3 SOLVING PROBLEMS BY SEARCHING

In which we see how an agent can find a sequence of actions that achieves its goals, when no single action **will** *do.*

The simplest agents discussed in Chapter 2 were the reflex agents, which base their actions on a direct mapping from states to actions. Such agents cannot operate well in environments for which this mapping would be too large to store and would take too long to learn. Goal-based agents, on the other hand, can succeed by considering future actions and the desirability of their outcomes.

PROBLEM-SOLVING AGENT

This chapter describes one kind of goal-based agent called a problem-solving agent. Problem-solving agents decide what to do by finding sequences of actions that lead to desirable states. We start by defining precisely the elements that constitute a "problem" and its "solution," and give several examples to illustrate these definitions. We then describe several general-purpose search algorithms that can be used to solve these problems and compare the advantages of each algorithm. The algorithms are uninformed, in the sense that they are given no information about the problem other than its definition. Chapter 4 deals with informed search algorithms, ones that have some idea of where to look for solutions.

This chapter uses concepts from the analysis of algorithms. Readers unfamiliar with the concepts of asymptotic complexity (that is, $O()$ notation) and NP-completeness should consult Appendix A.

3.1 PROBLEM-SOLVING AGENTS

Intelligent agents are supposed to maximize their performance measure. As we mentioned in Chapter 2, achieving this is sometimes simplified if the agent can adopt a goal and aim at satisfying it. Let us first look at why and how an agent might do this.

Imagine an agent in the city of Arad, Romania, enjoying a touring holiday. The agent's performance measure contains many factors: it wants to improve its suntan, improve its Romanian, take in the sights, enjoy the nightlife (such as it is), avoid hangovers, and so on. The decision problem is a complex one involving many tradeoffs and careful reading of guidebooks. Now, suppose the agent has a nonrefundable ticket to fly out of Bucharest the following day. In that case, it makes sense for the agent to adopt the goal of getting to Bucharest. Courses of action that don't reach Bucharest on time can be rejected without further consideration and the agent's decision problem is greatly simplified. Goals help organize behavior by limiting the objectives that the agent is trying to achieve. Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.

GOAL FORMULATION

We will consider a goal to be a set of world states—exactly those states in which the goal is satisfied. The agent's task is to find out which sequence of actions will get it to a goal state. Before it can do this, it needs to decide what sorts of actions and states to consider. If it were to try to consider actions at the level of "move the left foot forward an inch" or "turn the steering wheel one degree left," the agent would probably never find its way out of the parking lot, let alone to Bucharest, because at that level of detail there is too much uncertainty in the world and there would be too many steps in a solution. Problem formulation is the process of deciding what actions and states to consider, given a goal. We will discuss this process in more detail later. For now, let us assume that the agent will consider actions at the level of driving from one major town to another. The states it will consider therefore correspond to being in a particular town.[1]

PROBLEM FORMULATION

Our agent has now adopted the goal of driving to Bucharest, and is considering where to go from Arad. There are three roads out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind. None of these achieves the goal, so unless the agent is very familiar with the geography of Romania, it will not know which road to follow.[2] In other words, the agent will not know which of its possible actions is best, because it does not know enough about the state that results from taking each action. If the agent has no additional knowledge, then it is stuck. The best it can do is choose one of the actions at random.

But suppose the agent has a map of Romania, either on paper or in its memory. The point of a map is to provide the agent with information about the states it might get itself into, and the actions it can take. The agent can use this information to consider subsequent stages of a hypothetical journey via each of the three towns, trying to find a journey that eventually gets to Bucharest. Once it has found a path on the map from Arad to Bucharest, it can achieve its goal by carrying out the driving actions that correspond to the legs of the journey. In general, *an agent with several immediate options of unknown value can decide what to do by first examining different possible* sequences *of actions that lead to states of known value, and then choosing the best sequence.*

SEARCH
SOLUTION
EXECUTION

This process of looking for such a sequence is called search. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the execution phase. Thus, we have a simple "formulate, search, execute" design for the agent, as shown in Figure 3.1. After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as

[1] Notice that each of these "states" actually corresponds to a large ***set*** of world states, because a real world state specifies every aspect of reality. It is important to keep in mind the distinction between states in problem solving and world states.

[2] We are assuming that most readers are in the same position and can easily imagine themselves to be as clueless as our agent. We apologize to Romanian readers who are unable to take advantage of this pedagogical device.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over. Note that when it is executing the sequence it ignores its percepts: it assumes that the solution it has found will always work.

the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal.

We first describe the process of problem formulation, and then devote the bulk of the chapter to various algorithms for the SEARCH function. We will not discuss the workings of the UPDATE-STATE and FORMULATE-GOAL functions further in this chapter.

Before plunging into the details, let us pause briefly to see where problem-solving agents fit into the discussion of agents and environments in Chapter 2. The agent design in Figure 3.1 assumes that the environment is static, because formulating and solving the problem is done without paying attention to any changes that might be occurring in the environment. The agent design also assumes that the initial state is known; knowing it is easiest if the environment is observable. The idea of enumerating "alternative courses of action" assumes that the environment can be viewed as discrete. Finally, and most importantly, the agent design assumes that the environment is deterministic. Solutions to problems are single sequences of actions, so they cannot handle any unexpected events; moreover, solutions are executed without paying attention to the percepts! An agent that carries out its plans with its eyes closed, so to speak., must be quite certain of what is going on. (Control theorists call this an open-loop system, because ignoring the percepts breaks the loop between agent and environment.) All these assumptions mean that we are dealing with the easiest kinds of environments, which is one reason this chapter comes early on in the book. Section 3.6 takes a brief look at what happens when we relax the assumptions of observability and determinism. Chapters 12 and 17 go into much greater depth.

OPEN-LOOP

Well-defined problems and solutions

PROBLEM

A problem can be defined formally by four components:

INITIAL STATE

- The **initial state** that the agent starts in. For example, the initial state for our agent in Romania might be described as $In(Arad)$.
- A description of the possible **actions** available to the agent. The most common formulation[3] uses a **successor function.** Given a particular state x, SUCCESSOR-FN(x) returns a set of (action, successor) ordered pairs, where each action is one of the legal actions in state x and each successor is a state that can be reached from x by applying the action. For example, from the state $In(Arad)$, the successor function for the Romania problem would return

SUCCESSOR FUNCTION

  $\{\langle Go(Sibiu), In(Sibiu)\rangle, \ \langle Go(Timisoara), In(Tzmisoara)\rangle, \langle Go(Zerind), In(Zerind)\rangle\}$

STATE SPACE

  Together, the initial state and successor function implicitly define the **state space** of the problem—the set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions. (The map of Romania shown in Figure 3.2 can be interpreted as a state space graph if we view each road as standing for two driving actions, one in each direction.) A **path** in the state space is a sequence of states connected by a sequence of actions.

PATH

GOAL TEST

- The **goal test,** which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set $\{In(Bucharest)\}$. Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called "checkmate," where the opponent's king is under attack and can't escape.

PATH COST

- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers. In this chapter, we assume that the cost of a path can be described as the sum of the costs of the individual actions along the path. The **step cost** of taking action a to go from state x to state y is denoted by $c(x, a, y)$. The step costs for Romania are shown in Figure 3.2 as route distances. We will assume that step costs are nonnegative.[4]

STEP COST

The preceding elements define a problem and can be gathered together into a single data structure that is given as input to a problem-solving algorithm. A **solution** to a problem is a path from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

OPTIMAL SOLUTION

Formulating problems

In the preceding section we proposed a formulation of the problem of getting to Bucharest in terms of the initial state, successor function, goal test, and path cost. This formulation seems

[3] An alternative formulation uses a set of **operators** that can be applied to a state to generate successors.

[4] The implications of negative costs are explored in Exercise 3.17.

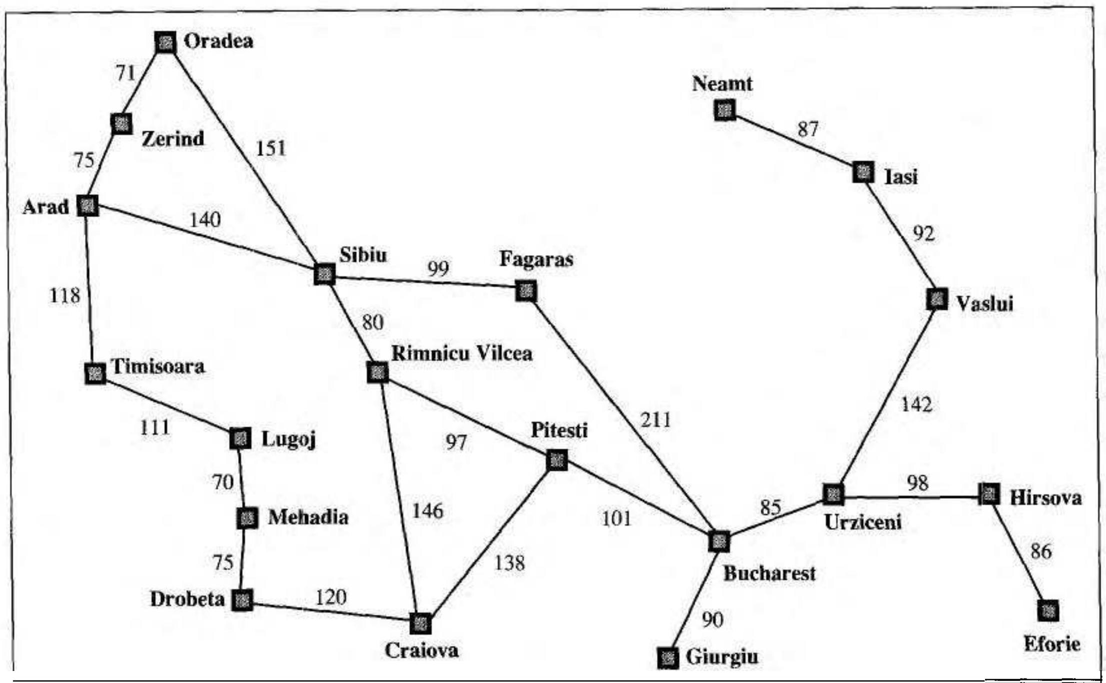

Figure 3.2 A simplified road map of part of Romania.

reasonable, yet it omits a great many aspects of the real world. Compare the simple state description we have chosen, *In(Arad)*, to an actual cross-country trip, where the state of the world includes so many things: the traveling companions, what is on the radio, the scenery out of the window, whether there are any law enforcement officers nearby, how far it is to the next rest stop, the condition of the road, the weather, and so on. All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest. The process of removing detail from a representation is called **abstraction.**

ABSTRACTION

In addition to abstracting the state description, we must abstract the actions themselves. A driving action has many effects. Besides changing the location of the vehicle and its occupants, it takes up time, consumes fuel, generates pollution, and changes the agent (as they say, travel is broadening). In our formulation, we take into account only the change in location. Also, there are many actions that we will omit altogether: turning on the radio, looking out of the window, slowing down for law enforcement officers, and so on. And of course, we don't specify actions at the level of "turn steering wheel to the left by three degrees."

Can we be more precise about defining the appropriate level of abstraction? Think of the abstract states and actions we have chosen as corresponding to large sets of detailed world states and detailed action sequences. Now consider a solution to the abstract problem: for example, the path from Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest. This abstract solution corresponds to a large number of more detailed paths. For example, we could drive with the radio on between Sibiu and Rimnicu Vilcea, and then switch it off for the rest of the trip. The abstraction is *valid* if we can expand any abstract solution into a solution in the more detailed world; a sufficient condition is that for every detailed state that is "in Arad," there is a detailed path to some state that is "in Sibiu," and so on. The abstraction is *useful* if carrying out each of the actions in the solution is easier than the original problem; in this case they are easy enough that they can be carried out without further search or planning by an average driving agent. The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

3.2 EXAMPLE PROBLEMS

The problem-solving approach has been applied to a vast array of task environments. We list some of the best known here, distinguishing between *toy* and *real-world* problems. A toy problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description. This means that it can be used easily by different researchers to compare the performance of algorithms. A real-world problem is one whose solutions people actually care about. They tend not to have a single agreed-upon description, but we will attempt to give the general flavor of their formulations.

TOY PROBLEM

REAL-WORLD PROBLEM

Toy problems

The first example we will examine is the vacuum world first introduced in Chapter 2. (See Figure 2.2.) This can be formulated as a problem as follows:

- ◇ States: The agent is in one of two locations, each of which might or might not contain dirt. Thus there are 2 x $2^2 = 8$ possible world states.
- ◇ Initial state: Any state can be designated as the initial state.
- ◇ Successor function: This generates the legal states that result from trying the three actions (Left, ***Right,*** and ***Suck).*** The complete state space is shown in Figure 3.3.
- ◇ Goal test: This checks whether all the squares are clean.
- ◇ Path cost: Each step costs 1, so the path cost is the number of steps in the path.

Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets messed up once cleaned. (In Section 3.6, we will relax these assumptions.) One important thing to note is that the state is determined by both the agent location and the dirt locations. A larger environment with n locations has n 2^n states.

8-PUZZLE

The 8-puzzle, an instance of which is shown in Figure 3.4, consists of a 3 x 3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

- ◇ **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- ◇ Initial state: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states (Exercise 3.4).

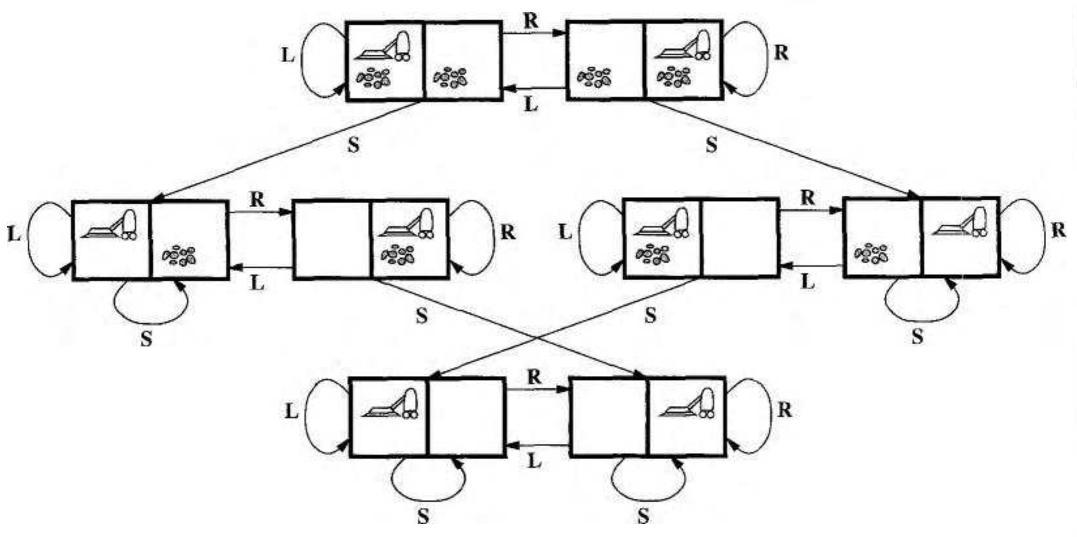

Figure 3.3 The state space for the vacuum world. Arcs denote actions: L = *Left*, R = *Right*, S = Suck.

◇ **Successor function:** This generates the legal states that result from trying the four actions (blank moves Left, Right, Up, or Down).

◇ **Goal test:** This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)

◇ **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

What abstractions have we included here? The actions are abstracted to their beginning and final states, ignoring the intermediate locations where the block is sliding. We've abstracted away actions such as shaking the board when pieces get stuck, or extracting the pieces with a knife and putting them back again. We're left with a description of the rules of the puzzle, avoiding all the details of physical manipulations.

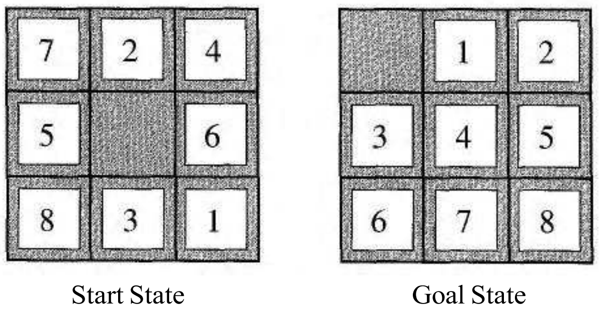

Figure 3.4 A typical instance of the 8-puzzle.

SLIDING-BLOCK PUZZLES

The 8-puzzle belongs to the family of **sliding-block puzzles,** which are often used as test problems for new search algorithms in AI. This general class is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next. The 8-puzzle has $9!/2 = 181{,}440$ reachable states and is easily solved. The 15-puzzle (on a 4 x 4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms. The 24-puzzle (on a 5 x 5 board) has around 10^{25} states, and random instances are still quite difficult to solve optimally with current machines and algorithms.

8-QUEENS PROBLEM

The goal of the **8-queens problem** is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.) Figure 3.5 shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.

