4 INFORMED SEARCH METHODS

In which we see how information about the state space can prevent algorithms from blundering about in the dark.

Chapter 3 showed that uninformed search strategies can find solutions to problems by systematically generating new states and testing them against the goal. Unfortunately, these strategies are incredibly inefficient in most cases. This chapter shows how an informed search strategy—one that uses problem-specific knowledge—can find solutions more efficiently. It also shows how optimization problems can be solved.

4.1 BEST-FIRST SEARCH

In Chapter 3, we found several ways to apply knowledge to the process of formulating a problem in terms of states and operators. Once we are given a well-defined problem, however, our options are more limited. If we plan to use the GENERAL-SEARCH algorithm from Chapter 3, then the only place where knowledge can be applied is in the queuing function, which determines the node to expand next. Usually, the knowledge to make this determination is provided by an

EVALUATION FUNCTION

evaluation function that returns a number purporting to describe the desirability (or lack thereof) of expanding the node. When the nodes are ordered so that the one with the best evaluation is

BEST-FIRST SEARCH

expanded first, the resulting strategy is called **best-first search.** It can be implemented directly with GENERAL-SEARCH, as shown in Figure 4.1.

The name "best-first search" is a venerable but inaccurate one. After all, if we could really expand the best node first, it would not be a search at all; it would be a straight march to the goal. All we can do is choose the node that *appears* to be best according to the evaluation function. If the evaluation function is omniscient, then this will indeed be the best node; in reality, the evaluation function will sometimes be off, and can lead the search astray. Nevertheless, we will stick with the name "best-first search," because "seemingly-best-first search" is a little awkward.

Just as there is a whole family of GENERAL-SEARCH algorithms with different ordering functions, there is also a whole family of BEST-FIRST-SEARCH algorithms with different evaluation

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
  inputs: problem, a problem
          Eval-Fn, an evaluation function

  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(problem, Queueing-Fn)
```

Figure 4.1 An implementation of best-first search using the general search algorithm.

functions. Because they aim to find low-cost solutions, these algorithms typically use some estimated measure of the cost of the solution and try to minimize it. We have already seen one such measure: the use of the path cost g to decide which path to extend. This measure, however, does not direct search *toward the goal. In order to focus the search, the measure must incorporate some estimate of the cost of the path from a state to the closest goal state.* We look at two basic approaches. The first tries to expand the node closest to the goal. The second tries to expand the node on the least-cost solution path.

Minimize estimated cost to reach a goal: Greedy search

One of the simplest best-first search strategies is to minimize the estimated cost to reach the goal. That is, the node whose state is judged to be closest to the goal state is always expanded first. For most problems, the cost of reaching the goal from a particular state can be estimated but cannot be determined exactly. A function that calculates such cost estimates is called a **heuristic function,** and is usually denoted by the letter h:

HEURISTIC FUNCTION

$h(n) =$ estimated cost of the cheapest path from the state at node n to a goal state.

GREEDY SEARCH

A best-first search that uses h to select the next node to expand is called **greedy search,** for reasons that will become clear. Given a heuristic function h, the code for greedy search is just the following:

