CHAPTER

4

HEURISTIC SEARCH

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

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

Of course, the problem of finding the best node to expand next is generally no easier than the search problem that we are trying to solve. So what we typically do is apply some sort of *heuristic*, or "rule of thumb," to decide what to do.[8] Given a list of nodes to be expanded, we might simply guess how far each is from a goal node and expand the one we thought to be closest.

As an example, consider the instance of the 8-puzzle shown in Figure 4.1, and suppose that our goal is to arrange the tiles from 1 to 8 clockwise around the edge of the puzzle. If we estimate the distance to the goal by

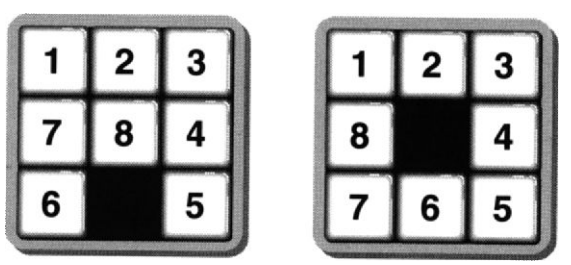


FIGURE 4.1 An instance of the 8-puzzle

8 The description of a heuristic as a rule of thumb is Feigenbaum's.

simply counting the number of misplaced tiles, then we will expect it to take three moves to solve the problem in Figure 4.1, since three tiles are misplaced there. (The 6, 7, and 8 are all misplaced.) If we move the 6 to the right, then only two tiles are misplaced. Moving the 8 down leaves the heuristic estimate of the distance to the goal unchanged, while moving the 5 to the left actually *increases* the expected distance. We summarize this below.

Blank Moves	Distance
left	2
right	4
up	3

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

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

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

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

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

Now consider the maze shown in Figure 4.2, where we enter the maze on the left and exit on the right. As shown, there are two solutions—a simple one that begins with a step downward and a much more complex one that begins with a step directly toward the exit but is then deflected in other directions. The algorithm we have described will find the longer

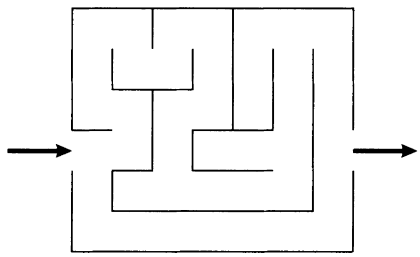


FIGURE 4.2
A maze with two solutions

of these two paths, since it will begin by moving toward the goal and then be committed to this choice.

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

4.1 SEARCH AS FUNCTION MAXIMIZATION

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

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

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

4.1.1 Hill Climbing

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

This technique is known as *hill climbing* or *steepest ascent* for the obvious reason; it is as if one attempted to find the highest point in a

landscape by simply walking as much uphill as possible. Here is the search version.

PROCEDURE 4.1.1 **Hill climbing**

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

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

PROCEDURE 4.1.2 **Best-first search**

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

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

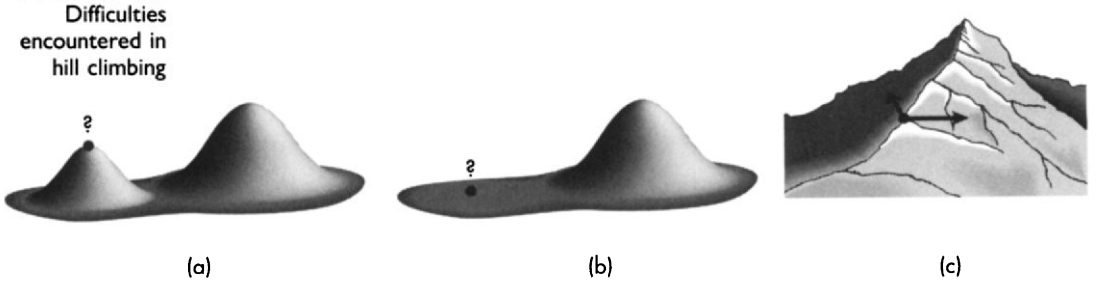


FIGURE 4.3 Difficulties encountered in hill climbing

has not yet been found. In a maze where we are measuring progress by our distance to the exit, it may be necessary to move away from the exit in order to find a solution. Solving the 8-puzzle typically requires that we dislodge correctly placed tiles in order to reposition others satisfactorily. This is known as the *foothill problem* because we have to avoid finding ourselves trapped on a foothill while trying to climb a mountain.

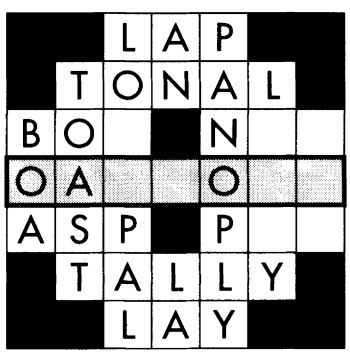
The second problem with hill climbing is that it is difficult to deal with plateaus, or situations where all of the local steps are indistinguishable. In the 8-puzzle, perhaps none of the immediately available moves influences the count of misplaced tiles. In a maze, perhaps one's only choices are to move parallel to the exit because the walls block any other alternative.

The two problems that we have discussed also appear in the crossword-puzzle problem, where the function we are trying to maximize is the number of words that have been successfully entered into the crossword frame. A local maximum occurs when we need to retract words that were inserted earlier in order to address a problem that was not previously recognized. We saw an example of this in Figure 2.10, which we repeat in Figure 4.4. The problem, as we discussed in Section 2.2.3, is that we may backtrack only part way to the real difficulty and then simply return to where we started. In functional terms, a large local maximum can be difficult to avoid simply because there is no incentive to move the distance required to escape it.

A crossword-puzzle plateau is shown in Figure 4.5. Here, most of the words that can be inserted into the marked slot will cause an insurmountable difficulty of some kind (either no legal crossword at the first letter or none at the third). But this may not be apparent when we fill the slot indicated in the figure.

The final problem with hill climbing (which has no analog in the crossword-puzzle problem of which I am aware) involves *ridges*. As shown in Figure 4.3, a ridge involves a situation where although there is a direction in which we would *like* to move, none of the allowed steps (indicated by arrows in the figure) actually takes us in that direction.

FIGURE 4.4
A local maximum



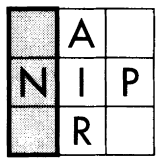


FIGURE 4.5
A plateau

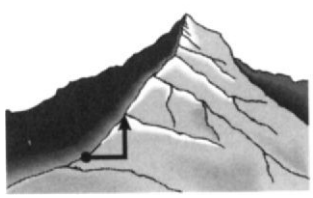
When encountering a problem such as this, we need to do something similar to what is shown in Figure 4.6, taking two steps (one in each direction) in order to move along the ridge. It's like tacking in sailing—if our intention is to move directly upwind, we can do it only by sailing partially upwind in one direction and then partially upwind in another.

More complex examples may require that we take more than two steps in order to move along the ridge. Consider Rubik's cube; if the path to the solution involves interchanging two of the small component cubes, or cubies, we will typically use a multimove sequence in order to interchange the given cubies without affecting the work completed thus far.[9]

These sequences of moves that achieve specific subgoals in the Rubik's cube problem are generally referred to as *macro operators* and have uses far outside the ridge problem. As an example, one way to think about macro operators is as specific, large "steps" that we can use in our function-maximization efforts. They can therefore help to address the problem of local maxima as well.

Another way to think of macro operators is in terms of the islands used in island-driven search (see Exercise 11 in Chapter 3). The macro operators allow us to identify islands that we know we will be able to reach from our current position in the search space. We will return to this idea in Chapter 14 when we discuss hierarchical problem solving.

FIGURE 4.6
Moving along a ridge



9 When I was in Oxford (my astrophysicist days), I did some research with Rubik's cousin. This was before Rubik's cubes had appeared throughout the West; to make a long story short, I own the third Rubik's cube to have made its way out of Hungary. The mechanism used in the early cubes was very different from that used in more recent versions—they were very stiff, needed to be aligned precisely before each move, and exuded a very fine black dust when turned. As the cubes were used, more and more dust would come out and they would get progressively looser. Eventually, they would just collapse into a pile of dust and oddly shaped pieces. Not that this has anything to do with AI, of course.

There is no free lunch, of course. The difficulty with solving search problems using macro operators is that these operators need to be discovered in the first place. This is *itself* a search problem—the space of macro operators in the 8-puzzle, for example, is of size approximately 3^l where l is the length of the operator. This space, too, grows exponentially with the difficulty of the problem. It is also not obvious what criteria should be used to select one element of this large space over another as a useful macro operator.

4.1.2 Simulated Annealing

There is another function-maximization technique that appears to have a search analog. This is the idea of *simulated annealing* that was developed in the early 1980s.

