5 Nets and Optimal Search

In this chapter, you learn how to deal with search situations in which the cost of traversing a path is of primary importance. In particular, you learn about the *British Museum procedure*, *branch and bound*, *discrete dynamic programming*, and A*. All but the British Museum procedure aspire to do their work efficiently.

By way of illustration, you see how it is possible to find the shortest route from one city to another using an ordinary highway map, and you see how a robot-planning system can find the most efficient way to move an object using *configuration space.*

Once you have finished this chapter, you will know what to do when you do not care as much about how long it takes to find a path as you do about finding the best path.

THE BEST PATH

In this section, you learn more about the map-traversal problem that emerged in Chapter 4; here, however, you pay attention to path length.

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 one from them. This plodding procedure, named in jest, is known as the **British Museum procedure**.

If you wish to find all possible paths, either a depth-first search or a breadth-first search will work, with one modification: Search continues until every solution is found. If the breadth and depth of the tree are small, as in the map-traversal example, then 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 the number of levels being small, it is moderately large. 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 analysis 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 billion paths. Fortunately, there are strategies that enable optimal paths to be found without all possible paths being found first.

Branch-and-Bound Search Expands the Least-Cost Partial Path

One way to find optimal paths with less work is to use **branch-and-bound search**. The basic idea is simple. Suppose an optimal solution is desired for the highway map shown Chapter 4. Also suppose that an oracle has told you that S–D–E–F–G is the optimal solution. Being a scientist, however, you do not trust oracles.

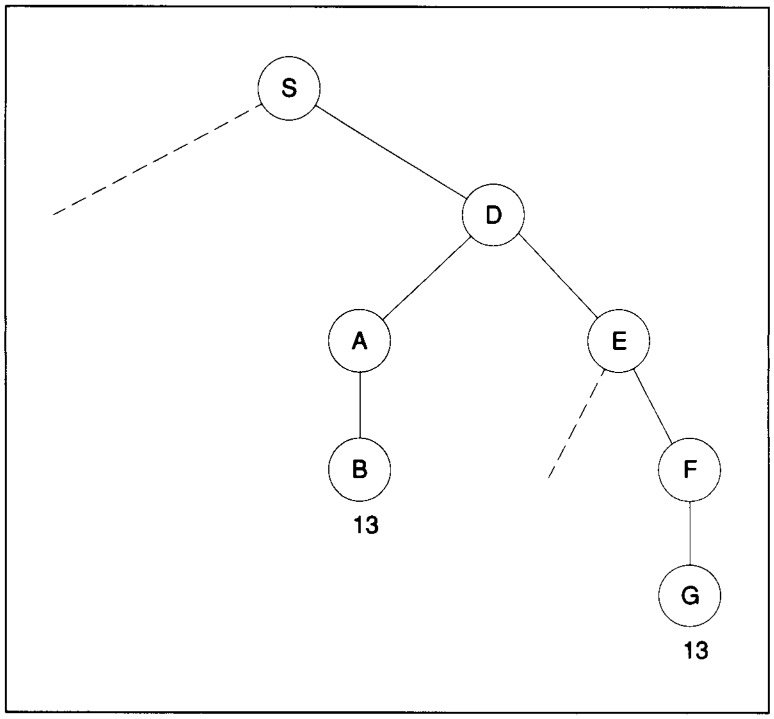
Nevertheless, knowing that the length of S–D–E–F–G is 13, you can eliminate some work that you might otherwise do. For example, as shown in figure 5.1, there is no need to consider paths that start with S–D–A–B, because their length has to be at least 13, given that the length of S–D–A–B is already 13.

More generally, the branch-and-bound scheme always keeps track of all partial paths contending for further consideration. The shortest one is extended one level, creating as many new partial paths as there are branches. Next, these new paths are considered, along with the remaining old ones: again, the shortest is extended. This process repeats until the goal is reached along some path. Because the shortest path was always the one chosen for extension, the path first reaching the goal is likely to be the optimal path.

To turn *likely* into *certain*, you have to extend all partial paths until they are as long as or longer than the complete path. The reason is that the last step in reaching the goal may be long enough to make the supposed solution longer than one or more partial paths. It might be that only a tiny step would extend one of the partial paths to the solution point. To be sure that this is not so, instead of terminating when a path is found, you terminate when the shortest partial path is longer than the shortest complete path.

Here, then, is the procedure; it differs from the basic search procedures you learned about in Chapter 4 only in the steps shown in italic type:

Figure 5.1 The length of the complete path from S to G, S–D–E–F–G is 13. Similarly, the length of the partial path S–D–A–B also is 13 and any additional movement along a branch will make it longer than 13. Accordingly, there is no need to pursue S–D–A–B any further because any complete path starting with S–D–A–B has to be longer than a complete path already known. Only the other paths emerging from S and from S–D–E have to be considered, as they may provide a shorter path.



To conduct a branch-and-bound search,

▷ Form a one-element queue consisting of a zero-length path that contains only the root node.

▷ Until the first path in the queue terminates at the goal node or the queue is empty,

  ▷ Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.

  ▷ Reject all new paths with loops.

  ▷ *Add the remaining new paths, if any, to the queue.*

  ▷ *Sort the entire queue by path length with least-cost paths in front.*

▷ If the goal node is found, announce success; otherwise, announce failure.

Now look again at the map-traversal problem, and note how branch-and-bound works when started with no partial paths. Figure 5.2 illustrates the exploration sequence. In the first step, the partial-path distance of S–A is found to be 3, and that of S–D is found to be 4; partial path S–A is therefore selected for expansion. Next, S–A–B and S–A–D are generated from S–A with partial path distances of 7 and 8.

Now S–D, with a partial path distance of 4, is expanded, leading to partial paths to S–D–A and S–D–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 path is the shortest one, but if you wish to be absolutely sure, you must 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, because 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.

Adding Underestimates Improves Efficiency

In some cases, you can improve branch-and-bound search 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, $e(\text{total path length})$:

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

where $d(\text{already traveled})$ is the known distance already traveled, and where $e(\text{distance remaining})$ 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 is revised 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 you 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 you to wander away from that optimal path permanently.

