4

"AI". by Winston, 1984

Exploring Alternatives

The purpose of this chapter is to understand how to deal with situations in which one choice leads to another, presenting us with a *search* problem. In contrast with the choices in chapter 3, the choices here are inherently ordered.

Search problems are ubiquitous, popping up everywhere Artificial Intelligence researchers and students go. In this chapter, the examples illustrate procedures used in route finding, recipe discovery, and game playing. In other chapters, there are examples involving problem reduction, rule-based problem solving, theorem proving, property inheritance, sentence analysis, obstacle avoidance, and learning.

Figure 4-1 is a roadmap for our trip through the space of possibilities. (1) First, we study the simple, basic procedures. These are *depth-first search*, *hill climbing*, *breadth-first search*, *beam search*, and *best-first search*. They are used to find paths from starting positions to goal positions when the length of the discovered paths is not important.

(2) Second, we study more complicated procedures that find shortest paths. These procedures are *the British Museum procedure*, *branch and bound*, *discrete dynamic programming*, and *A**. They are used when the cost of traversing a path is of primary importance, as when a trip-planning system is mapping out a route for a salesman. All but the British Museum procedure aspire to do their work efficiently.

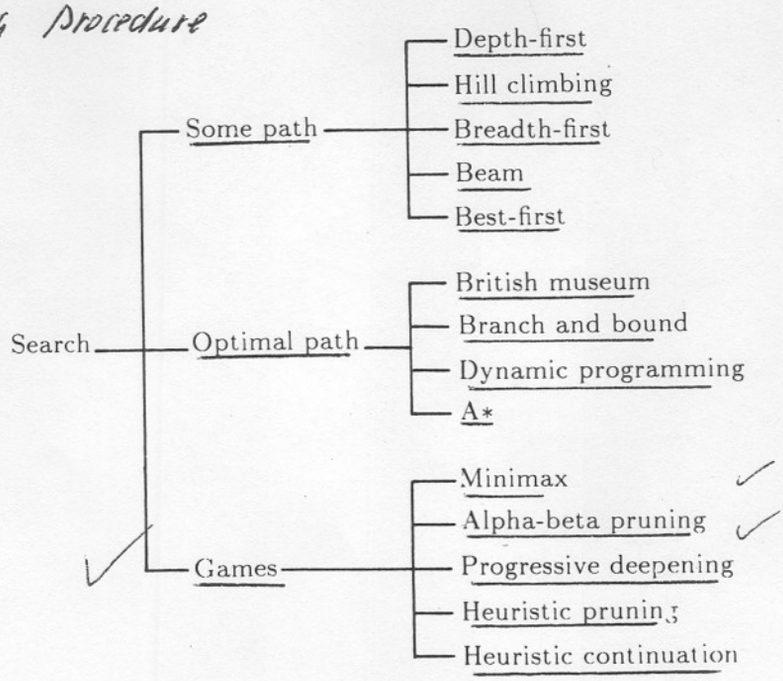

Figure 4-1. Search procedures. Many procedures address the problem of finding satisfactory paths. Others concentrate on the harder problem of finding optimal paths. Procedures for games differ from ordinary path-finding procedures, because games involve adversaries.

Third, we explore some special-case procedures that are appropriate when facing an adversary. These procedures are *minimax search*, *alpha-beta pruning*, *progressive deepening*, *heuristic pruning*, and *heuristic continuation*. They are common in programs that play board games, particularly checkers and chess.

In studying this chapter, you will develop a repertoire of search procedures. To use your repertoire well, you must develop skill in answering questions like the following ones:

- Which search procedures work, given the nature of the problem in hand?
- Which procedures are efficient?
- Which procedures are easy to implement?
- Is search the best thing to think about?

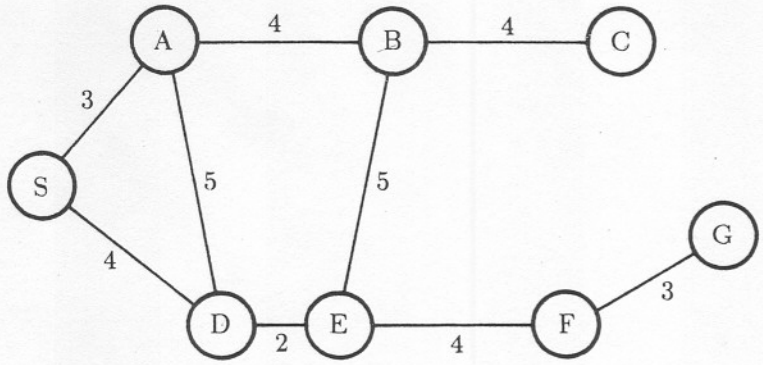

Figure 4-2. A basic search problem. A path is to be found from the start node, S, to the goal node, G. Search procedures explore nets like these, learning about connections and distances as they go.

1. [FINDING PATHS]

Suppose we want to find some path through a net of cities connected by highways, such as the net shown in figure 4-2. The path is to begin at city S, the starting point, and it is to end at city G, the final goal.

Finding a path involves two kinds of effort:

- First, there is the effort expended in *finding* either some path or the shortest path.
- And, second, there is the effort actually expended in *traversing* the path.

If it is necessary to go from S to G often, then it is worth a lot to find a really good path. On the other hand, if only *one* trip is required, and if the net is hard to force a way through, then it is proper to be content as soon as *some* path is found, even though better ones could be found with more work. For the moment we will consider only the problem of finding one path. We will return to finding optimal paths later.

The most obvious way to find a solution is to devise a bookkeeping scheme that allows orderly exploration of all possible paths. It is useful to note that the bookkeeping scheme must not allow itself to cycle in the net. It would be senseless to go through a sequence like S–A–D–S–A–D–... over and over again. With cyclic paths terminated, nets are equivalent to trees. The tree shown in figure 4-3 is made from the net in figure 4-2 by following each possible path outward from the net's starting point until it runs into a place already visited.

By convention, the places in a net or tree are called *nodes*. In nets, the connections between nodes are called *links*, and in trees, the connections are called *branches*. Also, it is common to talk about trees using terms borrowed from genealogy. Branches directly connect *parents* with *children*.

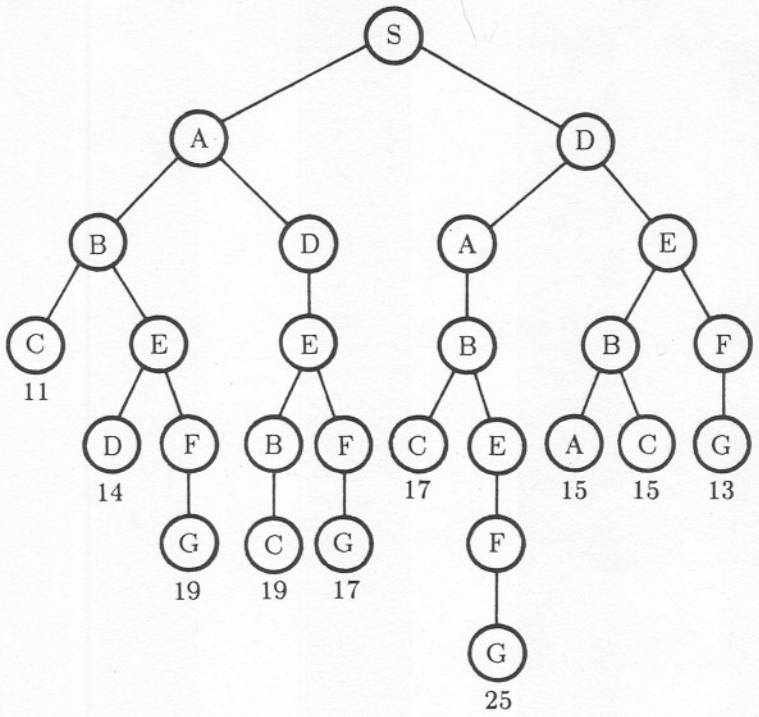

Figure 4-3. A tree made from a net. Nets are made into trees by tracing out all possible paths to the point where they reenter previously visited nodes. Node S is the root node. Node S is also a parent node, with its children being A and D, and an ancestor node, with all other nodes in the tree being descendants. The nodes with no children are the terminal nodes. The numbers beside the terminal nodes are accumulated distances.

The node at the top of a tree, the one with no parent, is called the *root node.* The nodes at the bottom, the ones with no children, are called *terminal nodes.* One node is the *ancestor* of another, a *descendant,* if there is a chain of branches between the two.

Finally, if the number of children is always the same for every node that has children, that number is said to be the *branching factor.*

Drawing in the children of a node is called *expanding* the node. Nodes are said to be *open* until they are expanded, whereupon they become *closed.*

If no node in a net is to be visited twice, there can be no more than n levels in the corresponding tree, where n is the total number of nodes, eight in the map traversal example. In the example, the goal is reached at the end of four distinct paths, each of which has a total path length given by adding up a few distances.