```
function GREEDY-SEARCH(problem) returns a solution or failure
  return BEST-FIRST-SEARCH(problem, h)
```

Formally speaking, h can be any function at all. We will require only that $h(n) = 0$ if n is a goal.

To get an idea of what a heuristic function looks like, we need to choose a particular problem, because heuristic functions are problem-specific. Let us return to the route-finding problem from Arad to Bucharest. The map for that problem is repeated in Figure 4.2.

STRAIGHT-LINE DISTANCE

A good heuristic function for route-finding problems like this is the **straight-line distance** to the goal. That is,

$h_{SLD}(n) =$ straight-line distance between n and the goal location.

HISTORY OF "HEURISTIC"

By now the space aliens had mastered my own language, but they still made simple mistakes like using "hermeneutic" when they meant "heuristic."
— a Louisiana factory worker in Woody Alien's *The UFO Menace*

The word "heuristic" is derived from the Greek verb *heuriskein*, meaning "to find" or "to discover." Archimedes is said to have run naked down the street shouting "*Heureka*" (I have found it) after discovering the principle of flotation in his bath. Later generations converted this to Eureka.

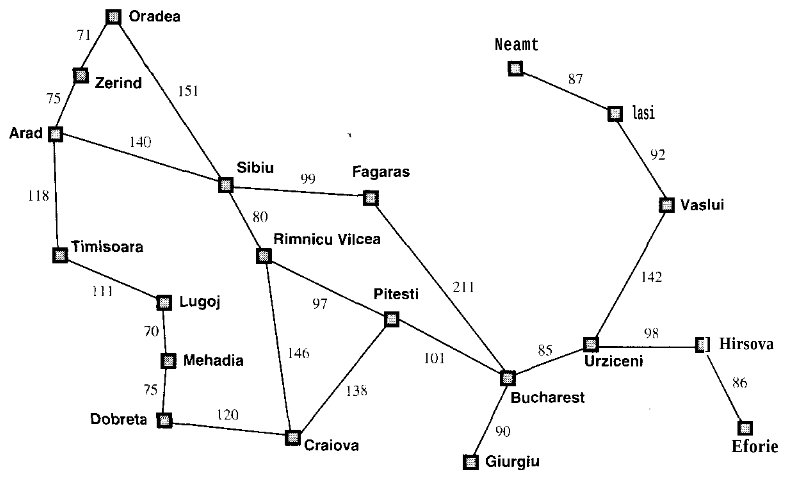
The technical meaning of "heuristic" has undergone several changes in the history of AI. In 1957, George Polya wrote an influential book called *How to Solve It* that used "heuristic" to refer to the study of methods for discovering and inventing problem-solving techniques, particularly for the problem of coming up with mathematical proofs. Such methods had often been deemed not amenable to explication.

Some people use heuristic as the opposite of algorithmic. For example, Newell, Shaw, and Simon stated in 1963, "A process that may solve a given problem, but offers no guarantees of doing so, is called a heuristic for that problem." But note that there is nothing random or nondeterministic about a heuristic search algorithm: it proceeds by algorithmic steps toward its result. In some cases, there is no guarantee how long the search will take, and in some cases, the quality of the solution is not guaranteed either. Nonetheless, it is important to distinguish between "nonalgorithmic" and "not precisely characterizable."

Heuristic techniques dominated early applications of artificial intelligence. The first "expert systems" laboratory, started by Ed Feigenbaum, Brace Buchanan, and Joshua Lederberg at Stanford University, was called the Heuristic Programming Project (HPP). Heuristics were viewed as "rules of thumb" that domain experts could use to generate good solutions without exhaustive search. Heuristics were initially incorporated directly into the structure of programs, but this proved too inflexible when a large number of heuristics were needed. Gradually, systems were designed that could accept heuristic information expressed as "rules," and rule-based systems were born.

Currently, heuristic is most often used as an adjective, referring to any technique that improves the average-case performance on a problem-solving task, but does not necessarily improve the worst-case performance. In the specific area of search algorithms, it refers to a function that provides an estimate of solution cost.

A good article on heuristics (and one on hermeneutics!) appears in the *Encyclopedia of AI* (Shapiro, 1992).



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Figure 4.2 Map of Romania with road distances in km, and straight-line distances to Bucharest.

Notice that we can only calculate the values of h_{SLD} if we know the map coordinates of the cities in Romania. Furthermore, h_{SLD} is only useful because a road from A to B usually tends to head in more or less the right direction. This is the sort of extra information that allows heuristics to help in reducing search cost.

Figure 4.3 shows the progress of a greedy search to find a path from Arad to Bucharest. With the straight-line-distance heuristic, the first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras, because it is closest. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, the heuristic leads to minimal search cost: it finds a solution without ever expanding a node that is not on the solution path. However, it is not perfectly optimal: the path it found via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Rimnicu Vilcea and Pitesti. This path was not found because Fagaras is closer to Bucharest in straight-line distance than Rimnicu Vilcea, so it was expanded first. The strategy prefers to take the biggest bite possible out of the remaining cost to reach the goal, without worrying about whether this will be best in the long run—hence the name "greedy search." Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. They tend to find solutions quickly, although as shown in this example, they do not always find the optimal solutions: that would take a more careful analysis of the long-term options, not just the immediate best choice.

Greedy search is susceptible to false starts. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first, but it is a dead end. The solution is to go first to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras. Hence, in this case, the heuristic causes unnecessary nodes to be expanded. Furthermore, if we are not careful to detect repeated states, the solution will never be found—the search will oscillate between Neamt and Iasi.

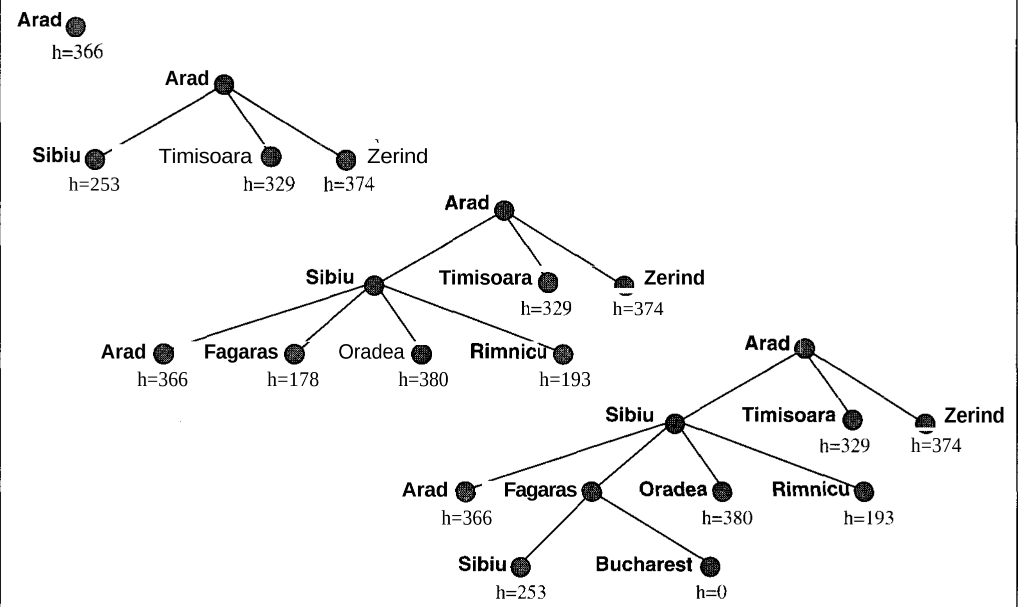


Figure 4.3 Stages in a greedy search for Bucharest, using the straight-line distance to Bucharest as the heuristic function h_{SLD}. Nodes are labelled with their h-values.

Greedy search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end. It suffers from the same defects as depth-first search—it is not optimal, and it is incomplete because it can start down an infinite path and never return to try other possibilities. The worst-case time complexity for greedy search is $O(b^m)$, where m is the maximum depth of the search space. Because greedy search retains all nodes in memory, its space complexity is the same as its time complexity. With a good heuristic function, the space and time complexity can be reduced substantially. The amount of the reduction depends on the particular problem and quality of the h function.

Minimizing the total path cost: A* search

Greedy search minimizes the estimated cost to the goal, $h(n)$, and thereby cuts the search cost considerably. Unfortunately, it is neither optimal nor complete. Uniform-cost search, on the other hand, minimizes the cost of the path so far, $g(n)$; it is optimal and complete, but can be very inefficient. It would be nice if we could combine these two strategies to get the advantages of both. Fortunately, we can do exactly that, combining the two evaluation functions simply by summing them:

$$f(n) = g(n) + h(n).$$

Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have

$$f(n) = \text{ estimated cost of the cheapest solution through } n$$

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of f. The pleasant thing about this strategy is that it is more than just reasonable. We can actually prove that it is complete and optimal, given a simple restriction on the h function.

ADMISSIBLE HEURISTIC

The restriction is to choose an h function that *never overestimates* the cost to reach the goal. Such an h is called an **admissible heuristic.** Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is. This optimism transfers to the f function as well: *If h is admissible, $f(n)$ never overestimates the actual cost of the best solution through n.* Best-first search using f as the evaluation function and an admissible h function is known as A* **search**

A* SEARCH

