4 BEYOND CLASSICAL SEARCH

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

Chapter 3 addressed a single category of problems: observable, deterministic, known environments where the solution is a sequence of actions. In this chapter, we look at what happens when these assumptions are relaxed. We begin with a fairly simple case: Sections 4.1 and 4.2 cover algorithms that perform purely **local search** in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state. These algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it. The family of local search algorithms includes methods inspired by statistical physics (**simulated annealing**) and evolutionary biology (**genetic algorithms**).

Then, in Sections 4.3–4.4, we examine what happens when we relax the assumptions of determinism and observability. The key idea is that if an agent cannot predict exactly what percept it will receive, then it will need to consider what to do under each **contingency** that its percepts may reveal. With partial observability, the agent will also need to keep track of the states it might be in.

Finally, Section 4.5 investigates **online search**, in which the agent is faced with a state space that is initially unknown and must be explored.

4.1 LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

The search algorithms that we have seen so far are designed to explore search spaces systematically. This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path. When a goal is found, the *path* to that goal also constitutes a *solution* to the problem. In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem (see page 71), what matters is the final configuration of queens, not the order in which they are added. The same general property holds for many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.

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

LOCAL SEARCH

CURRENT NODE

In addition to finding goals, local search algorithms are useful for solving pure **optimization problems**, in which the aim is to find the best state according to an **objective function**. Many optimization problems do not fit the "standard" search model introduced in Chapter 3. For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no "goal test" and no "path cost" for this problem.

OPTIMIZATION PROBLEM

OBJECTIVE FUNCTION

To understand local search, we find it useful to consider the **state-space landscape** (as in Figure 4.1). A landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function). If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**; if elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**. (You can convert from one to the other just by inserting a minus sign.) Local search algorithms explore this landscape. A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.

STATE-SPACE LANDSCAPE

GLOBAL MINIMUM

GLOBAL MAXIMUM

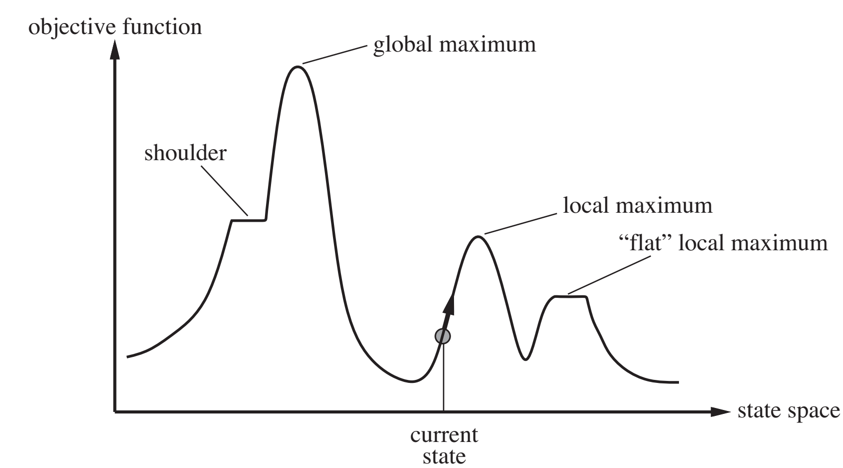


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

  current ← MAKE-NODE(*problem*.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of *current*
    if neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
    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; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

4.1.1 Hill-climbing search

HILL CLIMBING
STEEPEST ASCENT

The **hill-climbing** search algorithm (**steepest-ascent** version) is shown in Figure 4.2. It is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function. Hill climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia.

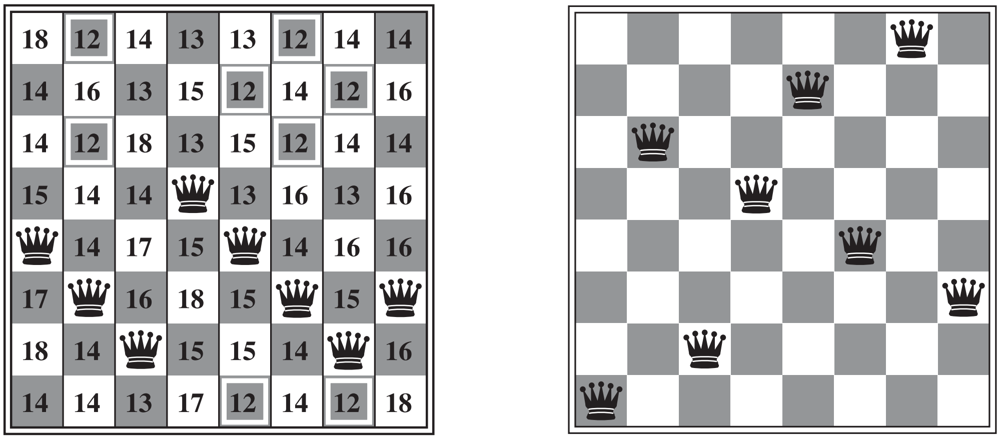
To illustrate hill climbing, we will use the **8-queens problem** introduced on page 71. Local search algorithms typically use a **complete-state formulation**, where each state has 8 queens on the board, one per column. The successors of a state are all possible states generated by moving a single queen 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, either directly or indirectly. The global minimum of this function is zero, which occurs only at perfect solutions. Figure 4.3(a) shows a state with $h = 17$. The figure also shows the values of all its successors, with the best successors having $h = 12$. Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.

GREEDY LOCAL SEARCH

Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing often makes 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(a), it takes just five steps to reach the state in Figure 4.3(b), which has $h = 1$ and is very nearly a solution. Unfortunately, hill climbing often gets stuck for 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



(a) (b)

Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

concretely, the state in Figure 4.3(b) 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.

PLATEAU

SHOULDER

- **Plateaux**: 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 might get lost on the plateau.

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

SIDEWAYS MOVE

The algorithm in Figure 4.2 halts if it reaches a plateau where the best successor has the same value as the current state. Might it not be a good idea to keep going—to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 4.1? The answer is usually yes, but we must take care. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sideways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

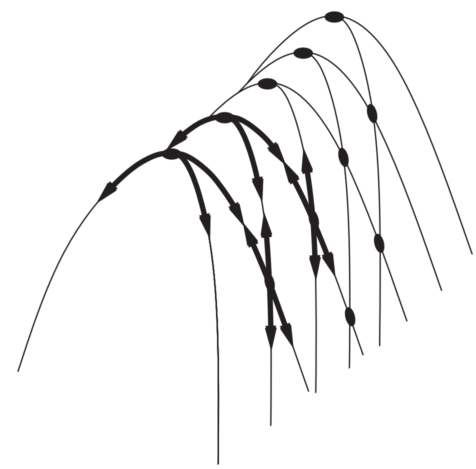


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

STOCHASTIC HILL CLIMBING

FIRST-CHOICE HILL CLIMBING

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

RANDOM-RESTART HILL CLIMBING

The hill-climbing algorithms described so far are incomplete—they often fail to find a goal when one exists because they can get stuck on local maxima. **Random-restart hill climbing** adopts the well-known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial states,[1] until a goal is found. It is trivially complete with probability approaching 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 under a minute.[2]

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

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

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

4.1.2 Simulated annealing

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

SIMULATED ANNEALING

GRADIENT DESCENT

The innermost loop of the simulated-annealing algorithm (Figure 4.5) is quite similar to hill climbing. Instead of picking the *best* move, however, it picks a *random* move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases. If the $schedule$ lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks. In Exercise 4.4, you are asked to compare its performance to that of random-restart hill climbing on the 8-queens puzzle.

