北京大學

超越经典搜索
Beyond Classical Search

·AIPKU·

人工智能引论第五课

主讲人：李文新

前言

- Classical Searching problems: observable, deterministic, known environments where the solution is a sequence of actions.

- Now, we look at what happens when these assumptions are relaxed.

- 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**).

前言

- 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.

内容提要

- 局部搜索算法和最优化问题
  - 爬山法、模拟退火搜索、局部束搜索、遗传算法
- 连续空间中的局部搜索
- 使用不确定动作的搜索
  - 不稳定的吸尘器世界、与或搜索树、不断尝试
- 使用部分可观察信息的搜索
  - 无观察信息的搜索、有观察信息的搜索、求解部分可观察环境中的问题、部分可观察环境中的Agent
- 联机搜索Agent和未知环境
  - 联机搜索问题、联机搜索Agent、联机局部搜索、联机搜索中的学习

内容提要 – 局部搜索算法和最优化问题

- 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, **the path to the goal is irrelevant**. For example, in the 8-queens problem, 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.

- **Local search** algorithms operate using a single **current node** (rather than multiple paths) and generally move only to neighbors of that node.

内容提要 – 局部搜索算法和最优化问题

- 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.

- 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.

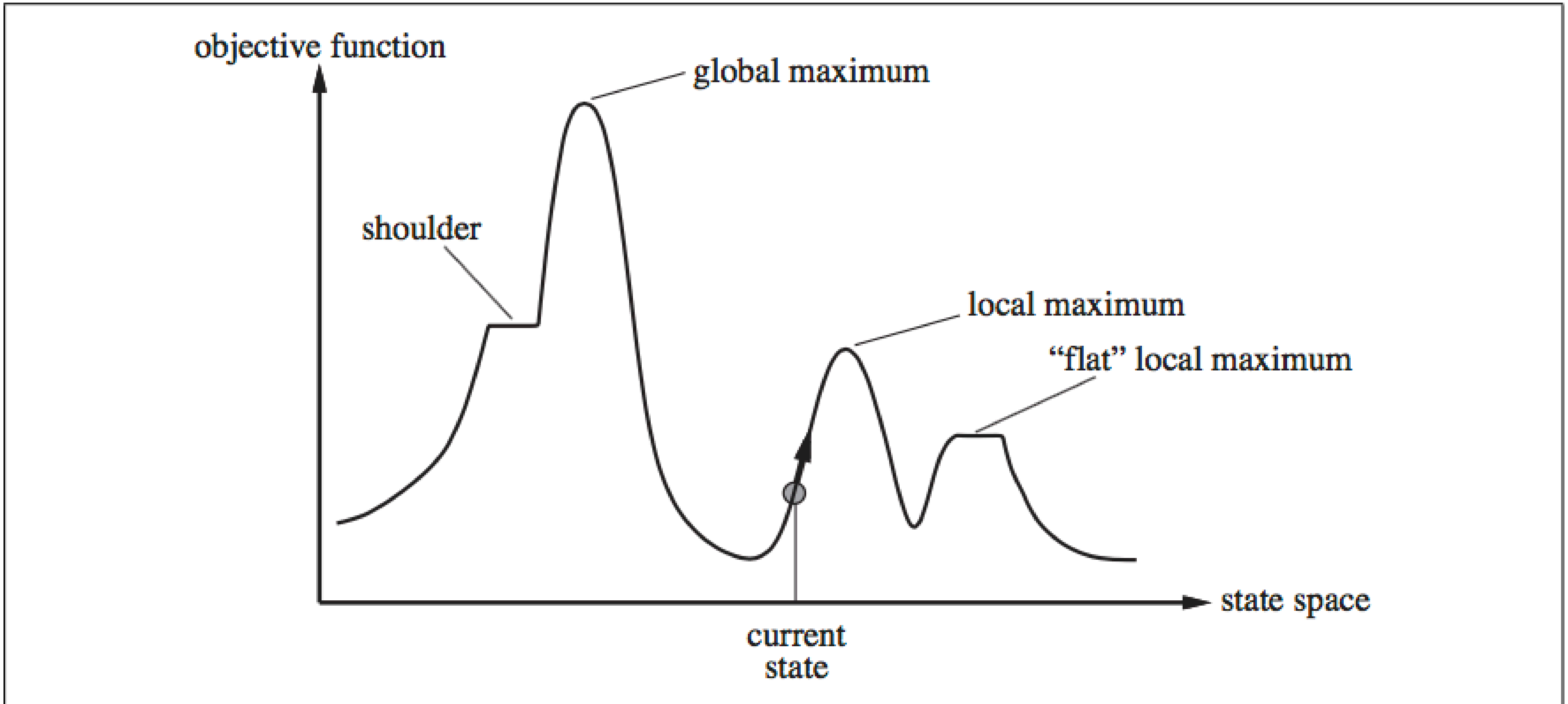
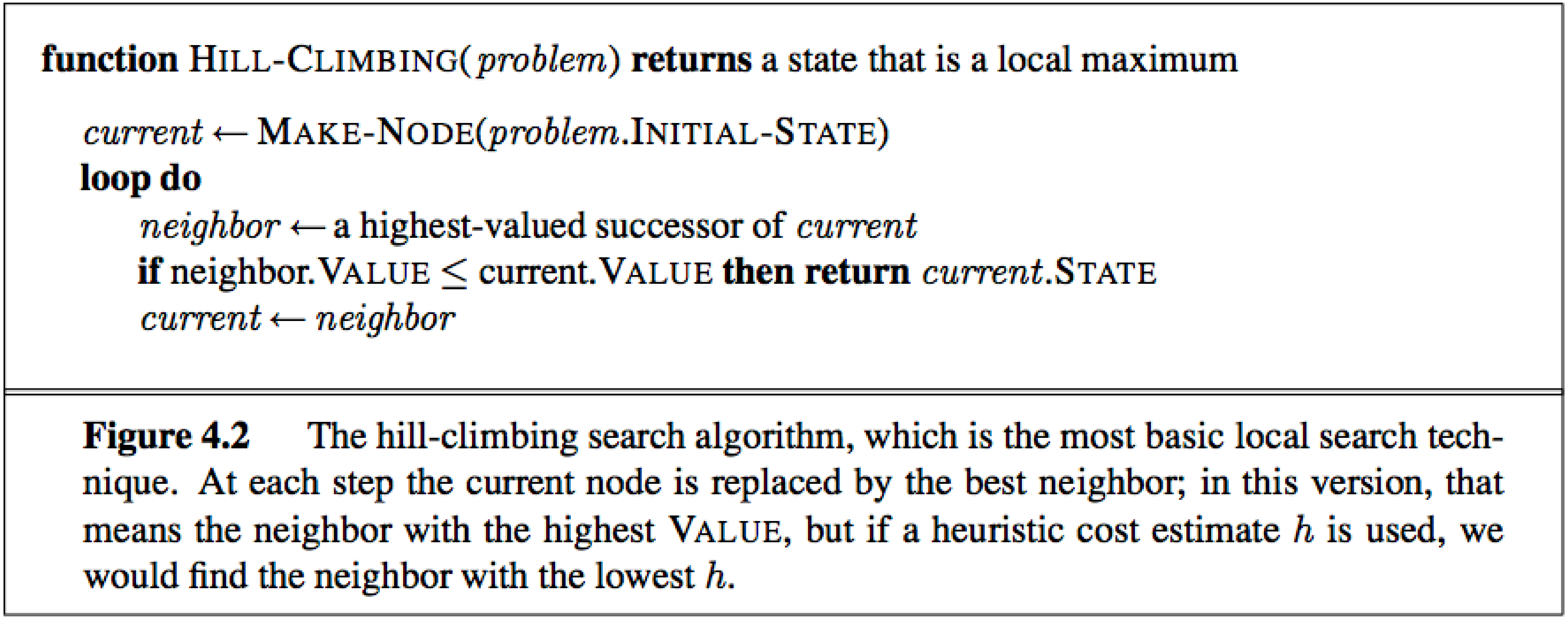


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

- A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.

