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 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 when 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 were 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 next *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 get killed as a result of the bounding functions have a B under them. Numbers inside the node correspond to the numbers in figure 7.2. Numbers outside the node 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. □

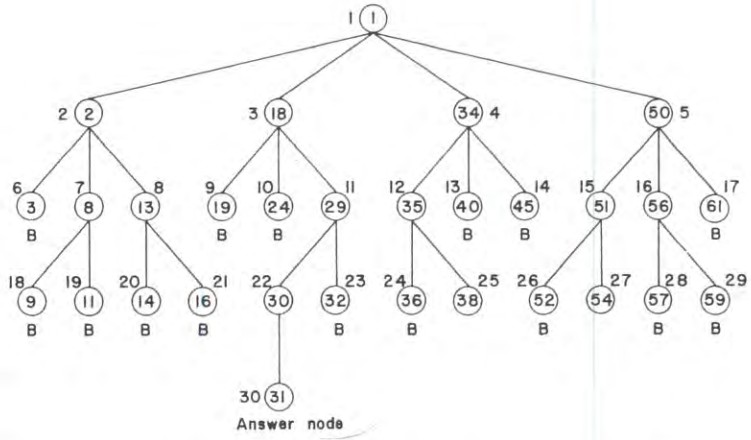


Figure 8.1 Portion of 4-queens state space tree generated by FIFO branch-and-bound

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 requires first the expansion of all live nodes generated before node 30 was generated.

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 (i) the number of nodes in the subtree X that need to be generated before an answer node is generated or more simply, (ii) it could be the number of levels the nearest answer node (in the subtree X) is from X. Using this latter measure, the cost of the root of the tree of figure 8.1 is 4 (node 31 is four levels from node 1). The cost of nodes (18 and 34); (29 and 35) and (30 and 38) is respectively 3, 2 and 1. The cost of all remaining nodes on levels 2, 3 and 4 is 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 (i) is used then the search would always generate the minimum number of nodes every branch-and-bound type algorithm must generate. If cost measure (ii) 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 will usually involve 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 based only on 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 usefullness 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 will be spending 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

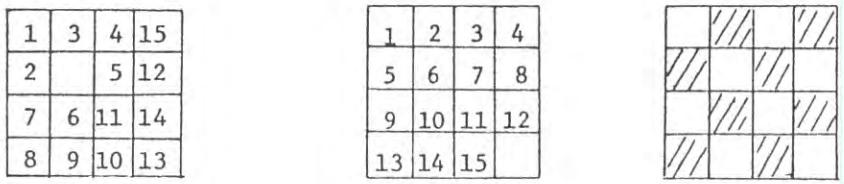
searched. This would be no cause for concern if $\hat{g}(X)$ was the true cost of X. Then, we would not wish to explore the remaining subtrees in any case (as X is guaranteed to get us to an answer node quicker than any other existing live node). However, $\hat{g}(X)$ is only an estimate of the true cost. So, it is quite possible that for two nodes W and Z, $\hat{g}(W) < \hat{g}(Z)$ and Z is actually much closer to an answer node than W. It is therefore desirable not to over bias 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 an LC-search generates nodes by levels. This is essentially the same as a BFS search. 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 will be called an LC branch-and-bound search.

In discussing LC-searches we will 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 $c(X) = \infty$ if 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 $c(X)$ will be referred to as the cost of X.

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 8.3(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 will be called the *states* of the puzzle. The initial and goal arrangements are called the initial and goal states. A state is reachable



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

Figure 8.2 15-puzzle arrangements

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 or not the goal state is reachable from the initial state. There is a very simple way to do this. Let us number the frame positions 1-16. 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*. POSITION(16) will denote the position of the empty spot. 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 $\sum_{i=1}^{16} \text{LESS}(i) + X$ is even.

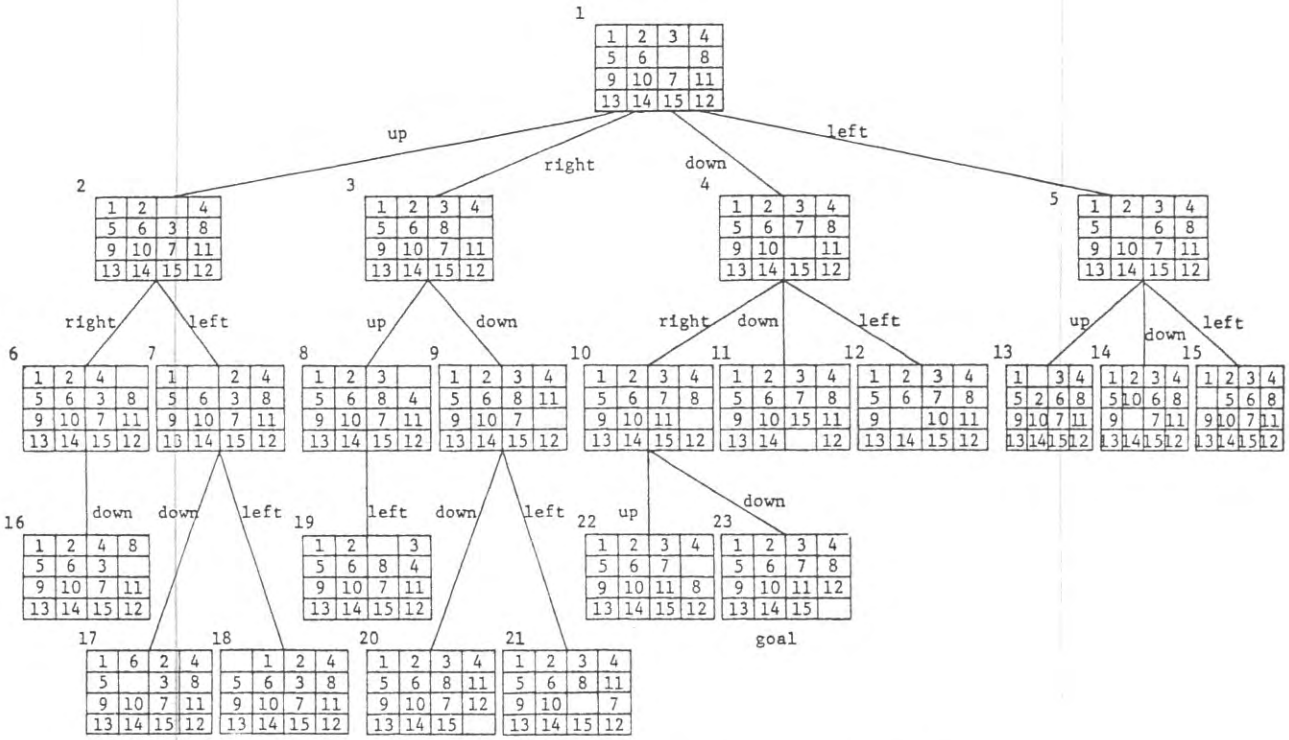
Proof: Left as an exercise. □

Theorem 8.1 may be used to determine whether or not the goal state is in the state space of the initial state. If it is, then we may proceed to determine a sequence of moves leading to the goal state. In order to carry

out this search, the state space may 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 actually 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(a) 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.

A depth first generation of the state space tree will generate the subtree of Figure 8.3(b) when next moves are attempted in the order: move the empty space up, right, down, left. It is clear from successive board configurations 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 left most path ends in such a node). In a FIFO search of the tree of Figure 8.3(a), 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.

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. With each node X in the state space tree we can associate a cost $c(X)$. $c(X)$ 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(a), $c(1) = c(4) = c(10) = c(23) = 3$. 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.



edges are labeled according to the direction in which the empty space moves

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

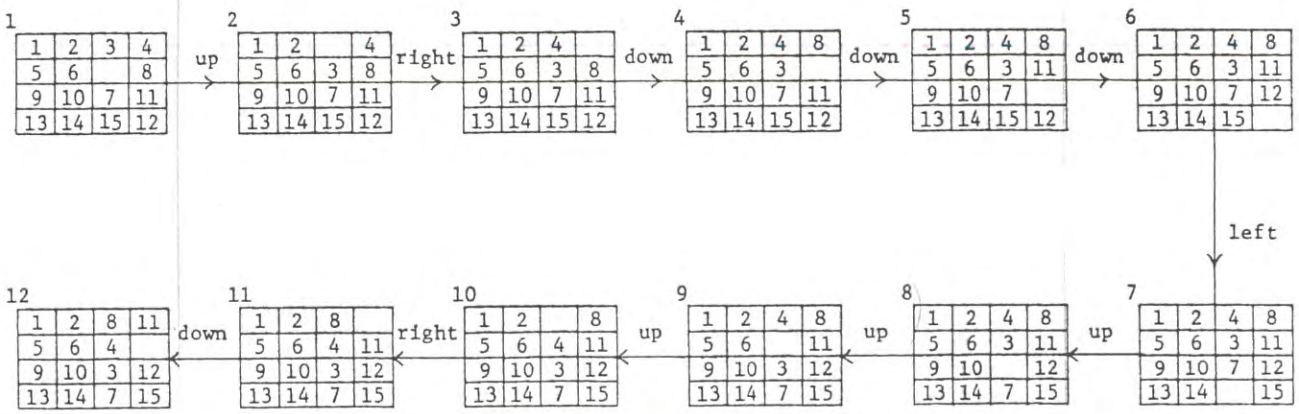


Figure 8.3(b) First ten steps in a depth first search

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 will have to be made to transform state X to a goal state. It is easy to see that more than $\hat{g}(X)$ moves may be needed to achieve this. To see this, examine the problem state of Figure 8.4. $\hat{g}(X) = 1$ 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 $\hat{g}(X)$. $\hat{c}(X)$ is a *lower bound* on the value of $\hat{c}(X)$.