```
function A*-SEARCH(problem) returns a solution or failure
    return BEST-FIRST-SEARCH(problem, $g + h$)
```

Perhaps the most obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line. In Figure 4.4, we show the first few steps of an A* search for Bucharest using the h_{SLD} heuristic. Notice that the A* search prefers to expand from Rimnicu Vilcea rather than from Fagaras. Even though Fagaras is closer to Bucharest, the path taken to get to Fagaras is not as *efficient* in getting close to Bucharest as the path taken to get to Rimnicu. The reader may wish to continue this example to see what happens next.

The behavior of A* search

Before we prove the completeness and optimality of A*, it will be useful to present an intuitive picture of how it works. A picture is not a substitute for a proof, but it is often easier to remember and can be used to generate the proof on demand. First, a preliminary observation: if you examine the search trees in Figure 4.4, you will notice an interesting phenomenon. Along any path from the root, the f-cost never decreases. This is no accident. It holds true for almost all admissible heuristics. A heuristic for which it holds is said to exhibit **monotonicity**.[1]

MONOTONICITY

If the heuristic is one of those odd ones that is not monotonic, it turns out we can make a minor correction that restores monotonicity. Let us consider two nodes n and n', where n is the parent of n'. Now suppose, for example, that $g(n) = 3$ and $h(n) = 4$. Then $f(n) = g(n)+h(n)$- 7—that is, we know that the true cost of a solution path through n is at least 7. Suppose also that $g(n') = 4$ and $h(n')$- 2, so that $f(n') = 6$. Clearly, this is an example of a nonmonotonic heuristic. Fortunately, from the fact that *any path through n' is also a path through* n, we can see that the value of 6 is meaningless, because we already know the true cost is at least 7. Thus, we should

[1] It can be proved (Pearl, 1984) that a heuristic is monotonic if and only if it obeys the triangle inequality. The triangle inequality says that two sides of a triangle cannot add up to less than the third side (see Exercise 4.7). Of course, straight-line distance obeys the triangle inequality and is therefore monotonic.

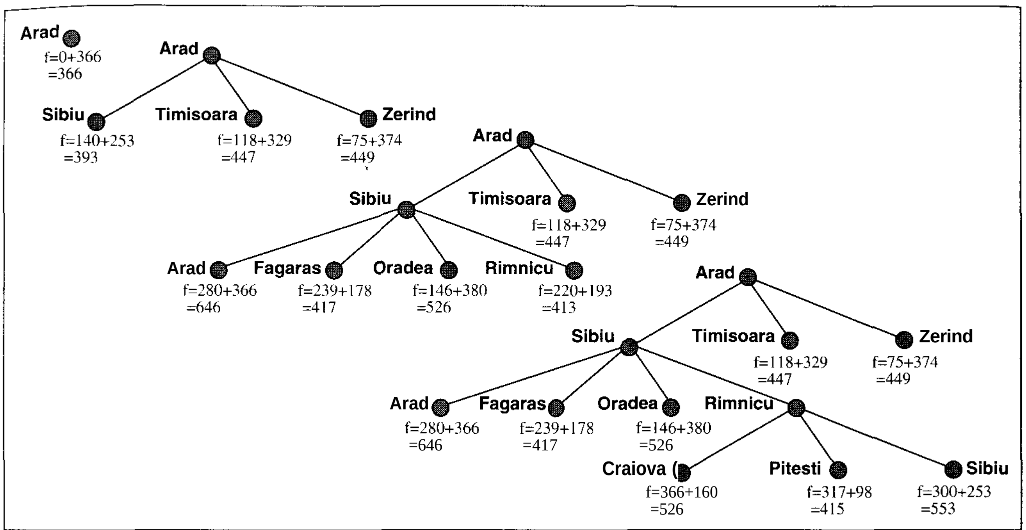


Figure 4.4 Stages in an A* search for Bucharest. Nodes are labelled with / = g + *h*. The *h* values are the straight-line distances to Bucharest taken from Figure 4.1.

check, each time we generate a new node, to see if its/-cost is less than its parent's f-cost; if it is, we use the parent's f-cost instead:

$$f(n') = max(f(n)g(n') + h(n')).$$

In this way, we ignore the misleading values that may occur with a nonmonotonic heuristic. This equation is called the **pathmax** equation. If we use it, then f will always be nondecreasing along any path from the root, provided *h* is admissible.

PATHMAX

The purpose of making this observation is to legitimize a certain picture of what A* does. If f never decreases along any path out from the root, we can conceptually draw **contours** in the state space. Figure 4.5 shows an example. Inside the contour labelled 400, all nodes have $f(n)$ less than or equal to 400, and so on. Then, because A* expands the leaf node of lowest f, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing f-cost.

CONTOURS

With uniform-cost search (A* search using $h = 0$), the bands will be "circular" around the start state. With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path. If we define f^* to be the cost of the optimal solution path, then we can say the following:

- A* expands all nodes with $f(n) < f^*$.
- A* may then expand some of the nodes right on the "goal contour," for which $f(n)$ =/*, before selecting a goal node.

Intuitively, it is obvious that the first solution found must be the optimal one, because nodes in all subsequent contours will have higher f-cost, and thus higher g-cost (because all goal states have $h(n) = 0$). Intuitively, it is also obvious that A* search is complete. As we add bands of

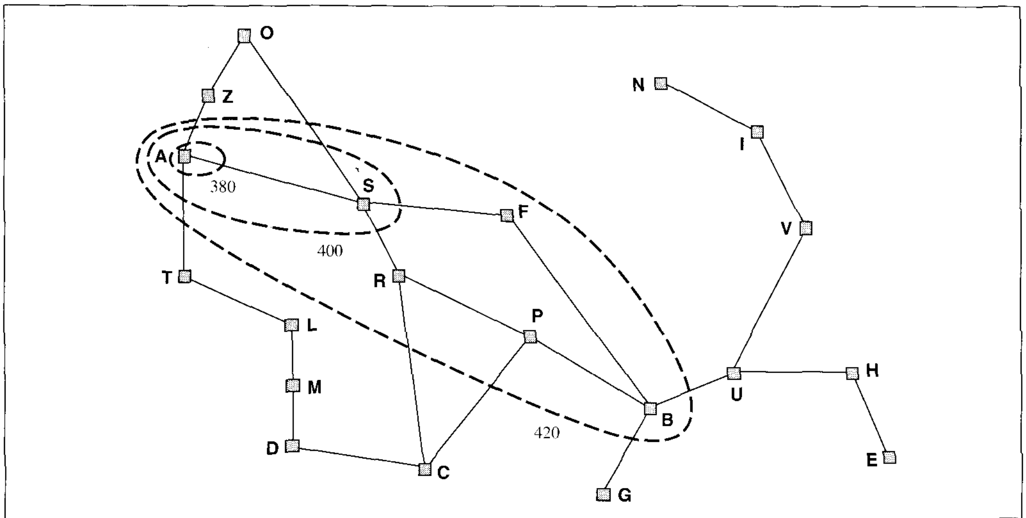


Figure 4.5 Map of Romania showing contours at $f = 380$, $f = 400$ and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs lower than the contour value.

increasing f, we must eventually reach a band where f is equal to the cost of the path to a goal state. We will turn these intuitions into proofs in the next subsection.

OPTIMALLY EFFICIENT

One final observation is that among optimal algorithms of this type—algorithms that extend search paths from the root—A* is **optimally efficient** for any given heuristic function. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A*. We can explain this as follows: any algorithm that *does not* expand all nodes in the contours between the root and the goal contour runs the risk of missing the optimal solution. A long and detailed proof of this result appears in Dechter and Pearl (1985).

Proof of the optimality of A*

Let G be an optimal goal state, with path cost f^*. Let G_2 be a suboptimal goal state, that is, a goal state with path cost $g(G_2) > f^*$. The situation we imagine is that A* has selected G_2 from the queue. Because G_2 is a goal state, this would terminate the search with a suboptimal solution (Figure 4.6). We will show that this is not possible.

Consider a node n that is currently a leaf node on an optimal path to G (there must be some such node, unless the path has been completely expanded—in which case the algorithm would have returned G). Because h is admissible, we must have

$$f^* \geq f(n).$$

Furthermore, if n is not chosen for expansion over G_2, we must have

$$f(n) \geq f(G_2).$$

Combining these two together, we get

$$f^* > f(G_2).$$

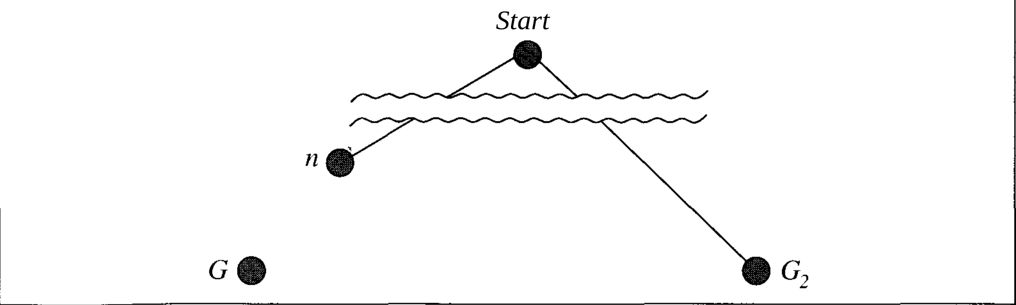


Figure 4.6 Situation at the point where a suboptimal goal state G_2 is about to be expanded. Node n is a leaf node on an optimal path from the start node to the optimal goal state G.

But because G_2 is a goal state, we have $h(G_2) = 0$; hence $f(G_2) = g(G_2)$. Thus, we have proved, from our assumptions, that

