Problem Solving as Search

Problem solving is an important aspect of Artificial Intelligence. A problem can be considered to consist of a **goal** and a set of actions that can be taken to lead to the goal. At any given time, we consider the **state** of the **search space** to represent where we have reached as a result of the actions we have applied so far.

For example, consider the problem of looking for a contact lens on a foot- ball field. The **initial state** is how we start out, which is to say we know that the lens is somewhere on the field, but we don't know where. If we use the representation where we examine the field in units of one square foot, then our first action might be to examine the square in the top-left corner of the field. If we do not find the lens there, we could consider the state now to be that we have examined the top-left square and have not found the lens. After a number of actions, the state might be that we have examined 500 squares, and we have now just found the lens in the last square we examined. This is a **goal state** because it satisfies the goal that we had of finding a contact lens.

Search is a method that can be used by computers to examine a problem space like this in order to find a goal. Often, we want to find the goal as quickly as possible or without using too many resources. A problem space can also be considered to be a **search space** because in order to solve the problem, we will search the space for a goal state. We will continue to use the term **search space** to describe this concept.

In this chapter, we will look at a number of methods for examining a search space. These methods are called **search methods**.

Data-Driven or Goal-Driven Search

There are two main approaches to searching a search tree, which roughly correspond to the top-down and bottom-up approaches discussed in Section 3.12.1. **Data-driven search** starts from an initial state and uses actions that are allowed to move forward until a goal is reached. This approach is also known as **forward chaining**.

Alternatively, search can start at the goal and work back toward a start state, by seeing what moves could have led to the goal state. This is **goal-driven search**, also known as **backward chaining**.

Most of the search methods we will examine in this chapter are data-driven search: they start from an initial state (the root node in the search tree) and work toward the goal node.

In many circumstances, goal-driven search is preferable to data driven- search, but for most of this part of the book, when we refer to "search," we are talking about data-driven search.

Goal-driven search and data-driven search will end up producing the same results but depending on the nature of the problem being solved, in some cases one can run more efficiently than the other—in particular, in some situations one method will involve examining more states than the other.

Goal-driven search is particularly useful in situations in which the goal can be clearly specified (for example, a theorem that is to be proved or finding an exit from a maze). It is also clearly the best choice in situations such as medical diagnosis where the goal (the condition to be diagnosed) is known, but the rest of the data (in this case, the causes of the condition) need to be found.

Data-driven search is most useful when the initial data are provided, and it is not clear what the goal is. For example, a system that analyzes astronomical data and thus makes deductions about the nature of stars and planets would receive a great deal of data, but it would not necessarily be given any direct goals. Rather, it would be expected to analyze the data and determine conclusions of its own. This kind of system has a huge number of possible goals that it might locate. In this case, data-driven search is most appropriate.

It is interesting to consider a maze that has been designed to be traversed from a start point in order to reach a particular end point. It is nearly always far easier to start from the end point and work back toward the start point. This is because several dead-end paths have been set up from the start (data) point, and only one path has been set up to the end (goal) point. As a result, working back from the goal to the start has only one possible path.

Generate and Test

The simplest approach to search is called **Generate and Test**. This simply involves generating each node in the search space and testing it to see if it is a goal node. If it is, the search has succeeded and need not carry on. Other- wise, the procedure moves on to the next node.

This is the simplest form of **brute-force search** (also called **exhaustive search**), so called because it assumes no additional knowledge other than how to traverse the search tree and how to identify leaf nodes and goal nodes, and it will ultimately examine every node in the tree until it finds a goal.

To successfully operate, Generate and Test needs to have a suitable **Generator**, which should satisfy three properties:

1. It must be complete: In other words, it must generate every possible solution; otherwise, it might miss a suitable solution.
2. It must be nonredundant: This means that it should not generate the same solution twice.
3. It must be well informed: This means that it should only propose suitable solutions and should not examine possible solutions that do not match the search space.

The Generate and Test method can be successfully applied to a number of problems and indeed is the manner in which people often solve problems where there is no additional information about how to reach a solution. For example, if you know that a friend lives on a particular road, but you do not know which house, a Generate and Test approach might be necessary; this would involve ringing the doorbell of each house in turn until you found your friend. Similarly, Generate and Test can be used to find solutions to combinatorial problems such as the eight queens problem that is introduced in Chapter 5.

Generate and Test is also sometimes referred to as a **blind** search technique because of the way in which the search tree is searched without using any information about the search space.

More systematic examples of brute-force search are presented in this chapter, in particular, depth-first search and breadth-first search.

More “intelligent” (or informed) search techniques are explored later in this chapter.

Depth-First Search

A commonly used search algorithm is **depth-first search**. Depth-first search is so called because it follows each path to its greatest depth before moving on to the next path. The principle behind the depth-first approach is illustrated in Figure 4.1. Assuming that we start from the left side and work toward the right, depth-first search involves working all the way down the left-most path in the tree until a leaf node is reached. If this is a goal state, the search is complete, and success is reported.

If the leaf node does not represent a goal state, search backtracks up to the next highest node that has an unexplored path. In Figure 4.1, after examining node G and discovering that it is not a leaf node, search will backtrack to node D and explore its other children. In this case, it only has one other child, which is H. Once this node has been examined, search backtracks to the next unexpanded node, which is A, because B has no unexplored children.

This process continues until either all the nodes have been examined, in which case the search has failed, or until a goal state has been reached, in which case the search has succeeded. In Figure 4.1, search stops at node J, which is the goal node. As a result, nodes F, K, and L are never examined.

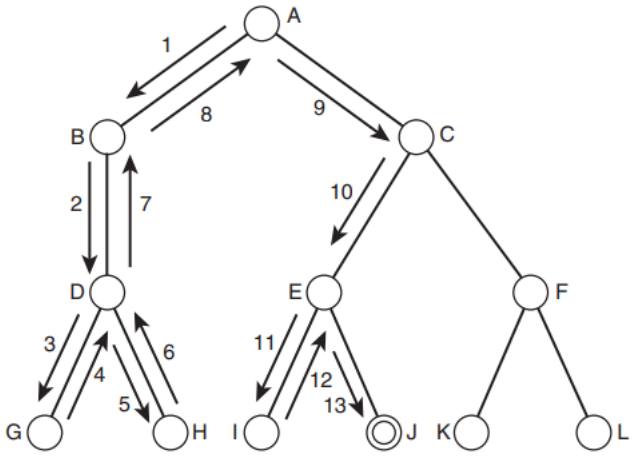

Figure 4.1
Illustrating depth-first search

Depth-first search uses a method called **chronological backtracking** to move back up the search tree once a dead end has been found. Chronological backtracking is so called because it undoes choices in reverse order of the time the decisions were originally made. We will see later in this chapter that **nonchronological backtracking**, where choices are undone in a more structured order, can be helpful in solving certain problems.

Depth-first search is an example of **brute-force search**, or **exhaustive search**.

Depth-first search is often used by computers for search problems such as locating files on a disk, or by search engines for **spidering** the Internet.

As anyone who has used the *find* operation on their computer will know, depth-first search can run into problems. In particular, if a branch of the search tree is extremely large, or even infinite, then the search algorithm will spend an inordinate amount of time examining that branch, which might never lead to a goal state.

Breadth-First Search

An alternative to depth-first search is breadth-first search. As its name suggests, this approach involves traversing a tree by breadth rather than by depth. As can be seen from Figure 4.2, the breadth-first algorithm starts by examining all nodes one level (sometimes called one **ply**) down from the root node.

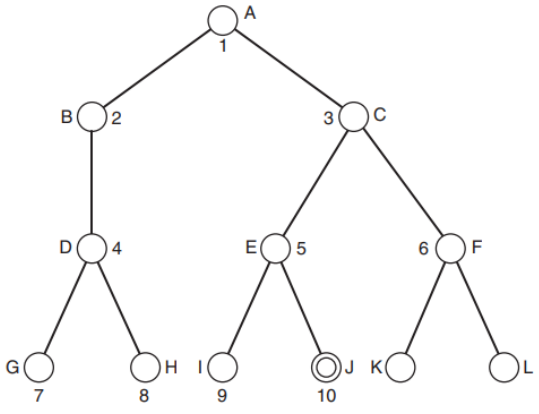

Figure 4.2
Illustrating breadth-first search. The numbers indicate the order in which the nodes are examined.

