

# 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.*

PROBLEM-SOLVING  
AGENT

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, consider future actions and the desirability of their outcomes.

This chapter describes one kind of goal-based agent called a **problem-solving agent**. Problem-solving agents use **atomic** representations, as described in Section 2.4.7—that is, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms. Goal-based agents that use more advanced **factored** or **structured** representations are usually called **planning agents** and are discussed in Chapters 7 and 10.

Our discussion of problem solving begins with precise definitions of **problems** and their **solutions** and give several examples to illustrate these definitions. We then describe several general-purpose search algorithms that can be used to solve these problems. We will see several **uninformed** search algorithms—algorithms that are given no information about the problem other than its definition. Although some of these algorithms can solve any solvable problem, none of them can do so efficiently. **Informed** search algorithms, on the other hand, can do quite well given some guidance on where to look for solutions.

In this chapter, we limit ourselves to the simplest kind of task environment, for which the solution to a problem is always a *fixed sequence* of actions. The more general case—where the agent’s future actions may vary depending on future percepts—is handled in Chapter 4.

This chapter uses the concepts of asymptotic complexity (that is,  $O()$  notation) and NP-completeness. Readers unfamiliar with these concepts 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.

## GOAL FORMULATION

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 and hence the actions it needs to consider. **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.

## PROBLEM 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 how to act, now and in the future, so that it reaches a goal state. Before it can do this, it needs to decide (or we need to decide on its behalf) what sorts of actions and states it should consider. If it were 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 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. Each state therefore corresponds to being in a particular town.

Our agent has now adopted the goal of driving to Bucharest and is considering where to go from Arad. Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind. None of these achieves the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.<sup>1</sup> In other words, the agent will not know which of its possible actions is best, because it does not yet know enough about the state that results from taking each action. If the agent has no additional information—i.e., if the environment is **unknown** in the sense defined in Section 2.3—then it has no choice but to try one of the actions at random. This sad situation is discussed in Chapter 4.

But suppose the agent has a map of Romania. 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 future actions that eventually lead to states of known value.*

To be more specific about what we mean by “examining future actions,” we have to be more specific about properties of the environment, as defined in Section 2.3. For now,

<sup>1</sup> 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.



we assume that the environment is **observable**, so the agent always knows the current state. For the agent driving in Romania, it's reasonable to suppose that each city on the map has a sign indicating its presence to arriving drivers. We also assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from. This is true for navigating in Romania because each city is connected to a small number of other cities. We will assume the environment is **known**, so the agent knows which states are reached by each action. (Having an accurate map suffices to meet this condition for navigation problems.) Finally, we assume that the environment is **deterministic**, so each action has exactly one outcome. Under ideal conditions, this is true for the agent in Romania—it means that if it chooses to drive from Arad to Sibiu, it does end up in Sibiu. Of course, conditions are not always ideal, as we show in Chapter 4.



*Under these assumptions, the solution to any problem is a fixed sequence of actions.* “Of course!” one might say, “What else could it be?” Well, in general it could be a branching strategy that recommends different actions in the future depending on what percepts arrive. For example, under less than ideal conditions, the agent might plan to drive from Arad to Sibiu and then to Rimnicu Vilcea but may also need to have a contingency plan in case it arrives by accident in Zerind instead of Sibiu. Fortunately, if the agent knows the initial state and the environment is known and deterministic, it knows exactly where it will be after the first action and what it will perceive. Since only one percept is possible after the first action, the solution can specify only one possible second action, and so on.

SEARCH

SOLUTION

EXECUTION

The process of looking for a sequence of actions that reaches the goal 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 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.

OPEN-LOOP

Notice that while the agent is executing the solution sequence it *ignores its percepts* when choosing an action because it knows in advance what they will be. 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.