The purpose of simulated annealing is to avoid the problem of local maxima that is the most important difficulty encountered in hill climbing. What we do is to solve a problem using conventional hill climbing techniques, but occasionally take a step in a direction other than that in which the rate of change is maximal. As time passes, the probability that a downhill step is taken is gradually reduced.

As shown in Figure 4.7, the idea behind simulated annealing is the following: The undirected steps that are taken clearly will not in general *reduce* the ability of the procedure to find a global maximum of the function in question, although these steps can be expected to increase the running time of the algorithm. But it is possible that these undirected steps dislodge the problem solver from a foothill that has been encountered by accident. The large size of the early steps is intended to avoid local maxima that extend over a wide area of the search space; subsequent undirected steps are smaller in the expectation that it is only more restricted local maxima that need to be avoided.

The name of the procedure is the result of an analogy with metal-casting techniques. When molten metal is poured into a mold, it gradually cools into a solid that will retain its overall shape. In order to make the final solid as nonbrittle as possible, the temperature should be reduced gradually; when this is done and the metal has solidified, it is said to have *annealed*. In simulated annealing, when the probability of the undirected step (related to the "temperature" of the procedure) becomes small enough that it can no longer move the problem solver's attention off of the maxi-

FIGURE 4.7
A non-uphill step avoids a local maximum

mum with which it is currently dealing, the algorithm's result can be said to have annealed in a similar way.

There are two remarkable things about simulated annealing. The first is that it took so long for it to be discovered—the problem of maximizing functions of multiple variables is an old one, and for an idea of this generality to have gone unnoticed until the past few years is remarkable.

More remarkable than simulated annealing's recency is its effectiveness. As a start, it is possible to prove that if the "temperature" of the algorithm is reduced only very slowly, a global maximum will always be found. The argument, roughly speaking, is the following: Suppose that the function's value at the global maximum is m and that the value at the best *local* maximum is $l < m$.

There will be some temperature t that is large enough to cause an undirected step off of the local maximum l but not off of the global maximum. Since the temperature is being reduced only slowly, the algorithm will spend enough time working with the temperature t that it will eventually stumble onto the global maximum and remain there.

Simulated annealing's practical effectiveness is no less impressive; this simple idea improves the performance of conventional function-maximization techniques tremendously. As an example, consider the problem of sequencing various sections of an automobile through the construction process. These parts need to be fabricated, finished, painted, and so on using a variety of tools that need to work on each part for differing amounts of time. Since the aim is to maximize the output of any particular factory, the problem can be viewed as one of function maximization. An engineer at General Motors recently applied simulated annealing to this problem; the result was so effective that the cost of manufacture of every automobile GM makes fell by about $20, a savings of millions of dollars annually.

Does simulated annealing have a search analog? It does. What this idea suggests is that when attempting to solve a search problem, one should occasionally examine a node that appears substantially worse than the best node to be found on L. In the crossword-puzzle problem, for example, we should occasionally erase some of the most recently inserted words and proceed afresh. Although this idea has received very little attention by the search community *per se*, experimentation on the crossword-puzzle problem shows that it does indeed improve performance in this restricted domain.

4.2 A*

Let us now return to the problem shown in Figure 4.2, where we saw that when hill climbing does manage to solve some search problem, it may well not find the *best* solution.

We have remarked from time to time that in many cases the quality of the solution is measured by its depth in the search tree, the intent being

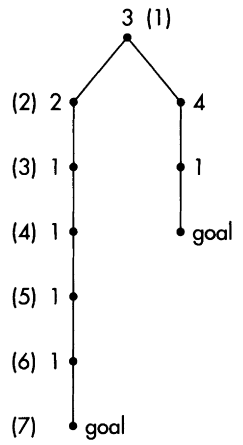


FIGURE 4.8
Finding a suboptimal solution

to find a solution at as shallow a depth as possible. When solving a maze, we look for the shortest path from the entrance to the exit; when solving the 8-puzzle, we look for the solution that slides as few tiles as possible.

We have already seen an example where hill-climbing fails to find an optimal solution in Figure 4.2; a simpler example where both hill climbing and best-first search find a suboptimal solution is shown in Figure 4.8. The number next to each node in the figure is the heuristic value assigned to that node, so that the label of 3 assigned to the root node means that we expect (correctly, in this case) this node to be three steps away from the goal. The heuristic becomes inaccurate for the nodes at depth 1, however; the left-hand node is labelled with a 2 even though the goal is five steps away along this path and the right-hand node is labelled with a 4 even though the goal could be reached in only two steps.

The parenthetical numbers in the figure show the order in which the nodes are expanded. The root node is expanded first, after which we expand its left-hand child because that child is believed to be closer to the goal than the right-hand one. Since each subsequent node on the left side of the tree is also expected to be closer to the goal than the right-hand node at depth 1, the left side of the tree is explored in its entirety and the goal at depth 6 is found instead of the goal at depth 3.

We should not be surprised that best-first search fails to find the best solution in cases such as this. After all, the idea in best-first search is to find *some* solution as quickly as possible by finding any node that is at distance 0 from a goal node; nowhere is any attempt made to find the goal node that is at minimal depth in the search tree. If we want to do this, we need to modify our search algorithm to reflect our interest in finding a goal at as shallow a depth as possible. The order in which we expand the nodes needs to take into account the fact that we want to expand shallow nodes in preference to deep ones.

Slightly more precisely, since the aim in best-first search is to find a goal as quickly as possible, it leads us to expand the node thought to be closest to a goal node. Since our modified aim is to find the *shallowest* goal as quickly as possible, we should expand instead the node that appears to be closest to a *shallow* goal. Instead of ordering the nodes in terms of distance to the goal, we should order them in terms of the quality (that is, expected depth) of the nearest goal to them.

So suppose that we have a node n at depth d in the search tree, and that we judge this node to be a distance $h'(n)$ from the nearest goal to it.[10] This goal is therefore judged to be at depth $d + h'(n)$ in the search space; it follows that instead of choosing for expansion the node with smallest $h'(n)$ as in best-first search (*since* $h'(n)$ is the expected distance to the goal), we should choose for expansion the node with smallest $d + h'(n)$. The depth of the node being considered is also a function of the node, and is typically denoted $g(n)$ and referred to as the "cost" of reaching the node n from the root node. Since our intention is to find a goal of minimal cost, we expand the nodes in order of increasing

$$f(n) = g(n) + h'(n) \tag{4.1}$$

This algorithm is known as the *A* algorithm*, and we apply it to the search problem of Figure 4.8 in Figure 4.9. Instead of labelling each node with the expected distance to the goal, we label it with two numbers, the depth of the node (0 for the root node, 1 for its children, 2 for their children and so on) *and* the expected distance to the goal. We then choose for expansion that node for which this sum is smallest.

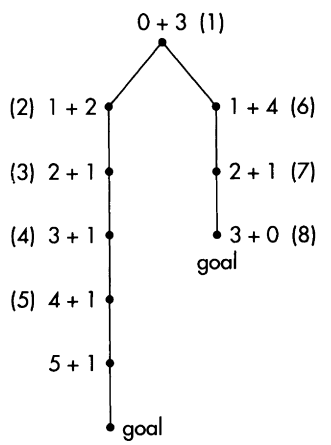


FIGURE 4.9 Finding an optimal solution. The first figure in the sum is the distance from the initial node and the second is the estimated distance to a goal.

10 We will assume throughout that h' is nonnegative, so that $h'(n) \geq 0$ for any node n.

PROCEDURE 4.2.1 **A***

1. *Set* L *to be a list of the initial nodes in the problem.*
2. *Let* n *be the node on* L *for which* f(n) = g(n) + h′(n) *is minimal. If* L *is empty, fail.*
3. *If* n *is a goal node, stop and return it and the path from the initial node to* n.
4. *Otherwise, remove* n *from* L *and add to* L *all of* n*'s children, labelling each with its path from the initial node. Return to step 2.*

As before, the optimistic value assigned to the left-hand node at depth 1 convinces us to expand this path before the other, since we expect to find a goal node at depth 3 below this node and expect the goal node below the other child to be at depth 5. However, when we actually *reach* depth 5 along the left-hand path and still have not reached the goal, we come to believe that the other path is the better one and examine it instead. The better of the two goal nodes is now found first.

There are two things to note here. The first is that there was no real difference between the nodes labelled (5) and (6) in Figure 4.9; both were expected to lead to goals at depth 5. We chose to expand (5) first because it was thought to be closer to the goal node, but the A* algorithm itself does not require us to make this choice.

The more important thing to note is that there is still no guarantee that A* finds the best solution to any particular search problem. In Figure 4.9, for example, if the node labelled (5) (that is, the one expanded fifth) were a goal node, it would be found before the optimal goal node on the right-hand path. Why is this?

