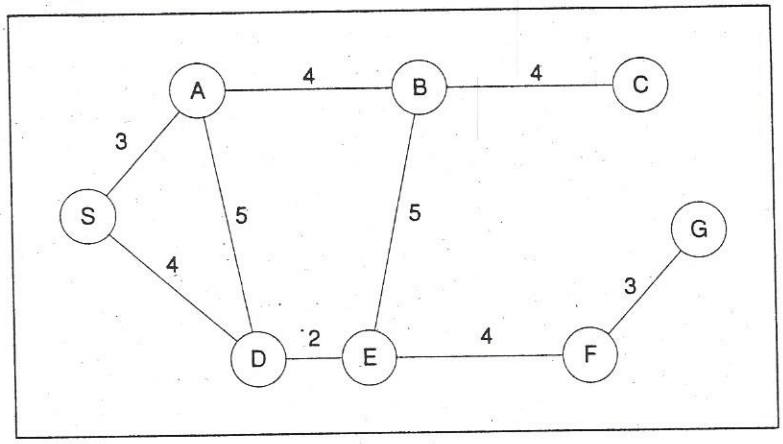

Figure 4.1 A basic search problem. A path is to be found from the start node, S, to the goal node, G. Search procedures explore nets such as these, learning about connections and distances as they go.

BLIND METHODS

Suppose you want to find some path from one city to another using a highway map such as the one shown in figure 4.1. Your path is to begin at city S, your starting point, and it is to end at city G, your goal. To find an appropriate path through the highway map, you need to consider two different costs:

- First, there is the computation cost expended when *finding* a path.
- And, second, there is the travel cost expended when *traversing* the path.

If you need to go from S to G often, then finding a really good path is worth a lot of search time. On the other hand, if you need to make the trip only once, and if it is hard to find any path, then you may be content as soon as you find some path, even though you could find a better path with more work.

In this chapter, you learn about the problem of finding one path. In the rest of this section, you learn about finding one path given no information about how to order the choices at the nodes so that the most promising are explored earliest.

Net Search Is Really Tree Search

The most obvious way to find a solution is to look at all possible paths. Of course, you should discard paths that revisit any particular city so that you cannot get stuck in a loop—such as S–A–D–S–A–D–S–A–D–....

With looping paths eliminated, you can arrange all possible paths from the start node in a **search tree**, a special kind of semantic tree in which each node denotes a path:

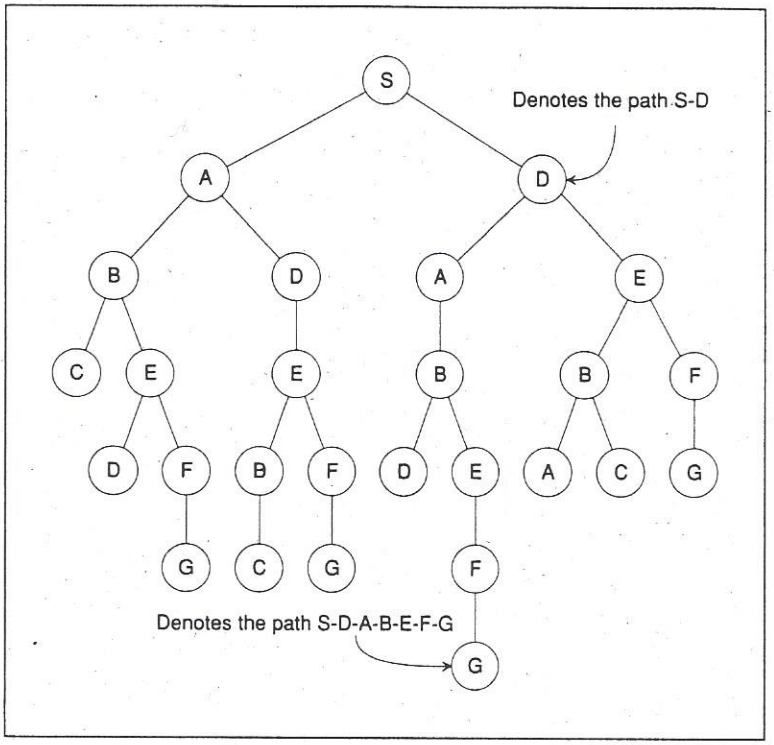

Figure 4.2 A search tree made from a net. Each node denotes a path. Each child node denotes a path that is a one-step extension of the path denoted by its parent. You convert nets into search trees by tracing out all possible paths until you cannot extend any of them without creating a loop.