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

### 3.1.1 Well-defined problems and solutions

PROBLEM

A **problem** can be defined formally by five 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  $In(Arad)$ .

```

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

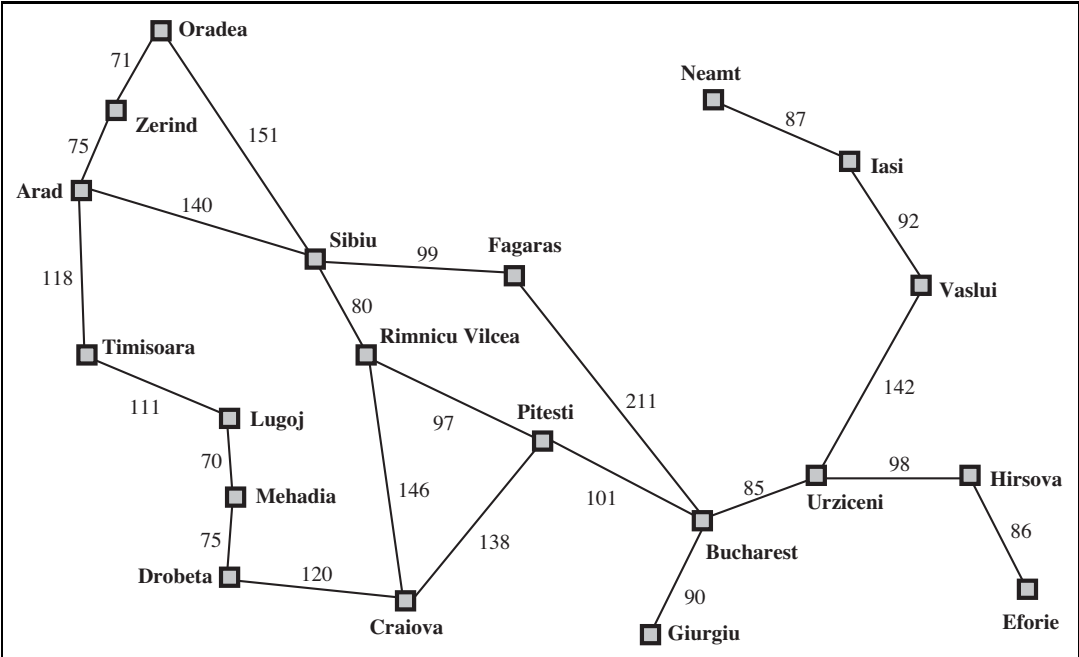
  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
    if seq = failure then return a null action
  action  $\leftarrow$  FIRST(seq)
  seq  $\leftarrow$  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.

ACTIONS	<ul style="list-style-type: none"> <li>• A description of the possible <b>actions</b> available to the agent. Given a particular state <math>s</math>, <math>\text{ACTIONS}(s)</math> returns the set of actions that can be executed in <math>s</math>. We say that each of these actions is <b>applicable</b> in <math>s</math>. For example, from the state <math>\text{In}(\text{Arad})</math>, the applicable actions are <math>\{\text{Go}(\text{Sibiu}), \text{Go}(\text{Timisoara}), \text{Go}(\text{Zerind})\}</math>.</li> </ul>
APPLICABLE	
TRANSITION MODEL	<ul style="list-style-type: none"> <li>• A description of what each action does; the formal name for this is the <b>transition model</b>, specified by a function <math>\text{RESULT}(s, a)</math> that returns the state that results from doing action <math>a</math> in state <math>s</math>. We also use the term <b>successor</b> to refer to any state reachable from a given state by a single action.<sup>2</sup> For example, we have</li> </ul>
SUCCESSOR	$\text{RESULT}(\text{In}(\text{Arad}), \text{Go}(\text{Zerind})) = \text{In}(\text{Zerind}) .$
STATE SPACE	Together, the initial state, actions, and transition model implicitly define the <b>state space</b> of the problem—the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed network or <b>graph</b> in which the nodes are states and the links 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 <b>path</b> in the state space is a sequence of states connected by a sequence of actions.
GRAPH	
PATH	
GOAL TEST	<ul style="list-style-type: none"> <li>• The <b>goal test</b>, 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 <math>\{\text{In}(\text{Bucharest})\}</math>.</li> </ul>

<sup>2</sup> Many treatments of problem solving, including previous editions of this book, use a **successor function**, which returns the set of all successors, instead of separate **ACTIONS** and **RESULT** functions. The successor function makes it difficult to describe an agent that knows what actions it can try but not what they achieve. Also, note some authors use  $\text{RESULT}(a, s)$  instead of  $\text{RESULT}(s, a)$ , and some use **DO** instead of **RESULT**.



**Figure 3.2** A simplified road map of part of Romania.

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.<sup>3</sup> The **step cost** of taking action  $a$  in state  $s$  to reach state  $s'$  is denoted by  $c(s, a, s')$ . The step costs for Romania are shown in Figure 3.2 as route distances. We assume that step costs are nonnegative.<sup>4</sup>

STEP COST

The preceding elements define a problem and can be gathered into a single data structure that is given as input to a problem-solving algorithm. A **solution** to a problem is an action sequence that leads 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

### 3.1.2 Formulating problems

In the preceding section we proposed a formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost. This formulation seems reasonable, but it is still a *model*—an abstract mathematical description—and not the

<sup>3</sup> This assumption is algorithmically convenient but also theoretically justifiable—see page 649 in Chapter 17.  
<sup>4</sup> The implications of negative costs are explored in Exercise 3.8.

real thing. 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, the current radio program, the scenery out of the window, the proximity of law enforcement officers, the distance 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). Our formulation takes into account only the change in location. Also, there are many actions that we 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 one degree."

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.<sup>5</sup> 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

---

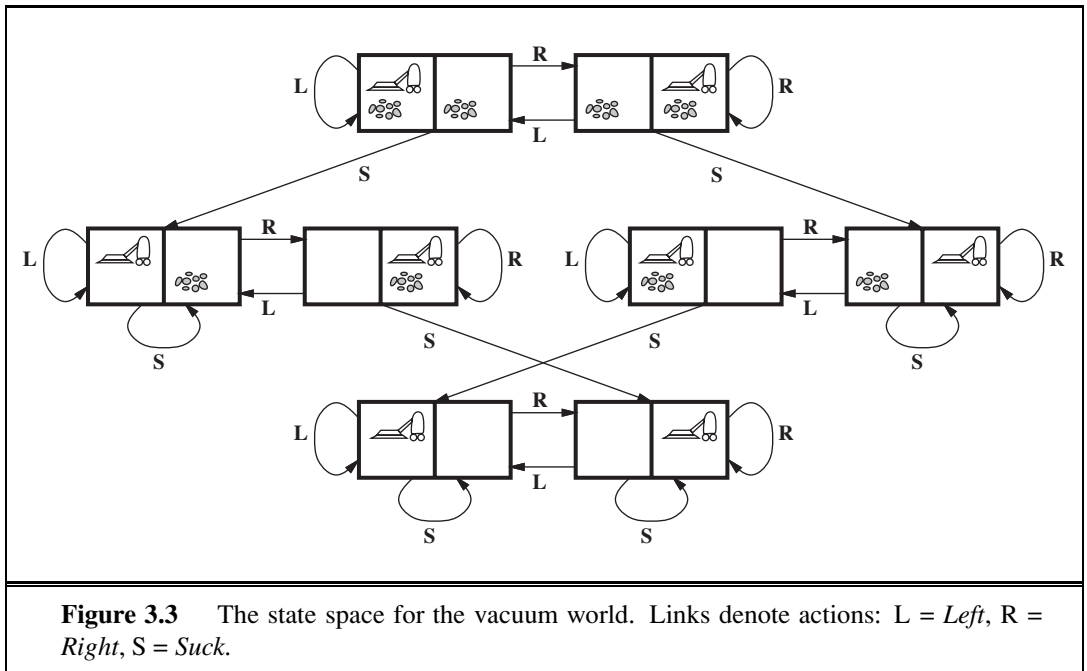
The problem-solving approach has been applied to a vast array of task environments. We list some of the best known here, distinguishing between *toy* and *real-world* problems. A **toy problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description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

TOY PROBLEM

REAL-WORLD  
PROBLEM

---

<sup>5</sup> See Section 11.2 for a more complete set of definitions and algorithms.



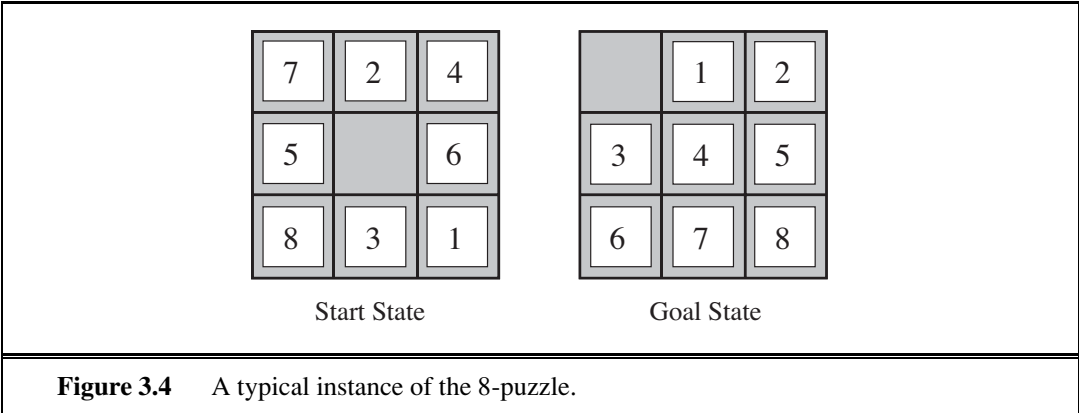
### 3.2.1 Toy problems

The first example we 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 state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are  $2 \times 2^2 = 8$  possible world states. A larger environment with  $n$  locations has  $n \cdot 2^n$  states.
- **Initial state:** Any state can be designated as the initial state.
- **Actions:** In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
- **Transition model:** The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Sucking* in a clean square have no effect. 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 any dirtier. Chapter 4 relaxes some of these assumptions.

The **8-puzzle**, an instance of which is shown in Figure 3.4, consists of a  $3 \times 3$  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).
- **Actions:** The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
- **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 have abstracted away actions such as shaking the board when pieces get stuck and ruled out extracting the pieces with a knife and putting them back again. We are left with a description of the rules of the puzzle, avoiding all the details of physical manipulations.

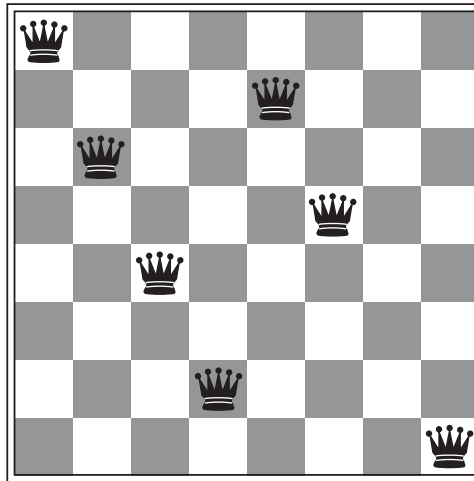
The 8-puzzle belongs to the family of **sliding-block puzzles**, which are often used as test problems for new search algorithms in AI. This family is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next. The 8-puzzle has  $9!/2 = 181,440$  reachable states and is easily solved. The 15-puzzle (on a  $4 \times 4$  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  $5 \times 5$  board) has around  $10^{25}$  states, and random instances take several hours to solve optimally.

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.

SLIDING-BLOCK  
PUZZLES

8-QUEENS PROBLEM





**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 for the whole  $n$ -queens family, it remains a useful 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:

- **States:** Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state:** No queens on the board.
- **Actions:** Add a queen to any empty square.
- **Transition model:** Returns the board with a queen added to the specified 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:** All possible arrangements of  $n$  queens ( $0 \leq n \leq 8$ ), one per column in the leftmost  $n$  columns, with no queen attacking another.
- **Actions:** 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  $1.8 \times 10^{14}$  to just 2,057, and solutions are easy to find. On the other hand, for 100 queens the reduction is from roughly  $10^{400}$  states to about  $10^{52}$  states (Exercise 3.5)—a big improvement, but not enough to make the problem tractable. Section 4.1 describes the complete-state formulation, and Chapter 6 gives a simple algorithm that solves even the million-queens problem with ease.

INCREMENTAL  
FORMULATION

COMPLETE-STATE  
FORMULATION

Our final toy problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise. Knuth conjectured that, starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer. For example, we can reach 5 from 4 as follows:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5.$$

The problem definition is very simple:

- **States:** Positive numbers.
- **Initial state:** 4.
- **Actions:** Apply factorial, square root, or floor operation (factorial for integers only).
- **Transition model:** As given by the mathematical definitions of the operations.
- **Goal test:** State is the desired positive integer.

To our knowledge there is no bound on how large a number might be constructed in the process of reaching a given target—for example, the number 620,448,401,733,239,439,360,000 is generated in the expression for 5—so the state space for this problem is infinite. Such state spaces arise frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects.

### 3.2.2 Real-world problems

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. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example. Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications. Consider the airline travel problems that must be solved by a travel-planning Web site:

- **States:** Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these “historical” aspects.
- **Initial state:** This is specified by the user’s query.
- **Actions:** Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- **Transition model:** The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.
- **Goal test:** Are we at the final destination specified by the user?
- **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 traveler 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 PROBLEM

**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. Later in this chapter, we present some algorithms capable of solving them.

## ROBOT NAVIGATION

**Robot navigation** is a generalization of the route-finding problem described earlier. Rather than following 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

PROTEIN DESIGN

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 actions 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.

### 3.3    SEARCHING FOR SOLUTIONS

SEARCH TREE

NODE

EXPANDING

GENERATING

PARENT NODE

CHILD NODE

Having formulated some problems, we now need to solve them. A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the branches are actions and the **nodes** correspond to states in the state space of the problem. Figure 3.6 shows the first few steps in growing the search tree for finding a route from Arad to Bucharest. The root node of the tree corresponds 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.”) Then we need to consider taking various actions. We do this by **expanding** the current state; that is, applying each legal action to the current state, thereby **generating** a new set of states. In this case, we add three branches from the **parent node** *In(Arad)* leading to three new **child nodes**: *In(Sibiu)*, *In(Timisoara)*, and *In(Zerind)*. Now we must choose which of these three possibilities to consider further.

LEAF NODE

FRONTIER

OPEN LIST

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. Each of these six nodes is a **leaf node**, that is, a node with no children in the tree. The set of all leaf nodes available for expansion at any given point is called the **frontier**. (Many authors call it the **open list**, which is both geographically less evocative and less accurate, because other data structures are better suited than a list.) In Figure 3.6, the frontier of each tree consists of those nodes with bold outlines.

SEARCH STRATEGY

The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. The general TREE-SEARCH algorithm is shown informally in Figure 3.7. Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy**.

REPEATED STATE

LOOPY PATH

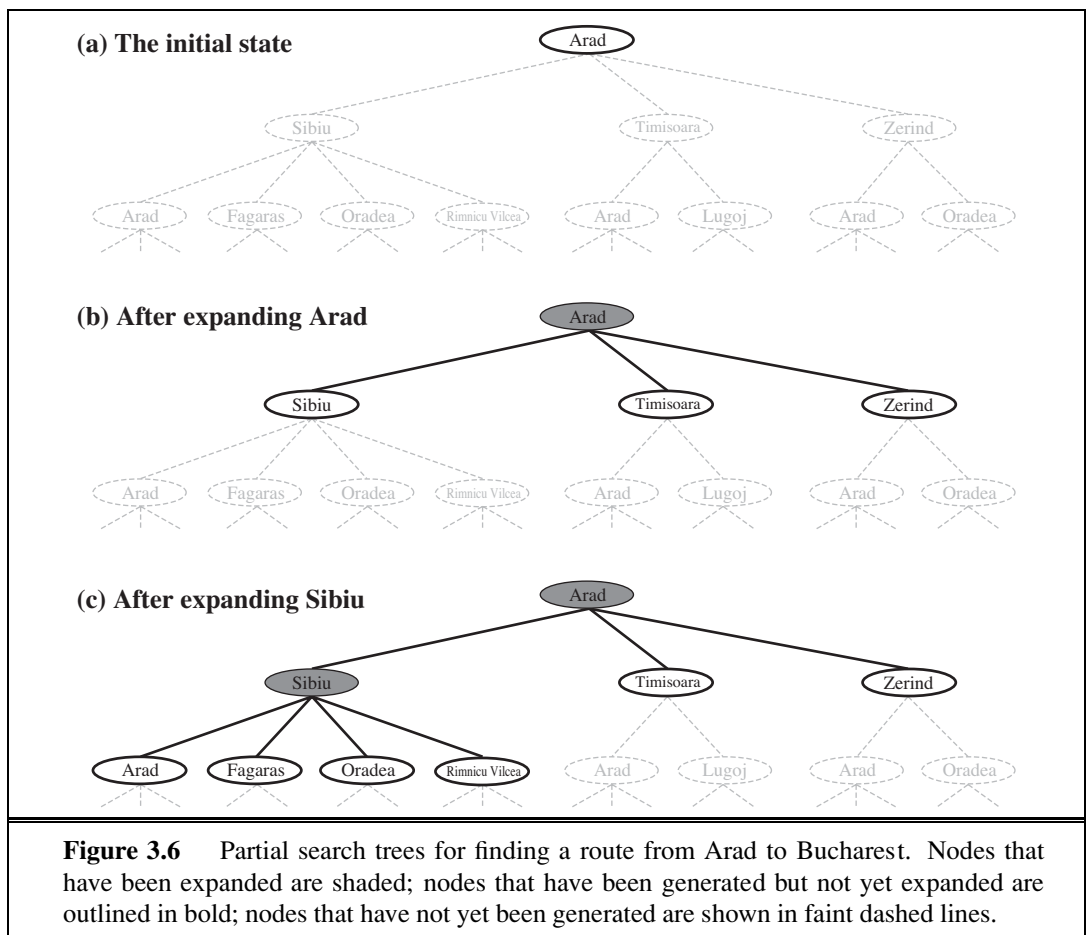
The eagle-eyed reader will notice one peculiar thing about the search tree shown in Figure 3.6: it includes the path from Arad to Sibiu and back to Arad again! We say that *In(Arad)* is a **repeated state** in the search tree, generated in this case by a **loopy path**. Considering such loopy paths means that the complete search tree for Romania is *infinite* because there is no limit to how often one can traverse a loop. On the other hand, the state space—the map shown in Figure 3.2—has only 20 states. As we discuss in Section 3.4, loops can cause

certain algorithms to fail, making otherwise solvable problems unsolvable. Fortunately, there is no need to consider loopy paths. We can rely on more than intuition for this: because path costs are additive and step costs are nonnegative, a loopy path to any given state is never better than the same path with the loop removed.

#### REDUNDANT PATH

Loopy paths are a special case of the more general concept of **redundant paths**, which exist whenever there is more than one way to get from one state to another. Consider the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long). Obviously, the second path is redundant—it's just a worse way to get to the same state. If you are concerned about reaching the goal, there's never any reason to keep more than one path to any given state, because any goal state that is reachable by extending one path is also reachable by extending the other.

In some cases, it is possible to define the problem itself so as to eliminate redundant paths. For example, if we formulate the 8-queens problem (page 71) so that a queen can be placed in any column, then each state with  $n$  queens can be reached by  $n!$  different paths; but if we reformulate the problem so that each new queen is placed in the leftmost empty column, then each state can be reached only through one path.