内容提要 – 局部搜索算法和最优化问题

爬山法The **hill-climbing** search algorithm (**steepest-ascent** version). 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.



```
**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 $\leq$ 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.

内容提要 – 局部搜索算法和最优化问题

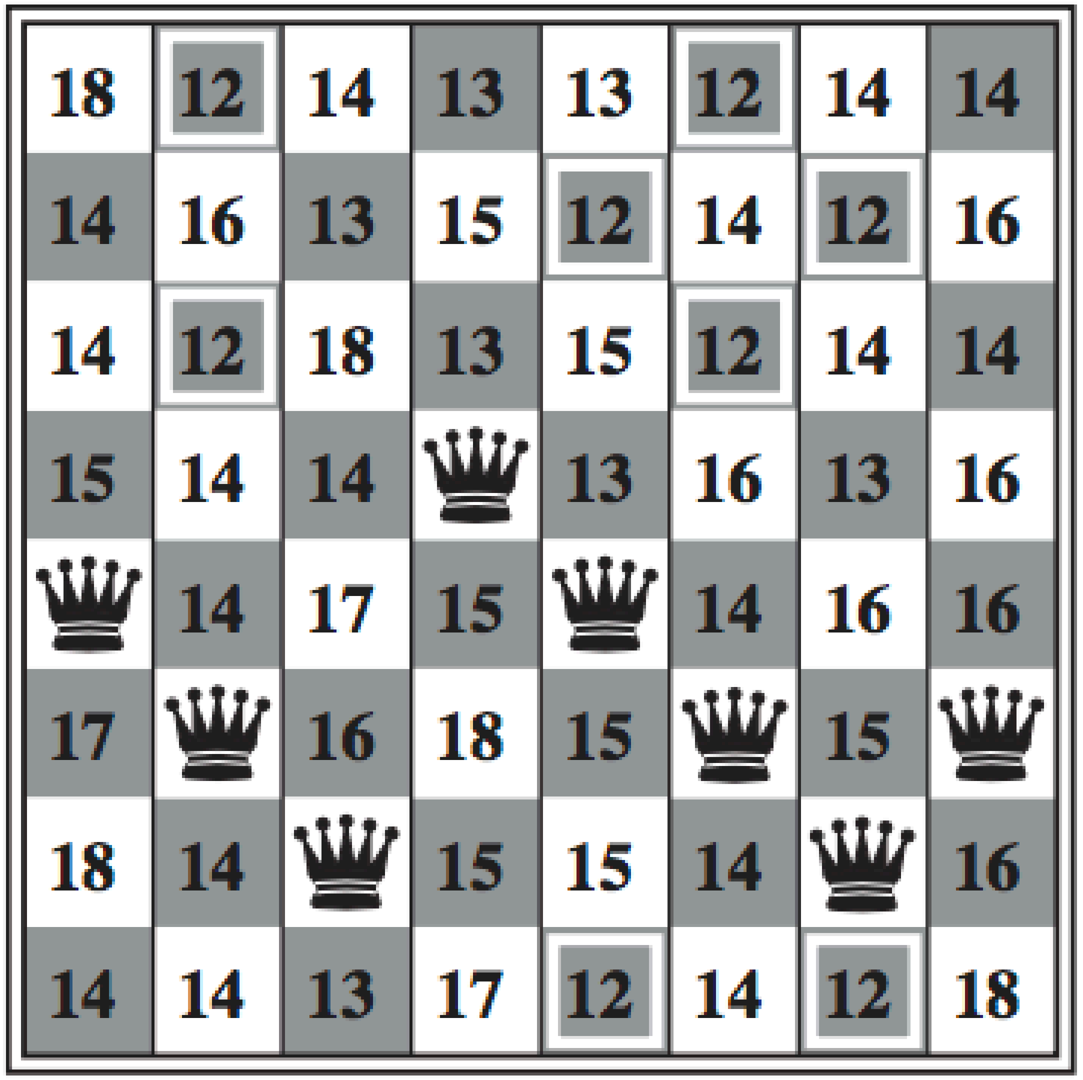
- **8-queens problem** ，**complete-state formulation**， 8 queens on the board , one per column

- The heuristic cost function **h** is the number of pairs of queens that are attacking each other, either directly or indirectly.

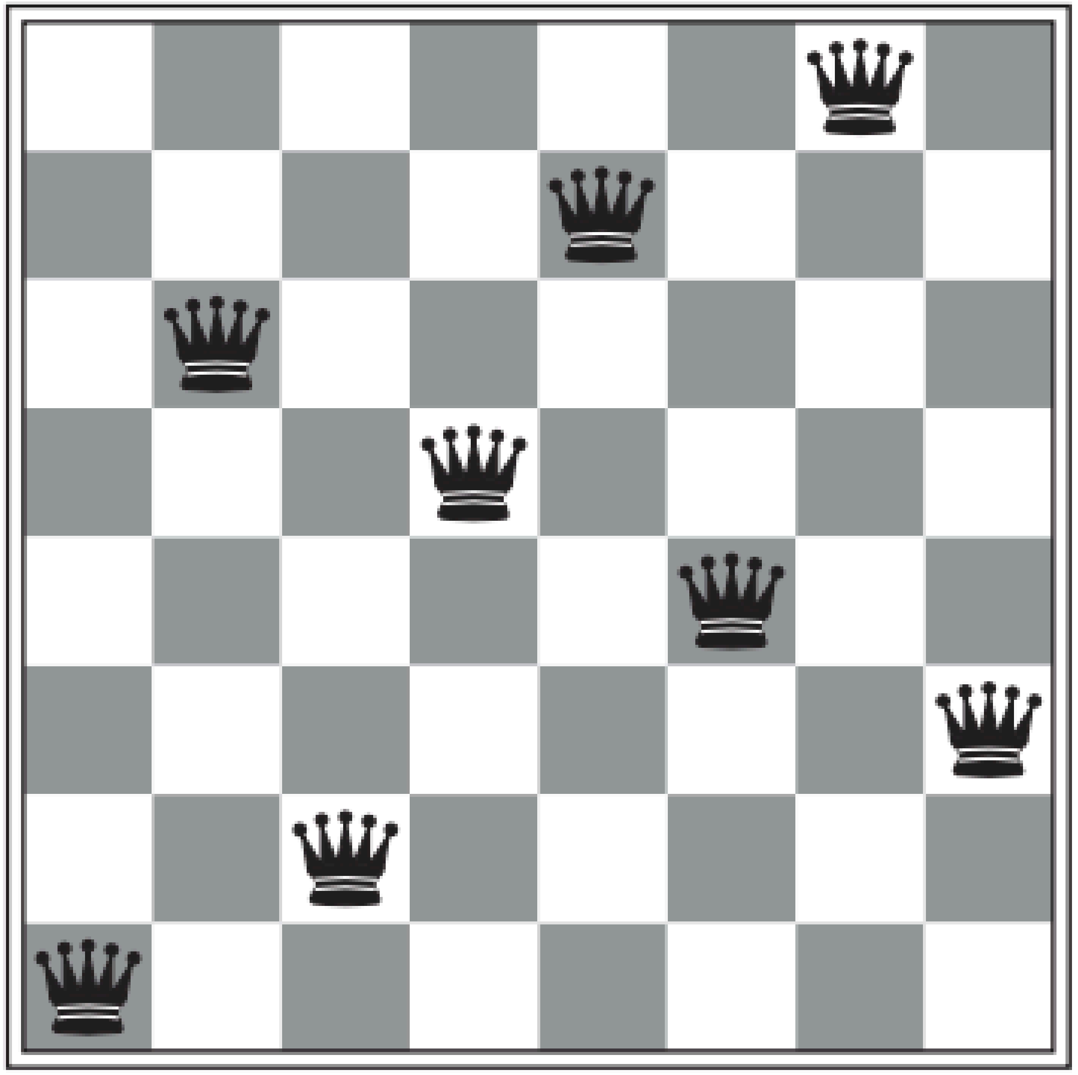
- moving a single queen to another square in the same column

- each state has 8 × 7 = 56 successors

内容提要 – 局部搜索算法和最优化问题



(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.

内容提要 – 局部搜索算法和最优化问题

- **Local maxima**
- **Ridges**
- **Plateaux:**

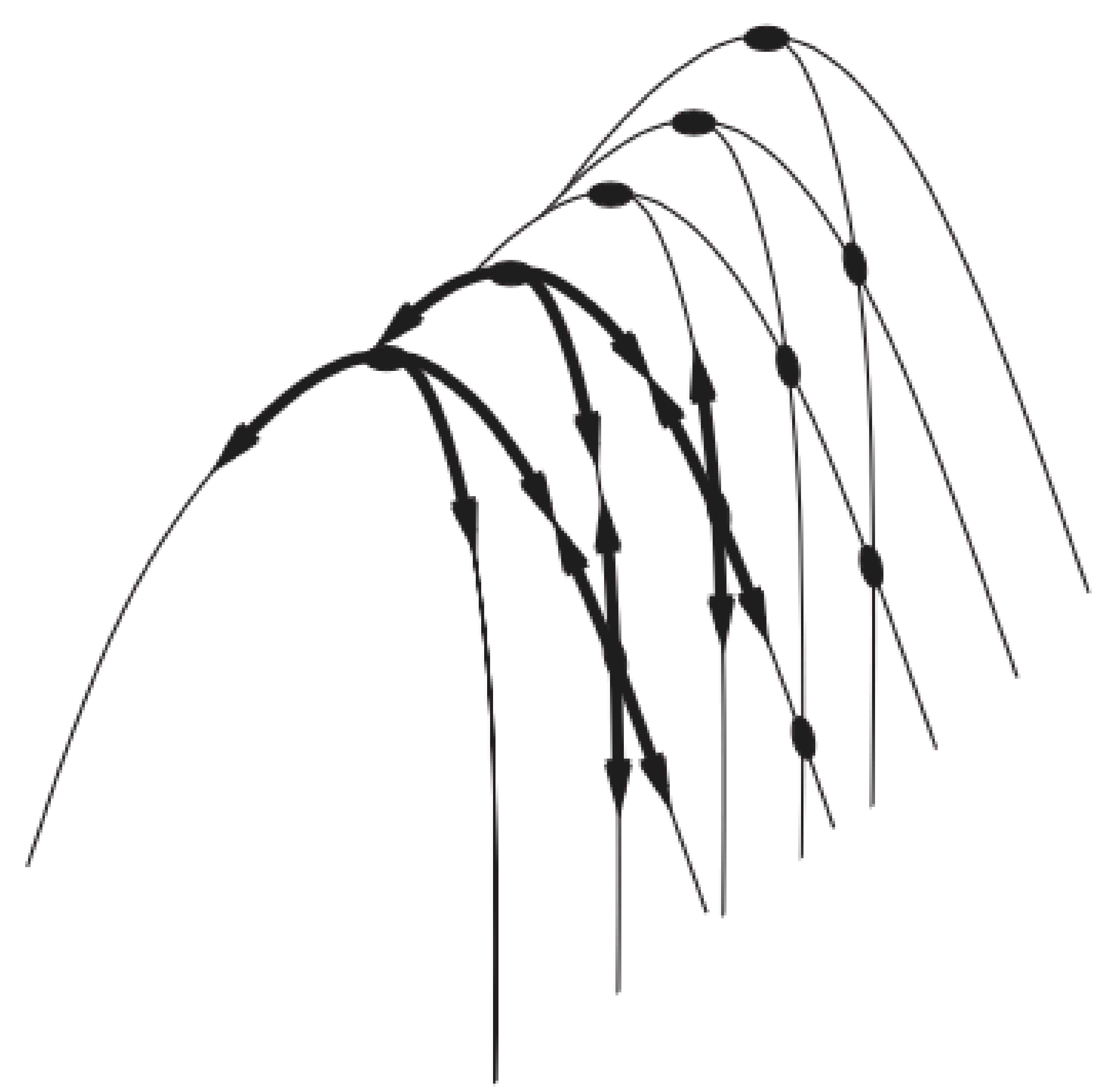


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point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

内容提要 – 局部搜索算法和最优化问题

- 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 88 ≈ 17 million states.

