We argued in the last chapter that intelligent action involves search, and described a variety of specific problems where search is needed for a solution—the 8-puzzle, game playing, crossword-puzzle generation, and reasoning or inference generally. In this chapter and the next two, we examine search in a bit more depth. This chapter discusses blind search, the next discusses heuristic search, and Chapter 5 discusses search procedures that are used in game playing.

We have already presented algorithms for depth-first search and breadth-first search, and we begin this chapter by examining these procedures a bit more closely. Specifically, we consider the question of how much memory and time these algorithms need.

In order to simplify the analysis, we will assume that we are working with a search tree of uniform branching factor \( b \) and depth \( d \), and that this tree has a single goal node that is at depth \( d \). A tree of this sort with \( b = 4 \) and \( d = 2 \) appears in Figure 3.1. We have denoted the root node by \( i \), the nodes at depth 1 by \( n_1, n_2, n_3, \) and \( n_4 \), and the children of \( n_i \) by \( n_{i0}, n_{i1}, n_{i2}, \) and \( n_{i3} \).

Except for the fact that it admits multiple solutions, the crossword-puzzle problem can be thought of in this way: We decide in advance on an order in which to fill the squares, view a completed crossword as a goal node if all of the words are legal and distinct, and view the node as a nongoal node if one of these conditions is violated. Given this interpretation, the branching factor for the crossword-puzzle problem is 25.

### 3.1 BREADTH-FIRST SEARCH

As discussed in Chapter 2, in breadth-first search we should view the list \( L \) of unexpanded nodes as a queue, so that newly generated children are put on the end of this list and expanded only after nodes at shallower depths have been examined. Thus during the first few iterations through the loop in Procedure 2.2.1, \( L \) takes the following values:
In order to reach the goal at depth \( d \), the interior (that is, nonfringe) nodes that must be examined are all of the nodes at depths \( 0, 1, \ldots, d - 1 \). The number of such nodes is given by

\[
1 + b + b^2 + \cdots + b^{d-1} = \frac{b^d - 1}{b - 1}
\]

(3.1)

What about the fringe nodes? We can't tell exactly how many fringe nodes will be examined, since we don't know where the fringe the goal node is located. But the best we can do is to examine only one fringe node (the goal); the worst is if we need to examine all \( b^d \) of them. The average number of nodes examined on the fringe is therefore

\[
\frac{1 + b^d}{2}
\]

(3.2)

and the average number of total nodes is the sum of (3.1) and (3.2):

\[
\frac{b^d - 1}{b - 1} + \frac{1 + b^d}{2} = \frac{2b^d - 2 + b^{d+1} + b - b^d - 1}{2(b - 1)} = \frac{b^{d+1} + b^d + b - 3}{2(b - 1)}
\]

(3.3)

For large depth, (3.3) reduces to approximately \( b^{d/2} \), which is roughly the same as the amount of time spent at the fringe. This is to be expected—in these cases, most of the time spent searching will be spent at the bottom of the tree. A graphical version of this argument appears in Figure 3.3. If \( b = 2 \) on the other hand, then (3.3) simplifies to

\[
\frac{2 - 2^d - 1}{2} = \frac{b^{d+1}}{2}
\]
Once again, we can understand this result directly. The size of the internal tree is

\[ 1 + 2 + \cdots + 2^{d-1} = 2^d - 1 \]

and the number of fringe nodes that we need to examine is approximately \(2^{d/2}\). These two values sum to \(\frac{3}{2} \cdot 2^d\).

### 3.2 Depth-First Search

If \(L\) is thought of as a queue in the breadth-first approach, it is a stack in depth-first search. Its first few values as we search the tree in Figure 3.1 are:

- \(L_0 = \{\}\)
- \(L_1 = \{n_1, n_2, n_3, n_4\}\)
- \(L_2 = \{n_1, n_2, n_3, n_4, n_5, n_6\}\)
- \(L_3 = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}\)

The complete search is shown in Figure 3.4.

The amount of memory needed by the depth-first approach is easy to compute. The most memory is needed at the first point that the algorithm reaches depth \(d\); in Figure 3.5, the marked nodes need to be saved. In general, we will need to store \(b - 1\) nodes at each depth (the siblings of the nodes that have already been expanded), together with one additional node at depth \(d\) (since we haven't expanded it yet). Thus the total space needed is given by

\[ d(b - 1) + 1 \]

### Figure 3.4

Depth-first search

![Diagram](image)

### Figure 3.5

Memory used by depth-first search

For fixed \(b\), depth-first search requires an amount of memory that is linear in \(d\), while breadth-first search requires an amount exponential in the depth of the search tree.