4.2.1 Admissibility

The reason that we avoided expanding the right-hand path in this example is that we have pessimistically labelled the right-hand node at depth 1 with a 4, indicating that we do not expect to find a goal at any depth less than 5 along this path. Given this, we are led to examine the node at depth 4 along the left-hand path before finding the goal at depth 3 on the right-hand side. If the function estimating the distance to the goal had been *optimistic*, this would not have happened.

To formalize this, suppose that we denote by $h(n)$ (without the ′) the *actual* distance from the node n to a goal node. Now the heuristic estimating function h' will be optimistic just in case we always have $h' \leq h$ and in this case we have:

THEOREM 4.2.2 *If* h′(n) ≤ h(n) *for every node* n, *the A* algorithm will never return a suboptimal goal node.*

PROOF Suppose that the procedure eventually returns the node x when there was a better goal node s. If the path from the root node to s is

$$\langle n_0, n_1, \ldots, n_k \rangle$$

where $n_k = s$, we will derive a contradiction by showing that if the A* algorithm expands some portion of this path

$$\langle n_0, \ldots, n_i \rangle$$

then it also expands the next node along this path, n_{i+1}. Since the root node n_0 is always expanded, it will follow by induction that the node $s = n_k$ is expanded as well, and therefore that s would have been returned instead of x.

Since we have expanded n_i, the parent of n_{i+1}, it is clear that n_{i+1} will be expanded if it appears to be a better node than x is. In other words, if we could show that

$$g(n_{i+1}) + h'(n_{i+1}) < g(x) + h'(x)$$

the inductive step would follow and the proof would be complete. But since h' is optimistic, we must have

$$\begin{aligned} g(n_{i+1}) + h'(n_{i+1}) &\leq g(n_{i+1}) + h(n_{i+1}) \\ &= g(s) \\ &< g(x) \\ &\leq g(x) + h'(x) \end{aligned}$$

The first equality holds because the sum of the costs of getting to n_{i+1} and then from n_{i+1} to the goal node s is just the cost of getting to s directly. The second equality holds because $h(x) = 0$ and $0 \leq h'(x) \leq h(x)$. ■

An optimistic heuristic function h' is called *admissible*.

4.2.2 Examples

As an example, we revisit the search space in Figure 4.9 after modifying the heuristic value assigned to the right-hand node at depth 1 so that the heuristic function is admissible; the result is shown in Figure 4.10. The optimal goal node is indeed found in this case.

There are other examples of interest as well. The "heuristic" function $h'(n) = 0$ is clearly an admissible one, and leads to expanding the nodes simply in order of increasing $g(n)$. Since $g(n)$ is the cost of reaching node n, the A* algorithm reproduces breadth-first search in this case. Since

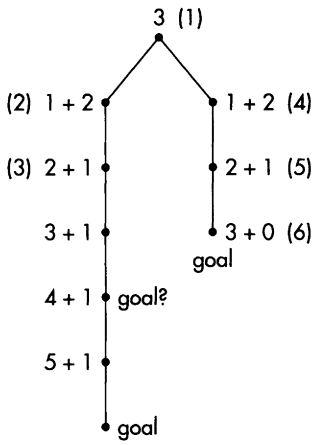


FIGURE 4.10
An admissible heuristic function

depth-first search often fails to find the best solution to a problem, it is clear that there is no admissible heuristic that mimics depth-first search in a similar way.

The perfect heuristic $h' = h$ is also always admissible, and is the topic of Exercise 6 at the end of this chapter.

In specific problems, there are often other admissible heuristics; the 8-puzzle provides a variety of examples. In this puzzle, the simple heuristic of counting the number of misplaced tiles is clearly admissible, since each move can reduce this value by at most one. The Manhattan distance heuristic is also admissible in this domain, since each move relocates only one tile and moves it in such a way that its contribution to the overall heuristic estimate drops by at most one.

Given that both heuristics are admissible, how are we to decide which one to use? Suppose that we denote the Manhattan distance heuristic by h'_M and the heuristic obtained by simply counting tiles by h'_C. Since

$$h'_C \leq h'_M \leq h$$

we see that h'_M is a better estimate of the true distance to the goal than is h'_C. In general, this means that we should prefer h'_M to h'_C; see also Exercise 12 at the end of this chapter.

Finally, we note that for finite graphs, it is always possible to construct an admissible heuristic function from a potentially inadmissible one by dividing the original heuristic by some large positive number. Of course, we want to choose the constant to be as near to 1 as possible, so that the eventual admissible heuristic is near to the true value h. Thus although counting the number of misplaced or misaligned cubies in Rubik's cube is not an admissible heuristic, this heuristic is admissible after being divided by 8, since each move affects eight cubies. (Unfortunately, the division by 8 tends to make this heuristic too small to be of much use.) The general

problem of constructing admissible heuristics from inadmissible ones is open.

4.3 EXTENSIONS AND IDA*

We conclude this chapter by discussing a few extensions to the A* algorithm.

The first situation we consider is that in which the "cost" of a node is not simply its depth in the search space, but is instead evaluated in some other fashion. We will continue to assume that the cost of getting to a node n is the sum of the costs of the individual arcs along the path from the root to n; we simply no longer assume that every arc is of unit cost.

It is now not hard to see that as long as we continue to use g(n) for the cost of reaching the node n, the use of the A* algorithm using the usual evaluation function given by (4.1) continues to find lowest-cost solutions. If we take $h'(n) = 0$ in this case, we get the search procedure known as *branch and bound*.

Next, suppose that we use A* to search a graph instead of a simple tree, as shown in Figure 4.11. Provided that we take g(n) to be the cost of the cheapest path found to the node n thus far and maintain a list of open nodes only, the algorithm remains principally unchanged.

PROCEDURE 4.3.1 **A* on graphs**

1. *Set* L *to be a list of the initial nodes in the problem.*
2. *Let* n *be the node on* L *for which* f(n) = g(n) + h'(n) *is minimal. If* L *is empty, fail.*

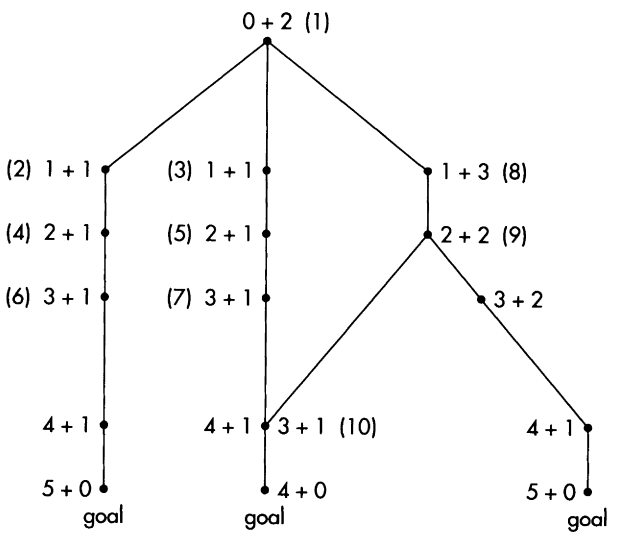


FIGURE 4.11 A* used to search a graph

3. *If* n *is a goal node, stop and return it and the path from the initial node to* n.
4. *Otherwise, remove* n *from* L *and add to* L *all of* n's *children, labelling each with its path from the initial node. If any child* c *is already on* L, *do not make a separate copy but instead relabel* c *with the shortest path connecting it to the initial node. Return to step 2.*

In the figure, the node that is expanded tenth is initially believed to be at depth 4 but is actually determined to be at depth 3 before it is expanded.

Finally, memory is a problem for the A* algorithm. Since it reduces to breadth-first search if $h' = 0$, we see that A* will potentially use an amount of memory that is exponential in the depth of the optimal goal node.

As in the case of blind search, this problem can be solved using iterative deepening. But now, instead of artificially pruning nodes that lie at a depth below some increasing cutoff, we prune nodes for which the nearest goal node can be shown to lie below the cutoff depth. As in iterative deepening, the cutoff depth is gradually increased until an answer is found.

PROCEDURE 4.3.2 **IDA***

1. *Set* c = 1; *this is the current depth cutoff.*
2. *Set* L *to be a list of the initial nodes in the problem. Set* $c' = \infty$; *this will be the cutoff on the next iteration.*
3. *Let* n *be the first node on* L. *If* L *is empty and* $c' = \infty$, *fail. If* L *is empty and* $c' \neq \infty$, *set* $c = c'$ *and return to step 2.*
4. *If* n *is a goal node, stop and return it and the path from the initial node to* n.
5. *Otherwise, remove* n *from* L. *For each child* n' *of* n, *if* $f(n') \leq c$, *add* n' *to the front of* L. *Otherwise, set* $c' = \min(c', f(n'))$. *Return to step 3.*

It is important to realize that the individual iterations of this algorithm are performed using conventional depth-first search and not using A*; the heuristic function is used to prune nodes but not to determine the order in which they should be expanded. After all, if the original iterations were to be expanded using the A* algorithm, an exponential amount of memory would still be needed! The algorithm is essentially the same as iterative deepening, with the heuristic being used to *anticipate* the nodes that will eventually be pruned because of the depth cutoff.