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[3] keeps track of k states rather than

LOCAL BEAM SEARCH

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

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
     T ← schedule(t)
     if T = 0 then return current
     next ← a randomly selected successor of current
     ΔE ← next.VALUE – current.VALUE
     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. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature T as a function of time.

just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

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

STOCHASTIC BEAM SEARCH

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

4.1.4 Genetic algorithms

GENETIC ALGORITHM

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

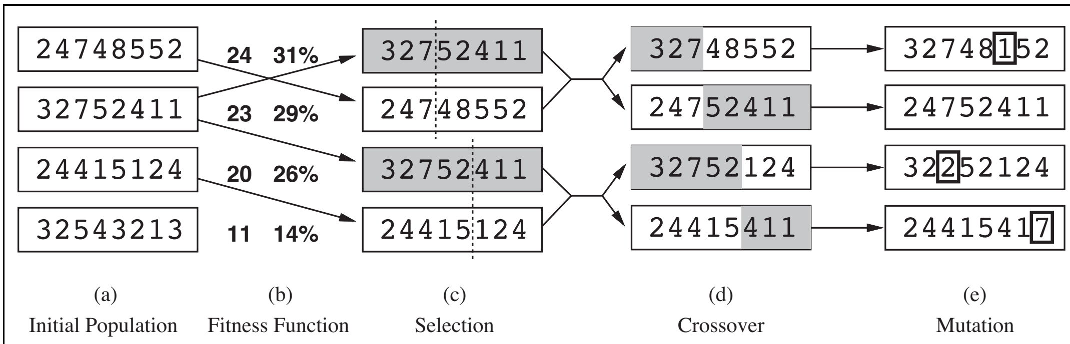


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

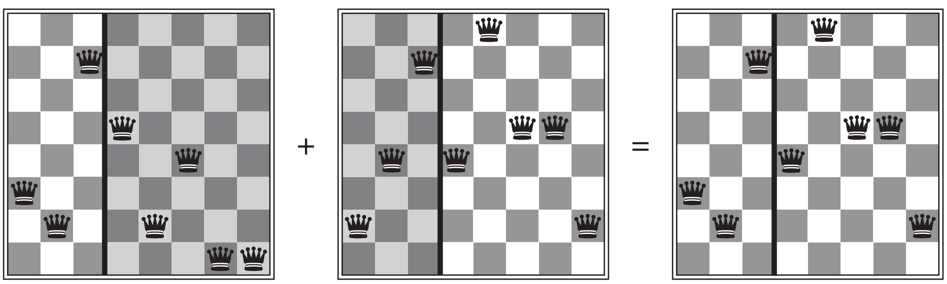


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

POPULATION

INDIVIDUAL

Like beam searches, GAs begin with a set of k randomly generated states, called the **population**. Each state, or **individual**, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires $8 \times \log_2 8 = 24$ bits. Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8. (We demonstrate later that the two encodings behave differently.) Figure 4.6(a) shows a population of four 8-digit strings representing 8-queens states.

FITNESS FUNCTION

The production of the next generation of states is shown in Figure 4.6(b)–(e). In (b), each state is rated by the objective function, or (in GA terminology) the **fitness function**. A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of *nonattacking* pairs of queens, which has a value of 28 for a solution. The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.

In (c), two pairs are selected at random for reproduction, in accordance with the prob-

CROSSOVER

abilities in (b). Notice that one individual is selected twice and one not at all.[4] For each pair to be mated, a **crossover** point is chosen randomly from the positions in the string. In Figure 4.6, the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.[5]

In (d), the offspring themselves are created by crossing over the parent strings at the crossover point. For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent. The 8-queens states involved in this reproduction step are shown in Figure 4.7. The example shows that when two parent states are quite different, the crossover operation can produce a state that is a long way from either parent state. It is often the case that the population is quite diverse early on in the process, so crossover (like simulated annealing) frequently takes large steps in the state space early in the search process and smaller steps later on when most individuals are quite similar.

MUTATION

Finally, in (e), each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column. Figure 4.8 describes an algorithm that implements all these steps.

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

SCHEMA

INSTANCE

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

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

[5] It is here that the encoding matters. If a 24-bit encoding is used instead of 8 digits, then the crossover point has a 2/3 chance of being in the middle of a digit, which results in an essentially arbitrary mutation of that digit.

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

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

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

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

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

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

4.2 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. Yet none of the algorithms we have described (except for first-choice hill climbing and simulated annealing) can handle continuous state and action spaces, because they have infinite branching factors. This section provides a *very brief* introduction to some local search techniques for finding optimal solutions in continuous spaces. The literature on this topic is vast; many of the basic techniques

EVOLUTION AND SEARCH

The theory of **evolution** was developed in Charles Darwin's *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* organisms rather than just one. The actual mechanisms of evolution are, however, far richer than most genetic algorithms allow. For example, mutations can involve reversals, duplications, and movement of large chunks of DNA; some viruses borrow DNA from one organism and insert it in another; and there are transposable genes that do nothing but copy themselves many thousands of times within the genome. There are even genes that poison cells from potential mates that do not carry the gene, thereby increasing their 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^{45} or so organisms without improving its search heuristics one iota. Fifty years before Darwin, however, the otherwise great French naturalist Jean Lamarck (1809) proposed a theory of evolution whereby traits *acquired by adaptation during an organism's lifetime* would be passed on to its offspring. Such a process would be effective but does not seem to occur in nature. Much later, James Baldwin (1896) proposed a superficially similar theory: that behavior learned during an organism's lifetime could accelerate the rate of evolution. Unlike Lamarck's, Baldwin's theory is entirely consistent with Darwinian evolution because it relies on selection pressures operating on individuals that have found local optima among the set of possible behaviors allowed by their genetic makeup. Computer simulations confirm that the "Baldwin effect" is real, once "ordinary" evolution has created organisms whose internal performance measure correlates with actual fitness.

originated in the 17th century, after the development of calculus by Newton and Leibniz.[6] We find uses for these techniques at several places in the 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 distances from each city on the map (Figure 3.2) to its nearest airport is minimized. The state space is then defined by the coordinates of the airports: (x_1, y_1), (x_2, y_2), and (x_3, y_3). This is a *six-dimensional* space; we also say that states are defined by six **variables**. (In general, states are defined by an n-dimensional vector of variables, $\mathbf{x}$.) Moving around in this space corresponds to moving one or more of the airports on the map. The objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state once we compute the closest cities. Let C_i be the set of cities whose closest airport (in the current state) is airport i. Then, *in the neighborhood of the current state*, where the C_is remain constant, we have

VARIABLE

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

This expression is correct *locally*, but not globally because the sets C_i are (discontinuous) functions of the state.

DISCRETIZATION

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

GRADIENT

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

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

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

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

Given a locally correct expression for the gradient, we can perform steepest-ascent hill climb-

[6] A basic knowledge of multivariate calculus and vector arithmetic is useful for reading this section.

ing by updating the current state according to the formula

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

STEP SIZE

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

EMPIRICAL GRADIENT

Hidden beneath the phrase "α is a small constant" lies a huge variety of methods for adjusting α. The basic problem is that, if α is too small, too many steps are needed; if α 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 α—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.

LINE SEARCH

NEWTON–RAPHSON

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

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

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

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

HESSIAN

where $\mathbf{H}_f(\mathbf{x})$ is the **Hessian** matrix of second derivatives, whose elements H_{ij} are given by $\partial^2 f / \partial x_i \partial x_j$. 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).[7] 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.

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

CONSTRAINED OPTIMIZATION

A final topic with which a passing acquaintance is useful is **constrained optimization**. An optimization problem is constrained if solutions must satisfy some hard constraints on the values of the variables. For example, in our airport-siting problem, we might constrain sites

[7] 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.

to be inside Romania and on dry land (rather than in the middle of lakes). The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function. The best-known category is that of **linear programming** problems, in which constraints must be linear inequalities forming a **convex set** [8] and the objective function is also linear. The time complexity of linear programming is polynomial in the number of variables.

LINEAR PROGRAMMING

CONVEX SET

Linear programming is probably the most widely studied and broadly useful class of optimization problems. 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).