The time calculation is more subtle. We begin by noting that if the goal is at the far left of the tree, then depth-first search will proceed directly to it, examining a total of \(d + 1\) nodes. If it is to the far right of the tree, the entire space will be examined, a total of

\[ 1 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1} \]

nodes in all. If we could average these two expressions, we could conclude that the number of nodes examined in the average case is

\[ \frac{b^{d+1} - 1 + b^d - d + b - 1}{2(b - 1)} = \frac{b^{d+1} + b^d + b - d - 2}{2(b - 1)} \]  

(3.4)

But is it legitimate to take the average? It isn't obvious that it is, since the numbers of nodes that might be examined are not distributed uniformly over the integers from \(d + 1\) to

\[ \frac{b^{d+1} - 1}{b - 1} \]

In Figure 3.4, for example, we might examine three, four, five, or six nodes, but can't examine exactly seven, since the seventh node we examine is \(n_7\), which is at depth 1 and not a fringe node. To see that we can take the average, we will use an argument due to the mathematician Karl Gauss. When Gauss was ten, he was making a nuisance of himself, and the teacher, in order to quiet him, asked him to add the numbers from 1 to 100. Gauss replied almost immediately that the answer was 5050, having reasoned as follows:

I can pair 1 with 100, they add to 101. If I pair 2 with 99, I get 101 again; similarly for 3 paired with 98 and so on. The last pair is 50 with 51, so the sum of all the numbers is 50 \times 101 = 5050.
Chapter 3. Blind Search

Not bad for a 10-year-old!

Gauss's argument would apply equally well to showing that the average of all the integers between 1 and 100 was 101/2, and this argument is one that we can apply to our situation. Specifically, if we denote by \( B \) the number of nodes examined in the best case and by \( W \) the number examined in the worst, then in the tree in Figure 3.4 it is possible to examine a total of \( B \) nodes, or \( B = 1, B = 2, B = 3, B = 4 \) (not \( B + 4 \) and so on). But on the other side of the picture, we might examine \( W \) nodes, or \( W = 1, W = 2, W = 3, W = 4 \) (not \( W + 4 \)) and so on. As shown for the tree with \( b = 3 \) in Figure 3.6, the symmetry of the search space guarantees that we can average the best and worst cases, so that the amount of time needed by depth-first search in the average case is indeed given by (3.4).

As in the breadth-first case, we will examine large \( d \) and \( b = 2 \) a lot more closely. For large \( d \), (3.4) is once again

\[
\frac{b^d}{2}
\]

so that the work at the fringe continues to dominate the computation (see Figure 3.7). Note, however, that the term of order \( b^{d+1} \) that appeared in the breadth-first result (3.3) is missing from (3.4): a slightly more accurate estimate of the time needed by breadth-first search is

\[
\frac{b^d}{2} \left( 1 + \frac{1}{b} \right)
\]

We see from this that breadth-first search is more computationally expensive than depth-first search by a factor of

\[
1 + \frac{1}{b} = \frac{b+1}{b}
\]

Typical values for this expression appear in Figure 3.6.

For \( b = 2 \), (3.4) simplifies to

\[
\frac{2^{d+1} + 2d + 2 - d - 2}{2} = \frac{2^{d+1} + d}{2} \approx 2^d
\]

As in the breadth-first case, we can understand this result directly. The entire search space is of size \( 2^{d+1} \), and depth-first search will, on average, examine about half of it—in other words, \( 2^d \) nodes.

Comparing the two approaches, we see that depth-first search is somewhat more efficient in time and vastly more efficient in space than its
3.3 ITERATIVE DEEPENING

Is there any way to get the best of both these approaches, using an algorithm that has the space requirements of depth-first search and the performance properties of breadth-first search? The answer is yes: the approach that does this is known as iterative deepening and was first thoroughly investigated by Richard Korf.

There are two separate ways to think about iterative deepening. Perhaps the simplest (although it doesn’t explain the name) is that it’s just the same as breadth-first search, except that instead of storing all of the nodes at intermediate depths, we reexplore them when the time comes to expand them.

