Chapter 8

BRANCH-AND-BOUND

8.1 THE METHOD

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

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

Example 8.1 [4-queens] Let us see how a FIFO branch-and-bound algorithm would search the state space tree (Figure 7.2) for the 4-queens problem. Initially, there is only one live node, node 1. This represents the case in which no queen has been placed on the chessboard. This node becomes the E-node. It is expanded and its children, nodes 2, 18, 34, and 50, are generated. These nodes represent a chessboard with queen 1 in row 1 and columns 1, 2, 3, and 4 respectively. The only live nodes now are nodes 2, 18, 34, and 50. If the nodes are generated in this order, then the next E-node is node 2. It is expanded and nodes 3, 8, and 13 are generated. Node 3 is immediately killed using the bounding function of Example 7.5. Nodes 8 and 13 are added to the queue of live nodes. Node 18 becomes the next E-node. Nodes 19, 24, and 29 are generated. Nodes 19 and 24 are killed as a result of the bounding functions. Node 29 is added to the queue of live

nodes. The E-node is node 34. Figure 8.1 shows the portion of the tree of Figure 7.2 that is generated by a FIFO branch-and-bound search. Nodes that are killed as a result of the bounding functions have a "B" under them. Numbers inside the nodes correspond to the numbers in Figure 7.2. Numbers outside the nodes give the order in which the nodes are generated by FIFO branch-and-bound. At the time the answer node, node 31, is reached, the only live nodes remaining are nodes 38 and 54. A comparison of Figures 7.6 and 8.1 indicates that backtracking is a superior search method for this problem. □

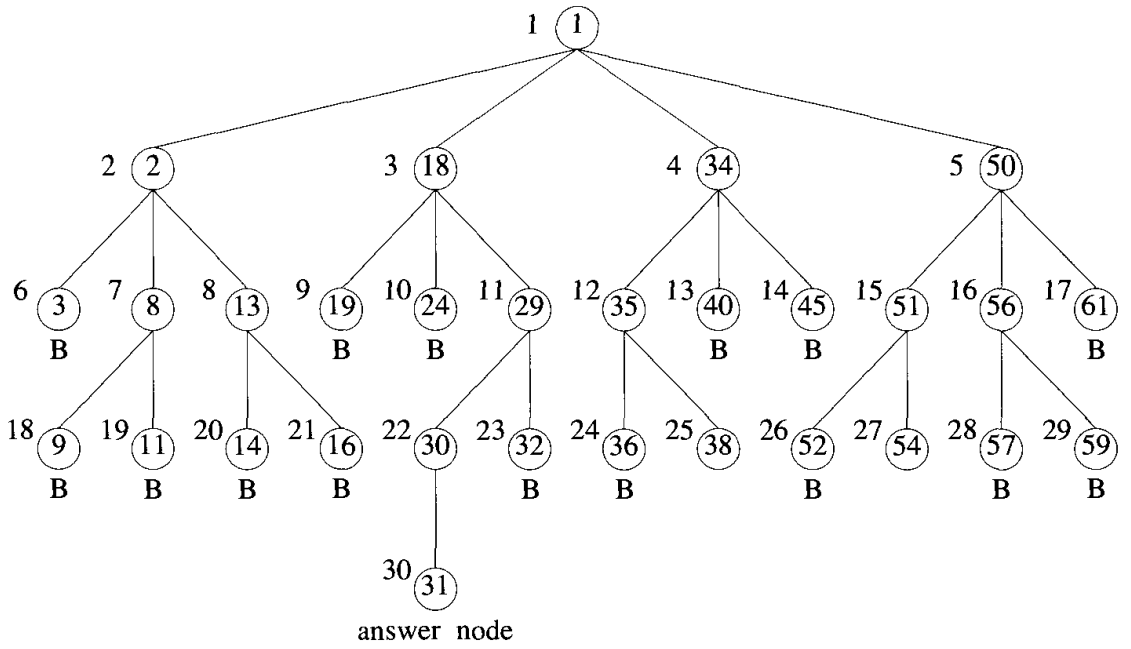


Figure 8.1 Portion of 4-queens state space tree generated by FIFO branch-and-bound