- to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 4.1? 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.

内容提要 – 局部搜索算法和最优化问题

- 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.

- 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 con- ducts 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.

内容提要 – 局部搜索算法和最优化问题

- 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 ≈ 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 ≈ 1.06 iterations are needed on average and (1 × 21) + (0.06/0.94) × 64 ≈ 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.

内容提要 – 局部搜索算法和最优化问题

- 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.

内容提要 – 模拟退火搜索

- 模拟退火搜索 **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.

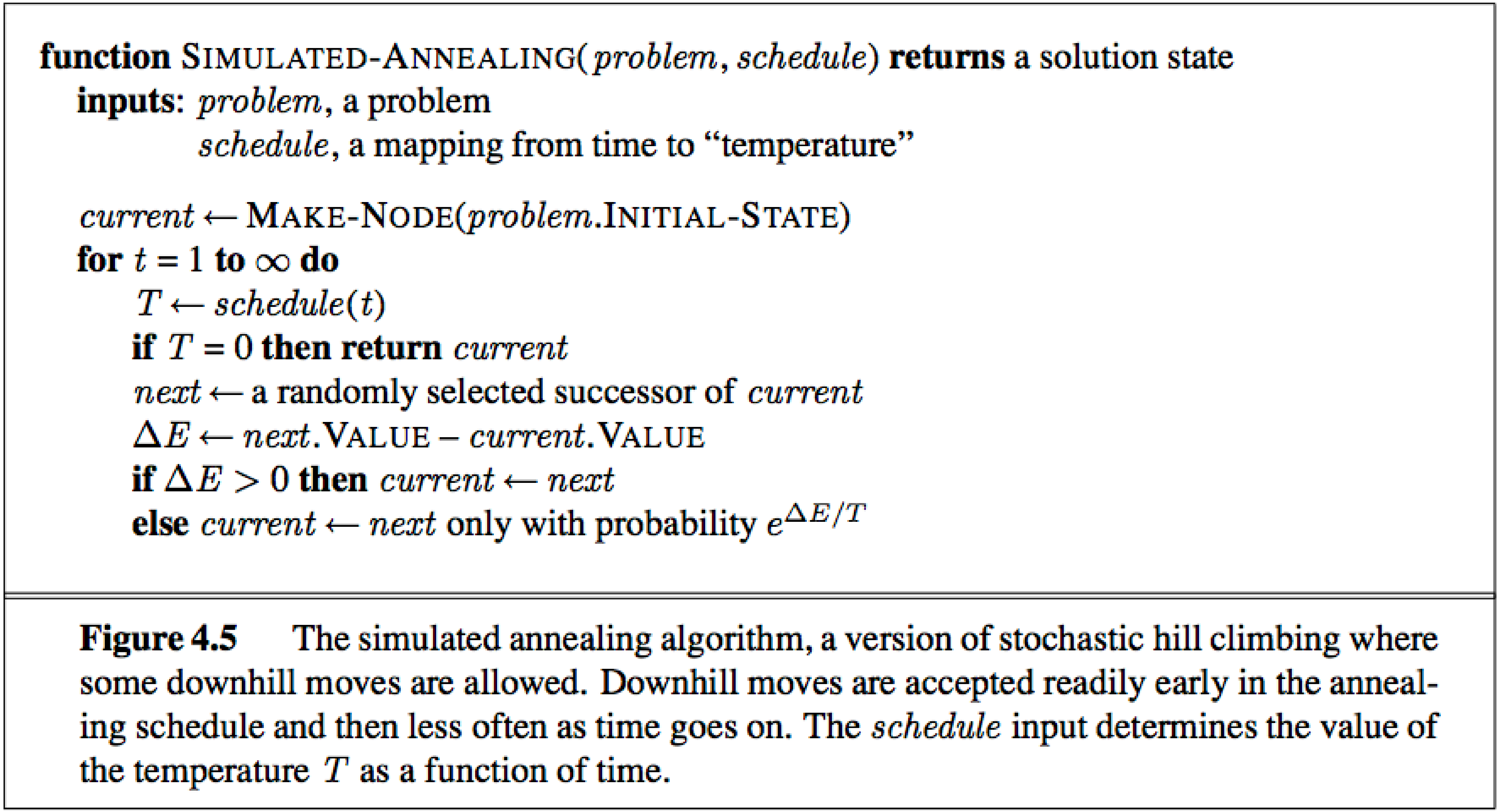
- Therefore, it seems reasonable to try to **combine hill climbing with a random walk** in some way that yields **both efficiency and completeness**. **Simulated annealing** is such an algorithm.

内容提要 – 模拟退火搜索

- **Two ideas**

- 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.

- A ping-pong ball. 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.