Why is this a viable approach? It seems as if the amount of time spent reexpanding the internal nodes would dwarf the time spent looking for the answer, but this is not the case—recall that in all search problems, the bulk of the computational effort is spent examining the nodes at the fringe.

The other way to think of iterative deepening is the following. In the cases in which depth-first search performs poorly, the depth of the goal node is less than the depth of the tree as a whole—either because the first goal found is at a greater depth than the shallowest solution to the problem, or because the depth of the tree is infinite, and so on. What iterative deepening does is search the tree in a way that guarantees that the goal depth and tree depth match.

The idea is search the tree initially with an artificial depth cutoff of 1, so that any node below depth 1 is not examined. If this approach succeeds in finding a solution at depth 1, the solution is returned. If not, the tree is searched again but with a depth cutoff of 2. Each of these iterative searches proceeds in depth-first fashion.

PROCEDURE Iterative deepening

1. Set c = 1; this is the current depth cutoff.
2. Set L to be a list of the initial nodes in the problem.
3. Let n be the first node on L. If L is empty, increment c and return to step 2.
4. If n is a goal node, stop and return it and the path from the initial node to n.
5. Otherwise, remove n from L. Provided that the depth of n is less than c, add to the front of L all of n’s children, labelling each with its path from the initial node. Return to step 3.

If the shallowest solution is at depth g, the depth-first search to that depth will succeed; it follows from this that iterative deepening will always return the shallowest solution to the problem in question. Since each of the individual searches is performed depth-first, the amount of memory required by the method is the same as for the depth-first approach.

How about the number of nodes examined? The order in which the nodes are expanded in our sample problem is shown in Figure 3.10; the
nodes at depth 0 and 1 are labelled with a sequence of numbers since they are examined multiple times.

In general, the number of nodes examined in the final (successful) iteration is given by \( (3.4) \). In the previous iterations, the failing searches to depths 1, 2, \ldots, \( d - 1 \) will need to examine the entire tree at these depths; the size of the tree to depth \( j \) is given by

\[
1 + b + \ldots + b^j = \frac{b^{j+1} - 1}{b - 1}
\]

The total number of nodes examined in the failing searches is therefore

\[
\sum_{j=0}^{d-1} \frac{b^{j+1} - 1}{b - 1} = \frac{1}{b - 1} \left( b^{d+1} - \frac{b^{d+1}}{b - 1} - d \right)
\]

Combining this with \( (3.4) \) and simplifying the result, the total number of nodes examined in this approach is seen to be

\[
\frac{b^{d+2} + b^{d+3} + b^d + b^2 - 4bd - 5b + 3d + 2}{2(b - 1)^2}
\]

For large \( d \), this is dominated by

\[
\frac{(b + 1)b^{d+1}}{2(b - 1)^2}
\]

Since the time needed by depth-first search is dominated by

\[
\frac{b^{d+1}}{2(b - 1)^2}
\]

the ratio of the time needed by iterative deepening to that needed by depth-first search is given by

\[
\frac{b + 1}{b - 1}
\]

Typical values of this expression appear in Figure 3.11. As we can see from the table, the cost of repeating the work at shallow depths is not prohibitive. Since iterative deepening avoids the problems encountered in depth-first search, it is the method of choice in many search problems.

As usual, we examine the case \( b = 2 \) a bit more closely. In the depth-first case, we need to examine half of the entire tree; since the tree is of size \( 2^{d+1} \), depth-first search needs to examine \( 2^d \) nodes. In iterative deepening, the failing searches take time

\[
1 + 2 + \ldots + 2^d = 2^{d+1} - 2
\]

since these are the sizes of the subtrees examined. Since this time is twice the time needed by the successful search to depth \( d \), the overall factor in the \( b = 2 \) case is 3, in agreement with \((3.6)\) and Figure 3.11.

How good is iterative deepening? In a very real sense, it is an optimal blind search procedure. The reason is that for a fixed branching factor, it takes space \( \Theta(b^d) \) and time \( \Theta(b^d) \), as we are about to see, it is impossible to improve on this behavior. It is clear, for example, that if the program is

![Figure 3.11](image)

<table>
<thead>
<tr>
<th>( b )</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1.5</td>
</tr>
<tr>
<td>10</td>
<td>1.2</td>
</tr>
<tr>
<td>25</td>
<td>1.06</td>
</tr>
<tr>
<td>100</td>
<td>1.02</td>
</tr>
</tbody>
</table>

\[ \text{By \( \Theta \) here, I mean that for large depth the amount of space needed is dominated by a term of the form } kd \text{ for some constant } k. \text{ In a similar way, the amount of time needed is dominated by a term of the form } k b^d. \]
expected to return the path to a goal node at depth \( d \), at least space \( d \) is required.

