

```

function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST(problem) applied to STATE[node] succeeds
      then return SOLUTION(node)
    fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

```

function EXPAND(*node*, *problem*) returns a set of nodes

```

successors ← the empty set
for each (action, result) in SUCCESSOR-FN[problem](STATE[node]) do
  s ← a new NODE
  STATE[s] ← result
  PARENT-NODE[s] ← node
  ACTION[s] ← action
  PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
  DEPTH[s] ← DEPTH[node] + 1
  add s to successors
return successors

```

Figure 3.9 The general tree-search algorithm. (Note that the *fringe* argument must be an empty queue, and the type of the queue will affect the order of the search.) The SOLUTION function returns the sequence of actions obtained by following parent pointers back to the root.

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state space graph, because the graph is viewed as an explicit data structure that is input to the search program. (The map of Romania is an example of this.) In AI, where the graph is represented implicitly by the initial state and successor function and is frequently infinite, complexity is expressed in terms of three quantities: b , the branching factor or maximum number of successors of any node; d , the depth of the shallowest goal node; and m , the maximum length of any path in the state space.

Time is often measured in terms of the number of nodes generated⁵ during the search, and space in terms of the maximum number of nodes stored in memory.

To assess the effectiveness of a search algorithm, we can consider just the search cost—which typically depends on the time complexity but can also include a term for memory usage—or we can use the total cost, which combines the search cost and the path cost of the solution found. For the problem of finding a route from Arad to Bucharest, the search cost

⁵ Some texts measure time in terms of the number of node expansions instead. The two measures differ by at most a factor of b . It seems to us that the execution time of a node expansion increases with the number of nodes generated in that expansion.

is the amount of time taken by the search and the solution cost is the total length of the path in kilometers. Thus, to compute the total cost, we have to add kilometers and milliseconds. There is no “official exchange rate” between the two, but it might be reasonable in this case to convert kilometers into milliseconds by using an estimate of the car’s average speed (because time is what the agent cares about). This enables the agent to find an optimal tradeoff point at which further computation to find a shorter path becomes counterproductive. The more general problem of tradeoffs between different goods will be taken up in Chapter 16.

3.4 UNINFORMED SEARCH STRATEGIES

This section covers five search strategies that come under the heading of uninformed search (also called **blind search**). The term means that they have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a nongol state. Strategies that know whether one non-goal state is “more promising” than another are called **informed search** or **heuristic search** strategies; they will be covered in Chapter 4. All search strategies are distinguished by the order in which nodes are expanded.

Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth-first search can be implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. In other words, calling TREE-SEARCH(*problem*, FIFO-QUEUE) results in a breadth-first search. The FIFO queue puts all newly generated successors at the end of the queue, which means that shallow nodes are expanded before deeper nodes. Figure 3.10 shows the progress of the search on a simple binary tree.

We will evaluate breadth-first search using the four criteria from the previous section. We can easily see that it is *complete*—if the shallowest goal node is at some finite depth d , breadth-first search will eventually find it after expanding all shallower nodes (provided the branching factor b is finite). The *shallowest* goal node is not necessarily the *optimal* one; technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. (For example, when all actions have the same cost.)

So far, the news about breadth-first search has been good. To see why it is not always the strategy of choice, we have to consider the amount of time and memory it takes to complete a search. To do this, we consider a hypothetical state space where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d . In the worst

case, we would expand all but the last node at level d (since the goal itself is not expanded), generating $b^{d+1} - b$ nodes at level $d + 1$. Then the total number of nodes generated is

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1}).$$

Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node. The space complexity is, therefore, the same as the time complexity (plus one node for the root).

Those who do complexity analysis are worried (or excited, if they like a challenge) by exponential complexity bounds such as $O(b^{d+1})$. Figure 3.11 shows why. It lists the time and memory required for a breadth-first search with branching factor $b = 10$, for various values of the solution depth d . The table assumes that 10,000 nodes can be generated per second and that a node requires 1000 bytes of storage. Many search problems fit roughly within these assumptions (give or take a factor of 100) when run on a modern personal computer.

There are two lessons to be learned from Figure 3.11. First, the memory requirements are a *bigger problem for breadth-first search than is the execution time*. 31 hours would not be too long to wait for the solution to an important problem of depth 8, but few computers have the terabyte of main memory it would take. Fortunately, there are other search strategies that require less memory.

The second lesson is that the time requirements are still a major factor. If your problem has a solution at depth 12, then (given our assumptions) it will take 35 years for breadth-first search (or indeed any uninformed search) to find it. In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances*.

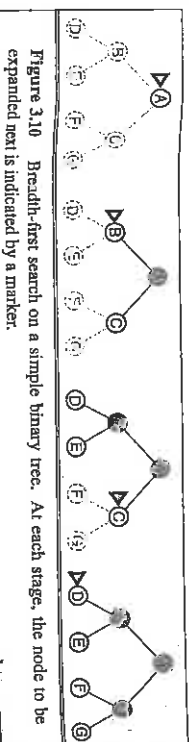


Figure 3.10 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

Uniform-cost search

Breadth-first search is optimal when all step costs are equal, because it always expands the *shallowest* unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost*. Note that if all step costs are equal, this is identical to breadth-first search.

Uniform-cost search does not care about the *number* of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if it ever expands a node that has a zero-cost action leading back to the same state (for example, a *NoOp* action). We can guarantee completeness provided the cost of every step is greater than or equal to some small positive constant ϵ . This condition is also sufficient to ensure *optimality*. It means that the cost of a path always increases as we go along the path. From this property, it is easy to see that the algorithm expands nodes in order of increasing path cost. Therefore, the first goal node selected for expansion is the optimal solution. (Remember that TREE-SEARCH applies the goal test only to the a nodes that are selected for expansion.) We recommend trying the algorithm out to find the shortest path to Bucharest.

Uniform-cost search is guided by path costs rather than depths, so its complexity cannot easily be characterized in terms of b and d . Instead, let C^* be the cost of the optimal solution, and assume that every action costs at least ϵ . Then the algorithm's worst-case time and space complexity is $O(b \cdot C^* / \epsilon)$, which can be much greater than b^d . This is because uniform-cost search can, and often does, explore large trees of small steps before exploring paths involving large and perhaps useful steps. When all step costs are equal, of course, $b \cdot C^* / \epsilon$ is just b^d .

Depth-first search

Depth-first search always expands the *deepest* node in the current fringe of the search tree. The progress of the search is illustrated in Figure 3.12. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack. As an alternative to the TREE-SEARCH implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn. (A recursive depth-first algorithm incorporating a depth limit is shown in Figure 3.13.)

Depth-first search has very modest memory requirements. It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. (See Figure 3.12.) For a state space with branching factor b and maximum depth m , depth-first search requires storage of only $b \cdot m + 1$ nodes. Using the same assumptions as Figure 3.11, and assuming that nodes at the same depth as the goal node have no successors, we find that depth-first search would require 118 kilobytes, instead of 10 petabytes at depth $d = 12$, a factor of 10 billion times less space.

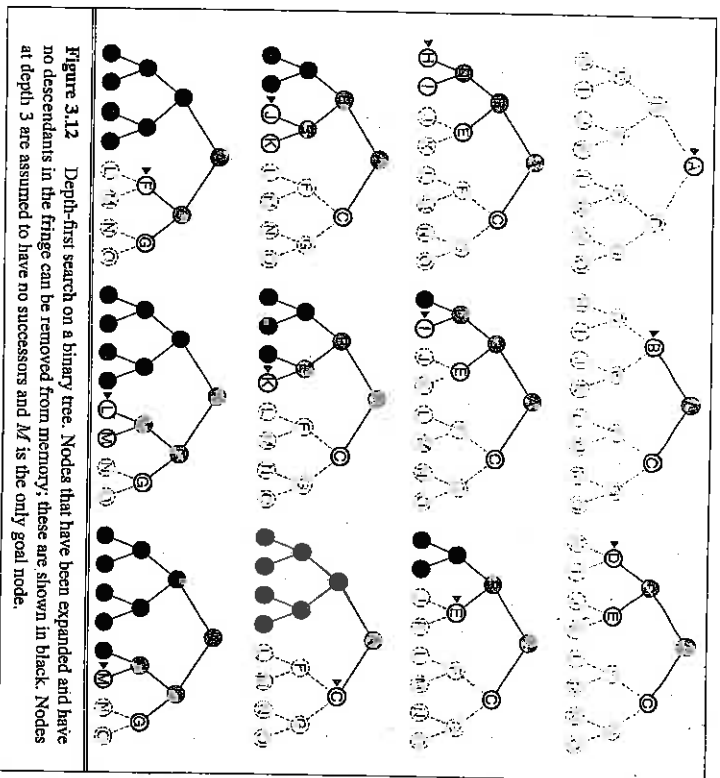


Figure 3.12 Depth-first search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

A variant of depth-first search called **backtracking search** uses still less memory. In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In this way, only $O(m)$ memory is needed rather than $O(mn)$. Backtracking search facilitates yet another memory-saving (and time-saving) trick: the idea of generating a successor by *modifying* the current state description directly rather than copying it first. This reduces the memory requirements to just one state description and $O(m)$ actions. For this to work, we must be able to undo each modification when we go back to generate the next successor. For problems with large state descriptions, such as robotic assembly, these techniques are critical to success.