$$f^* \geq g(G_2).$$

This contradicts the assumption that G_2 is suboptimal, so it must be the case that A* never selects a suboptimal goal for expansion. Hence, because it only returns a solution after selecting it for expansion, A* is an optimal algorithm.

Proof of the completeness of A*

We said before that because A* expands nodes in order of increasing f, it must eventually expand to reach a goal state. This is true, of course, unless there are infinitely many nodes with $f(n) < f^*$. The only way there could be an infinite number of nodes is either (a) there is a node with an infinite branching factor, or (b) there is a path with a finite path cost but an infinite number of nodes along it.[2]

LOCALLY FINITE GRAPHS

Thus, the correct statement is that A* is complete on **locally finite graphs** (graphs with a finite branching factor) provided there is some positive constant 6 such that every operator costs at least δ.

Complexity of A*

That A* search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying. Unfortunately, it does not mean that A* is the answer to all our searching needs. The catch is that, for most problems, the number of nodes within the goal contour search space is still exponential in the length of the solution. Although the proof of the result is beyond the scope of this book, it has been shown that exponential growth will occur unless the error in the heuristic

[2] Zeno's paradox, which purports to show that a rock thrown at a tree will never reach it, provides an example that violates condition (b). The paradox is created by imagining that the trajectory is divided into a series of phases, each of which covers half the remaining distance to the tree; this yields an infinite sequence of steps with a finite total cost.

function grows no faster than the logarithm of the actual path cost. In mathematical notation, the condition for subexponential growth is that

$$|h(n) - h^*(n)| \leq O(\log h^*(n)),$$

where $h^*(n)$ is the *true* cost of getting from n to the goal. For almost all heuristics in practical use, the error is at least proportional to the path cost, and the resulting exponential growth eventually overtakes any computer. Of course, the use of a good heuristic still provides enormous savings compared to an uninformed search. In the next section, we will look at the question of designing good heuristics.

Computation time is not, however, A*'s main drawback. Because it keeps all generated nodes in memory, A* usually runs out of space long before it runs out of time. Recently developed algorithms have overcome the space problem without sacrificing optimality or completeness. These are discussed in Section 4.3.

4.2 HEURISTIC FUNCTIONS

So far we have seen just one example of a heuristic: straight-line distance for route-finding problems. In this section, we will look at heuristics for the 8-puzzle. This will shed light on the nature of heuristics in general.

The 8-puzzle was one of the earliest heuristic search problems. As mentioned in Section 3.3, the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the initial configuration matches the goal configuration (Figure 4.7).

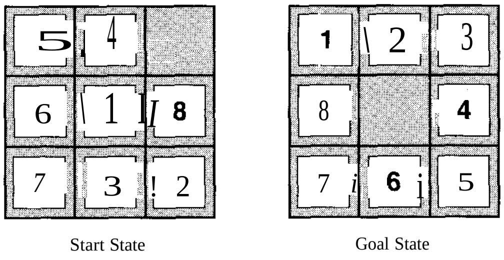


Figure 4.7 A typical instance of the 8-puzzle.

The 8-puzzle is just the right level of difficulty to be interesting. A typical solution is about 20 steps, although this of course varies depending on the initial state. The branching factor is about 3 (when the empty tile is in the middle, there are four possible moves; when it is in a corner there are two; and when it is along an edge there are three). This means that an exhaustive search to depth 20 would look at about $3^{20} = 3.5 \times 10^9$ states. By keeping track of repeated states, we could cut this down drastically, because there are only $9! = 362,880$ different arrangements of 9 squares. This is still a large number of states, so the next order of business is to find a good heuristic function. If we want to find the shortest solutions, we need a heuristic function that never overestimates the number of steps to the goal. Here are two candidates:

- h\ = the number of tiles that are in the wrong position. For Figure 4.7, none of the 8 tiles is in the goal position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.

MANHATTAN DISTANCE

- h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the **city block distance** or **Manhattan distance.** h_2 is also admissible, because any move can only move one tile one step closer to the goal. The 8 tiles in the start state give a Manhattan distance of

$$h_2 = 2 + 3 + 2 + 1 + 2 + 2 + 1 + 2 = 15$$

The effect of heuristic accuracy on performance

EFFECTIVE BRANCHING FACTOR

One way to characterize the quality of a heuristic is the **effective branching factor** b^*. If the total number of nodes expanded by A* for a particular problem is N, and the solution depth is d, then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain N nodes. Thus,

$$N = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d.$$

For example, if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.91. Usually, the effective branching factor exhibited by a given heuristic is fairly constant over a large range of problem instances, and therefore experimental measurements of b" on a small set of problems can provide a good guide to the heuristic's overall usefulness. A well-designed heuristic would have a value of b^* close to 1, allowing fairly large problems to be solved. To test the heuristic functions h\ and h_2, we randomly generated 100 problems each with solution lengths $2, 4, \ldots, 20$, and solved them using A* search with h\ and h_2, as well as with uninformed iterative deepening search. Figure 4.8 gives the average number of nodes expanded by each strategy, and the effective branching factor. The results show that h_2 is better than h_1, and that uninformed search is much worse.

DOMINATES

