# CHAPTER 4

# SEARCH IN COMPLEX ENVIRONMENTS

*In which we relax the simplifying assumptions of the previous chapter, to get closer to the real world.*

Chapter 3 addressed problems in fully observable, deterministic, static, known environments where the solution is a sequence of actions. In this chapter, we relax those constraints. We begin with the problem of finding a good state without worrying about the path to get there, covering both discrete (Section 4.1) and continuous (Section 4.2) states. Then we relax the assumptions of determinism (Section 4.3) and observability (Section 4.4). In a nondeterministic world, the agent will need a conditional plan and carry out different actions depending on what it observes—for example, stopping if the light is red and going if it is green. With partial observability, the agent will also need to keep track of the possible states it might be in. Finally, Section 4.5 guides the agent through an unknown space that it must learn as it goes, using **online search**.

## 4.1 Local Search and Optimization Problems

In the search problems of Chapter 3 we wanted to find paths through the search space, such as a path from Arad to Bucharest. But sometimes we care only about the final state, not the path to get there. For example, in the 8-queens problem (Figure 4.3), we care only about finding a valid final configuration of 8 queens (because if you know the configuration, it is trivial to reconstruct the steps that created it). This is also true for many important applications such as integrated-circuit design, factory floor layout, job shop scheduling, automatic programming, telecommunications network optimization, crop planning, and portfolio management.

Local search

**Local search** algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic—they might never explore a portion of the search space where a solution actually resides. However, they have two key advantages: (1) they use very little memory; and (2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

Optimization problem
Objective function

Local search algorithms can also solve **optimization problems**, in which the aim is to find the best state according to an **objective function**.

State-space landscape

To understand local search, consider the states of a problem laid out in a **state-space landscape**, as shown in Figure 4.1. Each point (state) in the landscape has an "elevation," defined by the value of the objective function. If elevation corresponds to an objective function,

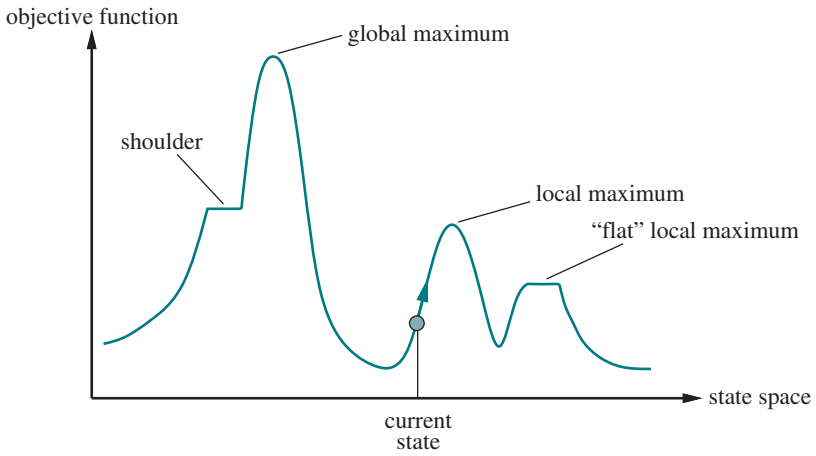



**Figure 4.1** A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
    neighbor ← a highest-valued successor state of current
    if VALUE(neighbor) ≤ VALUE(current) then return current
    current ← neighbor
```

**Figure 4.2** The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor.

then the aim is to find the highest peak—a **global maximum**—and we call the process **hill climbing**. If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**—and we call it **gradient descent**.

Global maximum

Global minimum

### 4.1.1 Hill-climbing search

The **hill-climbing** search algorithm is shown in Figure 4.2. It keeps track of one current state and on each iteration moves to the neighboring state with highest value—that is, it heads in the direction that provides the **steepest ascent**. It terminates when it reaches a "peak" where no neighbor has a higher value. Hill climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia. Note that one way to use hill-climbing search is to use the negative of a heuristic cost function as the objective function; that will climb locally to the state with smallest heuristic distance to the goal.

Hill climbing

Steepest ascent

To illustrate hill climbing, we will use the **8-queens problem** (Figure 4.3). We will use a **complete-state formulation**, which means that every state has all the components of a solution, but they might not all be in the right place. In this case every state has 8 queens

Complete-state formulation

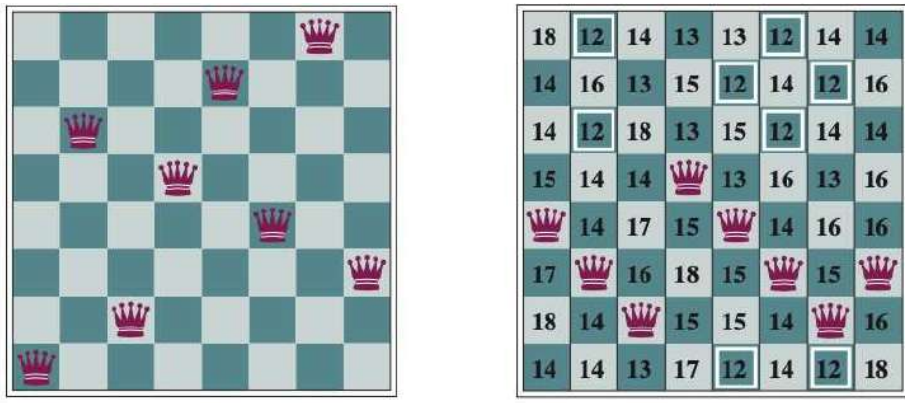



**Figure 4.3** (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h = 17$. The board shows the value of $h$ for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h = 12$. The hill-climbing algorithm will pick one of these.

on the board, one per column. The initial state is chosen at random, and the successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has $8 \times 7 = 56$ successors). The heuristic cost function $h$ is the number of pairs of queens that are attacking each other; this will be zero only for solutions. (It counts as an attack if two pieces are in the same line, even if there is an intervening piece between them.) Figure 4.3(b) shows a state that has $h = 17$. The figure also shows the $h$ values of all its successors.

Greedy local search

Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing can make rapid progress toward a solution because it is usually quite easy to improve a bad state. For example, from the state in Figure 4.3(b), it takes just five steps to reach the state in Figure 4.3(a), which has $h = 1$ and is very nearly a solution. Unfortunately, hill climbing can get stuck for any of the following reasons:

Local maximum

- **Local maxima**: A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go. Figure 4.1 illustrates the problem schematically. More concretely, the state in Figure 4.3(a) is a local maximum (i.e., a local minimum for the cost $h$); every move of a single queen makes the situation worse.

Ridge

- **Ridges**: A ridge is shown in Figure 4.4. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

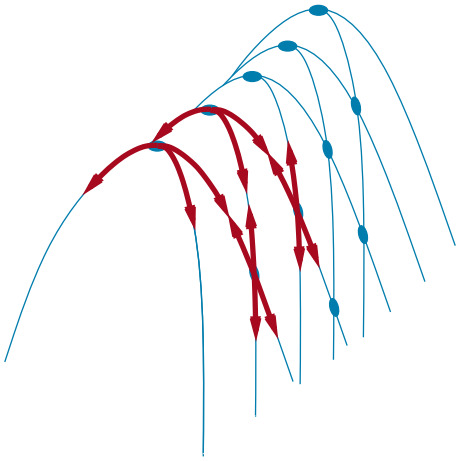


**Figure 4.4** Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill. Topologies like this are common in low-dimensional state spaces, such as points in a two-dimensional plane. But in state spaces with hundreds or thousands of dimensions, this intuitive picture does not hold, and there are usually at least a few dimensions that make it possible to escape from ridges and plateaus.

- **Plateaus**: A plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder**, from which progress is possible. (See Figure 4.1.) A hill-climbing search can get lost wandering on the plateau.

Plateau
Shoulder

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

How could we solve more problems? One answer is to keep going when we reach a plateau—to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 4.1. But if we are actually on a flat local maximum, then this approach will wander on the plateau forever. Therefore, we can limit the number of consecutive sideways moves, stopping after, say, 100 consecutive sideways moves. This raises the percentage of problem instances solved by hill climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

Sideways move

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

Stochastic hill climbing

First-choice hill climbing

Another variant is **random-restart hill climbing**, which adopts the adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly

Random-restart hill climbing

generated initial states, until a goal is found. It is complete with probability 1, because it will eventually generate a goal state as the initial state. If each hill-climbing search has a probability $p$ of success, then the expected number of restarts required is $1/p$. For 8-queens instances with no sideways moves allowed, $p \approx 0.14$, so we need roughly 7 iterations to find a goal (6 failures and 1 success). The expected number of steps is the cost of one successful iteration plus $(1-p)/p$ times the cost of failure, or roughly 22 steps in all. When we allow sideways moves, $1/0.94 \approx 1.06$ iterations are needed on average and $(1 \times 21) + (0.06/0.94) \times 64 \approx 25$ steps. For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in seconds.[1]

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

### 4.1.2 Simulated annealing

A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum. In contrast, a purely random walk that moves to a successor state without concern for the value will eventually stumble upon the global maximum, but will be extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in a way that yields both efficiency and completeness.

Simulated annealing

**Simulated annealing** is such an algorithm. In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state. To explain simulated annealing, we switch our point of view from hill climbing to **gradient descent** (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a very bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum—perhaps into a deeper local minimum, where it will spend more time. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

The overall structure of the simulated-annealing algorithm (Figure 4.5) is similar to hill climbing. Instead of picking the *best* move, however, it picks a *random* move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount $\Delta E$ by which the evaluation is worsened. The probability also decreases as the "temperature" $T$ goes down: "bad" moves are more likely to be allowed at the start when $T$ is high, and they become more unlikely as $T$ decreases. If the *schedule* lowers $T$ to 0 slowly enough, then a property of the Boltzmann distribution, $e^{\Delta E/T}$, is that

[1] Luby *et al.* (1993) suggest restarting after a fixed number of steps and show that this can be *much* more efficient than letting each search continue indefinitely.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current ← problem.INITIAL
  for t = 1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← VALUE(current) – VALUE(next)
    if ΔE > 0 then current ← next
    else current ← next only with probability e^{−ΔE/T}
```

**Figure 4.5** The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the "temperature" $T$ as a function of time.

all the probability is concentrated on the global maxima, which the algorithm will find with probability approaching 1.

Simulated annealing was used to solve VLSI layout problems beginning in the 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

### 4.1.3 Local beam search

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

Local beam search

At first sight, a local beam search with $k$ states might seem to be nothing more than running $k$ random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different. In a random-restart search, each search process runs independently of the others. *In a local beam search, useful information is passed among the parallel search threads.* In effect, the states that generate the best successors say to the others, "Come over here, the grass is greener!" The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

Local beam search can suffer from a lack of diversity among the $k$ states—they can become clustered in a small region of the state space, making the search little more than a $k$-times-slower version of hill climbing. A variant called **stochastic beam search**, analogous to stochastic hill climbing, helps alleviate this problem. Instead of choosing the top $k$ successors, stochastic beam search chooses successors with probability proportional to the successor's value, thus increasing diversity.