The drawback of depth-first search is that it can make a wrong choice and get stuck going down a very long (or even infinite) path when a different choice would lead to a solution near the root of the search tree. For example, in Figure 3.12, depth-first search will explore the entire left subtree even if node C is a goal node. If node J were also a goal node, then depth-first search would return it as a solution; hence, depth-first search is not optimal. If

```

function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE(problem)), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  cutOffOccurred  $\leftarrow$  false
  if GOAL-TEST(problem)(STATE(node)) then return SOLUTION(node)
  else if DEPTH(node) = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutOffOccurred  $\leftarrow$  true
    else if result  $\neq$  failure then return result
  if cutOffOccurred? then return cutoff else return failure
  
```

Figure 3.13 A recursive implementation of depth-limited search.

the left subtree were of unbounded depth but contained no solutions, depth-first search would never terminate; hence, it is not complete. In the worst case, depth-first search will generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node. Note that m can be much larger than d (the depth of the shallowest solution), and is infinite if the tree is unbounded.

Depth-limited search

The problem of unbounded trees can be alleviated by supplying depth-first search with a predetermined depth limit ℓ . That is, nodes at depth ℓ are treated as if they have no successors. This approach is called **depth-limited search**. The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose $\ell < d$, that is, the shallowest goal is beyond the depth limit. (This is not unlikely when d is unknown.) Depth-limited search will also be nonoptimal if we choose $\ell > d$. Its time complexity is $O(b^\ell)$ and its space complexity is $O(\ell b)$. Depth-first search can be viewed as a special case of depth-limited search with $\ell = \infty$.

Sometimes, depth limits can be based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so $\ell = 19$ is a possible choice. But in fact if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 steps. This number, known as the diameter of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search. For most problems, however, we will not know a good depth limit until we have solved the problem.

Depth-limited search can be implemented as a simple modification to the general tree-search algorithm or to the recursive depth-first search algorithm. We show the pseudocode for recursive depth-limited search in Figure 3.13. Notice that depth-limited search can terminate with two kinds of failure: the standard *failure* value indicates no solution; the *cutoff* value indicates no solution within the depth limit.

Iterative deepening depth-first search

ITERATIVE
DEEPENING SEARCH

Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found. This will occur when the depth limit reaches d , the depth of the shallowest goal node. The algorithm is shown in Figure 3.14. Iterative deepening combines the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are very modest: $O(bd)$ to be precise. Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node. Figure 3.15 shows four iterations of ITERATIVE-DEEPENING-SEARCH on a binary search tree, where the solution is found on the fourth iteration.

Iterative deepening search may seem wasteful, because states are generated multiple times. It turns out this is not very costly. The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times. In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next to bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number of nodes generated is

$$N(\text{IDS}) = (d)b + (d-1)b^2 + \dots + (1)b^d,$$

which gives a time complexity of $O(b^d)$. We can compare this to the nodes generated by a breadth-first search:

$$N(\text{BFS}) = b + b^2 + \dots + b^d + (b^{d+1} - b).$$

Notice that breadth-first search generates some nodes at depth $d+1$, whereas iterative deepening does not. The result is that iterative deepening is actually *faster* than breadth-first search, despite the repeated generation of states. For example, if $b = 10$ and $d = 5$, the numbers are

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100.$$

In general, *iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.*

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
inputs: *problem*, a problem

```
for depth ← 0 to ∞ do
  result ← DEPTH-LIMITED-SEARCH(problem, depth)
  if result ≠ failure then return result
```

Figure 3.14. The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns failure, meaning that no solution exists.

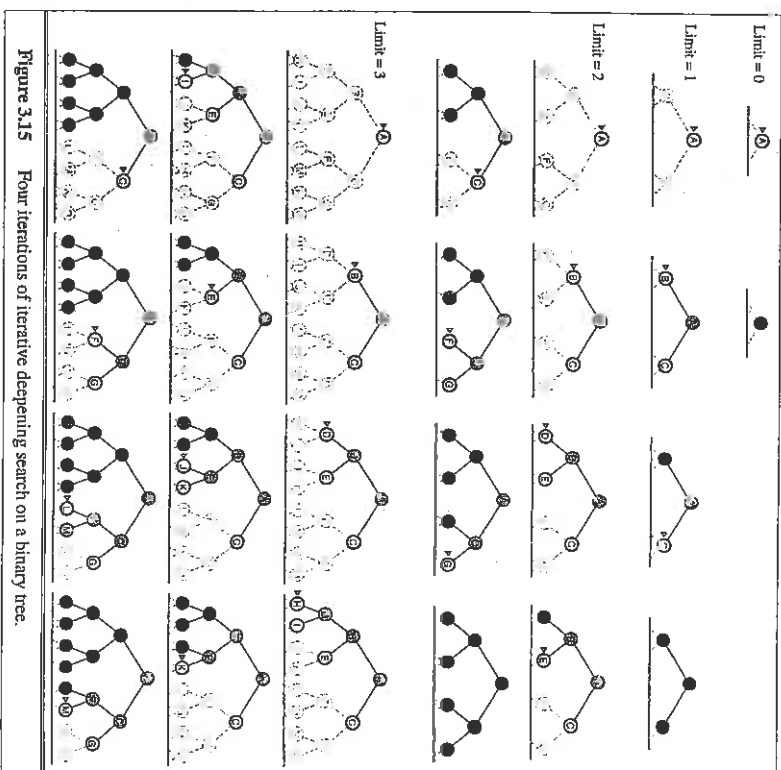


Figure 3.15 Four iterations of iterative deepening search on a binary tree.

Iterative deepening search is analogous to breadth-first search in that it explores a complete layer of new nodes at each iteration before going on to the next layer. It would seem worthwhile to develop an iterative analog to uniform-cost search, inheriting the latter algorithm's optimality guarantees while avoiding its memory requirements. The idea is to use increasing path-cost limits instead of increasing depth limits. The resulting algorithm, called *iterative lengthening search*, is explored in Exercise 3.11. It turns out, unfortunately, that iterative lengthening incurs substantial overhead compared to uniform-cost search.

Bidirectional search

The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal, stopping when the two searches meet

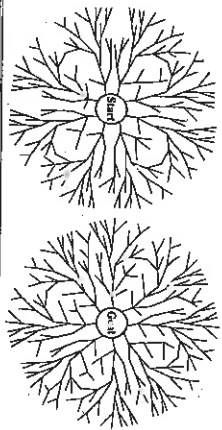


Figure 3.16 A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

in the middle (Figure 3.16). The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d , or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

Bidirectional search is implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree; if so, a solution has been found. For example, if a problem has solution depth $d = 6$, and each direction runs breadth-first search one node at a time, then in the worst case the two searches meet when each has expanded all but one of the nodes at depth 3. For $b = 10$, this means a total of 22,200 node generations, compared with 11,111,100 for a standard breadth-first search. Checking a node for membership in the other search tree can be done in constant time with a hash table, so the time complexity of bidirectional search is $O(b^{d/2})$. At least one of the search trees must be kept in memory so that the membership check can be done, hence the space complexity is also $O(b^{d/2})$. This space requirement is the most significant weakness of bidirectional search. The algorithm is complete and optimal (for uniform step costs) if both searches are breadth-first; other combinations may sacrifice completeness, optimality, or both.

The reduction in time complexity makes bidirectional search attractive, but how do we search backwards? This is not as easy as it sounds. Let the predecessors of a node n , $Pred(n)$, be all those nodes that have n as a successor. Bidirectional search requires that $Pred(n)$ be efficiently computable. The easiest case is when all the actions in the state space are reversible, so that $Pred(n) = Succ(n)$. Other cases may require substantial ingenuity.

Consider the question of what we mean by “the goal” in searching “backward from the goal.” For the 8-puzzle and for finding a route in Romania, there is just one goal state, so the backward search is very much like the forward search. If there are several explicitly listed goal states—for example, the two dirt-free goal states in Figure 3.3—then we can construct a new dummy goal state whose immediate predecessors are all the actual goal states. Alternatively, some redundant node generations can be avoided by viewing the set of goal states as a single state, each of whose predecessors is also a set of states—specifically, the set of states having a corresponding successor in the set of goal states. (See also Section 3.6.)

The most difficult case for bidirectional search is when the goal test gives only an implicit description of some possibly large set of goal states—for example, all the states satisfy-

ing the “checkmate” goal test in chess. A backward search would need to construct compact descriptions of “all states that lead to checkmate by move m_1 ” and so on; and those descriptions would have to be tested against the states generated by the forward search. There is no general way to do this efficiently.