What about time? Since the goal node is distributed randomly along the fringe and the search technique is blind, it follows that we will, on average, need to examine at least half of this fringe. Since the fringe is of size \( b^d \), we can therefore expect any blind search procedure to take time \( O(b^d) \).

Given this, we can strengthen our remarks with regard to the space needed by the following argument: Consider any algorithm that takes time \( O(b^d) \) to run: how much space does it need in order to distinguish its own internal states?

If all of the internal states of the algorithm are to remain distinct, it clearly needs at least as much space as it would need to count to \( b^d \). But counting to \( b^d \) uses \( \log(b^d) \) bits; since \( \log(b^d) = d \log(b) \) is \( \Theta(d) \) for fixed \( b \), we see that any blind search algorithm needs space \( \Theta(d) \) whether it is expected to return a path to the answer or not.

### 3.4 Iterative Broadening

Although iterative deepening is optimal in the sense that \( O(b^d) \) is the best one can do in general, there is one other blind search technique that can often improve on this result in practice—not by reducing the complexity of the problem, but by making it likely that a smaller portion of the fringe is examined if multiple goal nodes exist. This technique is known as iterative broadening.

As we noted in the previous chapter, it is important where possible to recognize bad choices early in the search so that we need not expand the entire search tree under these choices before finding a goal node. But how are we to do this if the search is blind?

The answer lies in realizing that in most practical problems, the goal nodes are not randomly distributed because it is possible to make “fatal” mistakes early in the search. In Figure 3.12, for example, it might well be that the insertion of the word TONAL basically doomed us. Although we could fill in a variety of additional words, we would eventually have to insert TOAST as described earlier and would then be unable to fill the marked word in the figure. What iterative broadening does is to take continuing failure to find a goal below a node high in the search tree as evidence that such a fatal mistake has been made.

Just as iterative deepening imposes artificial depth limits on the search and gradually increases those limits until a solution is found, iterative broadening imposes artificial breadth limits, increasing them until a solution is found.

#### Procedure

**Iterative broadening**

1. Set \( c = 2 \); this is the current breadth cutoff.
2. Set \( L \) to be the set of initial nodes in the problem.
3. Let \( n \) be the first node on \( L \). If \( L \) is empty, increment \( c \) and return to step 1.
4. If \( n \) is a goal node, stop and return it and the path from the initial node to \( n \).
5. Otherwise, remove \( n \) from \( L \). Add to the front of \( L \) the first \( c \) of \( n \)'s children, labelling each with its path from the initial node. Return to step 3.

We initially search the tree with a breadth cutoff of 2, then of 3, and so on. A breadth cutoff of \( c \) means that a maximum of \( c \) children of any given node can be examined before the node is abandoned as a failure. In Figure 3.13, we show a tree with branching factor 4 but a breadth cutoff of 3. The node ordering given by iterative broadening in our usual example is shown in Figure 3.14.

As with iterative deepening, iterative broadening is a viable search technique because there are at most \( b^d \) searches and the breadth searches with cutoff \( c \) take time \( \Theta(b^c) \), which will be small relative to \( b^d \) if \( c \ll b \). The maximum amount of time spent by the approach is approximately

\[
1 + b + b^2 + \cdots + b^d
\]
which is approximately $b^{d/2}$ for large $b$ (a factor of $b^d$ worse than depth-first search) and is approximately $b^d$ (virtually no cost at all) for large $d$.

Iterative broadening can lead to large savings in the amount of time needed to solve search problems if multiple goal nodes exist and if it is possible to make false errors early in the search. In the large depth limit, the method is likely to find a solution more quickly than single depth-first search whenever the number of goal nodes exceeds three.

**Nonsystematic search** Iterative broadening is a useful search technique because it keeps us from concentrating too much of our effort in a region of the search space that contains no goal nodes. This lesson is so important that a variety of authors have suggested recently that you fail to find a goal in solving a search problem, you should jump to a distant portion of the search space that might be better.