```

function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set

```

**Figure 3.7** An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

In other cases, redundant paths are unavoidable. This includes all problems where the actions are reversible, such as route-finding problems and sliding-block puzzles. Route-finding on a **rectangular grid** (like the one used later for Figure 3.9) is a particularly important example in computer games. In such a grid, each state has four successors, so a search tree of depth  $d$  that includes repeated states has  $4^d$  leaves; but there are only about  $2d^2$  distinct states within  $d$  steps of any given state. For  $d = 20$ , this means about a trillion nodes but only about 800 distinct states. Thus, following redundant paths can cause a tractable problem to become intractable. This is true even for algorithms that know how to avoid infinite loops.

As the saying goes, *algorithms that forget their history are doomed to repeat it*. The way to avoid exploring redundant paths is to remember where one has been. To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set** (also known as the **closed list**), which remembers every expanded node. Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier. The new algorithm, called GRAPH-SEARCH, is shown informally in Figure 3.7. The specific algorithms in this chapter draw on this general design.

Clearly, the search tree constructed by the GRAPH-SEARCH algorithm contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph, as shown in Figure 3.8. The algorithm has another nice property: the frontier **separates** the state-space graph into the explored region and the unexplored region, so that every path from

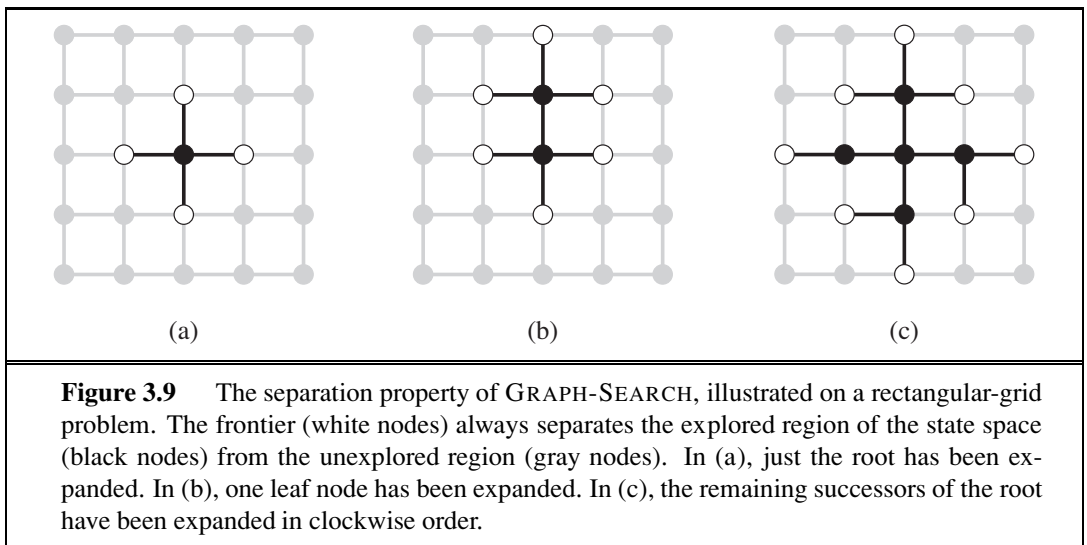
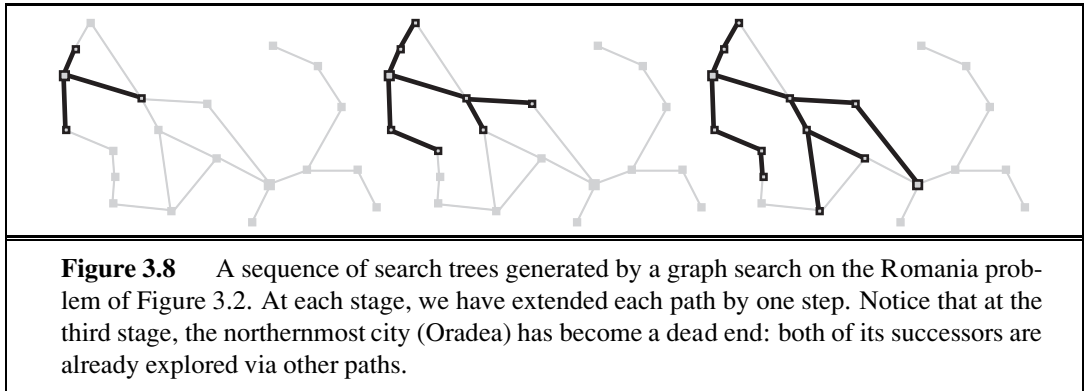
RECTANGULAR GRID



EXPLORED SET

CLOSED LIST

SEPARATOR

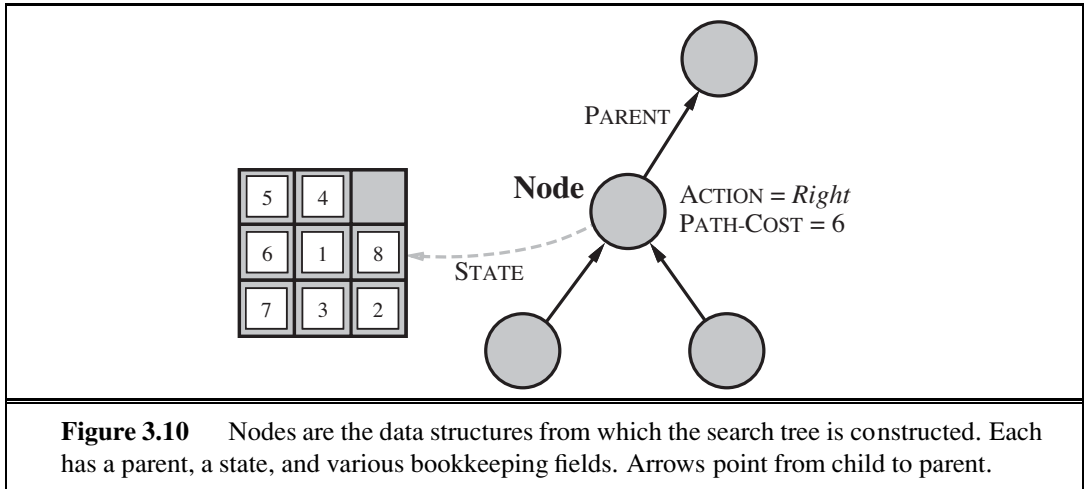


the initial state to an unexplored state has to pass through a state in the frontier. (If this seems completely obvious, try Exercise 3.13 now.) This property is illustrated in Figure 3.9. As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is *systematically* examining the states in the state space, one by one, until it finds a solution.

### 3.3.1 Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node  $n$  of the tree, we have a structure that contains four components:

- $n$ .STATE: the state in the state space to which the node corresponds;
- $n$ .PARENT: the node in the search tree that generated this node;
- $n$ .ACTION: the action that was applied to the parent to generate the node;
- $n$ .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.



Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function `CHILD-NODE` takes a parent node and an action and returns the resulting child node:

```
function CHILD-NODE(problem, parent, action) returns a node
return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

The node data structure is depicted in Figure 3.10. Notice how the PARENT pointers string the nodes together into a tree structure. These pointers also allow the solution path to be extracted when a goal node is found; we use the `SOLUTION` function to return the sequence of actions obtained by following parent pointers back to the root.

Up to now, we have not been very careful to distinguish between nodes and states, but in writing detailed algorithms it's important to make that distinction. A node is a bookkeeping 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 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.

Now that we have nodes, we need somewhere to put them. The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a **queue**. The operations on a queue are as follows:

- `EMPTY?(queue)` returns true only if there are no more elements in the queue.
- `POP(queue)` removes the first element of the queue and returns it.
- `INSERT(element, queue)` inserts an element and returns the resulting queue.



FIFO QUEUE  
LIFO QUEUE  
PRIORITY QUEUE

Queues are characterized by the *order* in which they store the inserted nodes. Three common variants are the first-in, first-out or **FIFO queue**, which pops the *oldest* element of the queue; the last-in, first-out or **LIFO queue** (also known as a **stack**), which pops the *newest* element of the queue; and the **priority queue**, which pops the element of the queue with the highest priority according to some ordering function.

CANONICAL FORM

The explored set can be implemented with a hash table to allow efficient checking for repeated states. With a good implementation, insertion and lookup can be done in roughly constant time no matter how many states are stored. One must take care to implement the hash table with the right notion of equality between states. For example, in the traveling salesperson problem (page 74), the hash table needs to know that the set of visited cities {Bucharest,Urziceni,Vaslui} is the same as {Urziceni,Vaslui,Bucharest}. Sometimes this can be achieved most easily by insisting that the data structures for states be in some **canonical form**; that is, logically equivalent states should map to the same data structure. In the case of states described by sets, for example, a bit-vector representation or a sorted list without repetition would be canonical, whereas an unsorted list would not.

3.3.2 Measuring problem-solving performance

Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them. We can evaluate an algorithm’s performance in four ways:

COMPLETENESS  
OPTIMALITY  
TIME COMPLEXITY  
SPACE COMPLEXITY

- **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
- **Optimality:** Does the strategy find the optimal solution, as defined on page 68?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

BRANCHING FACTOR  
DEPTH

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,  $|V| + |E|$ , where  $V$  is the set of vertices (nodes) of the graph and  $E$  is the set of edges (links). This is appropriate when the graph is an explicit data structure that is input to the search program. (The map of Romania is an example of this.) In AI, the graph is often represented *implicitly* by the initial state, actions, and transition model and is frequently infinite. For these reasons, 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 (i.e., the number of steps along the path from the root); and  $m$ , the maximum length of any path in the state space. Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory. For the most part, we describe time and space complexity for search on a tree; for a graph, the answer depends on how “redundant” the paths in the state space are.