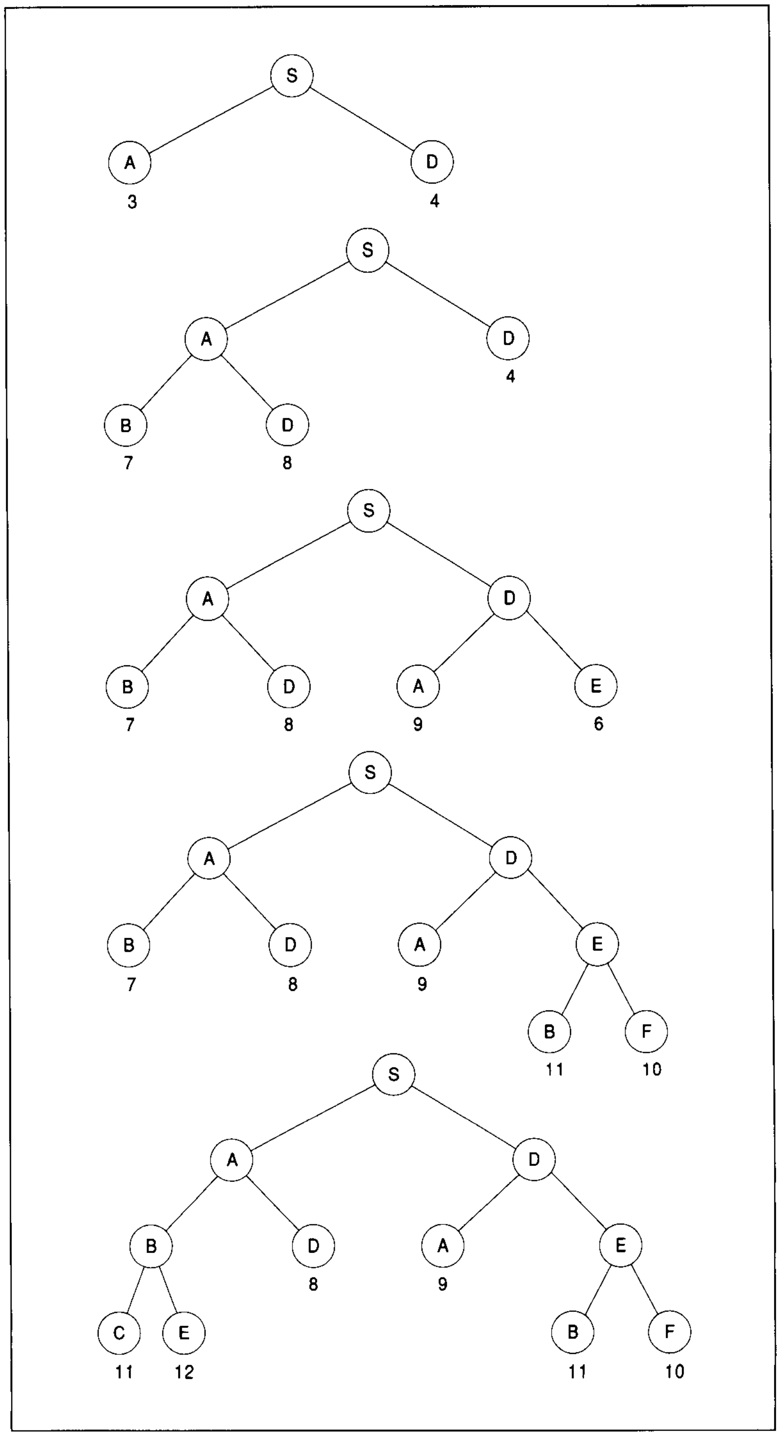
Note, however, that *underestimates* cannot cause the right path to be overlooked. An underestimate of the distance remaining yields an underestimate of total path length, $u(\text{total path length})$:

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

where $d(\text{already traveled})$ is the known distance already traveled, and where $u(\text{distance remaining})$ is an underestimate of the distance remaining.

Now, if you find a total path by extending the path with the smallest underestimate repeatedly, you need to do no further work once all partial-path distance estimates are longer than the best complete path distance so far encountered. You can stop because the real distance along a complete path cannot be less than an underestimate of that distance. If all estimates

Figure 5.2 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.