8.1.1 Least Cost (LC) Search

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

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

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

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

Using $f(\cdot) \equiv 0$ usually biases the search algorithm to make deep probes into the search tree. To see this, note that we would normally expect $\hat{g}(y) \leq \hat{g}(x)$ for y, a child of x. Hence, following x, y will become the E-node, then one of y's children will become the E-node, next one of y's grandchildren will become the E-node, and so on. Nodes in subtrees other than the subtree x will not get generated until the subtree x is fully searched. This would not

be a cause for concern if $\hat{g}(x)$ were the true cost of x. Then, we would not wish to explore the remaining subtrees in any case (as x is guaranteed to get us to an answer node quicker than any other existing live node). However, $\hat{g}(x)$ is only an estimate of the true cost. So, it is quite possible that for two nodes w and z, $\hat{g}(w) < \hat{g}(z)$ and z is much closer to an answer node than w. It is therefore desirable not to overbias the search algorithm in favor of deep probes. By using $f(\cdot) \not\equiv 0$, we can force the search algorithm to favor a node z close to the root over a node w which is many levels below z. This would reduce the possibility of deep and fruitless searches into the tree.

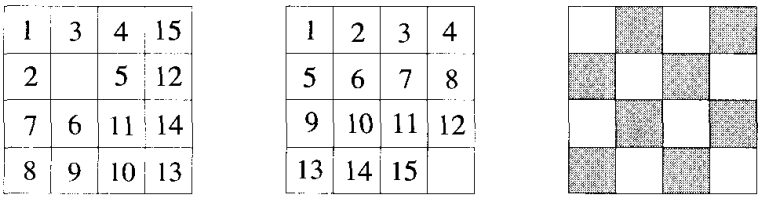
A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$. Hence, such a search strategy is called an LC-search (**L**east **C**ost search). It is interesting to note that BFS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

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

8.1.2 The 15-puzzle: An Example

The 15-puzzle (invented by Sam Loyd in 1878) consists of 15 numbered tiles on a square frame with a capacity of 16 tiles (Figure 8.2). We are given an initial arrangement of the tiles, and the objective is to transform this arrangement into the goal arrangement of Figure 8.2(b) through a series of legal moves. The only legal moves are ones in which a tile adjacent to the empty spot (ES) is moved to ES. Thus from the initial arrangement of Figure 8.2(a), four moves are possible. We can move any one of the tiles numbered 2, 3, 5, or 6 to the empty spot. Following this move, other moves can be made. Each move creates a new arrangement of the tiles. These arrangements are called the *states* of the puzzle. The initial and goal arrangements are called the initial and goal states. A state is reachable from the initial state iff there is a sequence of legal moves from the initial state to this state. The state space of an initial state consists of all states that can be reached from the initial state. The most straightforward way to solve the puzzle would be to search the state space for the goal state and use the

path from the initial state to the goal state as the answer. It is easy to see that there are 16! ($16! \approx 20.9 \times 10^{12}$) different arrangements of the tiles on the frame. Of these only one-half are reachable from any given initial state. Indeed, the state space for the problem is very large. Before attempting to search this state space for the goal state, it would be worthwhile to determine whether the goal state is reachable from the initial state. There is a very simple way to do this. Let us number the frame positions 1 to 16. Position i is the frame position containing tile numbered i in the goal arrangement of Figure 8.2(b). Position 16 is the empty spot. Let $position(i)$ be the position number in the initial state of the tile numbered i. Then $position(16)$ will denote the position of the empty spot.



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

Figure 8.2 15-puzzle arrangements

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

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

Proof: Left as an exercise. □

Theorem 8.1 can be used to determine whether the goal state is in the state space of the initial state. If it is, then we can proceed to determine a sequence of moves leading to the goal state. To carry out this search, the state space can be organized into a tree. The children of each node x in this tree represent the states reachable from state x by one legal move. It is convenient to think of a move as involving a move of the empty space rather than a move of a tile. The empty space, on each move, moves either up, right, down, or left. Figure 8.3 shows the first three levels of the state

space tree of the 15-puzzle beginning with the initial state shown in the root. Parts of levels 4 and 5 of the tree are also shown. The tree has been pruned a little. *No node p has a child state that is the same as p's parent.* The subtree eliminated in this way is already present in the tree and has root *parent(p)*. As can be seen, there is an answer node at level 4.