One might ask if h_2 is *always* better than h_1. The answer is yes. It is easy to see from the definitions of the two heuristics that for any node n, $h_2(n) > h_1(n)$. We say that h_2 **dominates** h_1. Domination translates directly into efficiency: A* using h_2 will expand fewer nodes, on average, than A* using h_1. We can show this using the following simple argument. Recall the observation on page 98 that every node with $f(n) < f^*$ will be expanded. This is the same as saying that every node with h(n)</* — $g(n)$ will be expanded. But because h_2 is at least as big as h\ for all nodes, every node that is expanded by A* search with h_2 will also be expanded with h_1, and h_1 may also cause other nodes to be expanded as well. *Hence, it is always better to use a heuristic function with higher values, as long as it does not overestimate.*

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 4.8 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h\, h_2. Data are averaged over 100 instances of the 8-puzzle, for various solution lengths.

Inventing heuristic functions

We have seen that both h\ and h_2 are fairly good heuristics for the 8-puzzle, and that h_2 is better. But we do not know how to invent a heuristic function. How might one have come up with h_2? Is it possible for a computer to mechanically invent such a heuristic?

h_1 and h_2 are estimates to the remaining path length for the 8-puzzle, but they can also be considered to be perfectly accurate path lengths for simplified versions of the puzzle. If the rules of the puzzle were changed so that a tile could move anywhere, instead of just to the adjacent empty square, then h \ would accurately give the number of steps to the shortest solution. Similarly, if a tile could move one square in any direction, even onto an occupied square, then h_2 would give the exact number of steps in the shortest solution. A problem with less restrictions on the operators is called a **relaxed problem.** *It is often the case that the cost of an exact solution to a relaxed problem is a good heuristic for the original problem.*

RELAXED PROBLEM

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.[3] For example, if the 8-puzzle operators are described as

A tile can move from square A to square B if A is adjacent to B and B is blank,

we can generate three relaxed problems by removing one or more of the conditions:

(a) A tile can move from square A to square B if A is adjacent to B.
(b) A tile can move from square A to square B if B is blank.
(c) A tile can move from square A to square B.

Recently, a program called ABSOLVER was written that can generate heuristics automatically from problem definitions, using the "relaxed problem" method and various other techniques (Prieditis, 1993). ABSOLVER generated a new heuristic for the 8-puzzle better than any existing heuristic, and found the first useful heuristic for the famous Rubik's cube puzzle.

[3] In Chapters 7 and 11, we will describe formal languages suitable for this task. For now, we will use English.

One problem with generating new heuristic functions is that one often fails to get one "clearly best" heuristic. If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem, and none of them dominates any of the others, which should we choose? As it turns out, we need not make a choice. We can have the best of all worlds, by defining

$$h(n) = max(h_1(n), \ldots, h_m(n)).$$

This composite heuristic uses whichever function is most accurate on the node in question. Because the component heuristics are admissible, *h* is also admissible. Furthermore, *h* dominates all of the individual heuristics from which it is composed.

Another way to invent a good heuristic is to use statistical information. This can be gathered by running a search over a number of training problems, such as the 100 randomly chosen 8-puzzle configurations, and gathering statistics. For example, we might find that when $h_2(n) = 14$, it turns out that 90% of the time the real distance to the goal is 18. Then when faced with the "real" problem, we can use 18 as the value whenever $h_2(n)$ reports 14. Of course, if we use probabilistic information like this, we are giving up on the guarantee of admissibility, but we are likely to expand fewer nodes on average.

FEATURES

Often it is possible to pick out **features** of a state that contribute to its heuristic evaluation function, even if it is hard to say exactly what the contribution should be. For example, the goal in chess is to checkmate the opponent, and relevant features include the number of pieces of each kind belonging to each side, the number of pieces that are attacked by opponent pieces, and so on. Usually, the evaluation function is assumed to be a linear combination of the feature values. Even if we have no idea how important each feature is, or even if a feature is good or bad, it is still possible to use a learning algorithm to acquire reasonable coefficients for each feature, as demonstrated in Chapter 18. In chess, for example, a program could learn that one's own queen should have a large positive coefficient, whereas an opponent's pawn should have a small negative coefficient.

Another factor that we have not considered so far is the search cost of actually running the heuristic function on a node. We have been assuming that the cost of computing the heuristic function is about the same as the cost of expanding a node, so that minimizing the number of nodes expanded is a good thing. But if the heuristic function is so complex that computing its value for one node takes as long as expanding hundreds of nodes, then we need to reconsider. After all, it is easy to have a heuristic that is perfectly accurate—if we allow the heuristic to do, say, a full breadth-first search "on the sly." That would minimize the number of nodes expanded by the real search, but it would not minimize the overall search cost. A good heuristic function must be efficient as well as accurate.

Heuristics for constraint satisfaction problems