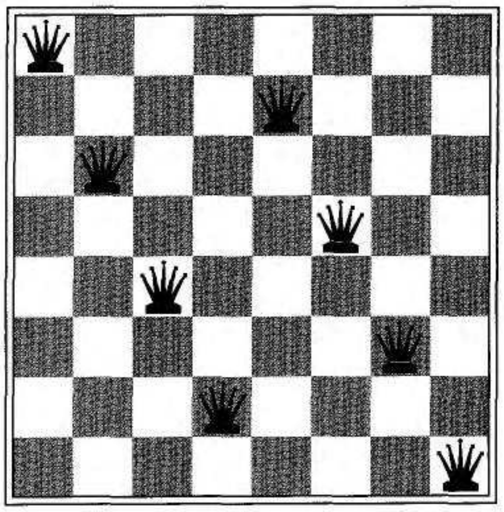

Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

Although efficient special-purpose algorithms exist for this problem and the whole n-queens family, it remains an interesting test problem for search algorithms. There are two main kinds of formulation. An **incremental formulation** involves operators that *augment* the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state. A **complete-state formulation** starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts. The first incremental formulation one might try is the following:

INCREMENTAL FORMULATION

COMPLETE-STATE FORMULATION

- ◇ **States:** Any arrangement of 0 to 8 queens on the board is a state.
- ◇ **Initial state:** No queens on the board.
- ◇ **Successor function:** Add a queen to any empty square.
- ◇ **Goal test:** **8** queens are on the board, none attacked.

In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

◇ States: Arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another are states.

◇ Successor function: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queens state space from 3×10^{14} to just 2,057, and solutions are easy to find. On the other hand, for 100 queens the initial formulation has roughly 10^{400} states whereas the improved formulation has about 10^{52} states (Exercise 3.5). This is a huge reduction, but the improved state space is still too big for the algorithms in this chapter to handle. Chapter 4 describes the complete-state formulation and Chapter 5 gives a simple algorithm that makes even the million-queens problem easy to solve.

Real-world problems

ROUTE-FINDING PROBLEM

We have already seen how the route-finding problem is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and airline travel planning systems. These problems are typically complex to specify. Consider a simplified example of an airline travel problem specified as follows:

◇ States: Each is represented by a location (e.g., an airport) and the current time.

◇ Initial state: This is specified by the problem.

◇ Successor **function**: This returns the states resulting from taking any scheduled flight (perhaps further specified by seat class and location), leaving later than the current time plus the within-airport transit time, from the current airport to another.

◇ Goal test: Are we at the destination by some prespecified time?

◇ Path cost: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

Commercial travel advice systems use a problem formulation of this kind, with many additional complications to handle the byzantine fare structures that airlines impose. Any seasoned traveller knows, however, that not all air travel goes according to plan. A really good system should include contingency plans—such as backup reservations on alternate flights—to the extent that these are justified by the cost and likelihood of failure of the original plan.

TOURING PROBLEMS

Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem, "Visit every city in Figure 3.2 at least once, starting and ending in Bucharest." As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the *set of cities the agent has visited.* So the initial state would be "In Bucharest; visited {Bucharest}," a typical intermediate state would be "In Vaslui; visited {Bucharest,Urziceni,Vaslui}," and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

TRAVELING SALESPERSON PROBLEM

The **traveling salesperson problem** (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

VLSI LAYOUT

A VLSI **layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase, and is usually split into two parts: **cell layout** and **channel routing.** In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving. In Chapter 4, we will see some algorithms capable of solving them.

ROBOT NAVIGATION

Robot navigation is a generalization of the route-finding problem described earlier. Rather than a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite. We examine some of these methods in Chapter 25. In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls.

AUTOMATIC ASSEMBLY SEQUENCING

Automatic assembly sequencing of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972). Progress since then has been slow but sure, to the point where the assembly of intricate objects such as electric motors is economically feasible. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal successors is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space. Another important assembly problem is **protein design,** in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

PROTEIN DESIGN

INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform **Internet searching,** looking for answers to questions, for related information, or for shopping deals. This is a good application for search techniques, because it is easy to conceptualize the Internet as a graph of nodes (pages) connected by links. A full description of Internet search is deferred until Chapter 10.

3.3 SEARCHING FOR SOLUTIONS

Having formulated some problems, we now need to solve them. This is done by a search through the state space. This chapter deals with search techniques that use an explicit **search tree** that is generated by the initial state and the successor function that together define the state space. In general, we may have a search *graph* rather than a search *tree,* when the same state can be reached from multiple paths. We defer consideration of this important complication until Section 3.5.

SEARCH TREE

Figure 3.6 shows some of the expansions in the search tree for finding a route from Arad to Bucharest. The root of the search tree is a **search node** corresponding to the initial state, *In(Arad).* The first step is to test whether this is a goal state. Clearly it is not, but it is important to check so that we can solve trick problems like "starting in Arad, get to Arad." Because this is not a goal state, we need to consider some other states. This is done by **expanding** the current state; that is, applying the successor function to the current state, thereby **generating** a new set of states. In this case, we get three new states: *In(Sibiu), In(Timisoara),* and *In(Zerind).* Now we must choose which of these three possibilities to consider further.

SEARCH NODE

EXPANDING

GENERATING

This is the essence of search — following up one option now and putting the others aside for later, in case the first choice does not lead to a solution. Suppose we choose Sibiu first. We check to see whether it is a goal state (it is not) and then expand it to get *In(Arad), In(Fagaras), In(Oradea),* and *In(RimnicuVilcea).* We can then choose any of these four, or go back and choose Timisoara or Zerind. We continue choosing, testing, and expanding until either a solution is found or there are no more states to be expanded. The choice of which state to expand is determined by the **search strategy.** The general tree-search algorithm is described informally in Figure 3.7.

SEARCH STRATEGY

It is important to distinguish between the state space and the search tree. For the route finding problem, there are only 20 states in the state space, one for each city. But there are an infinite number of paths in this state space, so the search tree has an infinite number of nodes. For example, the lhree paths Arad–Sibiu, Arad–Sibiu–Arad, Arad–Sibiu–Arad–Sibiu are the first three of an infinite sequence of paths. (Obviously, a good search algorithm avoids following such repeated paths; Section 3.5 shows how.)

There are many ways to represent nodes, but we will assume that a node is a data structure with five components:

- STATE: the state in the state space to which the node corresponds;
- PARENT-NODE: the node in the search tree that generated this node;
- ACTION: the action that was applied to the parent to generate the node;
- PATH-COST: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers; and
- DEPTH: the number of steps along the path from the initial state.

It is important to remember the distinction between nodes and states. A node is a boolkkeeping 'data structure used to represent the search tree. A state corresponds to a configuration of the world. Thus, nodes are on particular paths, as defined by PARENT-NODE pointers, whereas states are not. Furthermore, two different nodes can contain the same world state, if that state is generated via two different search paths. The node data structure is depicted in Figure 3.8.

FRINGE
LEAF NODE

We also need to represent the collection of nodes that have been generated but not yet expanded—this collection is called the **fringe.** Each element of the fringe is a **leaf node,** that

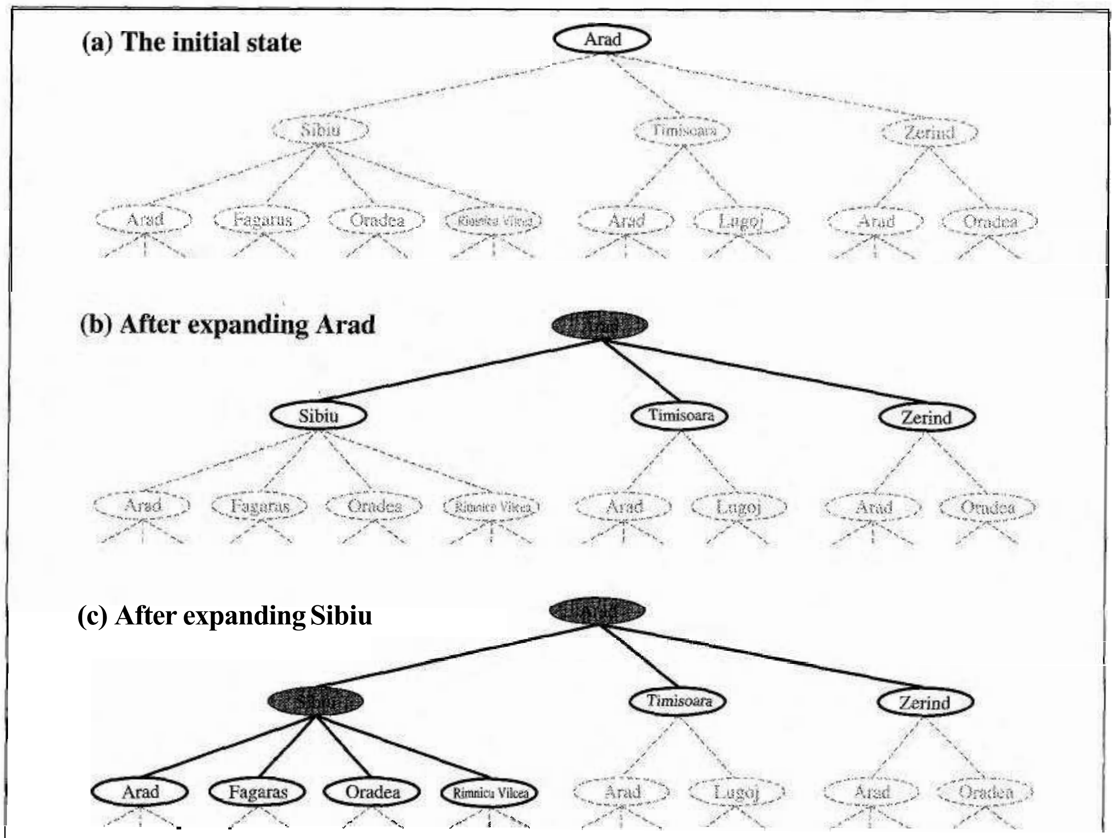

Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  ioop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

Figure 3.7 An informal description of the general tree-search algorithm.

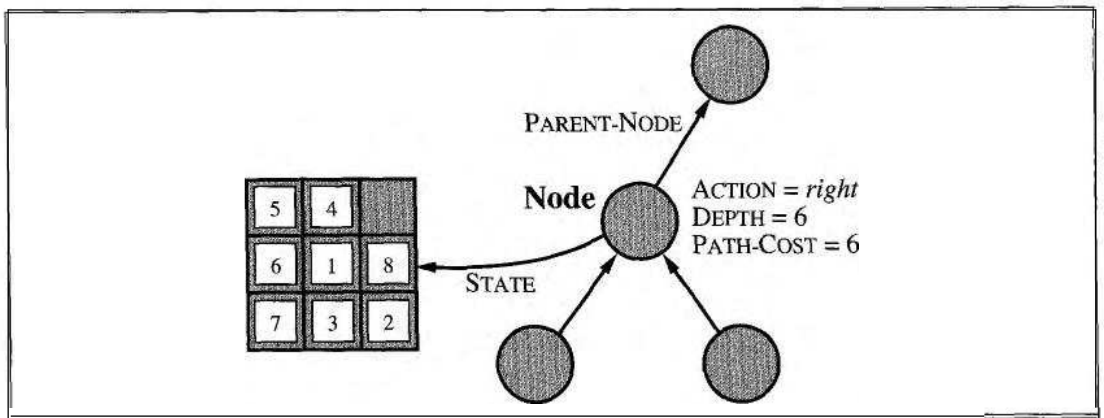

Figure 3.8 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

is, a node with no successors in the tree. In Figure 3.6, the fringe of each tree consists of those nodes with bold outlines. The simplest representation of the fringe would be a set of nodes. The search strategy then would be a function that selects the next node to be expanded from this set. Although this is conceptually straightforward, it could be computationally expensive, because the strategy function might have to look at every element of the set to choose the best one. Therefore, we will assume that the collection of nodes is implemented as a **queue.** The operations on a queue are as follows:

QUEUE

- MAKE-QUEUE(*element*, ...) creates a queue with the given element(s).
- EMPTY?(*queue*) returns true only if there are no more elements in the queue.
- FIRST(*queue*) returns the first element of the queue.
- REMOVE-FIRST(*queue*) returns FIRST(*queue*) and removes it from the queue.
- INSERT(*element*, queue) inserts an element into the queue and returns the resulting queue. (We will see that different types of queues insert elements in different orders.)
- INSERT-ALL(*elements*, queue) inserts a set of elements into the queue and returns the resulting queue.

With these definitions, we can write the more formal version of the general tree-search algorithm shown in Figure 3.9.

Measuring problem-solving performance

The output of a problem-solving algorithm is either *failure* or a solution. (Some algorithms might get stuck in an infinite loop and never return an output.) We will evaluate an algorithm's performance in four ways:

COMPLETENESS
◇ **Completeness:** Is the algorithm guaranteed to find a solution when there is one?

OPTIMALITY
◇ **Optimality:** Does the strategy find the optimal solution, as defined on page 62?

TIME COMPLEXITY
◇ **Time complexity:** How long does it take to find a solution?