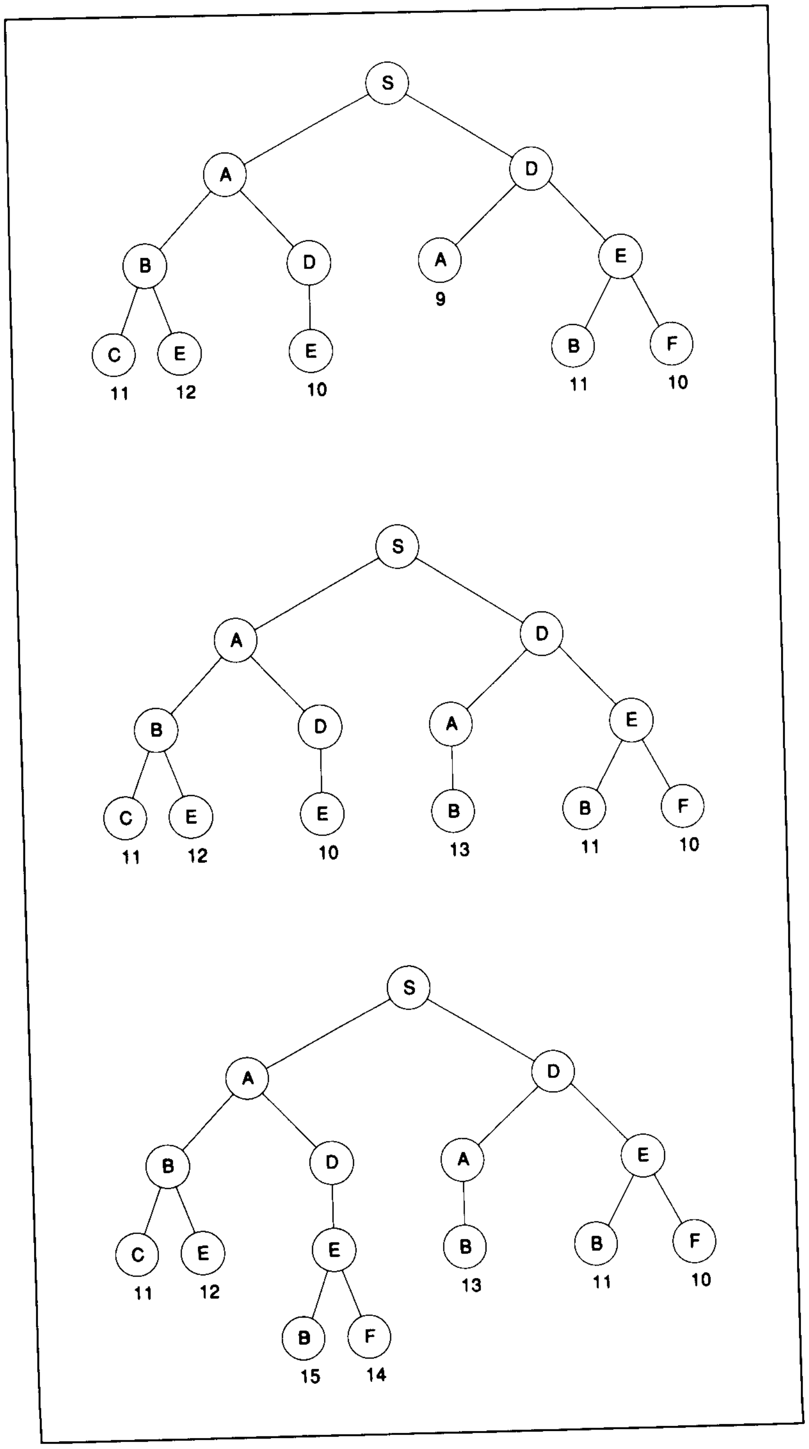
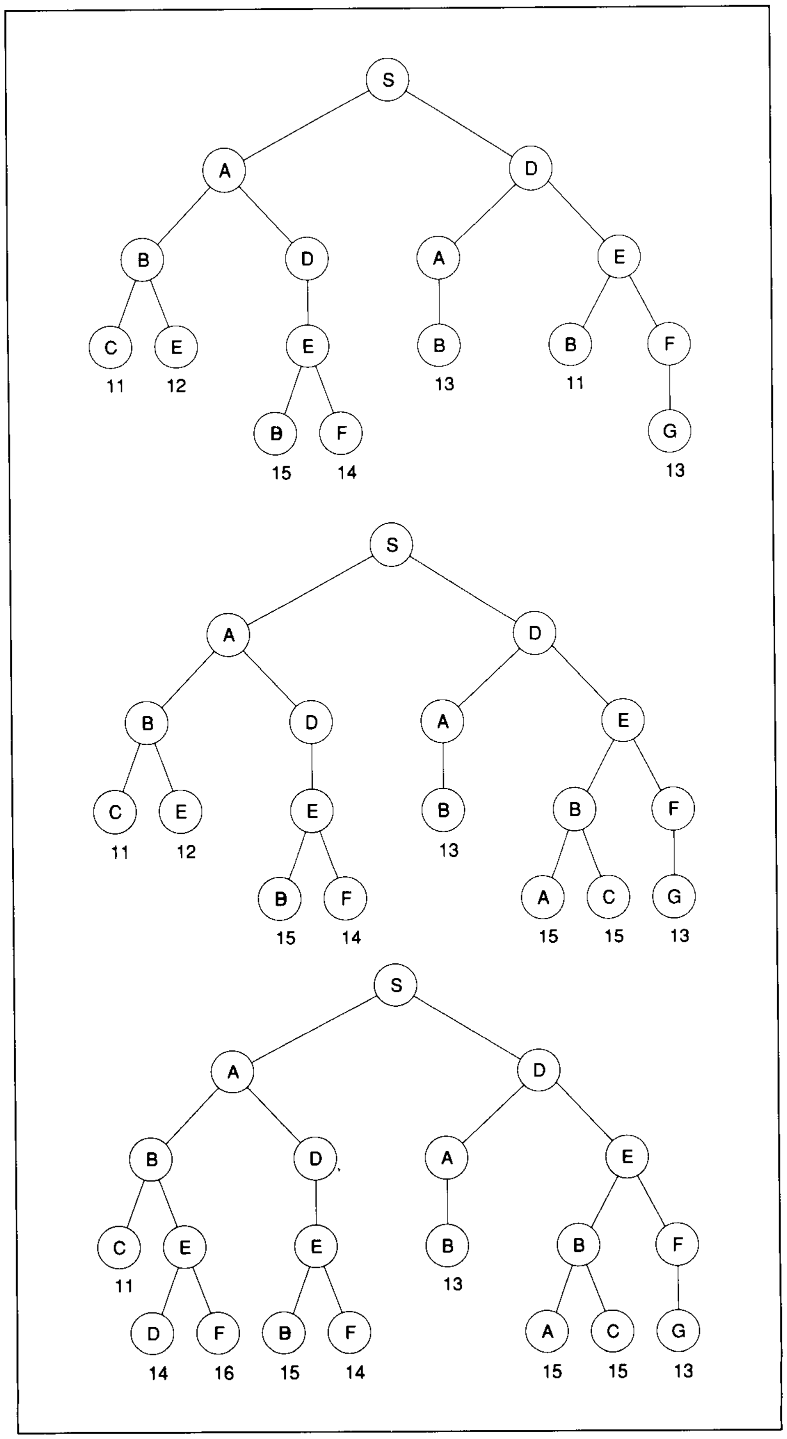


Figure 5.2 Continued.

Figure 5.2 Continued.



of remaining distance can be guaranteed to be underestimates, you cannot blunder.

When you are working out a path on a highway map, straight-line distance is guaranteed to be an underestimate. Figure 5.3 shows how straight-line distance helps to make the search efficient. As before, A and D are generated from S. This time, D is the node from which to search, because D's underestimated path length is 12.9, which is shorter than that for A, 13.4.