Stochastic beam search

### 4.1.4 Evolutionary algorithms

**Evolutionary algorithms** can be seen as variants of stochastic beam search that are explicitly motivated by the metaphor of natural selection in biology: there is a population of individuals (states), in which the fittest (highest value) individuals produce offspring (successor states) that populate the next generation, a process called **recombination**. There are endless forms of evolutionary algorithms, varying in the following ways:

Evolutionary algorithms

Recombination

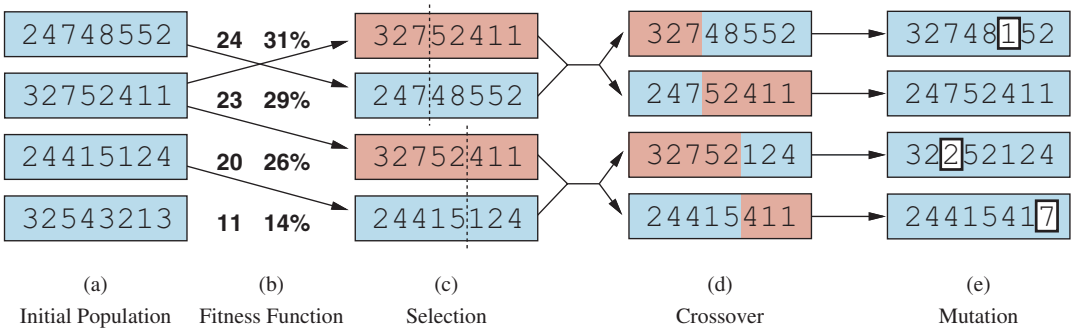



**Figure 4.6** A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

- The size of the population.

Genetic algorithm

- The representation of each individual. In **genetic algorithms**, each individual is a string over a finite alphabet (often a Boolean string), just as DNA is a string over the alphabet **ACGT**. In **evolution strategies**, an individual is a sequence of real numbers, and in **genetic programming** an individual is a computer program.

Evolution strategies
Genetic programming

- The mixing number, $\rho$, which is the number of parents that come together to form offspring. The most common case is $\rho = 2$: two parents combine their "genes" (parts of their representation) to form offspring. When $\rho = 1$ we have stochastic beam search (which can be seen as asexual reproduction). It is possible to have $\rho > 2$, which occurs only rarely in nature but is easy enough to simulate on computers.

Selection

- The **selection** process for selecting the individuals who will become the parents of the next generation: one possibility is to select from all individuals with probability proportional to their fitness score. Another possibility is to randomly select $n$ individuals ($n > \rho$), and then select the $\rho$ most fit ones as parents.

Crossover point

- The recombination procedure. One common approach (assuming $\rho = 2$), is to randomly select a **crossover point** to split each of the parent strings, and recombine the parts to form two children, one with the first part of parent 1 and the second part of parent 2; the other with the second part of parent 1 and the first part of parent 2.

Mutation rate

- The **mutation rate**, which determines how often offspring have random mutations to their representation. Once an offspring has been generated, every bit in its composition is flipped with probability equal to the mutation rate.

Elitism

- The makeup of the next generation. This can be just the newly formed offspring, or it can include a few top-scoring parents from the previous generation (a practice called **elitism**, which guarantees that overall fitness will never decrease over time). The practice of **culling**, in which all individuals below a given threshold are discarded, can lead to a speedup (Baum *et al.*, 1995).

Figure 4.6(a) shows a population of four 8-digit strings, each representing a state of the 8-queens puzzle: the $c$-th digit represents the row number of the queen in column $c$. In (b), each state is rated by the fitness function. Higher fitness values are better, so for the 8-

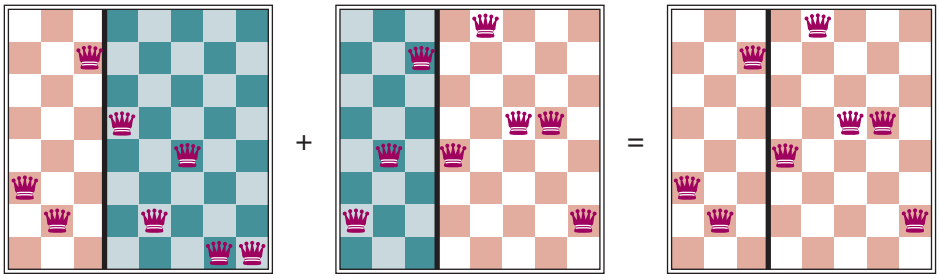


**Figure 4.7** The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The green columns are lost in the crossover step and the red columns are retained. (To interpret the numbers in Figure 4.6: row 1 is the bottom row, and 8 is the top row.)

queens problem we use the number of *nonattacking* pairs of queens, which has a value of $8 \times 7/2 = 28$ for a solution. The values of the four states in (b) are 24, 23, 20, and 11. The fitness scores are then normalized to probabilities, and the resulting values are shown next to the fitness values in (b).

In (c), two pairs of parents are selected, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all. For each selected pair, a crossover point (dotted line) is chosen randomly. In (d), we cross over the parent strings at the crossover points, yielding new offspring. For example, the first child of the first pair gets the first three digits (327) from the first parent and the remaining digits (48552) from the second parent. The 8-queens states involved in this recombination step are shown in Figure 4.7.

Finally, in (e), each location in each string is subject to random mutation with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column. It is often the case that the population is diverse early on in the process, so crossover frequently takes large steps in the state space early in the search process (as in simulated annealing). After many generations of selection towards higher fitness, the population becomes less diverse, and smaller steps are typical. Figure 4.8 describes an algorithm that implements all these steps.

Genetic algorithms are similar to stochastic beam search, but with the addition of the crossover operation. This is advantageous if there are blocks that perform useful functions. For example, it could be that putting the first three queens in positions 2, 4, and 6 (where they do not attack each other) constitutes a useful block that can be combined with other useful blocks that appear in other individuals to construct a solution. It can be shown mathematically that, if the blocks do not serve a purpose—for example if the positions of the genetic code are randomly permuted—then crossover conveys no advantage.

The theory of genetic algorithms explains how this works using the idea of a **schema**, which is a substring in which some of the positions can be left unspecified. For example, the schema 246***** describes all 8-queens states in which the first three queens are in positions 2, 4, and 6, respectively. Strings that match the schema (such as 24613578) are called **instances** of the schema. It can be shown that if the average fitness of the instances of a schema is above the mean, then the number of instances of the schema will grow over time.

Schema

Instance

## Evolution and Search

The theory of **evolution** was developed by Charles Darwin in *On the Origin of Species by Means of Natural Selection* (1859) and independently by Alfred Russel Wallace (1858). The central idea is simple: variations occur in reproduction and will be preserved in successive generations approximately in proportion to their effect on reproductive fitness.

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

The analogy to local search algorithms has already been described; the principal difference between stochastic beam search and evolution is the use of *sexual* reproduction, wherein successors are generated from *multiple* individuals rather than just one. The actual mechanisms of evolution are, however, far richer than most genetic algorithms allow. For example, mutations can involve reversals, duplications, and movement of large chunks of DNA; some viruses borrow DNA from one organism and insert it into another; and there are transposable genes that do nothing but copy themselves many thousands of times within the genome.

There are even genes that poison cells from potential mates that do not carry the gene, thereby increasing their own chances of replication. Most important is the fact that the *genes themselves encode the mechanisms* whereby the genome is reproduced and translated into an organism. In genetic algorithms, those mechanisms are a separate program that is not represented within the strings being manipulated.

Darwinian evolution may appear inefficient, having generated blindly some $10^{43}$ or so organisms without improving its search heuristics one iota. But learning does play a role in evolution. Although the otherwise great French naturalist Jean Lamarck (1809) was wrong to propose that traits acquired by adaptation during an organism's lifetime would be passed on to its offspring, James Baldwin's (1896) superficially similar theory is correct: learning can effectively relax the fitness landscape, leading to an acceleration in the rate of evolution. An organism that has a trait that is not quite adaptive for its environment will pass on the trait if it also has enough plasticity to learn to adapt to the environment in a way that is beneficial. Computer simulations (Hinton and Nowlan, 1987) confirm that this **Baldwin effect** is real, and that a consequence is that things that are hard to learn end up in the genome, but things that are easy to learn need not reside there (Morgan and Griffiths, 2015).

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (small random probability) then child ← MUTATE(child)
      add child to population2
    population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

**Figure 4.8** A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

Clearly, this effect is unlikely to be significant if adjacent bits are totally unrelated to each other, because then there will be few contiguous blocks that provide a consistent benefit. Genetic algorithms work best when schemas correspond to meaningful components of a solution. For example, if the string is a representation of an antenna, then the schemas may represent components of the antenna, such as reflectors and deflectors. A good component is likely to be good in a variety of different designs. This suggests that successful use of genetic algorithms requires careful engineering of the representation.

In practice, genetic algorithms have their place within the broad landscape of optimization methods (Marler and Arora, 2004), particularly for complex structured problems such as circuit layout or job-shop scheduling, and more recently for evolving the architecture of deep neural networks (Miikkulainen *et al.*, 2019). It is not clear how much of the appeal of genetic algorithms arises from their superiority on specific tasks, and how much from the appealing metaphor of evolution.

## 4.2 Local Search in Continuous Spaces

In Chapter 2, we explained the distinction between discrete and continuous environments, pointing out that most real-world environments are continuous. A continuous action space has an infinite branching factor, and thus can't be handled by most of the algorithms we have covered so far (with the exception of first-choice hill climbing and simulated annealing).

This section provides a *very brief* introduction to some local search techniques for continuous spaces. The literature on this topic is vast; many of the basic techniques originated

in the 17th century, after the development of calculus by Newton and Leibniz.[2] We find uses for these techniques in several places in this book, including the chapters on learning, vision, and robotics.

We begin with an example. Suppose we want to place three new airports anywhere in Romania, such that the sum of squared straight-line distances from each city on the map to its nearest airport is minimized. (See Figure 3.1 for the map of Romania.) The state space is then defined by the coordinates of the three airports: $(x_1, y_1)$, $(x_2, y_2)$, and $(x_3, y_3)$.

Variable

This is a *six-dimensional* space; we also say that states are defined by six **variables**. In general, states are defined by an $n$-dimensional vector of variables, $\mathbf{x}$. Moving around in this space corresponds to moving one or more of the airports on the map. The objective function $f(\mathbf{x}) = f(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state once we compute the closest cities. Let $C_i$ be the set of cities whose closest airport (in the state $\mathbf{x}$) is airport $i$. Then, we have

$$f(\mathbf{x}) = f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2 \,. \tag{4.1}$$

This equation is correct not only for the state $\mathbf{x}$ but also for states in the local neighborhood of $\mathbf{x}$. However, it is not correct globally; if we stray too far from $x$ (by altering the location of one or more of the airports by a large amount) then the set of closest cities for that airport changes, and we need to recompute $C_i$.

Discretization

One way to deal with a continuous state space is to **discretize** it. For example, instead of allowing the $(x_i, y_i)$ locations to be any point in continuous two-dimensional space, we could limit them to fixed points on a rectangular grid with spacing of size $\delta$ (delta). Then instead of having an infinite number of successors, each state in the space would have only 12 successors, corresponding to incrementing one of the 6 variables by $\pm\delta$. We can then apply any of our local search algorithms to this discrete space. Alternatively, we could make the branching factor finite by sampling successor states randomly, moving in a random direction by a small amount, $\delta$. Methods that measure progress by the change in the value of the objective function between two nearby points are called **empirical gradient** methods. Empirical gradient search is the same as steepest-ascent hill climbing in a discretized version of the state space. Reducing the value of $\delta$ over time can give a more accurate solution, but does not necessarily converge to a global optimum in the limit.