SPACE COMPLEXITY
◇ **Space complexity:** How much memory is needed to perform the search?

```
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(STATE[node], action, result)
      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.

BRANCHING FACTOR

Time is often measured in terms of the number of nodes generated[5] during the search, and space in terms of the maximum number of nodes stored in memory.

SEARCH COST

TOTAL COST

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

UNINFORMED SEARCH

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 nongoal 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.

INFORMED SEARCH

HEURISTIC SEARCH

Breadth-first search

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

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

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.*

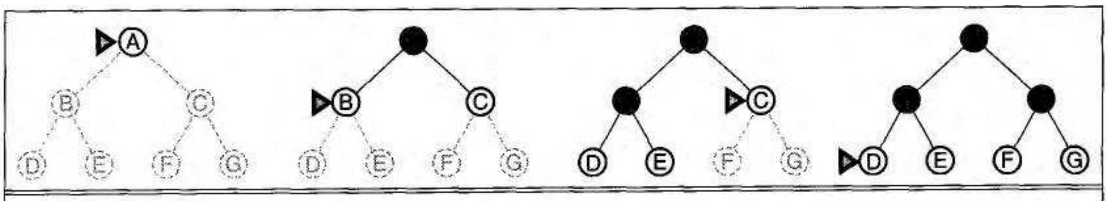

Figure 3.10 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

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 terabytes
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

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 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 $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, 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, $b^{1+\lfloor C^*/\epsilon \rfloor}$ is just b^d.

Depth-first search

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 $bm + 1$ 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.

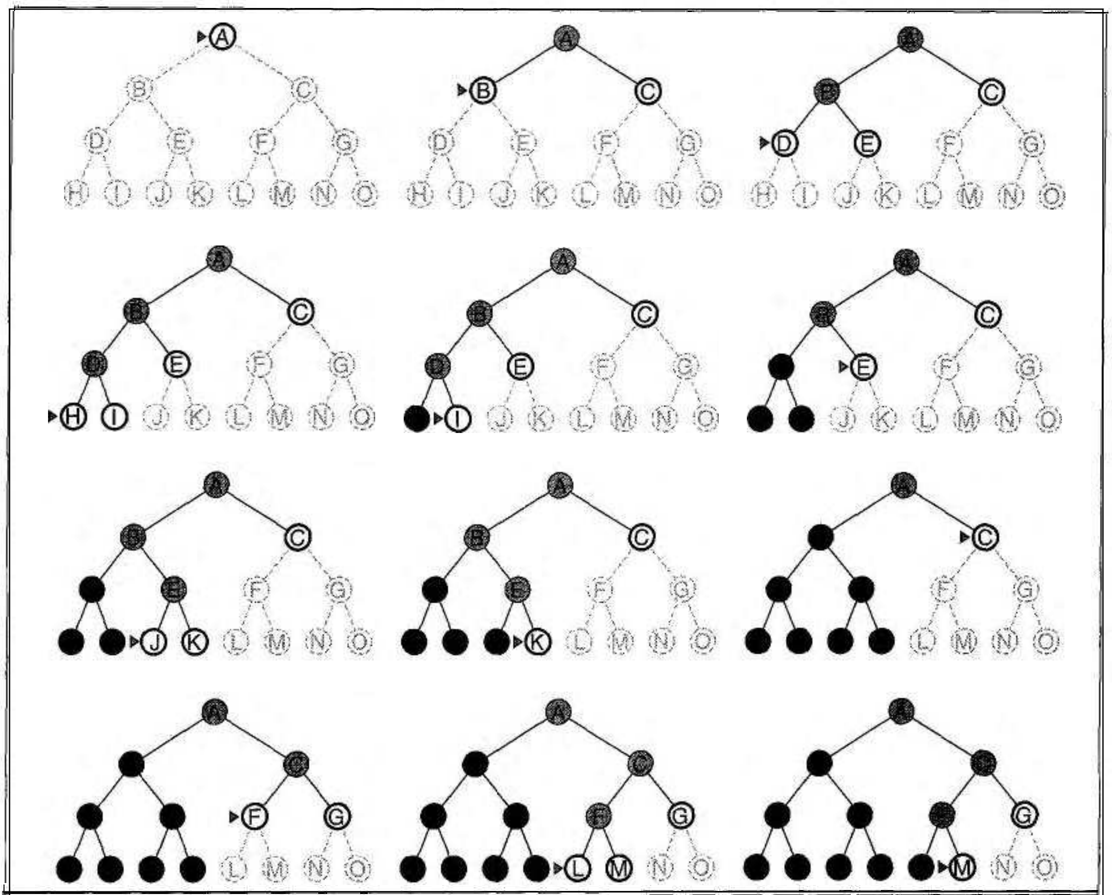

Figure 3.12 Depth-first search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

BACKTRACKING SEARCH

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 $O(bm)$. 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 failurelcutoff
  return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  cutoff_occurred? ← 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 ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff_occurred? ← true
      else if result ≠ failure then return result
  if cutoff_occurred? 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. DEPTH-LIMITED SEARCH 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 $O(b\ell)$. 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 DIAMETER 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 ı-ecursive 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

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

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

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

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

$$\begin{aligned} N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 \end{aligned}$$

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 ≠ cutoff 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.

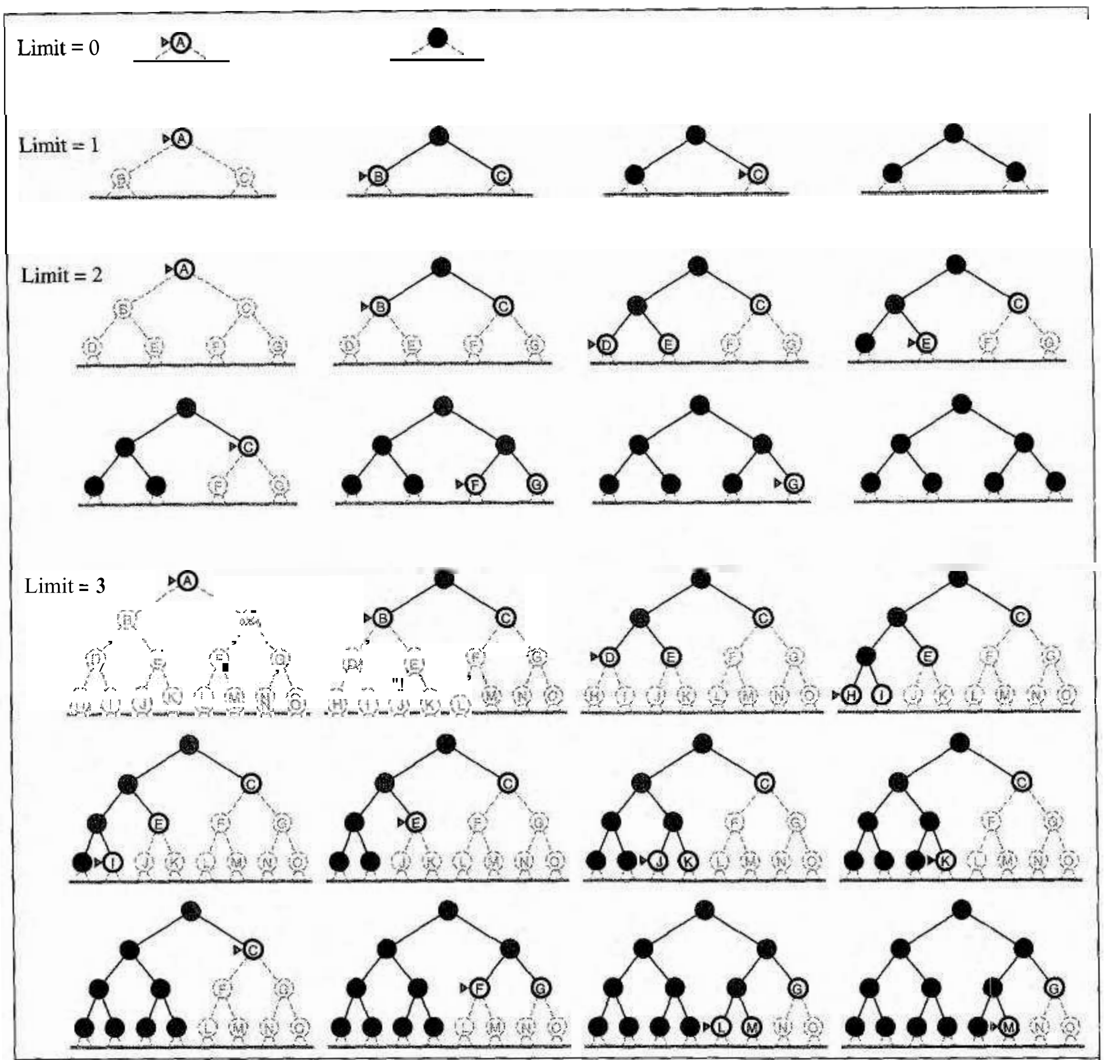

Figure 3.15 Four iterations of iterative deepening search on a binary tree.

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.

ITERATIVE LENGTHENING 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

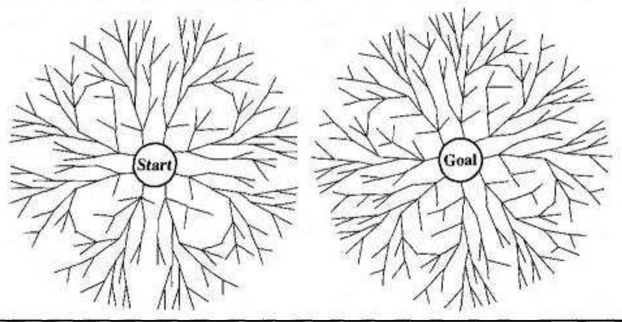

Figure 3.16 A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

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.

PREDECESSORS

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 state x, $Pred(x)$, be all those states that have x as a successor. Bidirectional search requires that $Pred(x)$ be efficiently computable. The easiest case is when all the actions in the state space are reversible, so that $Pred(x) = Succ(x)$. 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 satisfying 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	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
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

RECTANGULAR GRID

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 $O(b^d)$.

CLOSED LIST

OPEN LIST

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

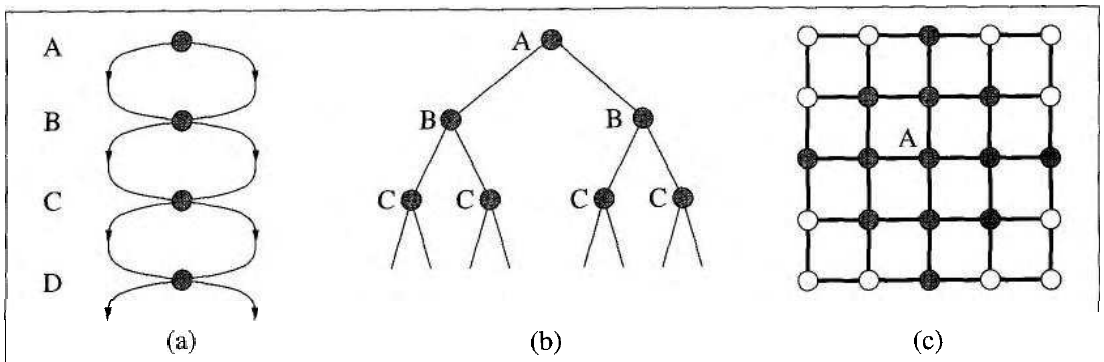

Figure 3.18 State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains d + 1 states, where d is the maximum depth. (b) The corresponding search tree, which has 2^d 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.

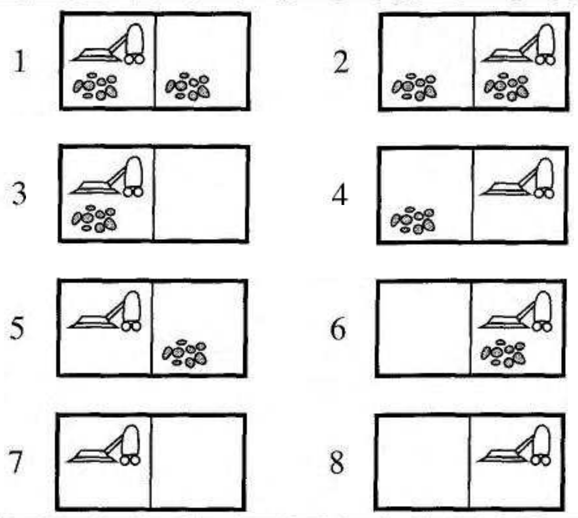

Figure 3.20 The eight possible states of the vacuum world.

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.)

COERCION

BELIEF STATE

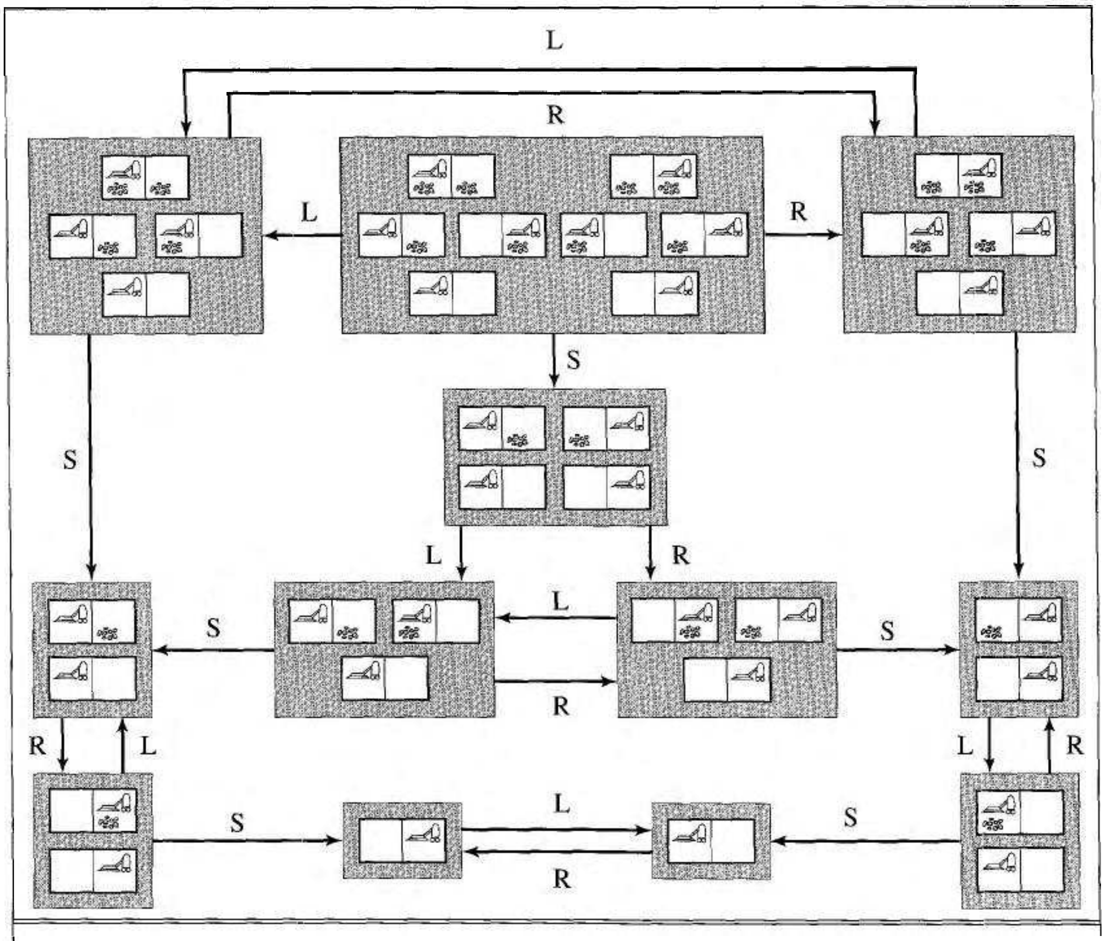

Figure 3.21 The reachable portion of the belief state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete 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 unioning 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* **i** *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

CONTINGENCY PROBLEM

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 modem, efficient home appliances who cannot take advantage of this pedagogical device.

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).

INTERLEAVING

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 $O(b^{d+1})$. 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 the 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 (1957) 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 AI 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 Story, 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 may safely be said to have engaged the attention of nine out of ten persons of both sexes and all ages and conditions of the community. But this would not have weighed with the editors to induce them to insert articles upon such a subject in the *American Journal of Mathematics*, but 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.

Uninformed 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 Artijicial 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 LEGAL-ACTIONS(~)denotes the set of actions that are legal in state s, and 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 Berlekamp 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 $\sqrt[3]{n!}$ 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.)

3.6 Does a finite state space always lead to a finite search tree? How about a finite state space that is a tree? Can you be more precise about what types of state spaces always lead to finite search trees? (Adapted from Bender, 1996.)

3.7 Give the initial state, goal test, successor function, and cost function for each of the following. Choose a formulation that is precise enough to be implemented.

a. You have to color a planar map using only four colors, in such a way that no two adjacent regions have the same color.

b. A 3-foot-tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. He would like to get the bananas. The room contains two stackable, movable, climbable 3-foot-high crates.

c. You have a program that outputs the message "illegal input record" when fed a certain file of input records. You know that processing of each record is independent of the other records. You want to discover what record is illegal.

d. You have three jugs, measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet. You can fill the jugs up or empty them out from one to another or onto the ground. You need to measure out exactly one gallon.

3.8 Consider a state space where the start state is number 1 and the successor function for state n returns two states, numbers $2n$ and $2n + 1$.

a. Draw the portion of the state space for states 1 to 15.

b. Suppose the goal state is 11. List the order in which nodes will be visited for breadth-first search, depth-limited search with limit 3, and iterative deepening search.

c. Would bidirectional search be appropriate for this problem? If so, describe in detail how it would work.

d. What is the branching factor in each direction of the bidirectional search?

e. Does the answer to (c) suggest a reformulation of the problem that would allow you to solve the problem of getting from state 1 to a given goal state with almost no search?

3.9 The **missionaries and cannibals** problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint (Amarel, 1968).

a. Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the complete state space.

b. Implement and solve the problem optimally using an appropriate search algorithm. Is it a good idea to check for repeated states?

c. Why do you think people have a hard time solving this puzzle, given that the state space is so simple?

3.10 Implement two versions of the successor function for the 8-puzzle: one that generates all the successors at once by copying and editing the 8-puzzle data structure, and one that generates one new successor each time it is called and works by modifying the parent state directly (and undoing the modifications as needed). Write versions of iterative deepening depth-first search that use these functions and compare their performance.

3.11 On page 79, we mentioned **iterative lengthening search,** an iterative analog of uniform cost search. The idea is to use increasing limits on path cost. If a node is generated whose path cost exceeds the current limit, it is immediately discarded. For each new iteration, the limit is set to the lowest path cost of any node discarded in the previous iteration.

a. Show that this algorithm is optimal for general path costs.

b. Consider a uniform tree with branching factor b, solution depth d, and unit step costs. How many iterations will iterative lengthening require?

c. Now consider step costs drawn from the continuous range $[0, 1]$ with a minimum positive cost ϵ. How many iterations are required in the worst case?

d. Implement the algorithm **and** apply it to instances of the 8-puzzle and traveling salesperson problems. Compare the algorithm's performance to that of uniform-cost search, and comment on your results.

3.12 Prove that uniform-cost search and breadth-first search with constant step costs are optimal when used with the GRAPH-SEARCH algorithm. Show a state space with varying step costs in which GRAPH-SEARCH using iterative deepening finds a suboptimal solution.

3.13 Describe a state space in which iterative deepening search performs much worse than depth-first search (for example, $O(n^2)$ vs. $O(n)$).

3.14 Write a program that will take as input two Web page URLs and find a path of links from one to the other. What is an appropriate search strategy? Is bidirectional search a good idea? Could a search engine be used to implement a predecessor function?

3.15 Consider the problem of finding the shortest path between two points on a plane that has convex polygonal obstacles as shown in Figure 3.22. This is an idealization of the problem that a robot has to solve to navigate its way around a crowded environment.

a. Suppose the state space consists of all positions (x,y) in the plane. How many states are there? How many paths are there to the goal?

b. Explain briefly why the shortest path from one polygon vertex to any other in the scene must consist of straight-line segments joining some of the vertices of the polygons. Define a good state space now. How large is this state space?

c. Define the necessary functions to implement the search problem, including a successor function that takes a vertex as input and returns the set of vertices that can be reached in a straight line from the given vertex. (Do not forget the neighbors on the same polygon.) Use the straight-line distance for the heuristic function.

d. Apply one or more of the algorithms in this chapter to solve a range of problems in the domain, and comment on their performance.

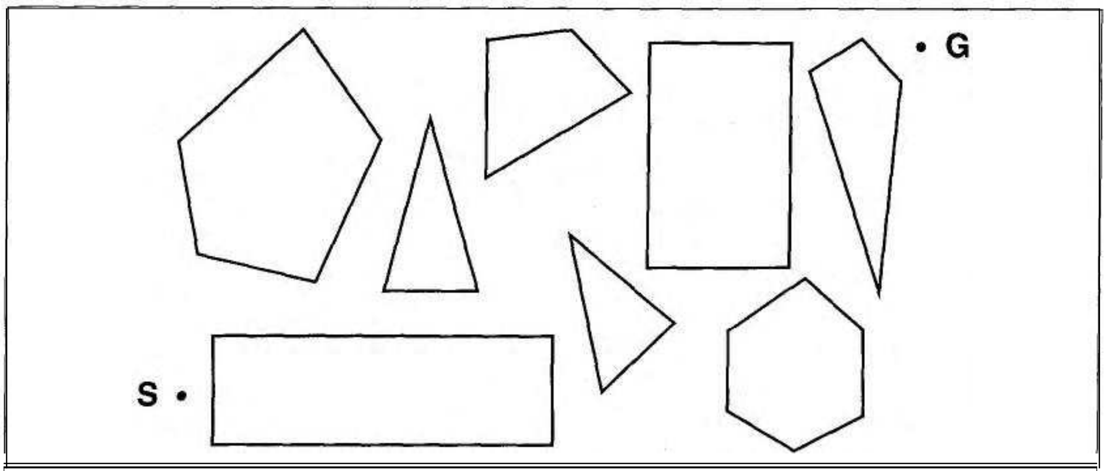

Figure 3.22 A scene with polygonal obstacles.

3.16 We can turn the navigation problem in Exercise 3.15 into an environment as follows:

- The percept will be a list of the positions, relative to the agent, of the visible vertices. The percept does not include the position of the robot! The robot must learn its own position from the map; for now, you can assume that each location has a different "view."
- Each action will be a vector describing a straight-line path to follow. If the path is unobstructed, the action succeeds; otherwise, the robot stops at the point where its path first intersects an obstacle. If the agent returns a zero motion vector and is at the goal (which is fixed and known), then the environment should teleport the agent to a random location (not inside an obstacle).
- The performance measure charges the agent 1 point for each unit of distance traversed and awards 1000 points each time the goal is reached.

a. Implement this environment and a problem-solving agent for it. The agent will need to formulate a new problem after each teleportation, which will involve discovering its current location.

b. Document your agent's performance (by having the agent generate suitable commentary as it moves around) and report its performance over 100 episodes.

c. Modify the environment so that 30% of the time the agent ends up at an unintended destination (chosen randomly from the other visible vertices if any, otherwise no move at all). This is a crude model of the motion errors of a real robot. Modify the agent so that when such an error is detected, it finds out where it is and then constructs a plan to get back to where it was and resume the old plan. Remember that sometimes getting back to where it was might also fail! Show an example of the agent successfully overcoming two successive motion errors and still reaching the goal.

d. Now try two different recovery schemes after an error: (1) Head for the closest vertex on the original route; and (2) replan a route to the goal from the new location. Compare the performance of the three recovery schemes. Would the inclusion of search costs affect the comparison?

e. Now suppose that there are locations from which the view is identical. (For example, suppose the world is a grid with square obstacles.) What kind of problem does the agent now face? What do solutions look like?

3.17 On page 62, we said that we would not consider problems with negative path costs. In this exercise, we explore this in more depth.

a. Suppose that actions can have arbitrarily large negative costs; explain why this possibility would force any optimal algorithm to explore the entire state space.

b. Does it help if we insist that step costs must be greater than or equal to some negative constant *c?* Consider both trees and graphs.

c. Suppose that there is a set of operators that form a loop, so that executing the set in some order results in no net change to the state. If all of these operators have negative cost, what does this imply about the optimal behavior for an agent in such an environment?

d. One can easily imagine operators with high negative cost, even in domains such as route finding. For example, some stretches of road might have such beautiful scenery as to far outweigh the normal costs in terms of time and fuel. Explain, in precise terms, within the context of state-space search, why humans do not drive round scenic loops indefinitely, and explain how to define the state space and operators for route finding so that artificial agents can also avoid looping.

e. Can you think of a real domain in which step costs are such as to cause looping?

3.18 Consider the sensorless, two-location vacuum world under Murphy's Law. Draw the belief state space reachable from the initial belief state $\{1,2,3,4,5,6,7,8\}$, and explain why the problem is unsolvable. Show also that if the world is fully observable then there is a solution sequence for each possible initial state.

3.19 Consider the vacuum-world problem defined in Figure 2.2.

a. Which of the algorithms defined in this chapter would be appropriate for this problem? Should the algorithm check for repeated states?

b. Apply your chosen algorithm to compute an optimal sequence of actions for a 3 x **3** world whose initial state has dirt in the three top squares and the agent in the center.

c. Construct a search agent for the vacuum world, and evaluate its performance in a set of **3** x **3** worlds with probability 0.2 of dirt in each square. Include the search cost as well as path cost in the performance measure, using a reasonable exchange rate.

d. Compare your best search agent with a simple randomized reflex agent that sucks if there is dirt and otherwise moves randomly.

e. Consider what would happen if the world were enlarged to n x n. How does the performance of the search agent and of the reflex agent vary with n?

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

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.

BEST-FIRST SEARCH

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.

EVALUATION FUNCTION

The name "best-first search" is a venerable but inaccurate one. After all, if we could *really* expand the best node first, it would not be a search at all; it would be a straight march to the goal. All we can do is choose the node that ***appears*** to be best according to the evaluation function. If the evaluation function is 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.

HEURISTIC FUNCTION

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

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

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

Greedy best-first search[3] 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: $f(n) = h(n)$.

STRAIGHT-LINE DISTANCE

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.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

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

[1] Exercise 4.3 asks you to show that this family includes several familiar uninformed algorithms.

[2] heuristic function $h(n)$ takes a node as input, but it 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).

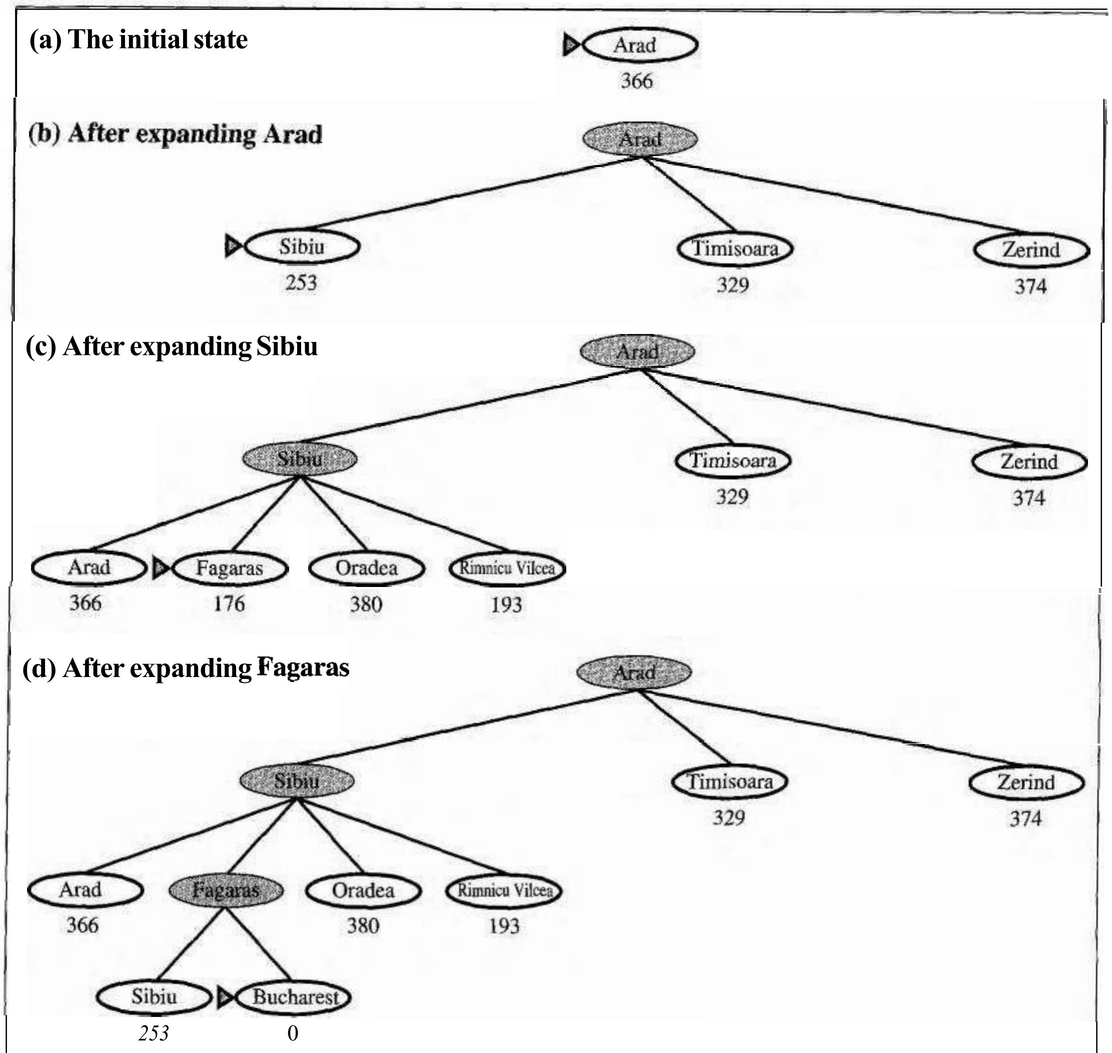

Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

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 Rimnicu 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

A* SEARCH

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 estirnated 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.

ADMISSIBLE HEURISTIC

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 a

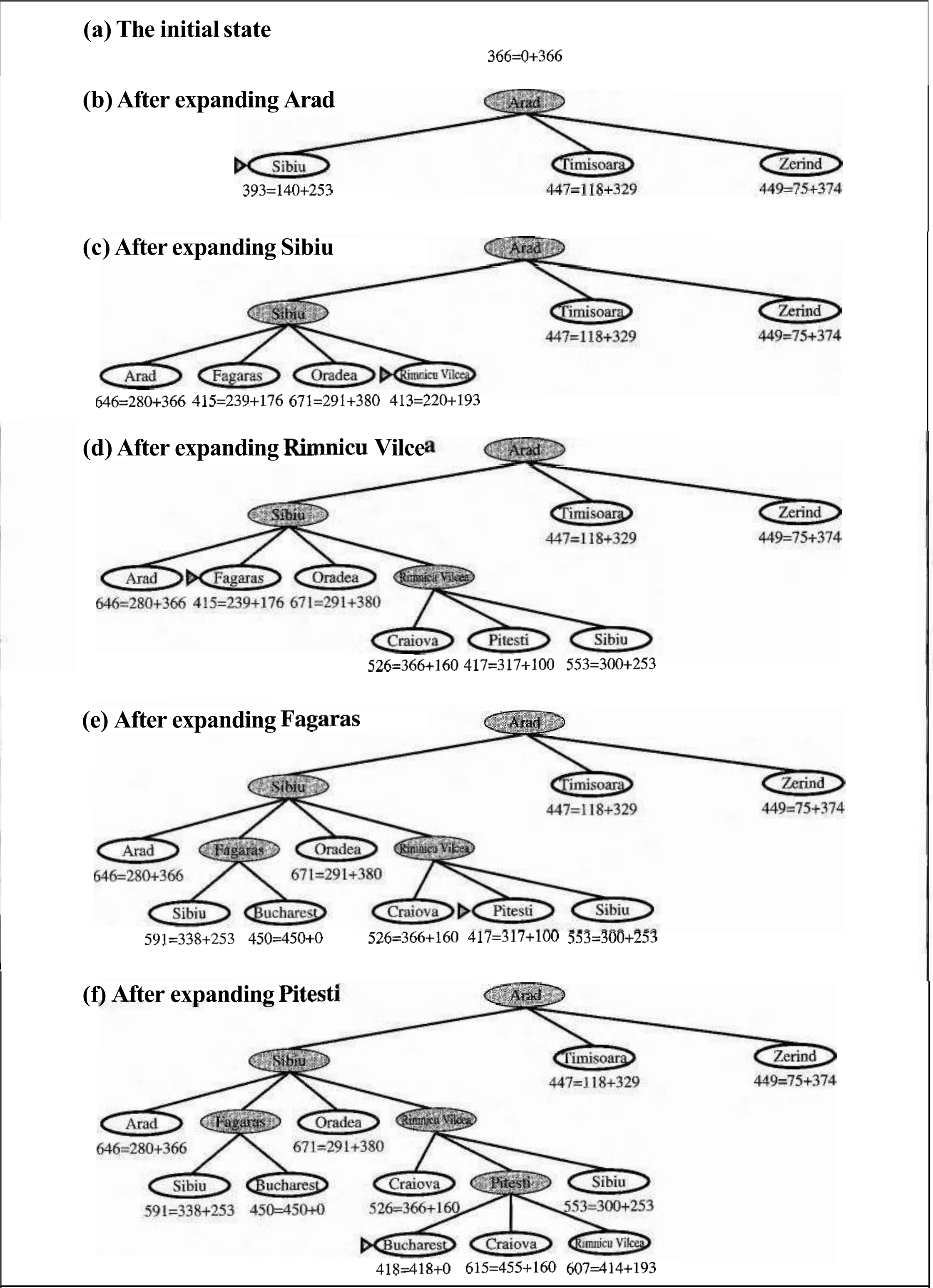

Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

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

CONSISTENCY

MONOTONICITY

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':

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

TRIANGLE INEQUALITY

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* off (n)*along any path are nondecueasing.* 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.

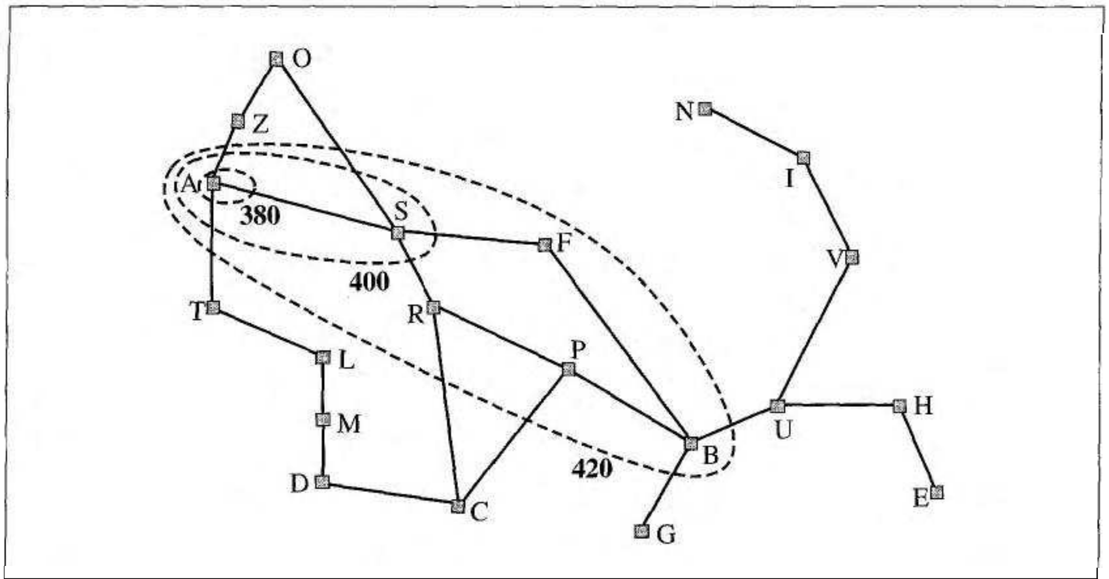

Figure 4.4 Map of Romania showing contours at $f = 380$, $f = 400$ and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

CONTOURS

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

PRUNING

Timisoara is **pruned;** because h_{SLD} is admissible, the algorithm can safely ignore this subtree

[4] Completeness requires that there be only finitely many nodes with cost less than or equal to C^*, a condition that is true if all step costs exceed some finite ϵ and if b is finite.

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

OPTIMALLY EFFICIENT

One final observation is that among optimal algorithms of this type—algorithms that extend search paths from the root—A* is **optimally efficient** for any given heuristic function. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A* (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

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

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

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