An LC search of Figure 8.3(a) using $\hat{c}(X)$ will begin by using node 1 as the E-node. All its children are generated. Node 1 dies leaving behind the live nodes 2, 3, 4 and 5. The next node to become the E-node is a live node with least $\hat{c}(X)$. $\hat{c}(2) = 1 + 4$, $\hat{c}(3) = 1 + 4$, $\hat{c}(4) = 1 + 2$ 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. $\hat{c}(10) = 2 + 1$, $\hat{c}(11) = 2 + 3$, $\hat{c}(12) = 2 + 3$. The live node with least $\hat{c}$ is node 10. This becomes the next E-node. Nodes 22 and 23 are next generated. 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 8.4 Problem state

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 will usually not be possible to find an easily computable function $c(\ )$ as defined above. Instead, a heuristic $\hat{c}(\ )$ that estimates $c(\ )$ will be used. This heuristic should be easy to compute and will generally have the property that if X is either an answer node or a leaf node then $c(X) = \hat{c}(X)$. Procedure LC (Algorithm 8.1) uses $\hat{c}$ to find an answer node. The algorithm uses two subalgorithms LEAST(X) and ADD(X) to respectively delete and add a live node from or to the list of live nodes. LEAST(X) finds a live node with least $\hat{c}(\ )$. This node is deleted from the list of live nodes and returned in variable X. 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 2.3). Procedure LC 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 variable PARENT(X) 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.

The correctness of algorithm LC is easy to establish. Variable E always points to the current E-node. By definition of LC-search, the root node is

```
line  procedure LC (T, ĉ)
      //search T for an answer node//
      0 if T is an answer node then output T; return; endif
 1    E ← T   //E-node//
 2    initialize the list of live nodes to be empty
 3      loop
 4        for each child X of E do
 5          if X is an answer node then output the path from X to T
 6                                                    return
 7          endif
 8          call ADD(X)   //X is a new live node//
 9          PARENT(X) ← E   //pointer for path to root//
10        repeat
11        if there are no more live nodes then print ('no answer node')
12                                                    stop
13        endif
14        call LEAST(E)
15      repeat
16    end LC
```

Algorithm 8.1 LC-search

the first E-node (line 1). Line 2 initializes the list of live nodes. At any time during the execution of LC, this list contains all live nodes except the E-node. Thus, initially this list should be empty (line 2). The **for** loop of lines 4-10 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 8) and its PARENT field set to E. When all the children of E have been generated, E becomes a dead node and line 11 is reached. This happens only if none of E's children is an answer node. So, the search must continue further. In case 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 12. Otherwise, LEAST(X), by definition correctly chooses the next E-node and the search continues from here.

From the preceding discussion, it is clear that LC terminates only when either an answer node is found or when 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, LC 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 LC 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(X) and ADD(X) being algorithms to delete an element from and add an element to the queue then LC will be transformed to a FIFO search schema. If the list of live nodes is implemented as a stack with LEAST(X) and ADD(X) being algorithms to delete and add elements to the stack then LC 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.

Properties of LC-Search

Let us explore some properties of procedure LC. In many applications it is desirable to find an answer node that has minimum cost among all answer

nodes. Does LC necessarily find an answer node G with minimum cost $c(G)$? The answer to this is no. Consider the state space tree of Figure 8.5. Square leaf nodes are answer nodes. Associated with each node is a pair of numbers. The upper number is the value of c and the lower the estimate $\hat{c}$. Thus, c(root) = 10 and $\hat{c}$(root) = 0. It is clear that LC will first generate the two children of the root and then the node with $\hat{c}(\ ) = 2$ will become the E-node. The expansion of this node leads us to the answer node G with $\hat{c}(G) = c(G) = 20$ and the algorithm terminates. The minimum cost answer node G has cost $c(G) = 10$. The reason LC did not get to the minimum cost answer node is that the function $\hat{c}$ is such that there exist two nodes X and Y such that $\hat{c}(X) < \hat{c}(Y)$ while $c(X) > c(Y)$. As a result LC will choose node X as an E-node before node Y and possibly terminate finding an answer node which is a descendent of X. If $\hat{c}(X) < \hat{c}(Y)$ for every pair of nodes X, Y such that $c(X) < c(Y)$ then, one may show that LC always finds a minimum cost answer node in a finite state space tree that has at least one answer node.

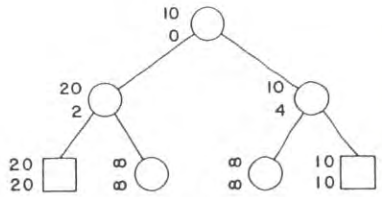


Figure 8.5 LC-search

Theorem 8.2 For every node X in a finite state space tree T let $\hat{c}(X)$ be an estimate of $c(X)$ such that for every pair of nodes Y, Z, $\hat{c}(Y) < \hat{c}(Z)$ iff $c(Y) < c(Z)$. Algorithm LC using $\hat{c}(\ )$ as an estimator for $c(\ )$ reaches a minimum cost answer node and terminates.

Proof: We have already stated that when T is finite, LC finds an answer node if T has such a node. So, assume LC terminates at an answer node G such that $c(G) > c(G')$ where G' is a minimum cost answer node. Let R be the nearest ancestor of G such that the subtree R includes a minimum cost answer node (Figure 8.6). Let $R, \alpha_1, \alpha_2, \ldots, \alpha_k, G'$ be the path from R to G' and let $R, \beta_1, \beta_2, \ldots, \beta_j, G$ be the path from R to G. By definition of R, $\alpha_1 \neq \beta_1$ and the subtree β_1 has no answer node with cost $c(G')$. In order for the search to reach node G, R must become an E-node at some time. At this time its children (including α_1 and β_1) become live

nodes. From the definition of $c(\ )$, it follows that $c(R) = c(\alpha_1) = c(\alpha_2) = \cdots = c(G') = c(\beta_1), c(\beta_2), \ldots, c(G) > c(R)$. Hence, from the condition on $\hat{c}(\ )$ it follows that $\hat{c}(\alpha_1), \hat{c}(\alpha_2), \ldots, \hat{c}(\alpha_k) < \hat{c}(\beta_1)$ and so β_1 cannot become an E-node until $\alpha_i\, 1 \leq i \leq k$ become E-nodes and G' is reached. □

This theorem is easily extended to infinite state space trees in which each node is of finite degree. It is usually not possible to obtain an easily computable $\hat{c}(\ )$ that meets the requirement of Theorem 8.2. We can often only find a $\hat{c}(\ )$ that is easy to compute and has the property that for each node X, $\hat{c}(X) \leq c(X)$. In this case, algorithm LC does not necessarily find a minimum cost answer node (Figure 8.5). When $\hat{c}(X) \leq c(X)$ for every node X and $\hat{c}(X) = c(X)$ for X an answer node, a slight modification to LC results in a search algorithm that terminates when a minimum cost answer node is reached. In this modification, the search continues until an answer node becomes the E-node. The new algorithm is LC1 (Algorithm 8.2).

```
line  procedure LC1 (T, ĉ)
        //search T for a minimum cost answer node.//
 1      E ← T   //first E-node//
 2      initialize the list of live nodes to be empty
 3      loop
 4        if E is an answer node then output path from E to T
 5                                   return
 6        endif
 7        for each child X of E do
 8          call ADD(X); PARENT(X) ← E
 9        repeat
10        if there are no more live nodes then print ('no answer node')
11                                   stop
12        endif
13        call LEAST(E)
14      repeat
15    end LC1
```

Algorithm 8.2 LC – search for least cost answer node

Theorem 8.3 Let $\hat{c}(\cdot)$ be such that $\hat{c}(X) \leq c(X)$ for every node X in a state space tree T and $\hat{c}(X) = c(X)$ for every answer node X in T. If algorithm LC1 terminates in line 5 then the answer node found is of minimum cost.

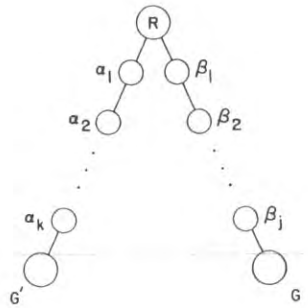


Figure 8.6 State space tree

Proof: At the time the E-node E is an answer node, $\hat{c}(E) \leq \hat{c}(L)$ for every live node L on the list of live nodes. By assumption, $\hat{c}(E) = c(E)$ and $\hat{c}(L) \leq c(L)$ for every live node L. Hence $c(E) \leq c(L)$ and so E is a minimum cost answer node. □

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 shall 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 U is an upper bound on the cost of a minimum cost solution then all live nodes X with $\hat{c}(X) > U$ may be killed as all answer nodes reachable from X have cost $c(X) \geq \hat{c}(X) > U$. In case an answer node with cost U has already been reached then all live nodes with $\hat{c}(X) \geq U$ may be killed. The starting value for U may be obtained by some heuristic or may be set to ∞. Clearly, so long as the initial value for U 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 U may be updated.