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 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 milliseconds and kilometers. 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 is taken up in Chapter 16.

## 3.4 UNINFORMED SEARCH STRATEGIES

UNINFORMED  
SEARCH  
BLIND SEARCH

This section covers several search strategies that come under the heading of **uninformed search** (also called **blind search**). The term means that the strategies 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 non-goal state. All search strategies are distinguished by the *order* in which nodes are expanded. Strategies that know whether one non-goal state is “more promising” than another are called **informed search** or **heuristic search** strategies; they are covered in Section 3.5.

INFORMED SEARCH  
HEURISTIC SEARCH

### 3.4.1 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 is an instance of the general graph-search algorithm (Figure 3.7) in which the *shallowest* unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is *generated* rather than when it is selected for expansion. This decision is explained below, where we discuss time complexity. Note also that the algorithm, following the general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found. Thus, breadth-first search always has the shallowest path to every node on the frontier.

Pseudocode is given in Figure 3.11. Figure 3.12 shows the progress of the search on a simple binary tree.

How does breadth-first search rate according to 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 generating all shallower nodes (provided the branching factor  $b$  is finite). Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test. Now, the *shallowest* goal node is not necessarily the *optimal* one;

```

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)

```

**Figure 3.11** Breadth-first search on a graph.

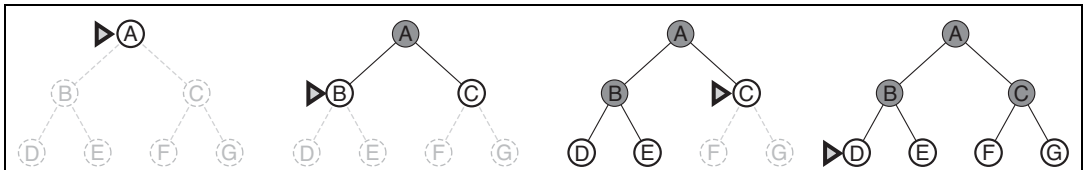
technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. The most common such scenario is that all actions have the same cost.

So far, the news about breadth-first search has been good. The news about time and space is not so good. Imagine searching a uniform tree 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, it is the last node generated at that level. Then the total number of nodes generated is

$$b + b^2 + b^3 + \dots + b^d = O(b^d).$$

(If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth  $d$  would be expanded before the goal was detected and the time complexity would be  $O(b^{d+1})$ .)

As for space complexity: for any kind of graph search, which stores every expanded node in the *explored* set, the space complexity is always within a factor of  $b$  of the time complexity. For breadth-first graph search in particular, every node generated remains in memory. There will be  $O(b^{d-1})$  nodes in the *explored* set and  $O(b^d)$  nodes in the frontier,



**Figure 3.12** Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

so the space complexity is  $O(b^d)$ , i.e., it is dominated by the size of the frontier. Switching to a tree search would not save much space, and in a state space with many redundant paths, switching could cost a great deal of time.

An exponential complexity bound such as  $O(b^d)$  is scary. Figure 3.13 shows why. It lists, for various values of the solution depth  $d$ , the time and memory required for a breadth-first search with branching factor  $b = 10$ . The table assumes that 1 million 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.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	$10^6$	1.1 seconds	1 gigabyte
8	$10^8$	2 minutes	103 gigabytes
10	$10^{10}$	3 hours	10 terabytes
12	$10^{12}$	13 days	1 petabyte
14	$10^{14}$	3.5 years	99 petabytes
16	$10^{16}$	350 years	10 exabytes

**Figure 3.13** Time and memory requirements for breadth-first search. The numbers shown assume branching factor  $b = 10$ ; 1 million nodes/second; 1000 bytes/node.



Two lessons can be learned from Figure 3.13. First, *the memory requirements are a bigger problem for breadth-first search than is the execution time*. One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take. Fortunately, other strategies require less memory.



The second lesson is that time is still a major factor. If your problem has a solution at depth 16, then (given our assumptions) it will take about 350 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*.

### 3.4.2 Uniform-cost search

When all step costs are equal, breadth-first search is optimal 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*  $g(n)$ . This is done by storing the frontier as a priority queue ordered by  $g$ . The algorithm is shown in Figure 3.14.

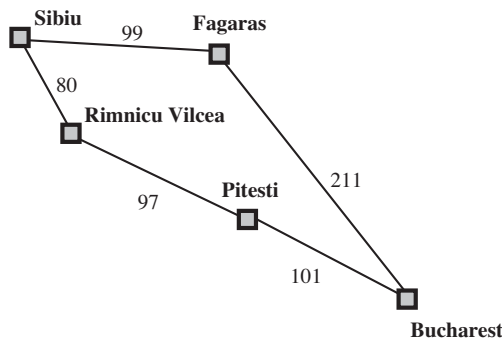
In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search. The first is that the goal test is applied to a node when it is *selected for expansion* (as in the generic graph-search algorithm shown in Figure 3.7) rather than when it is first generated. The reason is that the first goal node that is *generated*

```

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier  $\leftarrow$  INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child

```

**Figure 3.14** Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.



**Figure 3.15** Part of the Romania state space, selected to illustrate uniform-cost search.

may be on a suboptimal path. The second difference is that a test is added in case a better path is found to a node currently on the frontier.

Both of these modifications come into play in the example shown in Figure 3.15, where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost  $80 + 97 = 177$ . The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost  $99 + 211 = 310$ . Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path

to Bucharest with cost  $80 + 97 + 101 = 278$ . Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with  $g$ -cost 278, is selected for expansion and the solution is returned.

It is easy to see that uniform-cost search is optimal in general. First, we observe that whenever uniform-cost search selects a node  $n$  for expansion, the optimal path to that node has been found. (Were this not the case, there would have to be another frontier node  $n'$  on the optimal path from the start node to  $n$ , by the graph separation property of Figure 3.9; by definition,  $n'$  would have lower  $g$ -cost than  $n$  and would have been selected first.) Then, because step costs are nonnegative, paths never get shorter as nodes are added. These two facts together imply that *uniform-cost search expands nodes in order of their optimal path cost*. Hence, the first goal node selected for expansion must be the optimal solution.

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 there is a path with an infinite sequence of zero-cost actions—for example, a sequence of *NoOp* actions.<sup>6</sup> Completeness is guaranteed provided the cost of every step exceeds some small positive constant  $\epsilon$ .

Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of  $b$  and  $d$ . Instead, let  $C^*$  be the cost of the optimal solution,<sup>7</sup> and assume that every action costs at least  $\epsilon$ . Then the algorithm's worst-case time and space complexity is  $O(b^{1+\lceil C^*/\epsilon \rceil})$ , which can be much greater than  $b^d$ . This is because uniform-cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps. When all step costs are equal,  $b^{1+\lceil C^*/\epsilon \rceil}$  is just  $b^{d+1}$ . When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost; thus uniform-cost search does strictly more work by expanding nodes at depth  $d$  unnecessarily.

### 3.4.3 Depth-first search

**Depth-first search** always expands the *deepest* node in the current frontier of the search tree. The progress of the search is illustrated in Figure 3.16. 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 frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.

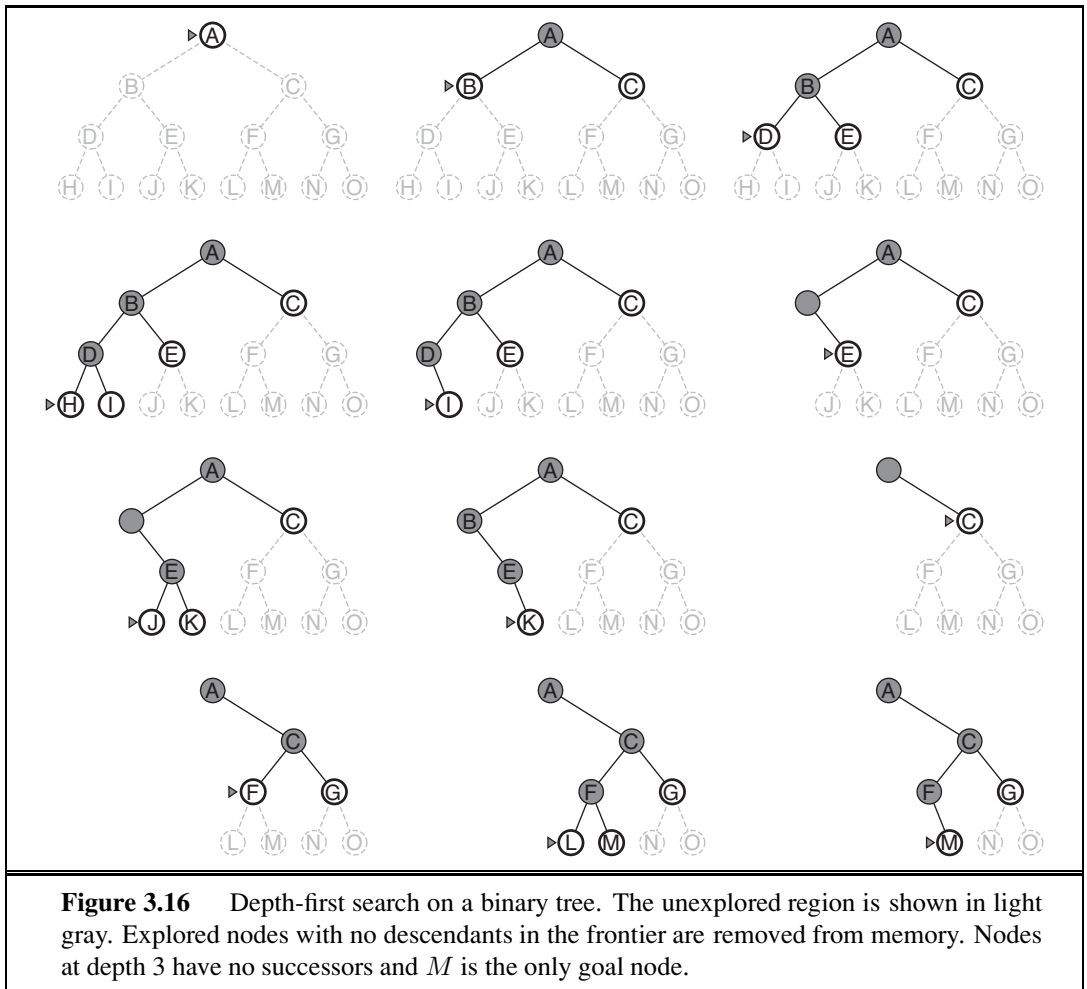
The depth-first search algorithm is an instance of the graph-search algorithm in Figure 3.7; whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

As an alternative to the GRAPH-SEARCH-style 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 3.17.)

<sup>6</sup> *NoOp*, or “no operation,” is the name of an assembly language instruction that does nothing.

<sup>7</sup> Here, and throughout the book, the “star” in  $C^*$  means an optimal value for  $C$ .