Depth-first Search Dives into the Search Tree

Given that one path is as good as any other, one no-fuss idea is to pick an alternative at every node visited and work forward from that alternative. Other alternatives at the same level are ignored completely as long as there is hope of reaching the destination using the original choice. This is the essence of *depth-first search*. Using a convention that the alternatives are tried in left-to-right order, the first action in working on the situation in figure 4-3 is a headlong dash to the bottom of the tree along the leftmost branches.

But since a headlong dash leads to terminal node C, without encountering G, the next step is to back up to the nearest ancestor node with an unexplored alternative. The nearest such node is B. The remaining alternative at B is better, bringing eventual success through E in spite of another dead end at D. Figure 4-4 shows the nodes encountered.

If the path through E had not worked out, then the procedure would move still further back up the tree seeking another viable decision point to move forward from. On reaching A, movement would go down again, reaching the destination through D.

Having seen an example of depth-first search, let us write out a procedure:

To conduct a depth-first search:

1 Form a one-element queue consisting of the root node.

2 Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node.

  2a If the first element is the goal node, do nothing.

  2b If the first element is not the goal node, remove the first element from the queue and add the first element's children, if any, to the *front* of the queue.

3 If the goal node has been found, announce success; otherwise announce failure.

Be warned: depth-first search can be dangerous. Imagine a tree in which C is the gateway to a vast subnetwork instead of the end of a short dead-end path. Depth-first movement through such a tree would slip past the levels at which the goal node appears and waste incredible energy in exhaustively exploring parts of the tree lower down. For such trees, depth-first search is the worst possible approach.

DFS

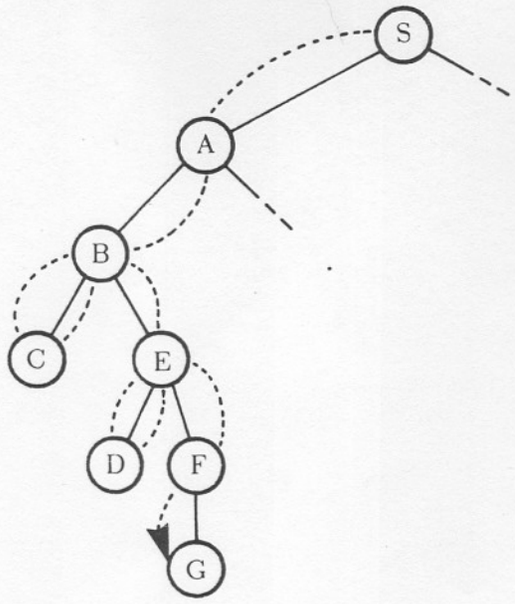

Figure 4-4. An example of depth-first search. One alternative is selected and pursued at each node until the goal is reached or a node is reached where further downward motion is impossible. When further downward motion is impossible, search is restarted at the nearest ancestor node with unexplored children.

Hill climbing

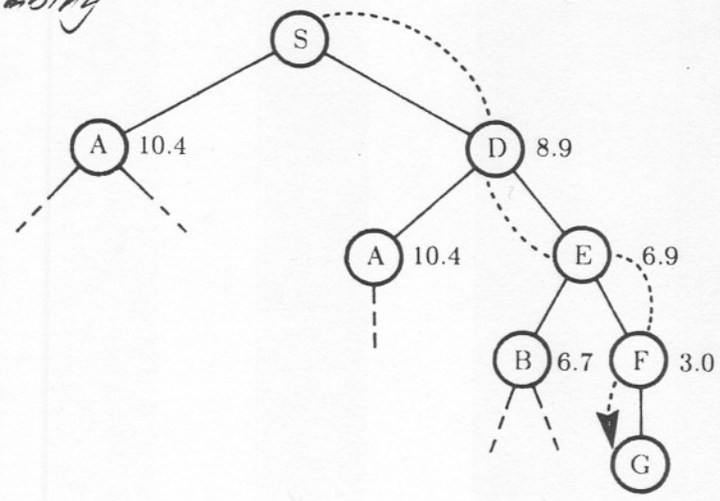

Figure 4-5. An example of hill climbing. Hill climbing is depth-first search with a heuristic measurement that orders choices as nodes are expanded. The numbers beside the nodes are straight-line distances to the goal node.

(2) Quality Measurements Turn Depth-first Search into Hill Climbing

Search efficiency may improve spectacularly if there is some way of ordering choices so that the most promising are explored first. In many situations, simple measurements can be made to determine a reasonable ordering.

[To move through a tree of paths using *hill climbing*, proceed as in depth-first search, but order the choices according to some heuristic measure of remaining distance. The better the heuristic measure is, the better hill climbing will be relative to ordinary depth-first search.]

Straight-line, as-the-crow-flies distance is an example of a heuristic measure of remaining distance. Figure 4-5 shows what happens when hill climbing is used on the map-traversal problem using as-the-crow-flies distance to order choices.

From a procedural point of view, hill climbing differs from depth-first search in only one detail, the added, italicized part:

To hill climb:

1 Form a one-element queue consisting of the root node.

2 Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node.

  2a If the first element is the goal node, do nothing.

  2b If the first element is not the goal node, remove the first element from the queue, *sort the first element's children, if any, by estimated remaining distance*, and add the first element's children, if any, to the front of the queue.

3 If the goal node has been found, announce success; otherwise announce failure.

A form of hill climbing is also used in parameter optimization. Here are some examples:

- On entering a room you find the temperature uncomfortable. Walking over to the thermostat, you find, to your surprise, that you cannot tell how to set it because the temperature markings have been obliterated.
- The picture on your TV set has deteriorated over a period of time. You must adjust the tuning, color, tint, and brightness controls for a better picture.
- You are halfway up a mountain when a dense fog comes in. You have no map or trail to follow but you do have a compass and a determination to get to the top.

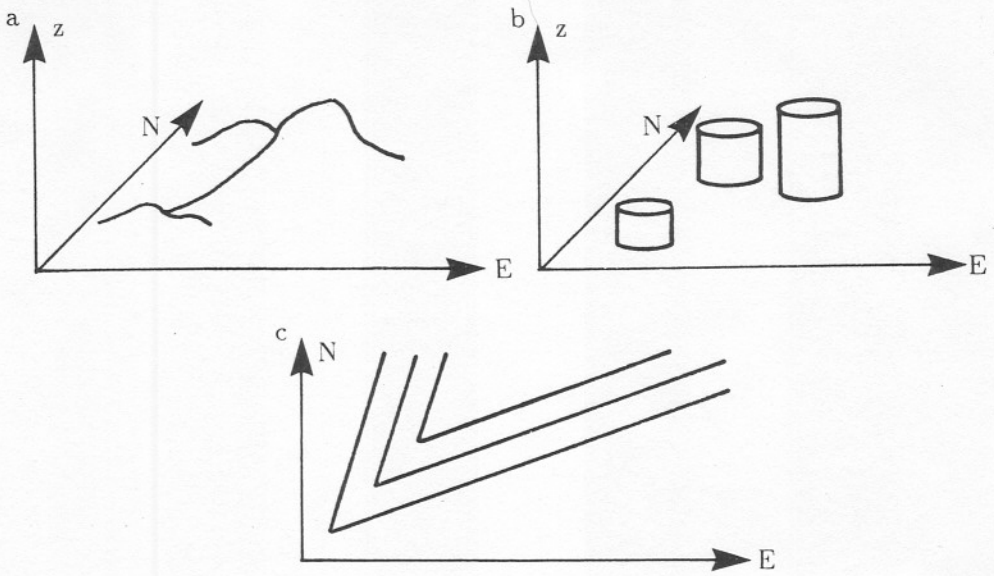

Figure 4-6. Hill climbing is a bad idea in difficult terrain. In *a*, foothills stop progress. In *b*, plains cause aimless wandering. In *c*, with the terrain described by a contour map, all ridge points look like peaks because all four east-west and north-south probe directions lead to lower quality measurements.

Each of these problems conforms to an abstraction in which there are adjustable parameters and a way of measuring the quality associated with any particular set of values for the parameters. Instead of an explicit goal, however, the procedure stops when a node is reached where all the node's children have lower quality measurements.

In the temperature example, the adjustable parameter is the thermostat setting, and the goodness is the resulting degree of comfort. In the TV example, there are various knobs, each of which interacts with the others to determine overall picture quality. And, of course, in the mountaineering example, position is adjustable, and movement is either up or down as position changes.

Thus, to move through a space of parameter values using parameter-oriented hill climbing, take one step in each of a fixed set of directions, move to the best alternative found, and repeat until reaching a point that is better than all of the surrounding points reached by one-step probes.