Empirical gradient

Often we have an objective function expressed in a mathematical form such that we can use calculus to solve the problem analytically rather than empirically. Many methods attempt to use the **gradient** of the landscape to find a maximum. The gradient of the objective function is a vector $\nabla f$ that gives the magnitude and direction of the steepest slope. For our problem, we have

Gradient

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

In some cases, we can find a maximum by solving the equation $\nabla f = 0$. (This could be done, for example, if we were placing just one airport; the solution is the arithmetic mean of all the cities' coordinates.) In many cases, however, this equation cannot be solved in closed form. For example, with three airports, the expression for the gradient depends on what cities are

[2] Knowledge of vectors, matrices, and derivatives is useful for this section (see Appendix A).

closest to each airport in the current state. This means we can compute the gradient *locally* (but not *globally*); for example,

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_1 - x_c) . \tag{4.2}$$

Given a locally correct expression for the gradient, we can perform steepest-ascent hill climbing by updating the current state according to the formula

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

where $\alpha$ (alpha) is a small constant often called the **step size**. There exist a huge variety of methods for adjusting $\alpha$. The basic problem is that if $\alpha$ is too small, too many steps are needed; if $\alpha$ is too large, the search could overshoot the maximum. The technique of **line search** tries to overcome this dilemma by extending the current gradient direction—usually by repeatedly doubling $\alpha$—until $f$ starts to decrease again. The point at which this occurs becomes the new current state. There are several schools of thought about how the new direction should be chosen at this point.

Step size

Line search

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

Newton–Raphson

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

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

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

where $\mathbf{H}_f(\mathbf{x})$ is the **Hessian** matrix of second derivatives, whose elements $H_{ij}$ are given by $\partial^2 f / \partial x_i \partial x_j$. For our airport example, we can see from Equation (4.2) that $\mathbf{H}_f(\mathbf{x})$ is particularly simple: the off-diagonal elements are zero and the diagonal elements for airport $i$ are just twice the number of cities in $C_i$. A moment's calculation shows that one step of the update moves airport $i$ directly to the centroid of $C_i$, which is the minimum of the local expression for $f$ from Equation (4.1).[3] For high-dimensional problems, however, computing the $n^2$ entries of the Hessian and inverting it may be expensive, so many approximate versions of the Newton–Raphson method have been developed.

Hessian

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

A final topic is **constrained optimization**. An optimization problem is constrained if solutions must satisfy some hard constraints on the values of the variables. For example, in our airport-siting problem, we might constrain sites to be inside Romania and on dry land (rather than in the middle of lakes). The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function. The best-known category is that of **linear programming** problems, in which constraints must be linear inequalities

Constrained optimization

Linear programming

[3] In general, the Newton–Raphson update can be seen as fitting a quadratic surface to $f$ at $\mathbf{x}$ and then moving directly to the minimum of that surface—which is also the minimum of $f$ if $f$ is quadratic.

Convex set

forming a **convex set**[4] and the objective function is also linear. The time complexity of linear programming is polynomial in the number of variables.

Convex optimization

Linear programming is probably the most widely studied and broadly useful method for optimization. It is a special case of the more general problem of **convex optimization**, which allows the constraint region to be any convex region and the objective to be any function that is convex within the constraint region. Under certain conditions, convex optimization problems are also polynomially solvable and may be feasible in practice with thousands of variables. Several important problems in machine learning and control theory can be formulated as convex optimization problems (see Chapter 20).

## 4.3 Search with Nondeterministic Actions

In Chapter 3, we assumed a fully observable, deterministic, known environment. Therefore, an agent can observe the initial state, calculate a sequence of actions that reach the goal, and execute the actions with its "eyes closed," never having to use its percepts.

When the environment is partially observable, however, the agent doesn't know for sure what state it is in; and when the environment is nondeterministic, the agent doesn't know what state it transitions to after taking an action. That means that rather than thinking "I'm in state $s_1$ and if I do action $a$ I'll end up in state $s_2$," an agent will now be thinking "I'm either in state $s_1$ or $s_3$, and if I do action $a$ I'll end up in state $s_2, s_4$ or $s_5$." We call a set of physical states that the agent believes are possible a **belief state**.

Belief state

Conditional plan

In partially observable and nondeterministic environments, the solution to a problem is no longer a sequence, but rather a **conditional plan** (sometimes called a contingency plan or a strategy) that specifies what to do depending on what percepts agent receives while executing the plan. We examine nondeterminism in this section and partial observability in the next.

### 4.3.1 The erratic vacuum world

The vacuum world from Chapter 2 has eight states, as shown in Figure 4.9. There are three actions—*Right*, *Left*, and *Suck*—and the goal is to clean up all the dirt (states 7 and 8). If the environment is fully observable, deterministic, and completely known, then the problem is easy to solve with any of the algorithms in Chapter 3, and the solution is an action sequence. For example, if the initial state is 1, then the action sequence [*Suck*, *Right*, *Suck*] will reach a goal state, 8.

Now suppose that we introduce nondeterminism in the form of a powerful but erratic vacuum cleaner. In the **erratic vacuum world**, the *Suck* action works as follows:

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
- When applied to a clean square the action sometimes deposits dirt on the carpet.[5]

To provide a precise formulation of this problem, we need to generalize the notion of a **transition model** from Chapter 3. Instead of defining the transition model by a RESULT function

[4] A set of points $\mathcal{S}$ is convex if the line joining any two points in $\mathcal{S}$ is also contained in $\mathcal{S}$. A **convex function** is one for which the space "above" it forms a convex set; by definition, convex functions have no local (as opposed to global) minima.

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

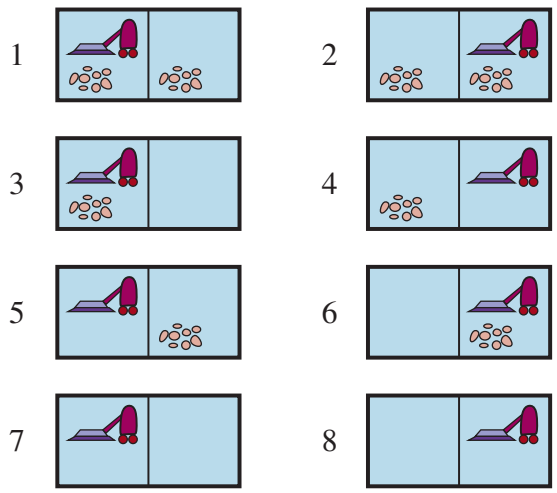



**Figure 4.9** The eight possible states of the vacuum world; states 7 and 8 are goal states.

that returns a single outcome state, we use a RESULTS function that returns a set of possible outcome states. For example, in the erratic vacuum world, the *Suck* action in state 1 cleans up either just the current location, or both locations:

$$\text{RESULTS}(1, \mathit{Suck}) = \{5, 7\}$$

If we start in state 1, no single *sequence* of actions solves the problem, but the following **conditional plan** does:

$$[\mathit{Suck}, \textbf{if } \mathit{State} = 5 \textbf{ then } [\mathit{Right}, \mathit{Suck}] \textbf{ else } [\,]\,]. \tag{4.3}$$

Here we see that a conditional plan can contain **if–then–else** steps; this means that solutions are *trees* rather than sequences. Here the conditional in the **if** statement tests to see what the current state is; this is something the agent will be able to observe at runtime, but doesn't know at planning time. Alternatively, we could have had a formulation that tests the percept rather than the state. Many problems in the real, physical world are contingency problems, because exact prediction of the future is impossible. For this reason, many people keep their eyes open while walking around.

### 4.3.2 AND–OR search trees

How do we find these contingent solutions to nondeterministic problems? As in Chapter 3, we begin by constructing search trees, but here the trees have a different character. In a deterministic environment, the only branching is introduced by the agent's own choices in each state: I can do this action or that action. We call these nodes **OR nodes**. In the vacuum world, for example, at an OR node the agent chooses *Left or Right or Suck*. In a nondeterministic environment, branching is also introduced by the *environment's* choice of outcome for each action. We call these nodes **AND nodes**. For example, the *Suck* action in state 1 results in the belief state $\{5, 7\}$, so the agent would need to find a plan for state 5 *and* for state 7. These two kinds of nodes alternate, leading to an **AND–OR tree** as illustrated in Figure 4.10.

Or node

And node

And–or tree

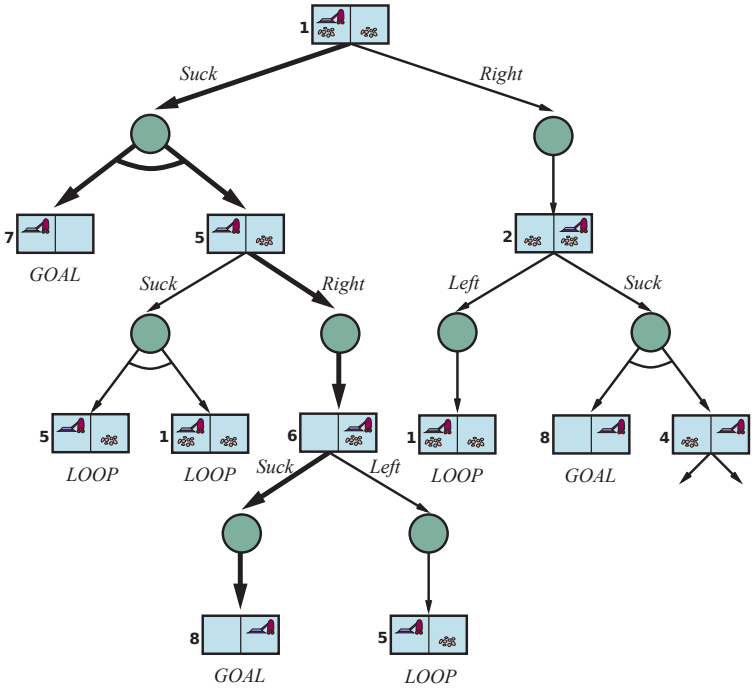



**Figure 4.10** The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

A solution for an AND–OR search problem is a subtree of the complete search tree that (1) has a goal node at every leaf, (2) specifies one action at each of its OR nodes, and (3) includes every outcome branch at each of its AND nodes. The solution is shown in bold lines in the figure; it corresponds to the plan given in Equation (4.3).

Figure 4.11 gives a recursive, depth-first algorithm for AND–OR graph search. One key aspect of the algorithm is the way in which it deals with cycles, which often arise in nondeterministic problems (e.g., if an action sometimes has no effect or if an unintended effect can be corrected). If the current state is identical to a state on the path from the root, then it returns with failure. This doesn't mean that there is *no* solution from the current state; it simply means that if there *is* a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded. With this check, we ensure that the algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state. Notice that the algorithm does not check whether the current state is a repetition of a state on some *other* path from the root, which is important for efficiency.