Let us see how these ideas may be used to arrive at branch-and-bound algorithms for optimization problems. In this section we shall deal directly only with minimization problems. A maximization problem is easily con-

verted into 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. Nodes representing infeasible solutions have $c(X) = \infty$. For nodes representing partial solutions $c(X)$ is the cost of the minimum cost node in the subtree with root X. Since $c(X)$ will in general be as hard to compute as solving the original optimization problem, 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 the 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 shall 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. J should be a subset 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 may be organized into a tree using either of the two formulations used for the sum of subsets problem (example 7.3). Figure 8.7 corresponds to the variable tuple size formulation while Figure 8.8 corresponds to the fixed tuple size formulation. In both figures square nodes represent infeasible subsets. In Figure 8.7 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 8.8 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 8.8 is given below the nodes.

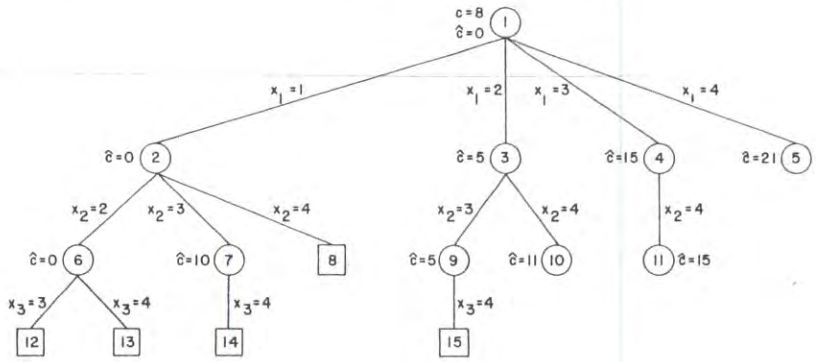


Figure 8.7 State space tree corresponding to variable tuple size formulation

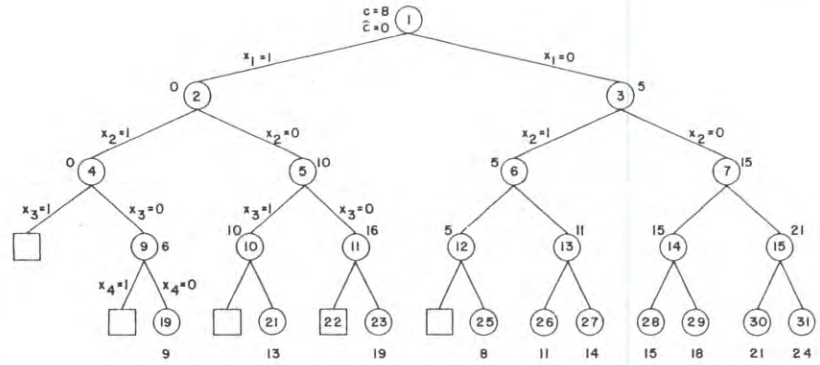


Figure 8.8 State space tree corresponding to fixed tuple size formulation

We can define a cost function $c(\ )$ for the state space formulations of Figures 8.7 and 8.8. For any circular node X, $c(X)$ is the minimum penalty corresponding to any node in the subtree with root X. $c(X) = \infty$ for a square node. In the tree of Figure 8.7, $c(3) = 8$, $c(2) = 9$ and $c(1) = 8$. In the tree of Figure 8.8, $c(1) = 8$, $c(2) = 9$, $c(5) = 13$ 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 $m = \max\{i \mid i \in S_X\}$ 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 8.7 and 8.8 the value of $\hat{c}(X)$ is the number outside node X. For a square node $\hat{c}(X) = \infty$. 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.

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with $U = \infty$ (or $U = \sum_{1 \leq i \leq n} p_i$) 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 8.7, nodes 2, 3, 4 and 5 are generated (in that order). $u(2) = 19$, $u(3) = 14$, $u(4) = 18$ and $u(5) = 21$. U is updated to 14 when node 3 is generated. Since $\hat{c}(4)$ and $\hat{c}(5)$ are greater than U, 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. $u(6) = 9$ and so U is updated to 9. $\hat{c}(7) = 10 > U$ 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. $u(9) = 8$ and so U becomes 8. $\hat{c}(10) = 11 > U$ 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 $\hat{c}(X) > U$ (or $\hat{c}(X) \geq U$ in case a node with cost U has been found) each time U is updated. This is so because live nodes are in the queue in the order in which they were generated. Hence, nodes with $\hat{c}(X) > U$ (or $\hat{c}(X) \geq U$) are distributed in some random way in the queue. Instead, live nodes with $\hat{c}(X) > U$ (or $\hat{c}(X) \geq U$) are killed when they are about to become E-nodes. Procedure FIFOBB is a program schema for a FIFO branch-and-bound algorithm. It uses a small positive constant ϵ such that if for any two feasible nodes X and Y $u(X) < u(Y)$, then $u(X) < u(X) + \epsilon < u(Y)$. This ϵ is needed to distinguish between the case when a solution with cost $u(X)$ has been found and the case when such a solution has not been found. If the latter is the case then U is updated to $\min\{U, u(X) + \epsilon\}$. When U is updated in this way, live nodes Y with $u(Y) \geq U$ may be killed. This does not kill the node that promised to lead to a solution with value $\leq U$. We may dispense with this use of ϵ if every feasible node X that is generated defines a feasible solution and $u(X) =$ cost of X. This is true, for example, for Figure 8.7 with $u(\cdot)$

as defined above. FIFOBB also uses the subalgorithms ADDQ(X) and DELETEQ(X). These algorithms respectively add a node to a queue and delete a node from a queue. For every solution node X in the state space tree $cost(X)$ is the cost of the solution corresponding to node X. FIFOBB assumes $\hat{c}(X) = \infty$ for infeasible nodes and $\hat{c}(X) \leq c(X) \leq u(X)$ for feasible nodes.

```
line  procedure FIFOBB (T, ĉ, u, ε, cost)
        //Search T for a least cost answer (solution) node. It is//
        //assumed that T contains at least one solution node and//
        //ĉ(X) ≤ c(X) ≤ u(X).//
1       E ← T; PARENT(E) ← 0;
2       if T is a solution node then U ← min(cost(T), u(T) + ε); ans ← T
3                               else U ← u(T) + ε; ans ← 0
4       endif
5       initialize queue to be empty
6       loop
7         for each child X of E do
8           if ĉ(X) < U then call ADDQ(X); PARENT(X) ← E
9                                case
10                                 :X is a solution node and cost(X) < U:
11                                   U ← min (cost(X), u(X) + ε)
12                                   ans ← X
13                                 :u(X) + ε < U:U ← u(X) + ε
14                               endcase
15          endif
16        repeat
17        loop  //get next E-node//
18          if queue is empty then print ('least cost =', U)
19            while ans ≠ 0 do
20              print (ans)
21              ans ← PARENT(ans)
22            repeat
23          endif
24          call DELETEQ(E)
25          if ĉ(E) < U then exit  //kill nodes with ĉ(E) ≥ U//
26        repeat
27      repeat
28    end FIFOBB
```

Algorithm 8.3 FIFO branch-and-bound to find minimum cost answer node

LC Branch-and-Bound

An LC branch-and-bound search of the tree of Figure 8.7 will begin with $U = \infty$ 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 FIFO branch-and-bound, U is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as $u(4) > U$ and $u(5) > U$. Node 2 is the next E-node as $\hat{c}(2) = 0$ while $\hat{c}(3) = 5$. Nodes 6, 7 and 8 are generated. U is updated to 9 when node 6 is generated. So, node 7 is killed as $\hat{c}(7) = 10 > U$. 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 U is updated to 8 as $u(9) = 8$. So, node 10 with $\hat{c}(10) = 11$ is killed upon 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. An LC branch-and-bound algorithm may also terminate when the next E-node E has $\hat{c}(E) \geq U$.

The control abstraction for LC branch-and-bound is LCBB. It operates under the same assumptions as FIFOBB. ADD and LEAST are algorithms to respectively add a node to a min-heap and delete a node from a min-heap.


```
line  procedure LCBB (T, ĉ, u, ∈, cost)
        //search T for a least cost answer (solution) node. It is assumed//
        //that T contains at least one solution node and ĉ(X) ≤ c(X) ≤//
        //u(X).//
1       E ← T; PARENT(E) ← 0
2       if T is a solution node then U ← min(cost(T), u(T) + ∈); ans ← T
3                              else U ← u(T) + ∈; ans ← 0
4       endif
5       initialize the list of live nodes to be empty
6       loop
7         for each child X of E do
8           if ĉ(X) < U then call ADD(X)
9                             PARENT(X) ← E
10                            case
11                              :X is a solution node and cost(X) < U:
12                                U ← min(cost (X), u(X) + ∈)
13
14                              :u(X) + ∈ < U: U ← u(X) + ∈
15                            endcase
16          endif
17        repeat
18        return
19        if there are no more live nodes or the next E-node
20        has ĉ ≥ U then print ('least cost = ', U)
21                        while ans ≠ 0 do
22                          print (ans)
23                          ans ← PARENT(ans)
24                        repeat
25        endif
26        call LEAST(E)
27      repeat
28    end LCBB
```

Algorithm 8.4 LC branch-and-bound to find minimum cost answer node

8.2 ZERO-ONE KNAPSACK PROBLEM

In order 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).