Although simple, parameter optimization via hill climbing suffers from various problems. The most severe of these problems are the foothill problem, the plateau problem, and the ridge problem:

- The *foothill problem* occurs whenever there are secondary peaks, as in figure 4-6a. The secondary peaks draw the hill-climbing procedure like

magnets. An optimal point is found, but it is local, not global, and the user is left with a false sense of accomplishment.

- The *plateau problem* comes up when there is mostly a flat area separating the peaks. In extreme cases, the peaks may look like telephone poles sticking up in a football field, as in figure 4-6b. The local improvement operation breaks down completely. For all but a small number of positions, all standard-step probes leave the quality measurement unchanged.
- The *ridge problem* is more subtle and, consequently, more frustrating. Suppose we are standing on what seems like a knife edge running generally from northeast to southwest, as in figure 4-6c. A contour map shows that each standard step takes us down even though we are not at any sort of maximum, local or global. Increasing the number of directions used for the probing steps may help.

In general, the foothill, plateau, and ridge problems are greatly exacerbated as the number of parameter dimensions increases.

(3) Breadth-first Search Pushes Uniformly into the Search Tree BFS

When depth-first search and hill climbing are bad choices, *breadth-first search* may be useful. Breadth-first search looks for the goal node among all nodes at a given level before using the children of those nodes to push on. In the situation shown in figure 4-7, node D would be checked just after A. The procedure would then move on, level by level, discovering G on the fourth level down from the root level.

Like hill-climbing, a procedure for breadth-first search resembles the one for depth-first search, differing only in the place where new elements are added to the queue.

To conduct a breadth-first search:

1 Form a one-element queue consisting of the root node.

2 Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node.

  2a If the first element is the goal node, do nothing.

  2b If the first element is not the goal node, remove the first element from the queue and add the first element's children, if any, to the *back* of the queue.

3 If the goal node has been found, announce success; otherwise announce failure.

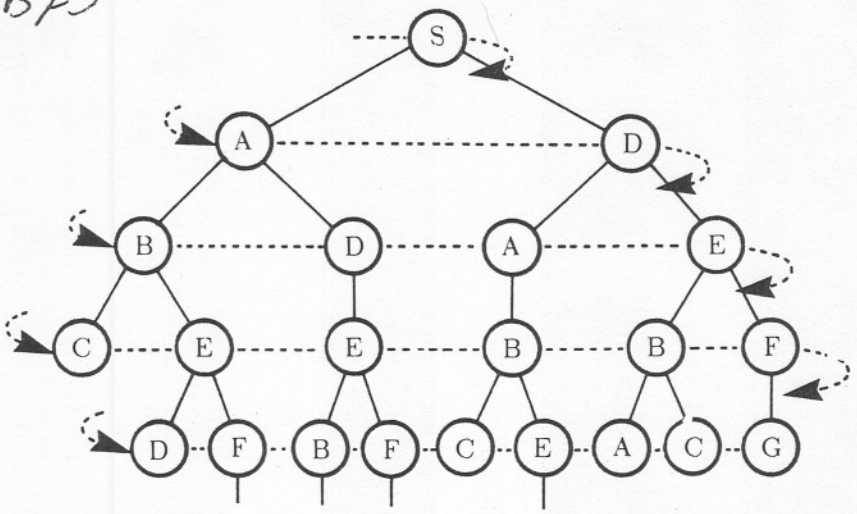

Figure 4-7. An example of breadth-first search. Downward motion proceeds level by level until the goal is reached.

Breadth-first search will work even in trees that are infinitely deep or effectively infinite. On the other hand, breadth-first search is wasteful when all paths lead to the destination node at more or less the same depth.

(4) Beam Search Expands Several Partial Paths and Purges the Rest

Beam search is like breadth-first search because beam search progresses level by level. Unlike breadth-first search, however, beam search only moves downward from the best w nodes at each level. The other nodes are ignored.

Consequently, the number of nodes explored remains manageable, even if there is a great deal of branching and the search is deep. If beam search of width w is used in a tree with branching factor b, there will be only wb nodes under consideration at any depth, not the explosive number there would be if breadth-first search were used. Figure 4-8 illustrates how beam search would handle the map-traversal problem.

(5) Best-first Search Expands the Best Partial Path

Recall that when forward motion is blocked, hill climbing demands forward motion from the last choice through the seemingly best child node. In *best-first search*, forward motion is from the best open node so far, no matter where it is in the partially developed tree. Best-first search works like a team of cooperating mountaineers seeking out the highest point in a mountain range: they maintain radio contact, move the highest subteam forward at all times, and divide subteams into sub-subteams at path junctions.

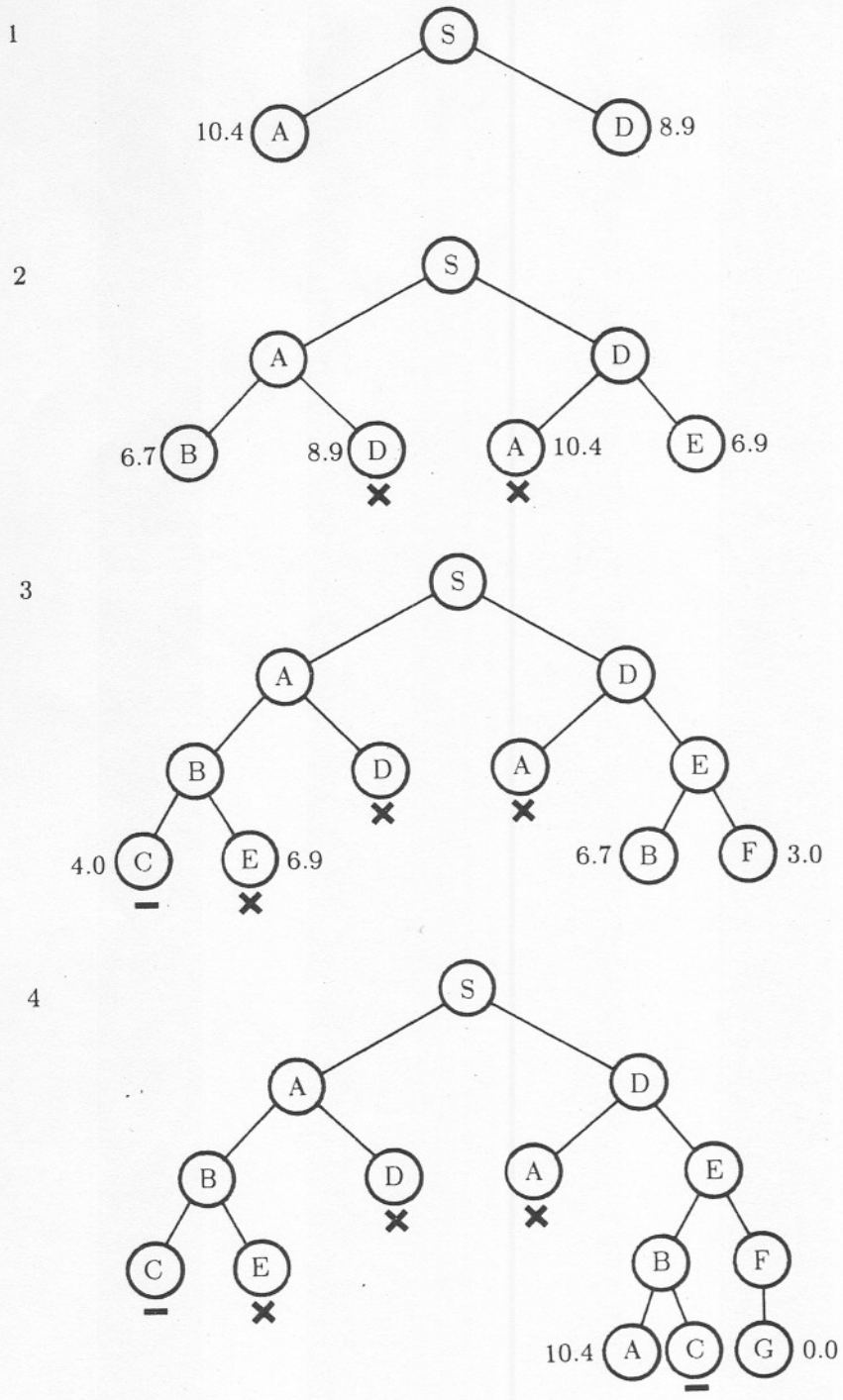

Figure 4-8. An example of beam search. Investigation spreads through the search tree level by level, but only the best w nodes are expanded, where $w = 2$ here. The numbers beside the nodes are straight-line distances to the goal node.

In the particular map-traversal problem we have been using, hill climbing and best-first search coincidentally explore the search tree in the same way.

The paths found by best-first search are more likely to be shorter than those found with other methods, because best-first search always moves forward from the node that seems closest to the goal node. Note that more likely does not mean certain, however.