An example of Procedure 4.3.2 in use appears in Figure 4.12. For $c = 1$ we examine the root node only; for $c = 2$ we examine two nodes and so on. The individual searches are conducted in depth-first fashion, which is why node 12 is expanded before node 14 in the figure even though the

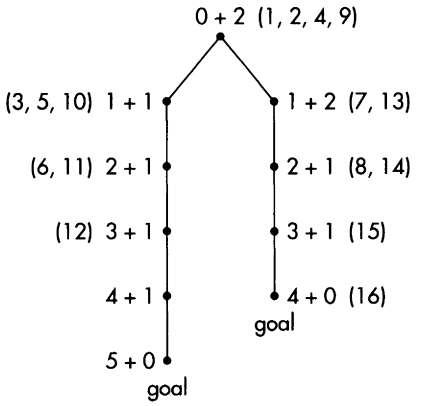


FIGURE 4.12
IDA*

heuristic estimates indicate that node 14 is likely to be the closest to a shallow goal.

Procedure 4.3.2 is known as *iterative-deepening A**, or simply IDA*. It is not too hard to show that it uses an amount of memory linear in the depth of the goal node, that it expands only the nodes expanded by A* itself, and that it continues to find optimal solutions when using an admissible heuristic.

4.4 FURTHER READING

Macro operators are discussed by Korf in [1985b]. Simulated annealing was introduced by Kirkpatrick *et al.* [1983]. I am afraid that I can give you no reference for the General Motors application; the work was reported to me personally by the engineer involved. The algorithm itself involves a variety of details that are omitted from the description in the text: An initial temperature needs to be chosen and a protocol needs to be selected by which the temperature is gradually reduced as the search proceeds.

The A* algorithm was developed by Hart, Nilsson, and Raphael [1968]. Although the problem of finding admissible heuristics in arbitrary search domains is open, it has been touched on by a variety of authors; one reasonable reference is Gaschnig [1979]. IDA* is discussed by Korf [1985a].

4.5 EXERCISES

1. Give an example of a search problem where hill climbing and best-first search behave differently.
2. In the text, we stated that the number of macro operators of length l in the 8-puzzle could be approximated by 3^l. Justify this. Can you improve on the estimate in the text?

3. We have defined a *heuristic* search technique to be one that uses domain-specific information; a heuristic function-maximization technique is one that uses information about the function's current value in deciding where to look next. Why is simulated annealing included in this chapter as opposed to the previous one?

4. Describe the similarity between simulated annealing (viewed as a search procedure) and iterative broadening.

5. Suppose that h'_1 and h'_2 are two admissible heuristics in a search problem. Show that $\max(h'_1, h'_2)$ is also admissible. Give an example of a search problem and two admissible heuristic functions, neither of which is always less than the other.

6. What is the behavior of the A* algorithm if $h' = h$? Prove your claim.

7. Write a computer program that implements A* and uses it to find solutions to mazes. Use the program to solve the maze in Figure 4.2.

8. (a) Prove the following generalization of Theorem 4.2.2:

PROPOSITION 4.5.1 *In searching a tree, suppose that* g(n) *is a cost function that increases along the paths from the root node, so that whenever a node* n_1 *is a child of a node* n_2, *then* $g(n_1) \geq g(n_2)$. *Then if* $h'(n) \leq h(n)$ *for every node* n, *any value returned by the A* algorithm will be a goal node for which* g *is minimal.*

(b) Why did we need to add to the proposition the condition that the A* algorithm return an answer?

9. Prove the following generalization of Theorem 4.2.2:

PROPOSITION 4.5.2 *Suppose that although the distance-estimating function* h′ *is not admissible, it is nearly admissible in the sense that there is some small constant* ϵ *such that*

$$h'(n) \leq h(n) + \epsilon$$

for every node n. *The value returned by the A* algorithm will be a goal node for which* g *is within* ϵ *of minimal.*

10. (a) Prove that Procedure 4.3.1 continues to find optimal solutions to search problems.

(b) How should the procedure be modified if we want to maintain a list of closed nodes as well as a list of open ones?

11. In Exercise 5 in Chapter 3, we defined a search ordering to be depth-first legal if it corresponded to depth-first search. Similarly, for a fixed choice of heuristic function h' we will define a search ordering to be

A legal* if and only if it matches the A* algorithm for this choice of heuristic function. We will say that depth-first search and A* search are *equivalent* if and only if every depth-first legal search ordering is A* legal and vice versa.

Prove that in a domain where every arc is of unit cost, there is a nonnegative heuristic function h' such that depth-first search and A* search are equivalent if the domain being considered has no paths of infinite length and no single node that can be reached by two paths of different lengths. (Note that we do not require the heuristic to be admissible.)

12. This exercise examines the choice between two admissible heuristic functions, one of which is always less than the other.

    Suppose that h_1' and h_2' are two admissible heuristic functions for some domain, and that $h_1'(n) \leq h_2'(n)$ for every node n.

    (a) Let S be a search ordering that is A* legal using the heuristic h_1', and suppose that exploring the search space using this ordering involves expanding a set N of nodes. Show that there is some search ordering that is A* legal using the heuristic h_2' that examines a subset of N in solving the problem in question. This shows that it is always *possible* that h_2' leads to a smaller search space than h_1' does.

    (b) Find a search problem and two admissible heuristics h_1' and h_2' with $h_1' \leq h_2'$ such that although it is possible that h_2' examines fewer nodes that h_1' does, it is far more likely that h_1' leads to the more efficient search.

13. In what order does IDA* examine the search space associated with the maze appearing in Figure 4.2?

14. Suppose that we modify IDA* as follows:

PROCEDURE 4.5.3

1. *Set* c = 1; *this is the current depth cutoff.*
2. *Set* L *to be a list of the initial nodes in the problem.*
3. *Let* n *be the first node on* L. *If* L *is empty, increment* c *and return to step 2.*
4. *If* n *is a goal node, stop and return it and the path from the initial node to* n.
5. *Otherwise, remove* n *from* L. *Add to the front of* L *every child* n′ *of* n *for which* f(n′) ≤ c. *Return to step 3.*

Why is the original Procedure 4.3.2 to be preferred?

CHAPTER

5

ADVERSARY SEARCH

This chapter concludes our discussion of search by examining the specific search issues that arise when analyzing game trees.

5.1 ASSUMPTIONS

AI research on game playing typically considers only games that have two specific properties: They are two-person games in which the players alternate moves, and they are games of perfect information, where the knowledge available to each player is the same. There are exceptions to each of these constraints, but most of the work does indeed make these assumptions.

Typical two-person games investigated by AI researchers are tic-tac-toe, checkers, chess, Go, Othello, and backgammon. Other than perhaps Go and Othello, these are probably familiar to you.

Go is played on a 19×19 board with stones of two colors; the object is to place your stones so that they surround those of your opponent. From an AI point of view, it is remarkable for two reasons:

1. The difference between two Go players can be measured quite precisely. A Go player who beats another by a certain margin in one game is very likely to beat him by a nearly identical margin in subsequent games.
2. Computer Go players are currently quite weak. The branching factor is large; perhaps more importantly, Go is a very "positional" game. The best move is (it seems) selected more on the basis of how the position "looks" than on intricate tactical analysis. In those cases where tactical analysis is required, the line that is analyzed is typically of very low branching factor (perhaps $b = 1$) and high depth (ten or more moves by each player).

Othello is played on a checkerboard. The pieces used are black on one

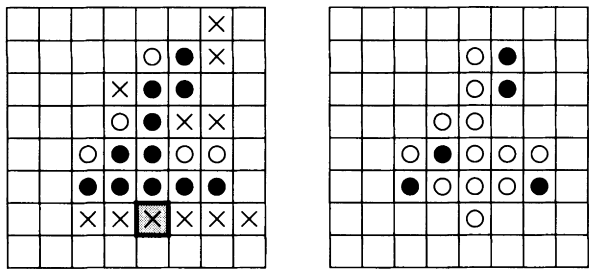


FIGURE 5.1
An Othello position

side and white on the other; a move for White is to place a piece so that a white piece is on each end of some line of Black's pieces. All such black pieces are then flipped and become White's; the object is to end the game with as many pieces of your color as possible. A typical Othello position is shown in Figure 5.1; White's legal moves are denoted by X's. The result if White moves to the square shown in boldface is depicted in the second part of the figure.

Computers are very good at Othello; it is a game that rewards the ability to conduct a brute-force search through one's alternatives and also a game for which it seems to be impossible to recognize a good position at a glance. In fact, computers are significantly better at Othello than people are.