Comparing uninformed search strategies

Figure 3.17 compares search strategies in terms of the four evaluation criteria set forth in Section 3.4.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes ^a	Yes ^{a,b}	No	No	Yes ^a	Yes ^{a,d}
Time	$O(b^{d+1})$	$O(b^{l^*} \epsilon^{1/l})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{l^*} \epsilon^{1/l})$	$O(bm)$	$O(b^l)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes ^c	Yes	No	No	Yes ^c	Yes ^{c,d}

Figure 3.17 Evaluation of search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^a complete if b is finite; ^b complete if step costs $\geq \epsilon$ for positive ϵ ; ^c optimal if step costs are all identical; ^d if both directions use breadth-first search.

3.5 AVOIDING REPEATED STATES

Up to this point, we have all but ignored one of the most important complications to the search process: the possibility of wasting time by expanding states that have already been encountered and expanded before. For some problems, this possibility never comes up; the state space is a tree and there is only one path to each state. The efficient formulation of the 8-queens problem (where each new queen is placed in the leftmost empty column) is efficient in large part because of this—each state can be reached only through one path. If we formulate the 8-queens problem so that a queen can be placed in any column, then each state with n queens can be reached by $n!$ different paths.

For some problems, repeated states are unavoidable. This includes all problems where the actions are reversible, such as route-finding problems and sliding-blocks puzzles. The search trees for these problems are infinite, but if we prune some of the repeated states, we can cut the search tree down to finite size, generating only the portion of the tree that spans the state-space graph. Considering just the search tree up to a fixed depth, it is easy to find cases where eliminating repeated states yields an exponential reduction in search cost. In the extreme case, a state space of size $d + 1$ (Figure 3.18(a)) becomes a tree with 2^d leaves (Figure 3.18(b)). A more realistic example is the rectangular grid as illustrated in Figure 3.18(c). On a grid, each state has four successors, so the search tree including repeated

states has 4^d leaves, but there are only about $2d^2$ distinct states within d steps of any given state. For $d = 20$, this means about a trillion nodes but only about 800 distinct states.

Repeated states, then, can cause a solvable problem to become unsolvable if the algorithm does not detect them. Detection usually means comparing the node about to be expanded to those that have been expanded already; if a match is found, then the algorithm has discovered two paths to the same state and can discard one of them.

For depth-first search, the only nodes in memory are those on the path from the root to the current node. Comparing those nodes to the current node allows the algorithm to detect looping paths that can be discarded immediately. This is fine for ensuring that finite state spaces do not become infinite search trees because of loops; unfortunately, it does not avoid the exponential proliferation of nonlooping paths in problems such as those in Figure 3.18. The only way to avoid these is to keep more nodes in memory. There is a fundamental tradeoff between space and time. *Algorithms that forget their history are doomed to repeat it.*

If an algorithm remembers every state that it has visited, then it can be viewed as exploring the state-space graph directly. We can modify the general TREE-SEARCH algorithm to include a data structure called the **closed list**, which stores every expanded node. (The fringe of unexpanded nodes is sometimes called the **open list**.) If the current node matches a node on the closed list, it is discarded instead of being expanded. The new algorithm is called GRAPH-SEARCH (Figure 3.19). On problems with many repeated states, GRAPH-SEARCH is much more efficient than TREE-SEARCH. Its worst-case time and space requirements are proportional to the size of the state space. This may be much smaller than $O(b^d)$.

Optimality for graph search is a tricky issue. We said earlier that when a repeated state is detected, the algorithm has found two paths to the same state. The GRAPH-SEARCH algorithm in Figure 3.19 always discards the *newly discovered* path; obviously, if the newly discovered path is shorter than the original one, GRAPH-SEARCH could miss an optimal solution. Fortunately, we can show (Exercise 3.12) that this cannot happen when using either

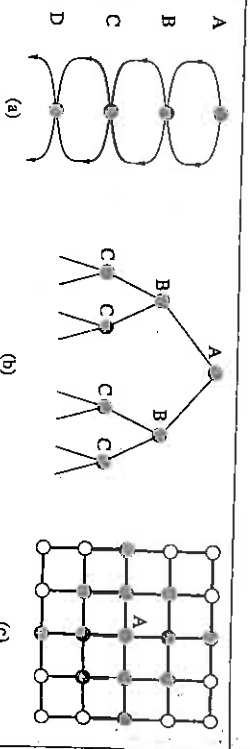


Figure 3.18 State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains $d + 1$ states, where d is the maximum depth. (b) The corresponding search tree, which has 2^d branches corresponding to the 2^d paths through the space. (c) A rectangular grid space. States within 2 steps of the initial state (A) are shown in gray.

```

function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST(problem)(STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

```

Figure 3.19 The general graph-search algorithm. The set *closed* can be implemented with a hash table to allow efficient checking for repeated states. This algorithm assumes that the first path to a state s is the cheapest (see text).

uniform-cost search or breadth-first search with constant step costs; hence, these two optimal tree-search strategies are also optimal graph-search strategies. Iterative deepening search, on the other hand, uses depth-first expansion and can easily follow a suboptimal path to a node before finding the optimal one. Hence, iterative deepening graph search needs to check whether a newly discovered path to a node is better than the original one, and if so, it might need to revise the depths and path costs of that node's descendants.

Note that the use of a closed list means that depth-first search and iterative deepening search no longer have linear space requirements. Because the GRAPH-SEARCH algorithm keeps every node in memory, some searches are infeasible because of memory limitations.

3.6 SEARCHING WITH PARTIAL INFORMATION

In Section 3.3 we assumed that the environment is fully observable and deterministic and that the agent knows what the effects of each action are. Therefore, the agent can calculate exactly which state results from any sequence of actions and always knows which state it is in. Its percepts provide no new information after each action. What happens when knowledge of the states or actions is incomplete? We find that different types of incompleteness lead to three distinct problem types:

1. **Sensorless problems** (also called **conformant problems**). If the agent has no sensors at all, then (as far as it knows) it could be in one of several possible initial states, and each action might therefore lead to one of several possible successor states.
2. **Contingency problems**. If the environment is partially observable or if actions are uncertain, then the agent's percepts provide *new* information after each action. Each possible percept defines a contingency that must be planned for. A problem is called **adversarial** if the uncertainty is caused by the actions of another agent.

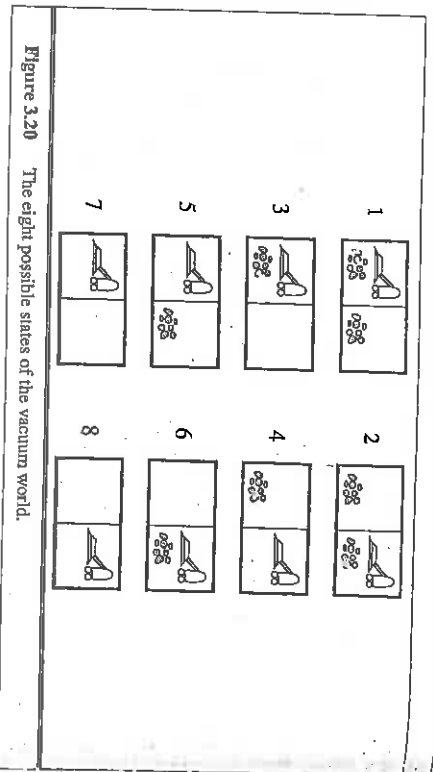


Figure 3.20 The eight possible states of the vacuum world.

3. Exploration problems: When the states and actions of the environment are unknown, the agent must act to discover them. Exploration problems can be viewed as an extreme case of contingency problems.

As an example, we will use the vacuum world environment. Recall that the state space has eight states, as shown in Figure 3.20. There are three actions—*Left*, *Right*, and *Suck*—and the goal is to clean up all the dirt (states 7 and 8). If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms we have described. For example, if the initial state is 5, then the action sequence [*Right*, *Suck*] will reach a goal state, 8. The remainder of this section deals with the sensorless and contingency versions of the problem. Exploration problems are covered in Section 4.5, adversarial problems in Chapter 6.

Sensorless problems

Suppose that the vacuum agent knows all the effects of its actions, but has no sensors. Then it knows only that its initial state is one of the set {1, 2, 3, 4, 5, 6, 7, 8}. One might suppose that the agent's predicament is hopeless, but in fact it can do quite well. Because it knows what its actions do, it can, for example, calculate that the action *Right* will cause it to be in one of the states {2, 4, 6, 8}, and the action sequence [*Right*, *Suck*] will always end up in one of the states {4, 8}. Finally, the sequence [*Right*, *Suck*, *Left*, *Suck*] is guaranteed to reach the goal state 7 no matter what the start state. We say that the agent can coerce the world into state 7, even when it doesn't know where it started. To summarize: when the world is not fully observable, the agent must reason about *sets* of states that it might get to, rather than single states. We call each such set of states a *belief state*, representing the agent's current belief about the possible physical states it might be in. (In a fully observable environment, each belief state contains one physical state.)