Like hill climbing, the best-first search procedure requires sorting. This time, however, the entire queue must be sorted.

To conduct a best-first search:

1 Form a one-element queue consisting of the root node.

2 Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node.

  2a If the first element is the goal node, do nothing.

  2b If the first element is not the goal node, remove the first element from the queue, add the first element's children, if any, to the queue, and *sort the entire queue by estimated remaining distance.*

3 If the goal node has been found, announce success; otherwise announce failure.

Search May Lead to Discovery

Finding physical paths and tuning parameters are only two applications for search ideas. More generally, the nodes in a search space may denote abstract entities, not just physical places or parameter settings.

Suppose, for example, that you are wild about cooking, particularly about creating your own omelet recipes. Deciding to be more systematic about your discovery procedure, you make a list of *recipe transformations* for varying existing recipes:

- Merge two similar recipes. That is, combine together half the amount of each of the ingredients from each recipe.
- Substitute something similar for a key ingredient.
- Double the amount of a flavoring.
- Halve the amount of a flavoring.
- Add a new flavoring.
- Eliminate a flavoring.

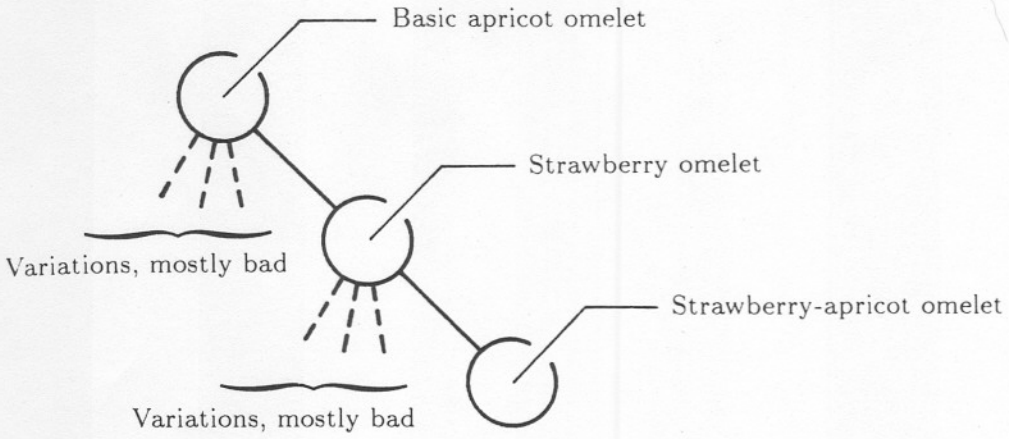

Figure 4-9. A search tree with recipe nodes. *Recipe transformations* build the tree; *interestingness heuristics* guide the best-first search to the better prospects.

Naturally, you speculate that most of the changes suggested by these recipe transformations will turn out awful, unworthy of further development. Consequently you need *interestingness heuristics* to help decide which recipes to continue to work on. Here are some interestingness heuristics:

- It tastes good.
- It looks good.
- Your friends eat a lot of it.
- Your friends ask for the recipe.

Interestingness heuristics can be used with hill climbing, with beam search, or with best-first search.

Figure 4-9 shows some of the search tree descending from a basic recipe for an apricot omelet, one similar to a particular favorite of the fictional detective and gourmand Nero Wolfe:

Apricot Omelet Recipe

1	ounce kümmel
1	cup apricot preserves
6	eggs
2	tablespoons cold water
1/2	teaspoon salt
2	teaspoons sugar
2	tablespoons unsalted butter
1	teaspoon powdered sugar

Using the substitution transformation on the apricot preserves enables the creation of many things, like apple, blueberry, cherry, orange, peach, pineapple, and strawberry omelets. Then, once we have more than one recipe, the merge transformation can be applied, producing, for example, an apricot-strawberry omelet.

Of course, to make a real recipe generator, we would have to be much better at generating plausible transformations, for we would waste too many eggs otherwise. This is consistent with the following general principles:

- More knowledge means less search.
- Search is seductive. While generally involved in many tasks, tuning a search procedure is rarely the right thing to do. More often the right thing is to improve understanding, thereby reducing the need for search.

The point of the omelet illustration, however, is that search is, nevertheless, one ingredient of the discovery procedure. The domain of discovery can be concrete, as in the world of cooking, or abstract, as in the world of mathematical concepts.

There Are Many Search Alternatives

Choices: We have seen that there are many ways for doing search, each with advantages, among them the following:

DFS
- Depth-first search is good when blind alleys do not get too deep.

BFS
- Breadth-first search is good when the number of alternatives at the choice points is not too large.

Hill Climbing
- Hill climbing is good when there is a natural measure of goal distance and a good choice is likely to be among the good-looking choices at each choice point.

Beam
- Beam search is good when there is a natural measure of goal distance and a good path is likely to be among the good-looking partial paths at all levels.

Best-FS
- Best-first search is good when there is a natural measure of goal distance and a good path may look bad at shallow levels.

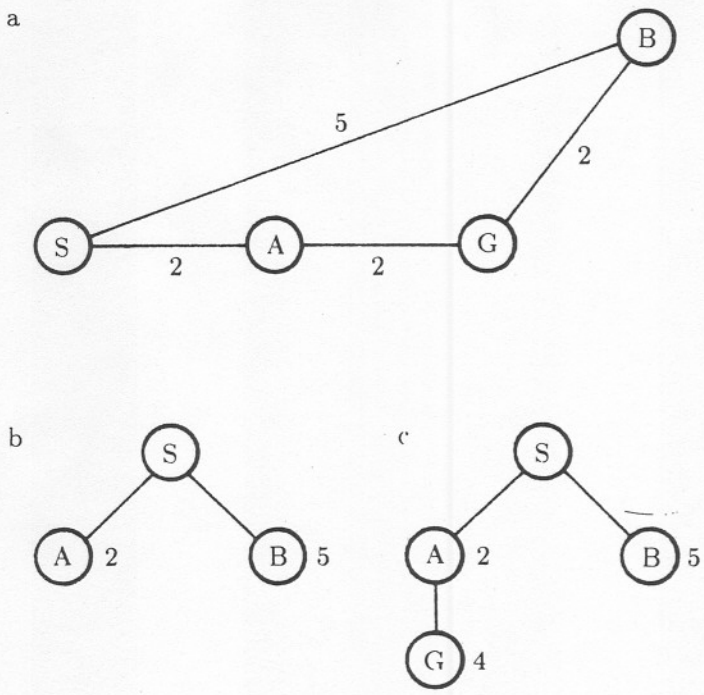

Figure 4-10. In branch-and-bound search, the node expanded is the one at the end of the shortest path leading to an open node. Expansion continues until there is a path reaching the goal that is of length equal to or shorter than all incomplete paths terminating at open nodes. A sample net is shown in *a*, along with partially developed search trees in *b* and *c*. The numbers beneath the nodes in the trees are accumulated distances. In *b*, node A might just as well be expanded, for even if a satisfactory path through B is found, there may be a shorter one through A. In *c*, however, it makes no sense to expand node B, because there is a complete path to the goal that is shorter than the path ending at B.

II. FINDING THE BEST PATH

In this section, we continue to explore the map-traversal problem, but now with attention to path length. In the end, we will bring together several distinct ideas to form the A* procedure.

(1) The British Museum Procedure Looks Everywhere

One procedure for finding the shortest path through a net is to find all possible paths and to select the best from them. This plodding procedure, named in jest, is known as the *British Museum procedure.*

To find all possible paths, either a depth-first search or a breadth-first search will work, with one modification: search does not stop when the first

path to the goal is found. If the breadth and depth of the tree are small, as in the map-traversal example, there are no problems.

Unfortunately, the size of search trees is often large, making any procedure for finding all possible paths extremely unpalatable. Suppose that instead of a few levels there is a moderately large number. Suppose further that the branching is completely uniform and that the number of alternative branches at each node is b. Then in the first level there will be b nodes. For each of these b nodes there will be b more nodes in the second level, or b^2. Continuing this leads to the conclusion that the number of nodes at depth d must be b^d. For even modest breadth and depth, the number of paths can be large: $b = 10$ and $d = 10$ yields 10^{10} = ten billion paths. Fortunately, there are strategies that enable optimal paths to be found without finding all possible paths first.

(2) Branch-and-bound Search Expands the Least-cost Partial Path

One way to find optimal paths with less work is by using *branch-and-bound search.* The basic idea is simple. Suppose an optimal solution is desired for the net shown in figure 4-10a. Looking only at the first level, in figure 4-10b, the distance from S to node A is clearly less than the distance to B. Following A to the destination at the next level reveals that the total path length is 4, as shown in figure 4-10c. But this means there is no point in calculating the path length for the alternative path through node B since at B the incomplete path's length is already 5 and hence longer than the path for the known solution through A.