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

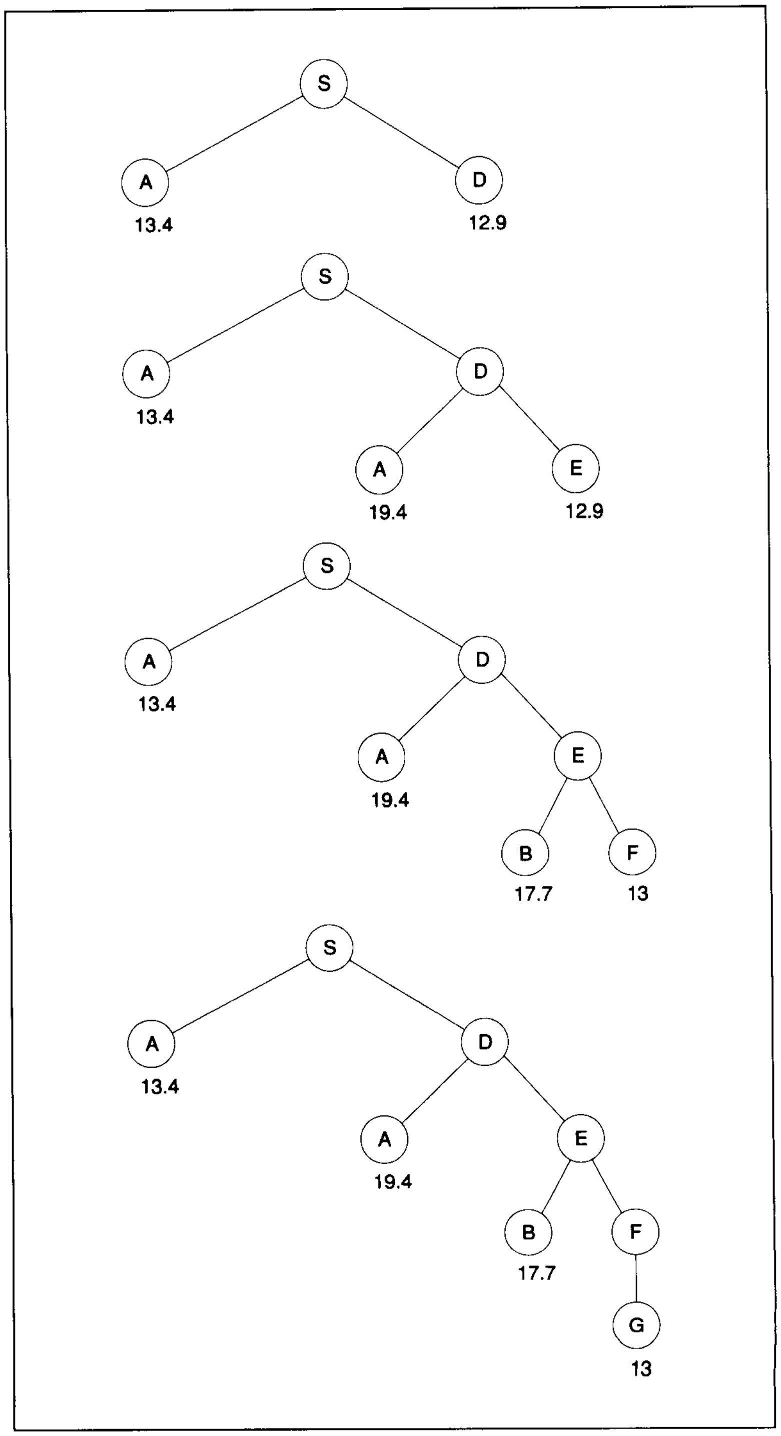
Expanding the partial path to F is the correct move, because it is the partial path with the minimum underestimated path length. This expansion leads to a complete path, S–D–E–F–G, with a total distance of 13.0. No partial path has a lower-bound distance so low, so no further search is required.

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

```
To conduct a branch-and-bound search with a lower-bound
estimate,
▷ Form a one-element queue consisting of a zero-length path
  that contains only the root node.
▷ Until the first path in the queue terminates at the goal
  node or the queue is empty,
    ▷ Remove the first path from the queue; create new paths
      by extending the first path to all the neighbors of the
      terminal node.
    ▷ Reject all new paths with loops.
    ▷ Add the remaining new paths, if any, to the queue.
    ▷ Sort the entire queue by the sum of the path length
      and a lower-bound estimate of the cost remaining, with
      least-cost paths in front.
▷ If the goal node is found, announce success; otherwise,
  announce failure.
```

Of course, the closer an underestimate is to the true distance, the more efficiently you search, because, 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,

Figure 5.3 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 would be expanded with branch-and-bound search operating without underestimates.



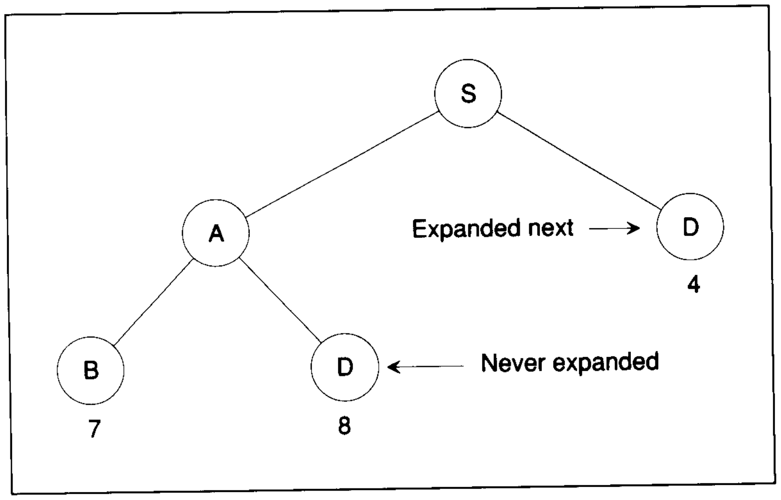


Figure 5.4 An illustration of the dynamic-programming principle. The numbers beside the nodes are accumulated distances. There is no point in expanding the instance of node D at the end of S–A–D, because getting to the goal via the instance of D at the end of S–D is obviously more efficient.

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.

REDUNDANT PATHS

In this section, you learn still more about the map-traversal problem that emerged in Chapter 4, but now you look at it with a view toward weeding out redundant partial paths that destroy search efficiency. In the end, you learn how to bring together several distinct ideas to form the A* procedure, and you see how A* can be put to work on a robot planning problem.

Redundant Partial Paths Should Be Discarded

Now let us consider another way to improve on basic branch-and-bound search. Look at figure 5.4. The root node, S, has been expanded, producing partial paths S–A and S–D. For the moment, let us use no underestimates for remaining path length.

Because S–A is shorter than S–D, S–A is extended first, leaving three paths: S–A–B, S–A–D, and S–D. Then, S–D will be extended, because 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. Because 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.

This example illustrates a general principle. 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 strangely named **dynamic-programming principle** holds that, when you look for the best path from S to G, you can ignore all paths from S to any intermediate node, I, other than the minimum-length path from S to I:

The **dynamic-programming principle**:

- ▷ The best way *through* a particular, intermediate place is the best way *to it* from the starting place, followed by the best way *from it* to the goal. There is no need to look at any other paths to or from the intermediate place.