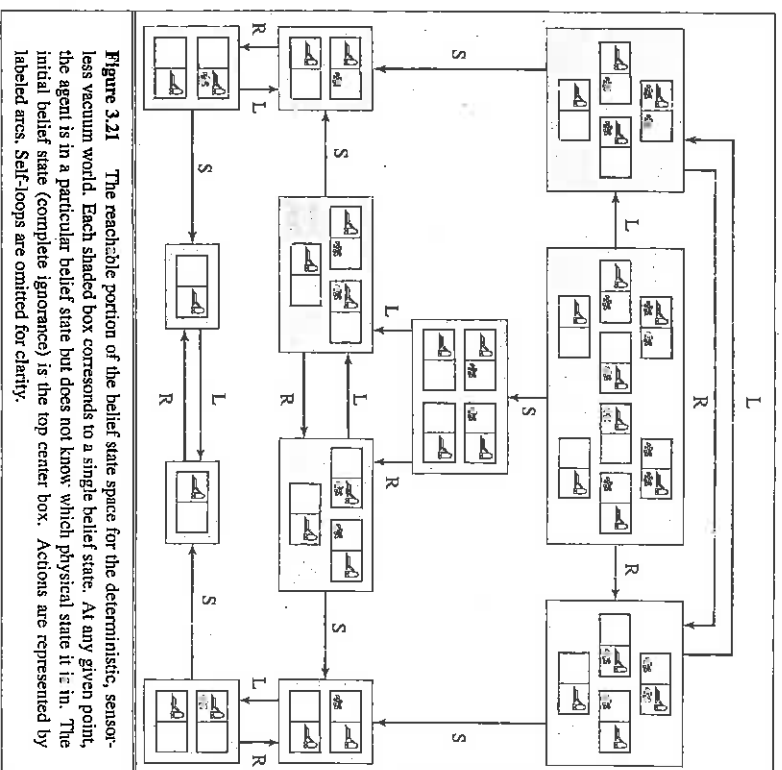


Figure 3.21 The reachable portion of the belief state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box. Actions are represented by labeled arcs. Self-loops are omitted for clarity.

To solve sensorless problems, we search in the space of belief states rather than physical states. The initial state is a belief state, and each action maps from a belief state to another belief state. An action is applied to a belief state by uniting the results of applying the action to each physical state in the belief state. A path now connects several belief states, and a solution is now a path that leads to a belief state, *all of whose members* are goal states. Figure 3.21 shows the reachable belief-state space for the deterministic, sensorless vacuum world. There are only 12 reachable belief states, but the entire belief state space contains every possible set of physical states, i.e., $2^8 = 256$ belief states. In general, if the physical state space has S states, the belief state space has 2^S belief states.

Our discussion of sensorless problems so far has assumed deterministic actions, but the analysis is essentially unchanged if the environment is nondeterministic—that is, if actions may have several possible outcomes. The reason is that, in the absence of sensors, the agent

has no way to tell which outcome actually occurred, so the various possible outcomes are just additional physical states in the successor belief state. For example, suppose the environment obeys Murphy's Law: the so-called *Suck* action *sometimes* deposits dirt on the carpet *but only if there is no dirt there already*.⁶ Then, if *Suck* is applied in physical state 4 (see Figure 3.20), there are two possible outcomes: states 2 and 4. Applied to the initial belief state, $\{1, 2, 3, 4, 5, 6, 7, 8\}$, *Suck* now leads to the belief state that is the union of the outcome sets for the eight physical states. Calculating this, we find that the new belief state is $\{1, 2, 3, 4, 5, 6, 7, 8\}$. So, for a sensorless agent in the Murphy's Law world, the *Suck* action leaves the belief state unchanged! In fact, the problem is unsolvable. (See Exercise 3.18.) Intuitively, the reason is that the agent cannot tell whether the current square is dirty and hence cannot tell whether the *Suck* action will clean it up or create more dirt.

Contingency problems

When the environment is such that the agent can obtain new information from its sensors after acting, the agent faces a **contingency problem**. The solution to a contingency problem often takes the form of a *tree*, where each branch may be selected depending on the percepts received up to that point in the tree. For example, suppose that the agent is in the Murphy's Law world and that it has a position sensor and a local dirt sensor, but no sensor capable of detecting dirt in other squares. Thus, the percept $[L, Dirty]$ means that the agent is in one of the states $\{1, 3\}$. The agent might formulate the action sequence [*Suck*, *Right*, *Suck*]. Sucking would change the state to one of $\{5, 7\}$, and moving right would then change the state to one of $\{6, 8\}$. Executing the final *Suck* action in state 6 takes us to state 8, a goal, but executing it in state 8 might take us back to state 6 (by Murphy's Law), in which case the plan fails.

By examining the belief-state space for this version of the problem, it can easily be determined that no fixed action sequence guarantees a solution to this problem. There is, however, a solution if we don't insist on a *fixed* action sequence:

[*Suck*, *Right*, *if (R, Dirty) then Suck*].

This extends the space of solutions to include the possibility of selecting actions based on contingencies arising during execution. Many problems in the real, physical world are contingency problems, because exact prediction is impossible. For this reason, many people keep their eyes open while walking around or driving.

Contingency problems *sometimes* allow purely sequential solutions. For example, consider a *fully observable* Murphy's Law world. Contingencies arise if the agent performs a *Suck* action in a clean square, because dirt might or might not be deposited in the square. As long as the agent never does this, no contingencies arise and there is a sequential solution from every initial state (Exercise 3.18).

The algorithms for contingency problems are more complex than the standard search algorithms in this chapter; they are covered in Chapter 12. Contingency problems also lend themselves to a somewhat different agent design, in which the agent can act *before* it has found a guaranteed plan. This is useful because rather than considering in advance every

⁶ We assume that most readers face similar problems and can sympathize with our agent. We apologize to owners of modern, efficient home appliances who cannot take advantage of this pedagogical device.

INTERLEAVING

possible contingency that *might* arise during execution, it is often better to start acting and see which contingencies *do* arise. The agent can then continue to solve the problem, taking into account the additional information. This type of interleaving of search and execution is also useful for exploration problems (see Section 4.5) and for game playing (see Chapter 6).

3.7 SUMMARY

This chapter has introduced methods that an agent can use to select actions in environments that are deterministic, observable, static, and completely known. In such cases, the agent can construct sequences of actions that achieve its goals; this process is called **search**.

- Before an agent can start searching for solutions, it must formulate a goal and then use the goal to formulate a problem.
- A problem consists of four parts: the **initial state**, a set of **actions**, a **goal test function**, and a **path cost function**. The environment of the problem is represented by a **state space**. A path through the state space from the initial state to a goal state is a **solution**.
- A single, general TREE-SEARCH algorithm can be used to solve any problem; specific variants of the algorithm embody different strategies.
- Search algorithms are judged on the basis of **completeness**, **optimality**, **time complexity**, and **space complexity**. Complexity depends on b , the branching factor in the state space, and d , the depth of the shallowest solution.
- **Breadth-first search** selects the shallowest unexpanded node in the search tree for expansion. It is complete, optimal for unit step costs, and has time and space complexity of $O(b^d)$. The space complexity makes it impractical in most cases. Uniform-cost search is similar to breadth-first search but expands the node with lowest path cost, $g(n)$. It is complete and optimal if the cost of each step exceeds some positive bound ϵ .
- **Depth-first search** selects the deepest unexpanded node in the search tree for expansion. It is neither complete nor optimal, and has time complexity of $O(b^m)$ and space complexity of $O(bm)$, where m is the maximum depth of any path in the state space.
- **Depth-limited search** imposes a fixed depth limit on a depth-first search.
- **Iterative deepening search** calls depth-limited search with increasing limits until a goal is found. It is complete, optimal for unit step costs, and has time complexity of $O(b^d)$ and space complexity of $O(bd)$.
- **Bidirectional search** can enormously reduce time complexity, but it is not always applicable and may require too much space.
- When the state space is a graph rather than a tree, it can pay off to check for repeated states in the search tree. The GRAPH-SEARCH algorithm eliminates all duplicate states.
- When the environment is partially observable, the agent can apply search algorithms in the space of **belief states**, or sets of possible states that the agent might be in. In some cases, a single solution sequence can be constructed; in other cases, the agent needs a **contingency plan** to handle unknown circumstances that may arise.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Most of the state-space search problems analyzed in this chapter have a long history in literature and are less trivial than they might seem. The missionaries and cannibals problem used in Exercise 3.9 was analyzed in detail by Amarel (1968). It had been considered earlier in AI by Simon and Newell (1961), and in operations research by Bellman and Dreyfus (1962). Studies such as these and Newell and Simon's work on the Logic Theorist (1958) and GPS (1961) led to the establishment of search algorithms as the primary weapons in the armory of 1960s AI researchers and to the establishment of problem solving as the canonical task. Unfortunately, very little work was done on the automation of the problem formulation step. A more recent treatment of problem representation and abstraction, including AI programs that themselves perform these tasks (in part), is in Knoblock (1990).