More generally, the branch-and-bound scheme works like this: During search there are many incomplete paths contending for further consideration. The shortest one is extended one level, creating as many new incomplete paths as there are branches. These new paths are then considered along with the remaining old ones, and again, the shortest is extended. This repeats until the destination is reached along some path. Since the shortest path was always chosen for extension, the path first reaching the destination is certain to be optimal.

There is a flaw in the explanation, as given. The last step in reaching the destination may be long enough to make the supposed solution longer than one or more incomplete paths. It might be that only a tiny step would extend one of the incomplete paths to the solution point. To be sure this is not so, a slightly better termination condition is needed. Instead of terminating when a path is found, terminate when the shortest incomplete path is longer than the shortest complete path.

Here, then, is the procedure with the proper terminating condition:

To conduct a branch-and-bound search:

1 Form a queue of partial paths. Let the initial queue consist of the zero-length, zero-step path from the root node to nowhere.

2 Until the queue is empty or the goal has been reached, determine if the first path in the queue reaches the goal node.

  2a If the first path reaches the goal node, do nothing.

  2b If the first path does not reach the goal node:

    2b1 Remove the first path from the queue.

    2b2 Form new paths from the removed path by extending one step.

    2b3 Add the new paths to the queue.

    2b4 Sort the queue by cost accumulated so far, with least-cost paths in front.

3 If the goal node has been found, announce success; otherwise announce failure.

Now look again at the map-traversal problem and see how branch-and-bound works there. Figure 4-11 illustrates the exploration sequence. In the first step, A and D are identified as the children of the only active node, S. The partial path distance of A is 3 and that of D is 4; A therefore becomes the active node. Then B and D are generated from A with partial path distances of 7 and 8. Now the first encountered D, with a partial path distance of 4, becomes the active node, leading to the generation of partial paths to A and E. At this point, there are four partial paths, with the path S–D–E being the shortest.

After the seventh step, partial paths S–A–D–E and S–D–E–F are the shortest partial paths. Expanding S–A–D–E leads to partial paths terminating at B and F. Expanding S–D–E–F, along the right side of the tree, leads to the complete path S–D–E–F–G, with a total distance of 13. This is the shortest path, but to be absolutely sure, it is necessary to extend two partial paths, S–A–B–E, with a partial path distance of 12, and S–D–E–B, with a partial path distance of 11. There is no need to extend the partial path S–D–A–B, since its partial path distance of 13 is equal to that of the complete path. In this particular example, little work is avoided relative to exhaustive search, British Museum style.

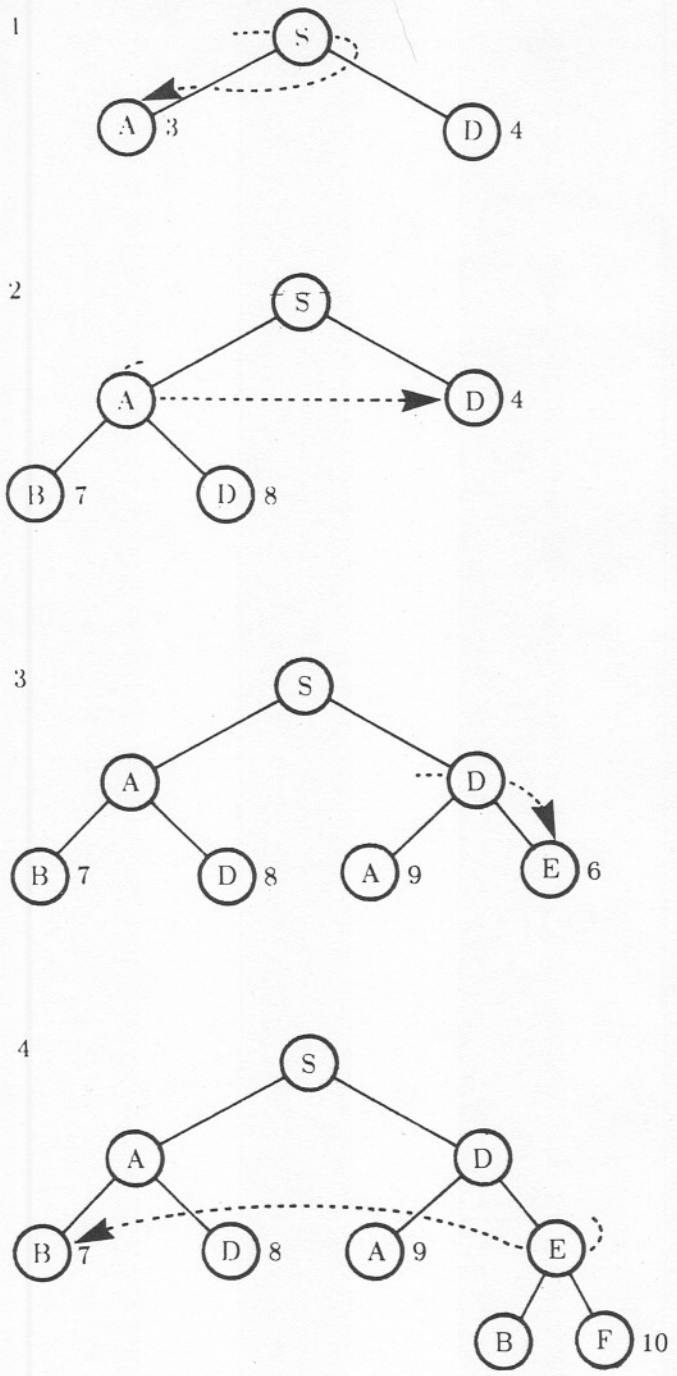

Figure 4-11. Branch-and-bound search determines that path S–D–E–F–G is optimal. The numbers beside the nodes are accumulated distances. Search stops when all partial paths to open nodes are as long as or longer than the complete path S–D–E–F–G.

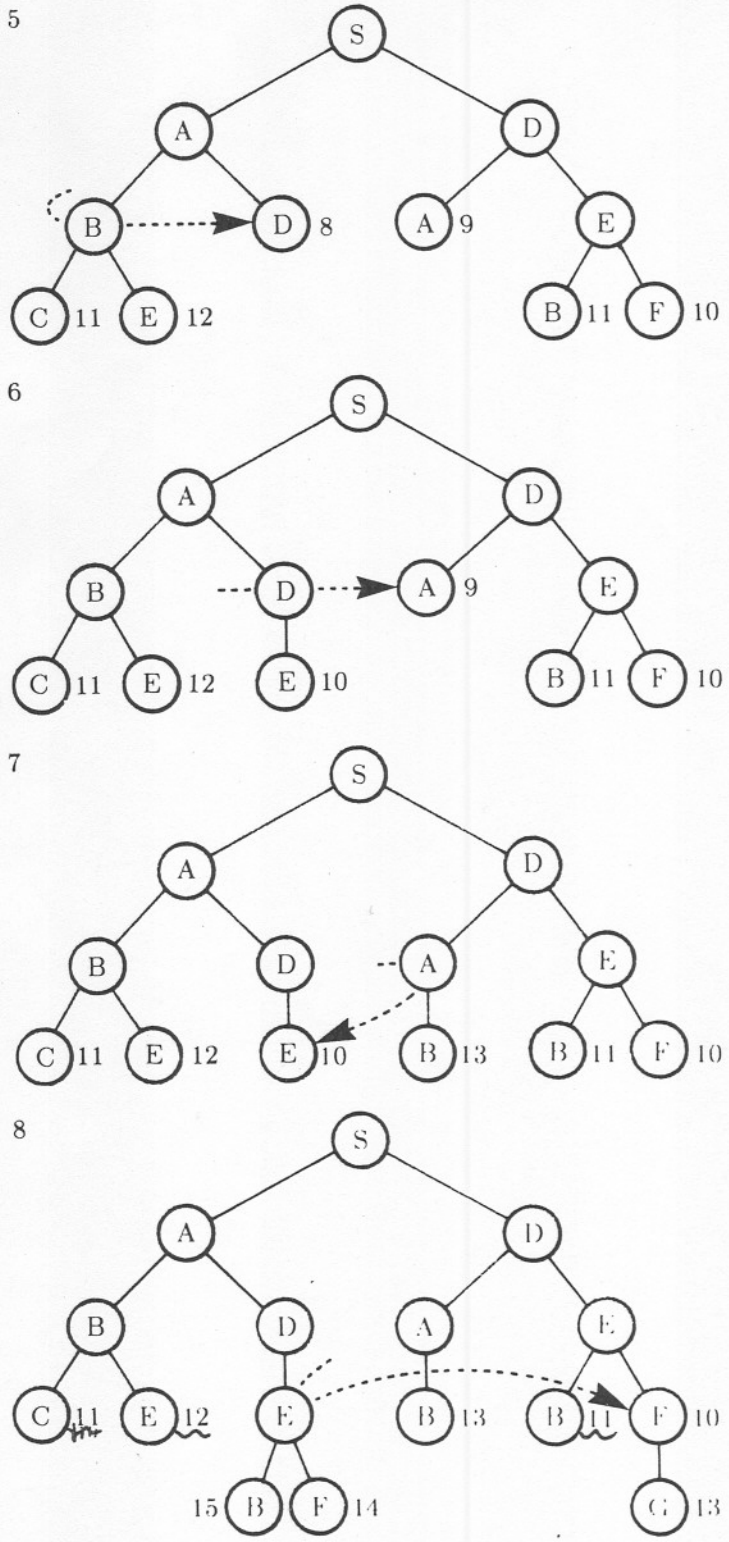

Figure 4-11. Continued.