The second assumption made in game search is that the game being considered is one of perfect information. This means that the information available to each of the two players is identical—there is nothing one player knows that the other player could not in principal also know.

Chess is a typical game of perfect information; the differences in beliefs between the two players reflect differing analyses of the position in which they find themselves—not a difference in their raw knowledge about where the pieces are located. Two chess players who disagree about the merits of some particular move will resolve the difference by analyzing the resulting position and not by revealing secrets to one another.

Poker is a classic example of a game of *imperfect* information. Here, one settles a dispute by revealing the contents of one's hand—information that is presumably not available to one's opponent.

Stratego is another game of imperfect information. Each player in this game has an army of pieces of various ranks that attack one another; in each individual battle, the piece of higher rank wins. Although both players know the *locations* of each other's pieces, only blue (for example) knows which blue pieces are of what rank.

In a game such as this, it is often sensible to attack a piece of unknown strength with a weak one; this is not because you expect to win the battle but because you can learn the strength of the opposing piece in this way. This is a typical feature of games of imperfect information—one can expend one's limited resources in order to improve the quality of the information available.

(This sort of activity occurs all the time, not just when playing games. When you stop to buy a paper to find out where some movie is playing, you are expending limited resources—time and money—to improve your knowledge about the behavior of the local theaters. And occasionally—just occasionally—stopping to get the paper will mean that you don't have time to get to the movie; time really is a limited resource that you are expending in order to extend your knowledge in some way.)

Computers make quite good poker players, incidentally. Poker appears to be mostly a matter of working out the probabilities and bluffing with a straight face.

Bridge is another game of imperfect information that has attracted some interest in the AI community. Although there is substantial commercial motivation for developing a competent automated bridge player, the results so far have been disappointing. The reason, it seems, is that bridge is somewhat like Go in that correct bidding and play often hinge (at least initially) upon the sort of pattern recognition that computers find difficult.

The last game of imperfect information that I will mention is called Diplomacy. This is a game where the object is to take over the world by first negotiating treaties with one's fellow players and then breaking them at a suitable moment. Interestingly enough, computers make excellent Diplomacy players once a suitable negotiation language has been developed.

5.2 MINIMAX

Let us return our attention to two-person games of perfect information. We have drawn such a game in Figure 5.2, where we have depicted the game as a tree where the root node is the starting position and the terminal nodes are the ending positions. These terminal nodes are labelled with either 1 or -1 depending on which of the players wins the game.

In a game such as this, one of the players will be trying to get to a node labelled -1, and the other will be trying to get to a node labelled 1. We will call the player trying to achieve an outcome of -1 the *minimizer*, and the other player the *maximizer*. We will assume that it is the maximizer's turn to move in the starting position of Figure 5.2, which corresponds to the root node a.

What is the value of the node o in this position? It's the maximizer's turn, and whatever move he makes will end the game. He can move to either p or r and lose, or he can move to q and win. Assuming that he is playing sensibly, he will obviously choose q. This means that the node o is in fact a win for the maximizer, and can be labelled with 1.

What about node k? Whatever the minimizer does, the maximizer will win—either immediately if the minimizer moves to node n, or in one move if he moves to o. So we can label k with a 1 as well. The node i is different,

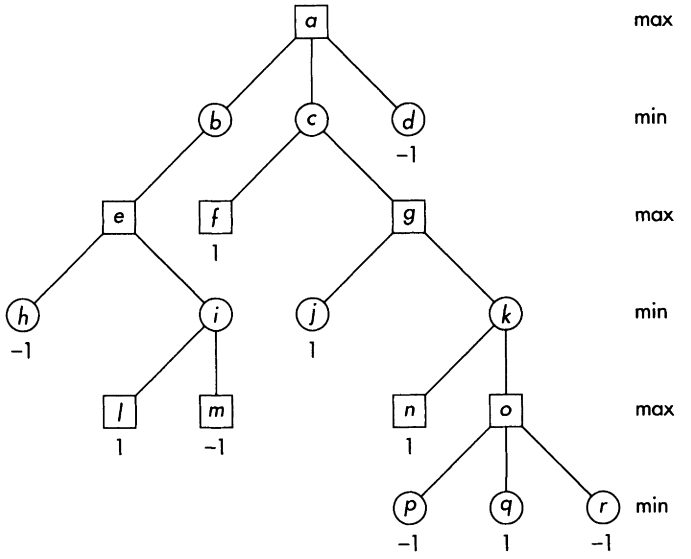


FIGURE 5.2
A simple game tree. Nodes with the maximizer to move are squares; nodes with the minimizer to move are circles.

since here the minimizer has a winning option available (*m*). So this node should be labelled with a -1.

The algorithm for backing the values up the trees is now apparent. A node with the maximizer to move should be labelled with the *maximum* value that labels any of its children; a node with the minimizer to move should be labelled with the minimum of its children's labels. We have done this in Figure 5.3, and we see from the fact that the starting position is

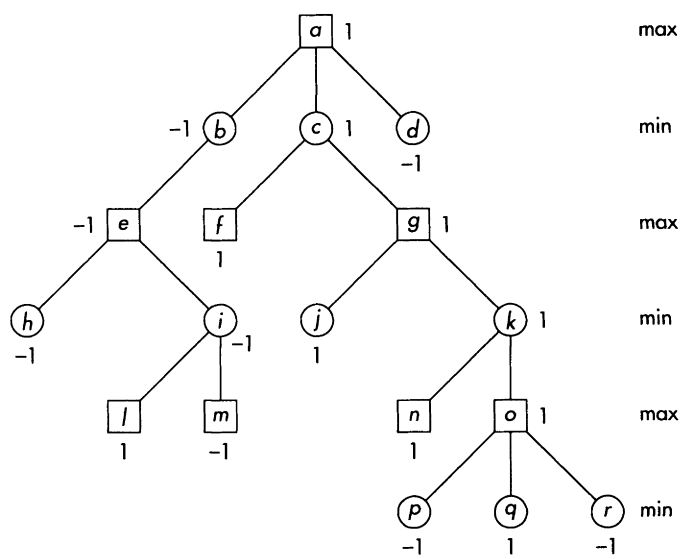


FIGURE 5.3
The maximizer wins

labelled with a 1 that perfect play will win this game for the maximizer. This technique is known as *minimax*.

Before proceeding, suppose that we had continued to back up from the node *k*, determining that g and then *c* had the values 1. We can now stop our analysis and conclude immediately that the value to be assigned to *a* is also 1! The reason is that we know that the maximizer can win from node *a* by moving to node *c*; there is no reason to look for another alternative to this winning line. "Real" games behave similarly; once we find a winning move in a chess game, we typically make that move without examining the alternatives.

Ignoring this possible enhancement, however, here is the basic algorithm for determining the values to be assigned to internal nodes in a game tree.

PROCEDURE 5.2.1 **Minimax** *To evaluate a node* n *in a game tree:*

1. *Expand the entire tree below* n.
2. *Evaluate the terminal nodes as wins for the minimizer or maximizer.*
3. *Select an unlabelled node all of whose children have been assigned values. If there is no such node, return the value assigned to the node* n.
4. *If the selected node is one at which the minimizer moves, assign it a value that is the minimum of the values of its children. If it is a maximizing node, assign it a value that is the maximum of the children's values. Return to step 3.*

If our game could end in a draw, this would correspond to terminal positions labelled 0 instead of 1 or -1. Draws now fit neatly into the minimax formalism, since each player will pick a winning move if possible and search for a draw if no win exists.

As a matter of terminology, by a *move* in a game tree we will mean a pair of individual actions, one for each player; this is at odds with common usage. An action by only one player is typically called a half-move or a *ply*. Thus the depth of the tree in Figure 5.3 is three moves, or 6 ply.

How much memory and time are needed by the minimax algorithm in Procedure 5.2.1? Since the entire tree needs to be expanded, we can expect this procedure to need an exponential amount of space, as the whole fringe apparently needs to be stored before the values are backed up. A moment's thought reveals, however, that the space needed can be reduced to an amount linear in the depth by searching in a depth-first instead of a breadth-first fashion. Thus in Figure 5.3, we would back the value -1 up to node *b* before expanding the children of node *c*, and so on. This produces the following result, where we update the values assigned to internal nodes as values are assigned to their children:

PROCEDURE 5.2.2 **Minimax** *To evaluate a node* n *in a game tree:*

1. *Set* L = {n}, *the unexpanded nodes in the tree.*
2. *Let* x *be the first node on* L. *If* x = n *and there is a value assigned to it, return this value.*
3. *If* x *has been assigned a value* v_x, *let* p *be the parent of* x *and* v_p *the value currently assigned to* p. *If* p *is a minimizing node, set* $v_p = \min(v_p, v_x)$. *If* p *is a maximizing node, set* $v_p = \max(v_p, v_x)$. *Remove* x *from* L *and return to step 2.*
4. *If* x *has not been assigned a value and is a terminal node, assign it the value 1 or* -1 *depending on whether it is a win for the maximizer or minimizer respectively. Assign* x *the value 0 if the position is a draw. Leave* x *on* L (*we still have to deal with its parent*) *and return to step 2.*
5. *If* x *has not been assigned a value and is a nonterminal node, set* v_x *to be* $-\infty$ *if* x *is a maximizing node and* $+\infty$ *if* x *is a minimizing node. (This is to make sure that the first minimization or maximization in step 3 is meaningful.) Add the children of* x *to the front of* L *and return to step 2.*

Unfortunately, Procedure 5.2.2 continues to use an exponential amount of *time* to determine the value that is assigned to the node *n*. In general, it is simply impractical to expand the whole tree; as an example, we have already remarked in Chapter 1 that the chess tree contains some 10^{160} nodes.[11]

How do people manage to play a game like chess? We don't analyze it all the way to the end; we just look far enough ahead so that we can estimate who's likely to win in some nonterminal position and then back up the values from the nonterminal positions we have found. Minimax still applies; we just apply it to internal nodes in the tree.