```
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[node]) then return node
    successors $\leftarrow$ EXPAND(node, problem)
    if successors is empty then return failure, $\infty$
    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.

than 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, RBFS 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 regeneration. 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 linear in the depth of the deepest optimal solution, 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,

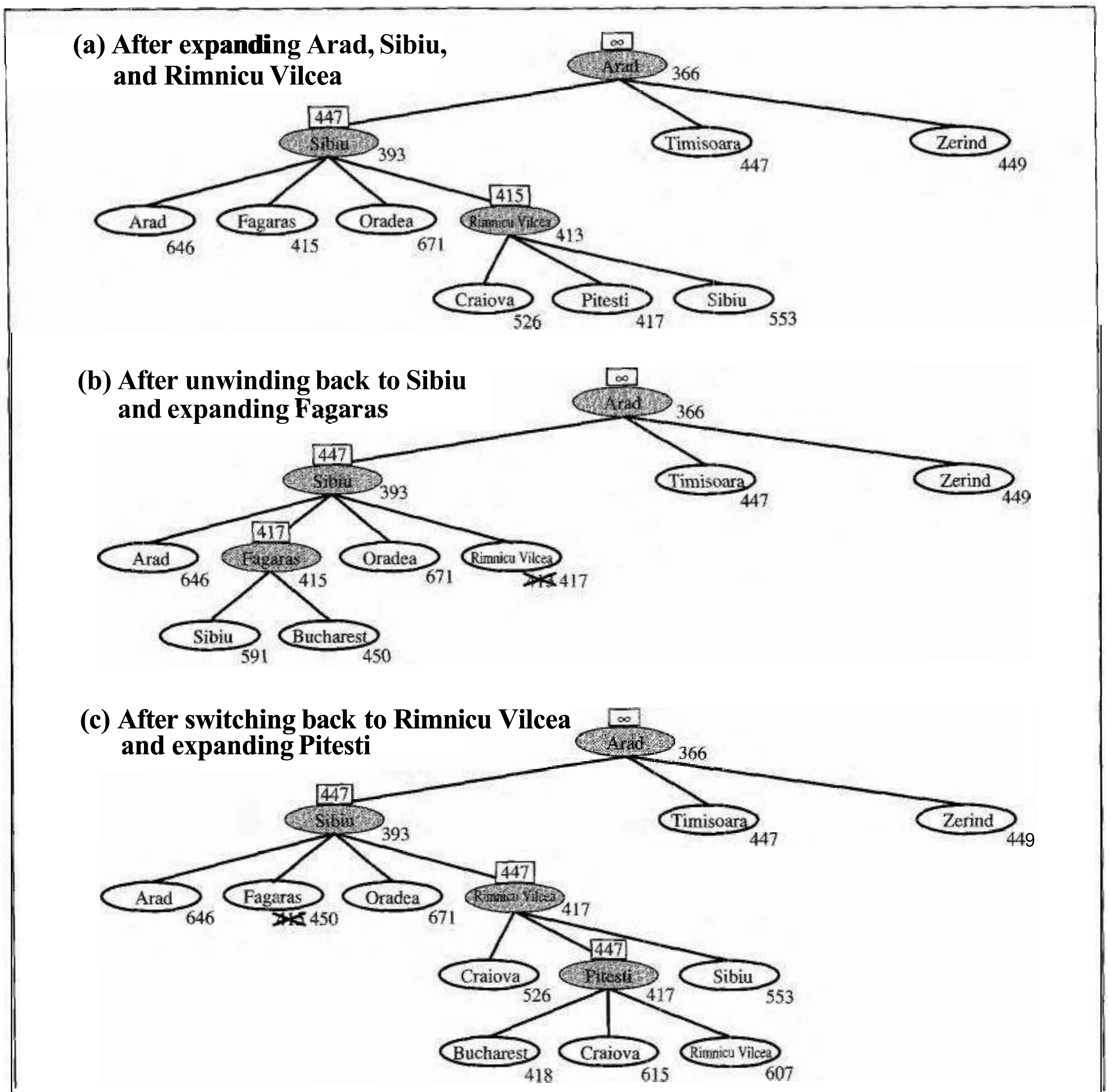

Figure 4.6 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (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 linear space: 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

MA*
SMA*

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 otherpaths* 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.