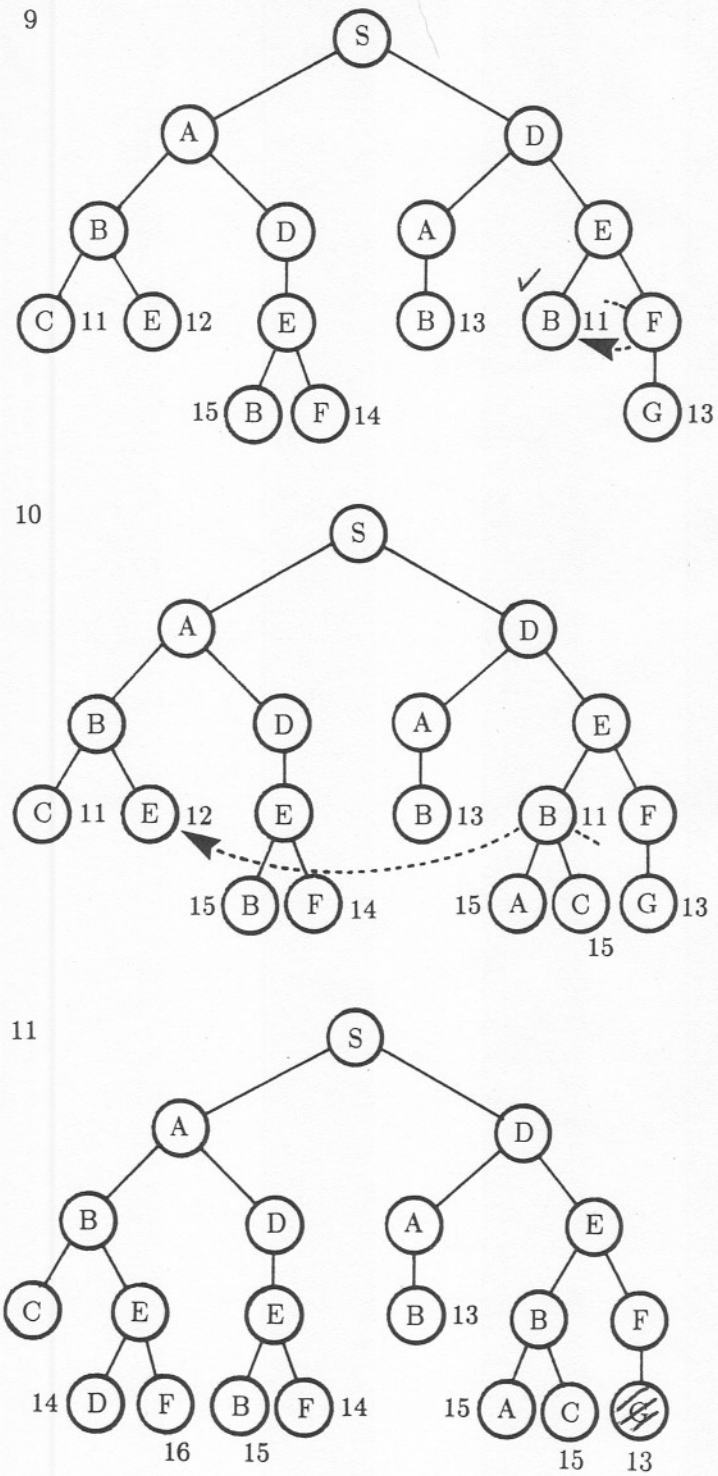

Figure 4-11. Continued.

Adding Underestimates Improves Efficiency

In some cases, branch-and-bound search can be improved greatly by using guesses about distances remaining as well as facts about distances already accumulated. After all, if a guess about distance remaining is good, then that guessed distance added to the definitely known distance already traversed should be a good estimate of total path length, e(total path length):

$$e(\text{total path length}) = d(\text{already traveled}) + e(\text{distance remaining})$$

where d(already traveled) is the known distance already traveled and where e(distance remaining) is an estimate of the distance remaining.

Surely it makes sense to work hardest on developing the path with the shortest estimated path length until the estimate changes upward enough to make some other path be the one with the shortest estimated path length. After all, if the guesses were perfect, this approach would keep us on the optimal path at all times.

In general, however, guesses are not perfect, and a bad overestimate somewhere along the true optimal path may cause us to wander off that optimal path permanently.

But note that *underestimates* cannot cause the right path to be overlooked. An underestimate of the distance remaining yields an underestimate of total path length, u(total path length):

$$u(\text{total path length}) = d(\text{already traveled}) + u(\text{distance remaining})$$

where d(already traveled) is the known distance already traveled and where u(distance remaining) is an underestimate of the distance remaining.

[Now if a total path is found by extending the path with the smallest underestimate repeatedly, no further work need be done once all incomplete path distance estimates are longer than some complete path distance. This is true because the real distance along a completed path cannot be shorter than an underestimate of the distance. If all estimates of remaining distance can be guaranteed to be underestimates, there can be no bungle]

In traveling through nets of cities, the straight-line distance is guaranteed to be an underestimate. Figure 4-12 shows how straight-line distance helps. As before, A and D are generated from S. This time D is the node to search from, since D's lower-bound path distance is 12.9, which is better than that for A, 13.4.

Expanding D leads to partial path S–D–A, with a lower-bound distance estimate of 19.4, and to partial path S–D–E, with a lower-bound distance estimate of 12.9. S–D–E is therefore the partial path to extend. The result is a path to B with a distance estimate of 17.7 and one to F with 13.0.

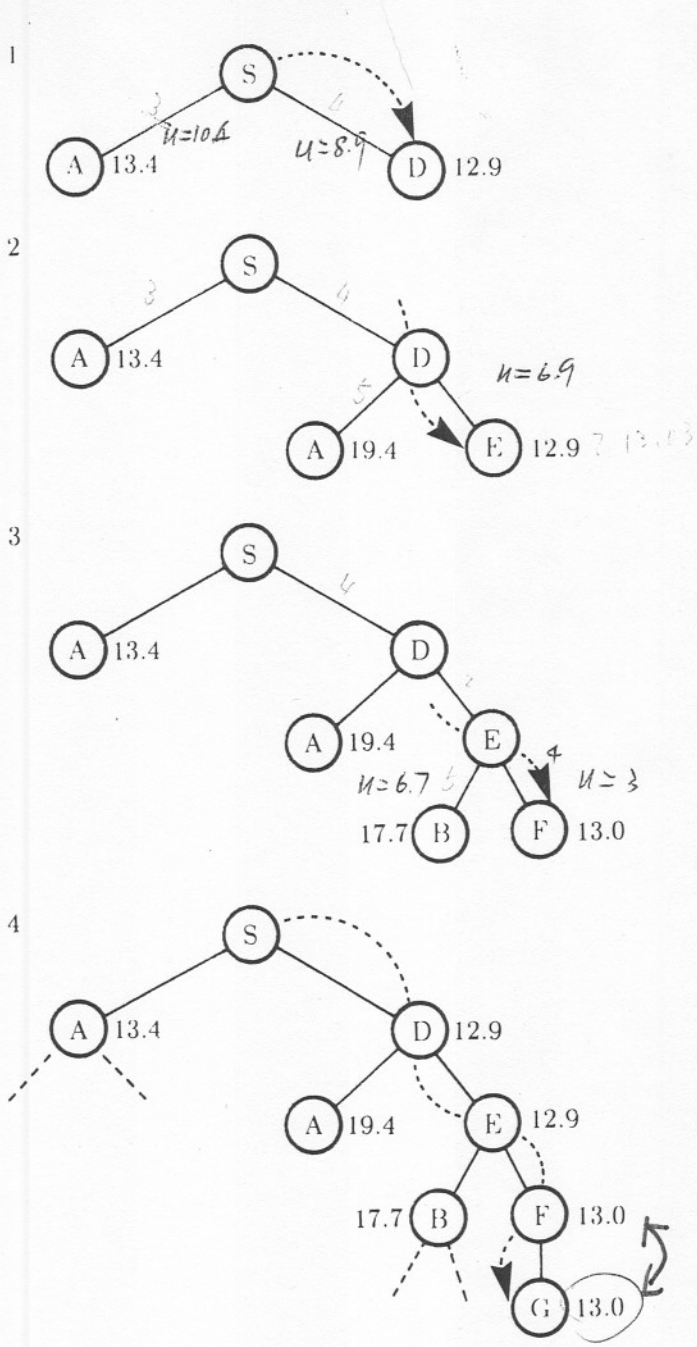

Figure 4-12. Branch-and-bound search augmented by underestimates determines that the path S–D–E–F–G is optimal. The numbers beside the nodes are accumulated distances plus underestimates of distances remaining. Underestimates quickly push up the lengths associated with bad paths. In this example, many fewer nodes are expanded than with branch-and-bound search operating without underestimates.