```
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 $\infty$ do
      $T \leftarrow$ schedule($t$)
      if $T = 0$ then return current
      next ← a randomly selected successor of current
      $\Delta E \leftarrow$ next.VALUE – current.VALUE
      if $\Delta E > 0$ then current ← next
      else current ← next only with probability $e^{\Delta E/T}$
```

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.

- 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.

内容提要 – 模拟退火搜索

- 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.

内容提要 – 局部束搜索

- 局部束搜索 **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** algorithm3 keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

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

内容提要 – 局部束搜索

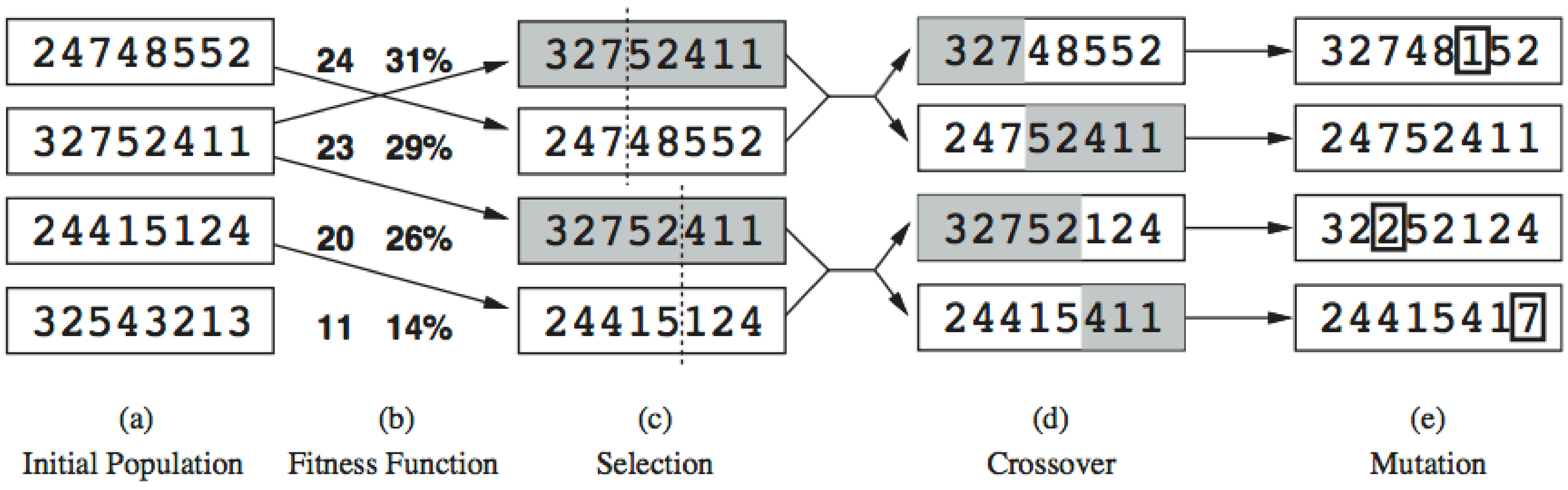
- 局部束搜索 **Local beam search**

- 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).

内容提要 – 遗传算法

- 遗传算法 **Genetic algorithms**

- 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.



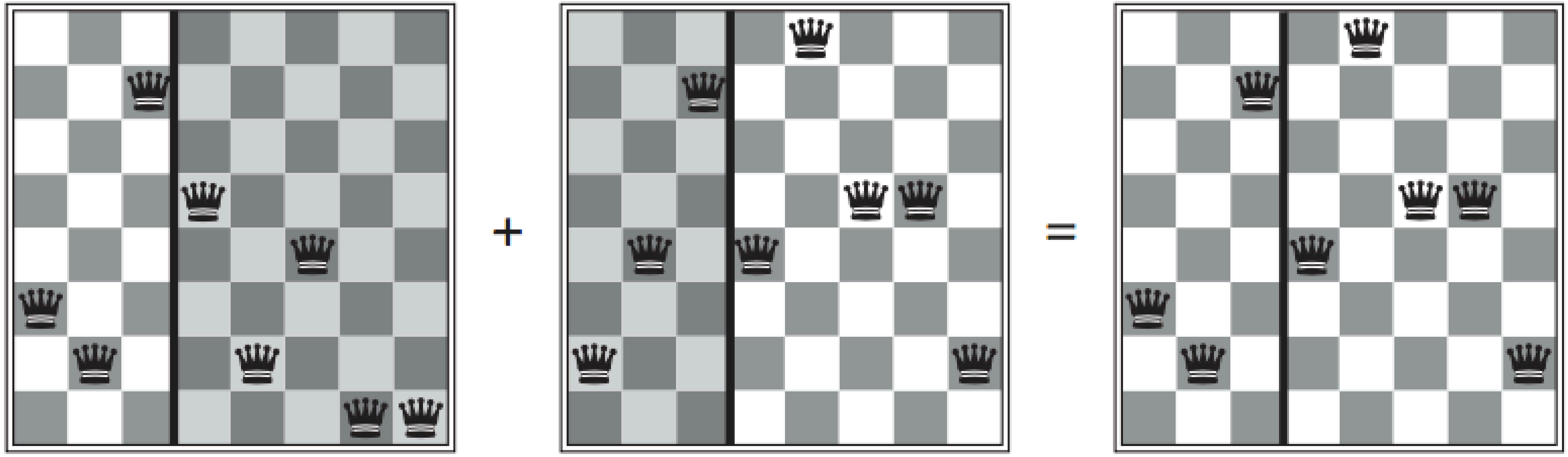
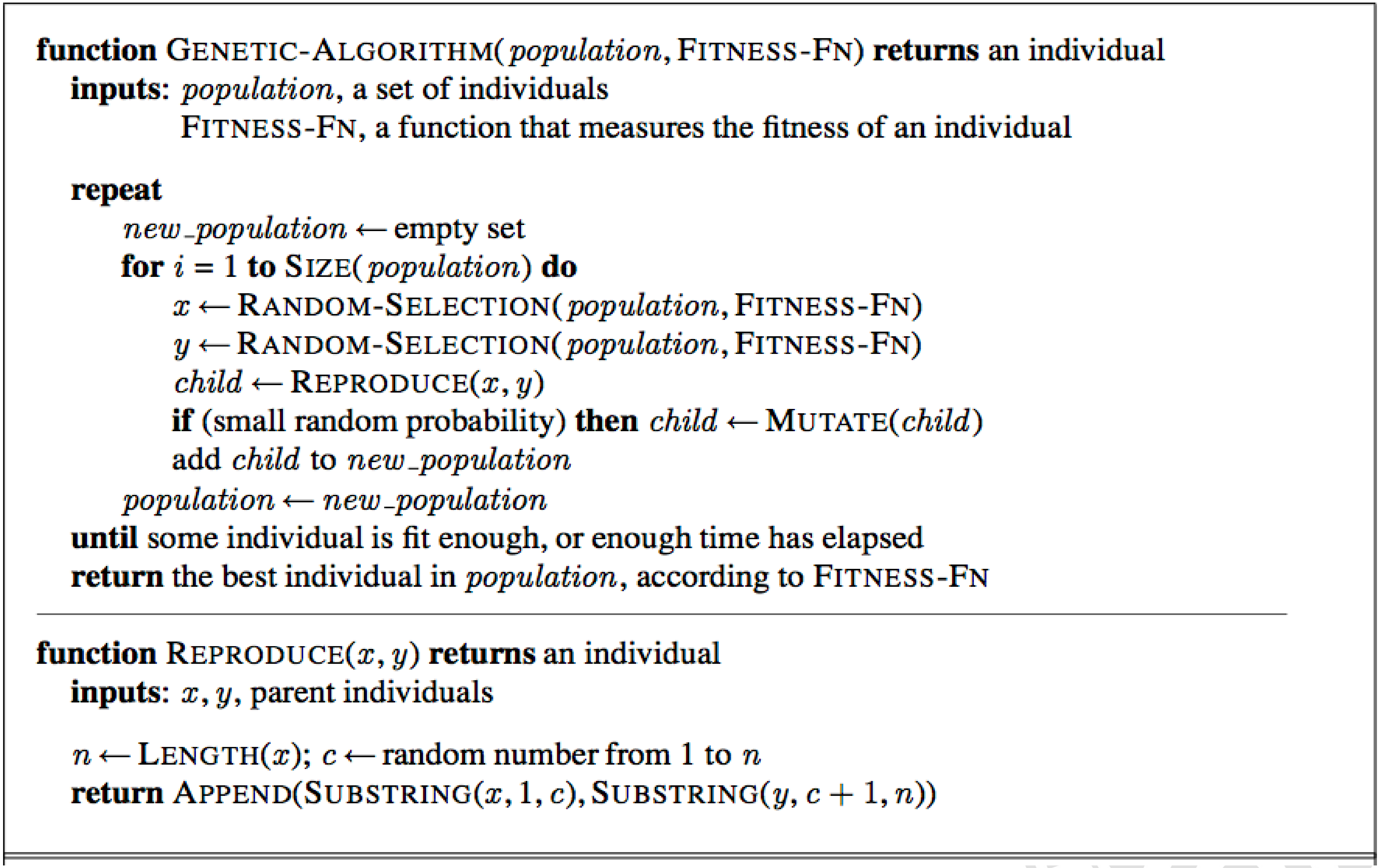


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.