THRASHING

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

METALEVEL STATE SPACE

OBJECT-LEVEL STATE SPACE

[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.

METALEVEL LEARNING

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).

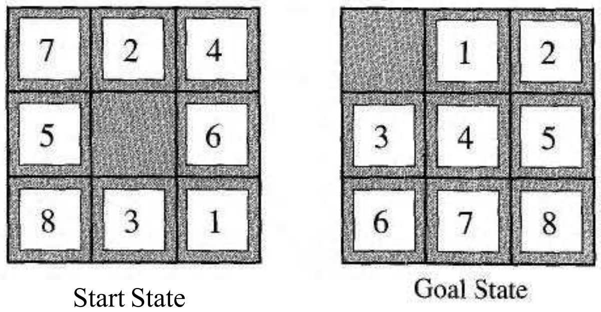

Figure 4.7 A typical instance of the 8-puzzle. The solution is 26 steps long.

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:

- 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

MANHATTAN DISTANCE

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

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

The effect of heuristic accuracy on performance

EFFECTIVE BRANCHING FACTOR

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,

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

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 generated 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.

DOMINATION

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 4.8 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the &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 relaxedproblem 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 and therefore must be at least as expensive as the optimal 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).

RELAXED PROBLEM

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 the proper score if we moved each tile in turn to its destination. The heuristic derived from (b) is discussed in Exercise 4.9. From (c), we can derive h_1 (misplaced tiles), because it would be the proper score if tiles could move to their intended destination in one step. Notice that it is crucial that the relaxed problems generated by this technique can be solved essentially ***without search,*** because the relaxed rules allow the problem to be decomposed into eight independent 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 (Prieditis, 1993). 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 one "clearly best" heuristic. If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem, and none of them dominates any of the others, which should we choose? As it turns out, we need not make a choice. We can have the best of all worlds, by defining

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

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

SUBPROBLEM

Admissible heuristics can also be derived from the solution cost of a **subproblem** of 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.

PATTERN DATABASES

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 the blank. (Notice that the locations of the other four tiles are irrelevant for the purposes of 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 sly." Thus, there is a tradeoff between accuracy and computation time for heuristic functions.

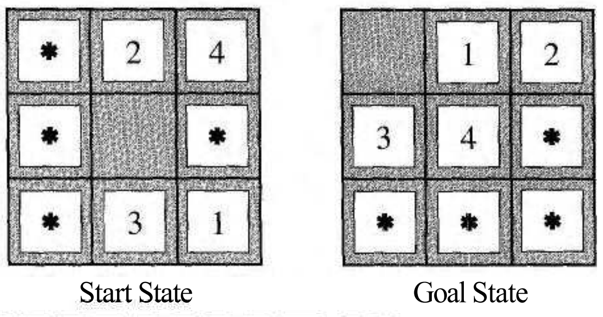

Figure 4.9 A subproblem of the 8-puzzle instance given in Figure 4.7. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

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 niot 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 numbes 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

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 cubies. 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 examples from which $h(n)$ can be learned. Each example consists of a state from the solution path and the actual cost of the solution from that point. From these examples, an **inductive learning** algorithm can be used to construct a function $h(n)$ that can (with luck) predict solution costs for other states that arise during search. Techniques for doing just this using neural nets, decision trees, and other methods are demonstrated in Chapter 18. (The reinforcement learning methods described in Chapter 21 are also applicable.)

FEATURES

Inductive learning methods work best when supplied with **features** of a state that are relevant to its evaluation, rather than with just the raw state description. For example, the feature "number of misplaced tiles" might be helpful in predicting the actual distance of a state from the goal. Let's call this feature $x_1(n)$. We could take 100 randomly generated 8-puzzle configurations and gather statistics on their actual solution costs. We might find that when $x_1(n)$ is 5, the average solution cost is around 14, and so on. Given these data, the value of x_1 can be used to predict $h(n)$. Of course, we can use several features. A second feature $x_2(n)$ might be "number of pairs of adjacent tiles that are also adjacent in the goal state." How should $x_1(n)$ and $x_2(n)$ be combined to predict $h(n)$? A common approach is to use a linear combination:

$$h(n) = c_1x_1(n) + c_2x_2(n) .$$

The constants c_1 and c_2 are adjusted to give the best fit to the actual data on solution costs. Presumably, c_1 should be positive and c_2 should be negative.

4.3 LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

The search algorithms that we have seen so far are designed to explore search spaces systematically. This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path and which have not. When a goal is found, the *path* to that goal also constitutes a solution to the problem.

In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem (see page 66), what matters is the final configuration of queens, not the order in which they are added. This class of problems includes many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.

LOCAL SEARCH

CURRENT STATE

If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. **Local search** algorithms operate using a single **current state** (rather than multiple paths) and generally move only to neighbors of that state. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages: (1) they use very little memory—usually a constant amount; and (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

OPTIMIZATION PROBLEMS

OBJECTIVE FUNCTION

In addition to finding goals, local search algorithms are useful for solving pure **optimization problems,** in which the aim is to find the best state according to an **objective function.** Many optimization problems do not fit the "standard" search model introduced in

Chapter 3. For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no "goal test" and no "path cost" for this problem.

STATE SPACE LANDSCAPE

To understand local search, we will find it very useful to consider the **state space landscape** (as in Figure 4.10). A landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function). If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum;** if elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum,.** (You can convert from one to the other just by inserting a minus sign.) Local search algorithms explore this landscape. A **complete,** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.

GLOBAL MINIMUM

GLOBAL MAXIMUM

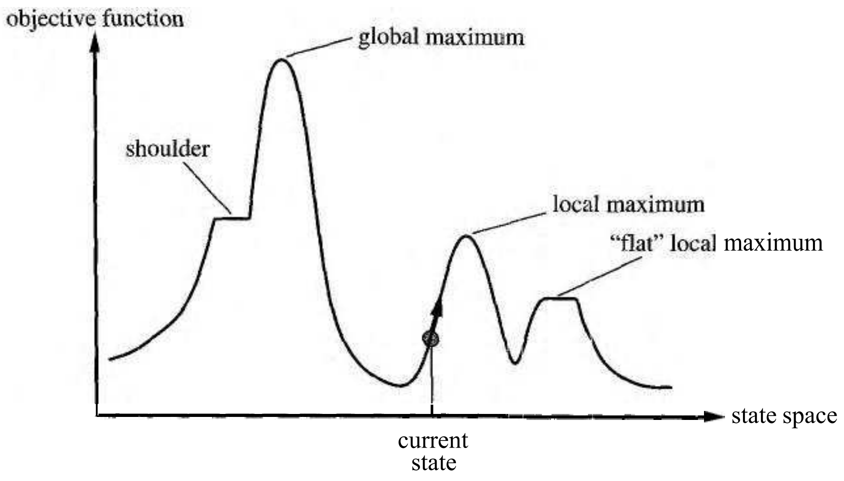

Figure 4.10 A one-dimensional state space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill-climbing search

HILL-CLIMBING

The **hill-climbing** search algorithm is shown in Figure 4.11. It is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the current node data structure need only record the state and its objective function value. Hill-climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia.

To illustrate hill-climbing, we will use the **8-queens problem** introduced on page 66. Local-search algorithms typically use a **complete-state formulation,** where each state has 8 queens on the board, one per column. The successor function returns all possible states generated by moving a single queen to another square in the same column (so each state has

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node

  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
```

Figure 4.11 The hill-climbing search algorithm **(steepest ascent** version), which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

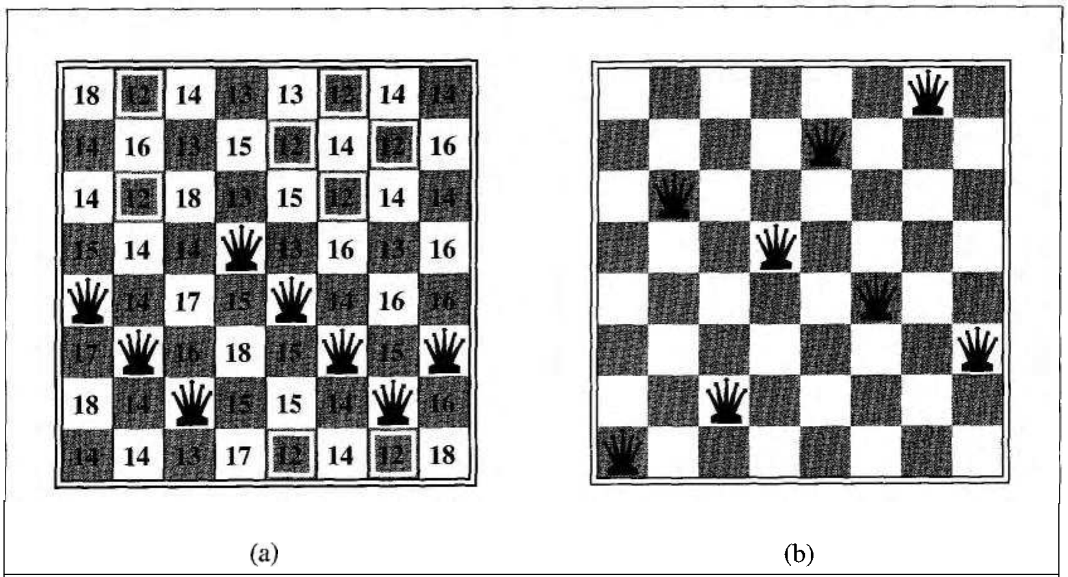

Figure 4.12 (a) An 8-queens state with heuristic cost estimate h = 17, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has h = 1 but every successor has a higher cost.

8 x 7 = 56 successors). The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly. The global minimum of this function is zero, which occurs only at perfect solutions. Figure 4.12(a) shows a state with h = 17. The figure also shows the values of all its successors, with the best successors having h = 12. Hill-climbing algorithms typically choose randomly among the set of best successors, if there is more than one.

GREEDY LOCAL SEARCH

Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing often makes very rapid progress towards a solution, because it is usually quite easy to improve a bad state. For example, from the state in Figure 4.12(a), it takes just five steps to reach the state in Figure 4.12(b), which has $h = 1$ and is very nearly a solution. Unfortunately, hill climbing often gets stuck for the following reasons:

◇ **Local maxima:** a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go. Figure 4.10 illusfrates the problem schematically. More concretely, the state in Figure 4.12(b) is in fact a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.

◇ **Ridges:** a ridge is shown in Figure 4.13. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

◇ **Plateaux:** a plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder,** from which it is possible to make progress. (See Figure 4.10.) A hill-climbing search might be unable to find its way off the plateau.

SHOULDER

In each case, the algorithm reaches a point at which no progress is being made. Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck—not bad for a state space with $8^8 \approx 17$ million states.

The algorithm in Figure 4.11 halts if it reaches a plateau where the best successor has the same value as the current state. Might it not be a good idea to keep going—to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 4.10? The answer is usually yes, but we must take care. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sidleways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill-climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

SIDEWAYS MOVE

STOCHASTIC HILL CLIMBING

Many variants of hill-climbing have been invented. **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes it finds better solutions. **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors. Exercise 4.16 asks you to investigate.

FIRST-CHOICE HILL CLIMBING

The hill-climbing algorithms described so far are incomplete—they often fail to find a goal when one exists because they can get stuck on local maxima. **Random-restart hill**

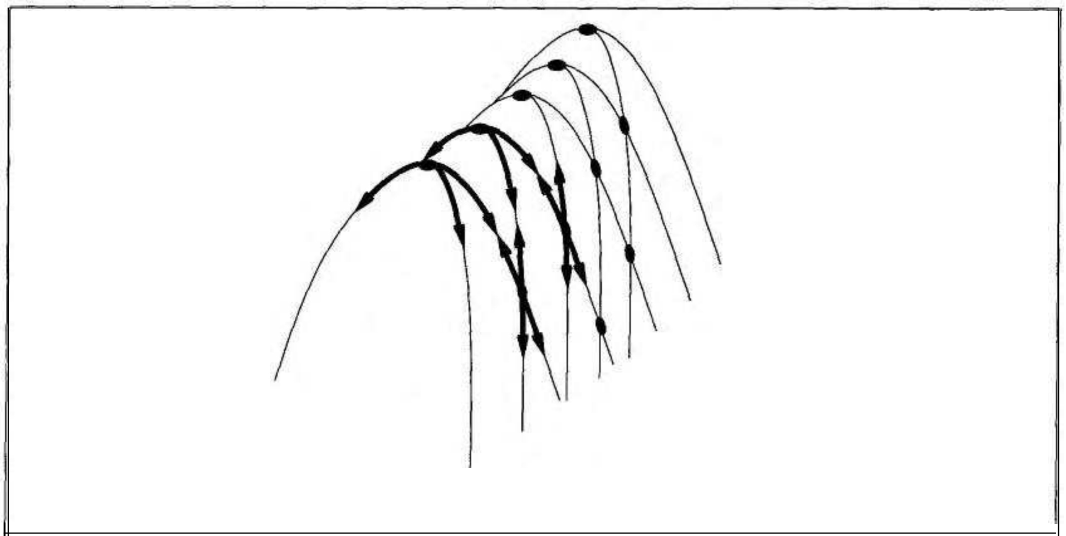

Figure 4.13 Illustration of why ridges cause difficulties for hill-climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

RANDOM-RESTART HILL CLIMBING

climbing adopts the well known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial states,[8] stopping when a goal is found. It is complete with probability approaching 1, for the trivial reason that it will eventually generate a goal state as the initial state. If each hill-climbing search has a probability p of success, then the expected number of restarts required is $1/p$. For 8-queens instances with no sideways moves allowed, $p \approx 0.14$, so we need roughly 7 iterations to find a goal (6 failures and 1 success). The expected number of steps is the cost of one successful iteration plus $(1-p)/p$ times the cost of failure, or roughly 22 steps. When we allow sideways moves, $1/0.94 \approx 1.06$ iterations are needed on average and $(1 \times 21)+(0.06/0.94) \times 64 \approx 25$ steps. For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in under a minute.[9]

The success of hill climbing depends very much on the shape of the state-space landscape: if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly. On the other hand, many real problems have a landscape that looks more like a family of porcupines on a flat floor, with miniature porcupines living on the tip of each porcupine needle, ad *infinitum*. NP-hard problems typically have an exponential number of local maxima to get stuck on. Despite this, a reasonably good local maximum can often be found after a small number of restarts.

[8] Generating a ***random*** state from an implicitly specified state space can be a hard problem in itself.

[9] Luby *et al.* (1993) prove that it is best, in some cases, to restart a randomized search algorithm after a particular, fixed amount of time and that this can be much more efficient than letting each search continue indefinitely. Disallowing or limiting the number of sideways moves is an example of this.

Simulated annealing search

A hill-climbing algorithm that ***never*** makes "downhill" moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete, but extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness. **Simulated annealing** is such an algorithm. In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus alllowing the material to coalesce into a low-energy crystalline state. To understand simulated annealing, let's switch our point of view from hill climbing to **gradient descent** (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local minima, but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

SIMULATED ANNEALING

GRADIENTDESCENT

The innermost loop of the simulated-annealing algorithm (Figure 4.14) is quite similar to hill climbing. Instead of picking the *best* move, however, it picks a ***random*** move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad moves are more likely to be allowed at the start when temperature is high, and they become more unlikely as T decreases. One can prove that if the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks. In Exercise 4.16, you are asked to compare its performance to that of random-restart hill climbing on the n-queens puzzle.

Local beam search

LOCAL BEAM SEARCH

Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The **local beam search** algorithm[10] keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different. In a random-restart search, each search process runs independently of

[10] Local beam search is an adaptation of **beam search,** which is a path-based algorithm.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
     T ← schedule[t]
     if T = 0 then return current
     next ← a randomly selected successor of current
     ΔE ← VALUE[next] – VALUE[current]
     if ΔE > 0 then current ← next
     else current ← next only with probability $e^{\Delta E/T}$
```

Figure 4.14 The simulated annealing search algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of T as a function of time.

the others. In *a local beam search, useful information is passed among the k parallel search threads.* For example, if one state generates several good successors and the other $k - 1$ states all generate bad successors, then the effect is that the first state says to the others, "Come over here, the grass is greener!" The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

In its simplest form, local beam search can suffer from a lack of diversity among the k states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing. A variant called **stochastic beam search,** analogous to stochastic hill climbing, helps to alleviate this problem. Instead of choosing the best k from the the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value. Stochastic beam search bears some resemblance to the process of natural selection, whereby the "successors" (offspring) of a "state" (organism) populate the next generation according to its "value" (fitness).

