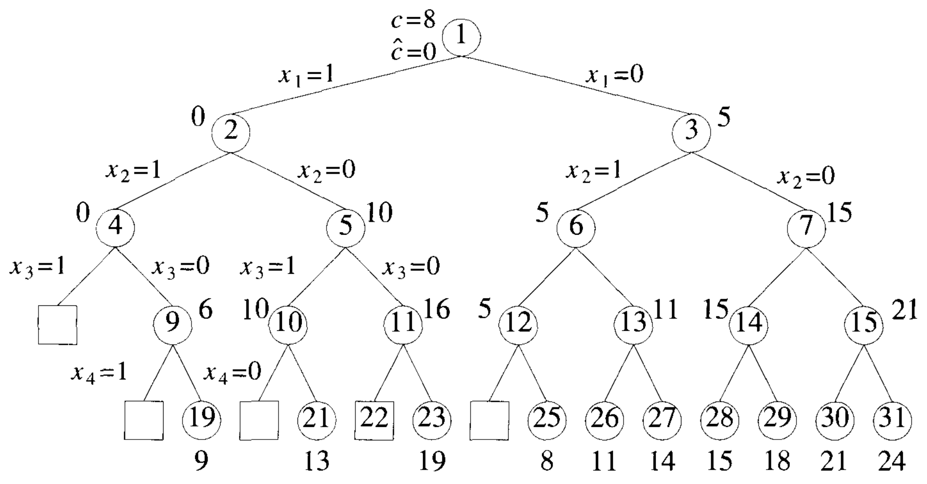


Figure 8.7 State space tree corresponding to fixed tuple size formulation

8.1.5 FIFO Branch-and-Bound

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with *upper* $= \infty$ (or *upper* $= \sum_{1 \le i \le n} p_i$) as an upper bound on the cost of a minimum-cost answer node. Starting with node 1 as the E-node and using the variable tuple size formulation of Figure 8.6, nodes 2, 3, 4, and 5 are generated (in that order). Then $u(2) = 19$, $u(3) = 14$, $u(4) = 18$, and $u(5) = 21$. For example, node 2 corresponds to the inclusion of job 1. Thus $u(2)$ is obtained by summing the penalties of all the other jobs. The variable *upper* is updated to 14 when node 3 is generated. Since $\hat{c}(4)$ and $\hat{c}(5)$ are greater than *upper*, nodes 4 and 5 get killed (or bounded). Only nodes 2 and 3 remain alive. Node 2 becomes the next E-node. Its children, nodes 6, 7, and 8 are generated. Then $u(6) = 9$ and so *upper* is updated to 9. The cost $\hat{c}(7) = 10 >$ *upper* and node 7 gets killed. Node 8 is infeasible and so it is killed. Next, node 3 becomes the E-node. Nodes 9 and 10 are now generated. Then $u(9) = 8$ and so *upper* becomes 8. The cost $\hat{c}(10) = 11 >$ *upper*, and this node is killed. The next E-node is node 6. Both its children are infeasible. Node 9's only child is also infeasible. The minimum-cost answer node is node 9. It has a cost of 8.

When implementing a FIFO branch-and-bound algorithm, it is not economical to kill live nodes with $\hat{c}(x) >$ *upper* each time *upper* is updated. This is so because live nodes are in the queue in the order in which they were generated. Hence, nodes with $\hat{c}(x) >$ *upper* are distributed in some random way in the queue. Instead, live nodes with $\hat{c}(x) > upper$ can be killed when they are about to become E-nodes.

From here on we shall refer to the FIFO-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ and $u(.)$ as FIFOBB.

8.1.6 LC Branch-and-Bound

An LC branch-and-bound search of the tree of Figure 8.6 will begin with $upper = \infty$ and node 1 as the first E-node. When node 1 is expanded, nodes 2, 3, 4, and 5 are generated in that order. As in the case of FIFOBB, *upper* is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as $\hat{c}(4) > upper$ and $\hat{c}(5) > upper$. Node 2 is the next E-node as $\hat{c}(2) = 0$ and $\hat{c}(3) = 5$. Nodes 6, 7, and 8 are generated and *upper* is updated to 9 when node 6 is generated. So, node 7 is killed as $\hat{c}(7) = 10 > upper$. Node 8 is infeasible and so killed. The only live nodes now are nodes 3 and 6. Node 6 is the next E-node as $\hat{c}(6) = 0 < \hat{c}(3)$. Both its children are infeasible. Node 3 becomes the next E-node. When node 9 is generated, *upper* is updated to 8 as $u(9) = 8$. So, node 10 with $\hat{c}(10) = 11$ is killed on generation. Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answer node.

From here on we refer to the LC(LIFO)-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ and $u(.)$ as LCBB (LIFOBB).

EXERCISES

1. Prove Theorem 8.1.

2. Present an algorithm schema FifoBB for a FIFO branch-and-bound search for a least-cost answer node.

3. Give an algorithm schema LcBB for a LC branch-and-bound search for a least-cost answer node.

4. Write an algorithm schema LifoBB, for a LIFO branch-and-bound search for a least-cost answer node.

5. Draw the portion of the state space tree generated by FIFOBB, LCBB, and LIFOBB for the job sequencing with deadlines instance $n = 5$, $(p_1, p_2, \ldots, p_5) = (6, 3, 4, 8, 5)$, $(t_1, t_2, \ldots, t_5) = (2, 1, 2, 1, 1)$, and $(d_1, d_2, \ldots, d_5) = (3, 1, 4, 2, 4)$. What is the penalty corresponding to an optimal solution? Use a variable tuple size formulation and $\hat{c}(\cdot)$ and $u(\cdot)$ as in Section 8.1.

6. Write a branch-and-bound algorithm for the job sequencing with deadlines problem. Use the fixed tuple size formulation.

7. (a) Write a branch-and-bound algorithm for the job sequencing with deadlines problem using a dominance rule (see Section 5.7). Your algorithm should work with a fixed tuple size formulation and should generate nodes by levels. Nodes on each level should be kept in an order permitting easy use of your dominance rule.

   (b) Convert your algorithm into a program and, using randomly generated problem instances, determine the worth of the dominance rule as well as the bounding functions. To do this, you will have to run four versions of your program: ProgA$\cdots$ bounding functions and dominance rules are removed, ProgB$\cdots$ dominance rule is removed, ProgC$\cdots$ bounding function is removed, and ProgD$\cdots$ bounding functions and dominance rules are included. Determine computing time figures as well as the number of nodes generated.

8.2 0/1 KNAPSACK PROBLEM

To use the branch-and-bound technique to solve any problem, it is first necessary to conceive of a state space tree for the problem. We have already seen two possible state space tree organizations for the knapsack problem (Section 7.6). Still, we cannot directly apply the techniques of Section 8.1 since these were discussed with respect to minimization problems whereas the knapsack problem is a maximization problem. This difficulty is easily overcome by replacing the objective function $\sum p_i x_i$ by the function $-\sum p_i x_i$. Clearly, $\sum p_i x_i$ is maximized iff $-\sum p_i x_i$ is minimized. This modified knapsack problem is stated as (8.1).

$$\text{minimize } -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \leq m \tag{8.1}$$

$$x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

We continue the discussion assuming a fixed tuple size formulation for the solution space. The discussion is easily extended to the variable tuple size formulation. Every leaf node in the state space tree representing an assignment for which $\sum_{1\leq i\leq n} w_i x_i \leq m$ is an answer (or solution) node. All other leaf nodes are infeasible. For a minimum-cost answer node to correspond to any optimal solution, we need to define $c(x) = -\sum_{1\leq i\leq n} p_i x_i$ for every answer node x. The cost $c(x) = \infty$ for infeasible leaf nodes. For nonleaf nodes, $c(x)$ is recursively defined to be min $\{c(lchild(x)),\ c(rchild(x))\}$.