```
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))
```

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.

内容提要 – 遗传算法

- 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.
- 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.
- Genetic algorithms work best when schemata correspond to meaningful components of a solution.
- This suggests that successful use of genetic algorithms requires careful engineering of the representation.

内容提要 – 遗传算法

- 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.

内容提要 – 连续空间中的局部搜索

- This section provides a *very brief* introduction to some local search techniques for **finding optimal solutions in continuous spaces**. The literature on this topic is vast; many of the basic techniques originated in the 17th century, after the development of calculus by Newton and Leibniz. 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: (x1,y1), (x2,y2), and (x3,y3). This is a *six-dimensional* space; we also say that states are defined by six **variables**.

内容提要 – 连续空间中的局部搜索

- 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 Ci 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

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

- 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 ±δ. With 6 variables, this gives 12 possible successors for each state. We can then apply any of the local search algorithms described previously.

内容提要 – 连续空间中的局部搜索

- 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.

- 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

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

内容提要 – 连续空间中的局部搜索

- In some cases, we can find a maximum by solving the equation ∇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,

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

内容提要 – 连续空间中的局部搜索

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

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

- where 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.

内容提要 – 连续空间中的局部搜索

- 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.

- 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

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

内容提要 – 连续空间中的局部搜索

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

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

- where $\mathbf{H}_f(\mathbf{x})$ is the **Hessian** matrix of second derivatives, whose elements H_{ij} are given by ∂2f/∂x ∂x . For our airport example, we can see from Equation (4.2) that **H** (**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 Ci. A moment’ s calculation shows that one step of the update moves airport i directly to the centroid of Ci, which is the minimum of the local expression for f from Equation (4.1).7 For high-dimensional problems, however, computing the n2 entries of the Hessian and inverting it may be expensive, so many approximate versions of the Newton–Raphson method have been developed.

内容提要 – 连续空间中的局部搜索

- 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.

内容提要 – 连续空间中的局部搜索

- 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 site 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 con- straints 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.

内容提要 – 连续空间中的局部搜索

- 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).

内容提要 – 连续空间中的局部搜索

- 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.

内容提要 – 使用不确定动作的搜索

- 不稳定的吸尘器世界
- 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.
- 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.

不稳定的吸尘器世界

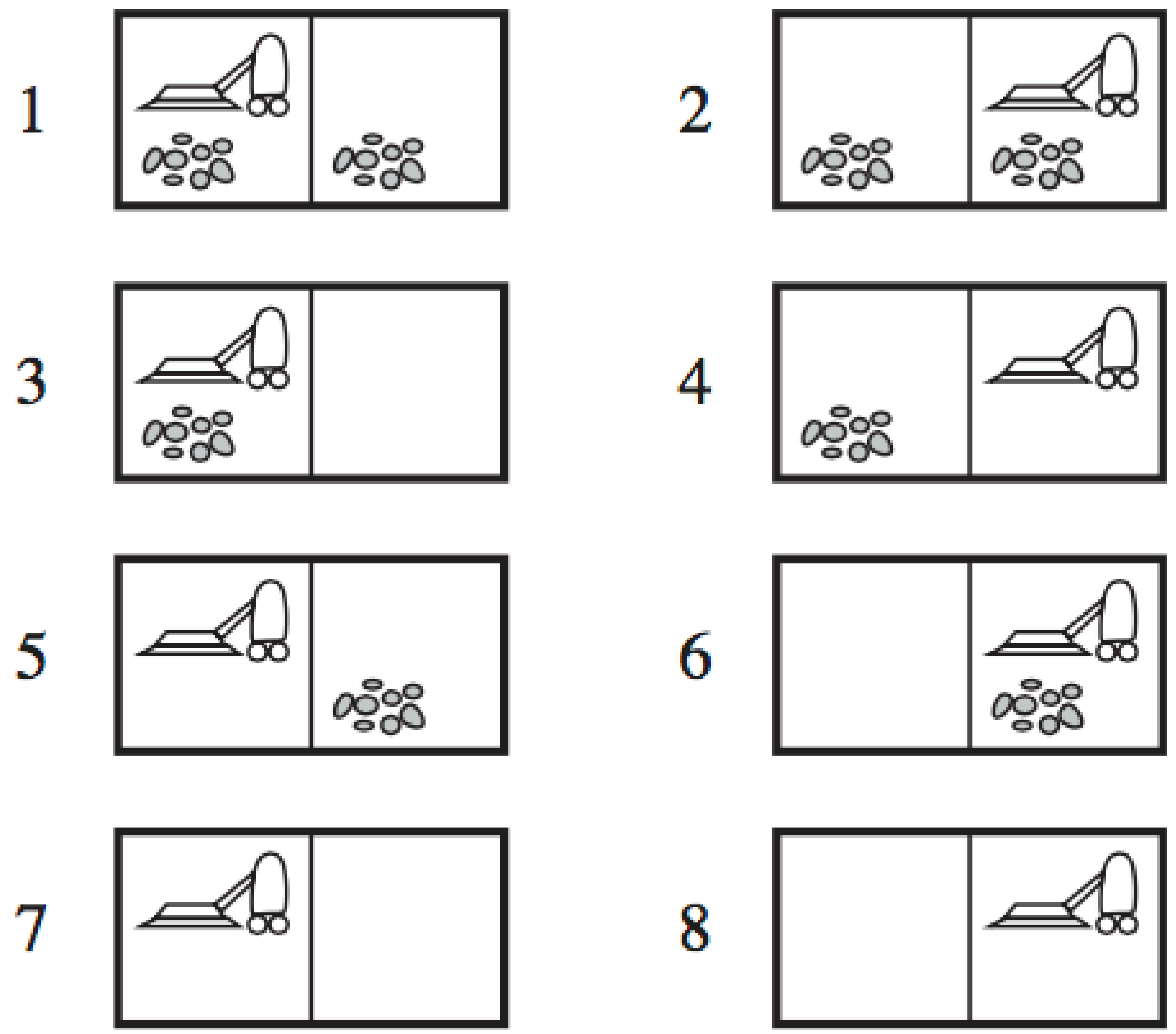


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

- 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.

不稳定的吸尘器世界

- 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:

- [*Suck*, **if** State = 5 **then** [*Right*, *Suck*] **else** [ ]] .

- 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.

AND–OR search trees

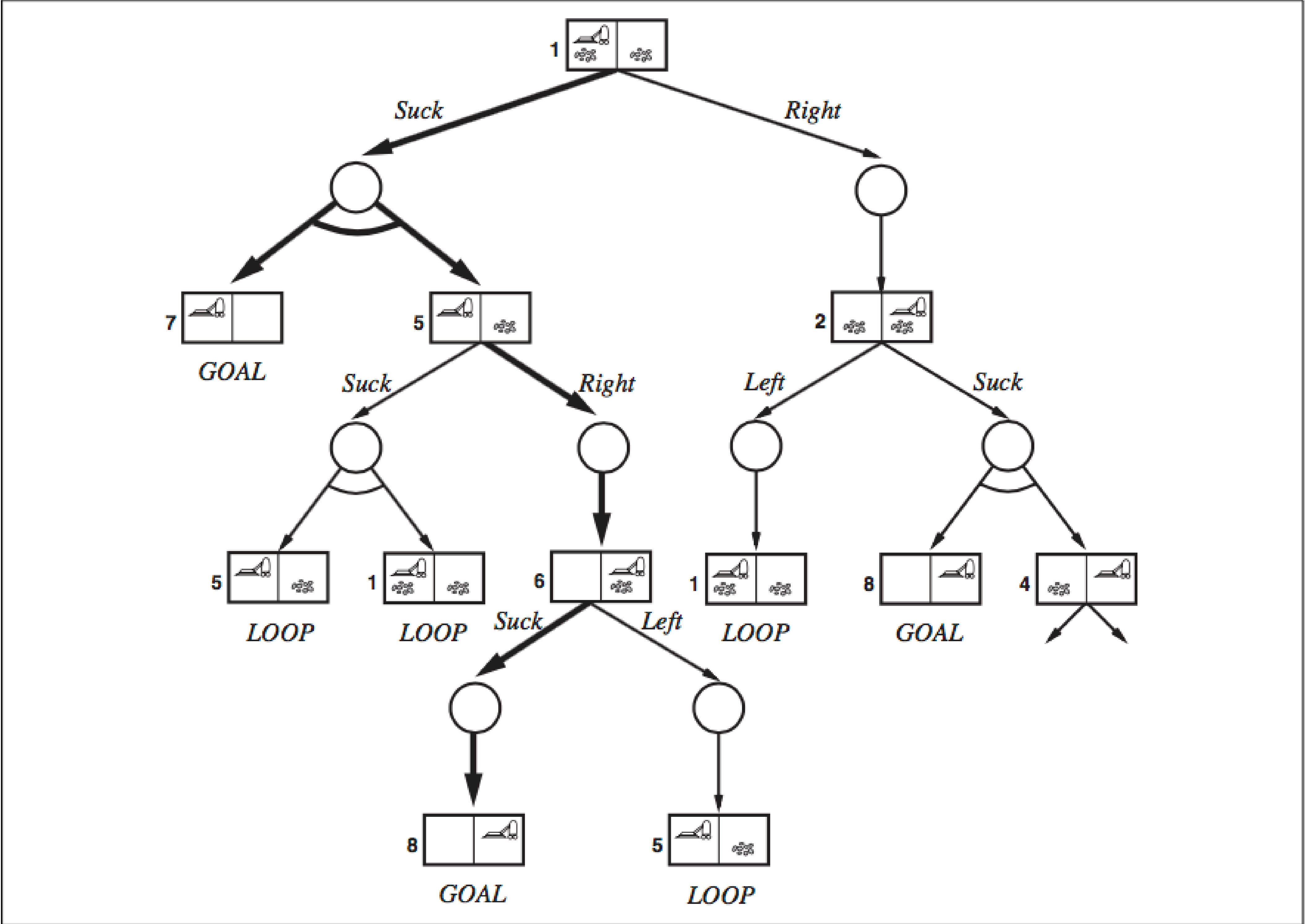
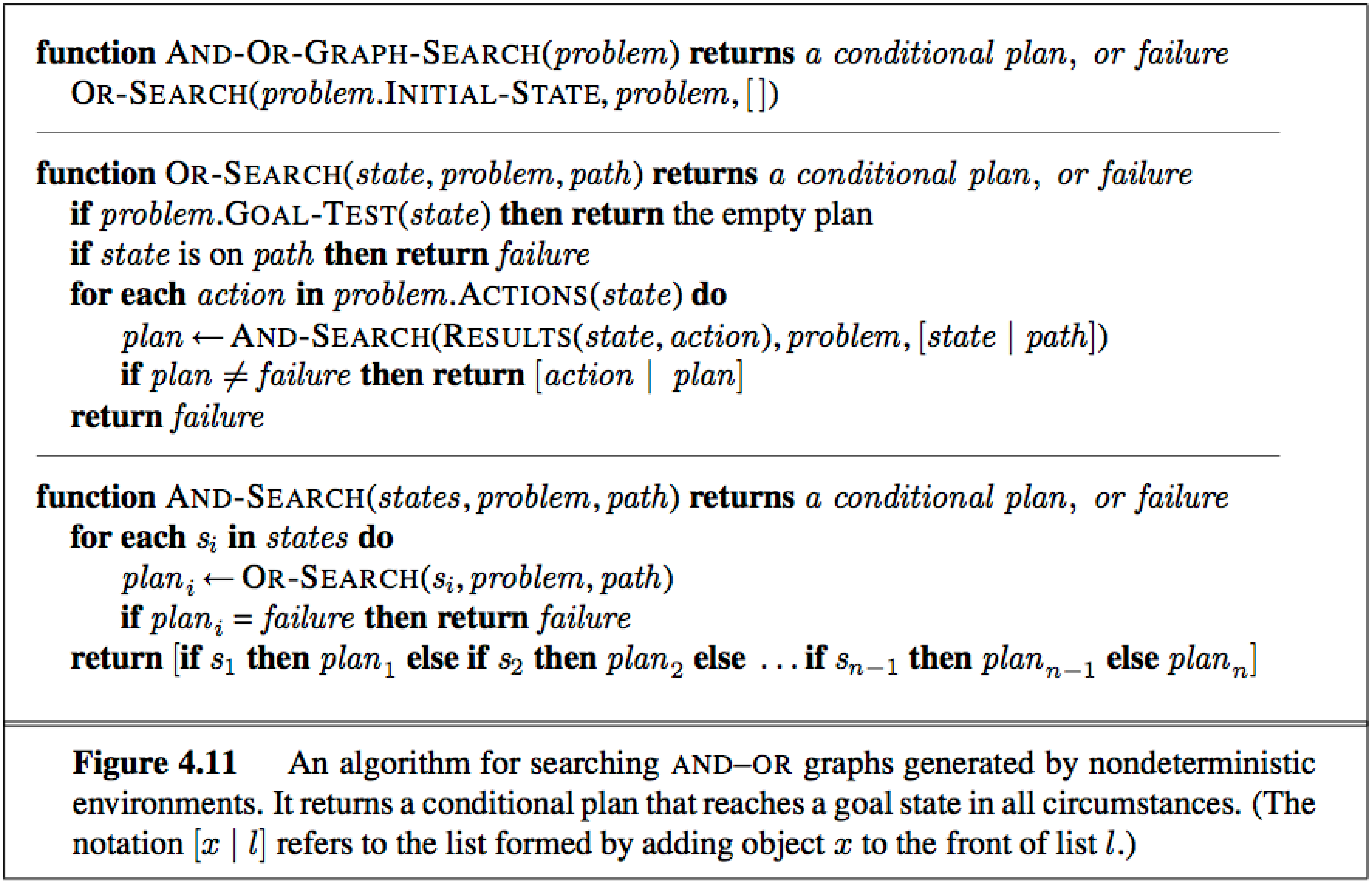


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

AND–OR search trees