The branch-and-bound procedure, with dynamic programming included, is as follows:

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

- ▷ Form a one-element queue consisting of a zero-length path that contains only the root node.
- ▷ Until the first path in the queue terminates at the goal node or the queue is empty,
  - ▷ Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - ▷ Reject all new paths with loops.
  - ▷ Add the remaining new paths, if any, to the queue.
  - ▷ *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.*
  - ▷ Sort the entire queue by path length with least-cost paths in front.
- ▷ If the goal node is found, announce success; otherwise, announce failure.

Figure 5.5 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.

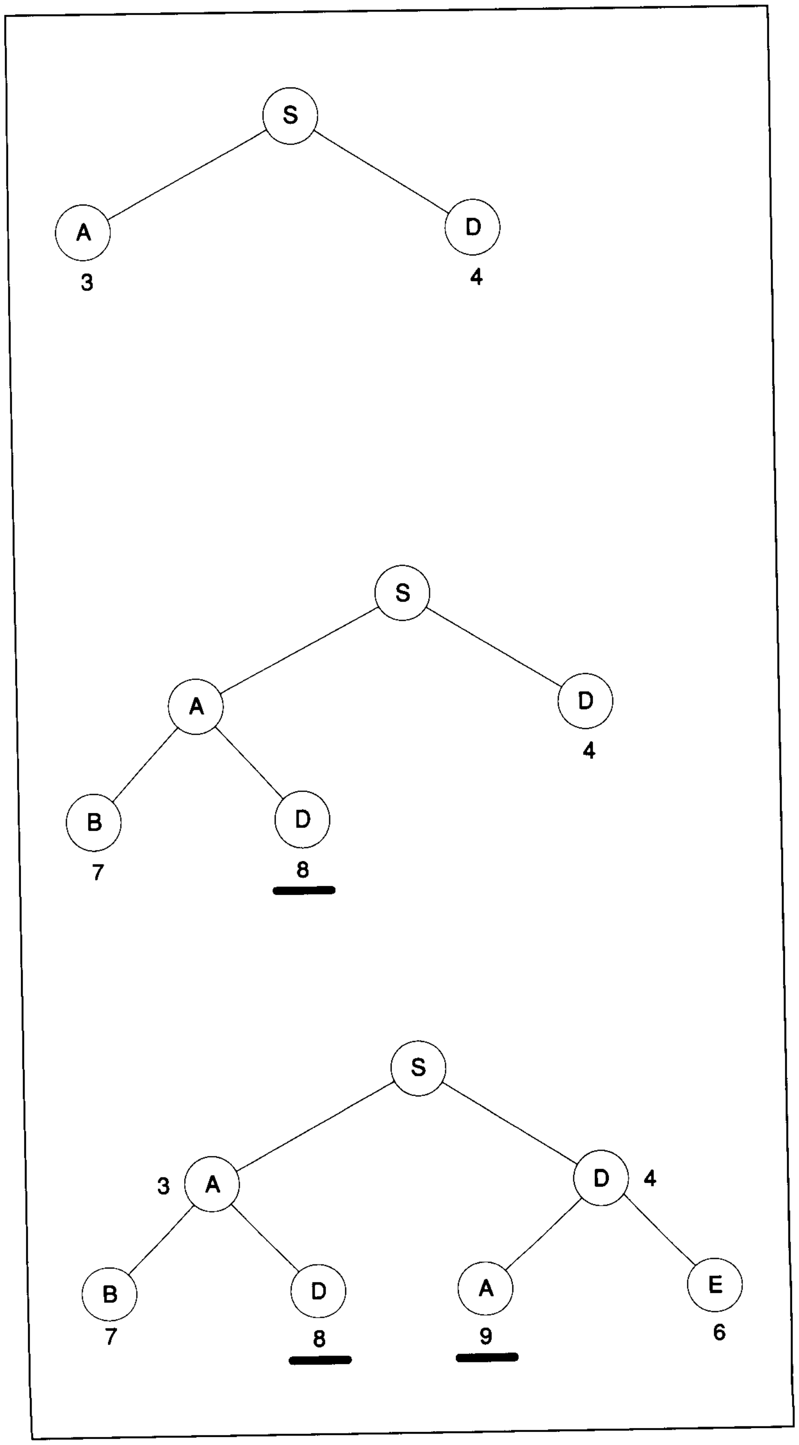
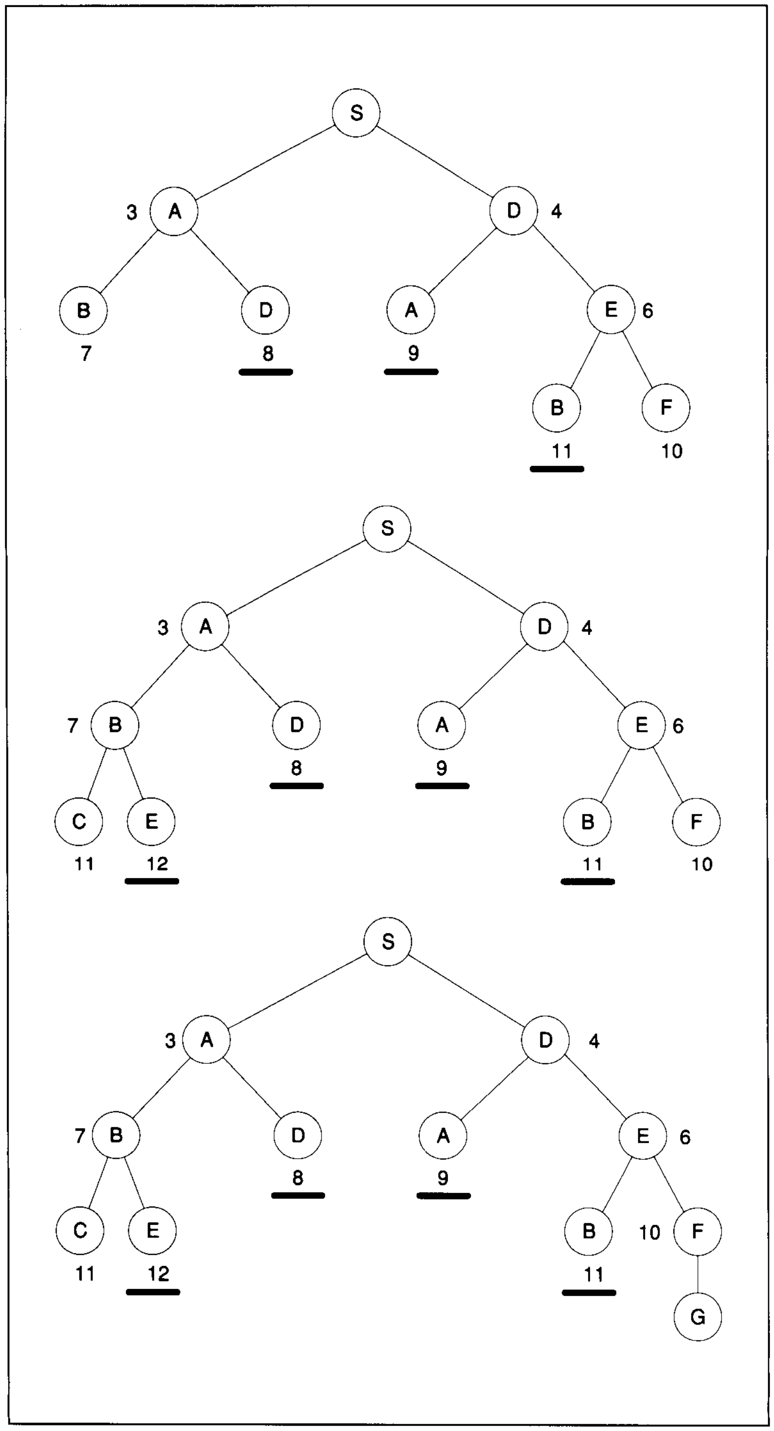