In order to do this, we need a way to assign a value to these internal nodes, so that given a node *n* we can label it with some estimated value *e*(*n*). If *e*(*n*) = 1, then we believe absolutely that the node is a win for the maximizer; if *e*(*n*) = -1, the node is believed to be a win for the minimizer. Intermediate values reflect lower levels of certainty; presumably *e*(*n*) = 0 is used to label a position in which neither side is perceived to have an advantage, just as *e*(*n*) = 0 is used to label a *terminal* node that is a draw (that is, a terminal node in which neither player has an advantage).

Crude evaluation functions are often easy to describe. In chess, for example, one can simply add up the values of the pieces each player has (where a pawn counts as 1, a knight 3, a rook 5, and so on according to

11 The *graph* of chess positions is somewhat smaller because many positions can be reached in a variety of ways. It's still intractably large, though.

conventional chess wisdom) and then normalize the result so that a value between -1 and 1 is returned. If the total value of white's pieces is w and the value of blacks is b, we might use

$$\frac{w - b}{w + b} \tag{5.1}$$

as the value to be assigned to the overall position.

There are, however, other features that can be included. In chess, a plan of attack for one player often involves targeting the opponent's pawns. The most natural way to defend a pawn is by using another pawn as shown in Figure 5.4; the white pawn on square a4 defends the one on b5.

In some cases, however, it is impossible to use one pawn to defend another. The white pawn on square d3 in the figure cannot be defended by another pawn, since there are no white pawns on either of the two neighboring files (labelled with c and e in the figure). A pawn with no pawns on adjacent files is called *isolated*, and is a positional flaw whose value has been estimated at $-1/3$ of a pawn.

Positional features abound in chess. How well protected is your king? How much room do you have in which to maneuver? Do you control the center of the board? And so on.

One interesting thing about these positional features is that evaluating them invariably involves more than simply locating a single one of your pieces on the board. Finding your bishops is a matter of simply that—finding them. But finding your isolated pawns involves finding your pawns and then analyzing their positions relative to your other pawns. As a result, including these positional features in the evaluation function makes $e(n)$ more expensive to evaluate, and time spent computing $e(n)$ for the various nodes in the tree is time that cannot be spent searching the tree itself. Here is the base-level/metalevel trade-off again.

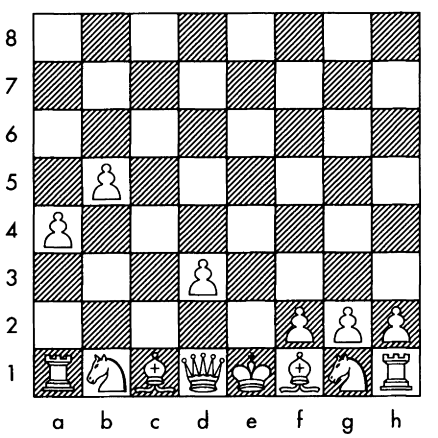


FIGURE 5.4 A pawn defending another

Suppose, however, that we have selected some evaluation function $e(n)$. The modified version of Procedure 5.2.2 is now:

PROCEDURE 5.2.3 **Minimax** *To evaluate a node* n *in a game tree:*

1. *Set* $L = \{n\}$.
2. *Let* x *be the first node on* L. *If* $x = n$ *and there is a value assigned to it, return this value.*
3. *If* x *has been assigned a value* v_x, *let* p *be the parent of* x *and* v_p *the value currently assigned to* p. *If* p *is a minimizing node, set* $v_p = \min(v_p, v_x)$. *If* p *is a maximizing node, set* $v_p = \max(v_p, v_x)$. *Remove* x *from* L *and return to step 2.*
4. *If* x *has not been assigned a value and* **either** x **is a terminal node or we have decided not to expand the tree further**, *compute its value using the evaluation function. Return to step 2.*
5. *Otherwise, set* v_x *to be* $-\infty$ *if* x *is a maximizing node and* $+\infty$ *if* x *is a minimizing node. Add the children of* x *to the front of* L *and return to step 2.*

The interesting new question is one that is implicit in step 4 of this procedure—how do we select the portion of the search tree to expand? Note that this is an issue only because we expect our evaluation function to be imperfect—we cannot typically look at a chess position and decide who will win without doing some amount of analysis. If our evaluation function *were* somehow perfect, we could simply evaluate the children of the root node and decide from that what to do!

The easiest approach to our computational difficulties is to expand the search to a constant depth, say p ply. The advantage of this is that it's simple—once again, any metalevel effort devoted to intricate decisions about which nodes to expand must be recovered in terms of more effective base-level search. The constant-depth approach is computationally very efficient, but there are several serious problems with it.

5.2.1 Quiescence and Singular Extensions

The first problem is that some portions of a game tree may well be "hotter" than others. A move that leads to tactical considerations involving lengthy exchanges of pieces should probably be investigated in more depth than one that leads to quieter positions. The obvious solution here is to search the tactical portions of the tree to greater depth than the quiet ones, but how are we to recognize these tactical portions automatically?

Tactical portions of the search space are characterized by rapidly changing values of the heuristic evaluation function $e(n)$. In some sense, this is what it *means* for a position to be tactical—single moves by each player drastically affect the apparent value of the position.

This also makes it clear why these portions of the space should be searched to greater depth. Since $e(n)$ is changing rapidly from one ply to the next, the value of $e(n)$ at any particular point is likely to be fairly unreliable; if the tree can be searched to a depth at which $e(n)$ is changing only slowly from ply to ply, this value is more likely to be accurate. The technical term *quiescence* is often used to describe the attempt to search to a depth at which the game becomes fairly quiet.

A related idea that has been implemented to address this concern is that of a *singular extension*. Here, the assumption is made that a particular node should be evaluated to greater depth if one of the opponent's moves leads to a result vastly preferable to him than all other options. The idea is that since this move is "forced" in some sense, the effective branching factor is small, and an extra ply or so of search will be computationally practical. Experimentation has shown that this idea can substantially improve the performance of chess-playing programs.

5.2.2 The Horizon Effect

The second general difficulty with searching a game tree to a fixed depth is known as the *horizon effect*, and is best illustrated by an example.

Suppose that we are searching a chess tree to a depth of 7 ply, and that our opponent has a threat that manifests itself at exactly this depth. We would expect that if we do not respond to the threat, our 7-ply search will reveal the problem and we will therefore be led to a move that addresses it.

But suppose that in the middle of the 7-ply combination, we have some useless move that serves no real purpose other than demanding a response from our opponent. Perhaps we throw in a "nuisance" check, forcing a king move before the rest of the combination can be executed.

The nuisance move hasn't really helped us in any way; in general, such maneuvers tend to make our position worse rather than better. But what the nuisance move *has* done is to make the length of our opponent's combination 9 ply instead of 7. As a result, our 7-ply search will fail to detect the threat when examining the line with the nuisance move in it, and will therefore conclude that the move defeats our opponent's threat! What we have done is to push the culmination of the threat over the search horizon, and mistakenly confused this with the presumably more effective (and necessary) option that deals with the threat instead.

This is a very subtle problem, since it isn't clear how we can distinguish between a move that really *does* neutralize the threat and one that only appears to do so by pushing the threat past the search horizon. The difference is that in the second case, the threat reappears two ply later; in the first case, it doesn't. But determining this involves a search to depth 9, not 7.

(People, of course, deal with this problem by realizing what's going on and determining whether the threat has been addressed or not. But people know what they are doing when they play games; computers are just searching through partial position trees.)

There have been two solutions proposed to the horizon problem; neither is really satisfactory, but we will discuss them both.

Secondary search One proposed solution is to examine the search space beneath the apparently best move to see if something has in fact been pushed just beyond the horizon so that an alternate move should be chosen.