We now need two functions $\hat{c}(x)$ and $u(x)$ such that $\hat{c}(x) \leq c(x) \leq u(x)$ for every node x. The cost $\hat{c}(\cdot)$ and $u(\cdot)$ satisfying this requirement may be obtained as follows. Let x be a node at level j, $1 \leq j \leq n+1$. At node x assignments have already been made to x_i, $1 \leq i < j$. The cost of these assignments is $-\sum_{1 \leq i < j} p_i x_i$. So, $c(x) \leq -\sum_{1 \leq i < j} p_i x_i$ and we may use $u(x) = -\sum_{1 \leq i < j} p_i x_i$. If $q = -\sum_{1 \leq i < j} p_i x_i$, then an improved upper bound function $u(x)$ is $u(x) = \mathsf{UBound}(q, \sum_{1 \leq i < j} w_i x_i, j-1, m)$, where UBound is defined in Algorithm 8.2. As for $c(x)$, it is clear that $\mathsf{Bound}(-q, \sum_{1 \leq i < j} w_i x_i, j-1) \leq c(x)$, where Bound is as given in Algorithm 7.11.

```
Algorithm UBound($cp, cw, k, m$)
// $cp, cw, k$, and $m$ have the same meanings as in
// Algorithm 7.11. $w[i]$ and $p[i]$ are respectively
// the weight and profit of the $i$th object.
{
    $b := cp$; $c := cw$;
    for $i := k+1$ to $n$ do
    {
        if $(c + w[i] \leq m)$ then
        {
            $c := c + w[i]$; $b := b - p[i]$;
        }
    }
    return $b$;
}
```

Algorithm 8.2 Function $u(\cdot)$ for knapsack problem

8.2.1 LC Branch-and-Bound Solution

Example 8.2 [LCBB] Consider the knapsack instance $n = 4$, $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$, $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$, and $m = 15$. Let us trace the working of an LC branch-and-bound search using $\hat{c}(\cdot)$ and $u(\cdot)$ as defined previously. We continue to use the fixed tuple size formulation. The search begins with the root as the E-node. For this node, node 1 of Figure 8.8, we have $\hat{c}(1) = -38$ and $u(1) = -32$.

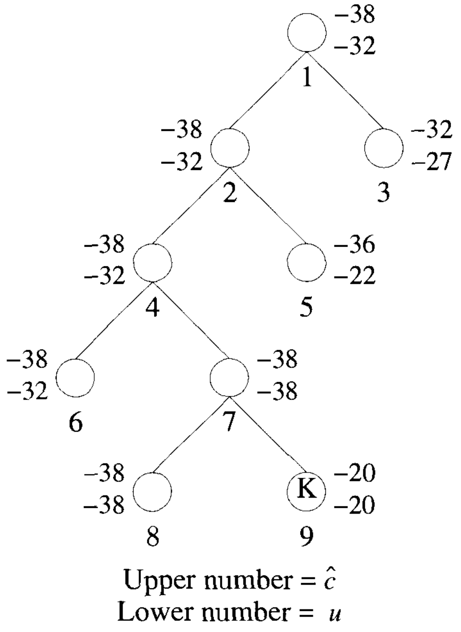


Figure 8.8 LC branch-and-bound tree for Example 8.2

The computation of $u(1)$ and $\hat{c}(1)$ is done as follows. The bound $u(1)$ has a value $\mathsf{UBound}(0, 0, 0, 15)$. UBound scans through the objects from left to right starting from j; it adds these objects into the knapsack until the first object that doesn't fit is encountered. At this time, the negation of the total profit of all the objects in the knapsack plus cw is returned. In Function UBound, c and b start with a value of zero. For $i = 1, 2$, and 3, c gets incremented by $2, 4$, and 6, respectively. The variable b also gets decremented by $10, 10$, and 12, respectively. When $i = 4$, the test $(c + w[i] \leq m)$ fails and hence the value returned is -32. Function Bound is similar to UBound, except that it also considers a fraction of the first object that doesn't fit the knapsack. For example, in computing $\hat{c}(1)$, the first object that doesn't fit is 4 whose weight is 9. The total weight of the objects 1, 2, and 3 is 12. So, Bound considers a fraction $\frac{3}{9}$ of the object 4 and hence returns $-32 - \frac{3}{9} * 18 = -38$.

Since node 1 is not a solution node, LCBB sets $ans = 0$ and $upper = -32$ (ans being a variable to store intermediate answer nodes). The E-node is expanded and its two children, nodes 2 and 3, generated. The cost $\hat{c}(2) = -38$, $\hat{c}(3) = -32$, $u(2) = -32$, and $u(3) = -27$. Both nodes are put onto the list of live nodes. Node 2 is the next E-node. It is expanded and nodes 4 and 5 generated. Both nodes get added to the list of live nodes. Node 4 is the live node with least $\hat{c}$ value and becomes the next E-node. Nodes 6 and 7 are generated. Assuming node 6 is generated first, it is added to the list of live nodes. Next, node 7 joins this list and $upper$ is updated to -38. The next E-node will be one of nodes 6 and 7. Let us assume it is node 7. Its two children are nodes 8 and 9. Node 8 is a solution node. Then $upper$ is updated to -38 and node 8 is put onto the live nodes list. Node 9 has $\hat{c}(9) > upper$ and is killed immediately. Nodes 6 and 8 are two live nodes with least $\hat{c}$. Regardless of which becomes the next E-node, $\hat{c}(E) \geq upper$ and the search terminates with node 8 the answer node. At this time, the value -38 together with the path 8, 7, 4, 2, 1 is printed out and the algorithm terminates. From the path one cannot figure out the assignment of values to the x_i's such that $\sum p_i x_i = upper$. Hence, a proper implementation of LCBB has to keep additional information from which the values of the x_i's can be extracted. One way is to associate with each node a one bit field, tag. The sequence of tag bits from the answer node to the root give the x_i values. Thus, we have $tag(2) = tag(4) = tag(6) = tag(8) = 1$ and $tag(3) = tag(5) = tag(7) = tag(9) = 0$. The tag sequence for the path 8, 7, 4, 2, 1 is 1 0 1 1 and so $x_4 = 1, x_3 = 0, x_2 = 1$, and $x_1 = 1$. □

To use LCBB to solve the knapsack problem, we need to specify (1) the structure of nodes in the state space tree being searched, (2) how to generate the children of a given node, (3) how to recognize a solution node, and (4) a representation of the list of live nodes and a mechanism for adding a node into the list as well as identifying the least-cost node. The node structure needed depends on which of the two formulations for the state space tree is being used. Let us continue with a fixed size tuple formulation. Each node

x that is generated and put onto the list of live nodes must have a *parent* field. In addition, as noted in Example 8.2, each node should have a one bit *tag* field. This field is needed to output the x_i values corresponding to an optimal solution. To generate x's children, we need to know the level of node x in the state space tree. For this we shall use a field *level*. The left child of x is chosen by setting $x_{level(x)} = 1$ and the right child by setting $x_{level(x)} = 0$. To determine the feasibility of the left child, we need to know the amount of knapsack space available at node x. This can be determined either by following the path from node x to the root or by explicitly retaining this value in the node. Say we choose to retain this value in a field *cu* (capacity unused). The evaluation of $\hat{c}(x)$ and $u(x)$ requires knowledge of the profit $\sum_{1 \le i < level(x)} p_i x_i$ earned by the filling corresponding to node x. This can be computed by following the path from x to the root. Alternatively, this value can be explicitly retained in a field *pe*. Finally, in order to determine the live node with least $\hat{c}$ value or to insert nodes properly into the list of live nodes, we need to know $\hat{c}(x)$. Again, we have a choice. The value $\hat{c}(x)$ may be stored explicitly in a field *ub* or may be computed when needed. Assuming all information is kept explicitly, we need nodes with six fields each: *parent*, *level*, *tag*, *cu*, *pe*, and *ub*.

