4 Nets and Basic Search

In this chapter, you learn how to find paths through nets, thus solving *search* problems.[†] In particular, you learn about *depth-first search* and *breath-first search*, both of which are methods that involve blind groping. You also learn about *hill-climbing*, *beam search*, and *best-first search*, all of which are methods that are guided by heuristic quality estimates.

Search problems pop up everywhere. In this chapter, you see examples involving map traversal and recipe discovery. Other chapters offer many other examples of the basic search methods in action, including an example, in Chapter 29, of how depth-first search can be used to provide natural-language access to a database.

Once you have finished this chapter, you will know that you must think about several questions if you are to use search methods wisely; these are examples:

- Is search the best way to solve the problem?
- Which search methods solve the problem?
- Which search method is most efficient for this problem?

[†]Many problem-solving paradigms, like search, require only a weak understanding of the domains to which they are applied. Consequently, some people call such problem-solving paradigms *weak methods*. The term *weak method* is not used in this book because it can be taken to mean *low powered*.

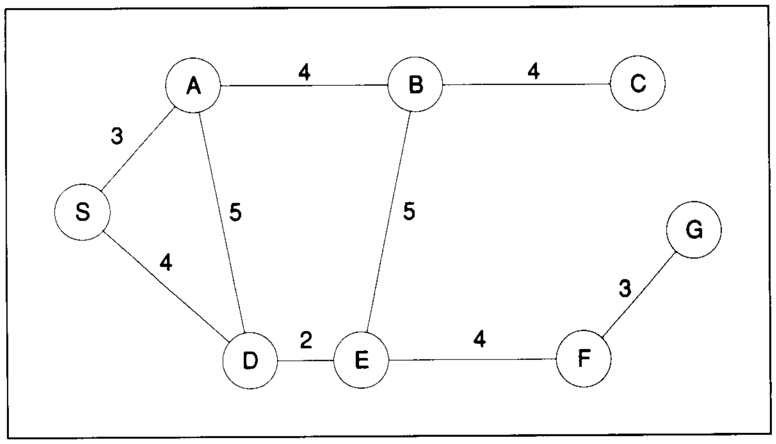


Figure 4.1 A basic search problem. A path is to be found from the start node, S, to the goal node, G. Search procedures explore nets such as these, learning about connections and distances as they go.

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.

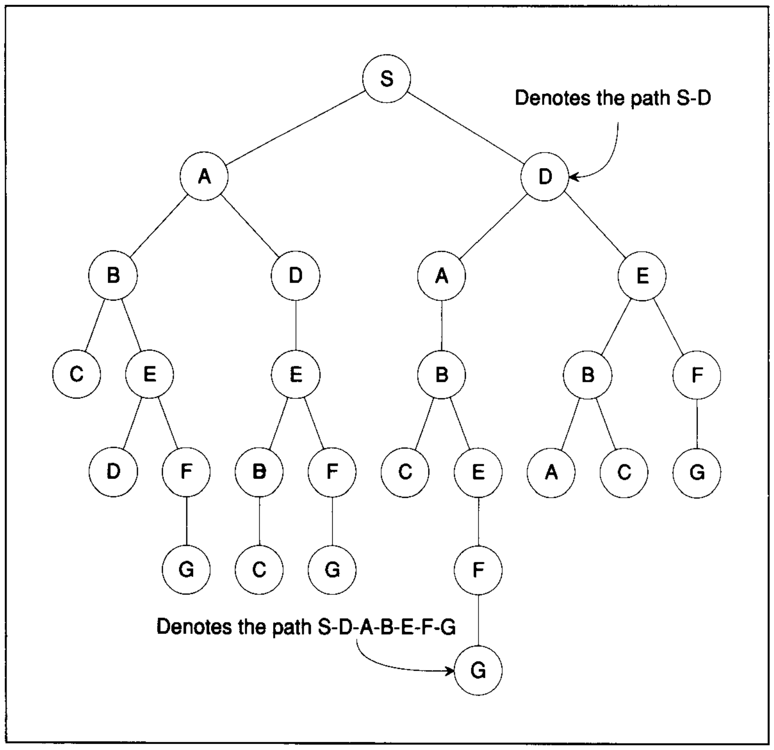
In this chapter, you learn about the problem of finding one path. In the rest of this section, you learn about finding one path given no information about how to order the choices at the nodes so that the most promising are explored earliest.