CONVEX OPTIMIZATION

4.3 SEARCHING WITH NONDETERMINISTIC ACTIONS

In Chapter 3, we assumed that the environment is fully observable and deterministic and that the agent knows what the effects of each action are. Therefore, the agent can calculate exactly which state results from any sequence of actions and always knows which state it is in. Its percepts provide no new information after each action, although of course they tell the agent the initial state.

When the environment is either partially observable or nondeterministic (or both), percepts become useful. In a partially observable environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals. When the environment is nondeterministic, percepts tell the agent which of the possible outcomes of its actions has actually occurred. In both cases, the future percepts cannot be determined in advance and the agent's future actions will depend on those future percepts. So the solution to a problem is not a sequence but a **contingency plan** (also known as a **strategy**) that specifies what to do depending on what percepts are received. In this section, we examine the case of nondeterminism, deferring partial observability to Section 4.4.

CONTINGENCY PLAN

STRATEGY

4.3.1 The erratic vacuum world

As an example, we use the vacuum world, first introduced in Chapter 2 and defined as a search problem in Section 3.2.1. Recall that the state space has eight states, as shown in Figure 4.9. There are three actions—*Left*, *Right*, and *Suck*—and the goal is to clean up all the dirt (states 7 and 8). If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms 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.

[8] 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.

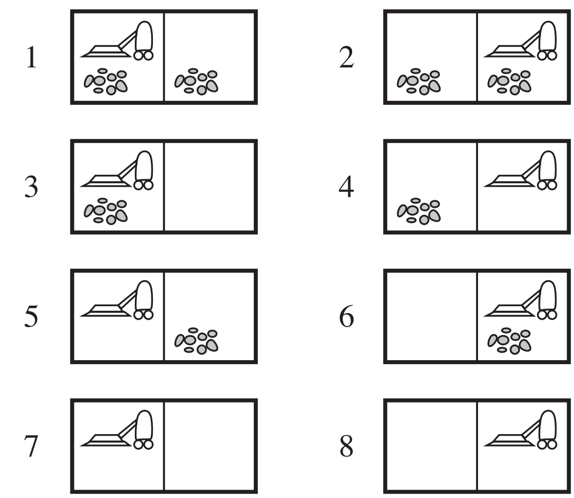


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

ERRATIC VACUUM WORLD

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.[9]

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 that returns a single 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 leads to a state in the set $\{5, 7\}$—the dirt in the right-hand square may or may not be vacuumed up.

We also need to generalize the notion of a **solution** to the problem. For example, if we start in state 1, there is no single *sequence* of actions that solves the problem. Instead, we need a contingency plan such as the following:

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

Thus, solutions for nondeterministic problems can contain nested **if–then–else** statements; this means that they are *trees* rather than sequences. This allows the selection of actions based on contingencies arising during execution. Many problems in the real, physical world are contingency problems because exact prediction is impossible. For this reason, many people keep their eyes open while walking around or driving.

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

4.3.2 AND–OR search trees

The next question is how to find 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. 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 leads to a state in the set $\{5, 7\}$, 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

A solution for an AND–OR search problem is a subtree 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). (The plan uses if–then–else notation to handle the AND branches, but when there are more than two branches at a node, it might be better to use a **case**