Using this six-field node structure, the children of any live node x can be easily determined. The left child y is feasible iff $cu(x) \ge w_{level(x)}$. In this case, $parent(y) = x$, $level(y) = level(x) + 1$, $cu(y) = cu(x) - w_{level(x)}$, $pe(y) = pe(x) + p_{level(x)}$, $tag(y) = 1$, and $ub(y) = ub(x)$. The right child can be generated similarly. Solution nodes are easily recognized too. Node x is a solution node iff $level(x) = n + 1$.

We are now left with the task of specifying the representation of the list of live nodes. The functions we wish to perform on this list are (1) test if the list is empty, (2) add nodes, and (3) delete a node with least *ub*. We have seen a data structure that allows us to perform these three functions efficiently: a min-heap. If there are m live nodes, then function (1) can be carried out in $\Theta(1)$ time, whereas functions (2) and (3) require only $O(\log m)$ time.

8.2.2 FIFO Branch-and-Bound Solution

Example 8.3 Now, let us trace through the FIFOBB algorithm using the same knapsack instance as in Example 8.2. Initially the root node, node 1 of Figure 8.9, is the E-node and the queue of live nodes is empty. Since this is not a solution node, *upper* is initialized to $u(1) = -32$.

We assume the children of a node are generated left to right. Nodes 2 and 3 are generated and added to the queue (in that order). The value of *upper* remains unchanged. Node 2 becomes the next E-node. Its children, nodes 4 and 5, are generated and added to the queue. Node 3, the next

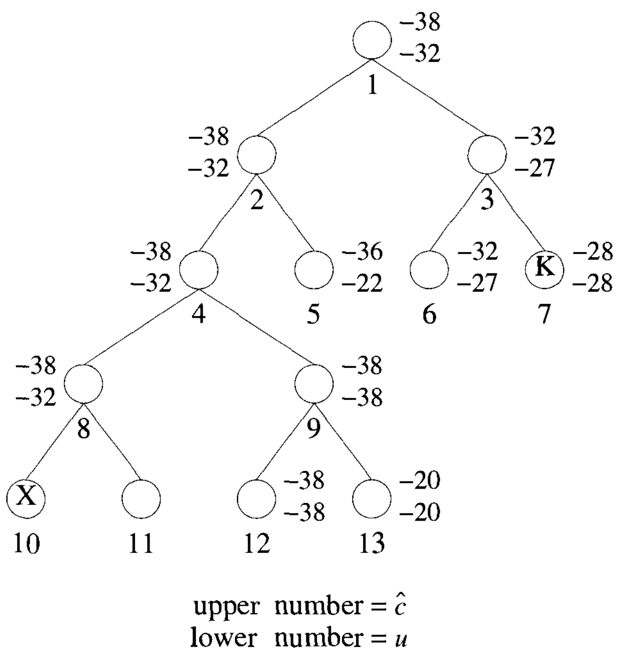


Figure 8.9 FIFO branch-and-bound tree for Example 8.3

E-node, is expanded. Its children nodes are generated. Node 6 gets added to the queue. Node 7 is immediately killed as $\hat{c}(7) > upper$. Node 4 is expanded next. Nodes 8 and 9 are generated and added to the queue. Then *upper* is updated to $u(9) = -38$. Nodes 5 and 6 are the next two nodes to become E-nodes. Neither is expanded as for each, $\hat{c}() > upper$. Node 8 is the next E-node. Nodes 10 and 11 are generated. Node 10 is infeasible and so killed. Node 11 has $\hat{c}(11) > upper$ and so is also killed. Node 9 is expanded next. When node 12 is generated, *upper* and *ans* are updated to -38 and 12 respectively. Node 12 joins the queue of live nodes. Node 13 is killed before it can get onto the queue of live nodes as $\hat{c}(13) > upper$. The only remaining live node is node 12. It has no children and the search terminates. The value of *upper* and the path from node 12 to the root is output. As in the case of Example 8.2, additional information is needed to determine the x_i values on this path. □

At first we may be tempted to discard FIFOBB in favor of LCBB in solving knapsack. Our intuition leads us to believe that LCBB will examine fewer nodes in its quest for an optimal solution. However, we should keep in mind that insertions into and deletions form a heap are far more expensive (proportional to the logarithm of the heap size) than the corresponding operations on a queue ($\Theta(1)$). Consequently, the work done for each E-node is more in LCBB than in FIFOBB. Unless LCBB uses far fewer E-nodes than FIFOBB, FIFOBB will outperform (in terms of real computation time) LCBB.

We have now seen four different approaches to solving the knapsack problem: dynamic programming, backtracking, LCBB, and FIFOBB. If we compare the dynamic programming algorithm DKnap (Algorithm 5.7) and FIFOBB, we see that there is a correspondence between generating the $S^{(i)}$'s and generating nodes by levels. $S^{(i)}$ contains all pairs (P, W) corresponding to nodes on level $i+1$, $0 \leq i \leq n$. Hence, both algorithms generate the state space tree by levels. The dynamic programming algorithm, however, keeps the nodes on each level ordered by their profit earned (P) and capacity used (W) values. No two tuples have the same P or W value. In FIFOBB we may have many nodes on the same level with the same P or W value. It is not easy to implement the dominance rule of Section 5.7 into FIFOBB as nodes on a level are not ordered by their P or W values. However, the bounding rules can easily be incorporated into DKnap. Toward the end of Section 5.7 we discussed some simple heuristics to determine whether a pair $(P,\ W) \in S^{(i)}$ should be killed. These heuristics are readily seen to be bounding functions of the type discussed here. Let the algorithm resulting from the inclusion of the bounding functions into DKnap be DKnap1. DKnap1 is expected to be superior to FIFOBB as it uses the dominance rule in addition to the bounding functions. In addition, the overhead incurred each time a node is generated is less.

To determine which of the knapsack algorithms is best, it is necessary to program them and obtain real computing times for different data sets. Since the effectiveness of the bounding functions and the dominance rule is highly data dependent, we expect a wide variation in the computing time for different problem instances having the same number of objects n. To get representative times, it is necessary to generate many problem instances for a fixed n and obtain computing times for these instances. The generation of these data sets and the problem of conducting the tests is discussed in a programming project at the end of this section. The results of some tests can be found in the references to this chapter.