In Section 3.7, we examined a class of problems called **constraint satisfaction problems** (CSPs). A constraint satisfaction problem consists of a set of variables that can take on values from a given domain, together with a set of constraints that specify properties of the solution. Section 3.7 examined uninformed search methods for CSPs, mostly variants of depth-first search. Here, we extend the analysis by considering heuristics for selecting a variable to instantiate and for choosing a value for the variable.

To illustrate the basic idea, we will use the map-coloring problem shown in Figure 4.9. (The idea of map coloring is to avoid coloring adjacent countries with the same color.) Suppose that we can use at most three colors (red, green, and blue), and that we have chosen green for country A and red for country B. Intuitively, it seems obvious that we should color E next, because the only possible color for E is blue. All the other countries have a choice of colors, and we might make the wrong choice and have to backtrack. In fact, once we have colored E blue, then we are forced to color C red and F green. After that, coloring D either blue or red results in a solution. In other words, we have solved the problem with no search at all.

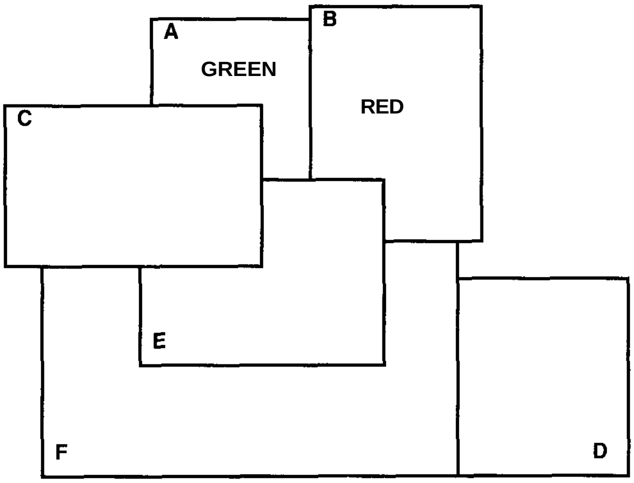


Figure 4.9 A map-coloring problem after the first two variables (A and B) have been selected. Which country should we color next?

MOST-CONSTRAINED-VARIABLE

This intuitive idea is called the **most-constrained-variable** heuristic. It is used with forward checking (see Section 3.7), which keeps track of which values are still allowed for each variable, given the choices made so far. At each point in the search, the variable with the *fewest* possible values is chosen to have a value assigned. In this way, the branching factor in the search tends to be minimized. For example, when this heuristic is used in solving n-queens problems, the feasible problem size is increased from around 30 for forward checking to approximately 100. The **most-constraining-variable** heuristic is similarly effective. It attempts to reduce the branching factor on future choices by assigning a value to the variable that is involved in the largest number of constraints on other unassigned variables.

MOST-CONSTRAINING-VARIABLE

Once a variable has been selected, we still need to choose a value for it. Suppose that we decide to assign a value to country C after A and B. One's intuition is that red is a better choice than blue, because it leaves more freedom for future choices. This intuition is the **least-constraining-value** heuristic—choose a value that rules out the smallest number of values in variables connected to the current variable by constraints. When applied to the n-queens problem, it allows problems up to n=1000 to be solved.

LEAST-CONSTRAINING-VALUE

4.3 MEMORY BOUNDED SEARCH

Despite all the clever search algorithms that have been invented, the fact remains that some problems are intrinsically difficult, by the nature of the problem. When we run up against these exponentially complex problems, something has to give. Figure 3.12 shows that *thefirst thing to give is usually the available memory.* In this section, we investigate two algorithms that are designed to conserve memory. The first, IDA*, is a logical extension of ITERATIVE-DEEPENING-SEARCH to use heuristic information. The second, SMA*, is similar to A*, but restricts the queue size to fit into the available memory.

Iterative deepening A* search (IDA*)

IDA*

In Chapter 3, we showed that iterative deepening is a useful technique for reducing memory requirements. We can try the same trick again, turning A* search into iterative deepening A*, or IDA* (see Figure 4.10). In this algorithm, each iteration is a depth-first search, just as in regular iterative deepening. The depth-first search is modified to use an f-cost limit rather than a depth limit. Thus, each iteration expands all nodes inside the contour for the current f-cost, peeping over the contour to find out where the next contour lies. (See the DFS-CONTOUR function in Figure 4.10.) Once the search inside a given contour has been completed, a new iteration is started using a new f-cost for the next contour.

IDA* is complete and optimal with the same caveats as A* search, but because it is depth-first, it only requires space proportional to the longest path that it explores. If b is the smallest operator cost and f^* the optimal solution cost, then in the worst case, IDA* will require bf^*/δ nodes of storage. In most cases, bd is a good estimate of the storage requirements.

The time complexity of IDA* depends strongly on the number of different values that the heuristic function can take on. The Manhattan distance heuristic used in the 8-puzzle takes on one of a small number of integer values. Typically, f only increases two or three times along any solution path. Thus, IDA* only goes through two or three iterations, and its efficiency is similar to that of A*—in fact, the last iteration of IDA* usually expands roughly the same number of nodes as A*. Furthermore, because IDA* does not need to insert and delete nodes on a priority queue, its overhead per node can be much less than that for A*. Optimal solutions for many practical problems were first found by IDA*, which for several years was the only optimal, memory-bounded, heuristic algorithm.

Unfortunately, IDA* has difficulty in more complex domains. In the travelling salesperson problem, for example, the heuristic value is different for every state. This means that each contour only includes one more state than the previous contour. If A* expands N nodes, IDA* will have to go through N iterations and will therefore expand $1 + 2 + \cdots + N = O(N^2)$nodes. Now if$N$ is too large for the computer's memory, then N^2 is almost certainly too long to wait!

One way around this is to increase the/-cost limit by a fixed amount ϵ on each iteration, so that the total number of iterations is proportional to $1/\epsilon$. This can reduce the search cost, at the expense of returning solutions that can be worse than optimal by at most ϵ. Such an algorithm is called **ϵ-admissible.**

ϵ-ADMISSIBLE