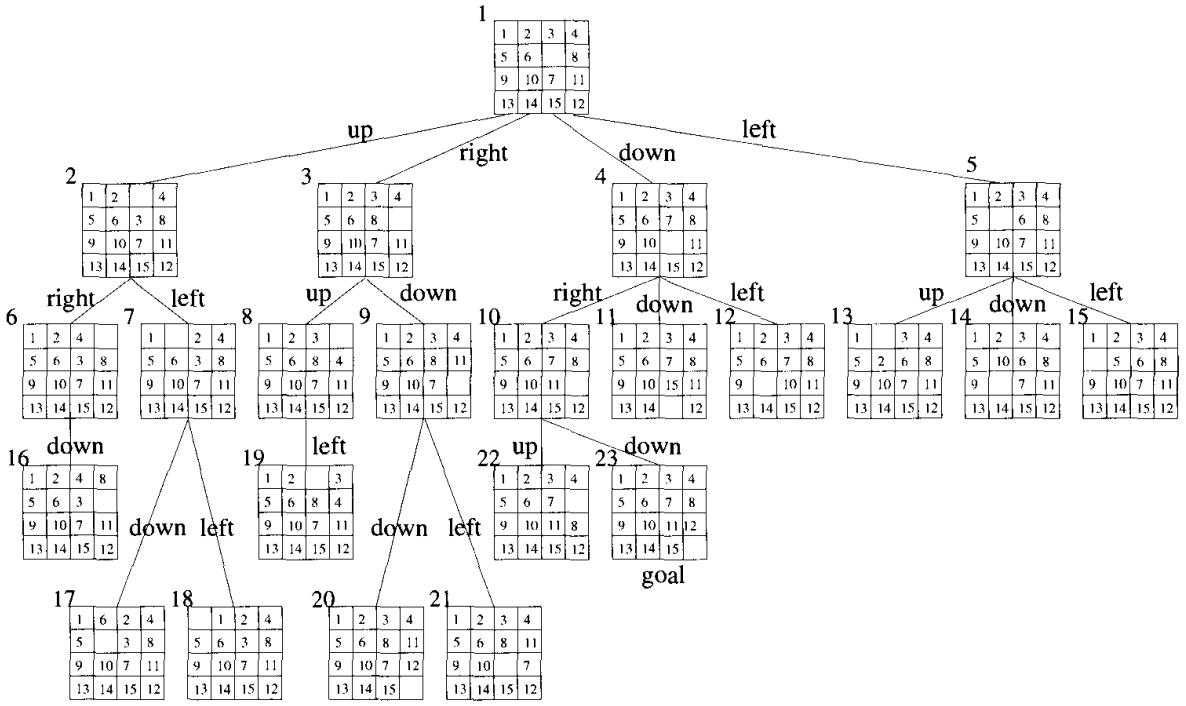


Figure 8.3 Part of the state space tree for the 15-puzzle

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

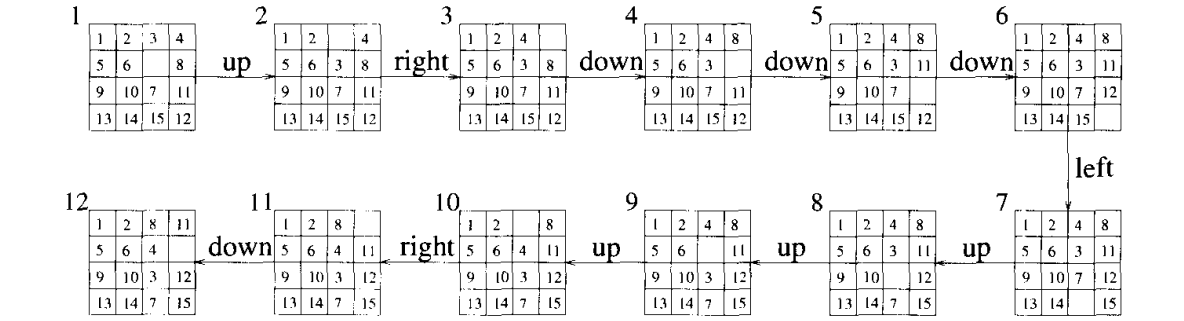


Figure 8.4 First ten steps in a depth first search

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

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

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

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

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

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

Figure 8.5 Problem state

8.1.3 Control Abstractions for LC-Search

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