Before closing our discussion of the knapsack problem, we briefly discuss a very effective heuristic to reduce a knapsack instance with large n to an equivalent one with smaller n. This heuristic, Reduce, uses some of the ideas developed for the branch-and-bound algorithm. It classifies the objects $\{1, 2, \ldots, n\}$ into one of three categories $I1, I2$, and $I3$. $I1$ is a set of objects for which x_i must be 1 in every optimal solution. $I2$ is a set for which x_i must be 0. $I3$ is $\{1, 2, \ldots, n\} - I1 - I2$. Once $I1$, $I2$, and $I3$ have been determined, we need to solve only the reduced knapsack instance

$$\text{maximize} \sum_{i \in I3} p_i x_i$$

$$\text{subject to} \sum_{i \in I3} w_i x_i \le m - \sum_{i \in I1} w_i x_i \qquad (8.2)$$

$$x_i = 0 \text{ or } 1$$

From the solution to (8.2) an optimal solution to the original knapsack instance is obtained by setting $x_i = 1$ if $i \in I1$ and $x_i = 0$ if $i \in I2$.

Function Reduce (Algorithm 8.3) makes use of two functions Ubb and Lbb. The bound Ubb($I1$, $I2$) is an upper bound on the value of an optimal solution to the given knapsack instance with added constraints $x_i = 1$ if $i \in I1$ and $x_i = 0$ if $i \in I2$. The bound Lbb($I1$, $I2$) is a lower bound under the constraints of $I1$ and $I2$. Algorithm Reduce needs no further explanation. It should be clear that $I1$ and $I2$ are such that from an optimal solution to (8.2), we can easily obtain an optimal solution to the original knapsack problem.

The time complexity of Reduce is $O(n^2)$. Because the reduction procedure is very much like the heuristics used in DKnap1 and the knapsack algorithms of this chapter, the use of Reduce does not decrease the overall computing time by as much as may be expected by the reduction in the number of objects. These algorithms do dynamically what Reduce does. The exercises explore the value of Reduce further.

```
Algorithm Reduce($p, w, n, m, I1, I2$)
// Variables are as described in the discussion.
// $p[i]/w[i] \geq p[i+1]/w[i+1],\ 1 \leq i < n$.
{
    $I1 := I2 := \emptyset$;
    $q :=$ Lbb($\emptyset, \emptyset$);
    $k :=$ largest $j$ such that $w[1] + \cdots + w[j] < m$;
    for $i := 1$ to $k$ do
    {
        if (Ubb($\emptyset, \{i\}$) $< q$) then $I1 := I1 \cup \{i\}$;
        else if (Lbb($\emptyset, \{i\}$) $> q$) then $q :=$ Lbb($\emptyset, \{i\}$);
    }
    for $i := k + 1$ to $n$ do
    {
        if (Ubb($\{i\}, \emptyset$) $< q$) then $I2 := I2 \cup \{i\}$;
        else if (Lbb($\{i\}, \emptyset$) $> q$) then $q :=$ Lbb($\{i\}, \emptyset$);
    }
}
```

Algorithm 8.3 Reduction pseudocode for knapsack problem

EXERCISES

1. Work out Example 8.2 using the variable tuple size formulation.
2. Work out Example 8.3 using the variable tuple size formulation.
3. Draw the portion of the state space tree generated by LCBB for the following knapsack instances:

   (a) $n = 5$, $(p_1, p_2, \ldots, p_5) = (10, 15, 6, 8, 4)$, $(w_1, w_2, \ldots, w_5) = (4, 6, 3, 4, 2)$, and $m = 12$.
   (b) $n = 5$, $(p_1, p_2, p_3, p_4, p_5) = (w_1, w_2, w_3, w_4, w_5) = (4, 4, 5, 8, 9)$ and $m = 15$.

4. Do Exercise 3 using LCBB on a dynamic state space tree (see Section 7.6). Use the fixed tuple size formulation.
5. Write a LCBB algorithm for the knapsack problem using the ideas given in Example 8.2.
6. Write a LCBB algorithm for the knapsack problem using the fixed tuple size formulation and the dynamic state space tree of Section 7.6.
7. [Programming project] Program the algorithms DKnap (Algorithm 5.7), DKnap1 (page 399), LCBB for knapsack, and Bknap (Algorithm 7.12). Compare these programs empirically using randomly generated data as below:

   (a) Random w_i and p_i, $w_i \in [1, 100]$, $p_i \in [1, 100]$, and $m = \sum_1^n w_i/2$.
   (b) Random w_i and p_i, $w_i \in [1, 100]$, $p_i \in [1, 100]$, and $m = 2 \max \{w_i\}$.
   (c) Random w_i, $w_i \in [1, 100]$, $p_i = w_i + 10$, and $m = \sum_1^n w_i/2$.
   (d) Same as (c) except $m = 2 \max \{w_i\}$.
   (e) Random p_i, $p_i \in [1, 100]$, $w_i = p_i + 10$, and $m = \sum_1^n w_i/2$.
   (f) Same as (e) except $m = 2 \max \{w_i\}$.

   Obtain computing times for $n = 5, 10, 20, 30, 40, \ldots$. For each n, generate (say) ten problem instances from each of the above data sets. Report average and worst-case computing times for each of the above data sets. From these times can you say anything about the expected behavior of these algorithms?

   Now, generate problem instances with $p_i = w_i$, $1 \le i \le n$, $m = \sum w_i/2$, and $\sum w_i x_i \ne m$ for any 0, 1 assignment to the x_i's. Obtain computing times for your four programs for $n = 10$, 20, and 30. Now study the effect of changing the range to [1, 1000] in data sets (a) through (f). In sets (c) to (f) replace $p_i = w_i + 10$ by $p_i = w_i + 100$ and $w_i = p_i + 10$ by $w_i = p_i + 100$.

8. [Programming project]

   (a) Program the reduction heuristic Reduce of Section 8.2. Generate several problem instances from the data sets of Exercise 7 and determine the size of the reduced problem instances. Use $n =$ 100, 200, 500, and 1000.

   (b) Program DKnap and the backtracking algorithm Bknap for the knapsack problem. Compare the effectiveness of Reduce by running several problem instances (as in Exercise 7). Obtain average and worst-case computing times for DKnap and Bknap for the generated problem instances and also for the reduced instances. To the times for the reduced problem instances, add the time required by Reduce. What conclusion can you draw from your experiments?

8.3 TRAVELING SALESPERSON (*)

An $O(n^2 2^n)$ dynamic programming algorithm for the traveling salesperson problem was arrived at in Section 5.9. We now investigate branch-and-bound algorithms for this problem. Although the worst-case complexity of these algorithms will not be any better than $O(n^2 2^n)$, the use of good bounding functions will enable these branch-and-bound algorithms to solve some problem instances in much less time than required by the dynamic programming algorithm.

Let $G = (V, E)$ be a directed graph defining an instance of the traveling salesperson problem. Let c_{ij} equal the cost of edge $\langle i, j \rangle$, $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$, and let $|V| = n$. Without loss of generality, we can assume that every tour starts and ends at vertex 1. So, the solution space S is given by $S = \{1, \pi, 1 | \pi$ is a permutation of $(2, 3, \ldots, n)\}$. Then $|S| = (n-1)!$. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots, i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1} \rangle \in E$, $0 \leq j \leq n-1$, and $i_0 = i_n = 1$. S can be organized into a state space tree similar to that for the n-queens problem (see Figure 7.2). Figure 8.10 shows the tree organization for the case of a complete graph with $|V| = 4$. Each leaf node L is a solution node and represents the tour defined by the path from the root to L. Node 14 represents the tour $i_0 = 1, i_1 = 3, i_2 = 4, i_3 = 2,$ and $i_4 = 1$.