Table 4.1 Comparison of depth-first and breadth-first search

Scenario	Depth first	Breadth first
Some paths are extremely long, or even infinite	Performs badly	Performs well
All paths are of similar length	Performs well	Performs well
All paths are of similar length, and all paths lead to a goal state	Performs well	Wasteful of time and memory
High branching factor	Performance depends on other factors	Performs poorly

If a goal state is reached here, success is reported. Otherwise, search continues by expanding paths from all the nodes in the current level down to the next level. In this way, search continues examining nodes in a particular level, reporting success when a goal node is found, and reporting failure if all nodes have been examined and no goal node has been found.

Breadth-first search is a far better method to use in situations where the tree may have very deep paths, and particularly where the goal node is in a shallower part of the tree. Unfortunately, it does not perform so well where the branching factor of the tree is extremely high, such as when examining **game trees** for games like Go or Chess.

Breadth-first search is a poor idea in trees where all paths lead to a goal node with similar length paths. In situations such as this, depth-first search would perform far better because it would identify a goal node when it reached the bottom of the first path it examined.

The comparative advantages of depth-first and breadth-first search are tabulated in Table 4.1.

As will be seen in the next section, depth-first search is usually simpler to implement than breadth-first search, and it usually requires less memory usage because it only needs to store information about the path it is currently exploring, whereas breadth-first search needs to store information about all paths that reach the current depth. This is one of the main reasons that depth-first search is used so widely to solve everyday computer problems.

The problem of infinite paths can be avoided in depth-first search by applying a **depth threshold**. This means that paths will be considered to have terminated when they reach a specified depth. This has the disadvantage that some goal states (or, in some cases, the only goal state) might be missed but ensures that all branches of the search tree will be explored in reasonable time. As is seen in Chapter 6, this technique is often used when examining game trees.

Properties of Search Methods

As we see in this chapter, different search methods perform in different ways. There are several important properties that search methods should have in order to be most useful.

In particular, we will look at the following properties:

- Complexity
- Completeness
- Optimality
- Admissibility
- irrevocability

In the following sections, we will explain what each of these properties means and why they are useful. We will continue to refer to many of these properties (in particular, completeness and complexity) as we examine a number of search methods in this chapter and in Chapter 5.

Complexity

In discussing a search method, it is useful to describe how efficient that method is, over time and space. The **time complexity** of a method is related to the length of time that the method would take to find a goal state. The **space complexity** is related to the amount of memory that the method needs to use.

It is normal to use Big-O notation to describe the complexity of a method. For example, breadth-first search has a time complexity of $O(b^d)$, where b is the branching factor of the tree, and d is the depth of the goal node in the tree.

Depth-first search is very efficient in space because it only needs to store information about the path it is currently examining, but it is not efficient in time because it can end up examining very deep branches of the tree.

Clearly, complexity is an important property to understand about a search method. A search method that is very inefficient may perform reasonably well for a small test problem, but when faced with a large real-world problem, it might take an unacceptably long period of time. As we will see, there can be a great deal of difference between the performance of two search methods and selecting the one that performs the most efficiently in a particular situation can be very important.

This complexity must often be weighed against the adequacy of the solution generated by the method. A very fast search method might not always find the best solution, whereas, for example, a search method that examines every possible solution will guarantee to find the best solution, but it will be very inefficient.

Completeness

A search method is described as being **complete** if it is guaranteed to find a goal state if one exists. Breadth-first search is complete, but depth-first search is not because it may explore a path of infinite length and never find a goal node that exists on another path.

Completeness is usually a desirable property because running a search method that never finds a solution is not often helpful. On the other hand, it can be the case (as when searching a game tree, when playing a game, for example) that searching the entire search tree is not necessary, or simply not possible, in which case a method that searches enough of the tree might be good enough.

A method that is not complete has the disadvantage that it cannot necessarily be believed if it reports that no solution exists.

Optimality

A search method is **optimal** if it is guaranteed to find the best solution that exists. In other words, it will find the path to a goal state that involves taking the least number of steps.

This does not mean that the search method itself is efficient—it might take a great deal of time for an optimal search method to identify the optimal solution—but once it has found the solution, it is guaranteed to be the best one. This is fine if the process of searching for a solution is less time consuming than actually implementing the solution. On the other hand, in some cases implementing the solution once it has been found is very simple, in which case it would be more beneficial to run a faster search method, and not worry about whether it found the optimal solution or not.

Breadth-first search is an optimal search method, but depth-first search is not. Depth-first search returns the first solution it happens to find, which may be the worst solution that exists. Because breadth-first search examines all nodes at a given depth before moving on to the next depth, if it finds a solution, there cannot be another solution before it in the search tree.

In some cases, the word optimal is used to describe an algorithm that finds a solution in the quickest possible time, in which case the concept of **admissibility** is used in place of optimality. An algorithm is then defined as **admissible** if it is guaranteed to find the best solution.

Irrevocability

Methods that use backtracking are described as tentative. Methods that do not use backtracking, and which therefore examine just one path, are described as irrevocable. Depth-first search is an example of tentative search. In Section 4.13 we look at hill climbing, a search method that is irrevocable.

Irrevocable search methods will often find suboptimal solutions to problems because they tend to be fooled by local optima—solutions that look good locally but are less favorable when compared with other solutions elsewhere in the search space.

Why Humans Use Depth-First Search?

Both depth-first and breadth-first search are easy to implement, although depth-first search is somewhat easier. It is also somewhat easier for humans to understand because it much more closely relates to the natural way in which humans search for things, as we see in the following two examples.

Example 1: Traversing a Maze

When traversing a maze, most people will wander randomly, hoping they will eventually find the exit (Figure 4.3). This approach will usually be successful eventually but is not the most rational and often leads to what we call "going round in circles." This problem, of course, relates to search spaces that contain loops, and it can be avoided by converting the search space into a search tree.

An alternative method that many people know for traversing a maze is to start with your hand on the left side of the maze (or the right side, if you prefer) and to follow the maze around, always keeping your left hand on the left edge of the maze wall. In this way, you are guaranteed to find the exit. As can be seen in Figure 4.3, this is because this technique corresponds exactly to depth-first search.

In Figure 4.3, certain special points in the maze have been labelled:

- A is the entrance to the maze.
- M is the exit from the maze.
- C, E, F, G, H, J, L, and N are dead ends.
- B, D, I, and K are points in the maze where a choice can be made as to which direction to go next.

In following the maze by running one's hand along the left edge, the following path would be taken:

A, B, E, F, C, D, G, H, I, J, K, L, M

You should be able to see that following the search tree using depth-first search takes the same path. This is only the case because the nodes of the search tree have been ordered correctly. The ordering has been chosen so that each node has its left-most child first and its right-most child last. Using a different ordering would cause depth-first search to follow a different path through the maze.

Example 2: Searching for a Gift

When looking for a Christmas present for a relative in a number of shops, each of which has several floors, and where each floor has several departments, depth-first search might be a natural, if rather simplistic, approach.

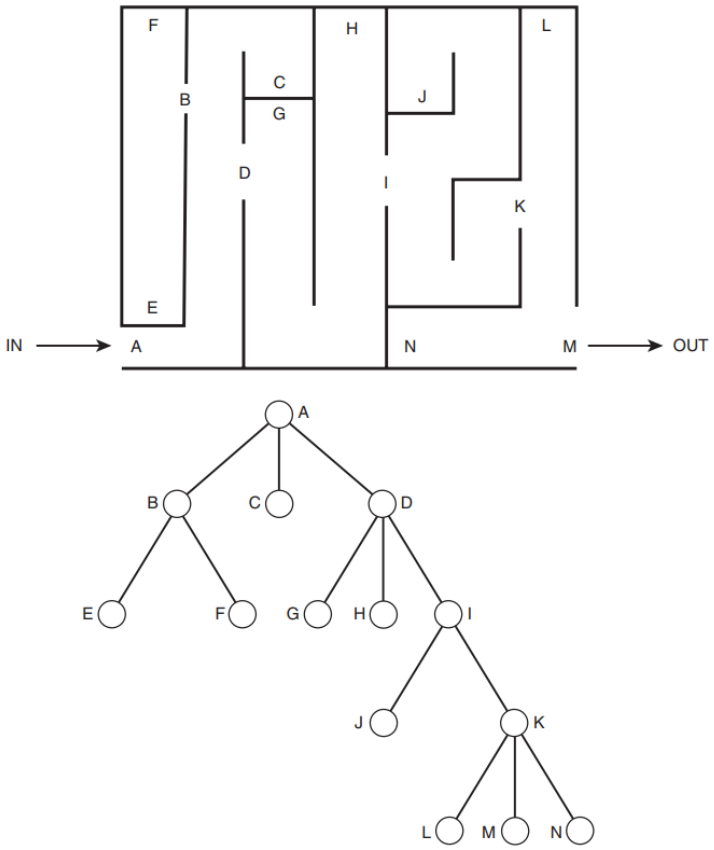

Figure 4.3
A maze and a search tree representation of the maze.