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 one 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.

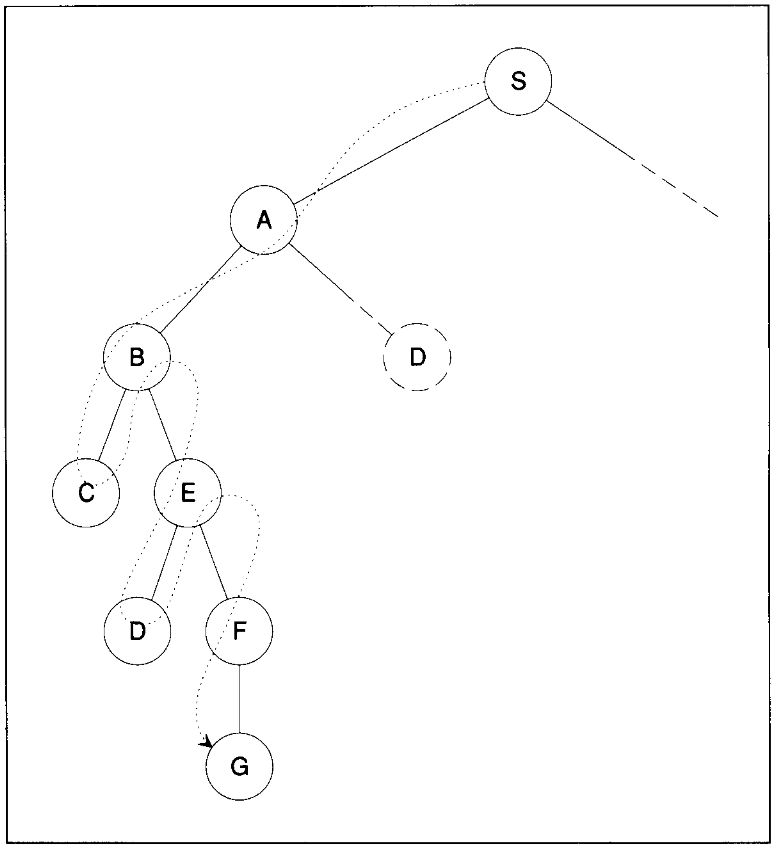
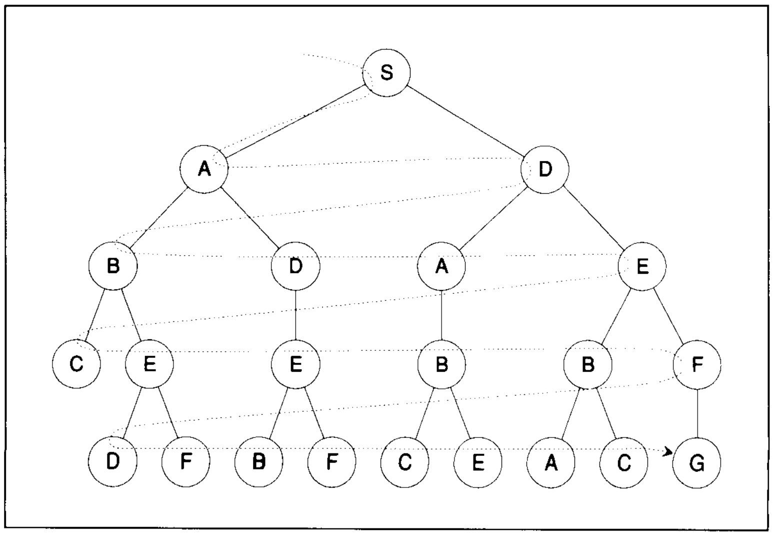


Figure 4.4 An example of breadth-first search. Downward motion proceeds level by level, until the goal is reached.



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, an-
  nounce 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, an-
  nounce 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, an-
  nounce failure.