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

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

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

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. In order 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. $c(X) = \infty$ for infeasible leaf nodes. For nonleaf nodes, $c(X)$ is recursively defined to be $\min\{c(\text{LCHILD}(X)), c(\text{RCHILD}(X))\}$.

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. $\hat{c}(\cdot)$ 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 $u(X) = \text{UBOUND}(q, \sum_{1 \leq i < j} w_i x_i, j - 1, M)$ where UBOUND is defined by Algorithm 8.5. As for $c(X)$, it is clear that $-\text{BOUND}(-q, \sum_{1 \leq i < j} w_i x_i, j - 1, M) \leq c(X)$ where BOUND is Algorithm 7.11.


```
procedure UBOUND (p, w, k, M)
    //p, w, k and M have the same meaning as in Algorithm 7.11//
    //W(i) and P(i) are respectively the weight and profit of the ith object//
      global W(1:n), P(1:n); integer i, k, n
      b ← p; c ← w
      for i ← k + 1 to n do
        if c + W(i) ≤ M then c ← c + W(i); b ← b - P(i) endif
      repeat
      return (b)
end UBOUND
```

Algorithm 8.5 Function $u(\cdot)$ for knapsack problem

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 above. We shall 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 8.9, we have $\hat{c}(1) = -38$ and $u(1) = -32$. Since this is not a solution node, procedure LCBB sets $ans = 0$ and $U = -32 + \epsilon$. The E-node is expanded and its two children, nodes 2 and 3 generated. $\hat{c}(2) = -38$, $\hat{c}(3) = -32$, $u(2) = -32$ 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 gets onto the list of live nodes. Next node 7 gets onto this list and U is updated to $-38 + \epsilon$. 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, U is updated to -38 and node 8 is put onto the live nodes list. Node 9 has $\hat{c}(9) > U$ and is killed immediately. Nodes 6 and 8 are two live nodes with least $\hat{c}$. Regardless of which becomes the next E-node, $\hat{c}(E) \geq U$ 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 $\sum p_i x_i = U$. Hence, a proper implementation of pro-

cedure LCBB will have to keep additional information from which the values of the x_is may 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 will 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 $x_1 = 1$. □

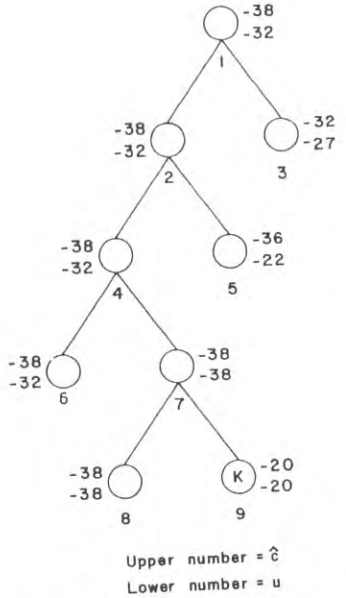


Figure 8.9 LC Branch-and-bound tree for Example 8.2

In order to use procedure LCBB (Algorithm 8.5) to solve the knapsack problem, we need to specify (i) the structure of nodes in the state space tree being searched, (ii) how to generate the children of a given node; (iii) how to recognize a solution node; (iv) a representation of the list of live nodes and subalgorithms ADD and LEAST. The node structure needed will depend 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. In order to be able to generate X's children, we

shall 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 $x_{\text{LEVEL}(X)} = 1$ and the right child by setting $x_{\text{LEVEL}(X)} = 0$. In order 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. 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 $\sum_{1 \le i < \text{LEVEL}(X)} p_i x_i$ earned by the filling corresponding to node X. This may be computed by following the path from X to the root. Alternatively, this value may 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. $\hat{c}(X)$ 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 may be easily determined. The left child, Y, is feasible iff $\text{CU}(X) \ge w_{\text{LEVEL}(X)}$. In this case, PARENT($Y$) = X; LEVEL(Y) = LEVEL(X) + 1; CU(Y) = CU(X) − $w_{\text{LEVEL}(X)}$; PE(Y) = PE(X) + $p_{\text{LEVEL}(X)}$; TAG(Y) = 1 and UB(Y) = UB(X). The right child may 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: a) test if the list is empty b) add nodes and c) 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 a) can be carried out in $\Theta(1)$ time while b) and c) require only $O(\log n)$ time.

While the preceding discussion together with procedure LCBB result in a complete specification of an LC branch-and-bound algorithm for the knapsack problem, some improvement in algorithm efficiency results if we tailor LCBB to this specific problem. First, our tailored algorithm will compute $-\hat{c}$ and $-u$, which are nonnegative quantities, rather than $\hat{c}$ and u. In addition, we shall retain $L = -U$ rather than U. Also, for any live node X, UB(X) = $-\hat{c}(X)$. These changes only result in minor changes in procedure LCBB. These changes are:

i) The conditional of line 8 becomes **if** UB(X) > L **then**
ii) the conditional of line 11 becomes :LEVEL(X) = $n + 1$ **and** PE(X) > L:

iii) line 12 becomes $L \leftarrow \text{PE}(X)$
iv) line 14 becomes $:-u(X) - \epsilon > L:\ L \leftarrow -u(X) - \epsilon$
v) the conditional of line 19 becomes $\text{UB}(X) \leq L$
vi) in line 25 the next E node is the live node with maximum UB.

While these changes do not materially affect the running time of the resulting algorithm, they result in an algorithm that mirrors the "maximization" formulation of the problem rather than the "minimization" formulation (8.1). Thus L is a *l*ower bound on the value of an optimal filling and $\text{UB}(X)$ is an *u*pper *b*ound on the maximum filling obtainable from any solution node in the subtree with root X. The remaining changes we shall make will reduce the running time of the search algorithm. The final algorithm is procedure LCKNAP.

LCKNAP makes use of the subalgorithms LUBOUND (Algorithm 8.6); NEWNODE (Algorithm 8.7(a)); FINISH (Algorithm 8.7(b)), INIT and GETNODE. LUBOUND computes $-\hat{c}(\cdot)$ and $-u(\cdot)$. NEWNODE creates a new six field node, sets the fields appropriately and adds this node to the list of live nodes. Procedure FINISH prints out the value of the optimal solution as well as the objects with $x_i = 1$ in an optimal solution. INIT initializes the list of available nodes and also the list of live nodes. Since nodes are never freed by the algorithms, nodes may be used sequentially *i.e.* nodes 1 through m may be assigned in the order $1, 2, \ldots, m$. GETNODE gets a free node. In accordance with conventions established in Section 8.1, L will be the larger of the value of the best solution found so far and the highest lower bound computed by LUBOUND less ϵ. ϵ is a "small" positive number.

The parameters to LCKNAP are P, W, M and N. N is the number of objects. $P(i)$ and $W(i)$, $1 \leq i \leq N$ are the profits and weights respectively. The objects are indexed such that $P(i)/W(i) \geq P(i+1)/W(i+1)$, $1 \leq i < N$. M is the capacity of the knapsack. Lines 1-5 initialize the list of free nodes and the root node of the search tree. This root node E is the first E-node. The loop of lines 6-24, successively examines each of the live nodes generated. The loop terminates either when there are no live nodes remaining (line 22) or when the next node, E, selected for expansion (the next E-node) is such that $\text{UB}(E) \leq L$ (line 24). The termination at line 24 is valid as the node selected to be the next E-node is a live node with maximum $\text{UB}(E)$. Hence, for all other live nodes X, $\text{UB}(X) \leq \text{UB}(E) \leq L$ and none of them can lead to a solution node with value greater than L. Within this loop, the new E-node E is examined. This node is either a leaf node ($\text{LEVEL}(E) = n + 1$) or it has exactly two children. In case it is a leaf, then it is a solution node and may be a new candidate for the answer node. Lines 9-11 determine this. In case E is not a leaf node, its two children are generated.

The left child, X, corresponds to $x_i = 1$ and the right, Y, to $x_i = 0$ where $i = \text{LEVEL}(E)$. The left child is feasible (i.e. can lead to a solution node) iff there is enough space left in the knapsack to accommodate x_i ($\text{cap} \geq W(i)$). In case this child is feasible and from the way the upper bound is computed by LUBOUND, it follows that $\text{UB}(X) = \text{UB}(E)$. Since $\text{UB}(E) > L$ (line 24) or $L = \text{LBB} - \epsilon < \text{UBB}$ (line 5) it follows that X is to be added to the list of live nodes. Note that there is no need to recompute the lower and upper bound values for this node. They are the same as for E! The right child R is always feasible since E is feasible. For this node the lower and upper bound values may differ from those of node E. Hence, a call to LUBOUND is made (line 16). $\text{UB}(R) = \text{UBB}$. Node R may be killed if $\text{UB}(R) \leq L$. Line 18 adds R to the list of live nodes when R is not be killed. Line 19 updates the value of L.

```
procedure LUBOUND(P, W, rw, cp, N, k, LBB, UBB)
  //rw is the remaining capacity and cp is the profit already earned//
  //objects k, ..., N have yet to be considered//
  //LBB = - u(X) and UBB = - ĉ(X)//
  LBB ← cp; c ← rw
  for i ← k to N do
    if c < W(i) then UBB ← LBB + c * P(i)/W(i)
                     for j ← i + 1 to N do
                       if c ≥ W(j) then c ← c - W(j)
                                        LBB ← LBB + P(j)
                       endif
                     repeat
                     return
    endif
    c ← c - W(i); LBB ← LBB + P(i)
  repeat
  UBB ← LBB
end LUBOUND
```