To use LCBB to search the traveling salesperson state space tree, we need to define a cost function $c(\cdot)$ and two other functions $\hat{c}(\cdot)$ and $u(\cdot)$ such that $\hat{c}(r) \leq c(r) \leq u(r)$ for all nodes r. The cost $c(\cdot)$ is such that the solution node with least $c(\cdot)$ corresponds to a shortest tour in G. One choice for $c(\cdot)$ is

$$c(A) = \begin{cases} \text{length of tour defined by the path from the root to } A, & \text{if } A \text{ is a leaf} \\ \text{cost of a minimum-cost leaf in the subtree } A, & \text{if } A \text{ is not a leaf} \end{cases}$$

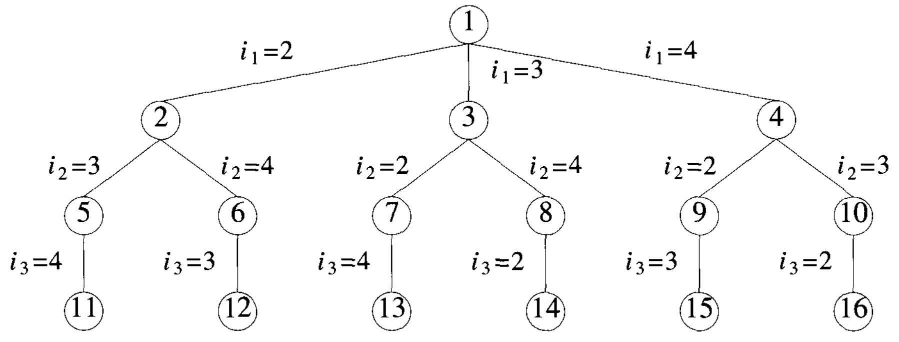


Figure 8.10 State space tree for the traveling salesperson problem with $n = 4$ and $i_0 = i_4 = 1$

A simple $\hat{c}(\cdot)$ such that $\hat{c}(A) \leq c(A)$ for all A is obtained by defining $\hat{c}(A)$ to be the length of the path defined at node A. For example, the path defined at node 6 of Figure 8.10 is $i_0, i_1, i_2 = 1, 2, 4$. It consists of the edges $\langle 1, 2\rangle$ and $\langle 2, 4\rangle$. A better $\hat{c}(\cdot)$ can be obtained by using the reduced cost matrix corresponding to G. A row (column) is said to be *reduced* iff it contains at least one zero and all remaining entries are non-negative. A matrix is *reduced* iff every row and column is reduced. As an example of how to reduce the cost matrix of a given graph G, consider the matrix of Figure 8.11(a). This corresponds to a graph with five vertices. Since every tour on this graph includes exactly one edge $\langle i, j\rangle$ with $i = k$, $1 \leq k \leq 5$, and exactly one edge $\langle i, j\rangle$ with $j = k$, $1 \leq k \leq 5$, subtracting a constant t from every entry in one column or one row of the cost matrix reduces the length of every tour by exactly t. A minimum-cost tour remains a minimum-cost tour following this subtraction operation. If t is chosen to be the minimum entry in row i (column j), then subtracting it from all entries in row i (column j) introduces a zero into row i (column j). Repeating this as often as needed, the cost matrix can be reduced. The total amount subtracted from the columns and rows is a lower bound on the length of a minimum-cost tour and can be used as the $\hat{c}$ value for the root of the state space tree. Subtracting 10, 2, 2, 3, 4, 1, and 3 from rows 1, 2, 3, 4, and 5 and columns 1 and 3 respectively of the matrix of Figure 8.11(a) yields the reduced matrix of Figure 8.11(b). The total amount subtracted is 25. Hence, all tours in the original graph have a length at least 25.

We can associate a reduced cost matrix with every node in the traveling salesperson state space tree. Let A be the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge $\langle i, j \rangle$ in the tour. If S is not a leaf, then the reduced cost matrix for S may be obtained as follows: (1) Change all entries in row i and column j of A to ∞. This prevents the use of any more edges leaving vertex i or entering vertex j. (2) Set $A(j, 1)$ to ∞. This prevents the use of edge $\langle j, 1 \rangle$. (3) Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞. Let the resulting matrix be B. Steps (1) and (2) are valid as no tour in the subtree s can contain edges of the type $\langle i, k \rangle$ or $\langle k, j \rangle$ or $\langle j, 1 \rangle$ (except for edge $\langle i, j \rangle$). If r is the total amount subtracted in step (3) then $\hat{c}(S) = \hat{c}(R) + A(i, j) + r$. For leaf nodes, $\hat{c}(\cdot) = c()$ is easily computed as each leaf defines a unique tour. For the upper bound function u, we can use $u(R) = \infty$ for all nodes R.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost matrix

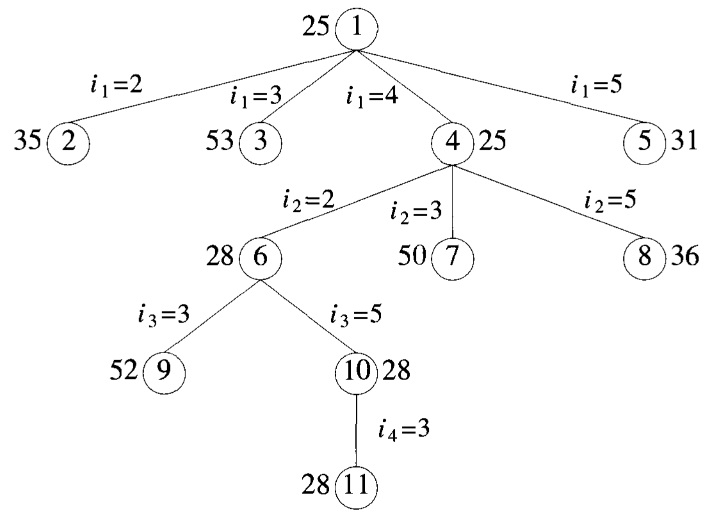
$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced cost matrix
L = 25

Figure 8.11 An example

Let us now trace the progress of the LCBB algorithm on the problem instance of Figure 8.11(a). We use $\hat{c}$ and u as above. The initial reduced matrix is that of Figure 8.11(b) and *upper* $= \infty$. The portion of the state space tree that gets generated is shown in Figure 8.12. Starting with the root node as the E-node, nodes 2, 3, 4, and 5 are generated (in that order). The reduced matrices corresponding to these nodes are shown in Figure 8.13. The matrix of Figure 8.13(b) is obtained from that of 8.11(b) by (1) setting all entries in row 1 and column 3 to ∞, (2) setting the element at position (3, 1) to ∞, and (3) reducing column 1 by subtracting by 11. The $\hat{c}$ for node 3 is therefore 25 + 17 (the cost of edge $\langle 1, 3 \rangle$ in the reduced matrix) + 11 = 53. The matrices and $\hat{c}$ value for nodes 2, 4, and 5 are obtained similarly. The value of *upper* is unchanged and node 4 becomes the next E-node. Its children 6, 7, and 8 are generated. The live nodes at this time are nodes 2, 3, 5, 6, 7, and 8. Node 6 has least $\hat{c}$ value and becomes the next E-node. Nodes 9 and 10 are generated. Node 10 is the next E-node. The solution node, node 11, is generated. The tour length for this node is $\hat{c}(11) = 28$ and *upper* is updated to 28. For the next E-node, node 5, $\hat{c}(5) = 31 >$ *upper*. Hence, LCBB terminates with 1, 4, 2, 5, 3, 1 as the shortest length tour.