Unfortunately, although this technique can be used to detect the horizon effect, it doesn't really tell us what to do about it. Consider the chess situation again. The move that actually addresses the threat may well appear to be a fairly weak one, since responding to our opponent's threat means that we won't be able to pursue our own intentions.

As a result, if we manage to use this sort of a secondary search to determine that the horizon effect invalidates the apparently best move at a depth of 7 ply, the second-best move is likely to fall prey to the same problem. The move that actually addresses the difficulty is not likely to be expanded until quite late in the search, and there will not be sufficient time to perform secondary searches beneath *all* of these other moves.

The killer heuristic The second idea that can help with the horizon problem is known as the *killer heuristic*. What this suggests is that if you find a move that is good for your opponent, you look at this move early when considering your opponent's options.

This can be combined with the ideas of the previous paragraph as follows: When the apparently best move is found, a secondary search is performed to check it for possible flaws. Let's say that such a flaw is found; our opponent can defeat our 7-ply plan with a particular move at ply 8. We can now do *partial* secondary searches beneath our other alternatives to determine whether or not these alternatives suffer from the same difficulty. By constraining the size of all of the secondary searches except one, some instances of the horizon problem can be avoided.

5.3 α-β SEARCH

The real problem with adversary search, however, is the one that we referred to in our parenthetical remark in the previous section: Computers, playing games, are simply searching through large trees looking for nodes with certain mathematical properties. As a result, most of the nodes examined in the search for a move have no bearing on the course of the game—they are at best pointless and at worst suicidal. Chess-playing programs consider irrelevant pawn moves and meaningless piece sacrifices with the same care that they consider winning combinations or careful positional

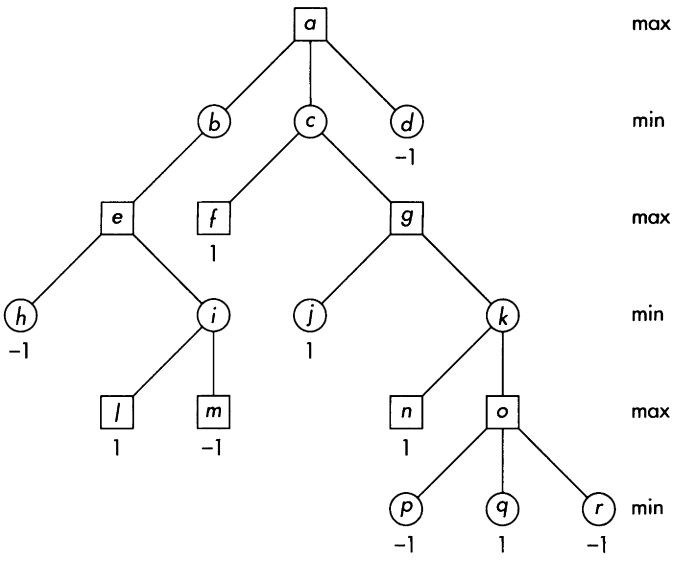


FIGURE 5.5
α-β search: Prune the nodes below *b*

improvements. If we want to improve the performance of these programs, we need a way to reduce the size of the search space.

The most powerful technique known for doing this is called α-β *search*. We've already seen an example of it in Figure 5.3, which we repeat here as Figure 5.5.

What we noticed about the search in Figure 5.5 is that once we realize that the maximizer can win by moving to *c*, we no longer need to analyze any of his other options from *a*. Specifically, the values assigned to all of the nodes under the node *b* are guaranteed not to affect the overall value assigned to the position *a*.

Another example appears in Figure 5.6. Here, we are considering our option *c* of attacking our opponent's queen and are assuming as usual that we are the maximizing player.

If we do attack our opponent's queen, we can be mated at the next move, thereby ending the game and achieving an outcome of −1 (that is, we lose). Clearly, there is no need to consider any of *c*'s other children—we've already seen enough to realize that we should not play *c*!

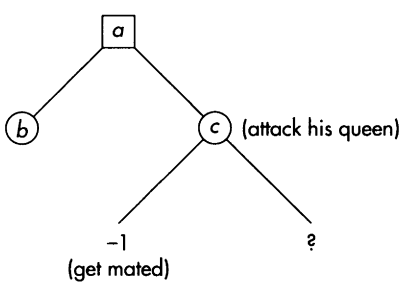


FIGURE 5.6
Another example of α-β search

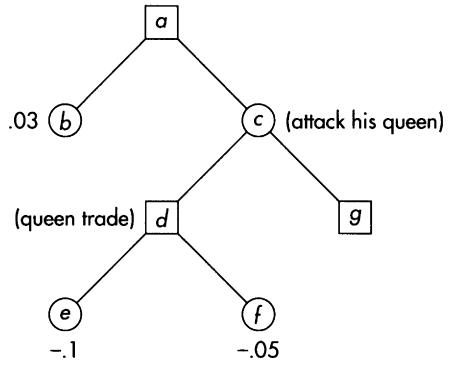


FIGURE 5.7
Another example of α-β search

A slightly more complicated example appears in Figure 5.7. Here, suppose that we are analyzing the tree in a depth-first manner and have determined that the value to be assigned to node b is .03, slightly favorable to us. We continue by examining our other option c, attacking our opponent's queen with our own.

The first response we consider is d, involving an exchange of queens. Further examination of d's children e and f show that f is the better of the two moves, and leads to a backed-up value of $-.05$ for the node d.

Before examining node g, suppose that we stop to take stock. Is c ever going to be our final selection? If we move to c, the *best* result we can expect to obtain is $-.05$, since that's how well we'll do if our opponent replies with d. If g is better for our opponent than d is, the value to be assigned to c will be even *lower* than $-.05$. Why should we accept such a value when we can obtain the outcome of .03 by moving to node b?

What we have shown, in effect, is that the node c is never on the *main line*, which is the course the game would take if both players played optimally. Since b is better for the maximizer than c is and the maximizer can select between them, c is guaranteed to be avoided. The values assigned to g and the nodes under it can never impact the final value assigned to the node a, and the subtree under g can be pruned.

We can reach the same conclusion algebraically. If we denote by g the backed-up value assigned to the node g, then the value assigned to the node c will be

$$c = \min(-.05, g)$$

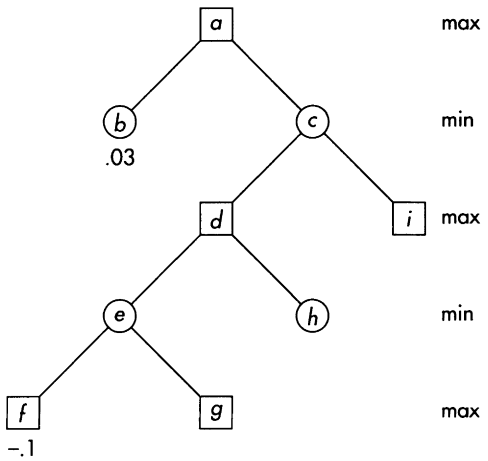
since it is the minimizer who will choose between the alternatives d and g.

Continuing, the value assigned to the root node a is

$$a = \max[.03, \min(-.05, g)] = .03 \qquad (5.2)$$

where the second equality holds because $\min(-.05, g) \leq -.05 < .03$. Since

FIGURE 5.8
A deep α-β cutoff



the value assigned to the node *a* is independent of the value g, we see that the nodes below g can be pruned.

A still more complicated example appears in Figure 5.8. Here, the values assigned to *b* and to *f* are .03 and $-.1$ respectively. What can we say about the children of the node g? Can it be shown that g is not on the main line?

Well, if g is going to be on the main line, then its ancestors will need to be as well, so that *e* will be on the main line. But from *e* the minimizer has the opportunity to move to node *f* and obtain a payoff of $-.1$. Since this is worse than the payoff of .03 that the maximizer can obtain by moving to *b*, it follows that g cannot be on the main line. Note that it is still possible that *h* is on the main line, or that *i* is—all we can say for sure is that g is *not*. A situation such as this, where the pruned node is more than one ply below the reason for the pruning, is typically referred to as *deep* α-β pruning.

What about the general case? Suppose that *n* is some node in a game tree at which the maximizer gets to move (like the node g in Figure 5.8), and that *s* is a sibling of *n* with backed-up value v_s (*s* would be the node *f* in the above figure, so that $v_s = -.1$). Suppose also that $\langle p_0, p_1, \ldots, p_k \rangle$ is the path from the root node down to *n*, with p_0 the root node and $p_k = n$. Now note that the nodes with odd indexes (p_1, p_3, and so on) are minimizing nodes; if any of these nodes has a sibling with backed-up value greater than v_s, then the node *n* can be pruned. (In the figure, $p_1 = c$ has the sibling *b* with backed-up value $.03 > v_s$.) Here is another way to put it.