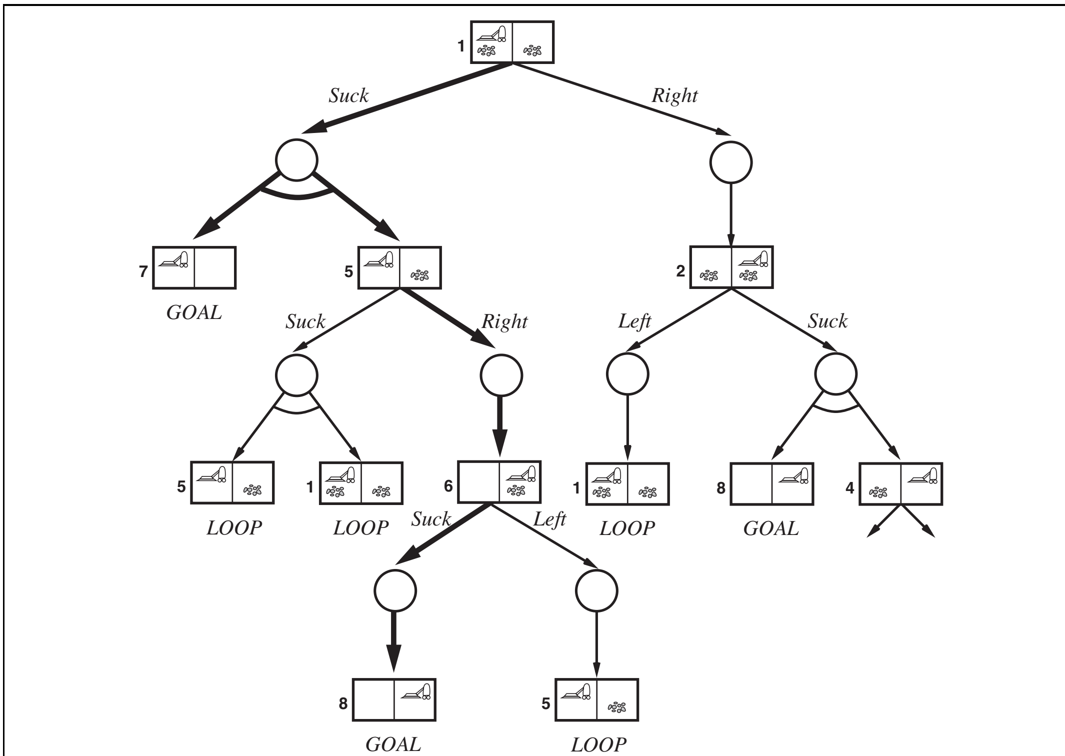


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

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

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

```
function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
   for each s_i in states do
       plan_i ← OR-SEARCH(s_i, problem, 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. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

construct.) Modifying the basic problem-solving agent shown in Figure 3.1 to execute contingent solutions of this kind is straightforward. One may also consider a somewhat different agent design, in which the agent can act *before* it has found a guaranteed plan and deals with some contingencies only as they arise during execution. This type of **interleaving** of search and execution is also useful for exploration problems (see Section 4.5) and for game playing (see Chapter 5).

INTERLEAVING

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. Exercise 4.5 investigates this issue.

AND–OR graphs can also be explored by breadth-first or best-first methods. 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. Pointers are given in the bibliographical notes at the end of the chapter.

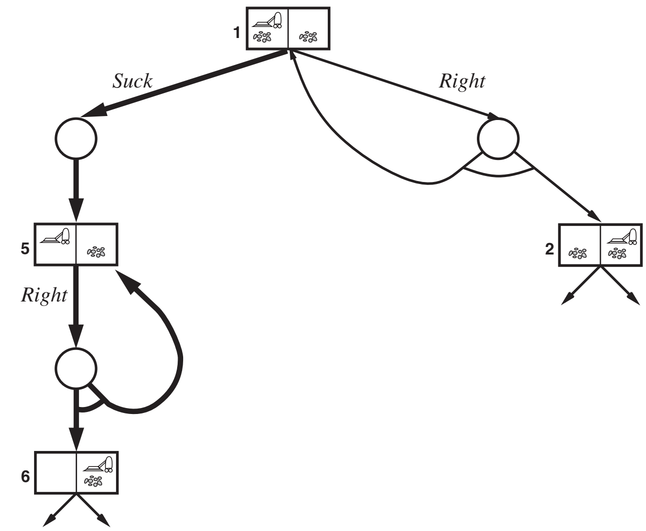


Figure 4.12 Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

4.3.3 Try, try again

Consider the 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 state set $\{1, 2\}$. Figure 4.12 shows part of the search graph; clearly, there are no longer any acyclic solutions from state 1, and AND-OR-GRAPH-SEARCH would return with failure. There is, however, a **cyclic solution**, which is to keep trying $Right$ until it works. We can express this solution by adding a **label** to denote some portion of the plan and using that label later instead of repeating the plan itself. Thus, our cyclic solution is

CYCLIC SOLUTION

LABEL

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

(A better syntax for the looping part of this plan would be "**while** $State = 5$ **do** $Right$.") In general a cyclic plan may be considered a solution provided that every leaf is a goal state and that a leaf is reachable from every point in the plan. The modifications needed to AND-OR-GRAPH-SEARCH are covered in Exercise 4.6. The key realization is that a loop in the state space back to a state L translates to a loop in the plan back to the point where the subplan for state L is executed.

Given the definition of a cyclic solution, an agent executing such a solution will eventually reach the goal *provided that each outcome of a nondeterministic action eventually occurs*. Is this condition reasonable? It depends on the reason for the nondeterminism. If the action rolls a die, then it's reasonable to suppose that eventually a six will be rolled. If the action is to insert a hotel card key into the door lock, but it doesn't work the first time, then perhaps it will eventually work, or perhaps one has the wrong key (or the wrong room!). After seven or eight tries, most people will assume the problem is with the key and will go back to the front desk to get a new one. One way to understand this decision is to say that the initial problem formulation (observable, nondeterministic) is abandoned in favor of a different formulation (partially observable, deterministic) where the failure is attributed to an unobservable property of the key. We have more to say on this issue in Chapter 13.

4.4 SEARCHING WITH PARTIAL OBSERVATIONS

We now turn to the problem of partial observability, where the agent's percepts do not suffice to pin down the exact state. As noted at the beginning of the previous section, if the agent is in one of several possible states, then an action may lead to one of several possible outcomes—*even if the environment is deterministic*. The key concept required for solving partially observable problems is the **belief state**, representing the agent's current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point. We begin with the simplest scenario for studying belief states, which is when the agent has no sensors at all; then we add in partial sensing as well as nondeterministic actions.

BELIEF STATE

4.4.1 Searching with no observation

When the agent's percepts provide *no information at all*, we have what is called a **sensorless** problem or sometimes a **conformant** problem. At first, one might think the sensorless agent has no hope of solving a problem if it has no idea what state it's in; in fact, sensorless problems are quite often solvable. Moreover, sensorless agents can be surprisingly 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. The high cost of sensing is another reason to avoid it: for example, doctors often prescribe a broad-spectrum antibiotic rather than using the contingent plan of doing an expensive blood test, then waiting for the results to come back, and then prescribing a more specific antibiotic and perhaps hospitalization because the infection has progressed too far.

SENSORLESS

CONFORMANT

We can make a sensorless version of the vacuum world. Assume that the agent knows the geography of its world, but doesn't know its location or the distribution of dirt. In that case, its initial state could be any element of the set $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Now, consider what happens if it tries the action *Right*. This will cause it to be in one of the states $\{2, 4, 6, 8\}$—the agent now has more information! Furthermore, the action sequence [*Right*,*Suck*] will always end up in one of the states $\{4, 8\}$. Finally, the sequence [*Right*,*Suck*,*Left*,*Suck*] is guaranteed to reach the goal state 7 no matter what the start state. We say that the agent can **coerce** the world into state 7.

COERCION

To solve sensorless problems, we search in the space of belief states rather than physical states.[10] Notice that in belief-state space, the problem is *fully observable* because the agent

[10] 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.

always knows its own belief state. Furthermore, the solution (if any) 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 nondeterminstic*.

It is instructive to see how the belief-state search problem is constructed. Suppose the underlying physical problem P is defined by ACTIONS_P, RESULT_P, GOAL-TEST_P, and STEP-COST_P. Then we can define the corresponding sensorless problem as follows:

- **Belief states**: The entire belief-state space contains every possible set of physical states. If P has N states, then the sensorless problem has up to 2^N states, although many may be unreachable from the initial state.
- **Initial state**: Typically the set 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 be the end of the world, 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**: The agent doesn't know which state in the belief state is the right one; so as far as it knows, it might get to any of the states resulting from applying the action to one of the physical states in the belief state. For deterministic actions, the set of states that might be reached is
$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\} . \qquad (4.4)$$
With deterministic actions, b' is never larger than b. With nondeterminism, we have