An exercise examines the implementation considerations for the LCBB algorithm. A different LCBB algorithm can be arrived at by considering



Numbers outside the node are $\hat{c}$ values

Figure 8.12 State space tree generated by procedure LCBB

a different tree organization for the solution space. This organization is reached by regarding a tour as a collection of n edges. If $G = (V, E)$ has e edges, then every tour contains exactly n of the e edges. However, for each $i, 1 \leq i \leq n$, there is exactly one edge of the form $\langle i, j \rangle$ and one of the form $\langle k, i \rangle$ in every tour. A possible organization for the state space is a binary tree in which a left branch represents the inclusion of a particular edge while the right branch represents the exclusion of that edge. Figure 8.14(b) and (c) represents the first two levels of two possible state space trees for the three vertex graph of Figure 8.14(a). As is true of all problems, many state space trees are possible for a given problem formulation. Different trees differ in the order in which decisions are made. Thus, in Figure 8.14(c) we first decide the fate of edge $\langle 1, 2 \rangle$. Rather than use a static state space tree, we now consider a dynamic state space tree (see Section 7.1). This is also a binary tree. However, the order in which edges are considered depends on the particular problem instance being solved. We compute $\hat{c}$ in the same way as we did using the earlier state space tree formulation.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

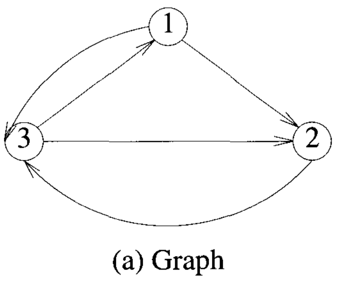
(h) Path 1,4,2,3; node 9

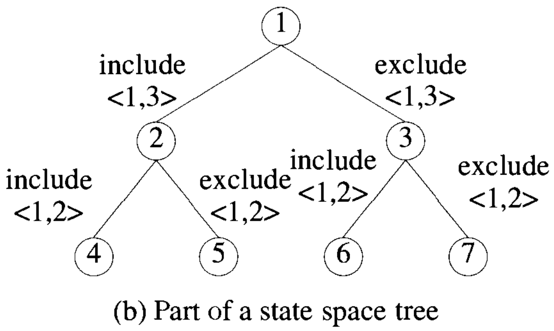
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i) Path 1,4,2,5; node 10

Figure 8.13 Reduced cost matrices corresponding to nodes in Figure 8.12

As an example of how LCBB would work on the dynamic binary tree formulation, consider the cost matrix of Figure 8.11(a). Since a total of 25





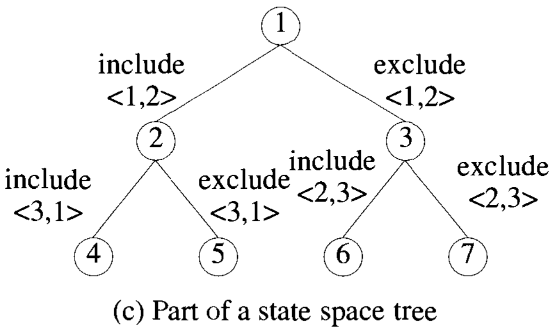


Figure 8.14 An example

needs to be subtracted form the rows and columns of this matrix to obtain the reduced matrix of Figure 8.11(b), all tours have a length at least 25. This fact is represented by the root of the state space tree of Figure 8.15. Now, we must decide which edge to use to partition the solution space into two subsets. If edge $\langle i, j \rangle$ is used, then the left subtree of the root represents all tours including edge $\langle i, j \rangle$ and the right subtree represents all tours that do not include edge $\langle i, j \rangle$. If an optimal tour is included in the left subtree, then only $n - 1$ edges remain to be selected. If all optimal tours lie in the right subtree, then we have still to select n edges. Since the left subtree selects fewer edges, it should be easier to find an optimal solution in it than to find one in the right subtree. Consequently, we would like to choose as the partitioning edge an edge $\langle i, j \rangle$ that has the highest probability of being

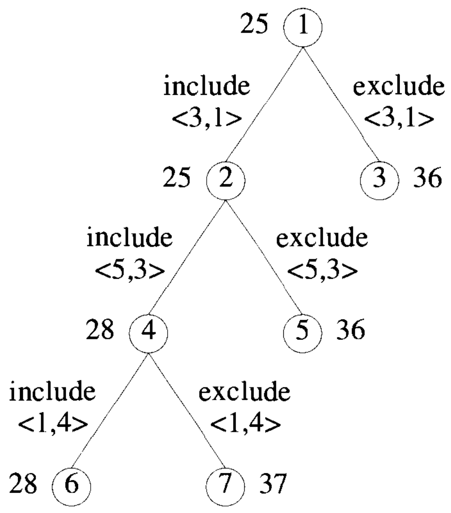


Figure 8.15 State space tree for Figure 8.11(a)

in an optimal tour. Several heuristics for determining such an edge can be formulated. A selection rule that is commonly used is select that edge which results in a right subtree that has highest $\hat{c}$ value. The logic behind this is that we soon have right subtrees (perhaps at lower levels) for which the $\hat{c}$ value is higher than the length of an optimal tour. Another possibility is to choose an edge such that the difference in the $\hat{c}$ values for the left and right subtrees is maximum. Other selection rules are also possible.

When LCBB is used with the first of the two selection rules stated above and the cost matrix of Figure 8.11(a), the tree of Figure 8.15 is generated. At the root node, we have to determine an edge $\langle i, j\rangle$ that will maximize the $\hat{c}$ value of the right subtree. If we select an edge $\langle i, j\rangle$ whose cost in the reduced matrix (Figure 8.11(b)) is positive, then the $\hat{c}$ value of the right subtree will remain 25. This is so as the reduced matrix for the right subtree will have $B(i, j) = \infty$ and all other entries will be identical to those in Figure 8.11(b). Hence B will be reduced and $\hat{c}$ cannot increase. So, we must choose an edge with reduced cost 0. If we choose $\langle 1, 4\rangle$, then $B(1, 4) = \infty$ and we need to subtract 1 from row 1 to obtain a reduced matrix. In this case $\hat{c}$ will be 26. If $\langle 3, 1\rangle$ is selected, then 11 needs to be subtracted from column 1 to obtain the reduced matrix for the right subtree. So, $\hat{c}$ will be 36. If A is the reduced cost matrix for node R, then the selection of edge $\langle i, j\rangle$ $(A(i, j) = 0)$ as the next partitioning edge will increase the $\hat{c}$ of the

$$\begin{bmatrix} \infty & 10 & \infty & 0 & 1 \\ \infty & \infty & 11 & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & 12 & \infty & 0 \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(a) Node 2

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 1 & \infty & 11 & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & 12 & \infty & 0 \\ 0 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Node 3

$$\begin{bmatrix} \infty & 7 & \infty & 0 & \infty \\ \infty & \infty & \infty & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(c) Node 4

$$\begin{bmatrix} \infty & 10 & \infty & 0 & 1 \\ \infty & \infty & 0 & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & 1 & \infty & 0 \\ \infty & 0 & \infty & 12 & \infty \end{bmatrix}$$

(d) Node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(e) Node 6

$$\begin{bmatrix} \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(f) Node 7

Figure 8.16 Reduced cost matrices for Figure 8.15

right subtree by $\triangle = \min_{k \neq j}\{A(i,k)\} + \min_{k \neq i}\{A(k,j)\}$ as this much needs to be subtracted from row i and column j to introduce a zero into both. For edges $\langle 1,4\rangle, \langle 2,5\rangle, \langle 3,1\rangle$ $\langle 3,4\rangle, \langle 4,5\rangle, \langle 5,2\rangle$, and $\langle 5,3\rangle, \triangle = 1$, 2, 11, 0, 3, 3, and 11 respectively. So, either of the edges $\langle 3,1\rangle$ or $\langle 5,3\rangle$ can be used. Let us assume that LCBB selects edge $\langle 3,1\rangle$. The $\hat{c}(2)$ (Figure 8.15) can be computed in a manner similar to that for the state space tree of Figure 8.12. In the corresponding reduced cost matrix all entries in row 3 and column 1 will be ∞. Moreover the entry (1, 3) will also be ∞ as inclusion of this edge will result in a cycle. The reduced matrices corresponding to nodes 2 and 3 are given in Figure 8.16(a) and (b). The $\hat{c}$ values for nodes 2 and 3 (as well as for all other nodes) appear outside the respective nodes.