PROCEDURE 5.3.1

α-β search *To evaluate a node* n *in a game tree:*

1. *Set* L = {n}.
2. *Let* x *be the first node on* L. *If* x = n *and there is a value assigned to it, return this value.*

3. *If* x *has been assigned a value* v_x, *let* p *be the parent of* x; *if* x *has not been assigned a value, go to step* 5. *We first determine whether or not* p *and its children can be pruned from the tree: If* p *is a minimizing node, let* α *be the maximum of all the current values assigned to siblings of* p *and of the minimizing nodes that are ancestors of* p. *(If there are no such values, set* $\alpha = -\infty$.*) If* $v_x \leq \alpha$, *remove* p *and all of its descendants from* L. *If* p *is a maximizing node, treat it similarly.*

4. *If* p *cannot be pruned, let* v_p *be the value currently assigned to* p. *If* p *is a minimizing node, set* $v_p = min(v_p, v_x)$. *If* p *is a maximizing node, set* $v_p = max(v_p, v_x)$. *Remove* x *from* L *and return to step 2.*

5. *If* x *has not been assigned a value and is either a terminal node or we have decided not to expand the tree further, compute its value using the evaluation function. Return to step 2.*

6. *Otherwise, set* v_x *to be* $-\infty$ *if* x *is a maximizing node and* $+\infty$ *if* x *is a minimizing node. Add all of the children of* x *to the front of* L *and return to step 2.*

The value corresponding to α but computed by considering p's maximizing ancestors is called β; the two values lead to α and β cutoffs respectively. Because of this, this method of reducing the size of the search tree is called α-β *pruning*. It's a pretty dumb name.

All of this is well and good, but what does it buy us? Although we have seen that α-β pruning can reduce the size of the search space associated with a game tree, we haven't discussed by how *much* the search space is reduced.

It is clear, for example, that in the worst case it is possible that α-β pruning fails to reduce the size of the search space at all. If we perversely order the children of every node so that the worst options are evaluated first, then the nodes examined later will always be the "main line" and will therefore never be pruned.

What about the best case? What if we somehow manage to order the nodes so that the best moves are examined first? We will still need to search the space to confirm that these actually *are* the best moves, but now α-β pruning can save us a great deal of work.

How much is saved? Suppose that we consider a response for the minimizer that is off the main line. In order to prune it, we need to examine just enough of the search space to demonstrate that it is a mistake for the minimizer—in other words, we have to examine the "refutation" of this move that shows how the maximizer can exploit it. In order to do this, we have to examine only one response on the maximizer's part—the best one. Then we have to examine all of the minimizer's options, the maximizer's best response to each, and so on. So although the branching factor for the

minimizer is unchanged, the branching factor for the maximizer is reduced to just 1.

If the maximizer deviates, the analysis is similar, but with the roles of the two players reversed. It follows that the *total* number of nodes that we need to examine to depth d is approximately

$$b^{d/2} + b^{d/2} \tag{5.3}$$

where b is the branching factor in the game. The two terms correspond to analysis of the situations where the maximizer and minimizer deviate respectively. In each case, the search to depth d involves $d/2$ nodes with b children and $d/2$ nodes with only one child. Thus the size of the associated search space is $b^{d/2}$ and the expression (5.3) follows.

Instead of searching a space of size b^d, we need to search one of size $2b^{d/2}$; this is potentially a tremendous savings. It is frequently described in terms of an "effective" branching factor, the branching factor b' such that b'^d is the size of the space searched. In this case, we have

$$b'^d = 2b^{d/2}$$

leading to

$$b' = 2^{1/d}\sqrt{b} \approx \sqrt{b}$$

so that the effective branching factor is approximately the square root of the actual branching factor. Yet another way to look at this is to realize that in this best case, α-β pruning allows us effectively to double the depth to which we can search in a fixed amount of time.

These observations mean that it is very important when using α-β search to do a good job of ordering the children of any particular node. A variety of methods exist for this—the evaluation function can be applied to the internal nodes to order them heuristically, the killer heuristic discussed in Section 5.2.2 can be used to move nodes that work well elsewhere to the front of the list of children, and so on.

In practice, this turns out to be an effective way to approximately order the internal nodes when searching a game tree using α-β pruning. Sophisticated chess programs, for example, typically investigate only 1½ times the theoretically minimal number of nodes using this method. We can come quite close to the theoretical limits described above.

5.4 FURTHER READING

The world's best Othello player is a computer program called BILL [Lee and Mahajan, 1990]. The Diplomacy player mentioned in the text is described in Kraus and Lehmann [1988].

Games that violate the typical AI assumptions by involving more than two players, simultaneous action, or including imperfect information are typically the focus of game theorists or economists and not computer scientists; a good introduction to game theory is Luce and Raiffa [1957]. Recently, some authors interested in *distributed* AI (the study of how our putatively intelligent artifacts will interact with one another) have observed that many game-theoretic results can be applied if these independent agents are viewed as players in a formal multiagent game. The work of Genesereth *et al.* [1986] and Rosenschein and Genesereth [1985], reprinted in Bond and Gasser [1988], is typical of this approach.

Singular extensions are introduced by Anantharaman *et al.* [1990] and are believed by many researchers to be the main reason that the chess program DEEP THOUGHT outperforms its predecessor, HITECH.

We remarked in the text that the techniques used to order the search in existing game-playing programs result in their behavior approximating that of the best-case analysis we have presented. As we have explained, this best case doubles the effective search depth; the worst case multiplies it by 1 (that is, leaves it unchanged). It is shown in Pearl [1982] that if the children of a node are randomly ordered, the effective search depth is multiplied by a factor of approximately 4/3.

5.5 EXERCISES

1. The work on game search assumes that deeper searches are more accurate than shallow ones, so that a search algorithm can improve its performance by searching to greater depth. Either prove this to be true or find a game and evaluation function for which it fails, in the sense that the result returned becomes uniformly less accurate as the search deepens. (Search to terminal nodes doesn't count, of course, since it always evaluates correctly.)

2. Consider the chess evaluation function given by (5.1).
   (a) Under what circumstances will this evaluation function take the values $+1$, -1, or 0?
   (b) What material value should be assigned to a king if this evaluation function is used?
   (c) The conventional wisdom in chess is that if you are a pawn ahead, you should try to exchange other pieces so that your extra pawn has a more substantial effect. Does (5.1) support this idea?

3. What might be a sensible heuristic evaluation function for checkers? Consider only the number of men and kings each side has.

4. What modifications should be made to Procedure 5.2.3 to take advantage of the fact that the space being searched may be a graph instead

of a tree? The hash table used to store the closed nodes in this case is usually called a *transposition table* because it is intended to deal with the fact that transposing moves often leads to identical positions in game search.

5. Suppose that generating the children of a node in a game tree takes time g, and that we are considering using one of two evaluation criteria, $e_1(n)$ or $e_2(n)$, that take times t_1 and t_2 to evaluate, respectively. If the branching factor of the tree is b and $t_1 < t_2$, under what conditions will the time needed to search to depth d using e_1 be the same as the time needed to search to depth $d-1$ using e_2? How about for α-β search, where both heuristics are nearly good enough to order the moves optimally? For which of these two approaches is it more important that the heuristic evaluation function be quick to evaluate?

6. Write a computer program that uses minimax to play a perfect game of tic-tac-toe. Modify your program to use α-β pruning as well.

7. Use Procedure 5.3.1 to show explicitly that the following nodes and their children can be pruned from the search space:
   (a) Node c in Figure 5.6.
   (b) Node c in Figure 5.7.
   (c) Node e in Figure 5.8.

8. Give an algebraic proof that the node g in Figure 5.8 can be pruned, similar to the derivation involving equation (5.2).

9. Use mathematical induction to prove that Procedure 5.3.1 never prunes a node that is needed to compute the value of the root node.

10. Give a "real" example of a deep α-β prune in a game such as chess.

11. Construct a search tree in which the size of the search space can be reduced by applying α-β pruning to both the maximizer and the minimizer.

12. Describe explicitly the way in which p should be treated in step 3 of Procedure 5.3.1 if it is a maximizing node.

13. Suppose that we are using α-β pruning to determine the value of a move in a game tree, and that we have decided that a particular node n and its children can be pruned. If the game tree is in fact a graph and there is another path to the node n, are we still justified in pruning this node? Either prove that we are or construct an example that shows that we cannot.

14. Suppose that instead of labelling a node with a single value, we label it with a *pair* of values $(a, -a)$ that sum to 0. The first value is the

"payoff" to the first player, and the second is the payoff to the second player.

(a) How should minimax be modified to deal with the new labels?

(b) How can these labels be extended to deal with three-player games where the order in which the players move is fixed?

(c) Is there an analog to shallow α-β pruning in three-player games? How about deep α-β pruning?

15. Describe a search technique that might be called *iterative deepening* α-β *search*. Why might you expect this technique to expand fewer nodes than conventional α-β search?