Figure 5.5 Branch-and-bound search, augmented by dynamic programming, determines that path S–D–E–F–G is optimal. The numbers beside the nodes are accumulated path distances. Many paths, those shown terminated with underbars, are found to be redundant. Thus, dynamic programming reduces the number of nodes expanded.

Figure 5.5 Continued.



Underestimates and Dynamic Programming Improve 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 conduct A* search,
▷ Form a one-element queue consisting of a zero-length path
  that contains only the root node.
▷ Until the first path in the queue terminates at the goal
  node or the queue is empty,
    ▷ Remove the first path from the queue; create new paths
      by extending the first path to all the neighbors of the
      terminal node.
    ▷ Reject all new paths with loops.
    ▷ 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.
    ▷ Sort the entire queue by the sum of the path length
      and a lower-bound estimate of the cost remaining, with
      least-cost paths in front.
▷ If the goal node is found, announce success; otherwise,
  announce failure.
```

Several Search Procedures Find the Optimal Path

You have seen that there are many ways to search for optimal paths, each of which has advantages:

- The British Museum procedure is good only 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 converge on the same place.
- The A* procedure is good when both branch-and-bound search with a guess and dynamic programming are good.

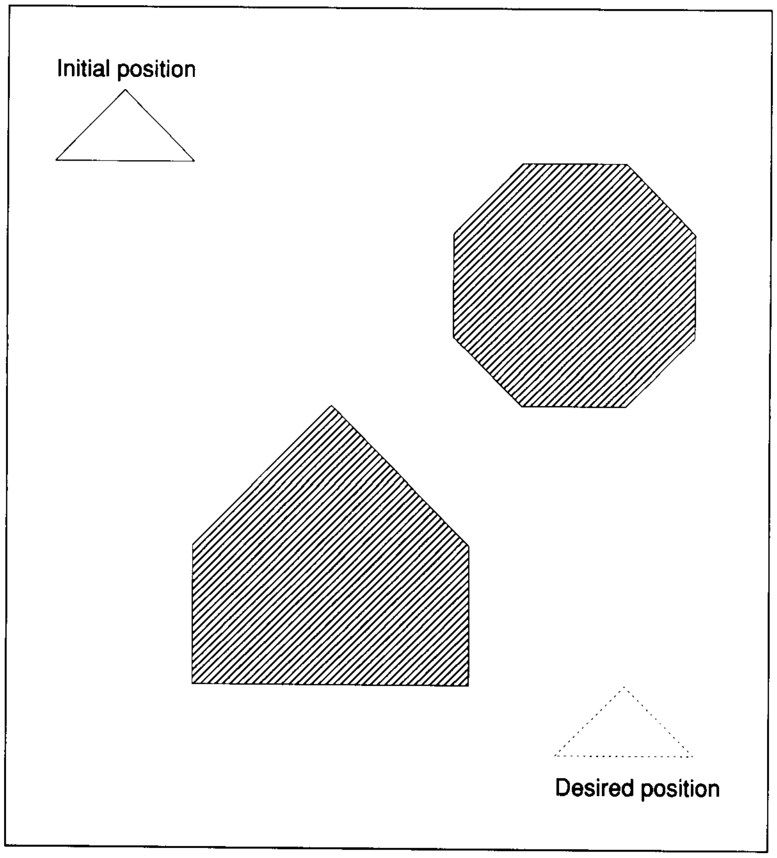


Figure 5.6 An obstacle-avoidance problem. The problem is to move the small triangular robot to a new position, shown dotted, without bumping into the pentagon or the octagon.

Robot Path Planning Illustrates Search

To see how the A* search procedure can be put to use, consider the collision-avoidance problem faced by robots. Before a robot begins to move in a cluttered environment, it must compute a collision-free path between where it is and where it wants to be. This requirement holds for locomotion of the whole robot through a cluttered environment and for robot hand motion through a component-filled workspace.

Figure 5.6 illustrates this motion-planning problem in a simple world inhabited by a triangular robot. The robot wants to move, without turning, from its initial position to the new position indicated by the dashed-line triangle. The question is, Can the robot make it through the gap between the pentagon and the octagon?

In two dimensions, a clever trick makes the problem easy. The general idea is to redescribe the problem in another, simpler representation, to solve the simpler problem, and to redescribe the solution in the original representation. Overall, taking this approach is like doing multiplication by moving back and forth between numbers and their logarithms or like

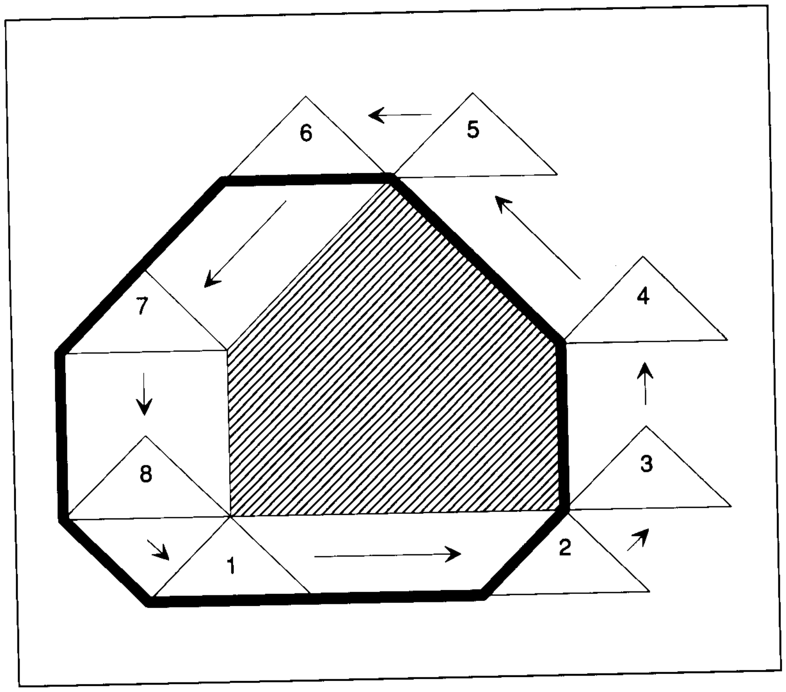