The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used. The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node. The tree-search version, on the other hand, is *not* complete—for example, in Figure 3.6 the algorithm will follow the Arad–Sibiu–Arad–Sibiu loop forever. Depth-first tree search can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node; this avoids infinite loops in finite state spaces but does not avoid the proliferation of redundant paths. In infinite state spaces, both versions fail if an infinite non-goal path is encountered. For example, in Knuth’s 4 problem, depth-first search would keep applying the factorial operator forever.

For similar reasons, both versions are nonoptimal. For example, in Figure 3.16, 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 instead of *C*, which would be a better solution; hence, depth-first search is not optimal.

The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite, of course). A depth-first tree search, on the other hand, may generate all of the  $O(b^m)$  nodes in the search tree, where  $m$  is the maximum depth of any node; this can be much greater than the size of the state space. Note that  $m$  itself can be much larger than  $d$  (the depth of the shallowest solution) and is infinite if the tree is unbounded.

So far, depth-first search seems to have no clear advantage over breadth-first search, so why do we include it? The reason is the space complexity. For a graph search, there is no advantage, but a depth-first tree search 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 3.16.) For a state space with branching factor  $b$  and maximum depth  $m$ , depth-first search requires storage of only  $O(bm)$  nodes. Using the same assumptions as for Figure 3.13 and assuming that nodes at the same depth as the goal node have no successors, we find that depth-first search would require 156 kilobytes instead of 10 exabytes at depth  $d = 16$ , a factor of 7 trillion times less space. This has led to the adoption of depth-first tree search as the basic workhorse of many areas of AI, including constraint satisfaction (Chapter 6), propositional satisfiability (Chapter 7), and logic programming (Chapter 9). For the remainder of this section, we focus primarily on the tree-search version of depth-first search.

BACKTRACKING  
SEARCH

A variant of depth-first search called **backtracking search** uses still less memory. (See Chapter 6 for more details.) 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.

### 3.4.4 Depth-limited search

DEPTH-LIMITED  
SEARCH

The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit  $\ell$ . That is, nodes at depth  $\ell$  are treated as if they have no successors. This approach is called **depth-limited search**. The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose  $\ell < d$ , that is, the shallowest goal is beyond the depth limit. (This is likely 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



```

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

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff_occurred?  $\leftarrow$  false
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)
      if result = cutoff then cutoff_occurred?  $\leftarrow$  true
      else if result  $\neq$  failure then return result
    if cutoff_occurred? then return cutoff else return failure

```

**Figure 3.17** A recursive implementation of depth-limited tree search.

map carefully, we would discover that any city can be reached from any other city in at most 9 steps. This number, known as the **diameter** of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search. For most problems, however, we will not know a good depth limit until we have solved the problem.

Depth-limited search can be implemented as a simple modification to the general tree- or graph-search algorithm. Alternatively, it can be implemented as a simple recursive algorithm as shown in Figure 3.17. 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.

### 3.4.5 Iterative deepening depth-first search

**Iterative deepening search** (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree 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 3.18. Iterative deepening combines the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are 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 3.19 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 too 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

DIAMETER

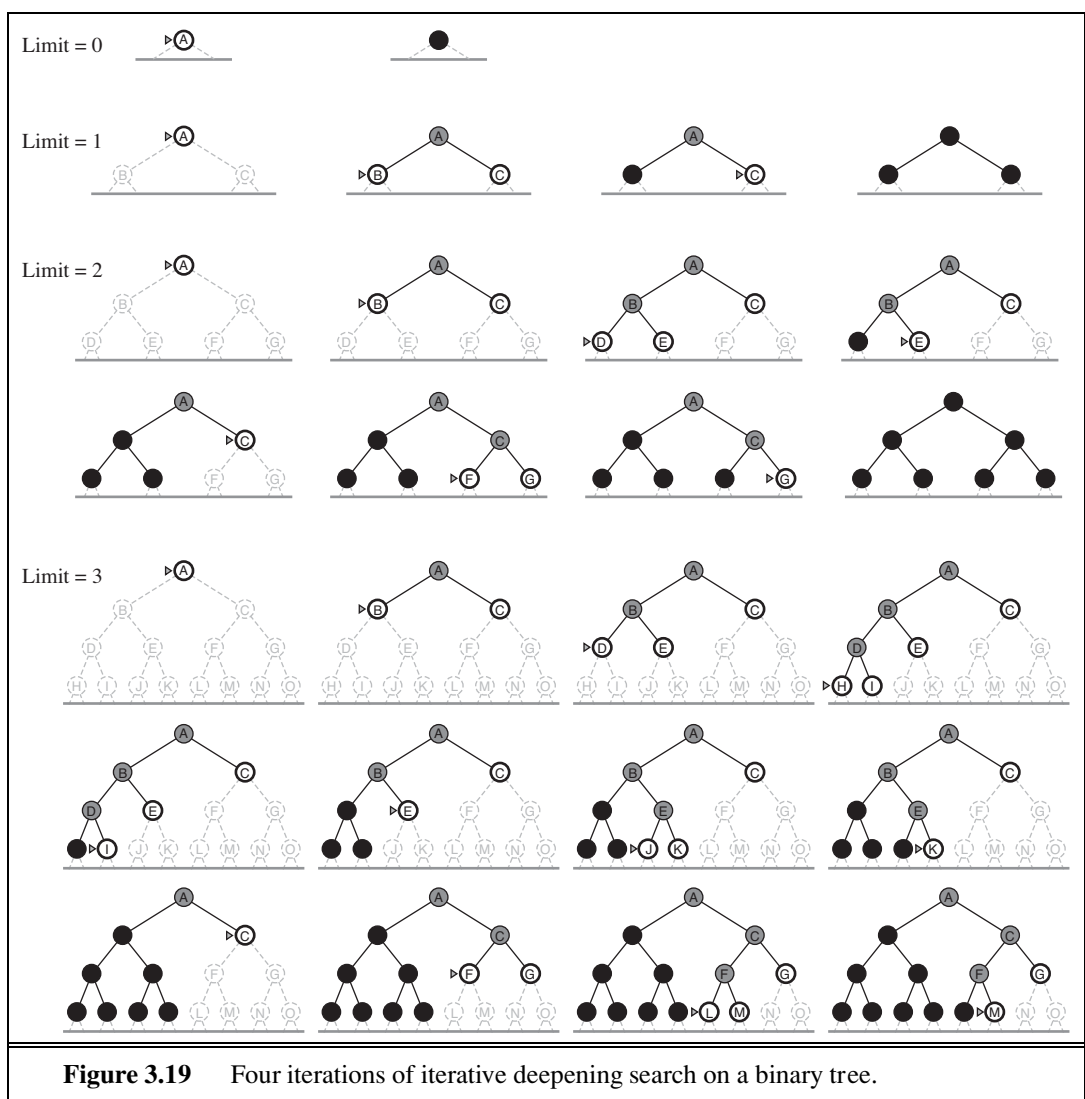
ITERATIVE  
DEEPENING SEARCH

```

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result

```

**Figure 3.18** 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 3.19** Four iterations of iterative deepening search on a binary tree.

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 in the worst case is

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

which gives a time complexity of  $O(b^d)$ —asymptotically the same as breadth-first search. There is some extra cost for generating the upper levels multiple times, but it is not large. For example, if  $b = 10$  and  $d = 5$ , the numbers are

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

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110.$$

If you are really concerned about repeating the repetition, you can use a hybrid approach that runs breadth-first search until almost all the available memory is consumed, and then runs iterative deepening from all the nodes in the frontier. *In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.*

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 3.17. It turns out, unfortunately, that iterative lengthening incurs substantial overhead compared to uniform-cost search.

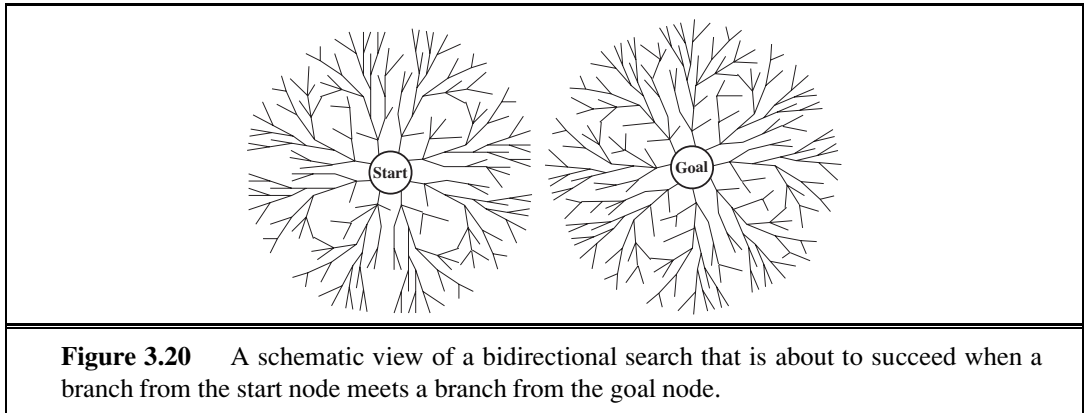


ITERATIVE  
LENGTHENING  
SEARCH

### 3.4.6 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—hoping that the two searches meet in the middle (Figure 3.20). 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 replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found. (It is important to realize that the first such solution found may not be optimal, even if the two searches are both breadth-first; some additional search is required to make sure there isn't another short-cut across the gap.) The check can be done when each node is generated or selected for expansion and, with a hash table, will take constant time. 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 they have generated all of the nodes at depth 3. For  $b = 10$ , this means a total of 2,220 node generations, compared with 1,111,110 for a standard breadth-first search. Thus, the time complexity of bidirectional search using breadth-first searches in both directions is  $O(b^{d/2})$ . The space complexity is also  $O(b^{d/2})$ . We can reduce this by roughly half if one of the two searches is done by iterative deepening, but at least one of the frontiers must be kept in memory so that the intersection check can be done. This space requirement is the most significant weakness of bidirectional search.



PREDECESSOR

The reduction in time complexity makes bidirectional search attractive, but how do we search backward? This is not as easy as it sounds. Let the **predecessors** of a state  $x$  be all those states that have  $x$  as a successor. Bidirectional search requires a method for computing predecessors. When all the actions in the state space are reversible, the predecessors of  $x$  are just its successors. 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. But if the goal is an abstract description, such as the goal that “no queen attacks another queen” in the  $n$ -queens problem, then bidirectional search is difficult to use.

### 3.4.7 Comparing uninformed search strategies

Figure 3.21 compares search strategies in terms of the four evaluation criteria set forth in Section 3.3.2. This comparison is for tree-search versions. For graph searches, the main differences are that depth-first search is complete for finite state spaces and that the space and time complexities are bounded by the size of the state space.

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

**Figure 3.21** Evaluation of tree-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: <sup>a</sup> complete if  $b$  is finite; <sup>b</sup> complete if step costs  $\geq \epsilon$  for positive  $\epsilon$ ; <sup>c</sup> optimal if step costs are all identical; <sup>d</sup> if both directions use breadth-first search.