STOCHASTIC BEAM SEARCH

Genetic algorithms

GENETIC ALGORITHM

A genetic algorithm (or **GA)** is a variant of stochastic beam search in which successor states are generated by combining *two* parent states, rather than by modifying a single state. The analogy to natural selection is the same as in stochastic beam search, except now we are dealing with sexual rather than asexual reproduction.

POPULATION

Like beam search, GAs begin with a set of k randomly generated states, called the **population.** Each state, or **individual,** is represented as a string over a finite alphabet—most

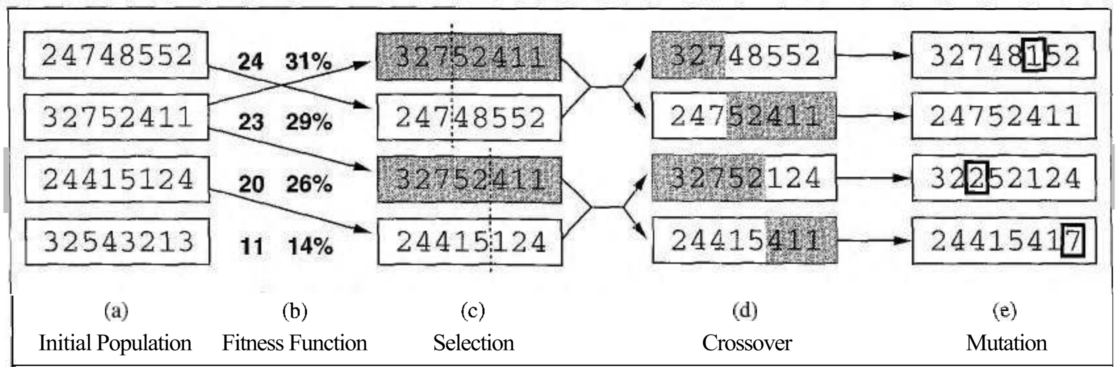

Figure 4.15 The genetic algorithm. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

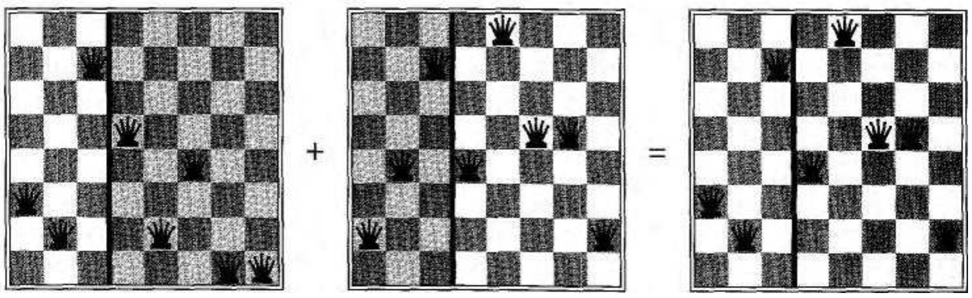

Figure 4.16 The 8-queens states corresponding to the first two parents in Figure 4.15(c) and the first offspring in Figure 4.15(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

commonly, a string of 0s and 1s. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires $8 \times \log_2 8 = 24$ bits. Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8. (We will see later that the two encodings behave differently.) Figure 4.15(a) shows a population of four 8-digit strings representing 8-queens states.

The production of the next generation of states is shown in Figure 4.15(b)–(e). In (b), each state is rated by the evaluation function or (in GA terminology) the **fitness function.** A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of *nonattacking* pairs of queens, which has a value of 28 for a solution. The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.

FITNESS FUNCTION

In (c), a random choice of two pairs is selected for reproduction, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all.[11] For each

[11] There are many variants of this selection rule. The method of **culling,** in which all individuals below a given threshold are discarded, can be shown to converge faster than the random version (Baum *et al.,* 1995).

CROSSOVER

pair to be mated, a **crossover** point is randomly chosen from the positions in the string. In Figure 4.15 the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.[12]

In (d), the offspring themselves are created by crossing over the parent strings at the crossover point. For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent. The 8-queens states involved in this reproduction step are shown in Figure 4.16. The example illustrates the fact that, when two parent states are quite different, the crossover operation can produce a state that is a long way from either parent state. It is often the case that the population is quite diverse early on in the process, so crossover (like simulated annealing) frequently takes large steps in the state space early in the search process and smaller steps later on when most individuals are quite similar.

MUTATION

Finally, in (e), each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column. Figure 4.17 describes an algorithm that implements all these steps.

Like stochastic beam search, genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads. The primary advantage, if any, of genetic algorithms comes from the crossover operation. Yet it can be shown mathematically that, if the positions of the genetic code is permuted initially in a random order, crossover conveys no advantage. Intuitively, the advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates. For example, it could be that putting the first three queens in positions 2, 4, and 6 (where they do not attack each other) constitutes a useful block that can be combined with other blocks to construct a solution.

SCHEMA

The theory of genetic algorithms explains how this works using the idea of a **schema,** which is a substring in which some of the positions can be left unspecified. For example, the schema 246***** describes all 8-queens states in which the first three queens are in positions 2, 4, and 6 respectively. Strings that match the schema (such as 24613578) are called **instances** of the schema. It can be shown that, if the average fitness of the instances of a schema is above the mean, then the number of instances of the schema within the population will grow over time. Clearly, this effect is unlikely to be significant if adjacent bits are totally unrelated to each other, because then there will be few contiguous blocks that provide a consistent benefit. Genetic algorithms work best when schemas correspond to meaningful components of a solution. For example, if the string is a representation of an antenna, then the schemas may represent components of the antenna, such as reflectors and deflectors. A good component is likely to be good in a variety of different designs. This suggests that successful use of genetic algorithms requires careful engineering of the representation.

[12] It is here that the encoding matters. If a 24-bit encoding is used instead of 8 digits, then the crossover point has a 213 chance of being in the middle of a digit, which results in an essentially arbitrary mutation of that digit.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new-population ← empty set
     loop for $i$ from 1 to SIZE(population) do
        $x$ ← RANDOM-SELECTION(population, FITNESS-FN)
        $y$ ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE($x$, $y$)
        if (small random probability) then child ← MUTATE(child)
        add child to new-population
     population ← new-population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE($x$, $y$) returns an individual
  inputs: $x$, $y$, parent individuals

  $n$ ← LENGTH($x$)
  $c$ ← random number from 1 to $n$
  return APPEND(SUBSTRING($x$, 1, $c$), SUBSTRING($y$, $c + 1$, $n$))
```

Figure 4.17 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.15, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

In practice, genetic algorithms have had a widespread impact on optimization problems, such as circuit layout and job-shop scheduling. At present, it is not clear whether the appeal of genetic algorithms arises from their performance or from their æsthetically pleasing origins in the theory of evolution. Much work remains to be done to identify the conditions under which genetic algorithms perform well.

4.4 LOCAL SEARCH IN CONTINUOUS SPACES

In Chapter 2, we explained the distinction between discrete and continuous environments, pointing out that most real-world environments are continuous. Yet none of the algorithms we have described can handle continuous state spaces—the successor function would in most cases return infinitely many states! This section provides a *very brief* introduction to some local search techniques for finding optimal solutions in continuous spaces. The literature on this topic is vast; many of the basic techniques originated in the 17th century, after the development of calculus by Newton and Leibniz.[13] We will find uses for these techniques at

[13] A basic knowledge of multivariate calculus and vector arithmetic is useful when one is reading this section.

EVOLUTION AND SEARCH

The theory of evolution was developed in Charles Darwin's *On the Origin of Species by Means of Natural Selection* (1859). The central idea is simple: variations (known as mutations) occur in reproduction and will be preserved in successive generations approximately in proportion to their effect on reproductive fitness.

Darwin's theory was developed with no knowledge of how the traits of organisms can be inherited and modified. The probabilistic laws governing these processes were first identified by Gregor Mendel (1866), a monk who experimented with sweet peas using what he called artificial fertilization. Much later, Watson and Crick (1953) identified the structure of the DNA molecule and its alphabet, AGTC (adenine, guanine, thymine, cytosine). In the standard model, variation occurs both by point mutations in the letter sequence and by "crossover" (in which the DNA of an offspring is generated by combining long sections of DNA from each parent).

The analogy to local search algorithms has already been described; the principal difference between stochastic beam search and evolution is the use of *sexual* reproduction, wherein successors are generated from *multiple* organisms rather than just one. The actual mechanisms of evolution are, however, far richer than most genetic algorithms allow. For example, mutations can involve reversals, duplications, and movement of large chunks of DNA; some viruses borrow DNA from one organism and insert it in another; and there are transposable genes that do nothing but copy themselves many thousands of times within the genome. There are even genes that poison cells from potential mates that do not carry the gene, thereby increasing their chances of replication. Most important is the fact that the *genes themselves encode the mechanisms* whereby the genome is reproduced and translated into an organism. In genetic algorithms, those mechanisms are a separate program that is not represented within the strings being manipulated.

Darwinian evolution might well seem to be an inefficient mechanism, having generated blindly some 10^{45} or so organisms without improving its search heuristics one iota. Fifty years before Darwin, however, the otherwise great French naturalist Jean Lamarck (1809) proposed a theory of evolution whereby traits *acquired* by *adaptation during an organism's lifetime* would be passed on to its offspring. Such a process would be effective, but does not seem to occur in nature. Much later, James Baldwin (1896) proposed a superficially similar theory: that behavior learned during an organism's lifetime could accelerate the rate of evolution. Unlike Lamarck's, Baldwin's theory is entirely consistent with Darwinian evolution, because it relies on selection pressures operating on individuals that have found local optima among the set of possible behaviors allowed by their genetic makeup. Modern computer simulations confirm that the "Baldwin effect" is real, provided that "ordinary" evolution can create organisms whose internal performance measure is somehow correlated with actual fitness.

several places in the book, including the chapters on learning, vision, and robotics. In short, anything that deals with the real world.

Let us begin with an example. Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map (Figure 3.2) to its nearest airport is minimized. Then the state space is defined by the coordinates of the airports: (x_1, y_1), (x_2, y_2), and (x_3, y_3). This is a *six-dimensional* space; we also say that states are defined by six **variables.** (In general, states are defined by an n-dimensional vector of variables, **x.)** Moving around in this space corresponds to moving one or more of the airports on the map. The objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state once we compute the closest cities, but rather tricky to write down in general.

One way to avoid continuous problems is simply to **discretize** the neighborhood of each state. For example, we can move only one airport at a time in either the x or y direction by a fixed amount ± 6. With 6 variables, this gives 12 possible successors for each state. We can then apply any of the local search algorithms described previously. One can also apply stochastic hill climbing and simulated annealing directly, without discretizing the space. These algorithms choose successors randomly, which can be done by generating random vectors of length 6.

GRADIENT

There are many methods that attempt to use the **gradient** of the landscape to find a maximum. The gradient of the objective function is a vector ∇f that gives the magnitude and direction of the steepest slope. For our problem, we have

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right) .$$

In some cases, we can find a maximum by solving the equation $\nabla f = 0$. (This could be done, for example, if we were placing just one airport; the solution is the arithmetic mean of all the cities' coordinates.) In many cases, however, this equation cannot be solved in closed form. For example, with three airports, the expression for the gradient depends on what cities are closest to each airport in the current state. This means we can compute the gradient *locally* but not *globally.* Even so, we can still perform steepest-ascent hill climbing by updating the current state via the formula

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}) ,$$

where a is a small constant. In other cases, the objective function might not be available in a differentiable form at all—for example, the value of a particular set of airport locations may be determined by running some large-scale economic simulation package. In those cases, a so-,called **empirical gradient** can be determined by evaluating the response to small increments and decrements in each coordinate. Empirical gradient search is the same as steepest-ascent hill climbing in a discretized version of the state space.

EMPIRICAL GRADIENT

Hidden beneath the phrase "a is a small constant" lies a huge variety of methods for adjusting a . The basic problem is that, if a is too small, too many steps are needed; if a is too large, the search could overshoot the maximum. The technique of **line search** tries to overcome this dilemma by extending the current gradient direction—usually by repeatedly doubling a—until f starts to decrease again. The point at which this occurs becomes the new

LINE SEARCH

current state. There are several schools of thought about how the new direction should be chosen at this point.

NEWTON-RAPHSON

For many problems, the most effective algorithm is the venerable **Newton–Raphson** method (Newton, 1671; Raphson, 1690). This is a general technique for finding roots of functions—that is, solving equations of the form $g(x) = 0$. It works by computing a new estimate for the root x according to Newton's formula

$$x \leftarrow x - g(x)/g'(x) .$$

To find a maximum or minimum of f, we need to find $\mathbf{x}$ such that the gradient is zero (i.e., $\nabla f(\mathbf{x}) = \mathbf{0}$). Thus $g(x)$ in Newton's formula becomes $\nabla f(\mathbf{x})$, and the update equation can be written in matrix–vector form as

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x})\nabla f(\mathbf{x}) ,$$

HESSIAN

where $\mathbf{H}_f(\mathbf{x})$ is the **Hessian** matrix of second derivatives, whose elements H_{ij} are given by $\partial^2 f/\partial x_i \partial x_j$. Since the Hessian has n^2 entries, Newton–Raphson becomes expensive in high-dimensional spaces, and many approximations have been developed.

Local search methods suffer from local maxima, ridges, and plateaux in continuous state spaces just as much as in discrete spaces. Random restarts and simulated annealing can be used and are often helpful. High-dimensional continuous spaces are, however, big places in which it is easy to get lost.

CONSTRAINED OPTIMIZATION

A final topic with which a passing acquaintance is useful is **constrained optimization.** An optimization problem is constrained if solutions must satisfy some hard constraints on the values of each variable. For example, in our airport-siting problem, we might constrain sites to be inside Romania and on dry land (rather than in the middle of lakes). The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function. The best-known category is that of **linear programming** problems, in which constraints must be linear inequalities forming a convex region and the objective function is also linear. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on.

LINEAR PROGRAMMING

4.5 Online Search Agents and Unknown Environments

OFFLINE SEARCH

ONLINE SEARCH

So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution before setting foot in the real world (see Figure 3.1), and then execute the solution without recourse to their percepts. In contrast, an **online search**[14] agent operates by **interleaving** computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semidynamic domains—domains where there is a penalty for sitting around and computing too long. Online search is an even better idea for stochastic domains. In general, an offline search would

[14] The term "online" is commonly used in computer science to refer to algorithms that must process input data as they are received, rather than waiting for the entire input data set to become available.

have to come up with an exponentially large contingency plan that considers all possible happenings, while an online search need only consider what actually does happen. For example, a chess playing agent is well-advised to make its first move long before it has figured out the complete course of the game.

EXPLORATION PROBLEM

Online search is a *necessary* idea for an **exploration** problem, where the states and actions are unknown to the agent. An agent in this state of Ignorance must use its actions as experiments to determine what to do next, and hence must interleave computation and action.

The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to B. Methods for escaping from labyrinths—required knowledge for aspiring heroes of antiquity—are also examples of online search algorithms. Spatial exploration is not the only form of exploration, however. Consider a newborn baby: it has many possible actions, but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach. The baby's gradual discovery of how the world works is, in part, an online search process.

Online search problems

An online search problem can be solved only by an agent executing actions, rather than by a purely computational process. We will assume that the agent knows just the following:

- ACTIONS(s), which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$—note that this cannot be used until the agent knows that s' is the outcome; and
- GOAL-TEST(s).

Note in particular that the agent cannot access the successors of a state except by actually trying all the actions in that state. For example, in the maze problem shown in Figure 4.18, the agent does not know that going Up from (1,1) leads to (1,2); nor, having done that, does it know that going *Down* will take it back to (1,1). This degree of ignorance can be reduced in some applications—for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

We wrll assume that the agent can always recognize a state that it has visited before, and we will assume that the actions are deterministic. (These last two assumptions are relaxed in Chapter 17.) Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state. For example, in Figure 4.18, the agent might know the location of the goal and be able to use the Manhattan distance heuristic.

Typically, the agent's objective is to reach a goal state while minimizing cost. (Another possible objective is simply to explore the entire environment.) The cost is the total path cost of the path that the agent actually travels. It is common to compare this cost with the path cost of the path the agent would follow *if* it knew the search space in advance—that is, the actual shortest path (or shortest complete exploration). In the language of online algorithms,

COMPETITIVE RATIO

this is called the competitive ratio; we would like it to be as small as possible.

Although this sounds like a reasonable request, it is easy to see that the best achievable competitive ratio is infinite in some cases. For example,, if some actions are irreversible, the online search might accidentally reach a dead-end state from which no goal state is reachable.

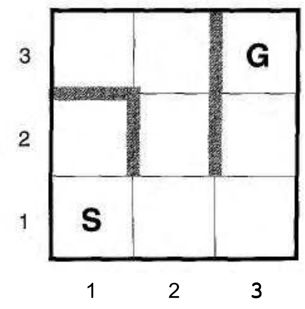

Figure 4.18 A simple maze problem. The agent starts at S and must reach G, but knows nothing of the environment.

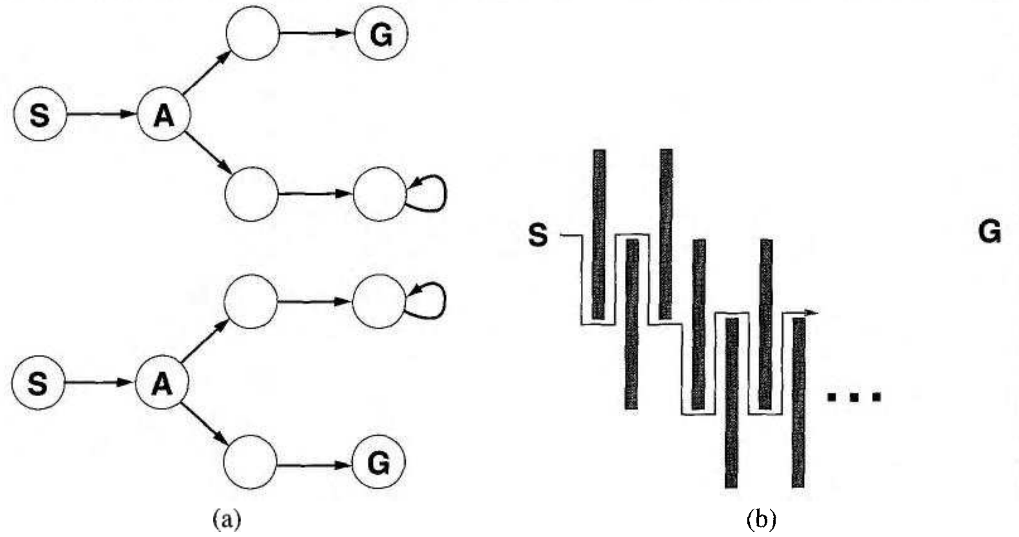

Figure 4.19 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

Perhaps you find the term "accidentally" unconvincing — after all, there might be an algorithm that happens not to take the dead-end path as it explores. Our claim, to be more precise, is that no *algorithm can avoid dead ends in all state spaces.* Consider the two dead-end state spaces in Figure 4.19(a). To an online search algorithm that has visited states S and A, the two state spaces look *identical,* so it must make the same decision in both. Therefore, it will fail in one of them. This is an example of an **adversary argument** — we can imagine an adversary that constructs the state space while the agent explores it and can put the goals and dead ends wherever it likes.

ADVERSARY ARGUMENT

Dead ends are a real difficulty for robot exploration--staircases,ramps, cliffs, and all kinds of natural terrain present opportunities for irreversible actions. To make progress, we will simply assume that the state space is safely explorable — thatis, some goal state is reachable from every reachable state. State spaces with reversible actions, such as mazes and 8-puzzles, can be viewed as undirected graphs and are clearly safely explorable.

SAFELY EXPLORABLE

Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. This is easy to show in environments with irreversible actions, but in fact it remains true for the reversible case as well, as Figure 4.19(b) shows. For this reason, it is common to describe the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

Online search agents

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously. For example, offline algorithms such as A* have the ability to expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real actions. An online algorithm, on the other hand, can expand only a node that it physically occupies. To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a *local* order. Depth-first search has exactly this property, because (except when backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first search agent is shown in Figure 4.20. This agent stores its map in a table, $result[a, s]$, that records the state resulting from executing action a in state s. Whenever an action from the current state has not been explored, the agent tries that action. The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack physically. In depth-first search, this means going back to the state from which the agent entered the current state most recently. That is achieved by keeping a table that lists, for each state, the predecessor states to which the agent has riot yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

We recommend that the reader trace through the progress of ONLINE-DFS-AGENT when applied to the maze given in Figure 4.18. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