```
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.)

AND–OR **search trees**

- Figure 4.11 gives a recursive, depth-first algorithm for AND–OR graph search.
- If the current state is identical to a state on the path from the root, then it returns with failure.
- 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.

Try , try again

- 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;
- [*Suck*, L1 : Right, **if** State = 5 **then** L1 **else** Suck] . (A better syntax for the looping part of this plan would be "**while** State=5 **do** Right.")

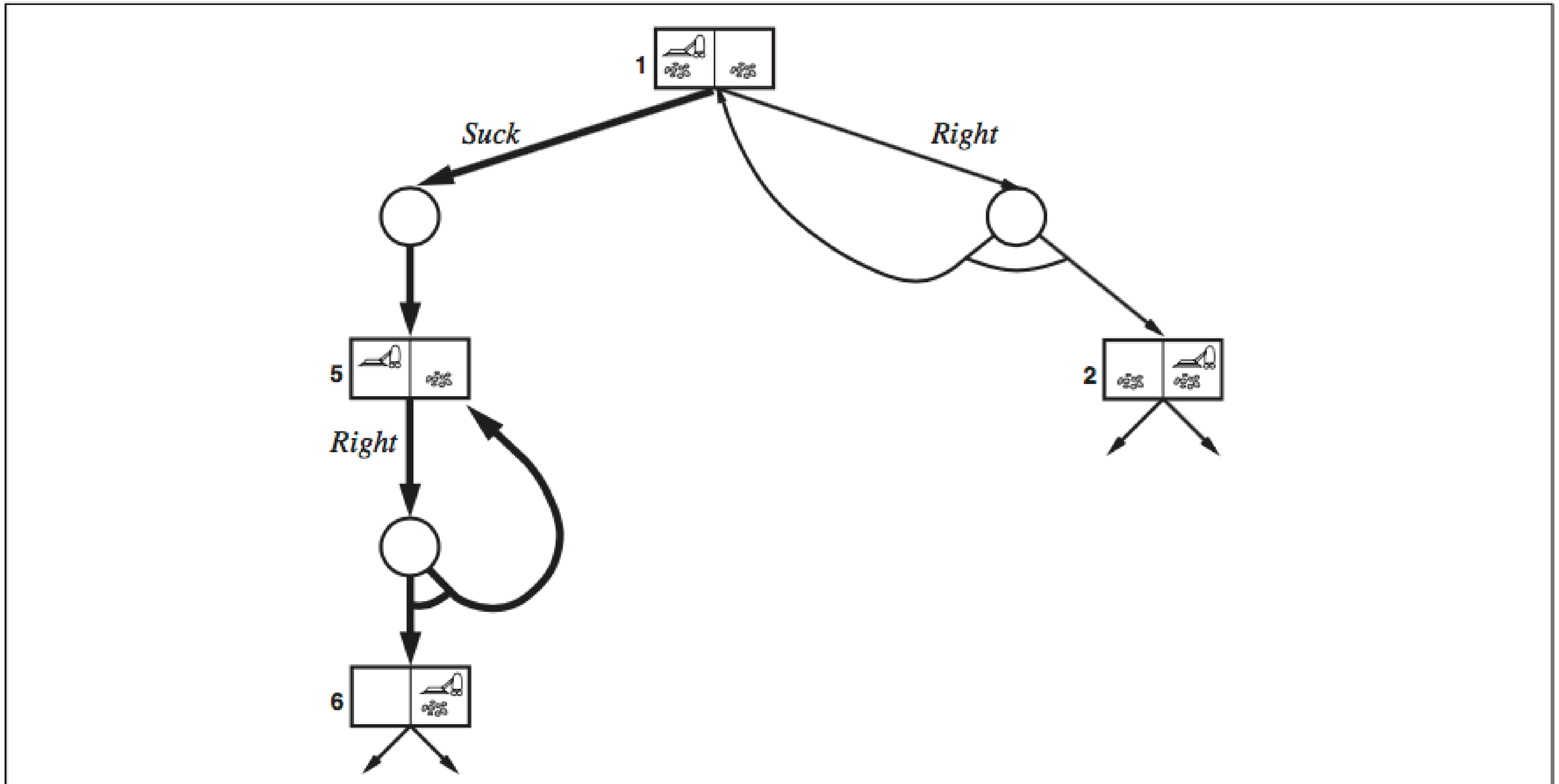


Figure 4.12 Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

内容提要 – 无观察信息的搜索

- 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.

内容提要 – 无观察信息的搜索

- **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 = {s1, s2}, but ACTIONSP (s1) /= ACTIONSP (s2); 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:

- ACTIONS(b) = ACTIONSP (s) . s∈b

- 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.

内容提要 – 无观察信息的搜索

- **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′ = RESULT(b,a) = {s′ : s′ =RESULTP(s,a) and s ∈ b} . (4.4) With deterministic actions, b′ is never larger than b. With nondeterminism, we have

- b′ = RESULT(b,a) = {s′ : s′ ∈ RESULTSP(s,a) and s ∈ b} = RESULTSP(s,a), s∈b

- 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′ = PREDICTP (b, a) will come in handy.

- **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-TESTP . The agent may *accidentally* achieve the goal earlier, but it won’ t *know* that it has done so.

内容提要 – 无观察信息的搜索

- **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. 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.

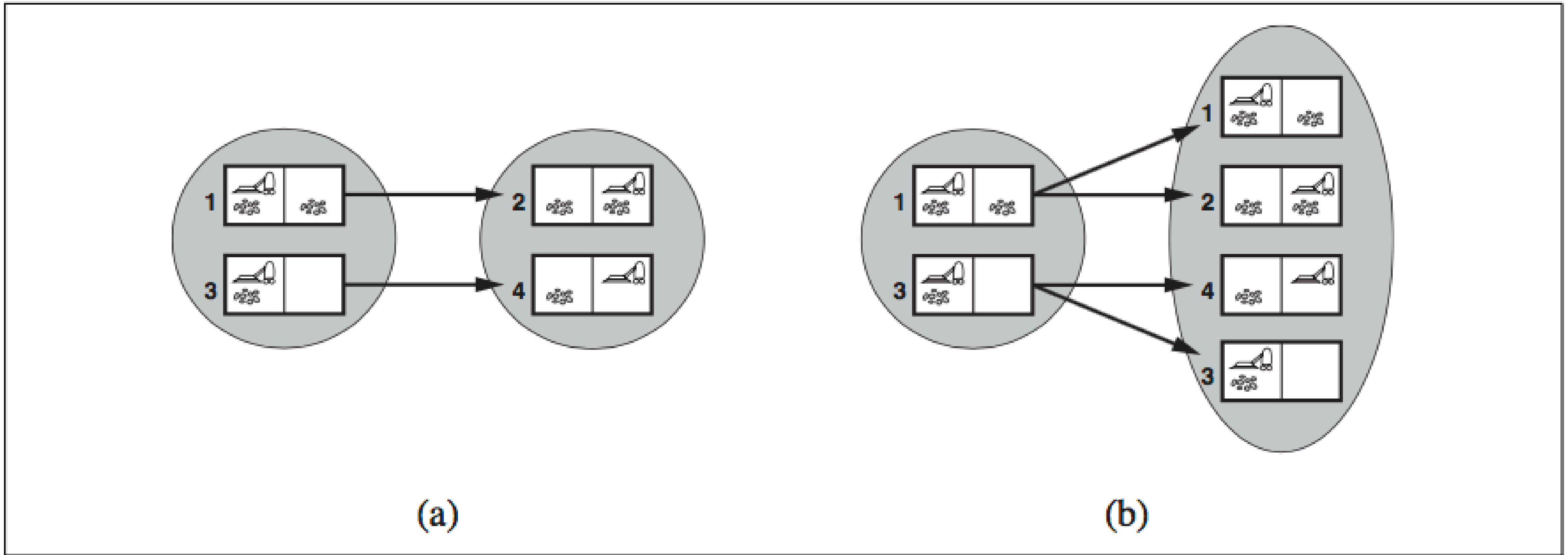


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

·AIPK

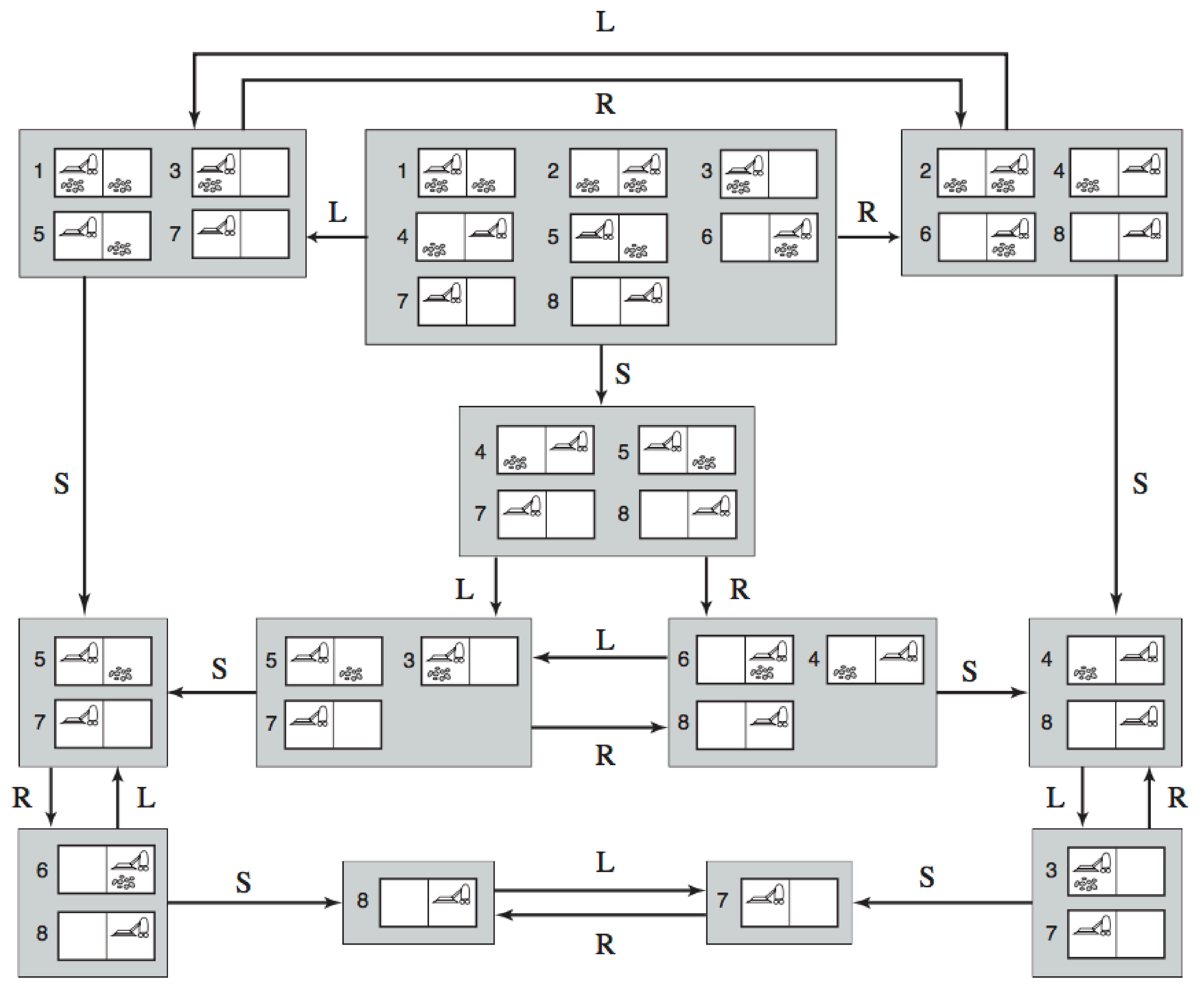


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete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

内容提要 – 无观察信息的搜索

- we can apply any of the search algorithms of Chapter 3.
- if an action sequence is a solution for a belief state b, it is also a solution for any subset of b.
- 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.
- 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 × 2100 or around 1032 physical states—far too many if we use the atomic representation, which is an explicit list of states.
- we can look *inside* the belief states and develop **incremental belief-state search** algorithms that build up the solution one physical state at a time.