The problem with doing this is that it is difficult to keep track of which nodes you've examined and which you haven't; you can't store all of the examined nodes without using an enormous amount of memory. So what these new approaches suggest is that you simply not worry about this, possibly searching some portions of the search space many times and other portions not at all. Because of the scatter-shot nature of the resulting search, these methods are called nonsystematic. There is no guarantee that a nonsystematic search will find a goal node every time that the search space contains one.

In spite of this, the observed performance of nonsystematic methods on large search problems is often comparable to (and in some cases better than) the performance of systematic approaches. Roughly speaking, the search spaces in these examples are so large that the nonsystematic methods are unlikely to examine some area of the space over and over again, and the spaces are also so large that a complete examination in the search for a solution (where the systematic methods could be expected to shine) is simply impractical. This work is all very new, but the next few years should shed substantial light on the general question of whether systematic or nonsystematic methods should be preferred when solving hard problems.

### 3.5 Searching Graphs

Throughout this chapter, we have assumed that the search space being examined is a tree and not a graph, so that it is impossible to reach the same node using each of several paths from the root node.

This assumption is clearly wrong; in sliding tile puzzles, for example, there may be many ways to arrive at a particular configuration of the tiles. The tower of Hanoi is similar, as is the crossword-puzzle problem. In Figure 3.15, we have redrawn Figure 2.11 to show explicitly that the search space in the missionaries and cannibals problem is a graph and not a tree.

We can, of course, search a graph by pretending that it is a tree, as in Figure 3.11 itself. The problem with doing this is that the search may become less efficient as a result. There are two separate ways in which graph search can be simplified.

#### 3.5.1 Open and Closed Lists

The first way is to simply avoid adding a node to $L$ if it already appears there. After all, there is never any reason to commit to searching the same node twice. More efficient still is to keep track of those nodes that have been examined and removed from $L$ since these nodes have already been considered. We have no need to consider them again should they be regenerated for some reason.

The basic search procedure, Procedure 2.2.1, does not keep track of the nodes that have been examined and removed from $L$. If we want to do this, we need to maintain a list of these nodes. This list is often referred to as the list of closed nodes, the list $L$ of nodes still to be examined is called the list of open nodes.

Of course, finding any particular node in the open or closed list takes both time and space, and those resources need to be justified via an effective reduction in the size of the graph being searched. By using hash tables, the time to find (or fail to find) a particular node in the open or closed list can be reduced to a constant; the space requirements are a bit more difficult to deal with.

---

**Figure 3.15** Missions and cannibals again
CHAPTER 4

HEURISTIC SEARCH

We argued in the last chapter that any technique used to search blindly through a space of depth \( d \) and branching factor \( b \) would of necessity take time \( O(b^d) \) to find a single goal node on the fringe of the search tree. In practice, this is unacceptable. As an example, if the time needed to generate a plan grows exponentially with the length of the plan, a planner will be unable to produce plans of interesting length.

In exploiting a large search space, the trick is to use additional information about the problem being considered to avoid examining most of the nodes that might conceivably lead to solutions. Instead of selecting the node to expand next in a domain-independent way as in the previous chapter, we need to use domain-specific information for the same purpose.

Of course, the problem of finding the best node to expand next is generally no easier than the search problem that we are trying to solve. So what we typically do is apply some sort of heuristic, or "rule of thumb," to decide what to do. Given a list of nodes to be expanded, we might simply guess how far each is from a goal node and expand the one we thought to be closest.

As an example, consider the instance of the 8-puzzle shown in Figure 4.1, and suppose that our goal is to arrange the tiles from 1 to 8 clockwise around the edge of the puzzle. If we estimate the distance to the goal by simply counting the number of misplaced tiles, then we will expect it to take three moves to solve the problem in Figure 4.1, since three tiles are misplaced there. (The 6, 7, and 8 are all misplaced.) If we move the 6 to the right, then only two tiles are misplaced. Moving the 8 down leaves the heuristic estimate of the distance to the goal unchanged, while moving the 5 to the left actually increases the expected distance. We summarize this below:

<table>
<thead>
<tr>
<th>Blank Moves</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>2</td>
</tr>
<tr>
<td>right</td>
<td>4</td>
</tr>
<tr>
<td>up</td>
<td>3</td>
</tr>
</tbody>
</table>

We see from this that if our search heuristic is to move as quickly toward the goal as possible, we will select the move of moving the blank to the left in Figure 4.1. Extending this analysis will lead us to move the blank up next and then to the right, arriving at the goal configuration after three moves. This technique always expands next the child of the current node that seems closest to the goal.