### 3.5 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 can an uninformed strategy.

#### BEST-FIRST SEARCH

The general approach we 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**,  $f(n)$ . The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first. The implementation of best-first graph search is identical to that for uniform-cost search (Figure 3.14), except for the use of  $f$  instead of  $g$  to order the priority queue.

#### EVALUATION FUNCTION

The choice of  $f$  determines the search strategy. (For example, as Exercise 3.21 shows, best-first tree search includes depth-first search as a special case.) Most best-first algorithms include as a component of  $f$  a **heuristic function**, denoted  $h(n)$ :

#### HEURISTIC FUNCTION

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

(Notice that  $h(n)$  takes a *node* as input, but, unlike  $g(n)$ , it depends only on the *state* at that 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 study heuristics in more depth in Section 3.6. For now, we consider them to be arbitrary, nonnegative, 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.

#### 3.5.1 Greedy best-first search

#### GREEDY BEST-FIRST SEARCH

**Greedy best-first search**<sup>8</sup> 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; that is,  $f(n) = h(n)$ .

#### STRAIGHT-LINE DISTANCE

Let us see how this works for route-finding problems in Romania; we use the **straight-line distance** heuristic, which we will call  $h_{SLD}$ . If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure 3.22. For example,  $h_{SLD}(In(Arad)) = 366$ . Notice that the values of  $h_{SLD}$  cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that  $h_{SLD}$  is correlated with actual road distances and is, therefore, a useful heuristic.

Figure 3.23 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

<sup>8</sup> 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).

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

**Figure 3.22** Values of  $h_{SLD}$ —straight-line distances to Bucharest.

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.

Greedy best-first tree search is also incomplete even in a finite state space, much like depth-first search. 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. The algorithm will never find this solution, however, because expanding Neamt puts Iasi back into the frontier, Iasi is closer to Fagaras than Vaslui is, and so Iasi will be expanded again, leading to an infinite loop. (The graph search version *is* complete in finite spaces, but not in infinite ones.) The worst-case time and space complexity for the tree version 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.

### 3.5.2 A\* search: Minimizing the total estimated solution cost

#### A\* SEARCH

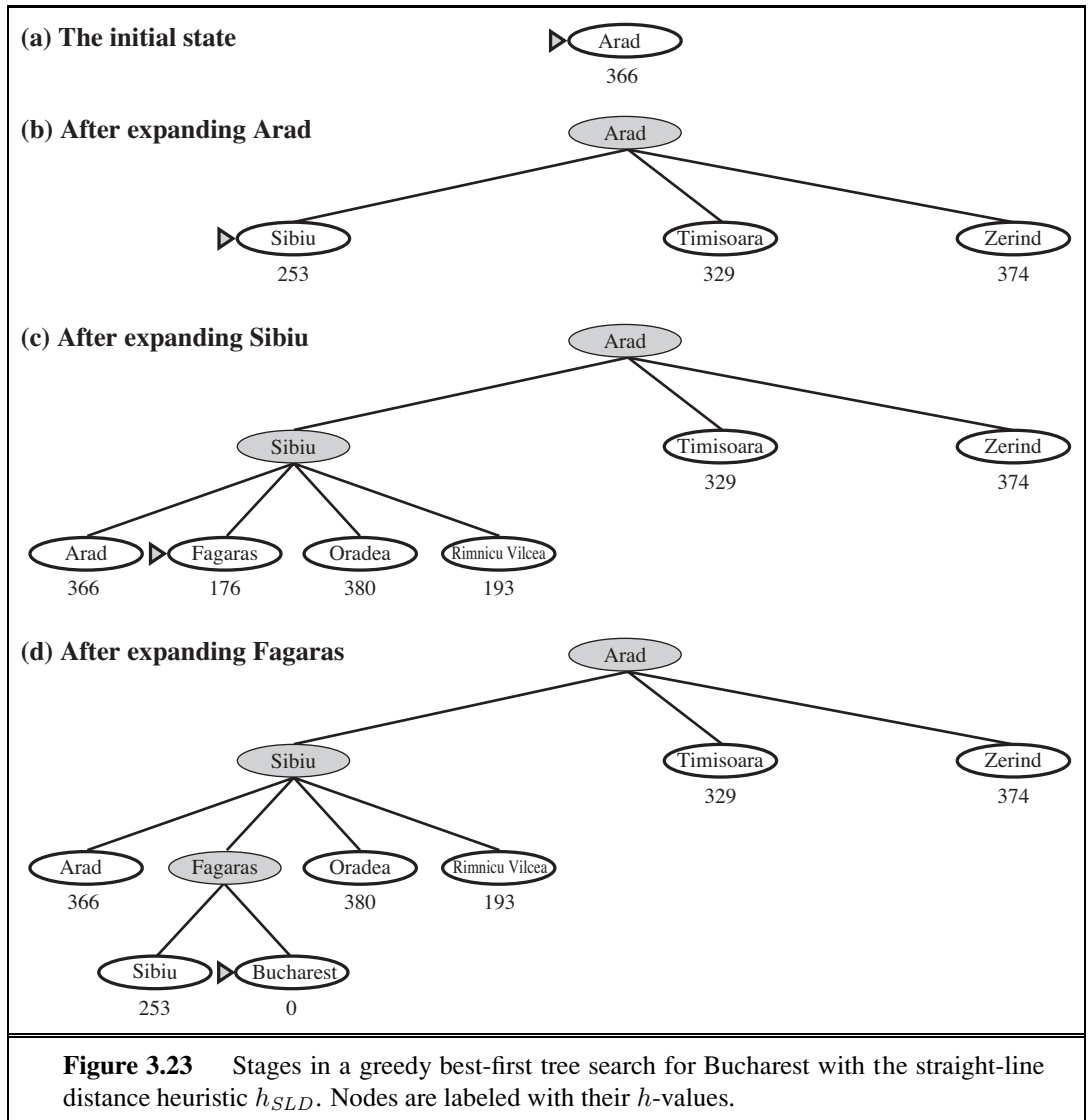
The most widely known form of best-first search is called **A\* search** (pronounced “A-star search”). It evaluates nodes by combining  $g(n)$ , the cost to reach the node, and  $h(n)$ , the cost to get from the node to the goal:

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

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

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

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of  $g(n) + h(n)$ . It turns out that this strategy is more than just reasonable: provided that the heuristic function  $h(n)$  satisfies certain conditions, A\* search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A\* uses  $g + h$  instead of  $g$ .



### Conditions for optimality: Admissibility and consistency

ADMISSIBLE  
HEURISTIC

The first condition we require for optimality is that  $h(n)$  be an **admissible heuristic**. An admissible heuristic is one that *never overestimates* the cost to reach the goal. Because  $g(n)$  is the actual cost to reach  $n$  along the current path, and  $f(n) = g(n) + h(n)$ , we have as an immediate consequence that  $f(n)$  never overestimates the true cost of a solution along the current path through  $n$ .

Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is. 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 3.24, 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 3.22. Notice in particular that Bucharest first appears on the frontier 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.

CONSISTENCY  
MONOTONICITY

A second, slightly stronger condition called **consistency** (or sometimes **monotonicity**) is required only for applications of A\* to graph search.<sup>9</sup> 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  $G_n$  closest to  $n$ . For an admissible heuristic, the inequality makes perfect sense: if there were a route from  $n$  to  $G_n$  via  $n'$  that was cheaper than  $h(n)$ , that would violate the property that  $h(n)$  is a lower bound on the cost to reach  $G_n$ .

It is fairly easy to show (Exercise 3.29) that every consistent heuristic is also admissible. Consistency is therefore a stricter requirement than admissibility, but 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.

### Optimality of A\*



As we mentioned earlier, A\* has the following properties: *the tree-search version of A\* is optimal if  $h(n)$  is admissible, while the graph-search version is optimal if  $h(n)$  is consistent.*

We show the second of these two claims since it is more useful. The argument essentially mirrors the argument for the optimality of uniform-cost search, with  $g$  replaced by  $f$ —just as in the A\* algorithm itself.



The first step is to establish the following: *if  $h(n)$  is consistent, then the values of  $f(n)$  along any path are nondecreasing.* The proof follows directly from the definition of consistency. Suppose  $n'$  is a successor of  $n$ ; then  $g(n') = g(n) + c(n, a, n')$  for some action  $a$ , and we have

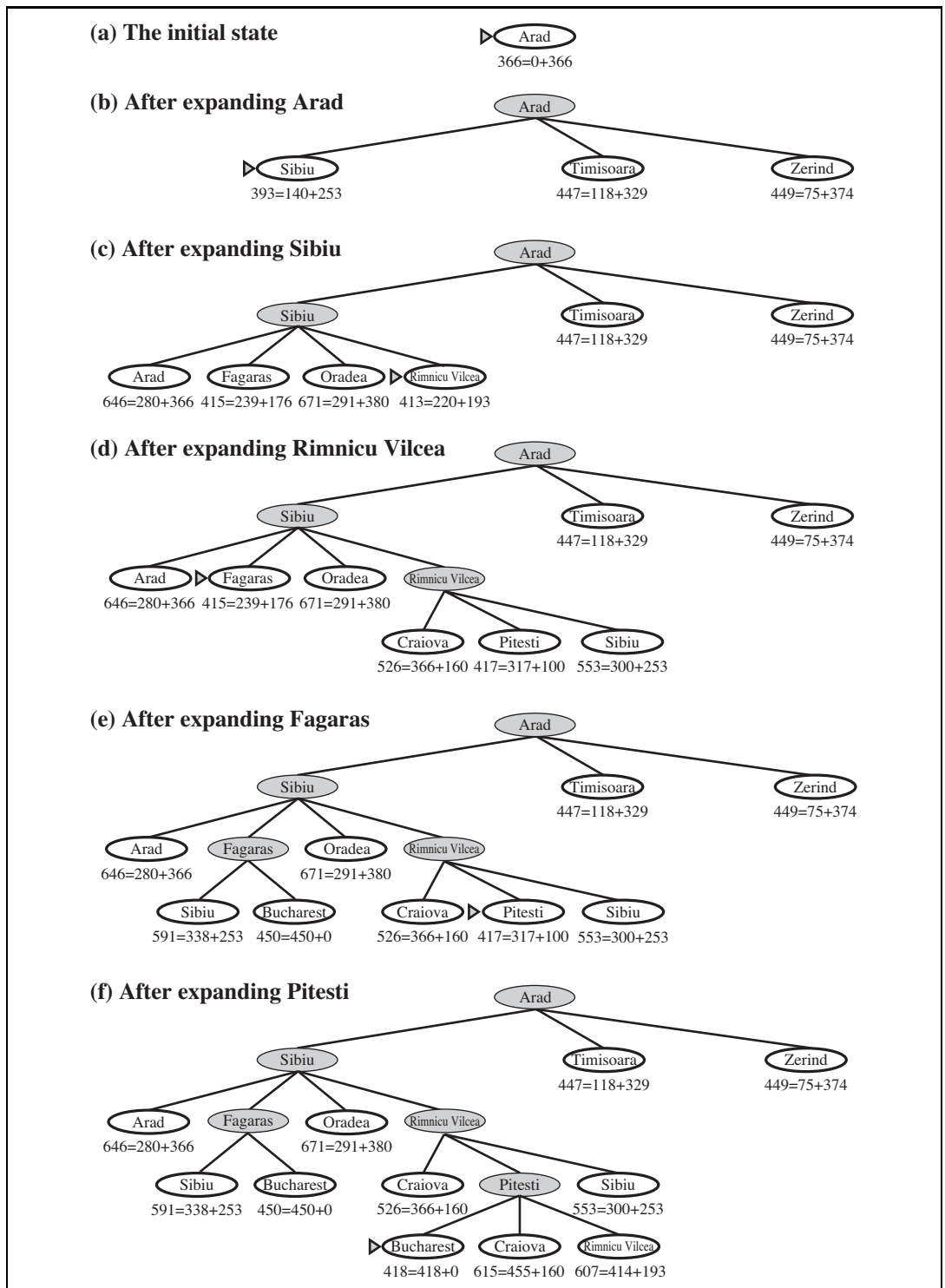
$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n) .$$



The next step is to prove that *whenever A\* selects a node  $n$  for expansion, the optimal path to that node has been found.* Were this not the case, there would have to be another frontier node  $n'$  on the optimal path from the start node to  $n$ , by the graph separation property of

<sup>9</sup> With an admissible but inconsistent heuristic, A\* requires some extra bookkeeping to ensure optimality.