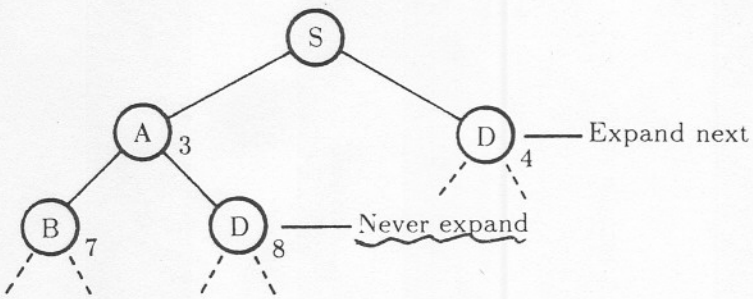

Figure 4-13. An illustration of the dynamic-programming principle. The numbers beside the nodes are accumulated distances. There is no point to expanding the instance of node D at the end of S–A–D because getting to the goal via the instance of D at the end of S–D is obviously better.

Expanding the partial path to F is the correct move since it is the partial path with the minimum lower-bound distance. This leads to a complete path, S–D–E–F–G, with a total distance of 13.0. No partial path has a lower-bound distance so low, so no further search is required.

In this particular example, a great deal of work is avoided. Here is the modified procedure:

To conduct a branch-and-bound search with underestimates:

1 Form a queue of partial paths. Let the initial queue consist of the zero-length, zero-step path from the root node to nowhere.

2 Until the queue is empty or the goal has been reached, determine if the first path in the queue reaches the goal node.

  2a If the first path reaches the goal node, do nothing.

  2b If the first path does not reach the goal node:

    2b1 Remove the first path from the queue.

    2b2 Form new paths from the removed path by extending one step.

    2b3 Add the new paths to the queue.

    2b4 Sort the queue by the *sum of* cost accumulated so far *and a lower-bound estimate of the cost remaining*, with least-cost paths in front.

3 If the goal node has been found, announce success; otherwise announce failure.

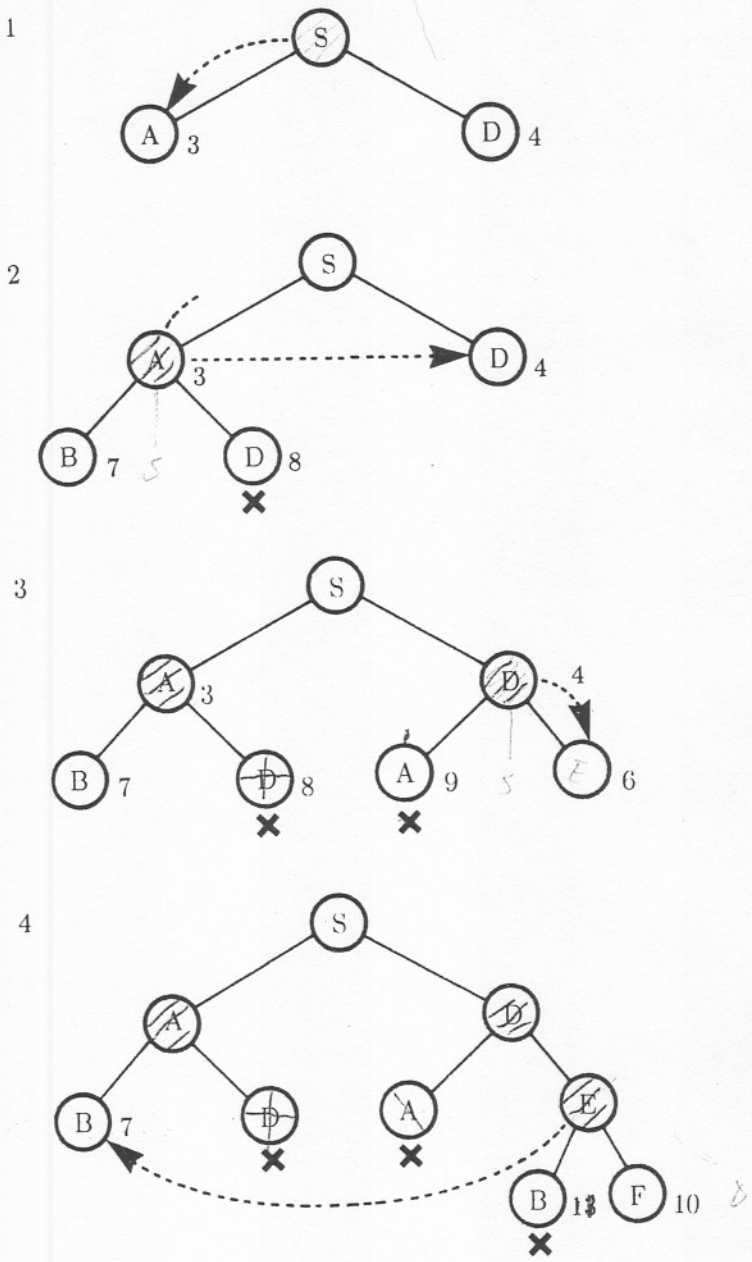

Figure 4-14. Branch-and-bound search, augmented by dynamic programming, determines that path S–D–E–F–G is optimal. The numbers beside the nodes are accumulated distances. Many nodes, those crossed out, are found to be redundant. Fewer nodes are expanded than with branch-and-bound search operating without dynamic programming.

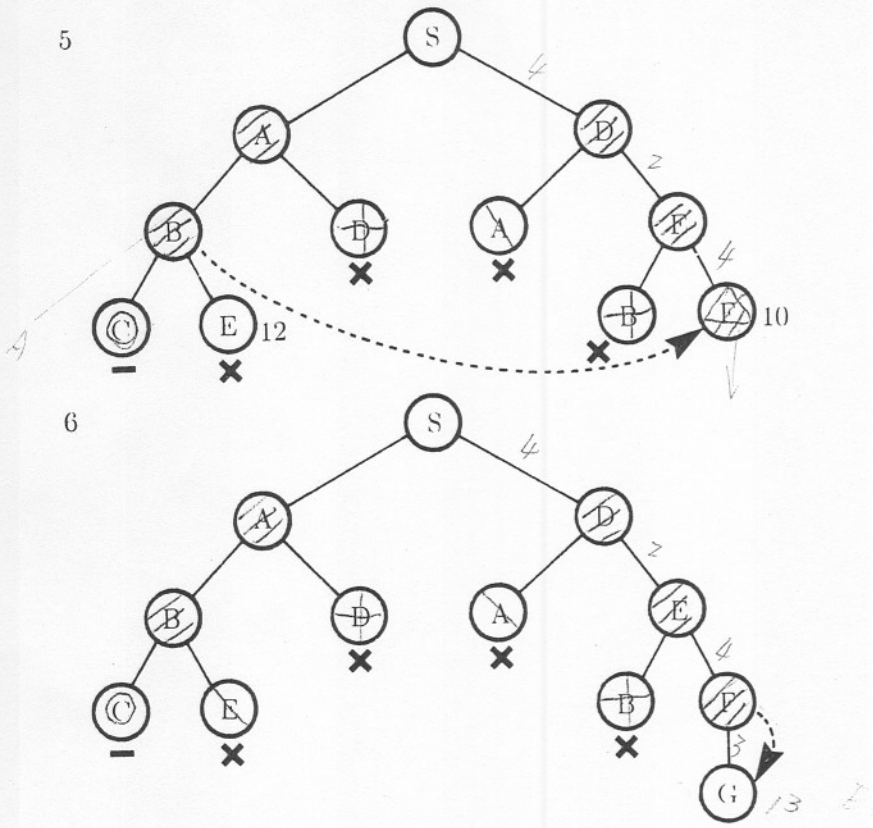

Figure 4-14. Continued.

Of course, the closer an underestimate is to the true distance, the better things will work, since if there is no difference at all, there is no chance of developing any false movement. At the other extreme, an underestimate may be so poor as to be hardly better than a guess of zero, which certainly must always be the ultimate underestimate of remaining distance. In fact, ignoring estimates of remaining distance altogether can be viewed as the special case in which the underestimate used is uniformly zero.