```

HEURISTICALLY INFORMED METHODS

Search efficiency may improve spectacularly if there is a way to order the choices so that the most promising are explored earliest. In many situations, you can make measurements to determine a reasonable ordering. In the rest of this section, you learn about search methods that take advantage of such measurements; they are called **heuristically informed methods**.

Quality Measurements Turn Depth-First Search into Hill Climbing

To move through a tree of paths using **hill climbing**, you proceed as you would in depth-first search, except that you order your choices according to some heuristic measure of the remaining distance to the goal. 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 the straight-line distances from each city to the goal; Figure 4.6 shows what happens when hill climbing is used on the map-traversal problem using as-the-crow-flies distance to order choices. Because node D is closer to the goal than is node A, the children of D are examined first. Then, node E appears closer to the goal than is node A. Accordingly, node E's children are examined, leading to a move to node F, which is closer to the goal than node B. Below node F, there is only one child: the goal node G.

From a procedural point of view, hill climbing differs from depth-first search in only one detail; there is an added step, shown in italic type:

```
To conduct a hill-climbing 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.
    ▷ *Sort the new paths, if any, by the estimated distances between their terminal nodes and the goal.*
    ▷ Add the new paths, if any, to the *front* of the queue.
▷ If the goal node is found, announce success; otherwise, announce failure.
```

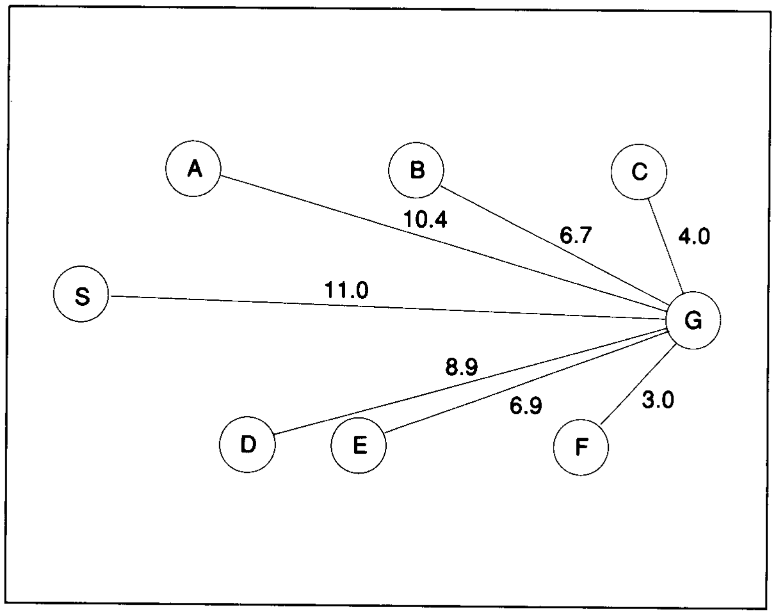


Figure 4.5 Figure 4.1 revisited. Here you see the distances between each city and the goal. If you wish to reach the goal, it is usually better to be in a city that is close, but not necessarily; city C is closer than all but city F, but city C is not a good place to be.