**Figure 3.24** 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 3.22.

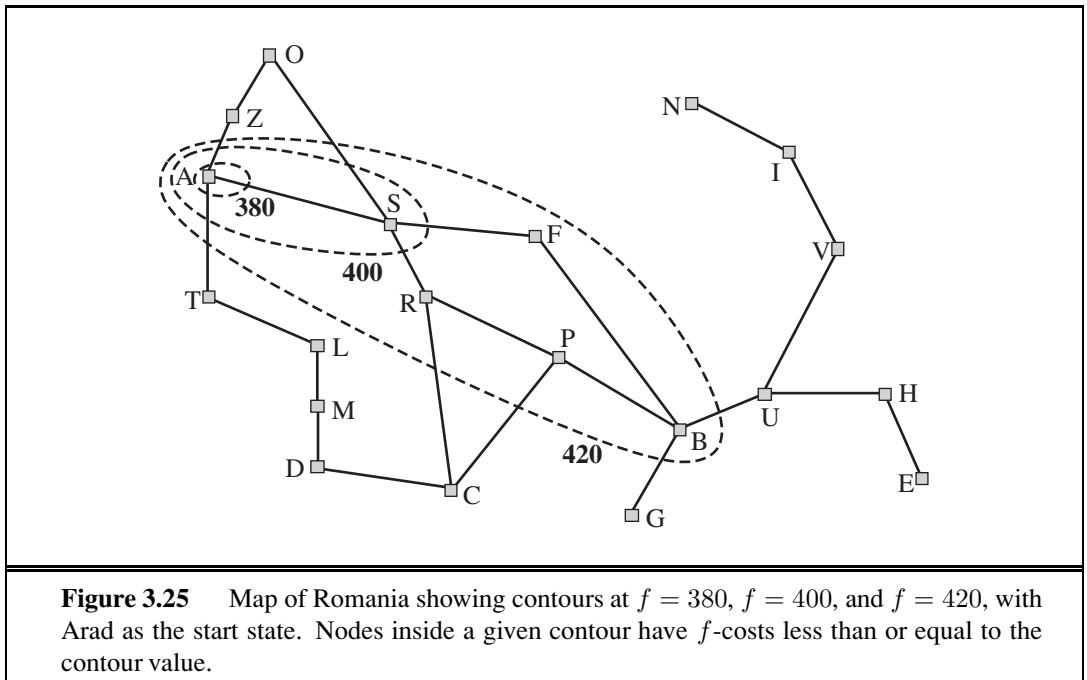


Figure 3.9; because  $f$  is nondecreasing along any path,  $n'$  would have lower  $f$ -cost than  $n$  and would have been selected first.

From the two preceding observations, 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 because  $f$  is the true cost for goal nodes (which have  $h = 0$ ) and all later goal nodes will be at least as expensive.

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 3.25 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 frontier 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.

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  $\epsilon$  and if  $b$  is finite.

Notice that  $A^*$  expands no nodes with  $f(n) > C^*$ —for example, Timisoara is not expanded in Figure 3.24 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 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 and use the same heuristic information— $A^*$  is **optimally efficient** for any given consistent heuristic. 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.

ABSOLUTE ERROR

RELATIVE ERROR

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 states within the goal contour search space is still exponential in the length of the solution. The details of the analysis are beyond the scope of this book, but the basic results are as follows. For problems with constant step costs, the growth in run time as a function of the optimal solution depth  $d$  is analyzed in terms of the **absolute error** or the **relative error** of the heuristic. The absolute error is defined as  $\Delta \equiv h^* - h$ , where  $h^*$  is the actual cost of getting from the root to the goal, and the relative error is defined as  $\epsilon \equiv (h^* - h)/h^*$ .

The complexity results depend very strongly on the assumptions made about the state space. The simplest model studied is a state space that has a single goal and is essentially a tree with reversible actions. (The 8-puzzle satisfies the first and third of these assumptions.) In this case, the time complexity of  $A^*$  is exponential in the maximum absolute error, that is,  $O(b^\Delta)$ . For constant step costs, we can write this as  $O(b^{\epsilon d})$ , where  $d$  is the solution depth. For almost all heuristics in practical use, the absolute error is at least proportional to the path cost  $h^*$ , so  $\epsilon$  is constant or growing and the time complexity is exponential in  $d$ . We can also see the effect of a more accurate heuristic:  $O(b^{\epsilon d}) = O((b^\epsilon)^d)$ , so the effective branching factor (defined more formally in the next section) is  $b^\epsilon$ .

When the state space has many goal states—particularly *near-optimal* goal states—the search process can be led astray from the optimal path and there is an extra cost proportional to the number of goals whose cost is within a factor  $\epsilon$  of the optimal cost. Finally, in the general case of a graph, the situation is even worse. There can be exponentially many states with  $f(n) < C^*$  even if the absolute error is bounded by a constant. For example, consider a version of the vacuum world where the agent can clean up any square for unit cost without even having to visit it: in that case, squares can be cleaned in any order. With  $N$  initially dirty squares, there are  $2^N$  states where some subset has been cleaned and all of them are on an optimal solution path—and hence satisfy  $f(n) < C^*$ —even if the heuristic has an error of 1.

The complexity of  $A^*$  often makes it 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 3.6, we 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

```

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

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    successors  $\leftarrow []$ 
    for each action in problem.ACTIONS(node.STATE) do
        add CHILD-NODE(problem, node, action) into successors
    if successors is empty then return failure,  $\infty$ 
    for each s in successors do /* update f with value from previous search, if any */
        s.f  $\leftarrow \max(s.g + s.h, \text{node.f})$ 
    loop do
        best  $\leftarrow$  the lowest f-value node in successors
        if best.f > f_limit then return failure, best.f
        alternative  $\leftarrow$  the second-lowest f-value among successors
        result, best.f  $\leftarrow$  RBFS(problem, best,  $\min(f\_limit, \text{alternative})$ )
        if result  $\neq$  failure then return result

```

**Figure 3.26** The algorithm for recursive best-first search.

before it runs out of time. For this reason,  $A^*$  is not practical for many large-scale problems. There are, however, algorithms that overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time. We discuss these next.

### 3.5.3 Memory-bounded heuristic search

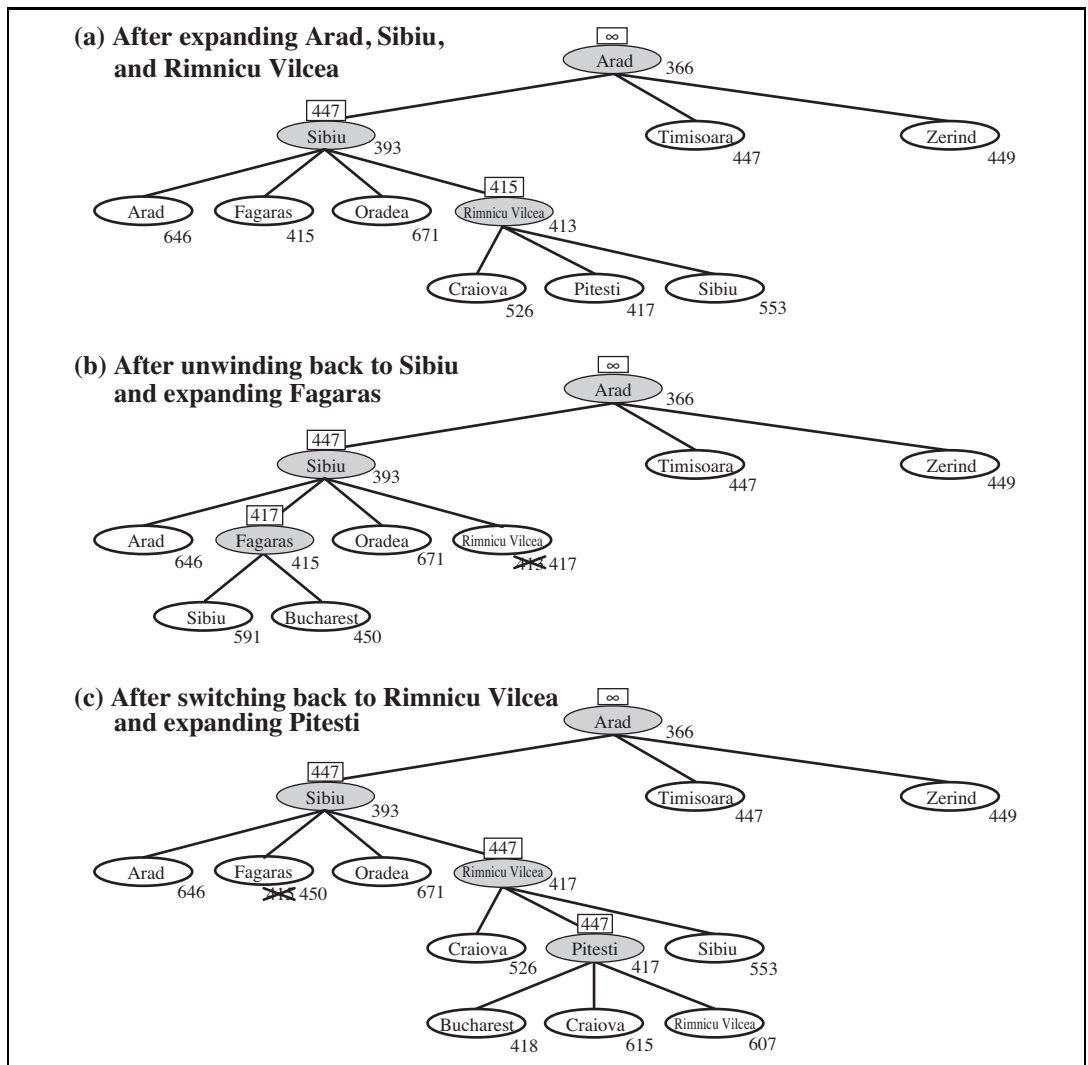
The simplest way to reduce memory requirements for  $A^*$  is to adapt the idea of iterative deepening to the heuristic search context, resulting in the **iterative-deepening  $A^*$**  (IDA\*) algorithm. The main difference between IDA\* and standard iterative deepening is that the cutoff used is the *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 3.17. This section briefly examines two other memory-bounded algorithms, called RBFS and MA\*.