Algorithm 8.6 Algorithm to compute lower and upper bounds

```
procedure NEWNODE (par, lev, t, cap, prof, ub)
  //create a new node I and add it to the list of live nodes.//
  call GETNODE(I)
  PARENT(I) ← par; LEVEL(I) ← lev; TAG(I) ← t
  CU(I) ← cap; PE(I) ← prof; UB(I) ← ub
  call ADD(I)
end NEWNODE
```

Algorithm 8.7 (a) Creating a new node

```
procedure FINISH(L, ANS, N)
  //print solution//
  real L; global TAG, PARENT
  print ('VALUE OF OPTIMAL FILLING IS', L)
  print ('OBJECTS IN KNAPSACK ARE')
  for j ← N to 1 by −1 do
    if TAG(ANS) = 1 then print(j) endif
    ANS ← PARENT(ANS)
  repeat
end FINISH
```

Algorithm 8.7 (b) Printing the answer

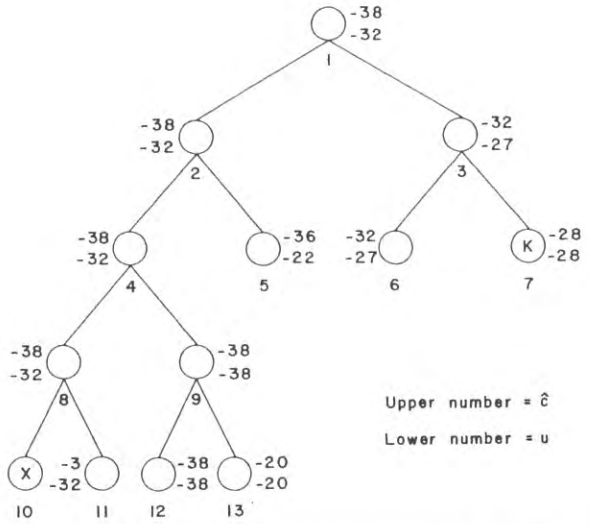


Figure 8.10 FIFO branch-and-bound tree for Example 8.3


```
line procedure LCKNAP(P, W, M, N, ϵ)
      //least cost branch-and-bound algorithm for the 0/1 knapsack//
      //problem. A fixed tuple size formulation is used. It is assumed//
      //that P(1)/W(1) ≥ P(2)/W(2) ≥ ... ≥ P(N)/W(N)//
      real P(N), W(N), M, L, LBB, UBB, cap, prof
      integer ANS, X, N
 1    call INIT  //initialize list of available nodes and list of live nodes//
 2    call GETNODE(E)  //root node//
 3    PARENT(E) ← 0; LEVEL(E) ← 1; CU(E) ← M; PE(E) ← 0
 4    call LUBOUND (P, W, M, 0, N, 1, LBB, UBB)
 5    L ← LBB − ϵ; UB(E) ← UBB
 6    loop
 7      i ← LEVEL(E); cap ← CU(E); prof ← PE(E)
 8      case
 9        :i = N + 1:  //solution node//
10          if prof > L then L ← prof; ANS ← E
11          endif
12        :else:  //E has two children//
13          if cap ≥ W(i) then  //feasible left child//
14          call NEWNODE(E, i + 1, 1, cap − W(i), prof + P(i), UB(E)
15          endif
              //see if right child is to live//
16          call LUBOUND (P, W, cap, prof, N, i + 1, LBB, UBB)
17          if UBB > L then  //right child is to live//
18            call NEWNODE(E, i + 1, 0, cap, prof, UBB)
19            L ← max (L, LBB − ϵ)
20          endif
21      endcase
22      if there are no more live nodes then exit endif
23      call LARGEST(E)  //next E-node is node with largest UB(·)//
24    until UB(E) ≤ L repeat
25    call FINISH(L, ANS, N)
26  end LCKNAP
```

Algorithm 8.8 LC-branch-and-bound algorithm for knapsack problem

FIFO Branch-and-Bound Solution

Example 8.3 (FIFOBB) Now, let us trace through procedure FIFOBB (Algorithm 8.3) using the same knapsack instance as in Example 8.2 and using the knapsack formulation (8.1). Initially the root node, node 1 of Figure 8.10, is the E-node and the queue of live nodes is empty. Since this is not a solution node, U is initialized to $u(1) + \epsilon = -32 + \epsilon$. We shall 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 U 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 E-node, is expanded. It's children nodes are generated. Node 6 gets added to the queue. Node 7 is immediately killed as $\hat{c}(7) \geq U$. Node 4 is next expanded. Nodes 8 and 9 are generated and added to the queue. U is updated to $u(9) + \epsilon = -38 + \epsilon$. Nodes 5 and 6 are the next two nodes to become E-nodes. Neither is expanded as for each, $\hat{c}(\ ) \geq U$. Node 8 is the next E-node. Nodes 10 and 11 are generated. Node 10 is infeasible and so killed. Node 11 has $\hat{c}(11) \geq U$ and so is also killed. Node 9 is next expanded. When node 12 is generated U 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 $\hat{c}(13) > U$. The only remaining live node is node 12. It has no children and the search terminates. The value of U 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. □

As in the case of LCKNAP, we shall tailor the FIFO branch-and-bound algorithm, FIFOKNAP to the problem at hand as well as to the state space tree formulation chosen. Since nodes will be generated and examined (i.e. become E-nodes) by levels, it is possible to keep track of the level of a node by the use of an end of level marker, '#', on the queue of live nodes. This leaves us with five fields per node: *CU*, *PE*, *TAG*, *UB* and *PARENT*. Procedure NNODE (Algorithm 8.9) generates a new live node, sets the fields and adds it to the queue of live nodes.


```
procedure NNODE(par, t, cap, prof, ub)
  //create a new live node I and add it to the queue of live nodes//
  call GETNODE(I)
  PARENT(I) ← par; TAG(I) ← t
  CU(I) ← cap; PI(I) ← prof; UB(I) ← ub
  call ADDQ(I)
end NNODE
```

Algorithm 8.9 Creating a new node

Algorithm FIFOKNAP works with the maximization formulation of the knapsack problem. L represents a lower bound on the value of an optimal solution. Since no solution nodes can be reached until nodes at level $N + 1$ are generated, we can dispense with ϵ as used in LCKNAP. Lines 3–6 initialize the list of free nodes, the root node E, L and the queue of live nodes. This queue initially contains the root node E and the end of level marker '#'. i is the level counter. During the algorithm, i will have as value the level number corresponding to the current E-node. Initially, $i = 1$. In each iteration of the main **while** loop (lines 7–26), all live nodes at level i are removed from the queue. In the loop of lines 8–23, nodes are removed from the queue one by one. In case the end of level marker is removed then the loop is exited (line 11). Otherwise, node E is expanded only if $UB(E) \geq L$. Lines 13–21 generate the left and right children of node E and are similar to the corresponding code in procedure LCKNAP. When we exit from the **while** loop, the only live nodes on the queue are nodes at level $N + 1$. Each of these is a solution node. A node with maximum PE value is an answer node. Such a node may be easily found by examining the PE values of the remaining live nodes one by one. Procedure FINISH (Algorithm 8.7) prints out the value of an optimal solution as well as the objects that must be included into the knapsack in order to obtain this profit.

```
    procedure FIFOKNAP(P, W, M, N)
      //same function and assumptions as LCKNAP//
1     real P(N), W(N), M, L, LBB, UBB, E, prof, cap
2     integer ANS, X, N
3     call INIT; i ← 1
4     call LUBOUND(P, W, M, 0, N, 1, L, UBB)
5     call NNODE(0, 0, M, 0, UBB)   //root node//
6     call ADDQ('#')   //level marker//
7     while i ≤ N do   //for all live nodes on level i//
8       loop
9         call DELETEQ(E)
10        case
11          :E = '#': exit   //end of level i.Exit to line 24//
12          :UB(E) ≥ L:   //E is to live//
13            cap ← CU(E); prof ← PE(E)
14            if cap ≥ W(i) then   //feasible left child//
15              call NNODE(E, 1, cap − W(i), prof + P(i), UB(E))
16            endif
17            call LUBOUND(P, W, cap, prof, N, i + 1, LBB, UBB)
18            if UBB ≥ L then   //right child is to live//
19              call NNODE(E, 0, cap, prof, UBB)
20              L ← max(L, LBB)
21            endif
22        endcase
23      repeat
24      call ADDQ('#')   //end of level//
25      i ← i + 1
26    repeat
27    ANS ← live node X with PE(X) = L
28    call FINISH(L, ANS, N)
29  end FIFOKNAP
```

Algorithm 8.10 FIFO branch-and-bound knapsack algorithm

At first, we may be tempted to discard FIFOKNAP in favor of LCKNAP. Our intuition leads us to believe that LCKNAP will examine fewer nodes in its quest for an optimal solution. However, we should keep in mind that insertions into and deletions from 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 LCKNAP than in FIFOKNAP. Unless LCKNAP uses far fewer E-nodes than FIFOKNAP, FIFOKNAP will outperform (in terms of real computation time) LCKNAP.