```
function ONLINE-DFS-AGENT($s'$) returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          unexplored, a table that lists, for each visited state, the actions not yet tried
          unbacktracked, a table that lists, for each visited state, the backtracks not yet tried
          $s$, $a$, the previous state and action, initially null

  if GOAL-TEST($s'$) then return stop
  if $s'$ is a new state then unexplored[$s'$] ← ACTIONS($s'$)
  if $s$ is not null then do
      result[$a, s$] ← $s'$
      add $s$ to the front of unbacktracked[$s'$]
  if unexplored[$s$] is empty then
      if unbacktracked[$s$] is empty then return stop
      else a ← an action $b$ such that result[$b, s$] = POP(unbacktracked[$s'$])
  else a ← POP(unexplored[$s'$])
  $s$ ← $s'$
  return $a$
```

Figure 4.20 An online search agent that uses depth-first exploration. The agent is applicable only in bidirected search spaces.

Online local search

Like depth-first search, **hill-climbing search** has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is *already* an online search algorithm! Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

RANDOM WALK

Instead of random restarts, one might consider using a **random walk** to explore the environment. A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will *eventually* find a goal or complete its exploration, provided that the space is finite.[15] On the other hand, the process can be very slow. Figure 4.21 shows an environment in which a random walk will take exponentially many steps to find the goal, because, at each step, backward progress is twice as likely as forward progress. The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.

Augmenting hill climbing with *memory* rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" $H(s)$ of the cost to reach the goal from each state that has been visited. $H(s)$ starts out being just the heuristic

[15] The infinite case is much more tricky. Random walks are complete on infinite one-dimensional and two dimensional grids, but not on three-dimensional grids! In the latter case, the probability that the walk ever returns to the starting point is only about 0.3405. (See Hughes, 1995, for a general introduction.)

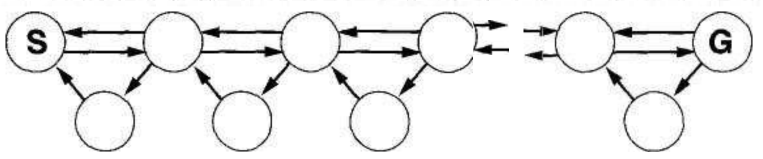

Figure 4.21 An environment in which a random walk will take exponentially many steps to find the goal.

estimate $h(s)$ and is updated as the agent gains experience in the state space. Figure 4.22 shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the shaded state. Rather than staying where it is, the agent should follow what seems to be the best path to the goal based on the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor s is the cost to get to s plus the estimated cost to get to a goal from there—that is, $c(s, a, s) + H(s')$. In the example, there are two actions with estimated costs $1 + 9$ and $1 + 2$, so it seems best to move right. Now, it is clear that the cost estimate of 2 for the shaded state was overly optimistic. Since the best move cost 1 and led to a state that is at least 2 steps from a goal, the shaded state must be at least 3 steps from a goal, so its H should be updated accordingly, as shown in Figure 4.22(b). Continuing this process, the agent will move back and forth twice more, updating H each time and "flatteningout" the local minimum until it escapes to the right.

LRTA*

An agent implementing this scheme, which is called learning real-time A* (LRTA*),is shown in Figure 4.23. Like ONLINE-DFS-AGENT, it builds a map of the environment using the *result* table. It updates the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost, namely $h(s)$. This **optimism under uncertainty** encourages the agent to explore new, possibly promising paths.

OPTIMISM UNDER UNCERTAINTY

An LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment. Unlike A*, however, it is not complete for infinite state spaces—there are cases where it can be led infinitely astray. It can explore an environment of n states in $O(n^2)$ steps in the worst case, but often does much better. The LRTA* agent is just one of a large family of online agents that can be defined by specifying the action selection rule and the update rule in different ways. We will discuss this family, which was developed originally for stochastic environments, in Chapter 21.

Learning in online search

The initial ignorance of online search agents provides several opportunities for learning. First, the agents learn a "map" of the environment—more precisely, the outcome of each action in each state—simply by recording each of their experiences. (Notice that the assumption of deterministic environments means that one experience is enough for each action.) Second, the local search agents acquire more accurate estimates of the value of each state by using local updating rules, as in LRTA*. In Chapter 21 we will see that these updates eventually

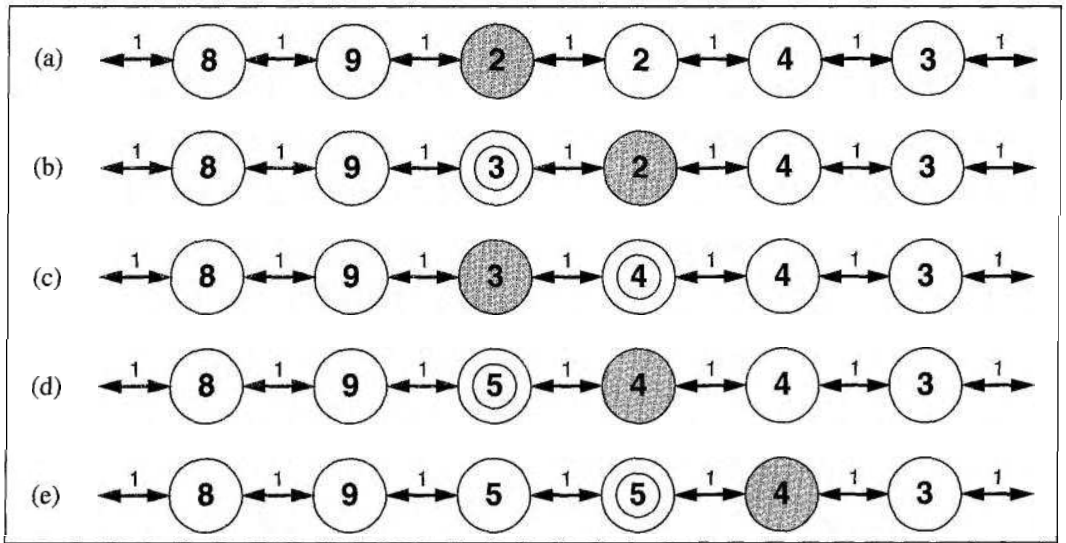

Figure 4.22 Five iterations of *LRTA** on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and each arc is labeled with its step cost. The shaded state marks the location of the agent, and the updated values at each iteration are circled.

```
function LRTA*-AGENT($s'$) returns an action
  inputs: $s'$, a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          $H$, a table of cost estimates indexed by state, initially empty
          $s$, $a$, the previous state and action, initially null

  if GOAL-TEST($s'$) then return stop
  if $s'$ is a new state (not in $H$) then $H[s'] \leftarrow h(s')$
  unless $s$ is null
     $result[a, s] \leftarrow s'$
     $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)}$ LRTA*-COST($s, b, result[b, s], H$)
  a ← an action $b$ in ACTIONS($s'$) that minimizes LRTA*-COST($s', b, result[b, s'], H$)
  $s \leftarrow s'$
  return $a$