**Recursive best-first search** (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. The algorithm is shown in Figure 3.26. Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the *f\_limit* variable to keep 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 a **backed-up value**—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

ITERATIVE-  
DEEPENING  
 $A^*$

RECURSIVE  
BEST-FIRST SEARCH

BACKED-UP VALUE



**Figure 3.27** 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, and every node is labeled with its  $f$ -cost. (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.

reexpanding the subtree at some later time. Figure 3.27 shows how RBFS reaches Bucharest.

RBFS is somewhat more efficient than IDA\*, but still suffers from excessive node re-generation. In the example in Figure 3.27, RBFS 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, its  $f$ -value is likely to increase— $h$  is usually less optimistic for nodes closer to the goal. When this happens, 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\* tree search, 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.

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, but it uses only linear space: even if more memory were available, RBFS has no way to make use of it. Because they forget most of what they have done, both algorithms may end up reexpanding the same states many times over. Furthermore, they suffer the potentially exponential increase in complexity associated with redundant paths in graphs (see Section 3.3).

It seems sensible, therefore, to use all available memory. Two algorithms that do this are **MA\*** (memory-bounded A\*) and **SMA\*** (simplified MA\*). SMA\* is—well—simpler, so we will describe it. SMA\* proceeds just like A\*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA\* always drops the *worst* leaf node—the one with the highest  $f$ -value. Like RBFS, SMA\* then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA\* regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten. Another way of saying this is that, if all the descendants of a node  $n$  are forgotten, then we will not know which way to go from  $n$ , but we will still have an idea of how worthwhile it is to go anywhere from  $n$ .

The complete algorithm is too complicated to reproduce here,<sup>10</sup> 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? To avoid selecting the same node for deletion and expansion, SMA\* expands the *newest* best leaf and deletes the *oldest* worst leaf. These coincide when there is only one leaf, but 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\* is a fairly robust choice 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 overhead of maintaining the frontier and the explored set.

<sup>10</sup> A rough sketch appeared in the first edition of this book.

MA\*  
SMA\*

THRASHING



On very hard problems, however, it will often be the case that SMA\* is forced to switch back and forth continually among many 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 no current theory explains the tradeoff between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.

### 3.5.4 Learning to search better

METALEVEL STATE SPACE

OBJECT-LEVEL STATE SPACE

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 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 3.24, 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

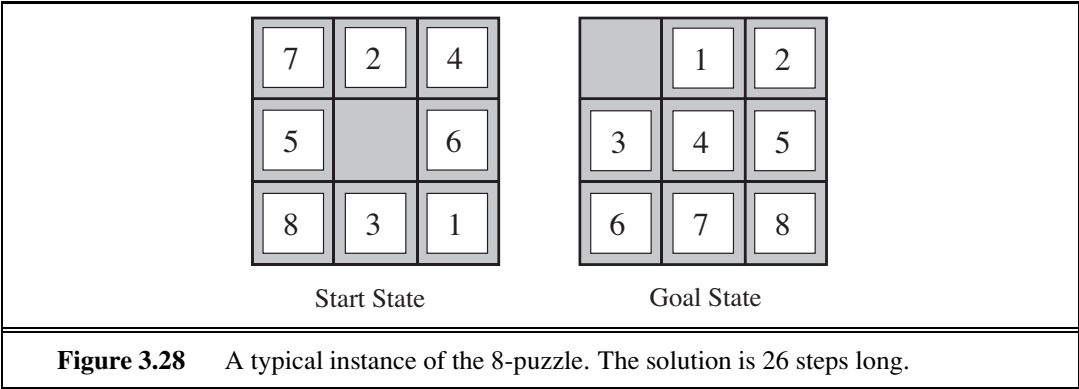
Now, the path in Figure 3.24 has five steps, including one step, the expansion of Fagaras, that is not especially helpful. For harder problems, there will be many such missteps, and a **metalevel learning** algorithm can learn from these experiences to avoid exploring unpromising subtrees. The techniques 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.

## 3.6 HEURISTIC FUNCTIONS

In this section, we 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 3.28).

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, four moves are possible; when it is in a corner, two; and when it is along an edge, three.) This means that an exhaustive tree search to depth 22 would look at about  $3^{22} \approx 3.1 \times 10^{10}$  states. A graph search would cut this down by a factor of about 170,000 because only  $9!/2 = 181,440$  distinct states 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 3.28, all of the eight tiles are out of position, so the start state would have  $h_1 = 8$ .  $h_1$  is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.
- $h_2$  = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the **city block distance** or **Manhattan distance**.  $h_2$  is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

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

As expected, neither of these overestimates the true solution cost, which is 26.

3.6.1 The effect of heuristic accuracy on performance

One way to characterize the quality of a heuristic is the **effective branching factor**  $b^*$ . If the total number of nodes generated by  $A^*$  for a particular problem is  $N$  and the solution depth is  $d$ , then  $b^*$  is the branching factor that a uniform tree of depth  $d$  would have to have in order to contain  $N + 1$  nodes. Thus,

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

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. (The existence of an effective branching factor follows from the result, mentioned earlier, that the number of nodes expanded by  $A^*$  grows exponentially with solution depth.) 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 at reasonable computational cost.



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 3.29 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. Even for small problems with  $d = 12$ , A\* with  $h_2$  is 50,000 times more efficient than uninformed iterative deepening search.

$d$	Search Cost (nodes generated)			Effective Branching Factor		
	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 3.29** Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A\* algorithms with  $h_1$ ,  $h_2$ . Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths  $d$ .

One might ask whether  $h_2$  is *always* better than  $h_1$ . The answer is “Essentially, 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 97 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 cause other nodes to be expanded as well. Hence, it is generally better to use a heuristic function with higher values, provided it is consistent and that the computation time for the heuristic is not too long.

DOMINATION

### 3.6.2 Generating admissible heuristics from relaxed problems

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

## RELAXED PROBLEM



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 state-space graph of the relaxed problem is a *supergraph* of the original state space because the removal of restrictions creates added edges in the graph.

Because the relaxed problem adds edges to the state space, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have *better* solutions if the added edges provide short cuts. Hence, *the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem*. Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore **consistent** (see page 95).

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.<sup>11</sup> 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,

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 3.31. 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.<sup>12</sup>

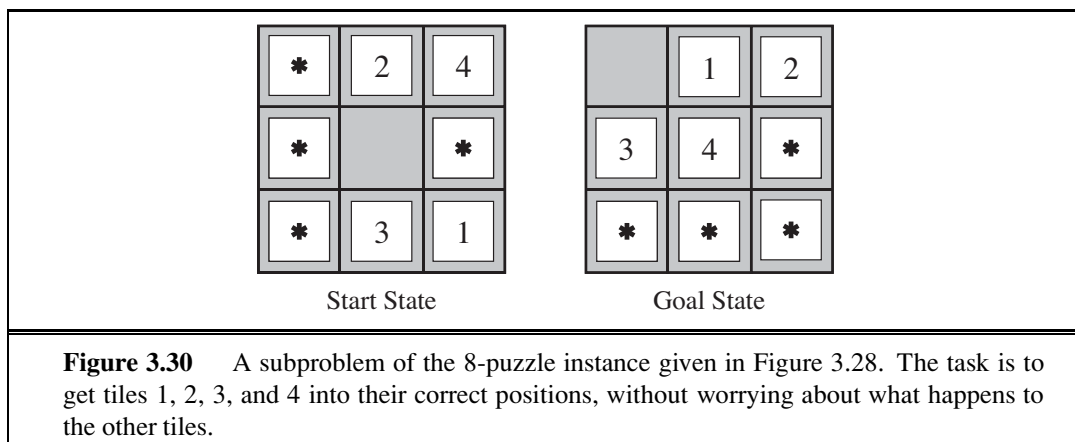
A program called ABSOLVER can generate heuristics automatically from problem definitions, using the “relaxed problem” method and various other techniques (Prieditis, 1993). ABSOLVER generated a new heuristic for the 8-puzzle that was better than any preexisting heuristic and found the first useful heuristic for the famous Rubik’s Cube puzzle.

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

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

<sup>11</sup> In Chapters 8 and 10, we 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 use English.

<sup>12</sup> 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.



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.

### 3.6.3 Generating admissible heuristics from subproblems: Pattern databases

SUBPROBLEM

Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem. For example, Figure 3.30 shows a subproblem of the 8-puzzle instance in Figure 3.28. 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 more accurate than Manhattan distance in some cases.

PATTERN DATABASE

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. (The locations of the other four tiles are irrelevant for the purposes of solving the subproblem, but moves of those tiles do count toward 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 back<sup>13</sup> from the goal and recording the cost of each new pattern encountered; the expense of this search is amortized over many subsequent problem instances.

The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, 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

<sup>13</sup> By working backward from the goal, the exact solution cost of every instance encountered is immediately available. This is an example of **dynamic programming**, which we discuss further in Chapter 17.

unlikely that 1-2-3-4 can be moved into place without touching 5-6-7-8, and vice versa. But what if we don't count those moves? That is, we record not the total cost of solving the 1-2-3-4 subproblem, but just the number of moves involving 1-2-3-4. Then it is easy to see that the sum of the two costs is still a lower bound on the cost of solving the entire problem. This is the idea behind **disjoint pattern databases**. With such databases, it is possible to solve random 15-puzzles in a few milliseconds—the number of nodes generated is reduced by a factor of 10,000 compared with the use of Manhattan distance. For 24-puzzles, a speedup of roughly a factor of a million can be obtained.

Disjoint pattern databases work for sliding-tile puzzles because the problem can be divided up in such a way that each move affects only one subproblem—because only one tile is moved at a time. For a problem such as Rubik's Cube, this kind of subdivision is difficult because each move affects 8 or 9 of the 26 cubies. More general ways of defining additive, admissible heuristics have been proposed that do apply to Rubik's cube (Yang *et al.*, 2008), but they have not yielded a heuristic better than the best nonadditive heuristic for the problem.

### 3.6.4 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 sections—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, a 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.)

Inductive learning methods work best when supplied with **features** of a state that are relevant to predicting the state's value, 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 not 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:

$$h(n) = c_1 x_1(n) + c_2 x_2(n) .$$

The constants  $c_1$  and  $c_2$  are adjusted to give the best fit to the actual data on solution costs. One expects both  $c_1$  and  $c_2$  to be positive because misplaced tiles and incorrect adjacent pairs make the problem harder to solve. Notice that this heuristic does satisfy the condition that  $h(n) = 0$  for goal states, but it is not necessarily admissible or consistent.