We have now seen four different approaches to solving the knapsack problem: dynamic programming; backtracking; LC branch-and-bound and FIFO branch-and-bound. If we compare the dynamic programming algorithm DKNAP (Algorithm 5.7) and FIFOKNAP 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 FIFOKNAP 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 5.5 into FIFOKNAP as nodes on a level are not ordered by their P or W values. However, the bounding rules can easily be incorporated into DKNAP. Towards the end of Section 5.5 we discussed some simple heuristics to determine if 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 FIFOKNAP 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. In order 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 chapter. The results of some tests may 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, actually 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 only the reduced knapsack instance:

$$\begin{aligned} &\text{maximize} \sum_{i \in I3} p_i x_i \\ &\text{subject to} \sum_{i \in I3} w_i x_i \leq M - \sum_{i \in I1} w_i x_i \\ &x_i = 0 \text{ or } 1 \end{aligned} \tag{8.2}$$

has to be solved. 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 $i \in I2$.

Procedure REDUCE makes use of two functions UBB($I1$, $I2$) and LBB ($I1$, $I2$). UBB($I1$, $I2$) is an upper bound on the value of an optimal solution to the given knapsack instance with the added constraints $x_i = 1$ if $i \in I1$ and $x_i = 0$ if $i \in I2$. LBB($I1$, $I2$) is a lower bound under the constraints of $I1$ and $I2$. Note that UBB($I1$, $I2$) and LBB($I1$, $I2$) are the same as UBB and LBB of LUBOUND provided they are computed at a node X representing the assignment $x_i = 1$ if $i \in I1$ and $x_i = 0$ if $i \in I2$. Procedure 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.

```
procedure REDUCE (P, W, n, M, I1, I2)
  //variables are as described above. P(i)/W(i) ≥ P(i + 1)/W(i + 1),//
  //1 ≤ i < n//
  I1 ← I2 ← φ
  L ← LBB(φ, φ)
  k ← largest j such that Σ_{1≤i≤j} W(i) < M
  for i ← 1 to k do   //determine I1//
     case
        :UBB(φ, {i}) < L:I1 ← I1 ∪ {i}
        :LBB(φ, {i}) > L:L ← LBB(φ,{i})
     endcase
  repeat
  for i ← k + 1 to n do   //determine I2//
     case
        :UBB( {i}, φ) < L: I2 ← I2 ∪ {i}
        :LBB( {i}, φ) > L:L ← LBB( {i}, φ)
     endcase
  repeat
end REDUCE
```

Algorithm 8.11 Reduction algorithm for knapsack problem

The time complexity of REDUCE is $O(n^2)$. Because the reduction procedure is very much like the heuristics used in DKNAP1, LCKNAP, BKNAP1 and BKNAP2, the use of REDUCE does not decrease the overall computing time by as much as may be expected by the reduction in number of objects. These algorithms do dynamically what REDUCE does. The exercises explore the value of REDUCE further.

8.3 TRAVELING SALESPERSON

An $O(n^2 2^n)$ dynamic programming algorithm for the traveling salesperson problem was arrived at in Section 5.7. We shall now investigate branch-and-bound algorithms for this problem. While 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} be 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 may assume that every tour starts and ends at vertex 1. So, the solution space S is given by $S = \{1, \pi, 1 | \pi \text{ is a permutation of } (2, 3, \ldots, n)\}$. $|S| = (n - 1)!$. The size of S may be reduced by restricting S so that $(1, i_1, i_2, \ldots, i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1} \rangle \in E$, $0 \leq j \leq n - 1$, $i_0 = i_n = 1$. S may be organized into a state space tree similar to that for the n-queens problem (see Figure 7.2). Figure 8.11 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 $i_4 = 1$.

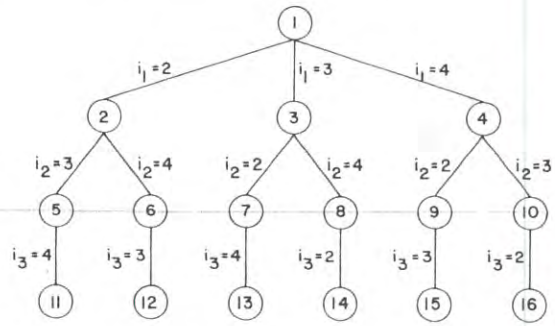


Figure 8.11 State space tree for the traveling salesperson problem with $n = 4$ and $i_0 = i_4 = 1$

In order to use LC-branch-and-bound 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. $c(\cdot)$ is such that the solution node with least $c(\cdot)$ corresponds to a shortest tour in G. One choice for $c(\cdot)$ is:

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

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 8.11 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 $\hat{c}(\cdot)$ may be obtained by using the reduced cost matrix corresponding to G. A row (column) is said to *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 8.12(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) will introduce a zero into row i (column j). Repeating this as often as needed, the cost matrix may be reduced. The total amount subtracted from all the columns and rows is a lower bound on the length of a minimum cost tour and may 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, 5 and columns 1 and 3 respectively of the matrix of Figure 8.12(a) yields the reduced matrix of Figure 8.12(b). The total amount subtracted is 25. Hence, all tours in the original graph have a length at least 25.

With every node in the traveling salesperson state space tree we may associate a reduced cost matrix. 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 (i) 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. (ii) set $A(j, 1)$ to ∞. This prevents the use of edge $\langle j, 1\rangle$. (iii) reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞. Let the resulting matrix be B. Steps (i) and (ii) 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 (iii) 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 may 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 8.12 An example

Let us now trace the progress of the LC branch-and-bound algorithm, LCBB(Algorithm 8.4), on the problem instance of Figure 8.12(a). We shall use $\hat{c}$ and u as above. The initial reduced matrix is that of Figure 8.12(b) and $U = \infty$. The portion of the state space tree that gets generated is shown in Figure 8.13. 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 8.14. The matrix of Figure 8.14(b) is obtained from that of 8.12(b) by (i) setting all entries in row 1 and column 3 to ∞; (ii) the element at position (3, 1) is set to ∞; (iii) column 1 is reduced by subtracting by 11. The $\hat{c}$ for node 3 is therefore 25 + 17 (cost of edge $\langle 1, 3\rangle$ in reduced matrix) + 11 = 53. The matrices and $\hat{c}$ values for nodes 2, 4, and 5 are obtained similarly. U 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 U is updated to 28. For the next E-node, node 5, $\hat{c}(5) = 31 > U$. Hence, LCBB terminates with 1, 4, 2, 5, 3, 1 as the shortest length tour.

An exercise examines the implementation considerations for the algorithm described above. A different LC branch-and-bound algorithm may be

arrived at by considering a different tree organization for the solution space. This organization is arrived at 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. Figures 8.15(b) and (c) represent the first two levels of two possible state space trees for the three vertex graph of Figure 8.15(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 8.15(b) we first decide the fate of edge $\langle 1, 3 \rangle$ while in Figure 8.15(c) we first decide the fate of edge $\langle 1, 2 \rangle$. Rather than use a static state space tree, we shall now consider a dynamic state space tree (see Section 7.1). This will also be a binary tree. However, the order in which edges will be considered will depend on the particular problem instance being solved. We shall compute $\hat{c}$ in the same way as we did using the earlier state space tree formulation.

As an example of how LCBB would work on the dynamic binary tree formulation, consider the cost matrix of Figure 8.12(a). Since a total of 25 needs to be subtracted from the rows and columns of this matrix in order

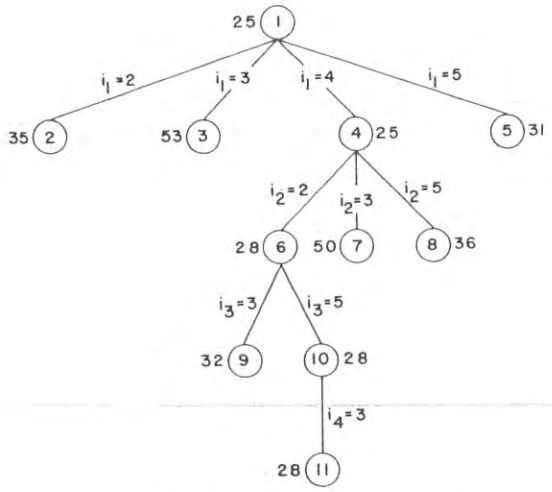


Figure 8.13 State space tree generated by procedure LCBB.

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

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 8.14 Reduced cost matrices corresponding to nodes in Figure 8.13

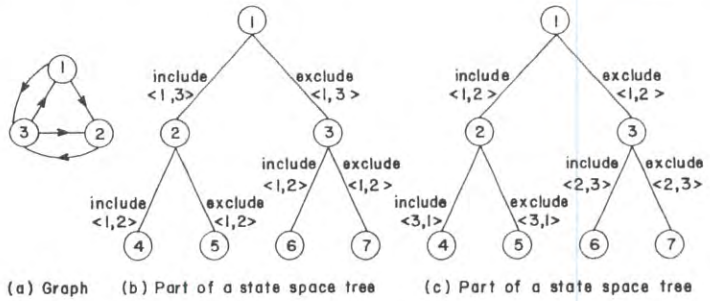


Figure 8.15 An example

to obtain the reduced matrix of Figure 8.12(b), all tours have a length at least 25. This fact is represented by the root of the state space tree of Figure 8.16. 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 will represent all tours including edge $\langle i, j \rangle$ and the right subtree

will represent 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 highest probability of being in an optimal tour. Several heuristics for determining such an edge may 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 will 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.