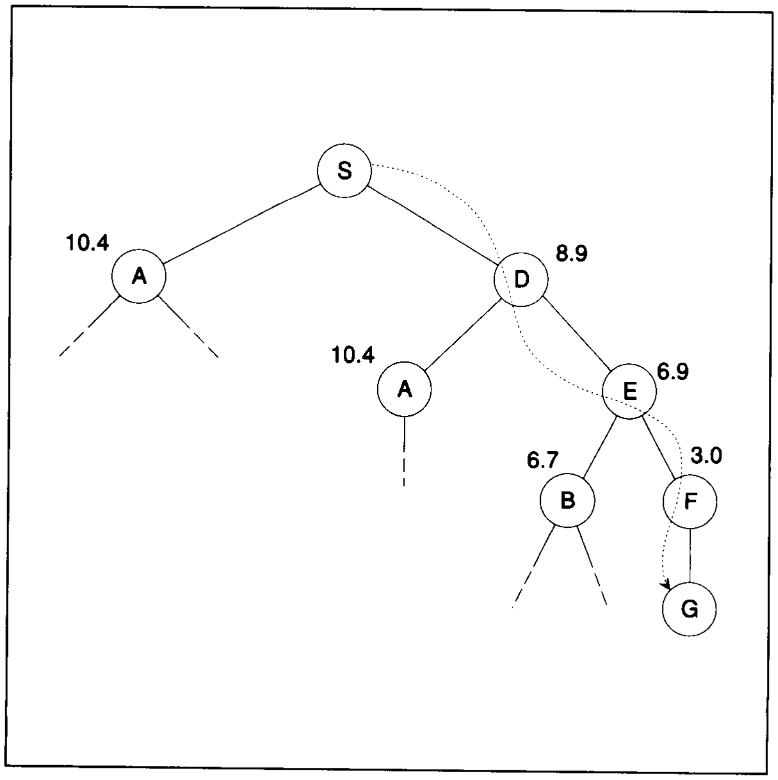


Figure 4.6 An example of hill climbing. Hill climbing is depth-first search with a heuristic measurement that orders choices as nodes are expanded. The numbers beside the nodes are straight-line distances from the path-terminating city to the goal city.

Generally, exploiting knowledge, as in hill climbing, reduces search time, a point worth elevating to the status of a powerful idea:

Whenever faced with a search problem, note that,

▷ More knowledge generally leads to reduced search time.

Sometimes, knowledge reduces search time by guiding choices, as in the example. Sometimes, knowledge reduces search time by enabling you to restate a problem in terms of a smaller, more easily searched net; this observation leads directly to another powerful idea:

When you think you need a better search method,

▷ Find another space to search instead.

Although search is involved in many tasks, devising a fancy, finely tuned search procedure is rarely the best way to spend your time. You usually do better if you improve your understanding of the problem, thereby reducing the need for fanciness and fine tuning.

Foothills, Plateaus, and Ridges Make Hills Hard to Climb

Although it is simple, hill climbing suffers from various problems. These problems are most conspicuous when hill climbing is used to optimize parameters, as in the following examples:

- On entering a room, you find that the temperature is uncomfortable. You decide to adjust the thermostat.
- The picture on your television set has deteriorated over time. You decide to adjust the color, tint, and brightness controls for a better picture.
- You are climbing a mountain when a dense fog rolls in. You have no map or trail to follow, but you do have a compass, an altimeter, and a determination to get to the top.

Each of these problems conforms to an abstraction in which there are *adjustable parameters* and a *measured quantity* that tells you about the quality or performance associated with any particular setting of the adjustable parameters.

In the temperature example, the adjustable parameter is the thermostat setting, and your comfort determines how well the parameter has been set. In the television example, the various controls are the parameters, and your subjective sense of the picture's quality determines how well the parameters have been set.

In the mountaineering example, your location is the adjustable parameter, and you can use your altimeter to determine whether you are progressing up the mountain. To get to the top using **parameter-oriented**

hill climbing, you tentatively try a step northward, then you retreat and try a step eastward. Next you try southward and westward as well. Then you commit to the step that increases your altitude most. You repeat until all tentative steps decrease your altitude.

More generally, to perform parameter-oriented hill climbing, you make a one-step adjustment, up and down, to each parameter value, move to the best of the resulting alternatives according to the appropriate measure of quality or performance, and repeat until you find a combination of parameter values that produces better quality or performance than all of the neighboring alternatives.

But note that you may encounter severe problems with parameter-oriented hill climbing:

- The **foothill problem** occurs whenever there are secondary peaks, as in the example at the top of figure 4.7. The secondary peaks draw the hill-climbing procedure like magnets. An optimal point is found, but it is a **local maximum**, rather than a **global maximum**, and the user is left with a false sense of accomplishment or security.