内容提要 – 有观察信息的搜索

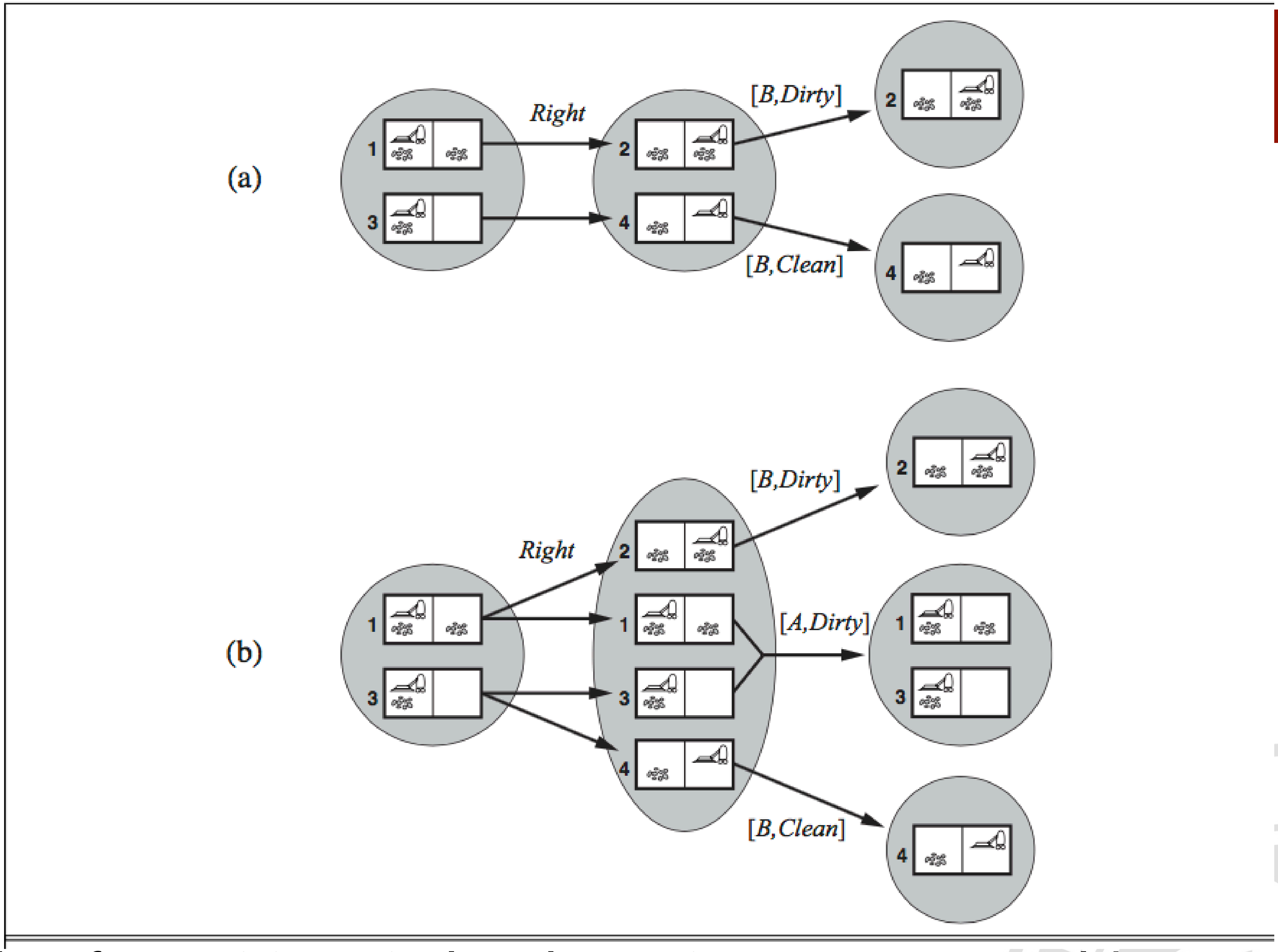
- For example, every 8-puzzle instance is solvable if just one square is visible—the solution involves moving each tile in turn into the visible square and then keeping track of its location.

- The formal problem specification includes a PERCEPT(s) function that returns the percept received in a given state.

- 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.

内容提要 – 有观察信息的搜索

- The **prediction** stage is the same as for sensorless problems: given the action a in belief state b, the predicted belief state is ˆb = PREDICT(b, a).11
- The **observation prediction** stage determines the set of percepts o that could be ob- served in the predicted belief state:
- P O S S I B L E - P E R C E P T S (ˆb) = { o : o = P E R C E P T (s) a n d s ∈ ˆb } .
- 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 ˆb that could have produced the percept:
- b_o =UPDATE(ˆb,o)={s:o=PERCEPT(s)ands∈ˆb}.
- Putting these three stages together, we obtain the possible belief states resulting from a given action and the subsequent possible percepts:
- RESULTS(b, a) = {bo : bo = UPDATE(PREDICT(b, a), o) and o ∈ POSSIBLE-PERCEPTS(PREDICT(b, a))} .



- 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 physi- cal states; for those states, the possible percepts are [A, Dirty], [B, Dirty], and [B, Clean], leading to three belief states as shown.

内容提要 – 求解部分可观察环境中的问题

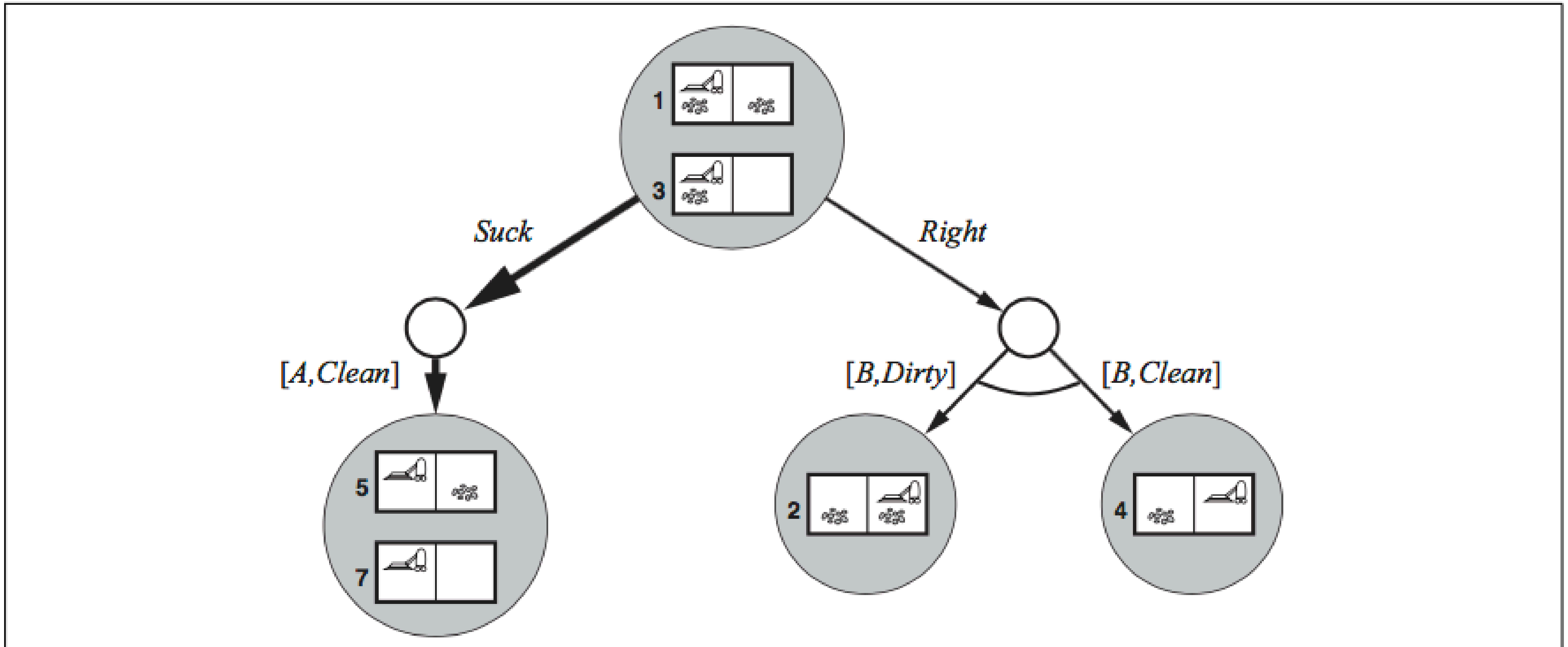


Figure 4.16 The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first step of the solution.

- Figure 4.16 shows part of the search tree for the local-sensing vacuum world, assuming an initial percept [A, Dirty]. The solution is the conditional plan
- [*Suck*, *Right*, **if** Bstate = {6} **then** *Suck* **else** [ ]] .

内容提要 – 部分可观察环境中的Agent

- 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.