A **search tree** is a representation
That is a semantic tree
In which
▷ Nodes denote paths.
▷ Branches connect paths to one-step path extensions.
With writers that
▷ Connect a path to a path description
With readers that
▷ Produce a paths's description

Figure 4.2 shows a search tree that consists of nodes denoting the possible paths that lead outward from the start node of the net shown in figure 4.1.

Note that, although each node in a search tree denotes a path, there is no room in diagrams to write out each path at each node. Accordingly, each node is labeled with only the terminal node of the path it denotes. Each **child** denotes a path that is a one-city extension of the path denoted by its **parent**.

In Chapter 3, the specification for the semantic-tree representation indicated that the node at the top of a semantic tree, the node with no parent, is called the **root node**. The nodes at the bottom, the ones with no children, are called **leaf nodes**. One node is the **ancestor** of another, a **descendant**, if there is a chain of two or more branches from the ancestor to the descendant.

If a node has b children, it is said to have a **branching factor** of b. If the number of children is always b for every nonleaf node, then the tree is said to have a branching factor of b.

In the example, the root node denotes the path that begins and ends at the start node S. The child of the root node labeled A denotes the path S-A. Each path, such as S-A, that does not reach the goal is called a **partial path**. Each path that does reach the goal is called a **complete path**, and the corresponding node is called a **goal node**.

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

Note that search procedures start out with no knowledge of the ultimate size or shape of the complete search tree. All they know is where to start and what the goal is. Each must expand open nodes, starting with the root node, until it discovers a node that corresponds to an acceptable path.

Search Trees Explode Exponentially

The total number of paths in a tree with branching factor b and depth d is b^d. Thus, the number of paths is said to **explode exponentially** as the depth of the search tree increases.

Accordingly, you always try to deploy a search method that is likely to develop the smallest number of paths. In the rest of this section, you learn about several search methods from which you can choose.

Depth-First Search Dives into the Search Tree

Given that one path is as good as any other, one simple way to find a path is to pick one of the children at every node visited, and to work forward from that child. Other alternatives at the same level are ignored completely, as long as there is hope of reaching the goal using the original choice. This strategy is the essence of **depth-first search**.

Using a convention that the alternatives are tried in left-to-right order, the first thing to do is to dash headlong to the bottom of the tree along the leftmost branches, as shown in figure 4.3.

But because a headlong dash leads to leaf node C, without encountering G, the next step is to back up to the nearest ancestor node that has 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 4.3 shows the nodes encountered.