- The **plateau problem** comes up when there is a mostly flat area separating the peaks. In extreme cases, the peaks may look like telephone poles sticking up in a football field, as in the middle example of figure 4.7. 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, is more frustrating. Suppose you are standing on what seems like a knife edge contour running generally from northeast to southwest, as in the bottom example of figure 4.7. A contour map shows that each standard step takes you down, even though you are not at any sort of local or global maximum. Increasing the number of directions used for the probing steps may help.

Among these problems, the foothill problem is particularly vexing, especially as the number of parameter dimensions increases. When you reach a point from which all steps lead down, you could retreat to a previous choice point and do something else, as hill climbing prescribes, but there may be millions of paths back to the same local maximum. If there are, you are really stuck if you stick to ordinary hill climbing.

Accordingly, you may want to do a bit of nondeterministic search when you detect a local maximum. The reason for using this strategy is that a random number of steps, of random size, in random directions, may shield you from the magnetlike attraction of the local maximum long enough for you to escape.

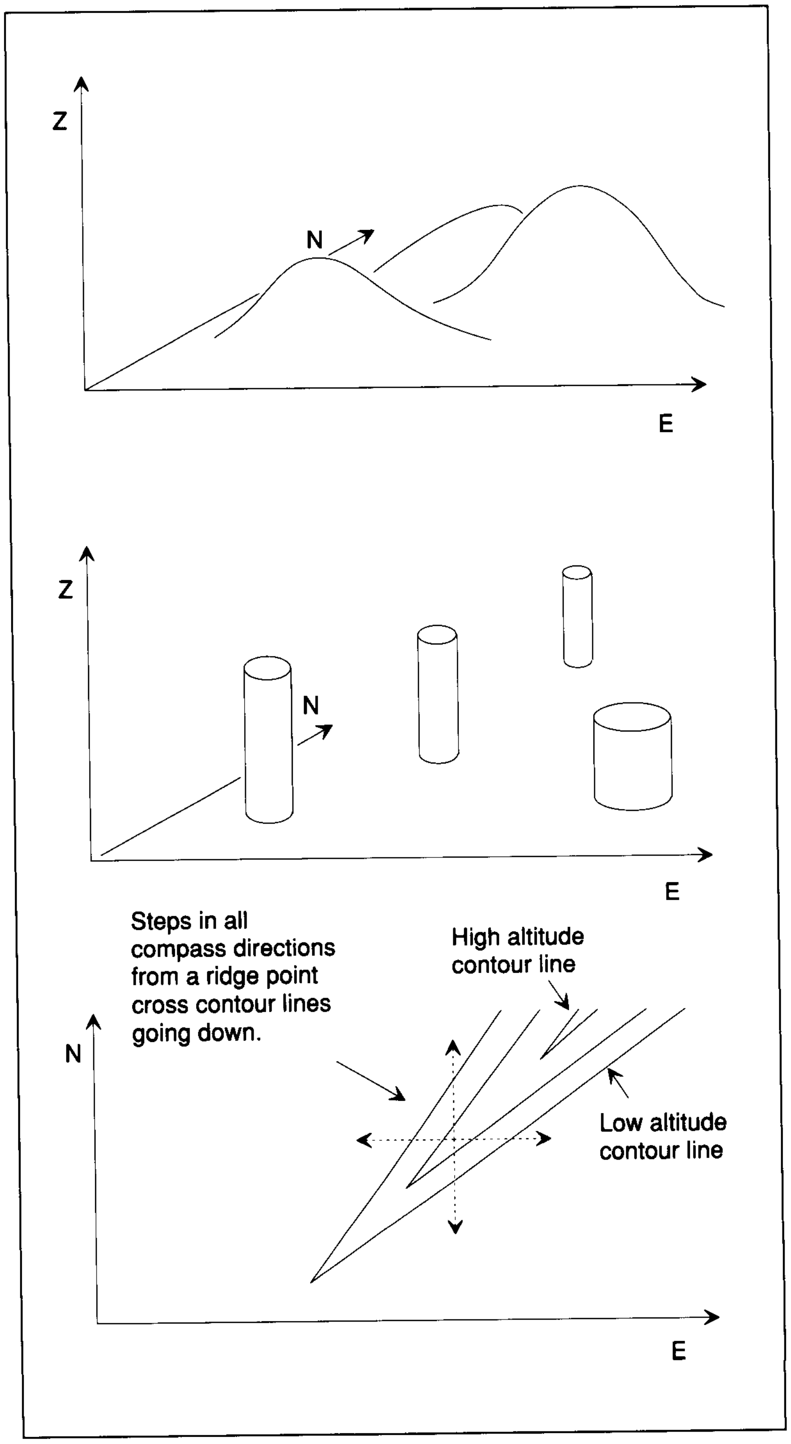

Z
N
E
Z
N
E
Steps in all compass directions from a ridge point cross contour lines going down.