AND–OR graphs can be explored either breadth-first or best-first. The concept of a heuristic function must be modified to estimate the cost of a contingent solution rather than a sequence, but the notion of admissibility carries over and there is an analog of the A* algorithm for finding optimal solutions. (See the bibliographical notes at the end of the chapter.)

```
function AND-OR-SEARCH(problem) returns a conditional plan, or failure
  return OR-SEARCH(problem, problem.INITIAL, [])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
  if problem.IS-GOAL(state) then return the empty plan
  if IS-CYCLE(path) then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
      if plan ≠ failure then return [action] + plan]
  return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(problem, s_i, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

**Figure 4.11** An algorithm for searching AND–OR graphs generated by nondeterministic environments. A solution is a conditional plan that considers every nondeterministic outcome and makes a plan for each one.

### 4.3.3 Try, try again

Consider a *slippery* vacuum world, which is identical to the ordinary (non-erratic) vacuum world except that movement actions sometimes fail, leaving the agent in the same location. For example, moving *Right* in state 1 leads to the belief state $\{1,2\}$. Figure 4.12 shows part of the search graph; clearly, there are no longer any acyclic solutions from state 1, and AND-OR-SEARCH would return with failure. There is, however, a **cyclic solution**, which is to keep trying *Right* until it works. We can express this with a new **while** construct:

Cyclic solution

$$[Suck, \textbf{while } State = 5 \textbf{ do } Right, Suck]$$

or by adding a **label** to denote some portion of the plan and referring to that label later:

$$[Suck, L_1 : Right, \textbf{if } State = 5 \textbf{ then } L_1 \textbf{ else } Suck].$$

When is a cyclic plan a solution? A minimum condition is that every leaf is a goal state and that a leaf is reachable from every point in the plan. In addition to that, we need to consider the cause of the nondeterminism. If it is really the case that the vacuum robot's drive mechanism works some of the time, but randomly and independently slips on other occasions, then the agent can be confident that if the action is repeated enough times, eventually it will work and the plan will succeed. But if the nondeterminism is due to some unobserved fact about the robot or environment—perhaps a drive belt has snapped and the robot will never move—then repeating the action will not help.

One way to understand this decision is to say that the initial problem formulation (fully observable, nondeterministic) is abandoned in favor of a different formulation (partially observable, deterministic) where the failure of the cyclic plan is attributed to an unobserved property of the drive belt. In Chapter 12 we discuss how to decide which of several uncertain possibilities is more likely.

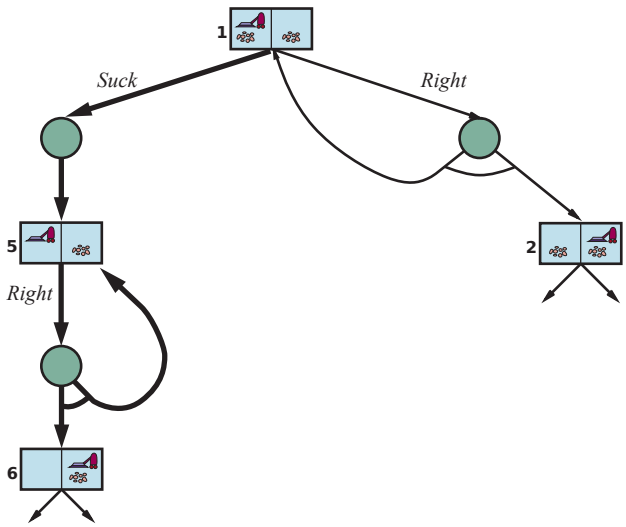



**Figure 4.12** Part of the search graph for a slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

## 4.4 Search in Partially Observable Environments

We now turn to the problem of partial observability, where the agent's percepts are not enough to pin down the exact state. That means that some of the agent's actions will be aimed at reducing uncertainty about the current state.

### 4.4.1 Searching with no observation

Sensorless
Conformant

When the agent's percepts provide *no information at all*, we have what is called a **sensorless** problem (or a **conformant** problem). At first, you might think the sensorless agent has no hope of solving a problem if it has no idea what state it starts in, but sensorless solutions are surprisingly common and useful, primarily because they *don't* rely on sensors working properly. In manufacturing systems, for example, many ingenious methods have been developed for orienting parts correctly from an unknown initial position by using a sequence of actions with no sensing at all. Sometimes a sensorless plan is better even when a conditional plan with sensing is available. For example, doctors often prescribe a broad-spectrum antibiotic rather than using the conditional plan of doing a blood test, then waiting for the results to come back, and then prescribing a more specific antibiotic. The sensorless plan saves time and money, and avoids the risk of the infection worsening before the test results are available.

Consider a sensorless version of the (deterministic) vacuum world. Assume that the agent knows the geography of its world, but not its own location or the distribution of dirt. In that case, its initial belief state is $\{1,2,3,4,5,6,7,8\}$ (see Figure 4.9). Now, if the agent moves *Right* it will be in one of the states $\{2,4,6,8\}$—the agent has gained information without perceiving anything! After [*Right*,*Suck*] the agent will always end up in one of the states $\{4,8\}$. Finally, after [*Right*,*Suck*,*Left*,*Suck*] the agent is guaranteed to reach the goal state 7, no matter what the start state. We say that the agent can **coerce** the world into state 7.

Coercion

The solution to a sensorless problem is a sequence of actions, not a conditional plan (because there is no perceiving). But we search in the space of belief states rather than physical states.[6] In belief-state space, the problem is *fully observable* because the agent always knows its own belief state. Furthermore, the solution (if any) for a sensorless problem is always a sequence of actions. This is because, as in the ordinary problems of Chapter 3, the percepts received after each action are completely predictable—they're always empty! So there are no contingencies to plan for. This is true *even if the environment is nondeterministic*.

We could introduce new algorithms for sensorless search problems. But instead, we can use the existing algorithms from Chapter 3 if we transform the underlying physical problem into a belief-state problem, in which we search over belief states rather than physical states. The original problem, $P$, has components $Actions_P, Result_P$ etc., and the belief-state problem has the following components:

- **States**: The belief-state space contains every possible subset of the physical states. If $P$ has $N$ states, then the belief-state problem has $2^N$ belief states, although many of those may be unreachable from the initial state.
- **Initial state**: Typically the belief state consisting of all states in $P$, although in some cases the agent will have more knowledge than this.
- **Actions**: This is slightly tricky. Suppose the agent is in belief state $b = \{s_1, s_2\}$, but $\text{ACTIONS}_P(s_1) \neq \text{ACTIONS}_P(s_2)$; then the agent is unsure of which actions are legal. If we assume that illegal actions have no effect on the environment, then it is safe to take the *union* of all the actions in any of the physical states in the current belief state $b$:

  $$\text{ACTIONS}(b) = \bigcup_{s \in b} \text{ACTIONS}_P(s) .$$

  On the other hand, if an illegal action might lead to catastrophe, it is safer to allow only the *intersection*, that is, the set of actions legal in *all* the states. For the vacuum world, every state has the same legal actions, so both methods give the same result.
- **Transition model**: For deterministic actions, the new belief state has one result state for each of the current possible states (although some result states may be the same):

  $$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\} . \tag{4.4}$$

  With nondeterminism, the new belief state consists of all the possible results of applying the action to any of the states in the current belief state:

  $$\begin{aligned} b' = \text{RESULT}(b, a) &= \{s' : s' \in \text{RESULTS}_P(s, a) \text{ and } s \in b\} \\ &= \bigcup_{s \in b} \text{RESULTS}_P(s, a) , \end{aligned}$$

  The size of $b'$ will be the same or smaller than $b$ for deterministic actions, but may be larger than $b$ with nondeterministic actions (see Figure 4.13).
- **Goal test**: The agent *possibly* achieves the goal if *any* state $s$ in the belief state satisfies the goal test of the underlying problem, $\text{IS-GOAL}_P(s)$. The agent *necessarily* achieves the goal if *every* state satisfies $\text{IS-GOAL}_P(s)$. We aim to necessarily achieve the goal.
- **Action cost**: This is also tricky. If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of

[6] In a fully observable environment, each belief state contains one physical state. Thus, we can view the algorithms in Chapter 3 as searching in a belief-state space of singleton belief states.

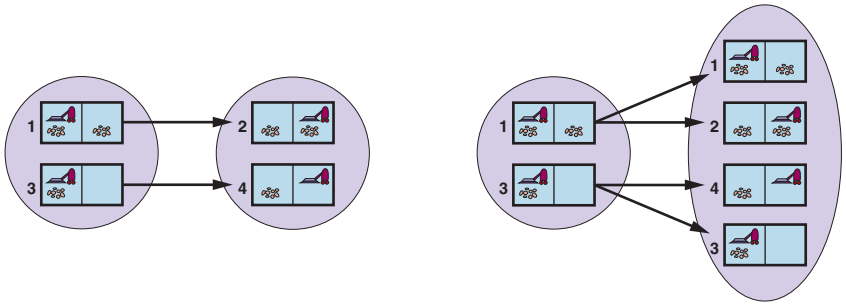



**Figure 4.13** (a) Predicting the next belief state for the sensorless vacuum world with the deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

several values. (This gives rise to a new class of problems, which we explore in Exercise 4.MVAL.) For now we assume that the cost of an action is the same in all states and so can be transferred directly from the underlying physical problem.

Figure 4.14 shows the reachable belief-state space for the deterministic, sensorless vacuum world. There are only 12 reachable belief states out of $2^8 = 256$ possible belief states.

The preceding definitions enable the automatic construction of the belief-state problem formulation from the definition of the underlying physical problem. Once this is done, we can solve sensorless problems with any of the ordinary search algorithms of Chapter 3.

In ordinary graph search, newly reached states are tested to see if they were previously reached. This works for belief states, too; for example, in Figure 4.14, the action sequence [*Suck*,*Left*,*Suck*] starting at the initial state reaches the same belief state as [*Right*,*Left*,*Suck*], namely, $\{5,7\}$. Now, consider the belief state reached by [*Left*], namely, $\{1,3,5,7\}$. Obviously, this is not identical to $\{5,7\}$, but it is a *superset*. We can discard (prune) any such superset belief state. Why? Because a solution from $\{1,3,5,7\}$ must be a solution for each of the individual states 1, 3, 5, and 7, and thus it is a solution for any combination of these individual states, such as $\{5,7\}$; therefore we don't need to try to solve $\{1,3,5,7\}$, we can concentrate on trying to solve the strictly easier belief state $\{5,7\}$.

Conversely, if $\{1,3,5,7\}$ has already been generated and found to be solvable, then any *subset*, such as $\{5,7\}$, is guaranteed to be solvable. (If I have a solution that works when I'm very confused about what state I'm in, it will still work when I'm less confused.) This extra level of pruning may dramatically improve the efficiency of sensorless problem solving.

Even with this improvement, however, sensorless problem-solving as we have described it is seldom feasible in practice. One issue is the vastness of the belief-state space—we saw in the previous chapter that often a search space of size $N$ is too large, and now we have search spaces of size $2^N$. Furthermore, each element of the search space is a set of up to $N$ elements. For large $N$, we won't be able to represent even a single belief state without running out of memory space.

One solution is to represent the belief state by some more compact description. In English, we could say the agent knows "Nothing" in the initial state; after moving *Left*, we could

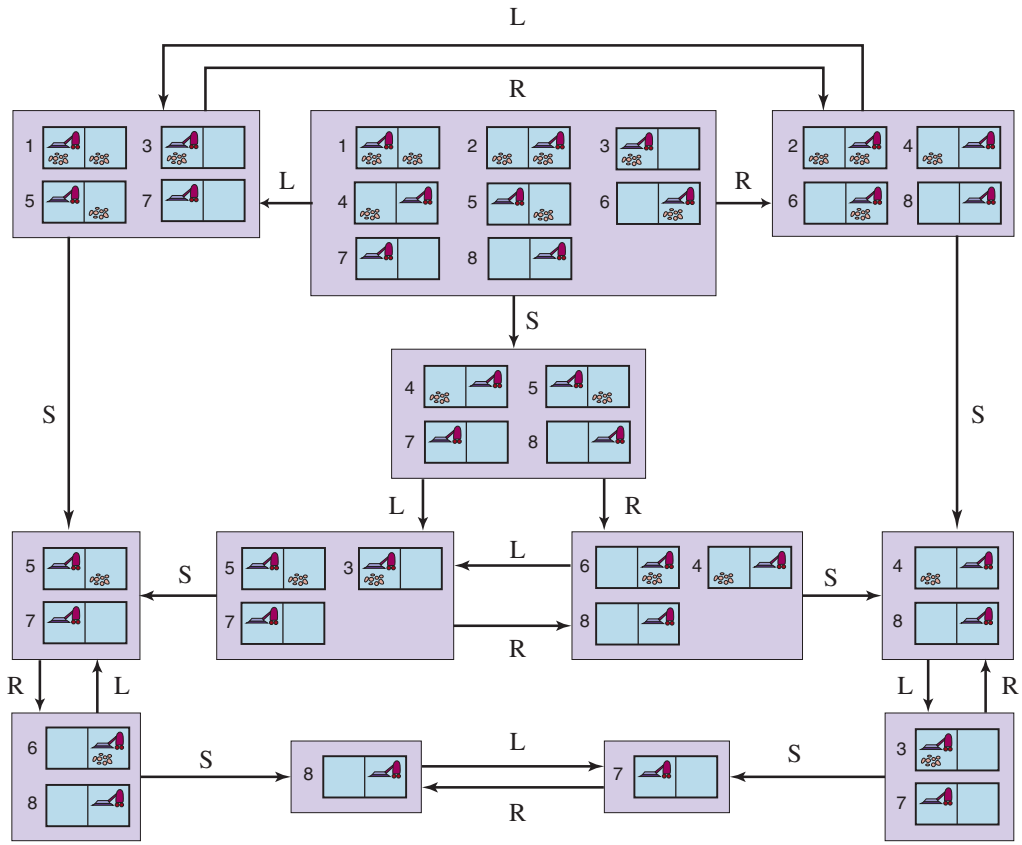



**Figure 4.14** The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each rectangular box corresponds to a single belief state. At any given point, the agent has a belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box.

say, "Not in the rightmost column," and so on. Chapter 7 explains how to do this in a formal representation scheme.

Another approach is to avoid the standard search algorithms, which treat belief states as black boxes just like any other problem state. Instead, we can look *inside* the belief states and develop **incremental belief-state search** algorithms that build up the solution one physical state at a time. For example, in the sensorless vacuum world, the initial belief state is $\{1,2,3,4,5,6,7,8\}$, and we have to find an action sequence that works in all 8 states. We can do this by first finding a solution that works for state 1; then we check if it works for state 2; if not, go back and find a different solution for state 1, and so on.

Incremental belief-state search

Just as an AND–OR search has to find a solution for every branch at an AND node, this algorithm has to find a solution for every state in the belief state; the difference is that AND–OR search can find a different solution for each branch, whereas an incremental belief-state search has to find *one* solution that works for *all* the states.

The main advantage of the incremental approach is that it is typically able to detect failure quickly—when a belief state is unsolvable, it is usually the case that a small subset of the

belief state, consisting of the first few states examined, is also unsolvable. In some cases, this leads to a speedup proportional to the size of the belief states, which may themselves be as large as the physical state space itself.

### 4.4.2 Searching in partially observable environments

Many problems cannot be solved without sensing. For example, the sensorless 8-puzzle is impossible. On the other hand, a little bit of sensing can go a long way: we can solve 8-puzzles if we can see just the upper-left corner square. The solution involves moving each tile in turn into the observable square and keeping track of its location from then on .

For a partially observable problem, the problem specification will specify a PERCEPT$(s)$ function that returns the percept received by the agent in a given state. If sensing is nondeterministic, then we can use a PERCEPTS function that returns a set of possible percepts. For fully observable problems, PERCEPT$(s) = s$ for every state $s$, and for sensorless problems PERCEPT$(s) = null$.

Consider a local-sensing vacuum world, in which the agent has a position sensor that yields the percept $L$ in the left square, and $R$ in the right square, and a dirt sensor that yields *Dirty* when the current square is dirty and *Clean* when it is clean. Thus, the PERCEPT in state 1 is $[L, Dirty]$. With partial observability, it will usually be the case that several states produce the same percept; state 3 will also produce $[L, Dirty]$. Hence, given this initial percept, the initial belief state will be $\{1, 3\}$. We can think of the transition model between belief states for partially observable problems as occurring in three stages, as shown in Figure 4.15:

- The **prediction** stage computes the belief state resulting from the action, RESULT$(b, a)$, exactly as we did with sensorless problems. To emphasize that this is a prediction, we use the notation $\hat{b} = \text{RESULT}(b, a)$, where the "hat" over the $b$ means "estimated," and we also use PREDICT$(b, a)$ as a synonym for RESULT$(b, a)$.
- The **possible percepts** stage computes the set of percepts that could be observed in the predicted belief state (using the letter $o$ for observation):

$$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}.$$

- The **update** stage computes, for each possible percept, the belief state that would result from the percept. The updated belief state $b_o$ is the set of states in $\hat{b}$ that could have produced the percept:

$$b_o = \text{UPDATE}(\hat{b}, o) = \{s : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}.$$

  The agent needs to deal with *possible* percepts at planning time, because it won't know the *actual* percepts until it executes the plan. Notice that nondeterminism in the physical environment can enlarge the belief state in the prediction stage, but each updated belief state $b_o$ can be no larger than the predicted belief state $\hat{b}$; observations can only help reduce uncertainty. Moreover, for deterministic sensing, the belief states for the different possible percepts will be disjoint, forming a *partition* of the original predicted belief state.

Putting these three stages together, we obtain the possible belief states resulting from a given action and the subsequent possible percepts:

$$\begin{aligned} \text{RESULTS}(b, a) = \{b_o : b_o &= \text{UPDATE}(\text{PREDICT}(b, a), o) \text{ and} \\ o &\in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\}. \end{aligned} \tag{4.5}$$

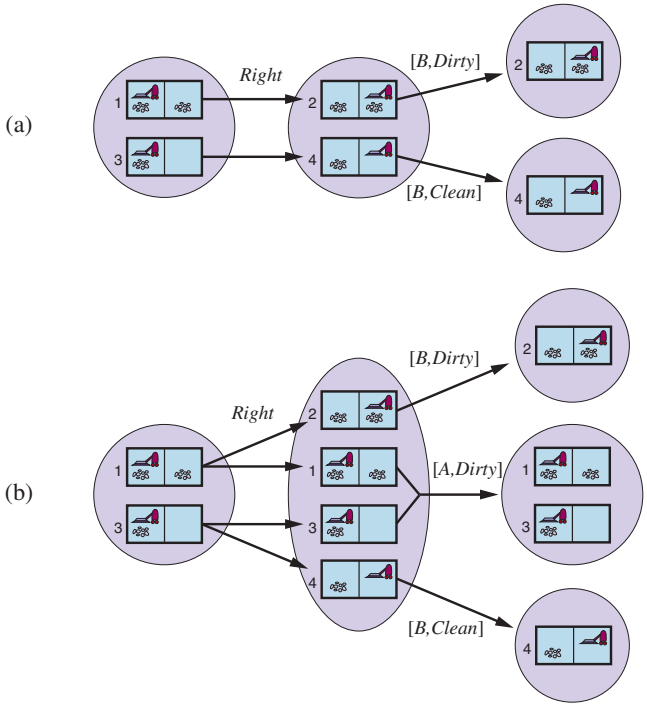



**Figure 4.15** Two examples of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new predicted belief state with two possible physical states; for those states, the possible percepts are $[R, Dirty]$ and $[R, Clean]$, leading to two belief states, each of which is a singleton. (b) In the slippery world, *Right* is applied in the initial belief state, giving a new belief state with four physical states; for those states, the possible percepts are $[L, Dirty]$, $[R, Dirty]$, and $[R, Clean]$, leading to three belief states as shown.

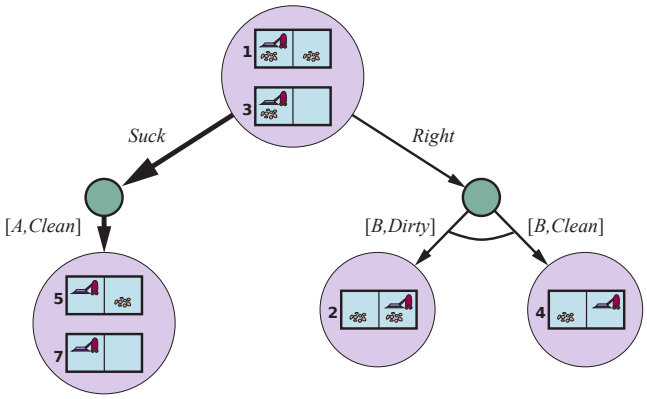



**Figure 4.16** The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first action in the solution.

### 4.4.3 Solving partially observable problems

The preceding section showed how to derive the RESULTS function for a nondeterministic belief-state problem from an underlying physical problem, given the PERCEPT function. With this formulation, the AND–OR search algorithm of Figure 4.11 can be applied directly to derive a solution. Figure 4.16 shows part of the search tree for the local-sensing vacuum world, assuming an initial percept $[A, Dirty]$. The solution is the conditional plan

$$[Suck, Right, \textbf{if } Bstate = \{6\} \textbf{ then } Suck \textbf{ else } [\,]\,].$$

Notice that, because we supplied a belief-state problem to the AND–OR search algorithm, it returned a conditional plan that tests the belief state rather than the actual state. This is as it should be: in a partially observable environment the agent won't know the actual state.

As in the case of standard search algorithms applied to sensorless problems, the AND–OR search algorithm treats belief states as black boxes, just like any other states. One can improve on this by checking for previously generated belief states that are subsets or supersets of the current state, just as for sensorless problems. One can also derive incremental search algorithms, analogous to those described for sensorless problems, that provide substantial speedups over the black-box approach.

### 4.4.4 An agent for partially observable environments

An agent for partially observable environments formulates a problem, calls a search algorithm (such as AND-OR-SEARCH) to solve it, and executes the solution. There are two main differences between this agent and the one for fully observable deterministic environments. First, the solution will be a conditional plan rather than a sequence; to execute an if–then–else expression, the agent will need to test the condition and execute the appropriate branch of the conditional. Second, the agent will need to maintain its belief state as it performs actions and receives percepts. This process resembles the prediction–observation–update process in Equation (4.5) but is actually simpler because the percept is given by the environment rather than calculated by the agent. Given an initial belief state $b$, an action $a$, and a percept $o$, the new belief state is:

$$b' = \text{UPDATE}(\text{PREDICT}(b, a), o)\,. \tag{4.6}$$

Consider a *kindergarten* vacuum world wherein agents sense only the state of their current square, and any square may become dirty at any time unless the agent is actively cleaning it at that moment.[7] Figure 4.17 shows the belief state being maintained in this environment.

Monitoring
Filtering
State estimation

In partially observable environments—which include the vast majority of real-world environments—maintaining one's belief state is a core function of any intelligent system. This function goes under various names, including **monitoring**, **filtering**, and **state estimation**. Equation (4.6) is called a recursive state estimator because it computes the new belief state from the previous one rather than by examining the entire percept sequence. If the agent is not to "fall behind," the computation has to happen as fast as percepts are coming in. As the environment becomes more complex, the agent will only have time to compute an approximate belief state, perhaps focusing on the implications of the percept for the aspects of the environment that are of current interest. Most work on this problem has been done for

[7] The usual apologies to those who are unfamiliar with the effect of small children on the environment.

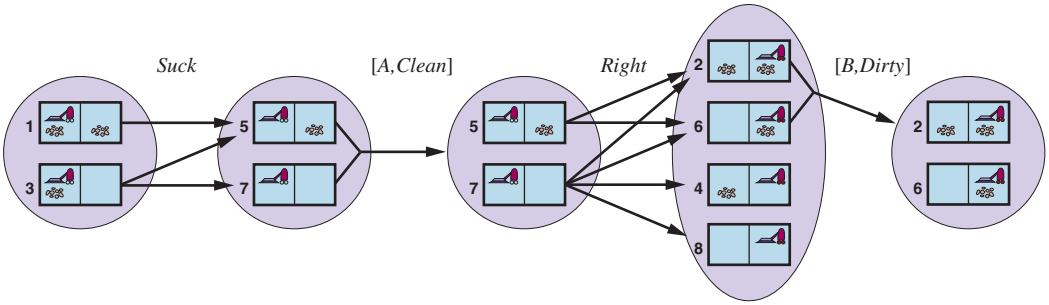



**Figure 4.17** Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

stochastic, continuous-state environments with the tools of probability theory, as explained in Chapter 14.

In this section we will show an example in a discrete environment with deterministic sensors and nondeterministic actions. The example concerns a robot with a particular state estimation task called **localization**: working out where it is, given a map of the world and a sequence of percepts and actions. Our robot is placed in the maze-like environment of Figure 4.18. The robot is equipped with four sonar sensors that tell whether there is an obstacle—the outer wall or a dark shaded square in the figure—in each of the four compass directions. The percept is in the form of a bit vector, one bit for each of the directions north, east, south, and west in that order, so 1011 means there are obstacles to the north, south, and west, but not east.

Localization

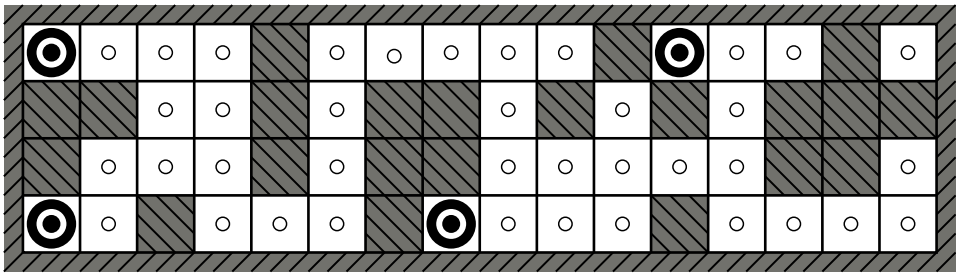
We assume that the sensors give perfectly correct data, and that the robot has a correct map of the environment. But unfortunately, the robot's navigational system is broken, so when it executes a *Right* action, it moves randomly to one of the adjacent squares. The robot's task is to determine its current location.

Suppose the robot has just been switched on, and it does not know where it is—its initial belief state $b$ consists of the set of all locations. The robot then receives the percept 1011 and does an update using the equation $b_o = \text{UPDATE}(1011)$, yielding the 4 locations shown in Figure 4.18(a). You can inspect the maze to see that those are the only four locations that yield the percept 1011.

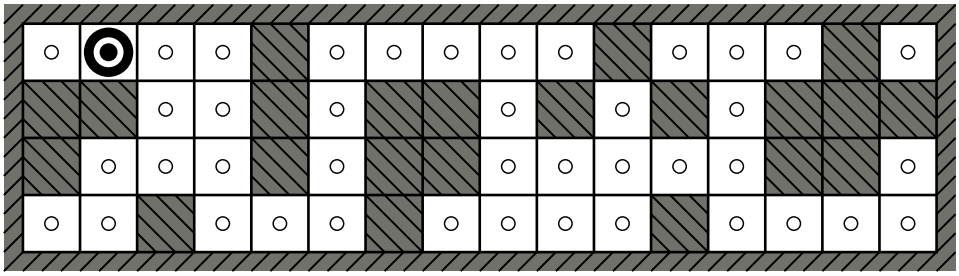
Next the robot executes a *Right* action, but the result is nondeterministic. The new belief state, $b_a = \text{PREDICT}(b_o, Right)$, contains all the locations that are one step away from the locations in $b_o$. When the second percept, 1010, arrives, the robot does $\text{UPDATE}(b_a, 1010)$ and finds that the belief state has collapsed down to the single location shown in Figure 4.18(b). That's the only location that could be the result of

$$\text{UPDATE}(\text{PREDICT}(\text{UPDATE}(b, 1011), Right), 1010)\,.$$

With nondeterministic actions the PREDICT step grows the belief state, but the UPDATE step shrinks it back down—as long as the percepts provide some useful identifying information. Sometimes the percepts don't help much for localization: If there were one or more long east-west corridors, then a robot could receive a long sequence of 1010 percepts, but never know



(a) Possible locations of robot after $E_1 = 1011$



(b) Possible locations of robot after $E_1 = 1011, E_2 = 1010$

**Figure 4.18** Possible positions of the robot, ⊙, (a) after one observation, $E_1 = 1011$, and (b) after moving one square and making a second observation, $E_2 = 1010$. When sensors are noiseless and the transition model is accurate, there is only one possible location for the robot consistent with this sequence of two observations.

where in the corridor(s) it was. But for environments with reasonable variation in geography, localization often converges quickly to a single point, even when actions are nondeterministic.

What happens if the sensors are faulty? If we can reason only with Boolean logic, then we have to treat every sensor bit as being either correct or incorrect, which is the same as having no perceptual information at all. But we will see that probabilistic reasoning (Chapter 12), allows us to extract useful information from a faulty sensor as long as it is wrong less than half the time.

## 4.5 Online Search Agents and Unknown Environments

Offline search
Online search

So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution before taking their first action. In contrast, an **online search**[8] agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi-dynamic environments, where there is a penalty for sitting around and computing too long. Online

[8] The term "online" here refers to algorithms that must process input as it is received rather than waiting for the entire input data set to become available. This usage of "online" is unrelated to the concept of "having an Internet connection."

search is also helpful in nondeterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that *might* happen but probably won't.

Of course, there is a tradeoff: the more an agent plans ahead, the less often it will find itself up the creek without a paddle. In unknown environments, where the agent does not know what states exist or what its actions do, the agent must use its actions as experiments in order to learn about the environment.

A canonical example of online search is the **mapping problem**: a robot is placed in an unknown building and must explore to build a map that can later be used for getting from $A$ to $B$. Methods for escaping from labyrinths—required knowledge for aspiring heroes of antiquity—are also examples of online search algorithms. Spatial exploration is not the only form of online exploration, however. Consider a newborn baby: it has many possible actions but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach.

Mapping problem

### 4.5.1 Online search problems

An online search problem is solved by interleaving computation, sensing, and acting. We'll start by assuming a deterministic and fully observable environment (Chapter 17 relaxes these assumptions) and stipulate that the agent knows only the following:

- ACTIONS$(s)$, the legal actions in state $s$;
- $c(s,a,s')$, the cost of applying action $a$ in state $s$ to arrive at state $s'$. Note that this cannot be used until the agent knows that $s'$ is the outcome.
- IS-GOAL$(s)$, the goal test.

Note in particular that the agent *cannot* determine RESULT$(s,a)$ except by actually being in $s$ and doing $a$. For example, in the maze problem shown in Figure 4.19, the agent does not know that going *Up* from (1,1) leads to (1,2); nor, having done that, does it know that going *Down* will take it back to (1,1). This degree of ignorance can be reduced in some applications—for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state. For example, in Figure 4.19, the agent might know the location of the goal and be able to use the Manhattan-distance heuristic (page 97).

Typically, the agent's objective is to reach a goal state while minimizing cost. (Another possible objective is simply to explore the entire environment.) The cost is the total path cost that the agent incurs as it travels. It is common to compare this cost with the path cost the agent would incur *if it knew the search space in advance*—that is, the optimal path in the known environment. In the language of online algorithms, this comparison is called the **competitive ratio**; we would like it to be as small as possible.

Competitive ratio

Online explorers are vulnerable to **dead ends**: states from which no goal state is reachable. If the agent doesn't know what each action does, it might execute the "jump into bottomless pit" action, and thus never reach the goal. In general, *no algorithm can avoid dead ends in all state spaces.* Consider the two dead-end state spaces in Figure 4.20(a). An online search algorithm that has visited states $S$ and $A$ cannot tell if it is in the top state or the bottom one; the two look identical based on what the agent has seen. Therefore, there is no

Dead end

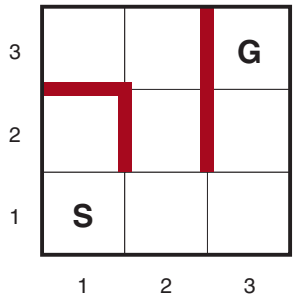



**Figure 4.19** A simple maze problem. The agent starts at *S* and must reach *G* but knows nothing of the environment.

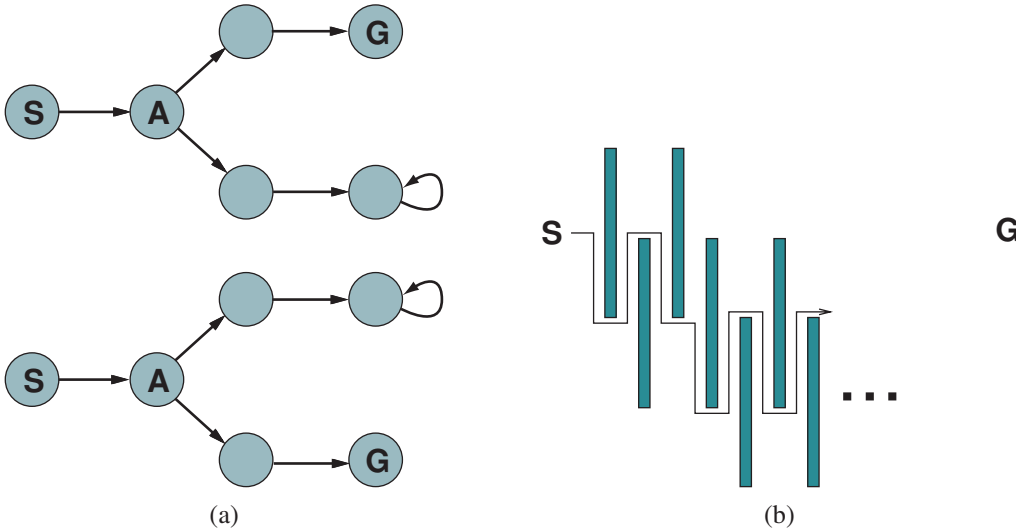



**Figure 4.20** (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

Adversary argument

way it could know how to choose the correct action in both state spaces. This is an example of an **adversary argument**—we can imagine an adversary constructing the state space while the agent explores it and putting the goals and dead ends wherever it chooses, as in Figure 4.20(b).

Irreversible action

Safely explorable

Dead ends are a real difficulty for robot exploration—staircases, ramps, cliffs, one-way streets, and even natural terrain all present states from which some actions are **irreversible**—there is no way to return to the previous state. The exploration algorithm we will present is only guaranteed to work in state spaces that are **safely explorable**—that is, some goal state is reachable from every reachable state. State spaces with only reversible actions, such as