The 8-puzzle is a smaller cousin of the 15-puzzle, which was invented by the famous American game designer Sam Loyd (1959) in the 1870s. The 15-puzzle quickly achieved immense popularity in the United States, comparable to the more recent sensation caused by Rubik's Cube. It also quickly attracted the attention of mathematicians (Johnson and Storck 1879; Tait, 1880). The editors of the *American Journal of Mathematics* stated "The 15 puzzle for the last few weeks has been prominently before the American public, and magazines and conditions of the community. But this would not have weighed with the editors if for the fact that..." (there follows a summary of the mathematical interest of the 15-puzzle). An exhaustive analysis of the 8-puzzle was carried out with computer aid by Schofield (1967). Ratner and Warmuth (1986) showed that the general $n \times n$ version of the 15-puzzle belongs to the class of NP-complete problems.

The 8-queens problem was first published anonymously in the German chess magazine *Schach* in 1848; it was later attributed to one Max Bezzel. It was republished in 1850 and at that time drew the attention of the eminent mathematician Carl Friedrich Gauss, who attempted to enumerate all possible solutions, but found only 72. Nauck published all 92 solutions later in 1850. Netto (1901) generalized the problem to n queens, and Abramson and Yung (1989) found an $O(n)$ algorithm.

Each of the real-world search problems listed in the chapter has been the subject of a good deal of research effort. Methods for selecting optimal airline flights remain proprietary for the most part, but Carl de Marcken (personal communication) has shown that airline ticket pricing and restrictions have become so convoluted that the problem of selecting an optimal flight is formally undecidable. The traveling-salesperson problem is a standard combinatorial problem in theoretical computer science (Lawler, 1985; Lawler et al., 1992). Karp (1972) proved the TSP to be NP-hard, but effective heuristic approximation methods were developed (Lin and Kernighan, 1973). Arora (1998) devised a fully polynomial approximation scheme for Euclidean TSPs. VLSI layout methods are surveyed by Shahookar and Mazumder (1991), and many layout optimization papers appear in VLSI journals. Robotic navigation and assembly problems are discussed in Chapter 25.

Uniformed search algorithms for problem solving are a central topic of classical computer science (Horowitz and Sahni, 1978) and operations research (Dreyfus, 1969). Deo and Pang (1984) and Gallo and Pallottino (1988) give more recent surveys. Breadth-first search was formulated for solving mazes by Moore (1959). The method of dynamic programming (Bellman and Dreyfus, 1962), which systematically records solutions for all subproblems of increasing lengths, can be seen as a form of breadth-first search on graphs. The two-point shortest-path algorithm of Dijkstra (1959) is the origin of uniform-cost search.

A version of iterative deepening designed to make efficient use of the chess clock was first used by Slate and Atkin (1977) in the Chess 4.5 game-playing program, but the application to shortest path graph search is due to Korf (1985a). Bidirectional search, which was introduced by Pohl (1969, 1971), can also be very effective in some cases.

Partially observable and nondeterministic environments have not been studied in great depth within the problem-solving approach. Some efficiency issues in belief-state search have been investigated by Genesereth and Nourbakhsh (1993). Koenig and Simmons (1998) studied robot navigation from an unknown initial position, and Erdmann and Mason (1988) studied the problem of robotic manipulation without sensors, using a continuous form of belief-state search. Contingency search has been studied within the planning subfield. (See Chapter 12.) For the most part, planning and acting with uncertain information have been handled using the tools of probability and decision theory (see Chapter 17).

The textbooks by Nilsson (1971, 1980) are good general sources of information about classical search algorithms. A comprehensive and more up-to-date survey can be found in Korf (1988). Papers about new search algorithms—which, remarkably, continue to be discovered—appear in journals such as *Artificial Intelligence*.

EXERCISES

- 3.1. Define in your own words the following terms: state, state space, search tree, search node, goal, action, successor function, and branching factor.
- 3.2. Explain why problem formulation must follow goal formulation.
- 3.3. Suppose that $\text{LEGAL-ACTIONS}(s)$ denotes the set of actions that are legal in state s , and $\text{RESULT}(a, s)$ denotes the state that results from performing a legal action a in state s . Define SUCCESSOR-FN in terms of LEGAL-ACTIONS and RESULT , and *vice versa*.
- 3.4. Show that the 8-puzzle states are divided into two disjoint sets, such that no state in one set can be transformed into a state in the other set by any number of moves. (Hint: See Bertekamp et al. (1982).) Devise a procedure that will tell you which class a given state is in, and explain why this is a good thing to have for generating random states.
- 3.5. Consider the n -queens problem using the "efficient" incremental formulation given on page 67. Explain why the state space size is at least $\sqrt{n!}$ and estimate the largest n for which exhaustive exploration is feasible. (Hint: Derive a lower bound on the branching factor by considering the maximum number of squares that a queen can attack in any column.)

4 INFORMED SEARCH AND EXPLORATION

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. Section 4.1 describes informed versions of the algorithms in Chapter 3, and Section 4.2 explains how the necessary problem-specific information can be obtained. Sections 4.3 and 4.4 cover algorithms that perform purely local search in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state. These algorithms are suitable for problems in which the path cost is irrelevant and all that matters is the solution state itself. The family of local-search algorithms includes methods inspired by statistical physics (simulated annealing) and evolutionary biology (genetic algorithms). Finally, Section 4.5 investigates online search, in which the agent is faced with a state space that is completely unknown.

4.1 INFORMED (HEURISTIC) SEARCH STRATEGIES

INFORMED SEARCH
BEST-FIRST SEARCH
EVALUATION FUNCTION

This section shows how an **informed search strategy**—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than an uninformed strategy.

The general approach we will consider is called **best-first search**. Best-first search is an instance of the general **TREE-SEARCH** or **GRAPH-SEARCH** algorithm in which a node is selected for expansion based on an **evaluation function**, $f(n)$. Traditionally, the node with the **lowest** evaluation is selected for expansion, because the evaluation measures distance to the goal. Best-first search can be implemented within our general search framework via a priority queue, a data structure that will maintain the fringe in ascending order of f -values.

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

HEURISTIC FUNCTION

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 exactly accurate, 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.

There is a whole family of **BEST-FIRST-SEARCH** algorithms with different evaluation functions.¹ A key component of these algorithms is a **heuristic function**,² denoted $h(n)$:

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

For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.

Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. We will study heuristics in more depth in Section 4.2. For now, we will consider them to be arbitrary problem-specific functions, with one constraint: if n is a goal node, then $h(n) = 0$. The remainder of this section covers two ways to use heuristic information to guide search.

Greedy best-first search

GREEDY BEST-FIRST SEARCH
DETERMINE

Greedy best-first search³ tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function: $f(n) = h(n)$.

Let us see how this works for route-finding problems in Romania, using the **straight-line distance heuristic**, which we will call h_{SLD} . If the goal is Bucharest, we will need to know the straight-line distances to Bucharest, which are shown in Figure 4.1. For example, $h_{SLD}(In(Arad)) = 366$. Notice that the values of h_{SLD} cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

Figure 4.1 Values of h_{SLD} —straight-line distances to Bucharest.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Cluj	160	Oradea	380
Debrece	242	Pitești	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Ghirgih	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zetind	374

¹ Exercise 4.3 asks you to show that this family includes several familiar uninformed algorithms.
² A heuristic function $h(n)$ takes a node as input, but is depends only on the state at that node.
³ Our first edition called this **greedy search**; other authors have called it **best-first search**. Our more general usage of the latter term follows Pearl (1984).

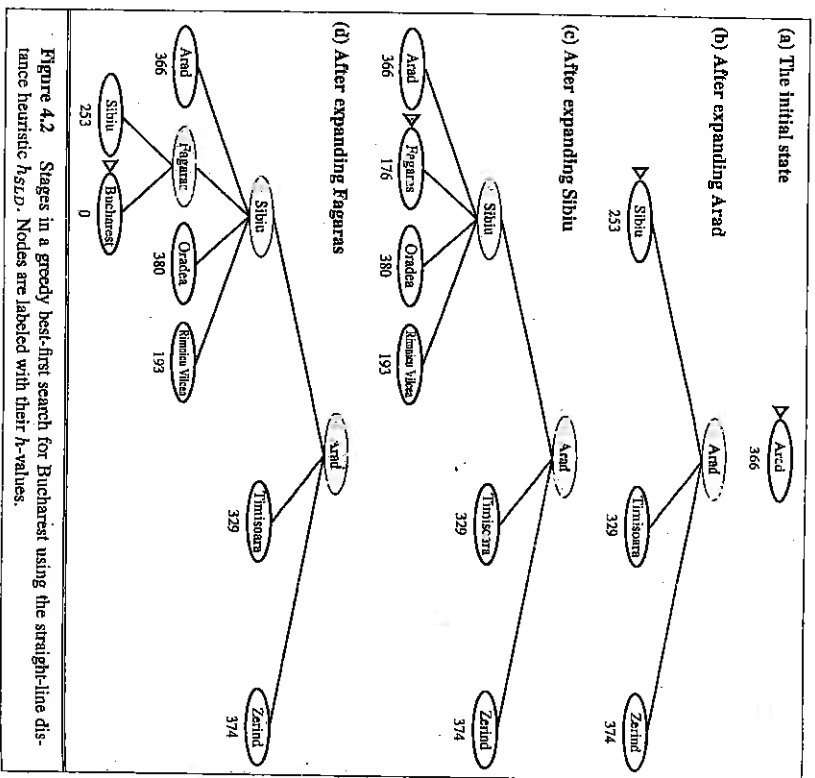


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD} . Nodes are labeled with their h -values.