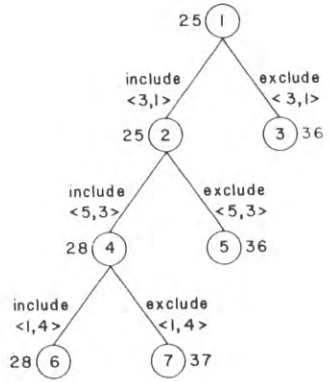


Figure 8.16 State space tree for Figure 8.12(a)

When procedure LCBB is used with the first of the two selection rules stated above and the cost matrix of Figure 8.12(a), the tree of Figure 8.16 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 8.12(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 8.12(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 in order 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 right subtree by $\Delta = \min_{k \neq j}\{A(i, k)\} + \min_{k \neq i}\{A(k, j)\}$ as this much needs to be subtracted from row i and column j in order 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 $\langle 5, 3 \rangle$, $\Delta = 1, 2, 11, 0, 3, 3$ and 11 respectively. So, either of the edges $\langle 3, 1 \rangle$ or $\langle 5, 3 \rangle$ may be used. Let us assume that LCBB selects edge $\langle 3, 1 \rangle$. $\hat{c}(2)$ (Figure 8.16) may be computed in a manner similar to that for the state space tree of Figure 8.13. 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 Figures 8.17(a) and (b). The $\hat{c}$ values for nodes 2 and 3 (as well as for all other nodes) appears outside the respective node.

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

(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 8.17 Reduced cost matrices for Figure 8.16

Node 2 is the next E-node. Now, for edges $\langle 1, 4 \rangle$, $\langle 2, 5 \rangle$, $\langle 4, 5 \rangle$, $\langle 5, 2 \rangle$ and $\langle 5, 3 \rangle$, $\Delta = 3, 2, 3, 3$ and 11 respectively. Edge $\langle 5, 3 \rangle$ is selected and nodes 4 and 5 generated. The corresponding reduced matrices are given in Figures 8.17(c) and (d). $\hat{c}(4)$ becomes 28 as we need to subtract 3 from column 2 in order to reduce this column. Note that entry (1, 5) has been set to ∞ in Figure 8.17(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 Δ 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 8.17(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 (an exercise 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. U is updated to 28. Node 3 is the next E-node. LCBB terminates now as $\hat{c}(3) = 36 > U$.

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 8.13. 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 and generate all of them, picking up 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.

The exercises examine yet another LC branch-and-bound algorithm for the traveling salesperson problem. This algorithm also uses a dynamic state space tree. Associated with each node in the state space tree is a graph. Each node represents a subproblem requiring us to find a minimum length tour in the graph associated with that node. The original graph G = (V, E) is associated with the root node. A lower bound $\hat{c}$ on the length of a shortest tour in the graph H = (V, A) associated with any node X is obtained by solving the following *assignment problem*:

$$\text{minimize} \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij}x_{ij}$$

$$\text{subject to} \sum_{i=1}^{n} x_{ij} = 1,\ 1 \le j \le n$$

$$\sum_{j=1}^{n} x_{ij} = 1,\ 1 \le i \le n \qquad (8.2)$$

$$x_{ij} = 0 \quad \text{if} \quad \langle i, j\rangle \notin A$$

$$x_{ij} = 0 \text{ or } 1, \qquad 1 \le i \le n,\ 1 \le j \le n$$

Note that $|V| = n$ and c_{ij} is the length of edge $\langle i, j\rangle$. $c_{ij} = \infty$ if $\langle i, j\rangle \notin E$. Algorithms to solve the assignment problem (8.2) are discussed in the texts: Linear Programming (pp. 227-228) by S. Gass, McGraw-Hill, New York, 1969 and Flows in Networks (pp. 111-112) by L. Ford and D. Fulkerson, Princeton University Press, 1962.

In case the solution to (8.2) is a tour then the length of a shortest tour in H has been obtained. Usually, however, the solution to (8.2) will be made up to several disjoint cycles. One of these cycles is used to partition the solution space of H. Let C be any one of the cycles in a solution to (8.2) (assume there are at least two cycles). Let $W = \{w_1, w_2, \ldots, w_r\}$ be the vertices in C. Define R_i and $\bar{R}_i$ as:

$$R_i = \{(w_i, j) \mid j \in W\}$$

$$\bar{R}_i = \{(w_i, j) \mid j \notin W\}$$

Now, define the edge sets:

$$E_1 = A - R_1$$
$$E_2 = A - \bar{R}_1 - R_2$$
$$E_3 = A - \bar{R}_1 - \bar{R}_2 - R_3$$
$$\vdots$$
$$E_r = A - \bar{R}_1 - \bar{R}_2 \ldots - R_r$$

The children of X correspond to the graphs (V, E_i), $1 \leq i \leq r$. The correctness of this partitioning rule follows from the following theorem:

Theorem 8.4 [Garfinkel] If T is a tour in H then T is a tour in exactly one of the graphs (V, E_i), $1 \leq i \leq r$.

Proof: Left as an exercise. □

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 best. The exercises describe computer experiments that determine empirically the relative performance of the strategies suggested.

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:

(i) Will the use of a better starting value for U always decrease the number of nodes generated?
(ii) Is it possible to decrease the number of nodes generated by actually expanding some nodes with $\hat{c}(\ ) > U$?
(iii) Will the use of a better $\hat{c}$ always result in a decrease in (or at least will not increase) the number of nodes generated? ($\hat{c}_2$ is better than $\hat{c}_1$ iff $\hat{c}_1(X) \leq \hat{c}_2(X) \leq c(X)$ for all nodes X).
(iv) Does the use of dominance relations ever result in the generation of more nodes than will otherwise be generated?

In this section we shall answer these questions. While the answers to most of the questions examined will agree with our intuition, the answers to others will be contrary to intuition. However, even in cases where the answer does not agree with intuition we can expect the performance of the algorithm to generally agree with the intuitive expectations. All of 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 8.5 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 $\hat{c}(X) \geq U$ where U is the current upper bound on the cost of a minimum cost solution node in T.

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

Theorem 8.6 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. 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 8.7 The use of a better $\hat{c}$ function in conjunction with FIFO and LIFO branch-and-bound algorithms will not increase the number of nodes generated.

Proof: Left as an exercise. □

Theorem 8.8 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 8.18. 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 $\begin{pmatrix} \hat{c}_1 \\ \hat{c}_2 \end{pmatrix}$. 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 9 nodes of the tree will get generated. When $\hat{c}_1$ is used, nodes 4, 5 and 6 are not generated. □

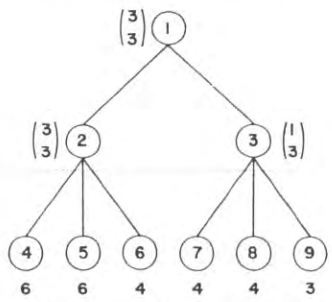


Figure 8.18 Example tree for Theorem 8.8

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 may 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), $(i_3, i_4) \cdots (i_n, i_1)$. 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 will donote the identity relation $\{(i, i) \mid 1 \leq i \leq n\}$.

Theorem 8.9 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: Just consider the state space tree of Figure 8.19. 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. $I = \{(i, i) | i \in D\}$. □

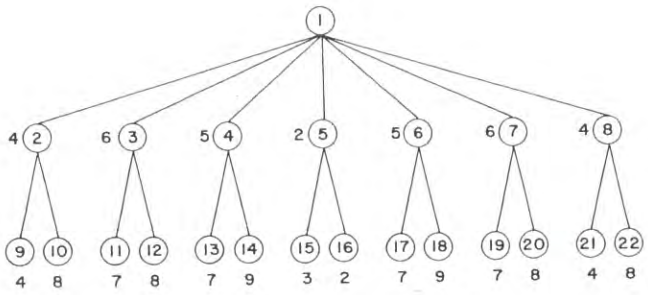


Figure 8.19 Example tree for Theorem 8.9

Theorem 8.10 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 8.11 If the condition $\hat{c}(i) < \hat{c}(j)$ in Theorem 8.10 is removed then an LC branch-and-bound using D_1 may generate fewer nodes than one using D_2.

Proof: Left as an exercise. □.

REFERENCES AND SELECTED READINGS

LC branch-and-bound algorithms have been extensively studied by researchers in areas such as artificial intelligence and operations research. Some of the more interesting general references are:

"*Problem Solving Methods in Artificial Intelligence*" by N. J. Nilsson, McGraw-Hill, New York, 1971.

"*Integer Programming*," by R. S. Garfinkel and G. L. Nemhauser, John Wiley and Sons, Inc., New York, 1972.

"Branch-and-bound methods: a survey," by E. L. Lawler and D. W. Wood, *Oper. Res.*, 14, pp. 699-719, 1966.

"Branch-and-bound methods: general formulation and properties," by L. Mitten, *Oper. Res.*, 18, pp. 24-34, 1970.

Branch-and-bound algorithms using dominance relations in a manner similar to that suggested by FIFOKNAP (resulting in DKNAP1) may be found in:

"A dynamic programming approach to sequencing problems," by M. Held and R. Karp, *Jr. of SIAM*, 10, pp. 196-210, 1962.