This would involve visiting each floor in the first building before moving on to the next building. A breadth-first approach would mean examining the first department in each shop, and then going back to examine the second department in each shop, and so on. This way does not make sense due to the spatial relationship between the departments, floors, and shops. For a computer, either approach would work equally well as long as a representation was used where moving from one building to another did not take any computation time.

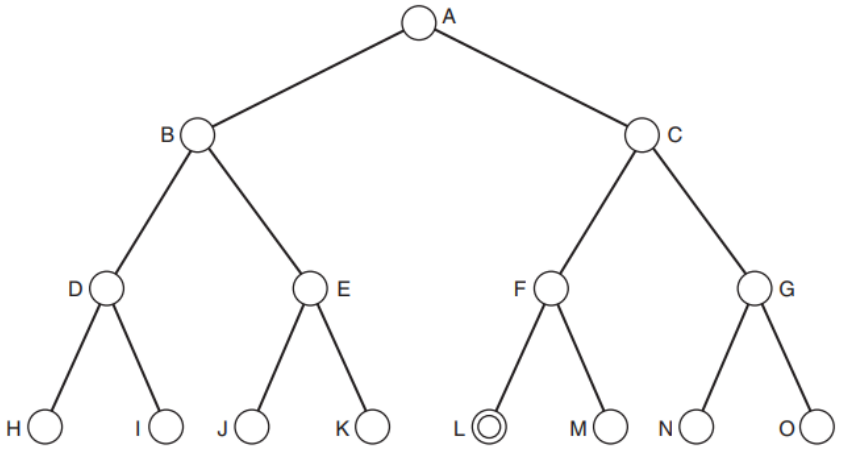

Figure 4.4
A simple search tree with fifteen nodes. The tree has a branching factor of two and a depth of three.

In both of the examples above, it can be seen that using breadth-first search, although a perfectly reasonable approach for a computer system, would be rather strange for a human. This is probably because with depth-first search, the approach is to explore each path fully before moving onto another path, whereas with breadth-first search, the approach involves revisiting and extending particular paths many times.

Despite this, implementations in software of both algorithms are nearly identical, at least when expressed in pseudocode.

Implementing Depth-First and Breadth-First Search

A pseudocode implementation of depth-first search is given below.

The variable **state** represents the current state at any given point in the algorithm, and **queue** is a data structure that stores a number of states, in a form that allows insertion and removal from either end. In this algorithm, we always insert at the front and remove from the front, which as we will see later on means that depth-first search can be easily implemented using a stack.

In this implementation, we have used the function **successors (state)**, which simply returns all successors of a given state.

```
Function depth ()
{
     queue = [];   // initialize an empty queue
     state = root_node; // initialize the start statewhile
     (true)
     {
          if is_goal (state)
                  then return SUCCESS
          else add_to_front_of_queue (successors (state));if
          queue == []
                  then report FAILURE;
          state = queue [0]; // state = first item in queue
          remove_first_item_from (queue);
     }
}
```

Table 4.2 shows the states that the variables **queue** and **state** take on when running the depth-first search algorithm over a simple search tree, as shown in Figure 4.3.

In fact, depth-first search can be readily implemented on most computer systems using a **stack**, which is simply a “last in first out” queue (sometimes called a LIFO). In this way, a recursive version of the algorithm given above can be used, as follows. Because this function is recursive, it needs to be called with an argument:

```
recursive_depth (root_node);
```

The function is defined as follows:

```
Function recursive_depth (state)
{
        if is_goal (state)
                then return SUCCESS
        else
        {
                remove_from_stack (state);
                add_to_stack (successors (state))
        }
        while (stack != [])
        {
                if recursive_depth (stack [0]) == SUCCESS
                        then return SUCCESS;
                remove_first_item_from (stack);
        }
        return FAILURE;
}
```

If you run through this algorithm on paper (or in a programming language such as C++ or LISP), you will find that it follows the tree in the same way as the previous algorithm, **depth**.

Table 4.2 Analysis of depth-first search of tree shown in Figure 4.5

Step	State	Queue	Notes
1	A	(empty)	The queue starts out empty, and the initial state is the root node, which is A.
2	A	B,C	The successors of A are added to the queue.
3	B	C	
4	B	D,E,C	The successors of the current state, B, are added to the front of the queue.
5	D	E,C	
6	D	H,I,E,C	
7	H	I,E,C	H has no successors, so no new nodes are added to the queue.
8	I	E,C	Similarly, I has no successors.
9	E	C	
10	E	J,K,C	
11	J	K,C	Again, J has no successors.
12	K	C	K has no successors. Now we have explored the entire branch below B, which means we back-track up to C.
13	C	(empty)	The queue is empty, but we are not at the point in the algorithm where this would mean failing because we are about to add successors of C to the queue.
14	C	F,G	
15	F	G	
16	F	L,M,G	
17	L	M,G	SUCCESS: the algorithm ends because a goal node has been located. In this case, it is the only goal node, but the algorithm does not know that and does not know how many nodes were left to explore.

As was mentioned previously, depth-first search and breadth-first search can be implemented very similarly. The following is a pseudocode of a non- recursive implementation of breadth-first search, which should be com- pared with the implementation above of depth-first search:

```
Function breadth ()
{
    queue = [];    // initialize an empty queue state =
    root_node;     // initialize the start statewhile
    (true)
{
       if is_goal (state)
              then return SUCCESS
       else add_to_back_of_queue (successors (state));
       if queue == []
              then report FAILURE;
       state = queue [0];    // state = first item in queue
       remove_first_item_from (queue);
       }
}
```

Notice that the only difference between depth and breadth is that where depth adds successor states to the front of the queue, breadth adds them to the back of the queue. So, when applied to the search tree in Figure 4.4, breadth will follow a rather different path from depth, as is shown in Table 4.3.

You will notice that in this particular case, depth-first search found the goal in two fewer steps than breadth-first search. As has been suggested, depth- first search will often find the goal quicker than breadth-first search if all leaf nodes are the same depth below the root node. However, in search trees where there is a very large subtree that does not contain a goal, breadth- first search will nearly always perform better than depth-first search.

Another important factor to note is that the queue gets much longer when using breadth-first search. For large trees, and in particular for trees with high branching factors, this can make a significant difference because the depth-first search algorithm will never require a queue longer than the maximum depth of the tree, whereas breadth-first search in the worst case will need a queue equal to the number of nodes at the level of the tree with the most nodes (eight in a tree of depth three with branching factor of two, as in Figure 4.3). Hence, we say that depth-first search is usually more memory efficient than breadth-first search.

Table 4.3 Analysis of breadth-first search of tree shown in Figure 4.4

Step	State	Queue	Notes
1	A	(empty)	The queue starts out empty, and the initial state is the root node, which is A.
2	A	B,C	The two descendents of A are added to the queue.
3	B	C	
4	B	C,D,E	The two descendents of the current state, B, are added to the back of the queue.
5	C	D,E	
6	C	D,E,F,G	
7	D	E,F,G	
8	D	E,F,G,H,I	
9	E	F,G,H,I	
10	E	F,G,H,I,J,K	
11	F	G,H,I,J,K	
12	F	G,H,I,J,K,L,M	
13	G	H,I,J,K,L,M	
14	G	H,I,J,K,L,M,N,O	
15	H	I,J,K,L,M,N,O	H has no successors, so we have nothing to add to the queue in this state, or in fact for any subsequent states.
16	I	J,K,L,M,N,O	
17	J	K,L,M,N,O	
18	K	L,M,N,O	
19	L	M,N,O	SUCCESS: A goal state has been reached.

As we have seen, however, depth-first search is neither optimal nor complete, whereas breadth-first search is both. This means that depth-first search may not find the best solution and, in fact, may not ever find a solution at all. In contrast, breadth-first search will always find the best solution.

Example: Web Spidering

An example of the importance of choosing the right search strategy can be seen in spidering the world wide web. The assumption is made that most of the web is connected, meaning that it is possible to get from one page to another by following a finite number of links, where a link connects two pages together.