$$\begin{aligned} b' = \text{RESULT}(b, a) &= \{s' : s' \in \text{RESULTS}_P(s, a) \text{ and } s \in b\} \\ &= \bigcup_{s \in b} \text{RESULTS}_P(s, a) , \end{aligned}$$
which may be larger than b, as shown in Figure 4.13. The process of generating the new belief state after the action is called the **prediction** step; the notation $b' = \text{PREDICT}_P(b, a)$ will come in handy.

  PREDICTION
- **Goal test**: The agent wants a plan that is sure to work, which means that a belief state satisfies the goal only if *all* the physical states in it satisfy GOAL-TEST_P. The agent may *accidentally* achieve the goal earlier, but it won't *know* that it has done so.
- **Path 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 several values. (This gives rise to a new class of problems, which we explore in Exercise 4.9.) 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.

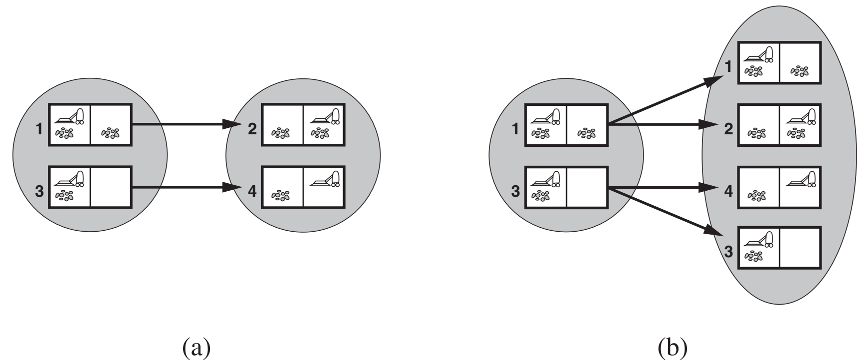


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, $Right$. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

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 apply any of the search algorithms of Chapter 3. In fact, we can do a little bit more than that. In "ordinary" graph search, newly generated states are tested to see if they are identical to existing states. 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*. It is easy to prove (Exercise 4.8) that if an action sequence is a solution for a belief state b, it is also a solution for any subset of b. Hence, we can discard a path reaching $\{1, 3, 5, 7\}$ if $\{5, 7\}$ has already been generated. 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. 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. The difficulty is not so much the vastness of the belief-state space—even though it is exponentially larger than the underlying physical state space; in most cases the branching factor and solution length in the belief-state space and physical state space are not so different. The real difficulty lies with the size of each belief state. For example, the initial belief state for the 10×10 vacuum world contains 100×2^{100} or around 10^{32} physical states—far too many if we use the atomic representation, which is an explicit list of states.

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 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

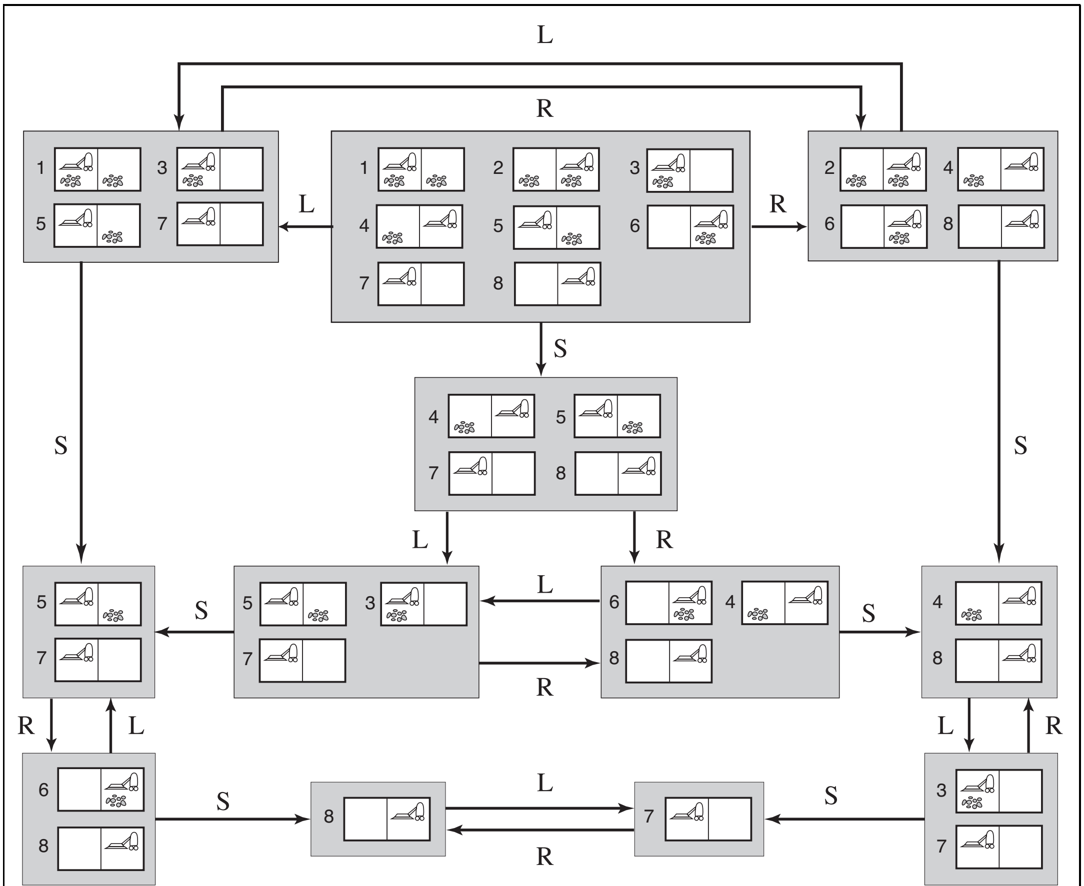


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box. Actions are represented by labeled links. Self-loops are omitted for clarity.

INCREMENTAL BELIEF-STATE SEARCH

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. 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.

Even the most efficient solution algorithm is not of much use when no solutions exist. Many things just cannot be done without sensing. For example, the sensorless 8-puzzle is impossible. On the other hand, a little bit of sensing can go a long way. For example, every 8-puzzle instance is solvable if just one square is visible—the solution involves moving each tile in turn into the visible square and then keeping track of its location.

4.4.2 Searching with observations

For a general partially observable problem, we have to specify how the environment generates percepts for the agent. For example, we might define the local-sensing vacuum world to be one in which the agent has a position sensor and a local dirt sensor but has no sensor capable of detecting dirt in other squares. The formal problem specification includes a PERCEPT(s) function that returns the percept received in a given state. (If sensing is nondeterministic, then we use a PERCEPTS function that returns a set of possible percepts.) For example, in the local-sensing vacuum world, the PERCEPT in state 1 is $[A, Dirty]$. Fully observable problems are a special case in which PERCEPT$(s) = s$ for every state s, while sensorless problems are a special case in which PERCEPT$(s) = null$.

When observations are partial, it will usually be the case that several states could have produced any given percept. For example, the percept $[A, Dirty]$ is produced by state 3 as well as by state 1. Hence, given this as the initial percept, the initial belief state for the local-sensing vacuum world will be $\{1, 3\}$. The ACTIONS, STEP-COST, and GOAL-TEST are constructed from the underlying physical problem just as for sensorless problems, but the transition model is a bit more complicated. We can think of transitions from one belief state to the next for a particular action as occurring in three stages, as shown in Figure 4.15:

- The **prediction** stage is the same as for sensorless problems: given the action a in belief state b, the predicted belief state is $\hat{b} = \text{PREDICT}(b, a)$.[11]
- The **observation prediction** stage determines the set of percepts o that could be observed in the predicted belief state:

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

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

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

  Notice that each updated belief state b_o can be no larger than the predicted belief state $\hat{b}$; observations can only help reduce uncertainty compared to the sensorless case. Moreover, for deterministic sensing, the belief states for the different possible percepts will be disjoint, forming a *partition* of the original predicted belief state.

[11] Here, and throughout the book, the "hat" in $\hat{b}$ means an estimated or predicted value for b.

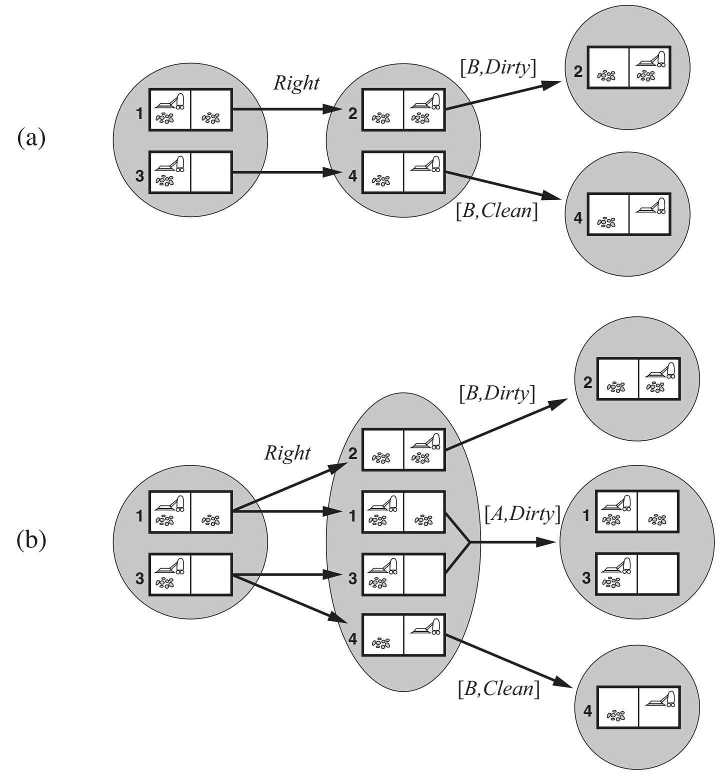


Figure 4.15 Two example of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are $[B, Dirty]$ and $[B, Clean]$, leading 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 $[A, Dirty]$, $[B, Dirty]$, and $[B, Clean]$, leading to three belief states as shown.

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

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

Again, the nondeterminism in the partially observable problem comes from the inability to predict exactly which percept will be received after acting; underlying nondeterminism in the physical environment may *contribute* to this inability by enlarging the belief state at the prediction stage, leading to more percepts at the observation stage.

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 and the PERCEPT function. Given

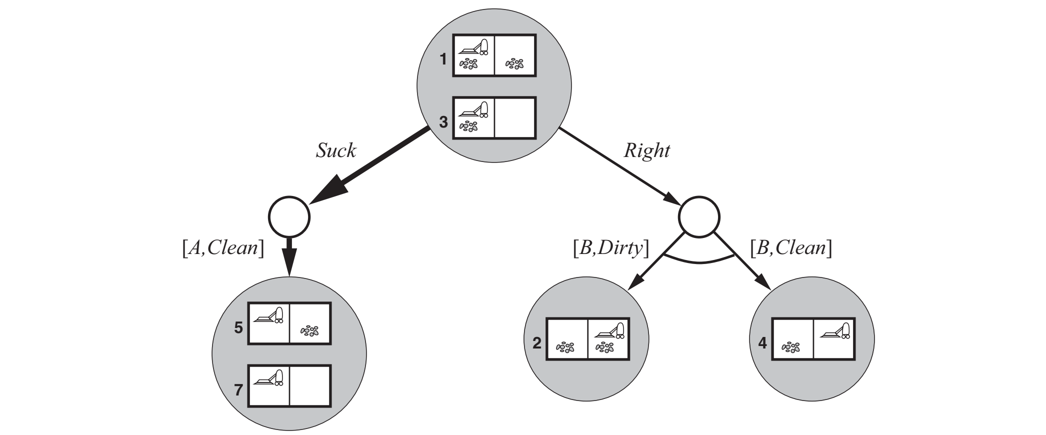


Figure 4.16 The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first step of the solution.

such a 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 $[A, \mathit{Dirty}]$. The solution is the conditional plan

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

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 be able to execute a solution that requires testing 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

The design of a problem-solving agent for partially observable environments is quite similar to the simple problem-solving agent in Figure 3.1: the agent formulates a problem, calls a search algorithm (such as AND-OR-GRAPH-SEARCH) to solve it, and executes the solution. There are two main differences. First, the solution to a problem will be a conditional plan rather than a sequence; if the first step is an if–then–else expression, the agent will need to test the condition in the if-part and execute the then-part or the else-part accordingly. 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