"Algorithms for scheduling independent tasks," by S. Sahni, *J. ACM*, 23(1), pp. 116-127, 1976.

"Exact and approximate algorithms for scheduling nonidentical processors," by E. Horowitz and S. Sahni, *J. ACM*, 23, pp. 317-327, 1976.

"General techniques for combinatorial approximation," by S. Sahni, *Oper. Res.*, 25(6), pp. 920-936, 1977.

"Branch-and-bound strategies for dynamic programming," by T. Morin and R. Marsten, *Oper. Res.*, 24, pp. 611-627, 1976.

The algorithms in the above five papers are very similar to dynamic programming type algorithms. Further branch-and-bound algorithms for scheduling problems appear in:

"Sequencing by enumerative methods," by J. Lenstra, *Math Centre. Tract 69*, Mathematisch Centrum, Amsterdam, 1976.

"Job-shop scheduling by implicit enumeration," by B. Lageweg, J. Lenstra and A. Rinnooy Kan, *Manag. Sci.*, 24(4), pp. 441-450, 1977.

"Application of the branch-and-bound technique to some flow-shop scheduling problems," by E. Ignall and L. Schrage, *Oper. Res.*, 13, pp. 400-412, 1965.

The reduction technique for the knapsack problem is due to Ingargiola and Korsh. It appears in:

"A reduction algorithm for zero-one single knapsack problems," by G. Ingargiola and J. Korsh, *Manag. Sci.*, 20(4), pp. 460-663, 1973.

A related reduction technique may be found in:

"A general algorithm for one dimensional knapsack problems," by G. Ingargiola and J. Korsh, *Oper. Res.*, 25(5), pp. 752-759, 1977.

Branch-and-bound algorithms for the traveling salesperson problem have been proposed by many researchers. A survey of these algorithms appears in:

"The traveling salesman problem: a survey," by M. Bellmore and G. Nemhauser, *Oper. Res.*, 16, pp. 538-558, 1968.

The reduced matrix technique to compute $\hat{c}$ is due to Little, Murty, Sweeny and Karel. It appears in the paper:

"An algorithm for the traveling salesman problem," by J. Little, K. Murty, D. Sweeny and C. Karel, *Oper Res.*, 11(6), pp. 972-989, 1963.

The above paper uses the dynamic state space tree approach. The partitioning scheme (8.3) is due to Garfinkel. His work is reported in:

"On partitioning the feasible set in a branch-and-bound algorithm for the asymmetric traveling salesman problem," by R. Garfinkel, *Oper. Res.*, 21(1), pp. 340-342, 1973.

A more efficient branch-and-bound algorithm for the traveling salesperson problem has been proposed by Held and Karp. Their algorithm can be used only when $c_{ij} = c_{ji}$ for all i and j. The following two papers describe the algorithm:

"The traveling salesman problem and minimum spanning trees," by M. Held and R. Karp, *Oper. Res.*, 18, pp. 1138-1162, 1970.

"The traveling salesman problem and minimum spanning trees: part II," by M. Held and R. Karp, *Math Prog.*, 1, pp. 6-25, 1971.

The results of section 8.4 are based on the work of Kohler, Steiglitz and Ibaraki. The relevent papers are:

"Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems," by W. Kohler and K. Steiglitz, *J. ACM* 21(1), pp. 140-156, 1974.

"Computational efficiency of approximate branch-and-bound algorithms," by T. Ibaraki, *Math of Oper. Res.*, 1(3), pp. 287-298, 1976.

"Theoretical comparisons of search strategies in branch-and-bound algorithms," by T. Ibaraki, *Int. Jr. of Comp. and Info. Sci.*, 5(4), pp. 315-344, 1976.

"On the computational efficiency of branch-and-bound algorithms," by T. Ibaraki, *Jr. of the Oper. Res. Soc. of Japan*, 20(1), pp. 16-35, 1977.

"The power of dominance relations in branch-and-bound algorithms," by T. Ibaraki, *J. ACM*, 24(2), pp. 264-279, 1977.

The papers by T. Ibaraki cited above also contain a discussion of heuristic search. More ideas on heuristic search can be found in N. Nilsson's book which was cited earlier.

EXERCISES

1. Prove Theorem 8.1.

2. Write a program schema DFBB, for a LIFO branch-and-bound search for a least cost answer node.

3. Draw the portion of the state space tree generated by FIFOBB, LCBB and a LIFO branch-and-bound for the job sequencing with deadlines instance $n = 5$; $(p_1, p_2, \ldots, p_5) = (6, 3, 4, 8, 5)$; $(t_1, t_2, \ldots, t_5) = (2, 1, 2, 1, 1)$; $(d_1, d_2, \ldots, d_5) = (3, 1, 4, 2, 4)$. 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.

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

5. Work out Example 8.2 using the variable tuple size formulation.

6. Work out Example 8.3 using the variable tuple size formulation.

7. Draw the portion of the state space tree generated by LCKNAP for the knapsack instances:

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

8. Do problem 7 using a LC branch-and-bound on a dynamic state space tree (see Section 7.6). Use the fixed tuple size formulation.

9. Write a LC branch-and-bound algorithm for the knapsack problem using the fixed tuple size formulation and the dynamic state space tree of Section 7.6.

10. [Programming Project] Program algorithms DKNAP (Alg. 5.7), DKNAP1 (see pag. 401), LCKNAP (Alg. 8.8), and BKNAP2 (Alg. 7.13). Compare these algorithms empirically using randomly generated data as below:

    Data Set

    (i) Random w_i and p_i, $w_i \in [1, 100]$, $p_i \in [1, 100]$, $M = \Sigma_1^n w_i/2$.
    (ii) Random w_i and p_i, $w_i \in [1, 100]$, $p_i \in [1, 100]$; $M = 2\max\{w_i\}$
    (iii) Random w_i, $w_i \in [1, 100]$; $p_i = w_i + 10$; $M = \Sigma_1^n w_i/2$
    (iv) Same as (iii) except $M = 2*\max\{w_i\}$
    (v) Random p_i, $p_i \in [1, 100]$; $w_i = p_i + 10$; $M = \Sigma_1^n w_i/2$
    (vi) Same as (v) except $M = 2*\max\{w_i\}$

    Obtain computing times for $n = 5, 10, 20, 30, 40, \ldots$. For each n generate (say) 10 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 $p_i = w_i$, $1 \le i \le n$, $M = \Sigma w_i/2$ and $\Sigma w_i x_i \neq M$ for any 0, 1 assignment to the x_i's. Obtain computing times for your four programs for $n = 10$, 20 and 30.

    If you still have computer time available, then study the effect of changing the range to [1, 1000] in data sets (i) through (vi). In sets (iii) to (vi) replace $p_i = w_i + 10$ by $p_i = w_i + 100$ and $w_i = p_i + 10$ by $w_i = p_i + 100$ respectively.

11. [Programming Project] (a) Program the reduction heuristic REDUCE of Section 8.2. Generate several problem instances from the data sets of Exercise 10 and determine the size of the reduced problem instances. Use $n = 100$, 200, 500 and 1000.

(b) Program DKNAP1 and the backtracking algorithm BKNAP2 for the knapsack problem. Compare the effectiveness of REDUCE by running several problem instances (as in Exercise 10). Obtain average and worst case computing times for DKNAP1 and BKNAP2 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 conclusions can you draw from your experiments?

12. a) Write a branch-and-bound algorithm for the job sequencing with deadlines problem using a dominance rule. 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 computer 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... bounding functions and dominance rules are removed; PROGB... dominance rule is removed; PROGC ... bounding function is removed and PROGD ... bounding functions and dominance rules are included. Determine both computing time figures as well as the number of nodes generated.

13. 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 8.11 and $\hat{c}(\cdot)$ 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.
    d) Solve the above traveling salesperson instance using the assignment problem formulation. Draw the state space tree and describe the progress of the method from node to node.
    e) Solve the given traveling salesperson problem using backtracking and the same $\hat{c}(\cdot)$ function as above. Use the static state space tree formulation.
    f) Do part e) using a dynamic state space tree.

14. Do problem 13 using the following traveling salesperson cost matrix:

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

15. a) Describe an efficient implementation for a LC branch-and-bound traveling salesperson problem using the reduced cost matrix approach and (i) a dynamic state space tree and (ii) a static tree as in Figure 8.11.

b) Are there any problem instances for which the LC branch-and-bound will generate fewer nodes using a static tree than using a dynamic tree? Prove your answer.

16. Consider the LC branch-and-bound 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.

17. [Programming Project]. Write computer programs for the following traveling salesperson algorithms:

i) the dynamic programming algorithm of chapter 5
ii) a backtracking algorithm using the static tree formulation of Section 8.3
iii) a backtracking algorithm using the dynamic tree formulation of Section 8.3
iv) a LC branch-and-bound algorithm corresponding to (ii)
v) a LC branch-and-bound algorithm corresponding to (iii)

Design data sets to be used to compare the efficiency of the above algorithms. Randomly generate problem instances from each of these data sets and obtain computing times for your programs. Obtain tables along the lines of those in Section 7.6. What conclusions can you draw from your computing times?

18. Prove theorem 8.4.

19. Prove theorem 8.6.

20. Prove theorem 8.7.

21. Prove theorem 8.10.

22. Prove theorem 8.11.

23. [Heuristic Search] Heuristic search is a generalization of FIFO, LIFO and LC search. 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 to. 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.