Some parts of the Internet have a very high branching factor, with many pages containing hundreds of links to other pages. On average though, the branching factor is reasonably low, and so it seems that breadth-first search might be a sensible approach to spidering. In practice, however, the search tree that represents the connected part of the Internet is huge and searching it by pure breadth-first search would involve a prohibitive storage requirement. Depth-first search would also not be practical because some paths might have almost infinite depth, particularly given that some pages on the Internet are generated automatically at the time they are accessed.

Hence, Internet spiders must use a combination of approaches, with a particular emphasis placed on web pages that change frequently and pages that are considered by some metric to be "important." Another important aspect of search engines is their ability to search in parallel. We discuss this concept in more detail in Chapter 5.

Depth-First Iterative Deepening

Depth-First Iterative Deepening, or DFID (also called Iterative Deepening Search or IDS), is an exhaustive search technique that combines depth-first with breadth-first search. The DFID algorithm involves repeatedly carrying out depth-first searches on the tree, starting with a depth-first search limited to a depth of one, then a depth-first search of depth two, and so on, until a goal node is found.

This is an algorithm that appears to be somewhat wasteful in terms of the number of steps that are required to find a solution. However, it has the advantage of combining the efficiency of memory use of depth-first search with the advantage that branches of the search tree that are infinite or extremely large will not sidetrack the search.

It also shares the advantage of breadth-first search that it will always find the path that involves the fewest steps through the tree (although, as we will see, not necessarily the *best* path).

Although it appears that DFID would be an extremely inefficient way to search a tree, it turns out to be almost as efficient as depth-first or breadth- first search. This can be seen from the fact that for most trees, the majority of nodes are in the deepest level, meaning that all three approaches spend most of their time examining these nodes.

For a tree of depth d and with a branching factor of b, the total number of nodes is

1 root node

b nodes in the first layer

b^2 nodes in the second layer

…

b^n nodes in the n^{th} layer

Hence, the total number of nodes is

$$1 + b + b^2 + b^3 + \ldots + b^d$$

which is a **geometric progression** equal to

$$\frac{1 - b^{d+1}}{1 - b}$$

For example, for a tree of depth 2 with a branching factor of 2, there are

$$\frac{1 - 8}{1 - 2} = 7 \text{ nodes}$$

Using depth-first or breadth-first search, this means that the total number of nodes to be examined is seven. Using DFID, nodes must be examined more than once, resulting in the following progression:

$$(d + 1) + b(d) + b^2 (d - 1) + (d - 2) + \ldots + b^d$$

Hence, DFID has a time complexity of $O(b^d)$. it has the memory efficiency of depth-first search because it only ever needs to store information about the current path. Hence, its space complexity is $O(b^d)$.

In the case of the tree with depth of 2 and branching factor of 2, this means examining the following number of nodes:

$$(3 + 1) + 3 \times 2 + 4 \times 2 = 18$$

Hence, for a small tree, DFID is far more inefficient in time than depth-first or breadth-first search.

However, if we compare the time needed for a larger tree with depth of 4 and branching factor of 10, the tree has the following number of nodes:

$$\frac{1 - 10^5}{1 - 10} = 11{,}111 \text{ nodes}$$

DFID will examine the following number of nodes:

$$(4 + 1) + 10 \times 4 + 100 \times 3 + 1{,}000 \times 2 + 10{,}000 = 12{,}345 \text{ nodes}$$

Hence, as the tree gets larger, we see that the majority of the nodes to be examined (in this case, 10,000 out of 12,345) are in the last row, which needs to be examined only once in either case.

Like breadth-first search, DFID is optimal and complete. Because it also has good space efficiency, it is an extremely good search method to use where the search space may be very large and where the depth of the goal node is not known.

Using Heuristics for Search

Depth-first and breadth-first search were described as brute-force search methods. This is because they do not employ any special knowledge of the search trees, they are examining but simply examine every node in order until they happen upon the goal. This can be likened to the human being who is traversing a maze by running a hand along the left side of the maze wall.

In some cases, this is the best that can be done because there is no additional information available that can be used to direct the search any better.

Often, however, such information does exist and can be used. Take the example of looking for a suitable Christmas gift. Very few people would simply walk into each shop as they came across it, looking in each department in turn until they happened upon a present. Most people would go straight to the shop that they considered to be most likely to have a suitable gift. If no gift was found in that shop, they would then proceed to the shop they considered to be the next most likely to have a suitable gift.

This kind of information is called a **heuristic**, and humans use them all the time to solve all kinds of problems. Computers can also use heuristics, and in many problems, heuristics can reduce an otherwise impossible problem to a relatively simple one.

A **heuristic evaluation function** is a function that when applied to a node gives a value that represents a good estimate of the distance of the node from the goal. For two nodes m and n, and a heuristic function *f*, if $f(m) < f(n)$, then it should be the case that m is more likely to be on an **optimal path** to the goal node than n. In other words, the lower the heuristic value of a node, the more likely it is that it is on an optimal path to a goal and the more sensible it is for a search method to examine that node.

The following sections provide details of a number of search methods that use heuristics and are thus thought of as **heuristic search methods**, or **heuristically informed search methods**.

Typically, the heuristic used in search is one that provides an estimate of the distance from any given node to a goal node. This estimate may or may not be accurate, but it should at least provide better results than pure guesswork.

Informed and Uninformed Methods

A search method or heuristic is informed if it uses additional information about nodes that have not yet been explored to decide which nodes to examine next. If a method is not informed, it is **uninformed**, or **blind**. In other words, search methods that use heuristics are informed, and those that do not are blind.

Best-first search is an example of informed search, whereas breadth-first and depth-first search are uninformed or blind.

A heuristic h is said to be **more informed** than another heuristic, *j*, if h(node) ≤ j(node) for all nodes in the search space. (In fact, in order for *h* to be more informed than j, there must be some node where h(node) < j(node). Otherwise, they are as informed as each other.)

The more informed a search method is, the more efficiently it will search.

Choosing a Good Heuristic

Some heuristics are better than others, and the better (more informed) the heuristic is, the fewer nodes it needs to examine in the search tree to find a solution. Hence, like choosing the right representation, choosing the right heuristic can make a significant difference in our ability to solve a problem.

In choosing heuristics, we usually consider that a heuristic that reduces the number of nodes that need to be examined in the search tree is a good heuristic. It is also important to consider the efficiency of running the heuristic itself. In other words, if it takes an hour to compute a heuristic value for a given state, the fact that doing so saves a few minutes of total search time is irrelevant. For most of this section, we will assume that heuristic functions we choose are extremely simple to calculate and so do not impact on the overall efficiency of the search algorithm.

The 8-puzzle

To illustrate the way in which heuristics are developed, we will use the 8- puzzle, as illustrated in Figure 4.5.

The puzzle consists of a 3 × 3 grid, with the numbers 1 through 8 on tiles within the grid and one blank square. Tiles can be slid about within the grid, but a tile can only be moved into the empty square if it is adjacent to the empty square. The start state of the puzzle is a random configuration, and the goal state is as shown in the second picture in Figure 4.5, where the numbers go from 1 to 8 clockwise around the empty middle square, with 1 in the top left.

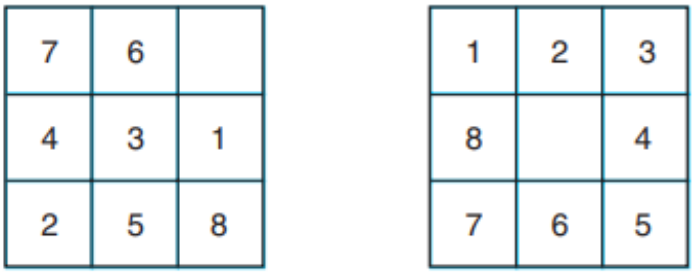

Figure 4.5
The 8-puzzle, start state and goal state

Typically, it takes about 20 moves to get from a random start state to the goal state, so the search tree has a depth of around 20. The branching factor depends on where the blank square is. If it is in the middle of the grid, the branching factor is 4; if it is on an edge, the branching factor is 3, and if it is in a corner, the branching factor is 2. Hence, the average branching factor of the search tree is 3.

So, an exhaustive search of the search tree would need to examine around 320 states, which is around 3.5 billion. Because there are only 9! or 362,880 possible states, the search tree could clearly be cut down significantly by avoiding repeated states.

It is useful to find ways to reduce the search tree further, in order to devise a way to solve the problem efficiently. A heuristic would help us to do this, by telling us approximately how many moves a given state is from the goal state. We will examine a number of possible heuristics that could be used with the 8-puzzle.