Figure 4.2 shows the progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. 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, greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal. It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnic Vilcea and Pitesti. This shows why the algorithm is called “greedy”—at each step it tries to get as close to the goal as it can.

Minimizing $h(n)$ is susceptible to false starts. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first, because it is closest

to Fagaras, 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. In this case, then, 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.

Greedy best-first 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 and space complexity is $O(b^m)$, where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially. The amount of the reduction depends on the particular problem and on the quality of the heuristic.

A* search: Minimizing the total estimated solution cost

The most widely-known form of best-first search is called A* search (pronounced “A-star search”). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

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

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 $g(n) + h(n)$. It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal.

The optimality of A* is straightforward to analyze if it is used with TREE-SEARCH. In this case, A* is optimal if $h(n)$ is an **admissible heuristic**—that is, provided that $h(n)$ *never overestimates* the cost to reach the goal. Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is. Since $g(n)$ is the exact cost to reach n , we have as immediate consequence that $f(n)$ never overestimates the true cost of a solution through n .

An 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, so the straight line cannot be an overestimate. In Figure 4.3, we show the progress of an A* tree search for Bucharest. The values of g are computed from the step costs in Figure 3.2, and the values of h_{SLD} are given in Figure 4.1. Notice in particular that Bucharest first appears on the fringe at step (e), but it is not selected for expansion because its f -cost (450) is higher than that of Pitesti (417). Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450. From this example, we can extract a general proof that A* using TREE-SEARCH is optimal if $h(n)$ is admissible. Suppose

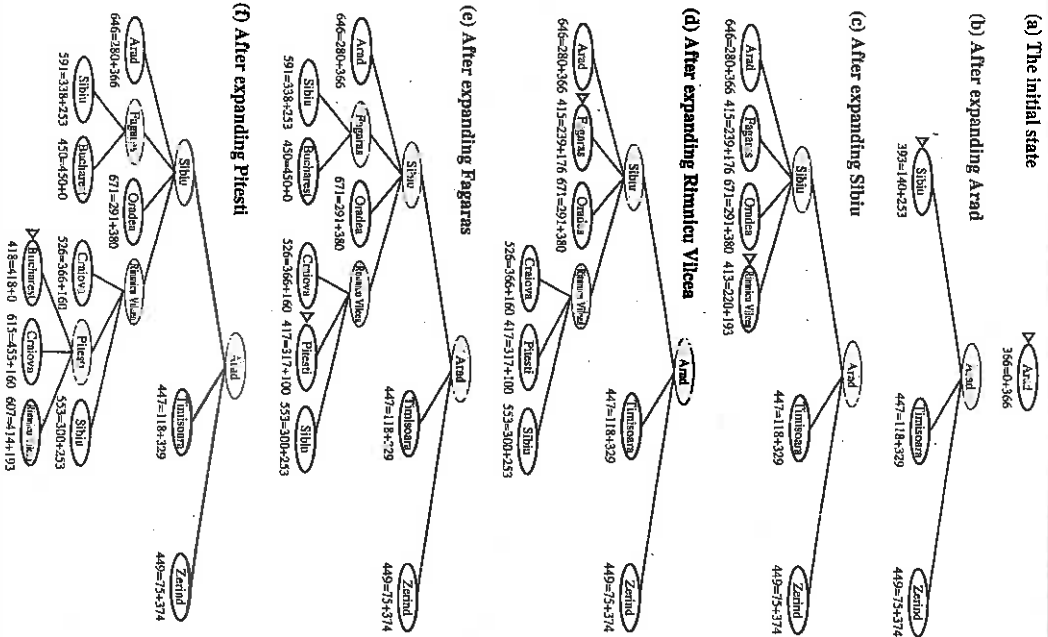


Figure 4.3 Stages in an A^* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

Section 4.1.

Informed (Heuristic) Search Strategies

99

suboptimal goal node G_2 appears on the fringe, and let the cost of the optimal solution be C^* . Then, because G_2 is suboptimal and because $h(G_2) = 0$ (true for any goal node), we know

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*.$$

Now consider a fringe node n that is on an optimal solution path—for example, Pitesti in the example of the preceding paragraph. (There must always be such a node if a solution exists.) If $h(n)$ does not overestimate the cost of completing the solution path, then we know that

$$f(n) = g(n) + h(n) \leq C^*.$$

Now we have shown that $f(n) \leq C^* < f(G_2)$, so G_2 will not be expanded and A^* must return an optimal solution.

If we use the GRAPH-SEARCH algorithm of Figure 3.19 instead of TREE-SEARCH, then this proof breaks down. Suboptimal solutions can be returned because GRAPH-SEARCH can discard the optimal path to a repeated state if it is not the first one generated. (See Exercise 4.4.) There are two ways to fix this problem. The first solution is to extend GRAPH-SEARCH so that it discards the more expensive of any two paths found to the same node. (See the discussion in Section 3.5.) The extra bookkeeping is messy, but it does guarantee optimality. The second solution is to ensure that the optimal path to any repeated state is always the first one followed—as is the case with uniform-cost search. This property holds if we impose an extra requirement on $h(n)$, namely the requirement of consistency (also called *monotonicity*). A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n' :

$$h(n) \leq c(n, a, n') + h(n').$$

This is a form of the **general triangle inequality**, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by n , n' , and the goal closest to n . It is fairly easy to show (Exercise 4.7) that every consistent heuristic is also admissible. The most important consequence of consistency is the following: A^* using GRAPH-SEARCH is optimal if $h(n)$ is consistent.

Although consistency is a stricter requirement than admissibility, one has to work quite hard to concoct heuristics that are admissible but not consistent. All the admissible heuristics we discuss in this chapter are also consistent. Consider, for example, h_{SLD} . We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance, and that the straight-line distance between n and n' is no greater than $c(n, a, n')$. Hence, h_{SLD} is a consistent heuristic.

Another important consequence of consistency is the following: If $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing. The proof follows directly from the definition of consistency. Suppose n' is a successor of n , then $g(n') = g(n) + c(n, a, n')$ for some a , and we have

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n).$$

It follows that the sequence of nodes expanded by A^* using GRAPH-SEARCH is in nondecreasing order of $f(n)$. Hence, the first goal node selected for expansion must be an optimal solution, since all later nodes will be at least as expensive.

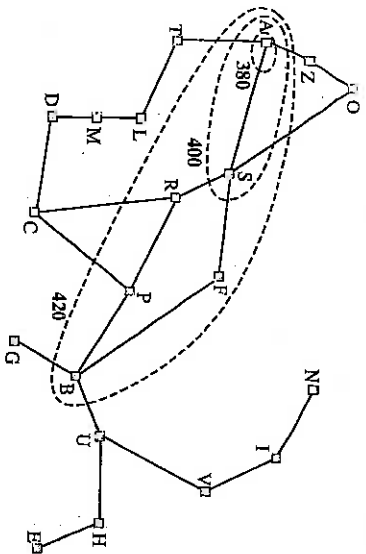


Figure 4.4 Map of Romania showing contours at $f = 380$, $f = 400$ and $f = 420$, with A as the start state. Nodes inside a given contour have f -costs less than or equal to the contour value.

The fact that f -costs are nondecreasing along any path also means that we can draw contours in the state space, just like the contours in a topographic map. Figure 4.4 shows an example. Inside the contour labeled 400, all nodes have $f(n)$ less than or equal to 400, and so on. Then, because A^* expands the fringe node of lowest f -cost, we can see that an A^* search fans out from the start node, adding nodes in concentric bands of increasing f -cost.

With uniform-cost search (A^* search using $h(n) = 0$), 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 C^* is the cost of the optimal solution path, then we can say the following:

- A^* expands all nodes with $f(n) < C^*$.
- A^* might then expand some of the nodes right on the “goal contour” (where $f(n) = C^*$) before selecting a goal node.