function LRTA*-COST($s, a, s', H$) returns a cost estimate
  if $s'$ is undefined then return $h(s)$
  else return $c(s, a, s') + H[s']$
```

Figure 4.23 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

converge to *exact* values for every state, provided that the agent explores the state space in the right way. Once exact values are known, optimal decisions can be taken simply by moving to the highest-valued successor—that is, pure hill climbing is then an optimal strategy.

If you followed our suggestion to trace the behavior of ONLINE-DFS-AGENT in the environment of Figure 4.18, you will have noticed that the agent is not very bright. For example, after it has seen that the Up action goes from (1,1) to (1,2), the agent still has no idea that the *Down* action goes back to (1,1), or that the *Up* action also goes from (2,1) to (2,2), from (2,2) to (2,3), and so on. In general, we would like the agent to learn that Up increases the y-coordinate unless there is a wall in the way, that Down reduces it, and so on. For this to happen, we need two things. First, we need a formal and explicitly manipulable representation for these kinds of general rules; so far, we have hidden the information inside the black box called the successor function. Part III is devoted to this issue. Second, we need algorithms that can construct suitable general rules from the specific observations made by the agent. These are covered in Chapter 18.

4.6 SUMMARY

This chapter has examined the application of **heuristics** to reduce search costs. We have looked at a number of algorithms that use heuristics and found that optimality comes at a stiff price in terms of search cost, even with good heuristics.

- **Best-first search** is just GRAPH-SEARCH where the minimum-cost unexpanded nodes (according to some measure) are selected for expansion. Best-first algorithms typically use a **heuristic** function $h(n)$ that estimates the cost of a solution from n.
- **Greedy best-first search** expands nodes with minimal $h(n)$. It is not optimal, but is often efficient.
- **A* search** expands nodes with minimal $f(n) = g(n) + h(n)$. A* is complete and optimal, provided that we guarantee that $h(n)$ is admissible (for TREE-SEARCH) or consistent (for GRAPH-SEARCH). The space complexity of A* is still prohibitive.
- The performance of heuristic search algorithms depends on the quality of the heuristic function. Good heuristics can sometimes be constructed by relaxing the problem definition, by precomputing solution costs for subproblems in a pattern database, or by learning from experience with the problem class.
- RBFS and SMA* are robust, optimal search algorithms that use limited amounts of memory; given enough time, they can solve problems that A* cannot solve because it runs out of memory.
- Local search methods such as **hill climbing** operate on complete-state formulations, keeping only a small number of nodes in memory. Several stochastic algorithms have been developed, including **simulated annealing,** which returns optimal solutions when given an appropriate cooling schedule. Many local search methods can also be used to solve problems in continuous spaces.

- **A genetic algorithm** is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by **mutation** and by **crossover,** which combines pairs of states from the population.
- **Exploration problems** arise when the agent has no idea about the states and actions of its environment. For safely explorable environments, **online search** agents can build a map and find a goal if one exists. Updating heuristic estimates from experience provides an effective method to escape from local minima.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The use of heuristic information in problem solving appears in an early paper by Simon and Newell (1958), but the phrase "heuristic search" and the use of heuristic functions that estimate the distance to the goal came somewhat later (Newell and Ernst, 1965; Lin, 1965). Doran and Michie (1966) conducted extensive experimental studies of heuristic search as applied to a number of problems, especially the 8-puzzle and the 15-puzzle. Although Doran and Michie carried out theoretical analyses of path length and "penetrance" (the ratio of path length to the total number of nodes examined so far) in heuristic search, they appear to have ignored the information provided by current path length. The A* algorithm, incorporating the current path length into heuristic search, was developed by Hart, Nilsson, and Raphael (1968), with some later corrections (Hart *et al.*, 1972). Dechter and Pearl (1985) demonstrated the optimal efficiency of A*.

The original A* paper introduced the consistency condition on heuristic functions. The monotone condition was introduced by Pohl (1977) as a simpler replacement, but Pearl (1984) showed that the two were equivalent. A number of algorithms predating A* used the equivalent of open and closed lists; these include breadth-first, depth-first, and uniform-cost search (Bellman, 1957; Dijkstra, 1959). Bellman's work in particular showed the importance of additive path costs in simplifying optimization algorithms.

Pohl (1970, 1977) pioneered the study of the relationship between the error in heuristic functions and the time complexity of A*. The proof that A* runs in linear time if the error in the heuristic function is bounded by a constant can be found in Pohl (1977) and in Gaschnig (1979). Pearl (1984) strengthened this result to allow a logarithmic growth in the error. The "effective branching factor" measure of the efficiency of heuristic search was proposed by Nilsson (1971).

There are many variations on the A* algorithm. Pohl (1973) proposed the use of ***dynamic weighting,*** which uses a weighted sum $f_w(n) = w_g g(n) + w_h h(n)$ of the current path length and the heuristic function as an evaluation function, rather than the simple sum $f(n) = g(n) + h(n)$ used in A*. The weights w_g and w_h are adjusted dynamically as the search progresses. Pohl's algorithm can be shown to be **ϵ-admissible**—that is, guaranteed to find solutions within a factor $1 + \epsilon$ of the optimal solution—where ϵ is a parameter supplied to the algorithm. The same property is exhibited by the A_ϵ^* algorithm (Pearl, 1984), which can select any node from the fringe provided its f-cost is within a factor $1 + \epsilon$ of the lowest-f-cost fringe node. The selection can be done so as to minimize search cost.

A* arid other state-space search algorithms are closely related to the *branch-and-bound* techniques that are widely used in operations research (Lawler and Wood, 1966). The relationships between state-space search and branch-and-bound have been investigated in depth (Kumar and Kanal, 1983; Nau *et al.,* 1984; Kumar *et al.*, 1988). Martelli and Montanari (1978) demonstrate a connection between dynamic programming (see Chapter 17) and certain types of state-space search. Kumar and Kanal (1988) attempt a "grand unification" of heuristic search, dynamic programming, and branch-and-bound techniques under the name of CDP—the "composite decision process."

Because computers in the late 1950s and early 1960s had at most a few thousand words of main memory, mernory-bounded heuristic search was an early research topic. The Graph Traverser (Doran and Michie, 1966), one of the earliest search programs, commits to an operator after searching best first up to the memory limit. IDA* (Korf, 1985a, 1985b) was the first widely used optimal, memory-bounded, heuristic search algorithm, and a large number of variants have been developed. An analysis of the efficiency of IDA* and of its difficulties with real-valued heuristics appears in Patrick *et al.* (1992).

ITERATIVE EXPANSION

RBFS (Korf, 1991, 1993) is actually somewhat more complicated than the algorithm shown in Figure 4.5, which is closer to an independently developed algorithm called **iterative expansion,** or IE (Russell, 1992). RBFS uses a lower bound as well as the upper bound; the two algorithms behave identically with admissible heuristics, but RBFS expands nodes in best-first order even with an inadmissible heuristic. The idea of keeping track of the best alternative path appeared earlier in Bratko's (1986) elegant Yrolog implementation of A* and in the DTA* algorithm (Russell and Wefald, 1991). The latter work also discusses metalevel state spaces and metalevel learning.

The MA* algorithm appeared in Chakrabarti *et al.* (1989). SMA*, or Simplified MA*, emerged from an attempt:to implement MA* as a comparison algorithm for IE (Russell, 1992). Kaindl and Khorsand (1994) have applied SMA* to produce a bidirectional search algorithm that is substantially faster than previous algorithms. Korf and Zhang (2000) describe a divide-and-conquer approach, and Zhou and Hansen (2002) introduce memory-bounded A* graph search. Korf (1995) surveys memory-bounded search techniques.

The idea that admissible heuristics can be derived by problem relaxation appears in the seminal paper by Held and Karp (1970), who used the the minimum-spanning-tree heuristic to solve the TSP. (See Exercise 4.8.)

The automation of the relaxation process was implemented successfully by Prieditis (1993), building on earlier work with Mostow (Mostow and Prieditis, 1989). The use of pattern databases to derive admissible heuristics is due to Gasser (1995) and Culberson and Schaeffer (1998); disjoint pattern databases are described by Korf and Felner (2002). The probabilistic interpretation of heuristics was investigated in depth by Pearl (1984) and Hansson and Mayer (1989).

By far the most comprehensive source on heuristics and heuristic search algorithms is Pearl's (1984) *Heuristics* text. This book provides especially good coverage of the wide variety of offshoots and variations of A*, including rigorous proofs of their formal properties. Kanal and Kumar (1988) present an anthology of important articles on heuristic search. New results on search algorithms appear regularly in the journal *Artificial Intelligence.*

Local-search techniques have a long history in mathematics and computer science. Indeed, the Newton–Raphson method (Newton, 1671; Raphson, 1690) can be seen as a very efficient local-search method for continuous spaces in which gradient information is available. Brent (1973) is a classic reference for optimization algorithms that do not require such information. Beam search, which we have presented as a local-search algorithm, originated as a bounded-width variant of dynamic programming for speech recognition in the HARPY system (Lowerre, 1976). A related algorithm is analyzed in depth by Pearl (1984, Ch. 5).

The topic of local search has been reinvigorated in recent years by surprisingly good results for large constraint satisfaction problems such as n-queens (Minton *et* al., 1992) and logical reasoning (Selman *et* al., 1992) and by the incorporation of randomness, multiple simultaneous searches, and other improvements. This renaissance of what Christos Papadimitriou has called "New Age" algorithms has also sparked increased interest among theoretical computer scientists (Koutsoupias and Papadimitriou, 1992; Aldous and Vazirani, 1994). In the field of operations research, a variant of hill climbing called **tabu search** has gained popularity (Glover, 1989; Glover and Laguna, 1997). Drawing on models of limited short-term memory in humans, this algorithm maintains a tabu list of k previously visited states that cannot be revisited; as well as improving efficiency when searching graphs, this can allow the algorithm to escape from some local minima. Another useful improvement on hill climbing is the STAGE algorithm (Boyan and Moore, 1998). The idea is to use the local maxima found by random-restart hill climbing to get an idea of the overall shape of the landscape. The algorithm fits a smooth surface to the set of local maxima and then calculates the global maximum of that surface analytically. This becomes the new restart point. The algorithm has been shown to work in practice on hard problems. (Gomes *et* al., 1998) showed that the run time distributions of systematic backtracking algorithms often have a **heavy-tailed distribution,** which means that the probability of a very long run time is more than would be predicted if the run times were normally distributed. This provides a theoretical justification for random restarts.

TABU SEARCH

HEAVY-TAILED DISTRIBUTION

Simulated annealing was first described by Kirkpatrick *et* al. (1983), who borrowed directly from the **Metropolis algorithm** (which is used to simulate complex systems in physics (Metropolis *et al.,* 1953) and was supposedly invented at a Los Alamos dinner party). Simulated annealing is now a field in itself, with hundreds of papers published every year.

Finding optimal solutions in continuous spaces is the subject matter of several fields, including **optimization theory, optimal control theory,** and the **calculus of variations.** Suitable (and practical) entry points are provided by Press *et* al. (2002) and Bishop (1995). **Linear programming** (LP) was one of the first applications of computers; the **simplex algorithm** (Wood and Dantzig, 1949; Dantzig, 1949) is still used despite worst-case exponential complexity. Karmarkar (1984) developed a practical polynomial-time algorithm for LP.

Work by Sewall Wright (1931) on the concept of a **fitness landscape** was an important precursor to the development of genetic algorithms. In the 1950s, several statisticians, including Box (1957) and Friedman (1959), used evolutionary techniques for optimization problems, but it wasn't until Rechenberg (1965, 1973) introduced **evolution strategies** to solve optimization problems for airfoils that the approach gained popularity. In the 1960s and 1970s, John Holland (1975) championed genetic algorithms, both as a useful tool and

EVOLUTION STRATEGIES

as a method to expand our understanding of adaptation, biological or otherwise (Holland, 1995). The **artificial life** movement (Langton, 1995) takes this idea one step further, viewing the products of genetic algorithms as *organisms* rather than solutions to problems. Work in this field by Hinton and Nowlan (1987) and Ackley and Littman (1991) has done much to clarify the implications of the Baldwin effect. For general background on evolution, we strongly recommend Smith and Szathmáry (1999).

ARTIFICIAL LIFE

Most comparisons of genetic algorithms to other approaches (especially stochastic hill-climbing) have found that the genetic algorithms are slower to converge (O'Reilly and Oppacher, 1994; Mitchell *et al.,* 1996; Juels and Wattenberg, 1996; Baluja, 1997). Such findings are not universally popular within the GA community, but recent attempts within that community to understand population-based search as an approximate form of Bayesian learning (see Chapter 20) might help to close the gap between the field and its critics (Pelikan *et al.,* 1999). The theory of **quadratic dynamical systems** may also explain the performance of GAs (Rabani *et al.*, 1998). See Lohn *et al.* (2001) for an example of GAs applied to antenna design, and Larrañaga *et al.* (1999) for an application to the traveling salesperson problem.

GENETIC PROGRAMMING

The field of **genetic programming** is closely related to genetic algorithms. The principal difference is that the representations that are mutated and combined are programs rather than bit strings. The programs are represented in the form of expression trees; the expressions can be in a standard language such as Lisp or can be specialty designed to represent circuits, robot controllers, and so on. Crossover involves splicing together subtrees rather than substrings. This form of mutation guarantees that the offspring are well-formed expressions, which would not be the case if programs were manipulated as strings.

Recent interest in genetic programming was spurred by John Koza's work (Koza, 1992, 1994), but it goes back at least to early experiments with machine code by Friedberg (1958) and with finite-state automata by Fogel *et al.* (1966). As with genetic algorithms, there is debate about the effectiveness of the technique. Koza *et al.* (1999) describe a variety of experiments on the automated design of circuit devices using genetic programming.

The journals *Evolutionary Computation* and *IEEE Transactions on Evolutionary Computation* cover genetic algorithms and genetic programming; articles are also found in *Complex Systems, Adaptive Behavior,* and *Artificial Life.* 'The main conferences are the *International Conference on Genetic Algorithms* and the *Conference on Genetic Programming,* recently merged to form the *Genetic and Evolutionary Computation Conference.* The texts by Melanie Mitchell (1996) and David Fogel (2000) give good overviews of the field.

EULERIANGRAPHS

Algorithms for exploring unknown state spaces have been of interest for many centuries. Depth-first search in a maze can be implemented by keeping one's left hand on the wall; loops can be avoided by marking each junction. Depth-first search fails with irreversible actions; the more general problem of exploring of **Eulerian graphs** (i.e., graphs in which each node has equal numbers of incoming and outgoing edges) was solved by an algorithm due to Hierholzer (1873). The first thorough algorithmic study of the exploration problem for arbitrary graphs was carried out by Deng and Papadimitriou (1990), who developed a completely general algorithm, but showed that no bounded competitive ratio is possible for exploring a general graph. Papadimitriou and Yannakakis (1991) examined the question of finding paths to a goal in geometric path-planning environments (where all actions are reversible). They showed that a small competitive ratio is achievable with square obstacles, but with general rectangular obstacles no bounded ratio can be achieved. (See Figure 4.19.)

REAL-TIME SEARCH

The LRTA* algorithm was developed by Korf (1990) as part of an investigation into **real-time search** for environments in which the agent must act after searching for only a fixed amount of time (a much more common situation in two-player games). LRTA* is in fact a special case of reinforcement learning algorithms for stochastic environments (Barto *et* al., 1995). Its policy of optimism under uncertainty—always head for the closest unvisited state--can result in an exploration pattern that is less efficient in the uninformed case than simple depth-first search (Koenig, 2000). Dasgupta *et al.* (1994) show that online iterative deepening search is optimally efficient for finding a goal in a uniform tree with no heuristic information. Several informed variants on the LRTA* theme have been developed with different methods for searching and updating within the known portion of the graph (Pemberton and Korf, 1992). As yet, there is no good understanding of how to find goals with optimal efficiency when using heuristic information.

PARALLEL SEARCH

The topic of **parallel search** algorithms was not covered in the chapter, partly because it requires a lengthy discussion of parallel computer architectures. Parallel search is becoming an important topic in both AI and theoretical computer science. A brief introduction to the AI literature can be found in Mahanti and Daniels (1993).

EXERCISES

4.1 Trace the operation of A* search applied to the problem of getting to Bucharest from Lugoj using the straight-line distance heuristic. That is, show the sequence of nodes that the algorithm will consider and the f, g, and h score for each node.

4.2 The **heuristic path algorithm** is a best-first search in which the objective function is $f(n) = (2 - w)g(n) + wh(n)$. For what values of w is this algorithm guaranteed to be optimal? (You may assume that h is admissible.) What kind of search does this perform when w = 0? When w = 1? When w = 2?

4.3 Prove each of the following statements:

a. Breadth-first search is a special case of uniform-cost search.

b. Breadth-first search, depth-first search, and uniform-cost search are special cases of best-first search.

c. Uniform-cost search is a special case of A* search.

4.4 Devise a state space in which A* using GRAPH-SEARCH returns a suboptimal solution with an $h(n)$ function that is admissible but inconsistent.

4.5 We saw on page 96 that the straight-line distance heuristic leads greedy best-first search astray on the problem of going from Iasi to Fagaras. However, the heuristic is perfect on the opposite problem: going from Fagaras to Iasi. Are there problems for which the heuristic is misleading in both directions?

4.6 Invent a heuristic function for the 8-puzzle that sometirnes overestimates, and show how it can lead to a suboptimal solution on a particular problem. (You can use a computer to help if you want.) Prove that, if h never overestimates by more than c, A* using h returns a solution whose cost exceeds that of the optimal solution by no more than c.

4.7 Prove that if a heuristic is consistent, it must be admissible. Construct an admissible heuristic that is not consistent.

4.8 The traveling salesperson problem (TSP) can be solved via the minimum spanning tree (MST) heuristic, which is used to estimate the cost of completing a tour, given that a partial tour has already been constructed. The MST cost of a set of cities is the smallest sum of the link costs of any tree that connects all the cities.

a. Show how this heuristic can be derived from a relaxed version of the TSP.

b. Show that the MST heuristic dominates straight-:linedistance.

c. Write a problem generator for instances of the TSP where cities are represented by random points in the unit square.

d. Find an efficient algorithm in the literature for constructing the MST, and use it with an admissible search algorithm to solve instances of the TSP.

4.9 On page 108, we defined the relaxation of the 8-puzzle in which a tile can move from square A to square B if B is blank. The exact solution of this problem defines **Gaschnig's heuristic** (Gaschnig, 1979). Explain why Gaschnig's heuristic is at least as accurate as h_1 (misplaced tiles), and show cases where it is more accurate than both h_1 and h_2 (Manhattan distance). Can you suggest a way to calculate Gaschnig's heuristic efficiently?

4.10 We gave two simple heuristics for the 8-puzzle:: Manhattan distance and misplaced tiles. Several heuristics in the literature purport to improve on this—see, for example, Nilsson (1971), Mostow and Prieditis (1989), and Hansson *et al.* (1992). Test these claims by implementing the heuristics and comparing the performance of the resulting algorithms.

4.11 Give the name of the algorithm that results from each of the following special cases:

a. Local beam search with $k = 1$.

b. Local beam search with one initial state and no limit on the number of states retained.

c. Simulated annealing with T = 0 at all times (and omitting the termination test).

d. Genetic algorithm with population size N = 1.

4.12 Sometimes there is no good evaluation function for a problem, but there is a good comparison method: a way to tell whether one node is better than another, without assigning numerical values to either. Show that this is enough to do a best-first search. Is there an analog of A*?

4.13 Relate the time complexity of LRTA* to its space complexity.

4.14 Suppose that an agent is in a **3** x **3** maze environment like the one shown in Figure 4.18. The agent knows that its initial location is (1,1), that the goal is at (3,3), and that the four actions Up, *Down*, *Left*, Right have their usual effects unless blocked by a wall. The agent does *not* know where the internal walls are. In any given state, the agent perceives the set of legal actions; it can also tell whether the state is one it has visited before or a new state.

a. Explain how this online search problem can be viewed as an offline search in belief state space, where the initial belief state includes all possible environment configurations. How large is the initial belief state? How large is the space of belief states?

b. How many distinct percepts are possible in the initial state?

c. Describe the first few branches of a contingency plan for this problem. How large (roughly) is the complete plan?

Notice that this contingency plan is a solution for *every possible environment* fitting the given description. Therefore, interleaving of search and execution is not strictly necessary even in unknown environments.

4.15 In this exercise, we will explore the use of local search methods to solve TSPs of the type defined in Exercise 4.8.

a. Devise a hill-climbing approach to solve TSPs. Compare the results with optimal solutions obtained via the A* algorithm with the MST heuristic (Exercise 4.8).

b. Devise a genetic algorithm approach to the traveling salesperson problem. Compare results to the other approaches. You may want to consult Larrañaga *et al.* (1999) for some suggestions for representations.

4.16 Generate a large number of 8-puzzle and 8-queens instances and solve them (where possible) by hill climbing (steepest-ascent and first-choice variants), hill climbing with random restart, and simulated annealing. Measure the search cost and percentage of solved problems and graph these against the optimal solution cost. Comment on your results.

4.17 In this exercise, we will examine hill climbing in the context of robot navigation, using the environment in Figure 3.22 as an example.

a. Repeat Exercise 3.16 using hill climbing. Does your agent ever get stuck in a local minimum? Is it *possible* for it to get stuck with convex obstacles?

b. Construct a nonconvex polygonal environment in which the agent gets stuck.

c. Modify the hill-climbing algorithm so that, instead of doing a depth-1 search to decide where to go next, it does a depth-k search. It should find the best k-step path and do one step along it, and then repeat the process.

d. Is there some k for which the new algorithm is guaranteed to escape from local minima?

e. Explain how LRTA* enables the agent to escape from local minima in this case.

4.18 Compare the performance of A* and RBFS on a set of randomly generated problems in the 8-puzzle (with Manhattan distance) and TSP (with MST—see Exercise 4.8) domains. Discuss your results. What happens to the performance of RBFS when a small random number is added to the heuristic values in the 8-puzzle domain?