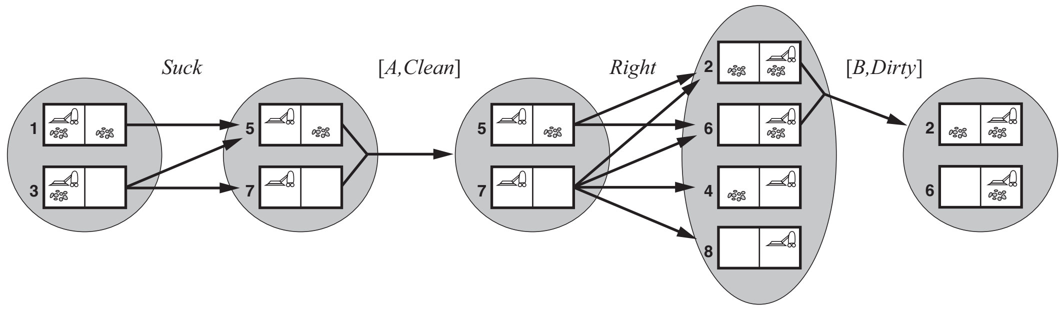


Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

agent. Given an initial belief state b, an action a, and a percept o, the new belief state is:

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

Figure 4.17 shows the belief state being maintained in the *kindergarten* vacuum world with local sensing, wherein any square may become dirty at any time unless the agent is actively cleaning it at that moment.[12]

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 exact update computation becomes infeasible and the agent will have 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 stochastic, continuous-state environments with the tools of probability theory, as explained in Chapter 15. Here we will show an example in a discrete environment with detrministic sensors and nondeterministic actions.

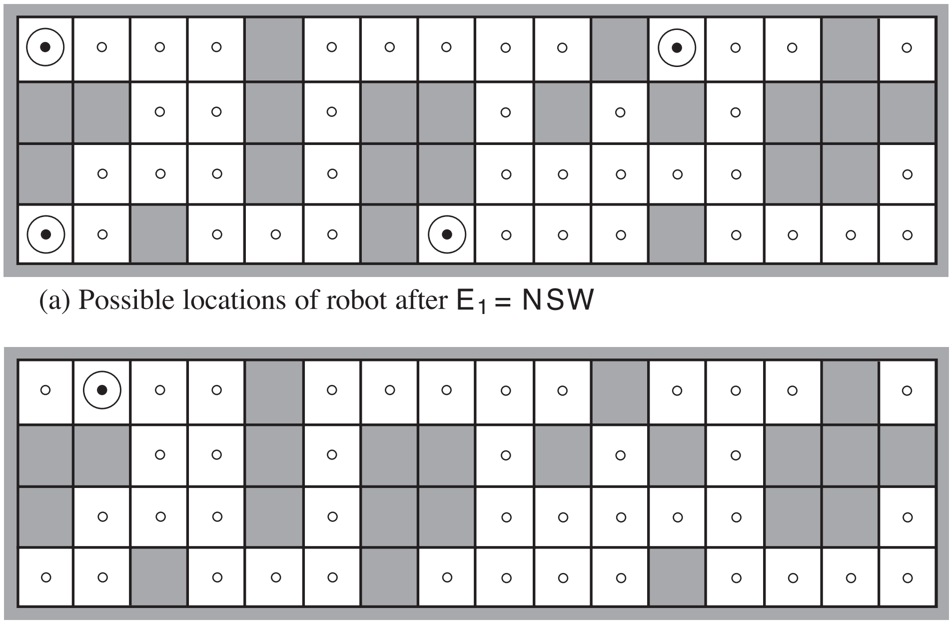
MONITORING
FILTERING
STATE ESTIMATION
RECURSIVE

LOCALIZATION

The example concerns a robot with the task of **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 black square in the figure—in each of the four compass directions. We assume that the sensors give perfectly correct data, and that the robot has a correct map of the enviornment. But unfortunately the robot's navigational system is broken, so when it executes a *Move* 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, so it does not know where it is. Thus its initial belief state b consists of the set of all locations. The the robot receives the percept

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



(a) Possible locations of robot after $\mathsf{E_1 = NSW}$

(b) Possible locations of robot After $\mathsf{E_1 = NSW, E_2 = NS}$

Figure 4.18 Possible positions of the robot, $\odot$, (a) after one observation $E_1 = NSW$ and (b) after a second observation $E_2 = NS$. When sensors are noiseless and the transition model is accurate, there are no other possible locations for the robot consistent with this sequence of two observations.

NSW, meaning there are obstacles to the north, west, and south, and does an update using the equation $b_o = \textsc{Update}(b)$, 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 NWS.

Next the robot executes a $Move$ action, but the result is nondeterministic. The new belief state, $b_a = \textsc{Predict}(b_o, Move)$, contains all the locations that are one step away from the locations in b_o. When the second percept, NS, arrives, the robot does $\textsc{Update}(b_a, NS)$ 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

$$\textsc{Update}(\textsc{Predict}(\textsc{Update}(b, NSW), Move), NS)\ .$$

With nondetermnistic actions the $\textsc{Predict}$ step grows the belief state, but the $\textsc{Update}$ 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 NS percepts, but never know where in the corridor(s) it was.

4.5 ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENTS

OFFLINE SEARCH

ONLINE SEARCH

So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution before setting foot in the real world and then execute the solution. In contrast, an **online search**[13] 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 semidynamic domains—domains where there is a penalty for sitting around and computing too long. Online 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.

EXPLORATION PROBLEM

Online search is a *necessary* idea for unknown environments, where the agent does not know what states exist or what its actions do. In this state of ignorance, the agent faces an **exploration problem** and must use its actions as experiments in order to learn enough to make deliberation worthwhile.

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

4.5.1 Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment (Chapter 17 relaxes these assumptions), but we stipulate that the agent knows only the following:

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

Note in particular that the agent *cannot* 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.

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

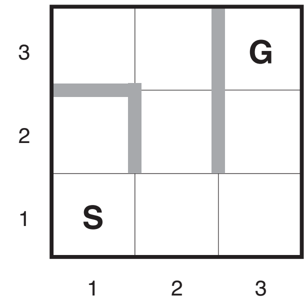


Figure 4.19 A simple maze problem. The agent starts at S and must reach G but knows nothing of the environment.

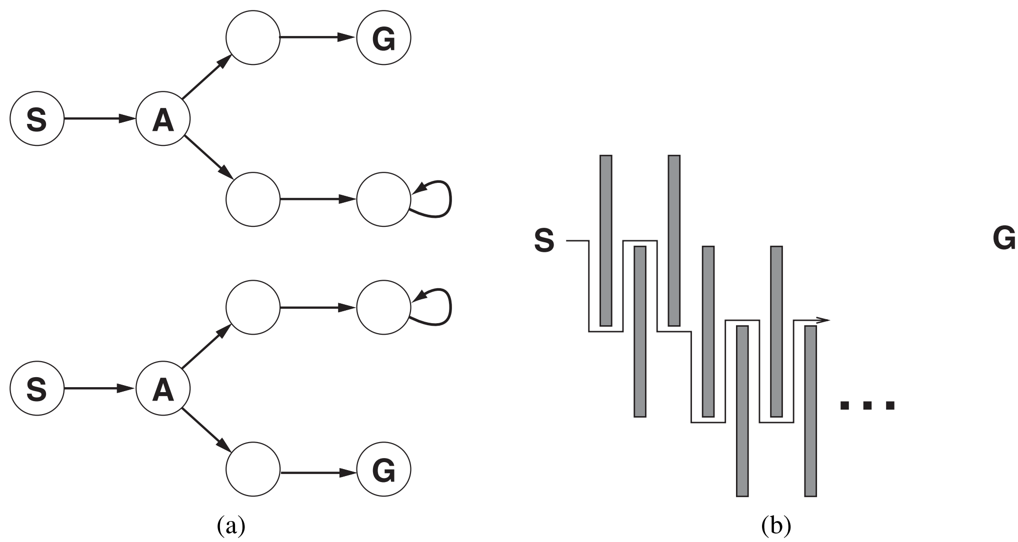


Figure 4.20 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent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

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.

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

COMPETITIVE RATIO

IRREVERSIBLE

DEAD END

ADVERSARY ARGUMENT

Although this sounds like a reasonable request, it is easy to see that the best achievable competitive ratio is infinite in some cases. For example, if some actions are **irreversible**—i.e., they lead to a state from which no action leads back to the previous state—the online search might accidentally reach a **dead-end** state from which no goal state is reachable. Perhaps the term "accidentally" is unconvincing—after all, there might be an algorithm that happens not to take the dead-end path as it explores. Our claim, to be more precise, is that *no algorithm can avoid dead ends in all state spaces.* Consider the two dead-end state spaces in Figure 4.20(a). To an online search algorithm that has visited states S and A, the two state spaces look *identical*, so it must make the same decision in both. Therefore, it will fail in one of them. This is an example of an **adversary argument**—we can imagine an adversary constructing the state space while the agent explores it and putting the goals and dead ends wherever it chooses.

SAFELY EXPLORABLE

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

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 describe the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

4.5.2 Online search agents

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

An online depth-first search agent is shown in Figure 4.21. This agent stores its map in a table, RESULT$[s, a]$, that records the state resulting from executing action a in state s. Whenever an action from the current state has not been explored, the agent tries that action. The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack physically. In depth-first search, this means going back to the state from which the agent most recently entered the current state. To achieve that, the algorithm keeps a table that

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

  if GOAL-TEST(s′) then return stop
  if s′ is a new state (not in untried) then untried[s′] ← 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 is applicable only in state spaces in which every action can be "undone" by some other action.

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, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

RANDOM WALK

Instead of random restarts, one might consider using a **random walk** to explore the environment. A random walk simply selects at random one of the available actions from the

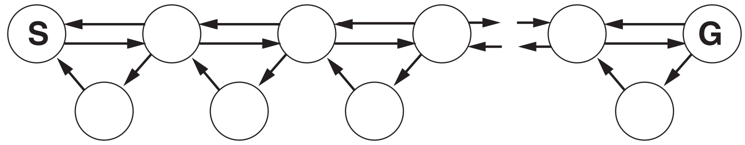


Figure 4.22 An environment in which a random walk will take exponentially many steps to find the goal.

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.[14] 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, at each step, backward progress is twice as likely as forward progress. The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.