```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: f-limit, the current f-COST limit
          mot, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f-COST(root)
  loop do
      solution, f-limit — DFS-CONTOUR(root, f-limit)
      if solution is non-null then return solution
      itf-limit = ∞ then return failure; end
```

```
function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f-COST limit
  inputs: node, a node
          f-limit, the current f-COST limit
  static: next-f, the f-COST limit for the next contour, initially ∞

  if f-COST[node] > f-limit then return null, f-COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
      solution, new-f ← DFS-CONTOUR(s, f-limit)
      if solution is non-null then return solution, f-limit
      next-f ← MIN(next-f, new-f); end
  return null, next-f
```

Figure 4.10 The IDA* (Iterative Deepening A*) search algorithm.

SMA* search

IDA*'s difficulties in certain problem spaces can be traced to using *too little* memory. Between iterations, it retains only a single number, the current f-cost limit. Because it cannot remember its history, IDA* is doomed to repeat it. This is doubly true in state spaces that are graphs rather than trees (see Section 3.6). IDA* can be modified to check the current path for repeated states, but is unable to avoid repeated states generated by alternative paths.

SMA* In this section, we describe the SMA* (Simplified Memory-Bounded A*) algorithm, which can make use of all available memory to carry out the search. Using more memory can only improve search efficiency—one could always ignore the additional space, but usually it is better to remember a node than to have to regenerate it when needed. SMA* has the following properties:

- It will utilize whatever memory is made available to it.
- It avoids repeated states as far as its memory allows.
- It is complete if the available memory is sufficient to store the *shallowest* solution path.
- It is optimal if enough memory is available to store the shallowest optimal solution path. Otherwise, it returns the best solution that can be reached with the available memory.
- When enough memory is available for the entire search tree, the search is optimally efficient.

The one unresolved question is whether SMA* is always optimally efficient among all algorithms given the same heuristic information and the same memory allocation.

The design of SMA* is simple, at least in overview. When it needs to generate a successor but has no memory left, it will need to make space on the queue. To do this, it drops a node from the queue. Nodes that are dropped from the queue in this way are called **forgotten nodes.** It prefers to drop nodes that are unpromising—that is, nodes with high/-cost. To avoid reexploring subtrees that it has dropped from memory, it retains in the ancestor nodes information about the quality of the best path in the forgotten subtree. In this way, it *only* regenerates the subtree when *all other paths* have been shown to look worse than the path it has forgotten. Another way of saying this is that if all the descendants of a node n are forgotten, then we will not know which way to go from n, but we will still have an idea of how worthwhile it is to go anywhere from n.

SMA* is best explained by an example, which is illustrated in Figure 4.11. The top of the figure shows the search space. Each node is labelled with $g + h — f$ values, and the goal nodes (D, F, I, J) are shown in squares. The aim is to find the lowest-cost goal node with enough memory for only *three* nodes. The stages of the search are shown in order, left to right, with each stage numbered according to the explanation that follows. Each node is labelled with its current f-cost, which is continuously maintained to reflect the least f-cost of any of its descendants.[4] Values in parentheses show the value of the best forgotten descendant. The algorithm proceeds as follows:

1. At each stage, one successor is added to the deepest lowest-/-cost node that has some successors not currently in the tree. The left child B is added to the root A.
2. Now $f(A)$ is still 12, so we add the right child G (f = 13). Now that we have seen all the children of A, we can update its f-cost to the minimum of its children, that is, 13. The memory is now full.
3. G is now designated for expansion, but we must first drop a node to make room. We drop the shallowest highest-/-cost leaf, that is, B. When we have done this, we note that A's best forgotten descendant has f = 15, as shown in parentheses. We then add H, with $f(H)$ = 18. Unfortunately, H is not a goal node, but the path to H uses up all the available memory. Hence, there is no way to find a solution through H, so we set $f(H) = \infty$.
4. G is expanded again. We drop H, and add I, with $f(I)$ = 24. Now we have seen both successors of G, with values of oo and 24, so $f(G)$ becomes 24. $f(A)$ becomes 15, the minimum of 15 (forgotten successor value) and 24. Notice that I is a goal node, but it might not be the best solution because A's/-cost is only 15.
5. A is once again the most promising node, so B is generated for the second time. We have found that the path through G was not so great after all.
6. C, the first successor of B, is a nongoal node at the maximum depth, so $f(C)$ = oo.
7. To look at the second successor, D, we first drop C. Then $f(D)$ = 20, and this value is inherited by B and A.
8. Now the deepest, lowest-/-cost node is D. D is therefore selected, and because it is a goal node, the search terminates.

[4] Values computed in this way are called **backed-up values.** Because $f(n)$ is supposed to be an estimate of the least-cost solution path through n, and a solution path through n is bound to go through one of n's descendants, backing up the least f-cost among the descendants is a sound policy.

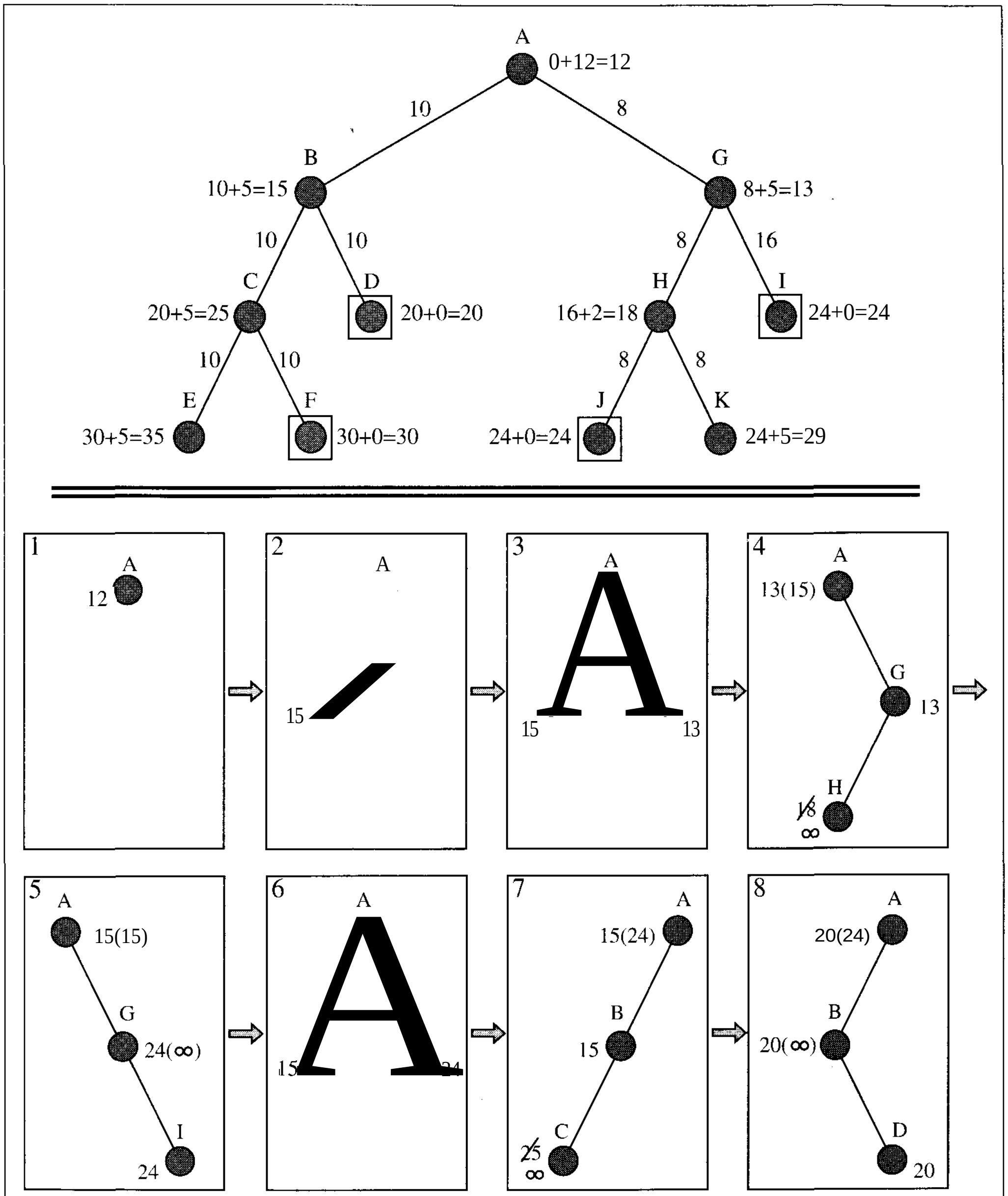


Figure 4.11 Progress of an SMA* search with a memory size of three nodes, on the state space shown at the top. Each node is labelled with its *current f*-cost. Values in parentheses show the value of the best forgotten descendant.

In this case, there is enough memory for the shallowest optimal solution path. If J had had a cost of 19 instead of 24, however, SMA* still would not have been able to find it because the solution path contains four nodes. In this case, SMA* would have returned D, which would be the best reachable solution. It is a simple matter to have the algorithm signal that the solution found may not be optimal.

A rough sketch of SMA* is shown in Figure 4.12. In the actual program, some gymnastics are necessary to deal with the fact that nodes sometimes end up with some successors in memory and some forgotten. When we need to check for repeated nodes, things get even more complicated. SMA* is the most complicated search algorithm we have seen yet.

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue — MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
     if Queue is empty then return failure
     n — deepest least-f-cost node in Queue
     if GOAL-TEST(n) then return success
     s — NEXT-SUCCESSOR(n)
     if s is not a goal and is at maximum depth then
        f(s) - ∞
     else
        f(s) ← MAX(f(n), g(s)+h(s))
     if all of n's successors have been generated then
        update n's f-cost and those of its ancestors if necessary
     if SUCCESSORS(n) all in memory then remove n from Queue
     if memory is full then
        delete shallowest, highest-f-cost node in Queue
        remove it from its parent's successor list
        insert its parent on Queue if necessary
     insert s on Queue
  end
```