(3) Dynamic Programming

Discarding Redundant Paths Improves Efficiency

Now let us consider another way to improve on the basic branch-and-bound search. Look at figure 4-13. The root node, S, has been expanded, producing A and D. For the moment we use no underestimates for remaining path length. Since the path from S to A is shorter than the path from S to D, A has been expanded also, leaving three paths: S–A–B, S–A–D, and S–D.

Thus the path S–D will be the next path extended, since it is the partial path with the shortest length.

But what about the path S–A–D? Will it ever make sense to extend it? Clearly it will not. Since there is one path to D with length 4, it cannot make sense to work with another path to D with length 8. The path S–A–D should be forgotten forever; it cannot produce a winner.

Dynamic Programming Principle:

This illustrates a general truth: Assume that the path from a starting point, S, to an intermediate point, I, does not influence the choice of paths for traveling from I to a goal point, G. Then the minimum distance from S to G through I is the sum of the minimum distance from S to I and the minimum distance from I to G. Consequently, the *dynamic-programming principle* holds that when looking for the best path from S to G, all paths from S to any intermediate node, I, other than the minimum-length path from S to I, can be ignored. Here is the procedure:

To conduct a branch-and-bound search with dynamic programming:

1 Form a queue of partial paths. Let the initial queue consist of the zero-length, zero-step path from the root node to nowhere.

2 Until the queue is empty or the goal has been reached, determine if the first path in the queue reaches the goal node.

- 2a If the first path reaches the goal node, do nothing.
- 2b If the first path does not reach the goal node:
  - 2b1 Remove the first path from the queue.
  - 2b2 Form new paths from the removed path by extending one step.
  - 2b3 Add the new paths to the queue.
  - 2b4 Sort the queue by cost accumulated so far, with least-cost paths in front.
  - 2b5 *If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.*

3 If the goal node has been found, announce success; otherwise announce failure.

Figure 4-14 shows the effect of using the dynamic-programming principle, together with branch-and-bound search, on the map-traversal problem. Four paths are cut off quickly, leaving only the dead-end path to node C and the optimal path, S–D–E–F–G.

A* Is Improved Branch-and-bound Search

The *A* procedure* is branch-and-bound search, with an estimate of remaining distance, combined with the dynamic-programming principle. If the estimate of remaining distance is a lower bound on the actual distance, then A* produces optimal solutions. Generally, the estimate may be assumed to be a lower bound estimate, unless specifically stated otherwise, implying that A*'s solutions are normally optimal. Note the similarity between A* and branch-and-bound search with dynamic programming:

To do A* search with lower-bound estimates:

1 Form a queue of partial paths. Let the initial queue consist of the zero-length, zero-step path from the root node to nowhere.

2 Until the queue is empty or the goal has been reached, determine if the first path in the queue reaches the goal node.

- 2a If the first path reaches the goal node, do nothing.
- 2b If the first path does not reach the goal node:
  - 2b1 Remove the first path from the queue.
  - 2b2 Form new paths from the removed path by extending one step.
  - 2b3 Add the new paths to the queue.
  - 2b4 Sort the queue by the *sum of* cost accumulated so far *and a lower-bound estimate of the cost remaining*, with least-cost paths in front.
  - 2b5 *If two or more paths reach a common node, delete all those paths except for one that reaches the common node with the minimum cost.*

3 If the goal node has been found, announce success; otherwise announce failure.

There Are Many Optimal Search Alternatives

Choices :

We have seen that there are many ways for searching for optimal paths, each with advantages, among them the following:

- The British Museum procedure is good when the search tree is small.
- Branch-and-bound search is good when the tree is big and bad paths turn distinctly bad quickly.
- Branch-and-bound search with a guess is good when there is a good lower-bound estimate of the distance remaining to the goal.
- Dynamic programming is good when many paths reach common nodes.
- The A* procedure is good when both branch-and-bound search with a guess and dynamic programming are good.

III. [DEALING WITH ADVERSARIES] 2/15/86. 11:30 PM

Another sort of search is done in playing games like checkers and chess. The nodes in a game tree naturally represent board configurations, and they are linked by way of branches that transform one situation into another, as figure 4-15 illustrates. Of course, there is a new twist in that the decisions are made by two people, acting as adversaries, each making a decision in turn.

The *ply* of a game tree is p if the tree has p levels, including the root level. If the depth of a tree is d, then $p = d + 1$. In the chess literature, a *move* consists of one player's single act and his opponent's single response. Here, however, we will be informal, referring to one player's single act as his *move*.

Games Require Different Search Procedures

Using the British Museum procedure to search game trees is definitely out. For chess, for example, if we take the effective branching factor to be something like 16 and the effective depth to be 100, then the number of branches in an exhaustive survey of chess possibilities would be on the order of 10^{120}, a ridiculously large number. In fact, if all the atoms in the universe had been computing chess moves at picosecond speeds since the big bang (if any), the analysis would be just getting started.

At the other end of the spectrum, if only there were some infallible way to rank the members of a set of board situations, it would be a simple matter to play by selecting the move that leads to the best situation that can be reached by one move. No search would be necessary. Unfortunately, no such situation-ranking formula exists. When board situations obviously differ, then a simple measure like piece count can be a rough guide to quality, but depending on such a measure to rank the available moves from a given situation produces poor results. Some other strategy is needed.

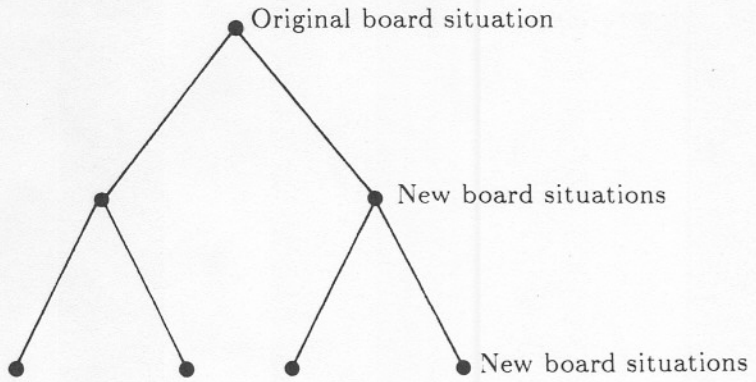

Figure 4-15. Games provide a search environment with a new twist, competition. The nodes represent game situations, and the branches represent the moves that connect them.

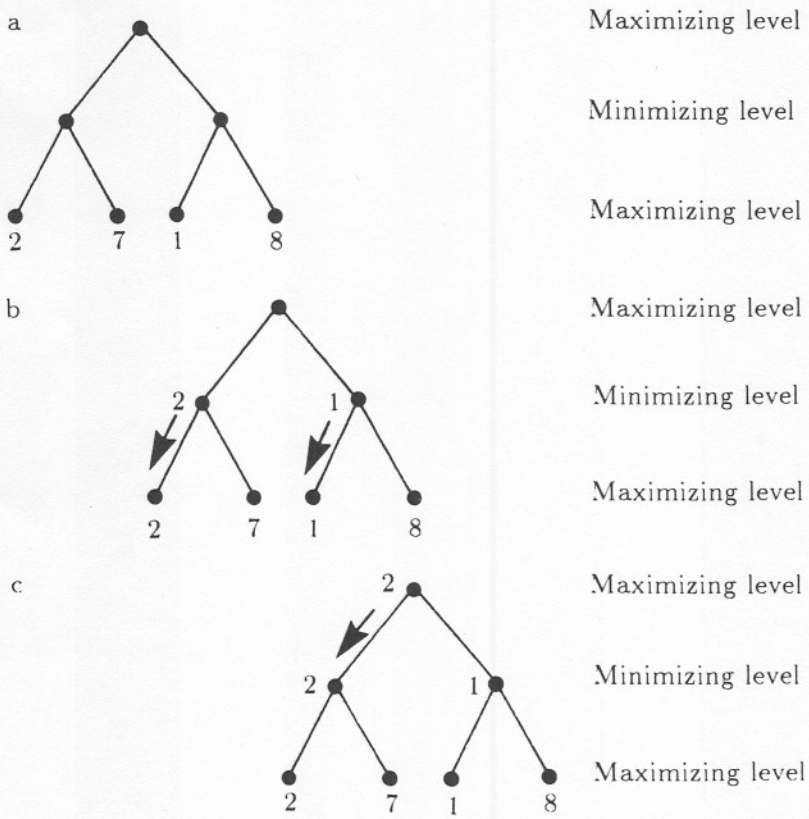

Figure 4-16. Minimaxing is a method for determining moves. Minimaxing employs a static evaluator to calculate advantage-specifying numbers for the game situations at the bottom of a partially developed game tree. One player works toward the higher numbers, seeking the advantage, while the opponent goes for the lower numbers.