```
function ONLINE-DFS-AGENT(problem, s′) returns an action
                s, a, the previous state and action, initially null
  persistent: result, a table mapping (s, a) to s′, initially empty
                untried, a table mapping s to a list of untried actions
                unbacktracked, a table mapping s to a list of states never backtracked to

  if problem.IS-GOAL(s′) then return stop
  if s′ is a new state (not in untried) then untried[s′] ← problem.ACTIONS(s′)
  if s is not null then
      result[s, a] ← s′
      add s to the front of unbacktracked[s′]
  if untried[s′] is empty then
      if unbacktracked[s′] is empty then return stop
      else a ← an action b such that result[s′, b] = POP(unbacktracked[s′])
  else a ← POP(untried[s′])
  s ← s′
  return a
```

**Figure 4.21** An online search agent that uses depth-first exploration. The agent can safely explore only in state spaces in which every action can be "undone" by some other action.

mazes and 8-puzzles, are clearly safely explorable (if they have any solution at all). We will cover the subject of safe exploration in more depth in Section 22.3.2.

Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. This is easy to show in environments with irreversible actions, but in fact it remains true for the reversible case as well, as Figure 4.20(b) shows. For this reason, it is common to characterize the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

### 4.5.2 Online search agents

After each action, an online agent in an observable environment receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The updated map is then used to plan where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously: offline algorithms explore their *model* of the state space, while online algorithms explore the real world. For example, A* can expand a node in one part of the space and then immediately expand a node in a distant part of the space, because node expansion involves simulated rather than real actions.

An online algorithm, on the other hand, can discover successors only for a state that it physically occupies. To avoid traveling all the way to a distant state to expand the next node, it seems better to expand nodes in a *local* order. Depth-first search has exactly this property because (except when the algorithm is backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first exploration agent (for deterministic but unknown actions) is shown in Figure 4.21. This agent stores its map in a table, $result[s, a]$, that records the state resulting from executing action $a$ in state $s$. (For nondeterministic actions, the agent could record a set

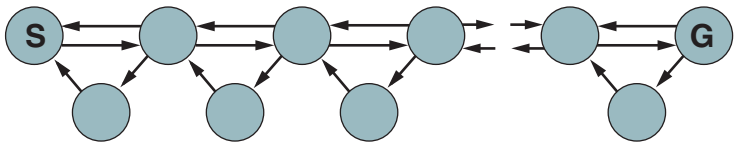



**Figure 4.22** An environment in which a random walk will take exponentially many steps to find the goal.

of states under *results*$[s, a]$.) Whenever the current state has unexplored actions, the agent tries one of those actions. The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack in the physical world. In depth-first search, this means going back to the state from which the agent most recently entered the current state. To achieve that, the algorithm keeps another table that lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

We recommend that the reader trace through the progress of ONLINE-DFS-AGENT when applied to the maze given in Figure 4.19. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

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

### 4.5.3 Online local search

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

Random walk

Instead of random restarts, one might consider using a **random walk** to explore the environment. A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will *eventually* find a goal or complete its exploration, provided that the space is finite and safely explorable.[9] On the other hand, the process can be very slow. Figure 4.22 shows an environment in which a random walk will take exponentially many steps to find the goal, because, for each state in the top row except S, backward progress is twice as likely as forward progress. The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.

[9] Random walks are complete on infinite one-dimensional and two-dimensional grids. On a three-dimensional grid, the probability that the walk ever returns to the starting point is only about 0.3405 (Hughes, 1995).

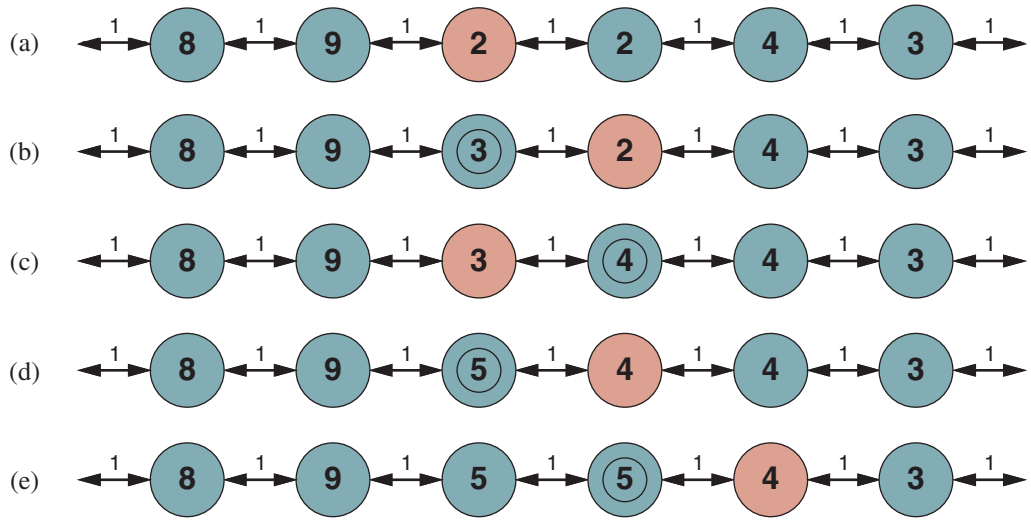



**Figure 4.23** Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and every link has an action cost of 1. The red state marks the location of the agent, and the updated cost estimates at each iteration have a double circle.

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

Figure 4.23 shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the red state. Rather than staying where it is, the agent should follow what seems to be the best path to the goal given the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor $s'$ is the cost to get to $s'$ plus the estimated cost to get to a goal from there—that is, $c(s,a,s')+H(s')$. In the example, there are two actions, with estimated costs $1+9$ to the left and $1+2$ to the right, so it seems best to move right.

In (b) it is clear that the cost estimate of 2 for the red state in (a) was overly optimistic. Since the best move cost 1 and led to a state that is at least 2 steps from a goal, the red state must be at least 3 steps from a goal, so its $H$ should be updated accordingly, as shown in Figure 4.23(b). Continuing this process, the agent will move back and forth twice more, updating $H$ each time and "flattening out" the local minimum until it escapes to the right.

LRTA*

An agent implementing this scheme, which is called learning real-time A* (**LRTA***), is shown in Figure 4.24. Like ONLINE-DFS-AGENT, it builds a map of the environment in the *result* table. It updates the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state $s$ are always assumed to lead immediately to the goal with the least possible cost, namely $h(s)$. This **optimism under uncertainty** encourages the agent to explore new, possibly promising paths.

Optimism under uncertainty