To be useful, our heuristic must never overestimate the cost of changing from a given state to the goal state. Such a heuristic is defined as being **admissible**. As we will see, in many search methods it is essential that the heuristics we use are admissible.

The first heuristic we consider is to count how many tiles are in the wrong place. We will call this heuristic, h_1(node). In the case of the first state shown in Figure 4.5, h_1 (node) = 8 because all the tiles are in the wrong place. However, this is misleading because we could imagine a state with a heuristic value of 8 but where each tile could be moved to its correct place in one move. This heuristic is clearly admissible because if a tile is in the wrong place, it must be moved at least once.

An improved heuristic, h_2, takes into account how far each tile had to move to get to its correct state. This is achieved by summing the **Manhattan distances** of each tile from its correct position. (Manhattan distance is the sum of the horizontal and vertical moves that need to be made to get from one position to another, named after the grid system of roads, used in Manhattan.)

For the first state in Figure 4.5, this heuristic would provide a value of

$$h_2 \text{ (node)} = 2 + 2 + 2 + 2 + 3 + 3 + 1 + 3 = 18$$

Clearly, this is still an admissible heuristic because in order to solve the puzzle, each tile must be moved one square at a time from where it starts to where it is in the goal state.

It is worth noting that h_2 (node) □ h_1 (node) for any node. This means that h_2 dominates h_1, which means that a search method using heuristic h_2 will always perform more efficiently than the same search method using h_1. This is because h_2 is more informed than h_1. Although a heuristic must never overestimate the cost, it is always better to choose the heuristic that gives the highest possible underestimate of cost. The ideal heuristic would thus be one that gave exactly accurate costs every time.

This efficiency is best understood in terms of the **effective branching factor**, b^*, of a search.

If a search method expands n nodes in solving a particular problem, and the goal node is at depth d, then b^* is the branching factor of a **uniform tree** that contains n nodes. Heuristics that give a lower effective branching factor perform better. A search method running with h_2 has a lower effective branching factor than the same search method running with h_1 in solving the 8-puzzle.

A third heuristic function, h_3, takes into account the fact that there is extra difficulty involved if two tiles have to move past each other because tiles cannot jump over each other. This heuristic uses a function k(node), which is equal to the number of direct swaps that need to be made between adjacent tiles to move them into the correct sequence.

$$h_3 \text{ (node)} = \text{h2 (node)} + (2 \times \text{k(node)})$$

Because k(node) must be at least 0, h_3 (node) must be greater than h_2 (node), meaning that h_3 is a more informed heuristic than h_2.

The heuristic functions h_1, h_2, and h_3 are all admissible, meaning that using the **A*** algorithm (see Section 4.16.1) with any of these heuristics would guarantee to find the quickest solution to the puzzle.

There are a number of possible ways to generate useful heuristic functions. Functions like h_1 and h_2 can be generated by **relaxing** the 8-puzzle problem. A **relaxed problem** is a version of a problem that has fewer **constraints**. For example, a relaxed version of the 8-puzzle might be that a tile can be moved to an adjacent square regardless of whether that square is empty or not. In that case, h_2 (node) would be exactly equal to the number of moves needed to get from a node to the goal node.

If the problem were relaxed further, we might say that a tile could move to any square, even if that square is not adjacent to the square it is starting from. In this case, h_1 (node) exactly equals the number of moves needed to get from a node to the goal node.

Hence, using an exact cost function for a relaxed version of a problem is often a good way to generate a heuristic cost function for the main problem.

It is clear that h_3 is the best heuristic function to use of the three we generated because it dominates both h_1 and h_2. In some cases, a number of heuristic functions may exist, none of which dominates the others. In that case, a new heuristic can be generated from the heuristics $h_1 ... h_n$, as follows:

$$h(\text{node}) = \max (h_1 [\text{node}], h_2 [\text{node}], ..., h_n [\text{node}])$$

Because all of h_1 to h_n is admissible, h(node) must also be admissible. The heuristic function h dominates all of the heuristics $h_1 ... h_n$ and so is clearly the best one to use.

As we see in Chapter 6, another way to find a heuristic is to take advantage of features of the problem that is being modeled by the search tree. For example, in the case of playing checkers, computers are able to use heuristics such as the fact that a player with more kings on the board is likely to win against a player with fewer kings.

Monotonicity

A search method is described as **monotone** if it always reaches a given node by the shortest possible path.

So, a search method that reaches a given node at different depths in the search tree is not monotone. A monotone search method must be admissible, provided there is only one goal state.

A **monotonic** heuristic is a heuristic that has this property.

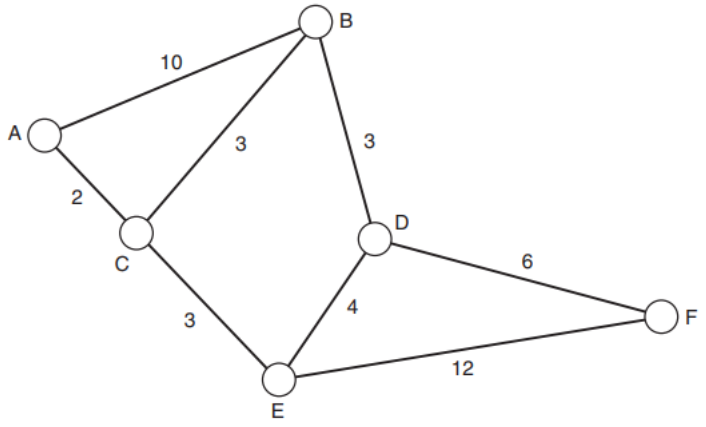

Figure 4.6
Map of five cities

An **admissible heuristic** is a heuristic that never overestimates the true distance of a node from the goal. A monotonic heuristic is also admissible, assuming there is only one goal state.

Example: The Modified Traveling Salesman Problem

It is usual when examining heuristic search methods to relate the search problem to a real-world situation in order to derive suitable heuristics. For this explanation, we will use the example of finding the best route between two cities, a variation of the **Traveling Salesman problem**, as shown in Figure 4.6.

In this diagram, each node represents a town, and the vertices between nodes represent roads that join towns together. A is the starting node, and F is the goal node. Each vertex is labeled with a distance, which shows how long that road is. Clearly the diagram is not drawn to scale.

The aim of this problem is to find the shortest possible path from city A to city F. This is different from the traditional Traveling Salesman problem, in which the problem is to find a way to travel around a group of cities and finally arrive back at the starting city.

We can represent the search space of the map in Figure 4.6 as a search tree by showing each possible path as a leaf node in the tree. In doing so, we need to be careful to remove repetitions of paths, or loops, because those would add redundancy to the graph and make searching it inefficient. The tree for this search space is shown in Figure 4.7.

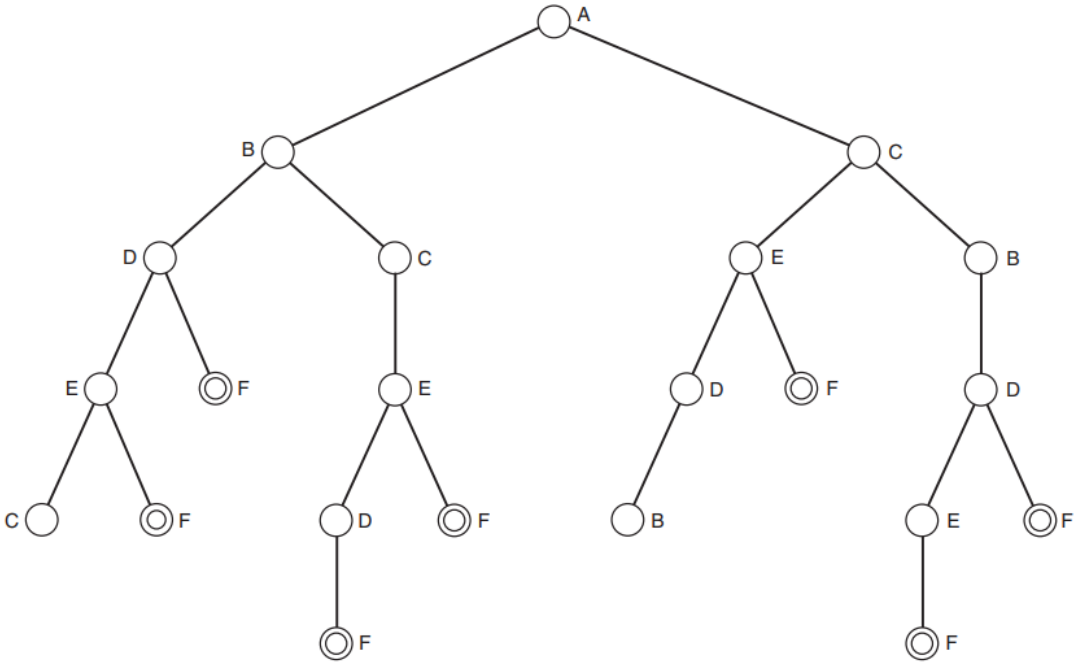

Figure 4.7
Search tree for map in Figure 4.6

You will notice that this tree has nine leaf nodes, seven of which are goal nodes. Two of the paths lead to cyclical paths and so are abandoned. There are seven distinct paths that successfully lead from A to F. These seven paths can be traced from the tree as follows:

1 A, B, D, E, F
2 A, B, D, F
3 A, B, C, E, D, F
4 A, B, C, E, F
5 A, C, E, F
6 A, C, B, D, E, F
7 A, C, B, D, F

The two cyclical paths are as follows:

1 A, B, D, E, C (which would then lead on to A or B)
2 A, C, E, D, B (which would then lead on to A or C)

A depth-first approach to this problem would provide path number 1, A, B, D, E, F, which has a total distance of 29.