Figure 5.7 The configuration-space transformation. The heavy line shows the locus of the small triangle's lower-left corner as the small triangle is moved around the big one. Numbered positions are the starting points for each straight-line run. Keeping the lower-left corner away from the heavy line keeps the small triangle away from the pentagon.

analyzing linear systems by moving back and forth between signals and their Fourier transforms.

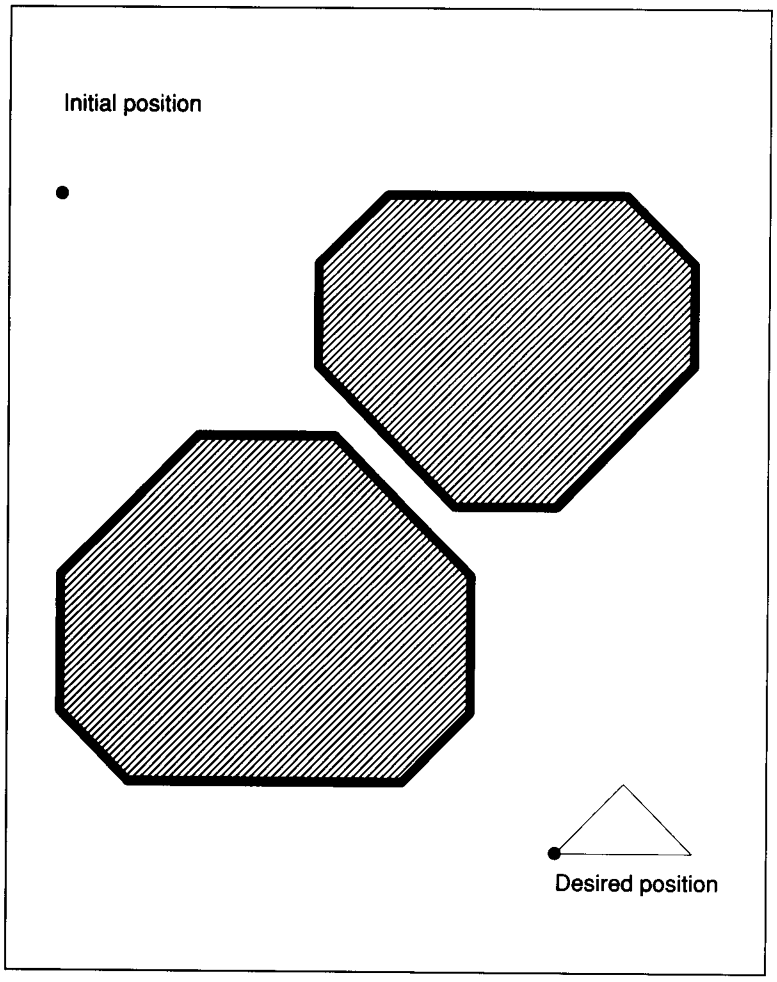
For obstacle avoidance, the original representation involves a moving object and stationary obstacles, and the new representation involves a moving point and larger, virtual obstacles called **configuration-space obstacles**.

Figure 5.7 shows how you can transform an ordinary obstacle into a configuration-space obstacle using the object to be moved and the obstacle to be avoided. Basically, you slide the object around the obstacle, maintaining contact between them at all times, keeping track of one arbitrary tracing point on the moving object as you go. As the tracing point moves around the obstacle, it builds an eight-sided fence. Plainly, there can be no collision between object and obstacle as long as the tracing point stays outside the fence that bounds the configuration-space obstacle associated with the original obstacle.

Figure 5.8 shows both of the configuration-space obstacles made from the original triangle and the pentagon and octagon shown in figure 5.6. The lower-left vertex of the triangular robot was used. Evidently, the robot can get through the gap, because the configuration-space obstacles are not large enough to close up the space.

It is not entirely clear that the shortest path is through the gap, however. To be sure that it is, you have to search.

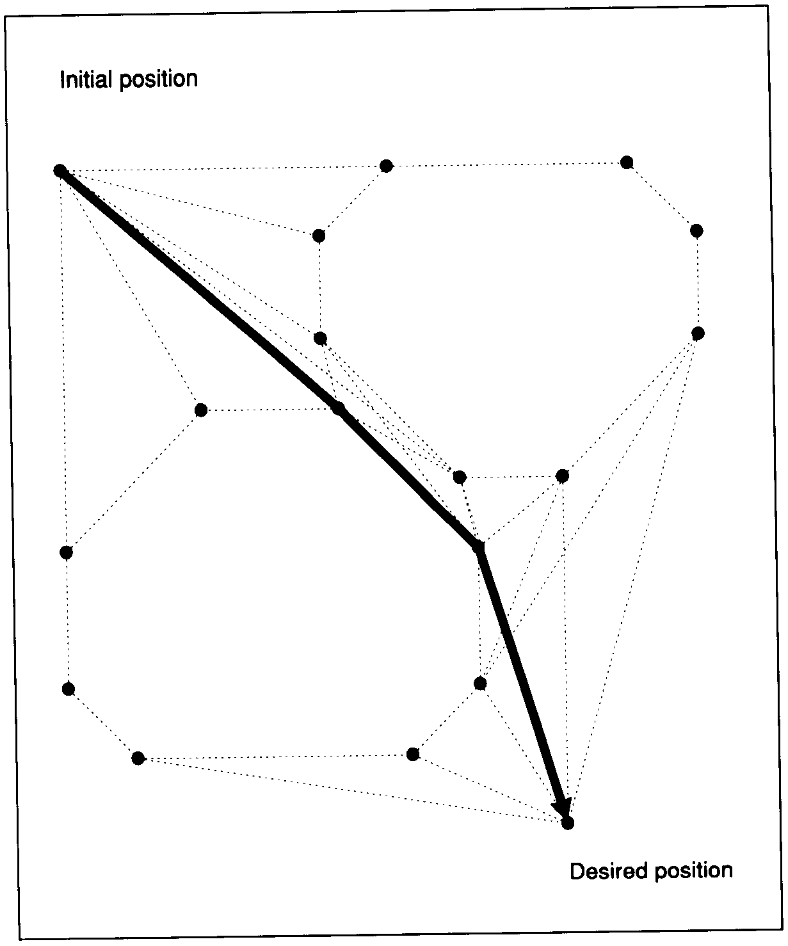
Figure 5.8 The configuration space for the problem shown in figure 5.6. No collision occurs if the point is kept out of the shaded area.



As yet, there is no net through which to search. The construction of a net is easy, however, for two-dimensional configuration-space problems. To see why it is easy, suppose that you are at any point in a configuration space. From where you are, you either can see the desired position or you cannot see it. If you can, you need to think no more, for the shortest path is the straight line between you and the desired position.

If you cannot see the desired position from where you are, then the only move that makes sense is a move to one of the vertexes that you can see. Accordingly, all motion is confined to motion from vertex to vertex, except at the beginning, when motion is from initial position to a vertex, and at the end, when motion is from a vertex to the desired position. Thus, the initial position, desired position, and vertexes are like the nodes in a net. Because the links between nodes are placed only when there is an