Node 2 is the next E-node. For edges $\langle 1,4\rangle, \langle 2,5\rangle, \langle 4,5\rangle, \langle 5,2\rangle$, and $\langle 5,3\rangle$, $\triangle = 3, 2, 3, 3$, and 11 respectively. Edge $\langle 5,3\rangle$ is selected and nodes 4 and 5 generated. The corresponding reduced matrices are given in Figure 8.16(c) and (d). Then $\hat{c}(4)$ becomes 28 as we need to subtract 3 from column 2 to reduce this column. Note that entry (1, 5) has been set to ∞ in Figure 8.16(c). This is necessary as the inclusion of edge $\langle 1,5\rangle$ to the collection $\{\langle 3,1\rangle, \langle 5,3\rangle\}$ will result in a cycle. In addition, entries in column 3 and row 5 are set to ∞. Node 4 is the next E-node. The $\triangle$ values corresponding to edges $\langle 1,4\rangle, \langle 2,5\rangle$, and $\langle 4,2\rangle$ are 9, 2, and 0 respectively. Edge $\langle 1,4\rangle$ is selected and nodes 6 and 7 generated. The edge selection at node 6 is $\{\langle 3,1\rangle, \langle 5,3\rangle, \langle 1,4\rangle\}$. This corresponds to the path 5, 3, 1, 4. So, entry (4, 5) is set to ∞ in Figure 8.16(e). In general if edge $\langle i,j\rangle$ is selected, then the entries in row i and column j are set to ∞ in the left subtree. In addition, one more entry needs to be set to ∞. This is an entry whose inclusion in the set of edges would create a cycle (Exercise 4 examines how to determine this). The next E-node is node 6. At this time three of the five edges have already been selected. The remaining two may be selected directly. The only possibility is $\{\langle 4,2\rangle, \langle 2,5\rangle\}$. This gives the path 5, 3, 1, 4, 2, 5 with length 28. So *upper* is updated to 28. Node 3 is the next E-node. Now LCBB terminates as $\hat{c}(3) = 36 > upper$.

In the preceding example, LCBB was modified slightly to handle nodes close to a solution node differently from other nodes. Node 6 is only two levels from a solution node. Rather than evaluate $\hat{c}$ at the children of 6 and then obtain their grandchildren, we just obtained an optimal solution for that subtree by a complete search with no bounding. We could have done something similar when generating the tree of Figure 8.12. Since node 6 is only two levels from the leaf nodes, we can simply skip computing $\hat{c}$ for the children and grandchildren of 6, generate all of them, and pick the best. This works out to be quite efficient as it is easier to generate a subtree with a small number of nodes and evaluate all the solution nodes in it than it is to compute $\hat{c}$ for one of the children of 6. This latter statement is true of many applications of branch-and-bound. Branch-and-bound is used on large subtrees. Once a small subtree is reached (say one with 4 or 6 nodes in it), then that subtree is fully evaluated without using the bounding functions.

We have now seen several branch-and-bound strategies for the traveling salesperson problem. It is not possible to determine analytically which of these is the best. The exercises describe computer experiments that determine empirically the relative performance of the strategies suggested.

EXERCISES

1. Consider the traveling salesperson instance defined by the cost matrix

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

   (a) Obtain the reduced cost matrix

   (b) Using a state space tree formulation similar to that of Figure 8.10 and $\hat{c}$ as described in Section 8.3, obtain the portion of the state space tree that will be generated by LCBB. Label each node by its $\hat{c}$ value. Write out the reduced matrices corresponding to each of these nodes.

   (c) Do part (b) using the reduced matrix method and the dynamic state space tree approach discussed in Section 8.3.

2. Do Exercise 1 using the following traveling salesperson cost matrix:

$$\begin{bmatrix} \infty & 11 & 10 & 9 & 6 \\ 8 & \infty & 7 & 3 & 4 \\ 8 & 4 & \infty & 4 & 8 \\ 11 & 10 & 5 & \infty & 5 \\ 6 & 9 & 5 & 5 & \infty \end{bmatrix}$$

3. (a) Describe an efficient implementation for a LCBB traveling salesperson algorithm using the reduced cost matrix approach and (i) a dynamic state space tree and (ii) a static tree as in Figure 8.10.
   (b) Are there any problem instances for which the LCBB will generate fewer nodes using a static tree than using a dynamic tree? Prove your answer.

4. Consider the LCBB traveling salesperson algorithm described using the dynamic state space tree formulation. Let A and B be nodes. Let B be a child of A. If the edge (A, B) represents the inclusion of edge $\langle i, j \rangle$ in the tour, then in the reduced matrix for B all entries in row i and column j are set to ∞. In addition, one more entry is set to ∞. Obtain an efficient way to determine this entry.

5. [Programming project] Write computer programs for the following traveling salesperson algorithms:
   (a) The dynamic programming algorithm of Chapter 5
   (b) A backtracking algorithm using the static tree formulation of Section 8.3
   (c) A backtracking algorithm using the dynamic tree formulation of Section 8.3
   (d) A LCBB algorithm corresponding to (b)
   (e) A LCBB algorithm corresponding to (c)

   Design data sets to be used to compare the efficiency of the above algorithms. Randomly generate problem instances from these data sets and obtain computing times for your programs. What conclusions can you draw from your computing times?

8.4 EFFICIENCY CONSIDERATIONS

One can pose several questions concerning the performance characteristics of branch-and-bound algorithms that find least-cost answer nodes. We might ask questions such as:

1. Does the use of a better starting value for *upper* always decrease the number of nodes generated?
2. Is it possible to decrease the number of nodes generated by expanding some nodes with $\hat{c}() > upper$?
3. Does the use of a better $\hat{c}$ always result in a decrease in (or at least not an increase in) the number of nodes generated? (A $\hat{c}_2$ is better than $\hat{c}_1$ iff $\hat{c}_1(x) \le \hat{c}_2(x) \le c(x)$ for all nodes x.)
4. Does the use of dominance relations ever result in the generation of more nodes than would otherwise be generated?

In this section we answer these questions. Although the answers to most of the questions examined agree with our intuition, the answers to others are contrary to intuition. However, even in cases in which the answer does not agree with intuition, we can expect the performance of the algorithm to generally agree with the intuitive expectations. All the following theorems assume that the branch-and-bound algorithm is to find a minimum-cost solution node. Consequently, $c(x)$ = cost of minimum-cost solution node in subtree x.

Theorem 8.2 Let t be a state space tree. The number of nodes of t generated by FIFO, LIFO, and LC branch-and-bound algorithms cannot be decreased by the expansion of any node x with $\hat{c}(x) \ge upper$, where *upper* is the current upper bound on the cost of a minimum-cost solution node in the tree t.