Augmenting hill climbing with *memory* rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" $H(s)$ of the cost to reach the goal from each state that has been visited. $H(s)$ starts out being just the heuristic 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 shaded 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 $1 + 9$ and $1 + 2$, so it seems best to move right. Now, it is clear that the cost estimate of 2 for the shaded state was overly optimistic. Since the best move cost 1 and led to a state that is at least 2 steps from a goal, the shaded state must be at least 3 steps from a goal, so its H should be updated accordingly, as shown in Figure 4.23(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

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,

[14] 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).

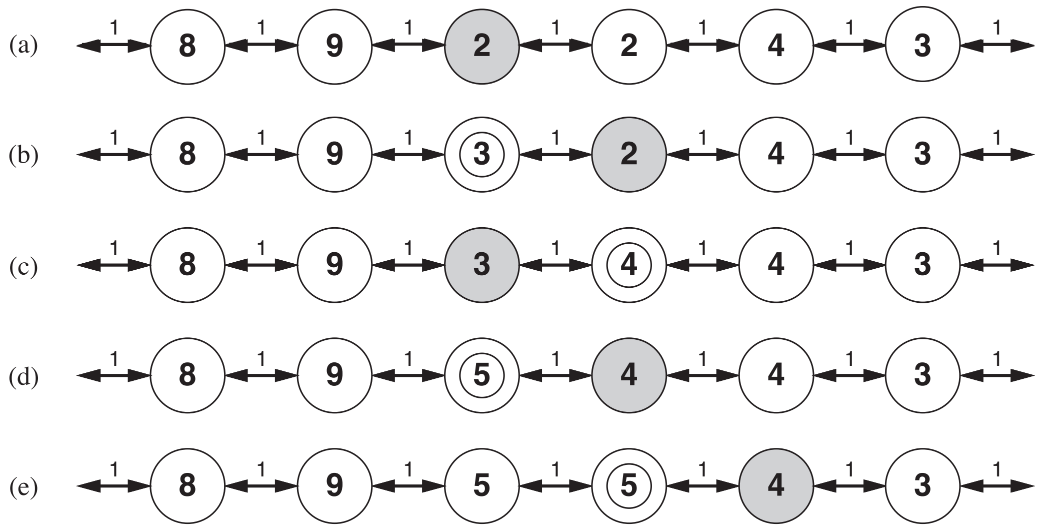


Figure 4.23 Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and each link is labeled with its step cost. The shaded state marks the location of the agent, and the updated cost estimates at each iteration are circled.

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

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

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

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

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 21.

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. (Notice that the assumption of deterministic environments means that one experience is enough for each action.) Second, the local search agents acquire more accurate estimates of the cost of each state by using local updating rules, as in LRTA*. In Chapter 21, 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. Part III is devoted to this issue. Second, we need algorithms that can construct suitable general rules from the specific observations made by the agent. These are covered in Chapter 18.

4.6 SUMMARY

This chapter has examined search algorithms for problems beyond the "classical" case of finding the shortest path to a goal in an observable, deterministic, discrete environment.

- *Local search* methods such as **hill climbing** operate on complete-state formulations, keeping only a small number of nodes in memory. Several stochastic algorithms have been developed, including **simulated annealing**, which returns optimal solutions when given an appropriate cooling schedule.
- Many local search methods 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.
- A **genetic algorithm** is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by **mutation** and by **crossover**, which combines pairs of states from the population.

- 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 logical reasoning (Selman *et al.*, 1992) and by the incorporation of randomness, multiple simultaneous searches, and other improvements. This renaissance of what Christos Papadimitriou has called "New Age" algorithms also sparked increased interest among theoretical computer scientists (Koutsoupias and Papadimitriou, 1992; Aldous and Vazirani, 1994). In the field of operations research, a variant of hill climbing called **tabu search** has gained popularity (Glover 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 surface to the set of local maxima and then calculates the global maximum of that surface analytically. This becomes the new restart point. The algorithm has been shown to work in practice on hard problems. Gomes *et al.* (1998) showed that the run 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.

TABU SEARCH

HEAVY-TAILED DISTRIBUTION

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

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

As Andrew Moore points out, researchers have taken inspiration for search and optimization algorithms from a wide variety of fields of study: metallurgy (simulated annealing), biology (genetic algorithms), economics (market-based algorithms), entomology (ant colony optimization), neurology (neural networks), animal behavior (reinforcement learning), mountaineering (hill climbing), and others.

Linear programming (LP) was first studied systematically by the Russian 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 tool and as a method to expand our understanding of adaptation, biological or otherwise (Holland, 1995). The **artificial life** movement (Langton, 1995) takes this idea one step further, viewing the products of genetic algorithms as *organisms* rather than solutions to problems. Work in this field by Hinton and Nowlan (1987) and Ackley and Littman (1991) has done much to clarify the implications of the Baldwin effect. For general background on evolution, we recommend Smith and Szathmáry (1999), Ridley (2004), and Carroll (2007).

EVOLUTION STRATEGY

ARTIFICIAL LIFE

Most comparisons of genetic algorithms to other approaches (especially stochastic hill climbing) have found that the genetic algorithms are slower to converge (O'Reilly and Oppacher, 1994; Mitchell *et al.*, 1996; Juels and Wattenberg, 1996; Baluja, 1997). Such findings are not universally popular within the GA community, but recent attempts within that community to understand population-based search as an approximate form of Bayesian learning (see Chapter 20) might help close the gap between the field and its critics (Pelikan *et al.*, 1999). The theory of **quadratic dynamical systems** may also explain the performance of GAs (Rabani *et al.*, 1998). See Lohn *et al.* (2001) for an example of GAs applied to antenna design, and Renner and Ekart (2003) for an application to computer-aided design.

GENETIC PROGRAMMING

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

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

The journals *Evolutionary Computation* and *IEEE Transactions on Evolutionary Computation* cover genetic algorithms and genetic programming; articles are also found in *Complex Systems*, *Adaptive Behavior*, and *Artificial Life*. The main conference is the *Genetic and Evolutionary Computation Conference* (GECCO). Good overview texts on genetic algorithms are given by Mitchell (1996), Fogel (2000), and Langdon and Poli (2002), and by the free online book by 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, 1974). 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 and formalized by Nilsson (1971), who also described AO*—which, as its name suggests, finds optimal solutions given an admissible heuristic. AO* was analyzed and 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 has undergone a revival in recent years, 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.

EULERIAN GRAPH

Algorithms for exploring unknown state spaces have been of interest for many centuries. Depth-first search in a maze can be implemented by keeping one's left hand on the wall; loops can be avoided by marking each junction. Depth-first search fails with irreversible actions; the more general problem of exploring **Eulerian graphs** (i.e., graphs in which each node has equal numbers of incoming and outgoing edges) was solved by an algorithm due to Hierholzer (1873). The first thorough algorithmic study of the exploration problem for arbitrary graphs was carried out by Deng and Papadimitriou (1990), who developed a completely general algorithm but showed that no bounded competitive ratio is possible for exploring a general graph. Papadimitriou and Yannakakis (1991) examined the question of finding paths to a goal in geometric path-planning environments (where all actions are reversible). They showed that a small competitive ratio is achievable with square obstacles, but with general rectangular obstacles no bounded ratio can be achieved. (See Figure 4.20.)