Intuitively, it is obvious that the first solution found must be an optimal one, because goal nodes in all subsequent contours will have higher f -cost, and thus higher g -cost (because all goal nodes have $h(n) = 0$). Intuitively, it is also obvious that A^* search is complete. As we add bands of increasing f , we must eventually reach a band where f is equal to the cost of the path to a goal state.⁴

Notice that A^* expands no nodes with $f(n) > C^*$ —for example, Timisoara is not expanded in Figure 4.3 even though it is a child of the root. We say that the subtree below Timisoara is **pruned**, because h_{SLD} is admissible, the algorithm can safely ignore this subtree that is true if all step costs exceed some finite ϵ and if b is finite.

PRELIMINARY

while still guaranteeing optimality. The concept of pruning—eliminating possibilities from consideration without having to examine them—is important for many areas of AI.

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^* (except possibly through tie-breaking among nodes with $f(n) = C^*$). This is because any algorithm that *does not* expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution.

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 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. For this reason, it is often impractical to insist on finding an optimal solution. One can use variants of A^* that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate, but not strictly admissible. In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search. In Section 4.2, 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 (as do all GRAPH-SEARCH algorithms), A^* usually runs out of space long before it runs out of time. For this reason, A^* is not practical for many large-scale problems. Recently developed algorithms have overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time. These are discussed next.

Memory-bounded heuristic search

The simplest way to reduce memory requirements for A^* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A^* (IDA*) algorithm. The main difference between IDA* and standard iterative deepening is that the cutoff used is the f -cost ($g + h$) rather than the depth; at each iteration, the cutoff value is the smallest f -cost of any node that exceeded the cutoff on the previous iteration. IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes. Unfortunately, it suffers from the same difficulties with real-valued costs as does the iterative version of uniform-cost search described in Exercise 3.11. This section briefly examines two more recent memory-bounded algorithms, called RBFS and MA*

Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. The algorithm is shown in Figure 4.5. Its structure is similar to that of a recursive depth-first search, but rather

RECURSIVE
BEST-FIRST SEARCH

```

function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
  RBFS(problem, MAKE-NODE(INITIAL-STATE(problem)),  $\infty$ )

function RBFS(problem, node, f-limit) returns a solution, or failure and a new f-cost limit
  if GOAL-TEST(problem)(state) then return node
  if successors = EXPAND(node, problem)
    for each s in successors do
       $f[s] \leftarrow \max(g(s) + h(s), f[node])$ 
    repeat
      best  $\leftarrow$  the lowest f-value node in successors
      if  $f[best] > f-limit$  then return failure,  $f[best]$ 
      alternative  $\leftarrow$  the second-lowest f-value among successors
      result, f[best]  $\leftarrow$  RBFS(problem, best,  $\min(f-limit, alternative)$ )
      if result  $\neq$  failure then return result

```

Figure 4.5 The algorithm for recursive best-first search.

then continuing indefinitely down the current path, it keeps track of the *f*-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBF replaces the *f*-value of each node along the path with the best *f*-value of its children. In this way, RBFS remembers the *f*-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time. Figure 4.6 shows how RBFS reaches Bucharest.

RBFS is somewhat more efficient than IDA*, but still suffers from excessive node re-generation. In the example in Figure 4.6, RBFS first follows the path via Rimnicu Vilcea then “changes its mind” and tries Fagaras, and then changes its mind back again. These mind changes occur because every time the current best path is extended, there is a good chance that its *f*-value will increase—*h* is usually less optimistic for nodes closer to the goal. When this happens, particularly in large search spaces, the second-best path might become the best path, so the search has to backtrack to follow it. Each mind change corresponds to an iteration of IDA*, and could require many reexpansions of forgotten nodes to recreate the best path and extend it one more node.

Like A*, RBFS is an optimal algorithm if the heuristic function $h(n)$ is admissible. Its space complexity is $O(bd)$, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. Both IDA* and RBFS are subject to the potentially exponential increase in complexity associated with searching on graphs (see Section 3.5), because they cannot check for repeated states other than those on the current path. Thus, they may explore the same state many times.

IDA* and RBFS suffer from using *too little* memory. Between iterations, IDA* retains only a single number: the current *f*-cost limit. RBFS retains more information in memory,

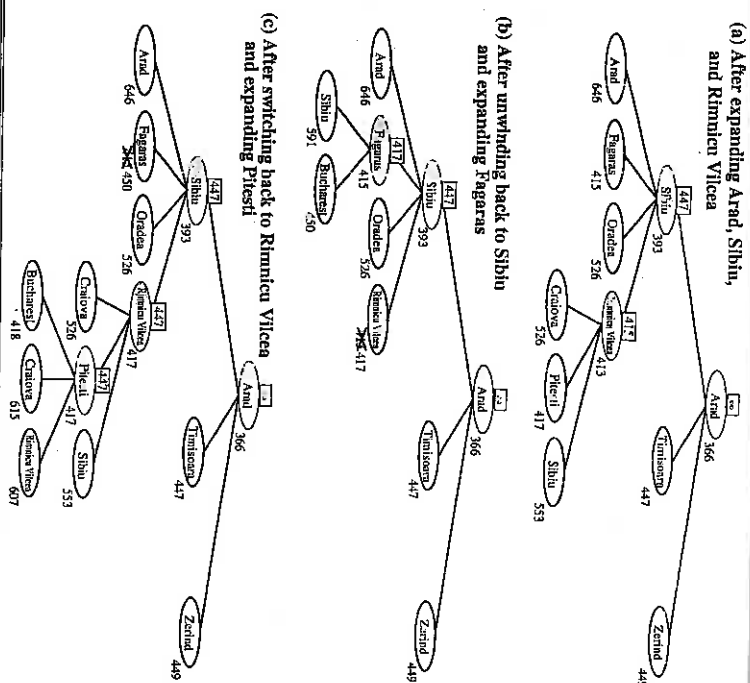


Figure 4.6 Stages in an RBFS search for the shortest route to Bucharest. The *f*-limit value for each recursive call is shown on top of each current node. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

but it uses only $O(bd)$ memory: even if more memory were available, RBFS has no way to make use of it.

It seems sensible, therefore, to use all available memory. Two algorithms that do this are MA* (memory-bounded A*) and SMA* (simplified MA*). We will describe SMA*, which

is—well—simpler. SMA* proceeds just like A*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the *worst* leaf node—the one with the highest f -value. Like RBFS, SMA* then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA* regenerates the subtree only 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 .

The complete algorithm is too complicated to reproduce here,⁵ but there is one subtlety worth mentioning. We said that SMA* expands the best leaf and deletes the worst leaf. What if *all* the leaf nodes have the same f -value? Then the algorithm might select the same node for deletion and expansion. SMA* solves this problem by expanding the *newest* best leaf and deleting the *oldest* worst leaf. These can be the same node only if there is only one leaf; in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then *even if it is on an optimal solution path*, that solution is not reachable with the available memory. Therefore, the node can be discarded exactly as if it had no successors.

SMA* is complete if there is any reachable solution—that is, if d , the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise it returns the best reachable solution. In practical terms, SMA* might well be the best general-purpose algorithm for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the additional overhead of maintaining the open and closed lists.

On very hard problems, however, it will often be the case that SMA* is forced to switch back and forth continually between a set of candidate solution paths, only a small subset of which can fit in memory. (This resembles the problem of *thrashing* in disk paging systems.) 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 tradeoff between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.

Learning to search better

We have presented several fixed strategies—breadth-first, greedy best-first, and so on—that have been designed by computer scientists. Could an agent *learn* how to search better? The answer is yes, and the method rests on an important concept called the *metalevel state space*. Each state in a *metalevel state space* captures the internal (computational) state of a program that is searching in an *object-level state space* such as Romania. For example, the internal state of the A* algorithm consists of the current search tree. Each action in the *metalevel state*

⁵ A rough sketch appeared in the first edition of this book.

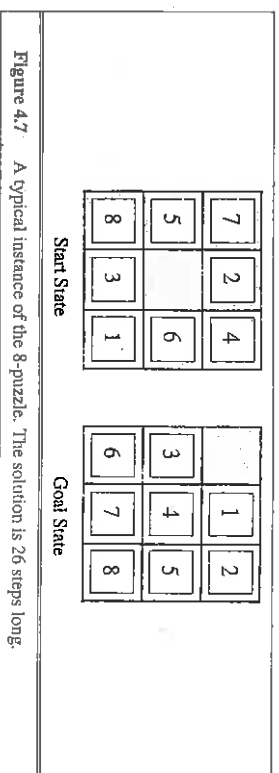
space is a computation step that alters the internal state; for example, each computation step in A* expands a leaf node and adds its successors to the tree. Thus, Figure 4.3, which shows a sequence of larger and larger search trees, can be seen as depicting a path in the *metalevel state space* where each state on the path is an *object-level* search tree.

Now, the path in Figure 4.3 has five steps, including one step, the expansion of Fagaras, that is not especially helpful. For harder problems, there will be many such missteps, and a *metalevel learning* algorithm can learn from these experiences to avoid exploring unpromising subtrees. The techniques used for this kind of learning are described in Chapter 21. The goal of learning is to minimize the *total cost* of problem solving, trading off computational expense and path cost.