Breadth-first search always produces the path that has the least steps, but not necessarily the shortest path. In this case, it would yield path 2, which is A, B, D, F and which has a length of 19. This is much shorter than the path produced by depth-first search but is not the shortest path (the shortest path is path 5, A, C, E, F, which has a length of 17).

Now we introduce two new search methods that use heuristics to more efficiently identify search solutions.

Hill Climbing

Hill climbing is an example of an **informed** search method because it uses information about the search space to search in a reasonably efficient manner. If you try to climb a mountain in fog with an altimeter but no map, you might use the hill climbing Generate and Test approach:

Check the height 1 foot away from your current location in each direction: north, south, east, and west.

As soon as you find a position where the height is higher than your current position, move to that location and restart the algorithm.
If all directions lead lower than your current position, then you stop and assume you have reached the summit. As we see later, this might not necessarily always be true.

In examining a search tree, hill climbing will move to the first successor node that is "better" than the current node—in other words, the first node that it comes across with a heuristic value lower than that of the current node.

Steepest Ascent Hill Climbing

Steepest ascent hill climbing is similar to hill climbing, except that rather than moving to the first position you find that is higher than the current position, you always check around you in all four directions and choose the position that is highest.

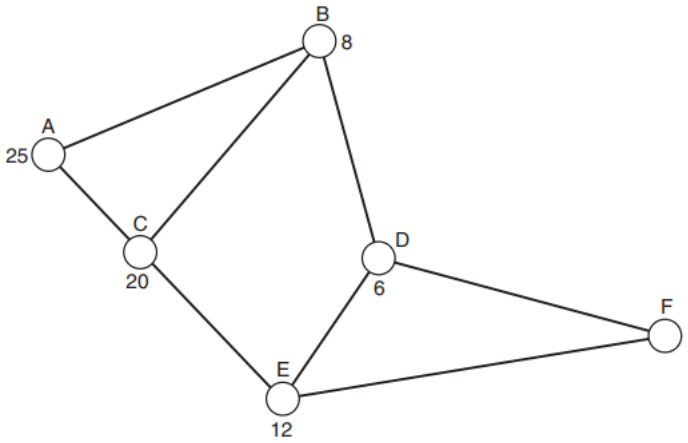

Figure 4.8
The map of five cities where the straight-line distance from each city to the goal city (F) is shown

Steepest ascent hill climbing can also be thought of as a variation on depth- first search, which uses information about how far each node is from the goal node to choose which path to follow next at any given point.

For this method, we apply a heuristic to the search tree shown in Figure 4.7, which is the straight-line distance from each town to the goal town. We are using this heuristic to approximate the actual distance from each town to the goal, which will of course be longer than the straight-line distance.

In Figure 4.8, we can see the same search problem as presented in Figure 4.6, but instead of noting the lengths of vertices, we note how far each city is (using a straight-line measurement) from the goal, city F.

Now hill climbing proceeds as with depth-first search, but at each step, the new nodes to be added to the queue are sorted into order of distance from the goal. Note that the only difference between this implementation and that given for depth-first search is that in hill climbing the successors of state are sorted according to their distance from the goal before being added to the queue:

```
Function hill ()
{
        queue = [];     // initialize an empty queue
        state = root_node;      // initialize the start state
        while (true)
        {
                if is_goal (state)
                        then return SUCCESS
                else
                {
                        sort (successors (state));
                        add_to_front_of_queue (successors (state));
                }
                if queue == []
                        then report FAILURE;
                state = queue [0]; // state = first item in queue
                remove_first_item_from (queue);
        }
}
```

This algorithm thus searches the tree in a depth-first manner, at each step choosing paths that appear to be most likely to lead to the goal.

The steps taken by a hill-climbing algorithm in solving the preceding problem are shown in Table 4.4:

Table 4.4 Analysis of hill climbing

Step	State	Queue	Notes
1	A	(empty)	The queue starts out empty, and the initial state is the root node, which is A.
2	A	B,C	The successors of A are sorted and placed on the queue. B is placed before C on the queue because it is closer to the goal state, F.
3	B	C	
4	B	D,C,C	
5	D	C,C	
6	D	F,E,C,C	F is placed first on the queue because it is closest to the goal. In fact, it is the goal, as will be discovered in the next step.
7	F	E,C,C	SUCCESS: Path is reported as A,B,D,F.

In this case, hill climbing has produced the same path as breadth-first search, which is the path with the least steps, but not the shortest path. In many cases though, using this heuristic enables hill climbing to identify shorter paths than would be identified by depth-first or breadth-first search. Hill climbing uses heuristics to identify paths efficiently but does not necessarily identify the best path.

If we ran the searches from right to left, instead of from left to right (or ordered the search tree the other way around), then we would find that breadth-first search would produce a different path: A, C, E, F (which is in fact the shortest path), but hill climbing would still produce the same path, A, B, D, F. In other words, the particular ordering of nodes used affects which result is produced by breadth-first and depth-first search but does not affect hill climbing in the same way. This can clearly be a useful property.

Foothills, Plateaus, and Ridges

Although we have been talking about using search techniques to traverse search trees, they can also be used to solve search problems that are represented in different ways. In particular, we often represent a search problem as a three-dimensional space, where the x- and y-axes are used to represent variables and the z-axis (or height) is used to rep- resent the outcome.

The goal is usually to maximize the outcome, and so search methods in these cases are aiming to find the highest point in the space.

Many such search spaces can be successfully traversed using hill climbing and other heuristically informed search methods. Some search spaces, however, will present particular difficulties for these techniques.

In particular, hill climbing can be fooled by **foothills**, **plateaus**, and **ridges**. Figure 4.9 has three illustrations, showing foothills, a plateau, and a ridge. This figure shows the search space represented as a three-dimensional terrain. In this kind of terrain, the aim of search is to find the x and y values that give the highest possible value of z—in other words, the highest point in the terrain. This is another way of looking at traditional search: search is normally aiming to maximize some function, which in this case is shown as the height of the terrain but is traditionally a function that details the distance of a node from the goal node.

Foothills are often called **local maxima** by mathematicians. A local maxi- mum is a part of the search space that appears to be preferable to the parts around it, but which is in fact just a foothill of a larger hill. Hill-climbing techniques will reach this peak, where a more sophisticated technique might move on from there to look for the **global maximum**. Figure 4.9 (a) shows a search space that has a single global maximum surrounded by a number of foothills, or local maxima. Many search methods would reach the top of one of these foothills and, because there was nowhere higher nearby, would conclude that this was the best solution to the problem.