REAL-TIME SEARCH

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

EXERCISES

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

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

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

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

d. Simulated annealing with $T = \infty$ at all times.

e. Genetic algorithm with population size $N = 1$.

4.2 Exercise 3.16 considers the problem of building railway tracks under the assumption that pieces fit exactly with no slack. Now consider the real problem, in which pieces don't fit exactly but allow for up to 10 degrees of rotation to either side of the "proper" alignment. Explain how to formulate the problem so it could be solved by simulated annealing.

4.3 In this exercise, we explore the use of local search methods to solve TSPs of the type defined in Exercise 3.30.

a. Implement and test a hill-climbing method to solve TSPs. Compare the results with optimal solutions obtained from the A* algorithm with the MST heuristic (Exercise 3.30).

b. Repeat part (a) using a genetic algorithm instead of hill climbing. You may want to consult Larrañaga *et al.* (1999) for some suggestions for representations.

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

4.5 The AND-OR-GRAPH-SEARCH algorithm in Figure 4.11 checks for repeated states only on the path from the root to the current state. Suppose that, in addition, the algorithm were to store *every* visited state and check against that list. (See BREADTH-FIRST-SEARCH in Figure 3.11 for an example.) Determine the information that should be stored and how the algorithm should use that information when a repeated state is found. (*Hint*: You will need to distinguish at least between states for which a successful subplan was constructed previously and states for which no subplan could be found.) Explain how to use labels, as defined in Section 4.3.3, to avoid having multiple copies of subplans.

4.6 Explain precisely how to modify the AND-OR-GRAPH-SEARCH algorithm to generate a cyclic plan if no acyclic plan exists. You will need to deal with three issues: labeling the plan steps so that a cyclic plan can point back to an earlier part of the plan, modifying OR-SEARCH so that it continues to look for acyclic plans after finding a cyclic plan, and augmenting the plan representation to indicate whether a plan is cyclic. Show how your algorithm works on (a) the slippery vacuum world, and (b) the slippery, erratic vacuum world. You might wish to use a computer implementation to check your results.

4.7 In Section 4.4.1 we introduced belief states to solve sensorless search problems. A sequence of actions solves a sensorless problem if it maps every physical state in the initial belief state b to a goal state. Suppose the agent knows $h^*(s)$, the true optimal cost of solving the physical state s in the fully observable problem, for every state s in b. Find an admissible heuristic $h(b)$ for the sensorless problem in terms of these costs, and prove its admissibilty. Comment on the accuracy of this heuristic on the sensorless vacuum problem of Figure 4.14. How well does A* perform?

4.8 This exercise explores subset–superset relations between belief states in sensorless or partially observable environments.

a. Prove that if an action sequence is a solution for a belief state b, it is also a solution for any subset of b. Can anything be said about supersets of b?

b. Explain in detail how to modify graph search for sensorless problems to take advantage of your answers in (a).

c. Explain in detail how to modify AND–OR search for partially observable problems, beyond the modifications you describe in (b).

4.9 On page 139 it was assumed that a given action would have the same cost when executed in any physical state within a given belief state. (This leads to a belief-state search problem with well-defined step costs.) Now consider what happens when the assumption does not hold. Does the notion of optimality still make sense in this context, or does it require modification? Consider also various possible definitions of the "cost" of executing an action in a belief state; for example, we could use the *minimum* of the physical costs; or the *maximum*; or a cost *interval* with the lower bound being the minimum cost and the upper bound being the maximum; or just keep the set of all possible costs for that action. For each of these, explore whether A* (with modifications if necessary) can return optimal solutions.

4.10 Consider the sensorless version of the erratic vacuum world. Draw the belief-state space reachable from the initial belief state $\{1, 2, 3, 4, 5, 6, 7, 8\}$, and explain why the problem is unsolvable.

4.11 We can turn the navigation problem in Exercise 3.7 into an environment as follows:

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

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

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

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

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

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

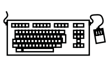
4.12 Suppose that an agent is in a 3×3 maze environment like the one shown in Figure 4.19. The agent knows that its initial location is (1,1), that the goal is at (3,3), and that the actions Up, $Down$, $Left$, $Right$ have their usual effects unless blocked by a wall. The agent does *not* know where the internal walls are. In any given state, the agent perceives the set of legal actions; it can also tell whether the state is one it has visited before.

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

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

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

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



4.13 In this exercise, we examine hill climbing in the context of robot navigation, using the environment in Figure 3.31 as an example.

a. Repeat Exercise 4.11 using hill climbing. Does your agent ever get stuck in a local minimum? Is it *possible* for it to get stuck with convex obstacles?

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

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

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

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

4.14 Like DFS, online DFS is incomplete for reversible state spaces with infinite paths. For example, suppose that states are points on the infinite two-dimensional grid and actions are unit vectors $(1, 0)$, $(0, 1)$, $(-1, 0)$, $(0, -1)$, tried in that order. Show that online DFS starting at $(0, 0)$ will not reach $(1, -1)$. Suppose the agent can observe, in addition to its current state, all successor states and the actions that would lead to them. Write an algorithm that is complete even for bidirected state spaces with infinite paths. What states does it visit in reaching $(1, -1)$?