Figure 4.12 Sketch of the SMA* algorithm. Note that numerous details have been omitted in the interests of clarity.

Given a reasonable amount of memory, SMA* can solve significantly more difficult problems than A* without incurring significant overhead in terms of extra nodes generated. It performs well on problems with highly connected state spaces and real-valued heuristics, on which IDA* has difficulty. On very hard problems, however, it will often be the case that SMA* is forced to continually switch back and forth between a set of candidate solution paths. Then the extra time required for repeated regeneration of the same nodes means that problems that would be practically solvable by A*, given unlimited memory, become intractable for SMA*. That is to say, memory limitations can make a problem intractable from the point of view of computation time. Although there is no theory to explain the trade-off between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.

4.4 ITERATIVE IMPROVEMENT ALGORITHMS

ITERATIVE IMPROVEMENT

We saw in Chapter 3 that several well-known problems (for example, 8-queens and VLSI layout) have the property that the state description itself contains all the information needed for a solution. The path by which the solution is reached is irrelevant. In such cases, **iterative improvement** algorithms often provide the most practical approach. For example, we start with all 8 queens on the board, or all wires routed through particular channels. Then, we might move queens around trying to reduce the number of attacks; or we might move a wire from one channel to another to reduce congestion. *The general idea is to start with a complete configuration and to make modifications to improve its quality.*

The best way to understand iterative improvement algorithms is to consider all the states laid out on the surface of a landscape. The height of any point on the landscape corresponds to the evaluation function of the state at that point (Figure 4.13). The idea of iterative improvement is to move around the landscape trying to find the highest peaks, which are the optimal solutions. Iterative improvement algorithms usually keep track of only the current state, and do not look ahead beyond the immediate neighbors of that state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia. Nonetheless, sometimes iterative improvement is the method of choice for hard, practical problems. We will see several applications in later chapters, particularly to neural network learning in Chapter 19.

HILL-CLIMBING

GRADIENT DESCENT

SIMULATED ANNEALING

Iterative improvement algorithms divide into two major classes. **Hill-climbing** (or, alternatively, **gradient descent** if we view the evaluation function as a cost rather than a quality) algorithms always try to make changes that improve the current state. **Simulated annealing** algorithms can sometimes make changes that make things worse, at least temporarily.

Hill-climbing search

The hill-climbing search algorithm is shown in Figure 4.14. It is simply a loop that continually moves in the direction of increasing value. The algorithm does not maintain a search tree, so the node data structure need only record the state and its evaluation, which we denote by VALUE. One important refinement is that when there is more than one best successor to choose from, the algorithm can select among them at random. This simple policy has three well-known drawbacks:

◇ **Local maxima:** a local maximum, as opposed to a global maximum, is a peak that is lower than the highest peak in the state space. Once on a local maximum, the algorithm will halt even though the solution may be far from satisfactory.

◇ **Plateaux:** a plateau is an area of the state space where the evaluation function is essentially flat. The search will conduct a random walk.