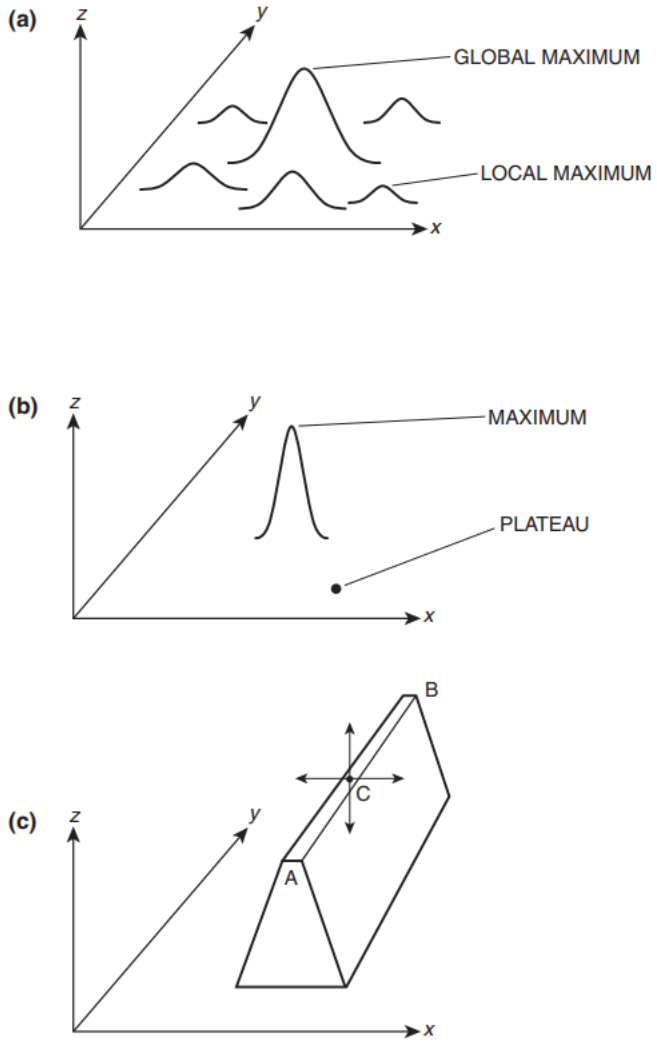

Figure 4.9
(a) FOOTHILLS
(b) PLATEAU
(c) RIDGE

Later in this chapter and in Chapter 5, we see methods such as simulated annealing that are good at avoiding being trapped by local maxima.

A plateau is a region in a search space where all the values are the same. In this case, although there may well be a suitable maximum value somewhere nearby, there is no indication from the local terrain of which direction to go to find it. Hill climbing does not perform well in this situation. Figure 4.9 (b) shows a search space that consists of just one peak surrounded by a plateau. A hill-climbing search method could well find itself stuck in the plateau with no clear indication of where to go to find a good solution.

The final problem for hill climbing is presented by ridges. A ridge is a long, thin region of high land with low land on either side. When looking in one of the four directions, north, south, east, and west from the ridge, a hill- climbing algorithm would determine that any point on the top of the ridge was a maximum because the hill falls away in those four directions. The correct direction is a very narrow one that leads up the top of the ridge but identifying this direction using hill climbing could be very tricky.

Figure 4.9 (c) shows a ridge. The point marked A is lower than the point marked B, which is the global maximum. When a hill-climbing method finds itself at point C, it might find it hard to get from there to B. The arrows on point C show that in moving north, south, east, or west, the method would find itself at a lower point. The correct direction is up the ridge.

Best-First Search

Best-first search employs a heuristic in a similar manner to hill climbing. The difference is that with best-first search, the entire queue is sorted after new paths have been added to it, rather than adding a set of sorted paths.

In practical terms, this means that best-first search follows the best path available from the current (partially developed) tree, rather than always following a depth-first style approach.

```
Function best ()
{
        queue = [];     // initialize an empty queue
        state = root_node;      // initialize the start state
        while (true)
        {
                if is_goal (state)
                        then return SUCCESS
                else
        {
                add_to_front_of_queue (successors (state));
                sort (queue);
        }
        if queue == []
                then report FAILURE;
        state = queue [0]; // state = first item in queue
        remove_first_item_from (queue);
        }
}
```

The path taken through the search tree shown in Figure 4.7 is shown in Table 4.5.

Table 4.5 Analysis of best-first search of tree shown in Figure 4.4

Step	State	Queue	Notes
1	A	(empty)	The queue starts out empty, and the initial state is the root node, which is A.
2	A	B,C	The successors of the current state, B and C, are placed in the queue.
	A	B,C	The queue is sorted, leaving B in front of C because it is closer to the goal state, F.
3	B	C	
4	B	D,C,C	The children of node B are added to the front of the queue.
5	B	D,C,C	The queue is sorted, leaving D at the front because it is closer to the goal node than C.
6	D	C,C	Note that although the queue appears to contain the same node twice, this is just an artifact of the way the search tree was constructed. In fact, those two nodes are distinct and represent different paths on our search tree.
7	D	E,F,C,C	The children of D are added to the front of the queue.
8	D	F,E,C,C	The queue is sorted, moving F to the front.
9	F	E,C,C	SUCCESS: Path is reported as A,B,D,F.

It can be seen that, in this case, best-first search happens to produce the same path as hill climbing and breadth-first search, although the queue is ordered differently during the process. As with hill climbing, best-first search will tend to provide a shorter path than depth first or breadth first, but not necessarily the shortest path.

Beam Search

Beam search is a form of breadth-first search that employs a heuristic, as seen with hill climbing and best-first search. Beam search works using a threshold so that only the best few paths are followed downward at each level. This method is very efficient in memory usage and would be particularly useful for exploring a search space that had a very high branching factor (such as in game trees for games, such as Go or Chess). It has the disadvantage of not exhaustively searching the entire tree and so may fail to ever find a goal node. In this implementation, the function call `select_best_paths (queue, n)` removes all but the best n paths from the queue.

```
Function beam ()
{
        queue = [];     // initialize an empty queue
        state = root_node;      // initialize the start state
        while (true)
        {
                if is_goal (state)
                        then return SUCCESS
                else
                {
                        add_to_back_of_queue (successors (state));
                        select_best_paths (queue, n);
                }
                if queue == []
                        then report FAILURE;
                state = queue [0]; // state = first item in queue
                remove_first_item_from (queue);
        }
}
```

In this pseudocode, *n* is used to represent the width threshold, which is set at the beginning of the procedure.

Table 4.6 Analysis of beam search of tree shown in Figure 4.7

Step	State	Queue	Notes
1	A	(empty)	The queue starts out empty, and the initial state is the root node, which is A.
2	A	B,C	The two children of the current node are added to the back of the queue.
3	B	C	
4	B	C,D,C	The two children of B are added to the back of the queue.
5	B	D,C	All but the two best paths are discarded from the queue.
6	D	C	
7	D	C,E,F	The two children of the current node are added to the back of the queue.
8	D	E,F	At this step, C is removed from the queue because we only require the two best paths.
9	E	F	
10	E	F,C,F	The two children of E are added to the back of the queue.
11	E	F,F	The path that leads to C is discarded, in favor of the two better paths, both of which lead to F.
12	F	F	SUCCESS: Path is reported as A,B,D,E,F.

The interesting aspect of this method is the choice of how to define the “best” paths to include in the queue. Often, the path that involves the fewest steps is used or the path that has reached the point with the highest heuristic value (in other words, the path that got closest to the goal).

In Table 4.6, the value of state and queue are shown for the problem tree shown in Figure 4.7, using beam search with a threshold of 2 (in other words, only two paths are extended down from each level). For this implementation, we have used the heuristic value of each node to determine which path is the “best” path. So, the “best” path will be the one that has reached the closest to a goal node so far.

Identifying Optimal Paths

Several methods exist that do identify the **optimal path** through a search tree. The optimal path is the one that has the lowest **cost** or involves traveling the shortest distance from start to goal node. The techniques described previously may find the optimal path by accident, but none of them are guaranteed to find it.

The simplest method for identifying the optimal path is called the **British Museum procedure**. This process involves examining every single path through the search tree and returning via the best path that was found. Because every path is examined, the optimal path must be found. This process is implemented as an extension of one of the exhaustive search techniques, such as depth-first or breadth-first search, but rather than stop- ping when a solution is found, the solution is stored, and the process continues until all paths have been explored. If an alternative solution is found, its path is compared with the stored path, and if it has a lower cost, it replaces the stored path.

The following more sophisticated techniques for identifying optimal paths are outlined in this section:

- A*
- uniform cost search (Branch and Bound)
- greedy search

The British Museum procedure also has the property that it generates all solutions. Most of the search methods we look at in this book stop when they find a solution. In some cases, this will be the best solution, and in other cases it may even be the worst available solution (depth-first search will do this if the worst solution happens to be the left-most solution).

In some cases, it may be necessary to identify all possible solutions, in which case something like the British Museum procedure would be useful.

Assuming that none of the branches of the tree is infinitely deep, and that no level has an infinite branching factor, then it does not matter which approach is used (depth first or breadth first, for example) when running the British Museum procedure: because the goal is to visit every node, the order the nodes are visited probably does not matter.