```
function LRTA*-AGENT(problem, s', h) returns an action
                s, a, the previous state and action, initially null
  persistent: result, a table mapping (s, a) to s', initially empty
                H, a table mapping s to a cost estimate, initially empty

  if IS-GOAL(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  if s is not null then
      result[s, a] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[s, b], H)
  a ← argmin_{b ∈ ACTIONS(s)} LRTA*-COST(problem, s', b, result[s', b], H)
  s ← s'
  return a

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

**Figure 4.24** LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

An LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment. Unlike A*, however, it is not complete for infinite state spaces—there are cases where it can be led infinitely astray. It can explore an environment of $n$ states in $O(n^2)$ steps in the worst case, but often does much better. The LRTA* agent is just one of a large family of online agents that one can define by specifying the action selection rule and the update rule in different ways. We discuss this family, developed originally for stochastic environments, in Chapter 22.

### 4.5.4 Learning in online search

The initial ignorance of online search agents provides several opportunities for learning. First, the agents learn a "map" of the environment—more precisely, the outcome of each action in each state—simply by recording each of their experiences. Second, the local search agents acquire more accurate estimates of the cost of each state by using local updating rules, as in LRTA*. In Chapter 22, we show that these updates eventually converge to *exact* values for every state, provided that the agent explores the state space in the right way. Once exact values are known, optimal decisions can be taken simply by moving to the lowest-cost successor—that is, pure hill climbing is then an optimal strategy.

If you followed our suggestion to trace the behavior of ONLINE-DFS-AGENT in the environment of Figure 4.19, you will have noticed that the agent is not very bright. For example, after it has seen that the *Up* action goes from (1,1) to (1,2), the agent still has no idea that the *Down* action goes back to (1,1) or that the *Up* action also goes from (2,1) to (2,2), from (2,2) to (2,3), and so on. In general, we would like the agent to learn that *Up* increases the $y$-coordinate unless there is a wall in the way, that *Down* reduces it, and so on.

For this to happen, we need two things. First, we need a formal and explicitly manipulable representation for these kinds of general rules; so far, we have hidden the information inside

the black box called the RESULT function. Chapters 8 to 11 are devoted to this issue. Second, we need algorithms that can construct suitable general rules from the specific observations made by the agent. These are covered in Chapter 19.

If we anticipate that we will be called upon to solve multiple similar problems in the future then it makes sense to invest time (and memory) to make those future searches easier. There are several ways to do this, all falling under the heading of **incremental search**. We could keep the search tree in memory and reuse the parts of it that are unchanged in the new problem. We could keep the heuristic $h$ values and update them as we gain new information—either because the world has changed or because we have computed a better estimate. Or we could keep the best-path $g$ values, using them to piece together a new solution, and updating them when the world changes.

Incremental search

## Summary

This chapter has examined search algorithms for problems in partially observable, nondeterministic, unknown, and continuous environments.