Of course, things may not always work out so easily because our heuristic function estimating the distance to the goal may make mistakes in some cases. After moving the blank up in Figure 4.1, we still estimate the distance to the goal as being only three moves, although it is not too hard to see that four are actually required.

Another problem is that we need to limit the amount of time spent computing the heuristic values used in selecting a node for expansion. We have already discussed this in Chapter 2, where we discussed the inevitable trade-off between base-level and meta-level activity.

There is no real "solution" to these problems of inaccurate or computationally expensive heuristics; the bottom line is that search problems sometimes will take an exponential amount of time to solve, and there is simply no way around this. But there is a third problem with our approach that is more tractable.

This problem can best be understood by considering an example. Suppose that we are looking for the solution to a maze, where our estimate of the value of a node is simply the Manhattan distance between our current position and the goal from the maze. Then at any given point we do our best to move closer to the goal.

Now consider the maze shown in Figure 4.2, where we enter the maze on the left and exit on the right. As shown, there are two solutions—a simple one that begins with a step downward and a much more complex one that begins with a step directly toward the exit but is then deflected in other directions. The algorithm we have described will find the longer...
of these two paths, since it will begin by moving toward the goal and then be committed to this choice.

If we want to find the shortest path to the goal in a problem such as this, a method that avoids this difficulty is a search algorithm known as A*, and we will discuss it later in this chapter. Before doing so, however, let me present heuristic search from a rather different perspective, drawing an analogy between these problems and the conventional problem of maximizing a function of several variables.

4.1 SEARCH AS FUNCTION MAXIMIZATION

Suppose that we have designed a robot whose purpose is to explore the surface of Mars. After being released from a landing craft of some sort, the robot is expected to wander around the surface of the planet to the most interesting location it can find, and then to take a variety of surface measurements at that point.

What makes a surface location interesting is a function of a variety of factors—apparent surface makeup, geological features, alien footprints, what have you. The robot has no difficulty evaluating the interest of the point at which it finds itself, but cannot predict the interest of any other point without actually visiting it. How is it to find the most interesting point on the planet’s surface?

From a formal point of view, this is a simple function maximization problem: we have a function $f$ that measures the interestingness of a point $p$ on the surface, and need to find that value of $p$ for which the function’s value is maximal.

4.1.1 Hill Climbing

What is the functional analog to the search procedure described earlier? Informally, we always move in the direction of apparently largest $f$; the functional analog is to attempt to find the global maximum of a function of many variables by always moving in the direction in which the rate of change is greatest.

This technique is known as hill climbing or steepest ascent for the obvious reason: it is as if one attempted to find the highest point in a landscape by simply walking as much uphill as possible. Here is the search version.

**PROCEDURE**

4.1.1 Hill climbing

1. Set $L$ to be a list of the initial nodes in the problem, sorted by their expected distance to the goal. Nodes expected to be close to the goal should precede those that are farther from it.
2. Let $a$ be the first node on $L$. If $L$ is empty, fail.
3. If $a$ is a goal node, stop and return it and the path from the initial node to $a$.
4. Otherwise, remove $a$ from $L$. Sort $a$’s children by their expected distance to the goal, label each child with its path from the initial node, and add the children to the front of $L$. Return to step 2.

In this version of the algorithm, we always take a step from a child of the previously expanded node when possible; this gives hill climbing a “depth-first” flavor. If we drop this restriction, we get an algorithm known as best-first search.

**PROCEDURE**

4.1.2 Best-first search

1. Set $L$ to be a list of the initial nodes in the problem.
2. Let $a$ be the node on $L$ that is expected to be closest to the goal. If $L$ is empty, fail.
3. If $a$ is a goal node, stop and return it and the path from the initial node to $a$.
4. Otherwise, remove $a$ from $L$ and add to $L$ all of $a$’s children, labelling each with its path from the initial node. Return to step 2.

There are three obvious problems with hill climbing, and these are depicted in Figure 4.3. The first, and most important, involves the problem of local maxima. It is all too easy to construct situations in which the rate of change is negative in all directions even though the global maximum.