A Algorithms*

A* algorithms are similar to best-first search but use a somewhat more complex heuristic to select a path through the tree. The best-first algorithm always extends paths that involve moving to the node that appears to be closest to the goal, but it does not take into account the cost of the path to that node so far.
The A* algorithm operates in the same manner as best-first search but uses the following function to evaluate nodes:

$$f(node) = g(node) + h(node)$$

g(node) is the cost of the path so far leading up to the node, and h(node) is an underestimate of the distance of the node from a goal state; f is called a **path-based evaluation function**. When operating A*, f(node) is evaluated for successor nodes and paths extended using the nodes that have the *lowest* values of f.

If h(node) is always an *underestimate* of the distance of a node to a goal node, then the A* algorithm is optimal: it is *guaranteed* to find the shortest path to a goal state. A* is described as being optimally efficient, in that in finding the path to the goal node, it will expand the fewest possible paths. Again, this property depends on h(node) always being an underestimate.

Note that running the A* algorithm on the search tree shown in Figure 4.4 would not be guaranteed to find the shortest solution because the estimated values for h(node) are not all underestimates. In other words, the heuristic that is being used is not admissible. If a nonadmissible heuristic for h(node) is used, then the algorithm is called **A**.

A* is the name given to the algorithm where the h(node) function is admissible. In other words, it is guaranteed to provide an underestimate of the true cost to the goal.

A* is optimal and complete. In other words, it is guaranteed to find a solution, and that solution is guaranteed to be the best solution.

A* is in fact only complete if the tree it is searching has a finite branching factor and does not contain a path of finite cost, which has an infinite number of nodes along it. Both of these conditions are likely to be met in all real-world situations, and so for simplicity we can state that A* is complete; although, to be more accurate:

A* is complete if the graph it is searching is **locally finite** (that is, it has a finite branching factor) and if every arc between two nodes in the graph has a non-zero cost.

That A* is optimal can be proved by considering a counter-example:

Imagine we are applying the A* algorithm to a graph with two goals, G1 and G2. The path cost of G1 is f1 and the path cost of G2 is f2, where f2 > f1. G1 is the goal with the lower cost but let us imagine a scenario where the A* algorithm has reached G2 without having explored G1. In other words, we are imagining a scenario where the algorithm has not chosen the goal with the lesser cost.

If we consider a node, n, that is on an optimal path from the root node to G1, then because h is an admissible heuristic:

$$f1 \geq f(n)$$

The only reason the algorithm would not choose to expand n before it reaches G2 would be if

$$f(n) > f(G2)$$

Hence, by combining these two expressions together, we arrive at

$$f1 \geq f(G2)$$

Because G2 is a goal state, it must be the case that h(G2) = 0, and thus f(G2) = g(G2). Thus, we have

$$f1 \geq g(G2)$$

This, therefore, contradicts our original assumption that G2 had a higher path cost than G1, which proves that A* can only ever choose the least cost path to a goal.

It was mentioned that A* is similar to breadth-first search. In fact, breadth- first search can be considered to be a special case of A*, where h(node) is always 0, so f(node) = g(node), and where every direct path between a node and its immediate successor has a cost of 1.

Uniform Cost Search

Uniform cost search (or **Branch and Bound**) is a variation on best-first search that uses the evaluation function g(node), which for a given node evaluates to the cost of the path leading to that node. In other words, this is an A* algorithm but where h(node) is set to zero. At each stage, the path that has the lowest cost so far is extended. In this way, the path that is generated is likely to be the path with the lowest overall cost, but this is not guaranteed. To find the best path, the algorithm needs to continue running after a solution is found, and if a preferable solution is found, it should be accepted in place of the earlier solution.

Uniform cost search is complete and is optimal, providing the cost of a path increases monotonically. In other words, if for every node m that has a successor n, it is true that $g(m) < g(n)$, then uniform cost is optimal. If it is possible for the cost of a node to be less than the cost of its parent, then uniform cost search may not find the best path.

Uniform cost search was invented by Dijkstra in 1959 and is also known as Dijkstra's algorithm.

Greedy Search

Greedy search is a variation of the A* algorithm, where g(node) is set to zero, so that only h(node) is used to evaluate suitable paths. In this way, the algorithm always selects the path that has the lowest heuristic value or estimated distance (or cost) to the goal.

Greedy search is an example of a best-first strategy.

Greedy-search methods tend to be reasonably efficient, although in the worst case, like depth-first search, it may never find a solution at all. Additionally, greedy search is not optimal and can be fooled into following extremely costly paths. This can happen if the first step on the shortest path toward the goal is longer than the first step along another path, as is shown in Figure 4.10.

Example: The Knapsack Problem

The knapsack problem is an interesting illustration of the use of greedy- search algorithms and their pitfalls. The **fractional knapsack problem** can be expressed as follows:

A man is packing items into his knapsack. He wants to take the most valuable items he can, but there is a limit on how much weight he can fit in his knapsack. Each item has a weight w_i and is worth v_i. He can only fit a total weight of W in his knapsack. The items that he wants to take are things that can be broken up and still retain their value (like flour or milk), and he is able to take fractions of items. Hence, the problem is called the *fractional* knapsack problem.

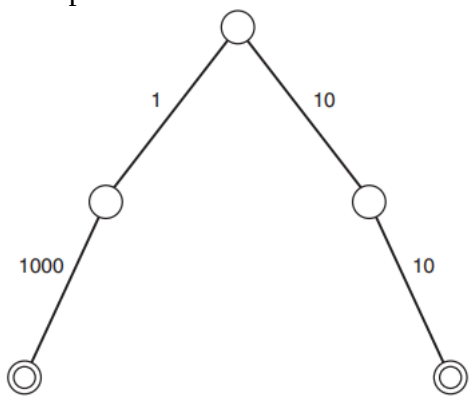

Figure 4.10
A search tree where a greedy-search method will not find the best solution

In solving this problem, a greedy-search algorithm provides the best solution.

The problem is solved by calculating the value per unit weight of each item: v_i/w_i, and then taking as much as he can carry of the item with the greatest value per unit weight. If he still has room, he moves on to the item with the next highest value per unit weight, and so on.

The **0-1 knapsack** problem is the same as the fractional knapsack problem, except that he cannot take parts of items. Each item is thus something like a television set or a laptop computer, which must be taken whole. In solving this problem, a greedy-search approach does not work, as can be seen from the following example:

Our man has a knapsack that lets him carry a total of 100 pounds. His items are:

1 gold brick worth $1800 and weighing 50 pounds
1 platinum brick worth $1500 and weighing 30 pounds
1 laptop computer worth $2000 and weighing 50 pounds

Hence, we have four items, whose values of v and w are as follows:

$v_1 = 1800$	$w_1 = 50$	$v_1/w_1 = 36$
$v_2 = 1500$	$w_2 = 30$	$v_2/w_2 = 50$
$v_3 = 2000$	$w_3 = 50$	$v_3/w_3 = 40$

In this case, a greedy-search strategy would pick item 2 first, and then would take item 3, giving a total weight of 80 pounds, and a total value of $3500. In fact, the best solution is to take items 1 and 3 and to leave item 2 behind giving a total weight of 100 pounds and a total value of $3800.

Chapter Summary

- ✓ Generate and Test is an extremely simple example of a brute-force or exhaustive search technique.
- ✓ Depth-first search and breadth-first search are extremely commonly used and well understood exhaustive search methods.
- ✓ In analyzing search methods, it is important to examine the complexity (in time and space) of the method.
- ✓ A search method is complete if it will always find a solution if one exists. A search method is optimal (or admissible) if it always finds the best solution that exists.
- ✓ Depth-First Iterative Deepening (DFID) is a search method that has the low memory requirements of depth-first search and is optimal and complete, like breadth-first search.
- ✓ Heuristics can be used to make search methods more informed about the problem they are solving. A heuristic is a method that provides a better guess about the correct choice to make at any junction that would be achieved by random guessing.
- ✓ One heuristic is more informed than another heuristic if a search method that uses it needs to examine fewer nodes to reach a goal.
- ✓ Relaxing problems is one way to identify potentially useful heuristics.
- ✓ Hill climbing is a heuristic search method that involves continually moving from one potential solution to a better potential solution until no better solution can be found.
- ✓ Hill climbing has problems in search spaces that have foothills, plateaus, and ridges.
- ✓ A* is a heuristic search method that in most situations is optimal and complete. It uses the path evaluation function to choose suit- able paths through the search space.