Proof: The theorem follows from the observation that the value of *upper* cannot be decreased by expanding x (as $\hat{c}(x) \ge upper$). Hence, such an expansion cannot affect the operation of the algorithm on the remainder of the tree. □

Theorem 8.3 Let U_1 and $U_2, U_1 < U_2$, be two initial upper bounds on the cost of a minimum-cost solution node in the state space tree t. Then FIFO, LIFO, and LC branch-and-bound algorithms beginning with U_1 will generate no more nodes than they would if they started with U_2 as the initial upper bound.

Proof: Left as an exercise. □

Theorem 8.4 The use of a better $\hat{c}$ function in conjunction with FIFO and LIFO branch-and-bound algorithms does not increase the number of nodes generated.

Proof: Left as an exercise. □

Theorem 8.5 If a better $\hat{c}$ function is used in a LC branch-and-bound algorithm, the number of nodes generated may increase.

Proof: Consider the state space tree of Figure 8.17. All leaf nodes are solution nodes. The value outside each leaf is its cost. From these values it follows that $c(1) = c(3) = 3$ and $c(2) = 4$. Outside each of nodes 1, 2, and 3 is a pair of numbers $\binom{\hat{c}_1}{\hat{c}_2}$. Clearly, $\hat{c}_2$ is a better function than $\hat{c}_1$. However, if $\hat{c}_2$ is used, node 2 can become the E-node before node 3, as $\hat{c}_2(2) = \hat{c}_2(3)$. In this case all nine nodes of the tree will get generated. When $\hat{c}_1$ is used, nodes 4, 5, and 6 are not generated. □

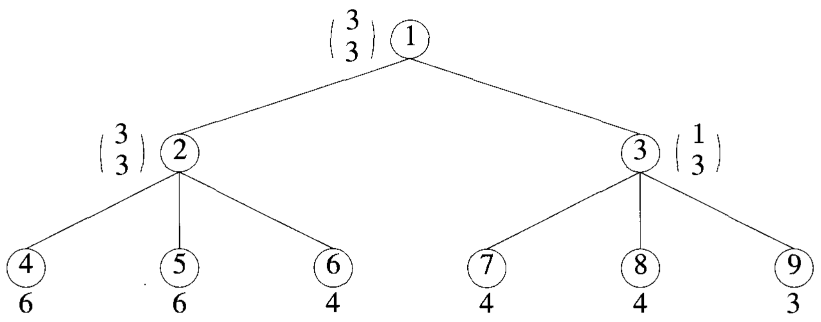


Figure 8.17 Example tree for Theorem 8.5

Now, let us look at the effect of dominance relations. Formally, a dominance relation D is given by a set of tuples, $D = \{(i_1, i_2), (i_3, i_4), (i_5, i_6), \ldots\}$. If $(i, j) \in D$, then node i is said to dominate node j. By this we mean that subtree i contains a solution node with cost no more than the cost of a minimum-cost solution node in subtree j. Dominated nodes can be killed without expansion.

Since every node dominates itself, $(i, i) \in D$ for all i and D. The relation (i, i) should not result in the killing of node i. In addition, it is quite possible for D to contain tuples (i_1, i_2), $(i_2, i_3), (i_3, i_4), \ldots, (i_n, i_1)$. In this case, the transitivity of D implies that each node i_k dominates all nodes $i_j, 1 \leq j \leq n$. Care should be taken to leave at least one of the i_j's alive. A dominance relation D_2 is said to be *stronger* than another dominance relation D_1 iff $D_1 \subset D_2$. In the following theorems I denotes the identity relation $\{(i, i) | 1 \leq i \leq n\}$.

Theorem 8.6 The number of nodes generated during a FIFO or LIFO branch-and-bound search for a least-cost solution node may increase when a stronger dominance relation is used.

Proof: Consider the state space tree of Figure 8.18. The only solution nodes are leaf nodes. Their cost is written outside the node. For the remaining nodes the number outside each node is its $\hat{c}$ value. The two dominance relations to use are $D_1 = I$ and $D_2 = I \cup \{(5,2),(5,8)\}$. Clearly, D_2 is stronger than D_1 and fewer nodes are generated using D_1 rather than D_2. □

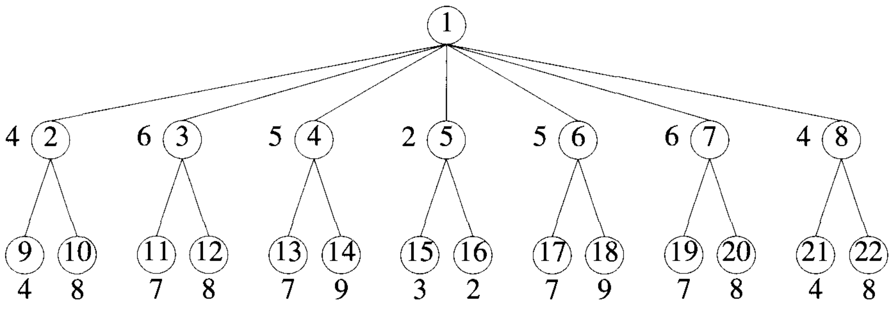


Figure 8.18 Example tree for Theorem 8.6

Theorem 8.7 Let D_1 and D_2 be two dominance relations. Let D_2 be stronger than D_1 and such that $(i,j) \in D_2, i \neq j$, implies $\hat{c}(i) < \hat{c}(j)$. An LC branch-and-bound using D_1 generates at least as many nodes as one using D_2.

Proof: Left as an exercise. □

Theorem 8.8 If the condition $\hat{c}(i) < \hat{c}(j)$ in Theorem 8.7 is removed then an LC branch-and-bound using the relation D_1 may generate fewer nodes than one using D_2.

Proof: Left as an exercise. □

EXERCISES

1. Prove Theorem 8.3.
2. Prove Theorem 8.4.
3. Prove Theorem 8.7.
4. Prove Theorem 8.8.
5. [Heuristic search] Heuristic search is a generalization of FIFO, LIFO, and LC searches. A heuristic function $h(\cdot)$ is used to evaluate all live nodes. The next E-node is the live node with least $h(\cdot)$. Discuss the advantages of using a heuristic function $h(\cdot)$ different from $\hat{c}(\cdot)$ in the search for a least-cost answer node. Consider the knapsack and traveling salesperson problems as two example problems. Also consider any other problems you wish. For these problems devise reasonable functions $h(\cdot)$ (different from $\hat{c}(\cdot)$). Obtain problem instances on which heuristic search performs better than LC-search.

8.5 REFERENCES AND READINGS

LC branch-and-bound algorithms have been extensively studied by researchers in areas such as artificial intelligence and operations research.

Branch-and-bound algorithms using dominance relations in a manner similar to that suggested by FIFOKNAP (resulting in DKnap1) were given by M. Held and R. Karp.

The reduction technique for the knapsack problem is due to G. Ingargiola and J. Korsh.

The reduced matrix technique to compute $\hat{c}$ is due to J. Little, K. Murty, D. Sweeny, and C. Karel. They employed the dynamic state space tree approach.

The results of Section 8.4 are based on the work of W. Kohler, K. Steiglitz, and T. Ibaraki.

The application of branch-and-bound and other techniques to the knapsack and related problems is discussed extensively in *Knapsack Problems: Algorithms and Computer Implementations*, by S. Martello and P. Toth, John Wiley and Sons, 1990.