- *Local search* methods such as **hill climbing** keep only a small number of states in memory. They have been applied to optimization problems, where the idea is to find a high-scoring state, without worrying about the path to the state. Several stochastic local search algorithms have been developed, including **simulated annealing**, which returns optimal solutions when given an appropriate cooling schedule.
- Many local search methods apply also to problems in continuous spaces. **Linear programming** and **convex optimization** problems obey certain restrictions on the shape of the state space and the nature of the objective function, and admit polynomial-time algorithms that are often extremely efficient in practice. For some mathematically well-formed problems, we can find the maximum using calculus to find where the gradient is zero; for other problems we have to make do with the empirical gradient, which measures the difference in fitness between two nearby points.
- An **evolutionary algorithm** is a stochastic hill-climbing search in which a population of states is maintained. New states are generated by **mutation** and by **crossover**, which combines pairs of states.
- In **nondeterministic** environments, agents can apply AND–OR search to generate **contingent** plans that reach the goal regardless of which outcomes occur during execution.
- When the environment is partially observable, the **belief state** represents the set of possible states that the agent might be in.
- Standard search algorithms can be applied directly to belief-state space to solve **sensorless problems**, and belief-state AND–OR search can solve general partially observable problems. Incremental algorithms that construct solutions state by state within a belief state are often more efficient.
- **Exploration problems** arise when the agent has no idea about the states and actions of its environment. For safely explorable environments, **online search** agents can build a map and find a goal if one exists. Updating heuristic estimates from experience provides an effective method to escape from local minima.

## Bibliographical and Historical Notes

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

The topic of local search was reinvigorated in the early 1990s by surprisingly good results for large constraint-satisfaction problems such as $n$-queens (Minton *et al.*, 1992) and Boolean satisfiability (Selman *et al.*, 1992) and by the incorporation of randomness, multiple simultaneous searches, and other improvements. This renaissance of what Christos Papadimitriou has called "New Age" algorithms also sparked increased interest among theoretical computer scientists (Koutsoupias and Papadimitriou, 1992; Aldous and Vazirani, 1994).

Tabu search

In the field of operations research, a variant of hill climbing called **tabu search** has gained popularity (Glover and Laguna, 1997). This algorithm maintains a tabu list of $k$ previously visited states that cannot be revisited; as well as improving efficiency when searching graphs, this list can allow the algorithm to escape from some local minima.

Another useful improvement on hill climbing is the STAGE algorithm (Boyan and Moore, 1998). The idea is to use the local maxima found by random-restart hill climbing to get an idea of the overall shape of the landscape. The algorithm fits a smooth quadratic surface to the set of local maxima and then calculates the global maximum of that surface analytically. This becomes the new restart point. Gomes *et al.* (1998) showed that the run times of systematic backtracking algorithms often have a **heavy-tailed distribution**, which means that the probability of a very long run time is more than would be predicted if the run times were exponentially distributed. When the run time distribution is heavy-tailed, random restarts find a solution faster, on average, than a single run to completion. Hoos and Stützle (2004) provide a book-length coverage of the topic.

Heavy-tailed distribution

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

Finding optimal solutions in continuous spaces is the subject matter of several fields, including **optimization theory**, **optimal control theory**, and the **calculus of variations**. The basic techniques are explained well by Bishop (1995); Press *et al.* (2007) cover a wide range of algorithms and provide working software.

Researchers have taken inspiration for search and optimization algorithms from a wide variety of fields of study: metallurgy (simulated annealing); biology (genetic algorithms); neuroscience (neural networks); mountaineering (hill climbing); economics (market-based algorithms (Dias *et al.*, 2006)); physics (particle swarms (Li and Yao, 2012) and spin glasses (Mézard *et al.*, 1987)); animal behavior (reinforcement learning, grey wolf optimizers (Mirjalili and Lewis, 2014)); ornithology (Cuckoo search (Yang and Deb, 2014)); entomology (ant colony (Dorigo *et al.*, 2008), bee colony (Karaboga and Basturk, 2007), firefly (Yang, 2009) and glowworm (Krishnanand and Ghose, 2009) optimization); and others.

**Linear programming** (LP) was first studied systematically by the mathematician Leonid Kantorovich (1939). It was one of the first applications of computers; the **simplex algorithm** (Dantzig, 1949) is still used despite worst-case exponential complexity. Karmarkar (1984) developed the far more efficient family of **interior-point** methods, which was shown to have polynomial complexity for the more general class of convex optimization problems by Nesterov and Nemirovski (1994). Excellent introductions to convex optimization are provided by Ben-Tal and Nemirovski (2001) and Boyd and Vandenberghe (2004).

Work by Sewall Wright (1931) on the concept of a **fitness landscape** was an important precursor to the development of genetic algorithms. In the 1950s, several statisticians, including Box (1957) and Friedman (1959), used evolutionary techniques for optimization problems, but it wasn't until Rechenberg (1965) introduced **evolution strategies** to solve optimization problems for airfoils that the approach gained popularity. In the 1960s and 1970s, John Holland (1975) championed genetic algorithms, both as a useful optimization tool and as a method to expand our understanding of adaptation (Holland, 1995).

The **artificial life** movement (Langton, 1995) took this idea one step further, viewing the products of genetic algorithms as *organisms* rather than solutions to problems. The the Baldwin effect discussed in the chapter was proposed roughly simultaneously by Conwy Lloyd Morgan (1896) and James (Baldwin, 1896). Computer simulations have helped to clarify its implications (Hinton and Nowlan, 1987; Ackley and Littman, 1991; Morgan and Griffiths, 2015). Smith and Szathmáry (1999), Ridley (2004), and Carroll (2007) provide general background on evolution.

Most comparisons of genetic algorithms to other approaches (especially stochastic hill climbing) have found that the genetic algorithms are slower to converge (O'Reilly and Oppacher, 1994; Mitchell *et al.*, 1996; Juels and Wattenberg, 1996; Baluja, 1997). Such findings are not universally popular within the GA community, but recent attempts within that community to understand population-based search as an approximate form of Bayesian learning (see Chapter 20) might help close the gap between the field and its critics (Pelikan *et al.*, 1999). The theory of **quadratic dynamical systems** may also explain the performance of GAs (Rabani *et al.*, 1998). There are some impressive practical applications of GAs, in areas as diverse as antenna design (Lohn *et al.*, 2001), computer-aided design (Renner and Ekart, 2003), climate models (Stanislawska *et al.*, 2015), medicine (Ghaheri *et al.*, 2015), and designing deep neural networks (Miikkulainen *et al.*, 2019).

The field of **genetic programming** is a subfield of genetic algorithms in which the representations are programs rather than bit strings. The programs are represented in the form of syntax trees, either in a standard programming language or in specially designed formats to represent electronic circuits, robot controllers, and so on. Crossover involves splicing together subtrees in such a way that the offspring are guaranteed to be well-formed expressions.

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

The journals *Evolutionary Computation* and *IEEE Transactions on Evolutionary Computation* cover evolutionary algorithms; articles are also found in *Complex Systems*, *Adaptive Behavior*, and *Artificial Life*. The main conference is the *Genetic and Evolutionary Com-*

*putation Conference* (GECCO). Good overview texts on genetic algorithms include those by Mitchell (1996), Fogel (2000), Langdon and Poli (2002), and Poli *et al.* (2008).

The unpredictability and partial observability of real environments were recognized early on in robotics projects that used planning techniques, including Shakey (Fikes *et al.*, 1972) and FREDDY (Michie, 1972). The problems received more attention after the publication of McDermott's (1978a) influential article *Planning and Acting*.

The first work to make explicit use of AND–OR trees seems to have been Slagle's SAINT program for symbolic integration, mentioned in Chapter 1. Amarel (1967) applied the idea to propositional theorem proving, a topic discussed in Chapter 7, and introduced a search algorithm similar to AND-OR-GRAPH-SEARCH. The algorithm was further developed by Nilsson (1971), who also described AO*—which, as its name suggests, finds optimal solutions. AO* was further improved by Martelli and Montanari (1973).

AO* is a top-down algorithm; a bottom-up generalization of A* is A*LD, for A* Lightest Derivation (Felzenszwalb and McAllester, 2007). Interest in AND–OR search underwent a revival in the early 2000s, with new algorithms for finding cyclic solutions (Jimenez and Torras, 2000; Hansen and Zilberstein, 2001) and new techniques inspired by dynamic programming (Bonet and Geffner, 2005).

The idea of transforming partially observable problems into belief-state problems originated with Astrom (1965) for the much more complex case of probabilistic uncertainty (see Chapter 17). Erdmann and Mason (1988) studied the problem of robotic manipulation without sensors, using a continuous form of belief-state search. They showed that it was possible to orient a part on a table from an arbitrary initial position by a well-designed sequence of tilting actions. More practical methods, based on a series of precisely oriented diagonal barriers across a conveyor belt, use the same algorithmic insights (Wiegley *et al.*, 1996).

The belief-state approach was reinvented in the context of sensorless and partially observable search problems by Genesereth and Nourbakhsh (1993). Additional work was done on sensorless problems in the logic-based planning community (Goldman and Boddy, 1996; Smith and Weld, 1998). This work has emphasized concise representations for belief states, as explained in Chapter 11. Bonet and Geffner (2000) introduced the first effective heuristics for belief-state search; these were refined by Bryce *et al.* (2006). The incremental approach to belief-state search, in which solutions are constructed incrementally for subsets of states within each belief state, was studied in the planning literature by Kurien *et al.* (2002); several new incremental algorithms were introduced for nondeterministic, partially observable problems by Russell and Wolfe (2005). Additional references for planning in stochastic, partially observable environments appear in Chapter 17.

Algorithms for exploring unknown state spaces have been of interest for many centuries. Depth-first search in a reversible maze can be implemented by keeping one's left hand on the wall; loops can be avoided by marking each junction. The more general problem of exploring **Eulerian graphs** (i.e., graphs in which each node has equal numbers of incoming and outgoing edges) was solved by an algorithm due to Hierholzer (1873).

Eulerian graph

The first thorough algorithmic study of the exploration problem for arbitrary graphs was carried out by Deng and Papadimitriou (1990), who developed a completely general algorithm but showed that no bounded competitive ratio is possible for exploring a general graph. Papadimitriou and Yannakakis (1991) examined the question of finding paths to a goal in geometric path-planning environments (where all actions are reversible). They showed that

a small competitive ratio is achievable with square obstacles, but with general rectangular obstacles no bounded ratio can be achieved. (See Figure 4.20.)

In a dynamic environment, the state of the world can spontaneously change without any action by the agent. For example, the agent can plan an optimal driving route from *A* to *B*, but an accident or unusually bad rush hour traffic can spoil the plan. Incremental search algorithms such as Lifelong Planning A* (Koenig *et al.*, 2004) and D* Lite (Koenig and Likhachev, 2002) deal with this situation.

The LRTA* algorithm was developed by Korf (1990) as part of an investigation into **real-time search** for environments in which the agent must act after searching for only a fixed amount of time (a common situation in two-player games). LRTA* is in fact a special case of reinforcement learning algorithms for stochastic environments (Barto *et al.*, 1995). Its policy of optimism under uncertainty—always head for the closest unvisited state—can result in an exploration pattern that is less efficient in the uninformed case than simple depth-first search (Koenig, 2000). Dasgupta *et al.* (1994) show that online iterative deepening search is optimally efficient for finding a goal in a uniform tree with no heuristic information.

Several informed variants on the LRTA* theme have been developed with different methods for searching and updating within the known portion of the graph (Pemberton and Korf, 1992). As yet, there is no good theoretical understanding of how to find goals with optimal efficiency when using heuristic information. Sturtevant and Bulitko (2016) provide an analysis of some pitfalls that occur in practice.