High altitude contour line
Low altitude contour line
N
E

Figure 4.7 Hill climbing is a bad idea in difficult terrain. In the top example, foothills stop progress. In the middle example, plains cause aimless wandering. In the bottom example, with the terrain described by a contour map, all ridge points look like peaks because both east–west and north–south probe directions lead to lower-quality measurements.

Beam Search Expands Several Partial Paths and Purges the Rest

Beam search is like breadth-first search in that it progresses level by level. Unlike breadth-first search, however, beam search moves downward only through 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. Whenever beam search is used, there are only w nodes under consideration at any depth, rather than the exponentially explosive number of nodes with which you must cope whenever you use breadth-first search. Figure 4.8 illustrates how beam search would handle the map-traversal problem.

Best-First Search Expands the Best Partial Path

Recall that, when forward motion is blocked, hill climbing demands forward motion from the most recently created open node. In **best-first search**, forward motion is from the best open node so far, no matter where that node is in the partially developed tree.

In the example map-traversal problem, hill climbing and best-first search coincidentally explore the search tree in the same way.

The paths found by best-first search are 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 *likely to be* does not mean *certainly are*, however.

Search May Lead to Discovery

Finding physical paths and tuning parameters are only two applications for search methods. More generally, the nodes in a search tree may denote abstract entities, rather than 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 *ingredient transformations* for varying your existing recipes:

- Replace an ingredient with a similar ingredient.
- Double the amount of an ingredient.
- Halve the amount of an ingredient.
- Add a new ingredient.
- Eliminate an ingredient.

Naturally, you speculate that most of the changes suggested by these ingredient transformations will turn out to taste awful, and thus to be unworthy of further development.

Figure 4.8 An example of beam search. The numbers beside the nodes are straight-line distances to the goal node. Investigation spreads through the search tree level by level, but only the best w nodes are expanded, where $w = 2$ here. The remaining paths, those shown terminated by underbars, are rejected.

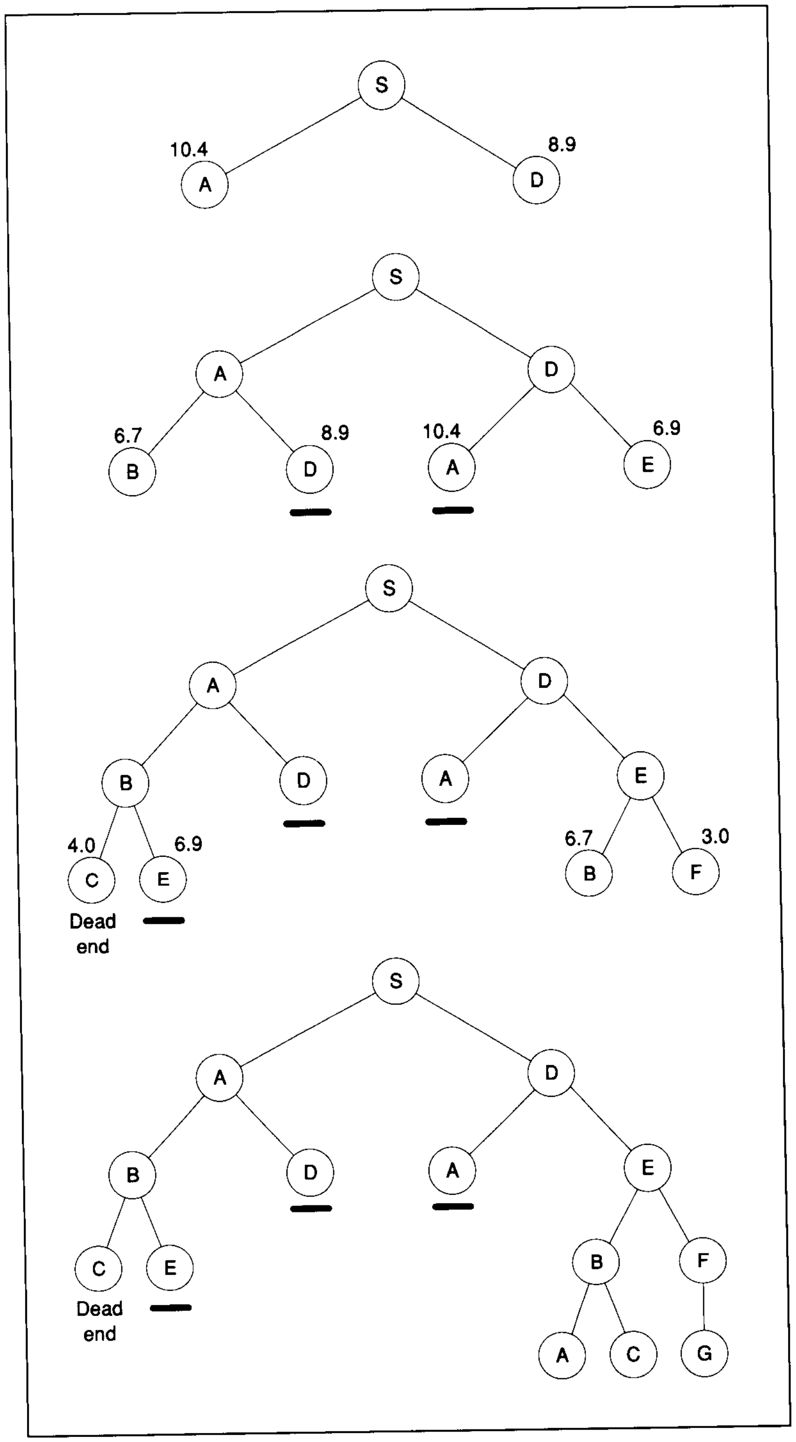
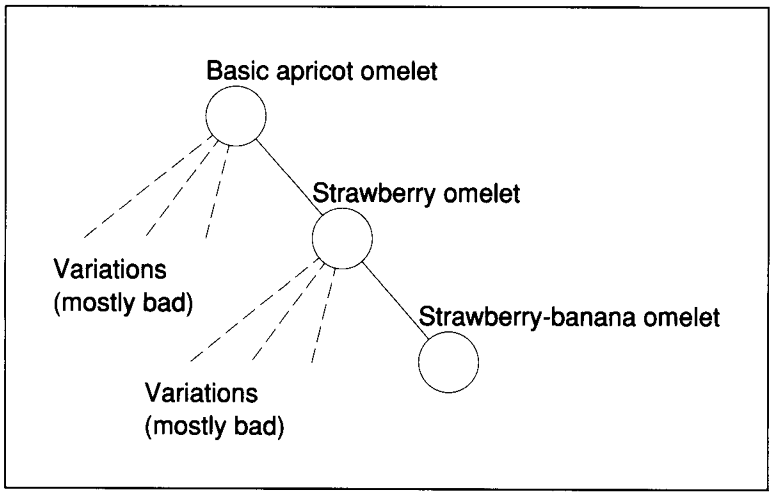


Figure 4.9 A search tree with recipe nodes. Ingredient transformations build the tree; interestingness heuristics guide the best-first search to the better prospects.