Figure 5.9 In a two-dimensional configuration space, the point robot moves along the straight lines of a visibility graph. An A* search through the visibility graph produces the shortest path from the initial position to the desired position. The heavy arrow shows the shortest path.



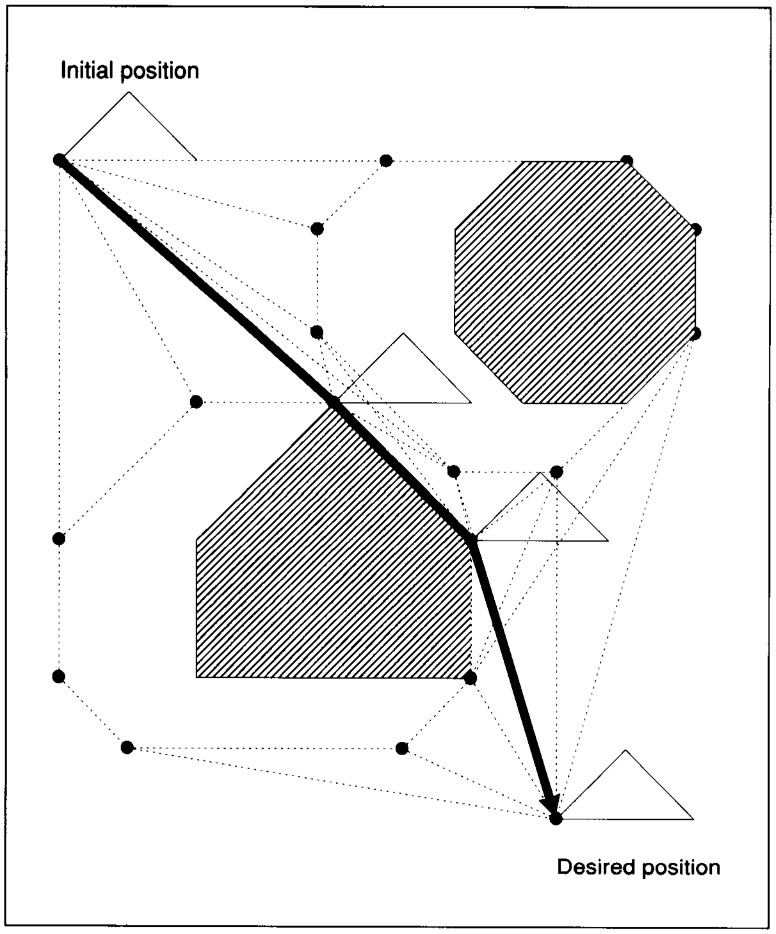
unobstructed line of sight between the nodes, the net is call a **visibility graph**.

Figure 5.9 illustrates the visibility graph for the robot-motion example, along with the result of performing an A* search to establish the shortest path for the configuration-space robot, a point, through the space of configuration-space obstacles, all oddly shaped. Figure 5.10 shows motion of the actual robot, along with the actual obstacles it circumnavigates, all superimposed on the solution in configuration space.

If you allow the moving object to rotate, you can make several configuration spaces corresponding to various degrees of rotation for the moving object. Then, the search involves motion not only through individual configuration spaces, but also from space to space.

Still more generally, when you need to move an arm, holding an object, in three dimensions, rather than just two, the construction of a suitable con-

Figure 5.10 The robot's movement is dictated by the shortest path found in the visibility graph. The lower-left corner of the triangular robot—the one used to produce configuration-space obstacles—is moved along the shortest path. Note that the triangular robot never collides with either the pentagon or the octagon.



figuration space is extremely complicated mathematically. Complication-loving mathematicians have produced a flood of literature on the subject.

SUMMARY

- The British Museum procedure is one of many search procedures oriented toward finding the shortest path between two points. The British Museum procedure relies on working out all possible paths.
- Branch-and-bound search usually saves a lot of time relative to the British Museum procedure. It works by extending the least-cost partial path until that path reaches the goal.
- Adding underestimates to branch-and-bound search improves efficiency. Deleting redundant partial paths, a form of dynamic programming, also improves efficiency. Adding underestimates and deleting redundant partial paths converts branch-and-bound search into A* search.

- The configuration-space transformation turns object-obstacle problems into point-obstacle problems. So transformed, robot path-planning problems succumb to A* search.

BACKGROUND

Optimal search methods are discussed in great detail in many books on algorithms. In particular, see the textbook, *Introduction to Algorithms*, by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest [1990] for an excellent treatment.

The discussion of configuration space is based on the work of Tomás Lozano-Pérez [1980].