4.2 HEURISTIC FUNCTIONS

In this section, we will look at heuristics for the 8-puzzle, in order to shed light on the nature of heuristics in general.

The 8-puzzle was one of the earliest heuristic search problems. As mentioned in Section 3.2, the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration (Figure 4.7).



The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. 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 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states. By keeping track of repeated states, we could cut this down by a factor of about 170,000, because there are only $9!/2 = 181,440$ distinct states that are reachable. (See Exercise 3.4.) This is a manageable number, but the corresponding number for the 15-puzzle is roughly 10^{13} , so the next order of business is to find a good heuristic function. If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 15-puzzle; here are two commonly-used candidates:

MANHATTAN DISTANCE

- h_1 = the number of misplaced tiles. For Figure 4.7, all of the eight tiles are out of 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.
- 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 all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.$$

As we would hope, neither of these overestimates the true solution cost, which is 26.

The effect of heuristic accuracy on performance

One way to characterize the quality of a heuristic is the **effective branching factor** b^* . If the total number of nodes generated 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 $N + 1$ nodes. Thus,

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

For example, if A^* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. 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_1 and h_2 , we generated 1200 random problems with solution lengths from 2 to 24 (100 for each even number) and solved them with iterative deepening search and with A^* tree search using both h_1 and h_2 . Figure 4.8 gives the average number of nodes expanded by each strategy and the effective branching factor. The results suggest that h_2 is better than h_1 , and is far better than using iterative deepening search. On our solutions with length 14, A^* with h_2 is 30,000 times more efficient than uninformed iterative deepening search.

One might ask whether 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 , $h_2(n) \geq h_1(n)$. We thus say that h_2 **dominates** h_1 . Domination translates directly into efficiency: A^* using h_2 will never expand more nodes than A^* using h_1 (except possibly for some nodes with $f(n) = C^*$). The argument is simple. Recall the observation on page 100 that every node with $f(n) < C^*$ will surely be expanded. This is the same as saying that every node with $h(n) < C^* - g(n)$ will surely be expanded. But because h_2 is at least as big as h_1 for all nodes, every node that is surely expanded by A^* search with h_2 will also surely be expanded with h_1 , and h_1 might also cause other nodes to be expanded as well. Hence, it is always better to use a heuristic function with higher values, provided it does not overestimate and that the computation time for the heuristic is not too large.

d	IDS	Search Cost		Effective Branching Factor			
		$A^*(h_1)$	$A^*(h_2)$	IDS	$A(h_1)$	$A(h_2)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79	1.79
4	112	13	12	2.67	1.48	1.45	1.45
6	680	20	18	2.73	1.34	1.30	1.30
8	6384	39	25	2.80	1.33	1.24	1.24
10	47127	93	39	2.79	1.38	1.22	1.22
12	3644035	227	73	2.78	1.42	1.24	1.24
14	—	539	113	—	1.44	1.23	1.23
16	—	1301	211	—	1.45	1.25	1.25
18	—	3056	363	—	1.46	1.26	1.26
20	—	7276	676	—	1.47	1.27	1.27
22	—	18094	1219	—	1.48	1.28	1.28
24	—	39135	1641	—	1.48	1.26	1.26

Figure 4.8 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A^* algorithms with h_1 , h_2 . Data are averaged over 100 instances of the 8-puzzle, for various solution lengths.

Inventing admissible heuristic functions

We have seen that both h_1 (misplaced tiles) and h_2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h_2 is better. How might one have come up with h_2 ? Is it possible for a computer to invent such a heuristic mechanically?

h_1 and h_2 are estimates of the remaining path length for the 8-puzzle, but they are also 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_1 would give the exact number of steps in 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 fewer restrictions on the actions is called a **relaxed problem**. The cost of an optimal solution to a relaxed problem is an **admissible heuristic for the original problem**. The heuristic is admissible because the optimal solution in the original problem is, by definition, also a solution in the relaxed problem. Because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore **consistent** (see page 99).

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.⁶ For example, if the 8-puzzle actions are described as

A tile can move from square A to square B if

A is horizontally or vertically adjacent to B and B is blank,

⁶ In Chapters 8 and 11, we will describe formal languages suitable for this task, with formal descriptions that can be manipulated, the construction of relaxed problems can be automated. For now, we will use English.

we can generate three relaxed problems by removing one or both 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.

From (a), we can derive h_2 (Manhattan distance). The reasoning is that h_2 would be proper score if we moved each tile in turn to its destination. The heuristic derived from (b) the proper score if tiles could move to their intended destination in one step. Notice that h_2 is crucial that the relaxed problems generated by this technique can be solved essentially *without* subproblems. If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to obtain.⁷

A program called ABSOLVER can generate heuristics automatically from problem definitions, using the "relaxed problem" method and various other techniques (Priditis, 1997). ABSOLVER generated a new heuristic for the 8-puzzle better than any preexisting heuristic and found the first useful heuristic for the famous Rubik's cube puzzle.

One problem with generating new heuristic functions is that one often fails to get on "clearly best" heuristic. If a collection of admissible heuristics $h_1, \dots, h_m$ 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

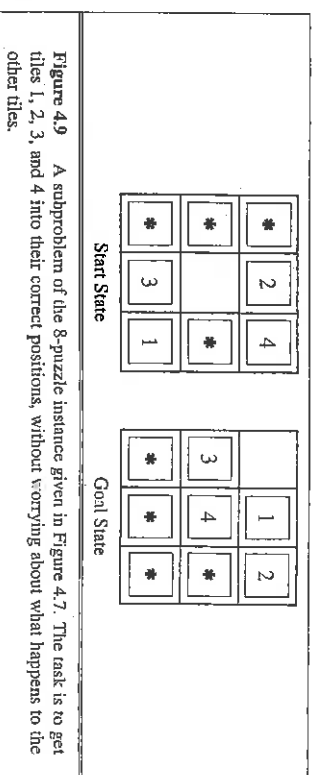
$$h(n) = \max(h_1(n), \dots, h_m(n)).$$

This composite heuristic uses whichever function is most accurate on the node in question. Because the component heuristics are admissible, h is admissible; it is also easy to prove that h is consistent. Furthermore, h dominates all of its component heuristics.

Admissible heuristics can also be derived from the solution cost of a subproblem on a given problem. For example, Figure 4.9 shows a subproblem of the 8-puzzle instance in Figure 4.7. The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions. Clearly, the cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem. It turns out to be substantially more accurate than Manhattan distance in some cases.

The idea behind **pattern databases** is to store these exact solution costs for every possible subproblem instance—in our example, every possible configuration of the four tiles and solving the subproblem, but moves of those tiles do count towards the cost.) Then, we compute an admissible heuristic h_{db} for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database. The database itself is constructed by searching backwards from the goal state and recording the cost of each new pattern encountered; the expense of this search is amortized over many subsequent problem instances.

⁷ Note that a perfect heuristic can be obtained simply by allowing h to run a full breadth-first search "on the fly." Thus, there is a tradeoff between accuracy and computation time for heuristic functions.



The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, and for 2-4-6-8, and so on. Each database yields an admissible heuristic, and these heuristics can be combined, as explained earlier, by taking the maximum value. A combined heuristic of this kind is much more accurate than the Manhattan distance; the number of nodes generated when solving random 15-puzzles can be reduced by a factor of 1000.

One might wonder whether the heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be *added*, since the two subproblems seem not to overlap. Would this still give an admissible heuristic? The answer is no, because the solutions of the 1-2-3-4 subproblem and the 5-6-7-8 subproblem for a given state will almost certainly share some moves—it is unlikely that 1-2-3-4 can be moved into place without touching 5-6-7-8, and vice versa. But what if we don't count those moves? That is, we record not the total cost of solving the 1-2-3-4 subproblem, but just the number of moves involving 1-2-3-4. Then it is easy to see that the sum of the two costs is still a lower bound on the cost of solving the entire problem. This is the idea behind **disjoint pattern databases**. Using such databases, it is possible to solve random 15-puzzles in a few milliseconds—the number of nodes generated is reduced by a factor of 10,000 compared with using Manhattan distance. For 24-puzzles, a speedup of roughly a million can be obtained.

Disjoint pattern databases work for sliding-tile puzzles because the problem can be divided up in such a way that each move affects only one subproblem—because only one tile is moved at a time. For a problem such as Rubik's cube, this kind of subdivision cannot be done because each move affects 8 or 9 of the 26 cubes. Currently, it is not clear how to define disjoint databases for such problems.

Learning heuristics from experience

A heuristic function $h(n)$ is supposed to estimate the cost of a solution beginning from the state at node n . How could an agent construct such a function? One solution was given in the preceding section—namely, to devise relaxed problems for which an optimal solution can be found easily. Another solution is to learn from experience. "Experience" here means solving lots of 8-puzzles, for instance. Each optimal solution to an 8-puzzle problem provides ex-