Consequently, you need **interestingness heuristics** to help you to decide on which recipes to continue to work. Here are four 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 part of the search tree descending from a basic recipe for an apricot omelet, one similar to a particular favorite of Rex Stout's fictional detective and gourmand Nero Wolfe:

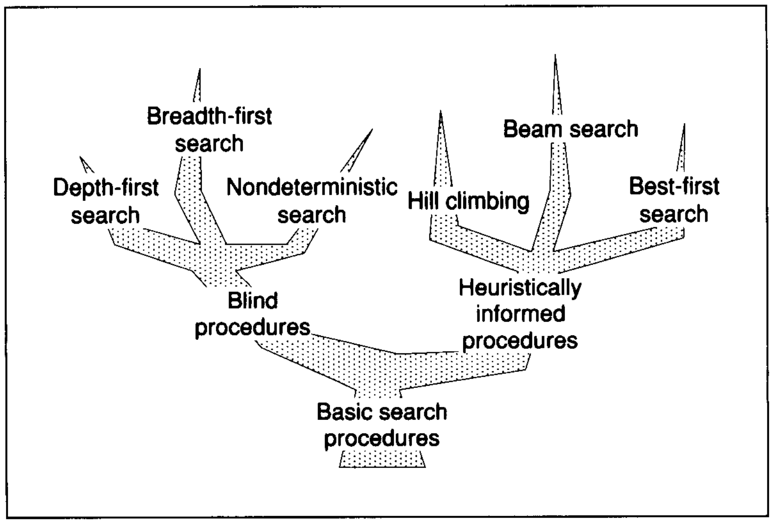
Ingredients for Apricot Omelet Recipe

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

As shown in figure 4.9, you can discover a strawberry omelet recipe using the substitution transformation on the basic apricot omelet recipe. Once you have a strawberry omelet, you can go on to discover a strawberry-peach recipe, using the addition transformation.

Of course, to be a real recipe generator, you would have to be skilled at generating plausible transformations, for you would waste too many

Figure 4.10 Part of the search family of procedures.



eggs otherwise. Essentially, you need to remember that more knowledge generally leads to reduced search time.

Search Alternatives Form a Procedure Family

You have seen that there are many ways for doing search, each with advantages:

- Depth-first search is good when unproductive partial paths are never too long.
- Breadth-first search is good when the branching factor is never too large.
- Nondeterministic search is good when you are not sure whether depth-first search or breadth-first search would be better.
- Hill climbing is good when there is a natural measure of distance from each place to the goal and a good path is likely to be among the partial paths that appear to be good at each choice point.
- Beam search is good when there is a natural measure of goal distance and a good path is likely to be among the partial paths that appear to be good at all levels.
- Best-first search is good when there is a natural measure of goal distance and a good partial path may look like a bad option before more promising partial paths are played out.

All these methods form part of the search family of procedures, as shown in figure 4.10.

SUMMARY

- Depth-first search dives into the search tree, extending one partial path at a time.

- Breadth-first search pushes uniformly into the search tree, extending many partial paths in parallel.
- Nondeterministic search moves randomly into the search tree, picking a partial path to extend at random.
- Heuristic quality measurements turn depth-first search into hill climbing. Foothills, plateaus, and ridges make hills hard to climb.
- Heuristic quality measurements also are used in beam search and best-first search. Beam search expands a fixed number of partial paths in parallel and purges the rest. Best-first search expands the best partial path.
- Heuristically guided search may lead to discovery, as in the example of recipe improvement.
- Search alternatives form a procedure family.
- More knowledge generally means less search.
- When you think you need a better search method, try to find another space to search instead.

BACKGROUND

Artificial intelligence is but one of many fields in which search is an important topic. Artificial intelligence's particular contribution lies largely in the development of heuristic methods such as those discussed in this chapter, in Chapter 6, and in many of the chapters in Part II.

You can learn more about the contributions of other fields to search from many books. Look for those with titles that include words such as *algorithms*, *linear programming*, *mathematical programming*, *optimization*, and *operations research.*

You can learn more about heuristic search from *Principles of Artificial Intelligence*, by Nils J. Nilsson [1980].

The discussion of omelet-recipe generation is based on ideas introduced in the work of Douglas B. Lenat [1977]. Lenat's AM program, a breakthrough in learning research, discovered concepts in mathematics. AM developed new concepts from old ones using various transformations, and it identified the most interesting concepts for further development.