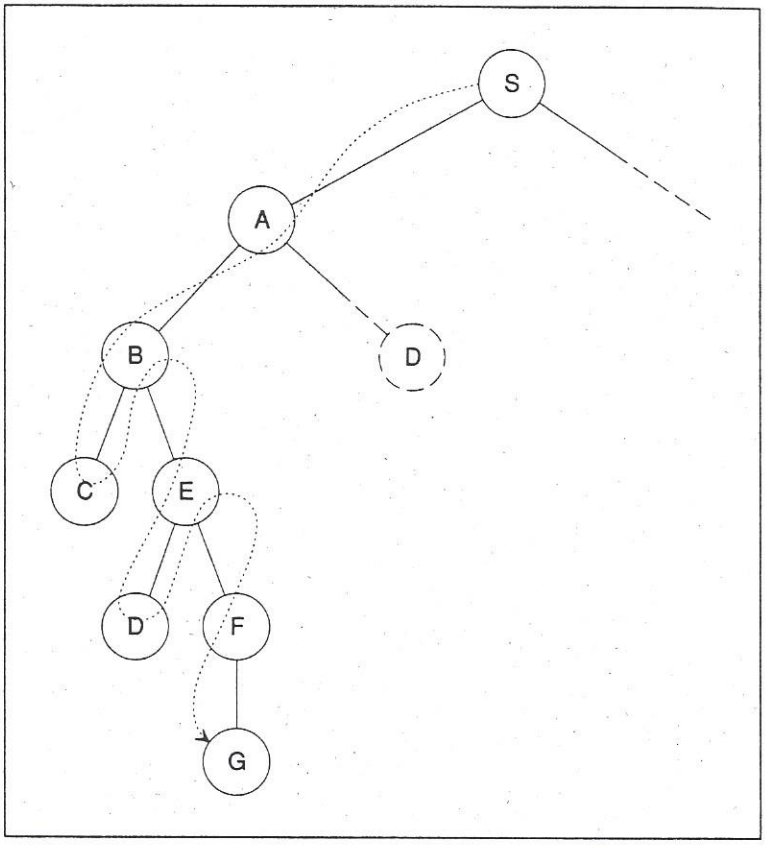

Figure 4.3 An example of depth-first search. One alternative is selected and pursued at each node until the goal is reached or a node is reached where further downward motion is impossible. When further downward motion is impossible, search is restarted at the nearest ancestor node with unexplored children.

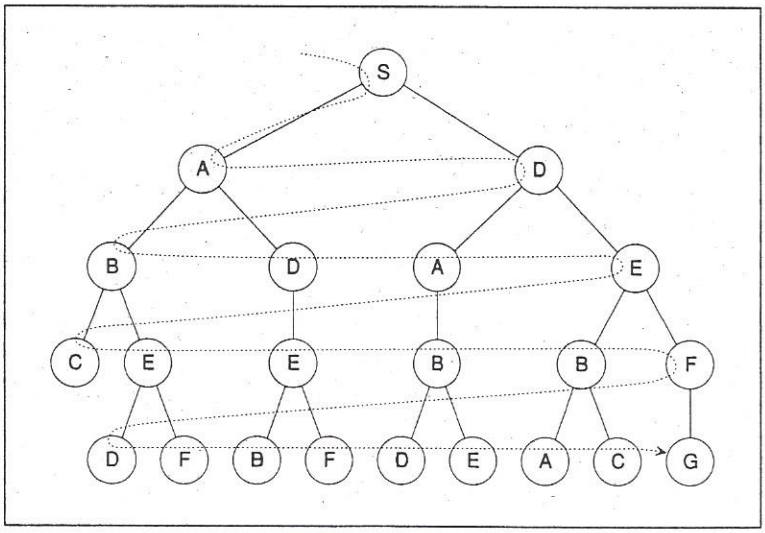

Figure 4.4 An example of breadth-first search. Downward motion proceeds level by level, until the goal is reached.

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

Having learned about depth-first search by way of an example, you can see that the procedure, expressed in procedural English, is as follows:

To conduct a depth-first 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 new paths, if any, to the *front* of the queue.
- ▷ If the goal node is found, announce success; otherwise, announce failure.

Breadth-First Search Pushes Uniformly into the Search Tree

As shown in figure 4.4, **breadth-first search** checks all paths of a given length before moving on to any longer paths. In the example, breadth-first search discovers a complete path to node G on the fourth level down from the root level.

A procedure for breadth-first search resembles the one for depth-first search, differing only in where new elements are added to the queue:

To conduct a breadth-first 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 new paths, if any, to the *back* of the queue.
- ▷ If the goal node is found, announce success; otherwise, announce failure.

The Right Search Depends on the Tree

Depth-first search is a good idea when you are confident that all partial paths either reach dead ends or become complete paths after a reasonable number of steps. In contrast, depth-first search is a bad idea if there are long paths, even infinitely long paths, that neither reach dead ends nor become complete paths. In those situations, you need alternative search methods.

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

Note that breath-first search is a bad idea if the branching factor is large or infinite, because of exponential explosion. Breadth-first search is a good idea when you are confident that the branching factor is small. You may also choose breadth-first search, instead of depth-first search, if you are worried that there may be long paths, even infinitely long paths, that neither reach dead ends nor become complete paths.

Nondeterministic Search Moves Randomly into the Search Tree

You may be so uninformed about a search problem that you cannot rule out either a large branching factor or long useless paths. In such situations, you may want to seek a middle ground between depth-first search and breadth-first search. One way to seek such a middle ground is to choose **nondeterministic search**. When doing nondeterministic search, you expand an open node that is chosen at random. That way, you ensure that you cannot get stuck chasing either too many branches or too many levels:

To conduct a nondeterministic 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 new paths at random places in the queue.
- ▷ If the goal node is found, announce success; otherwise, announce failure.