内容提要 – 部分可观察环境中的Agent

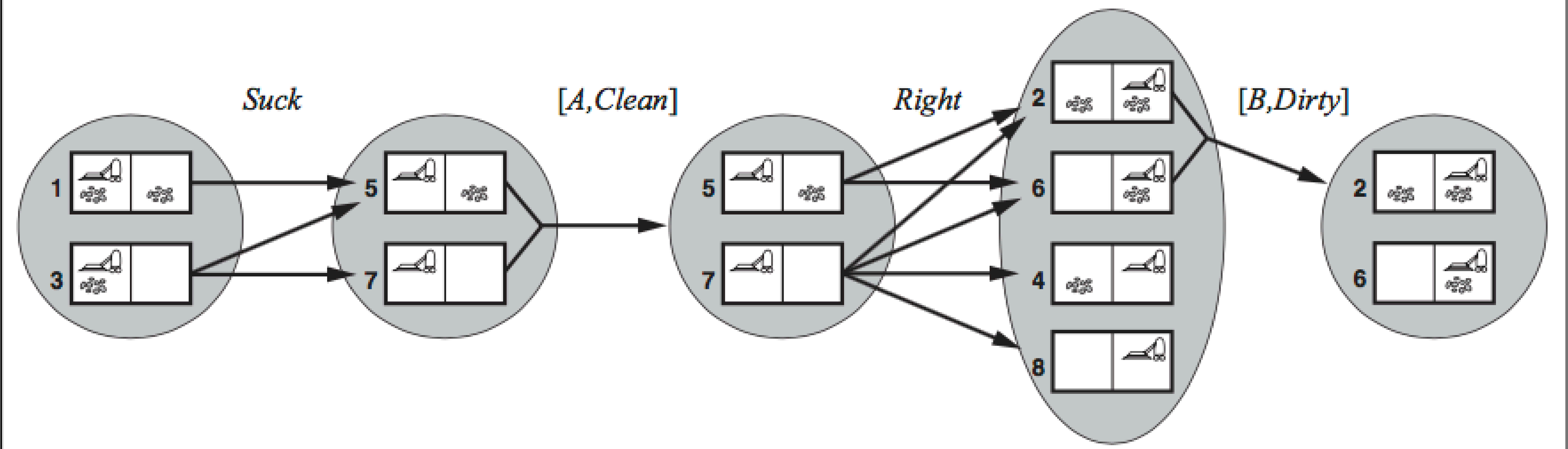
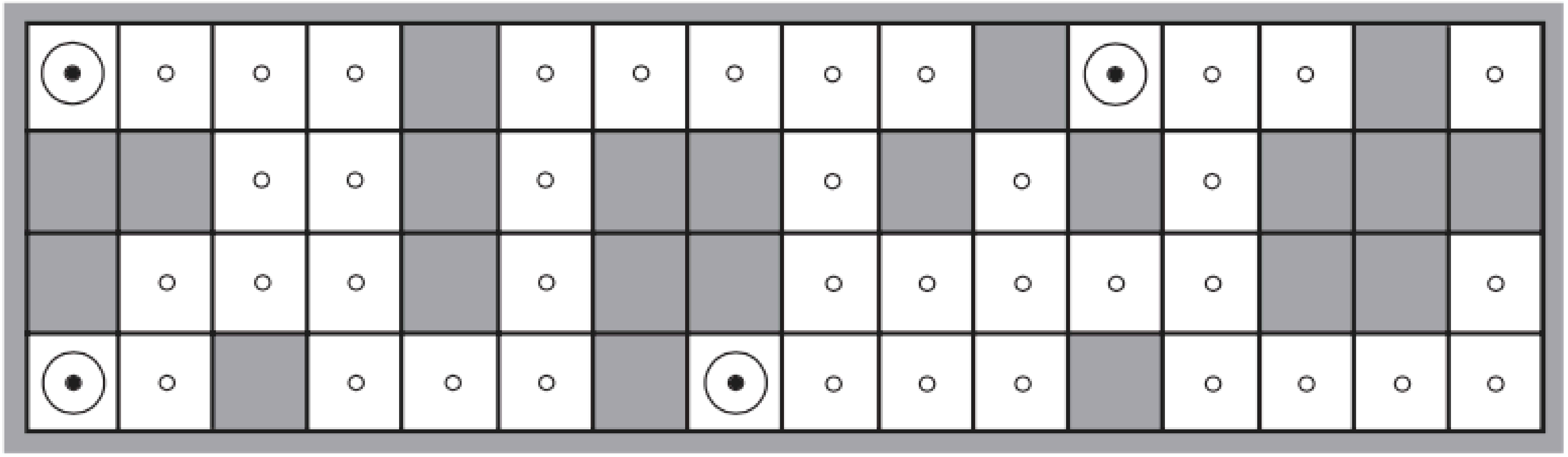


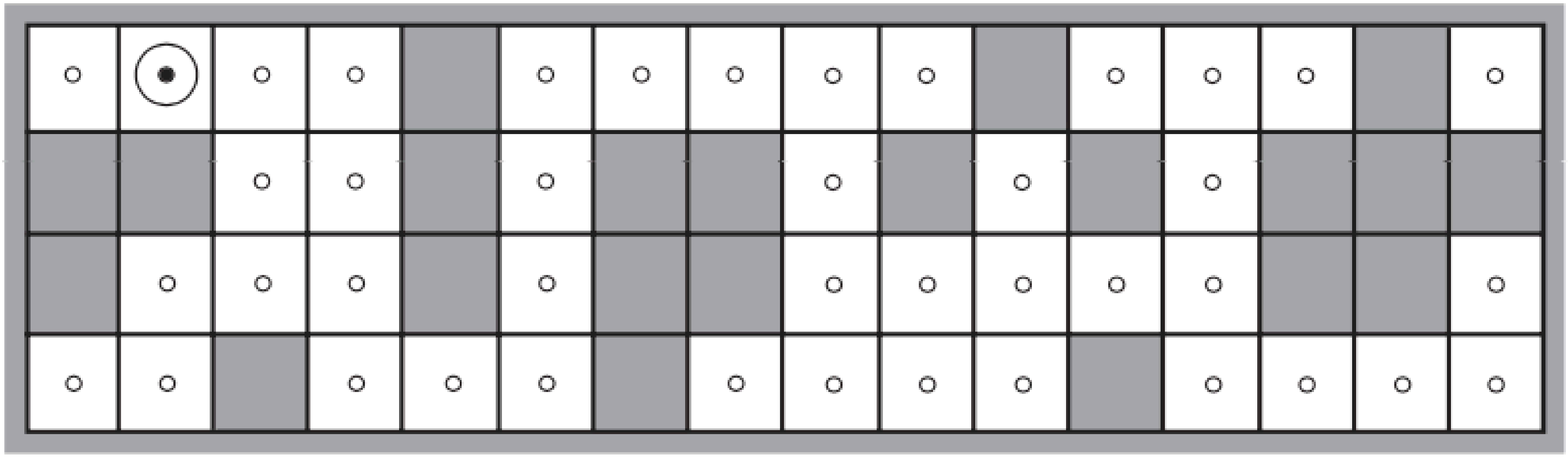
Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

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

- b′ = UPDATE(PREDICT(b, a), o) .



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



(b) Possible locations of robot After $E_1 = NSW, E_2 = NS$

Figure 4.18 Possible positions of the robot, ⊙, (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.

内容提要 – 部分可观察环境中的Agent

- 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 environment. 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.

- That’ s the only location that could be the result of

- UPDATE(PREDICT(UPDATE(b, NSW), Move), NS) .

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

内容提要 – 联机搜索Agent和未知环境

- an **online search** agent **interleaves** computation and action: first it takes an action, then it observes the environment and computes the next action.

- 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.

- 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.

  - 联机搜索问题、联机搜索Agent、联机局部搜索、联机搜索中的学习

内容提要 –联机搜索Agent和未知环境

- 联机搜索问题
- 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).

联机搜索问题

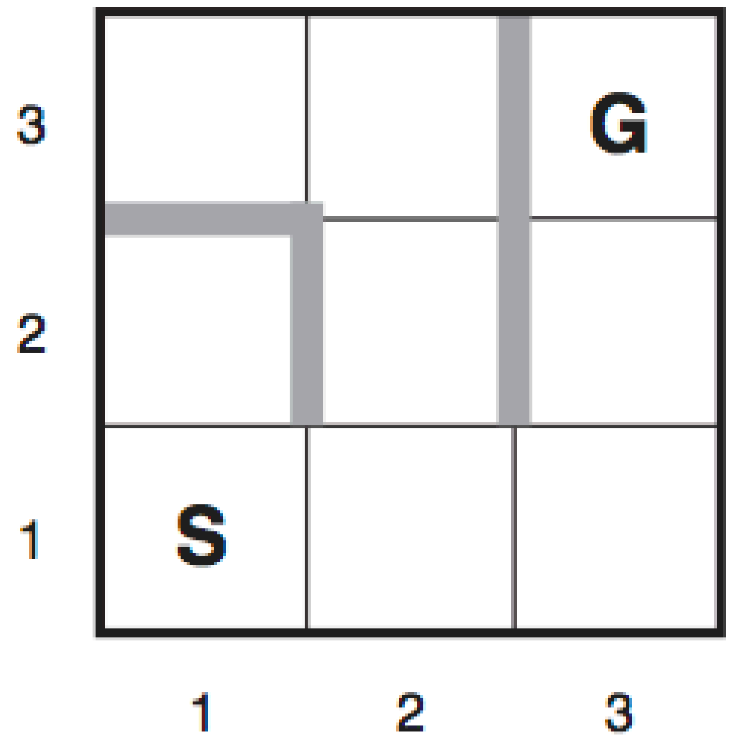


Figure 4.19 A simple maze problem. The agent starts at *S* and must reach *G* but knows nothing of the environment.

- Finally, the agent might have access to an admissible heuristic function h(s) that es- timates 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.

联机搜索问题

- 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.

联机搜索问题

- Although this sounds like a reasonable request, it is easy to see that the best achievable competitive ratio is infinite in some cases. 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.

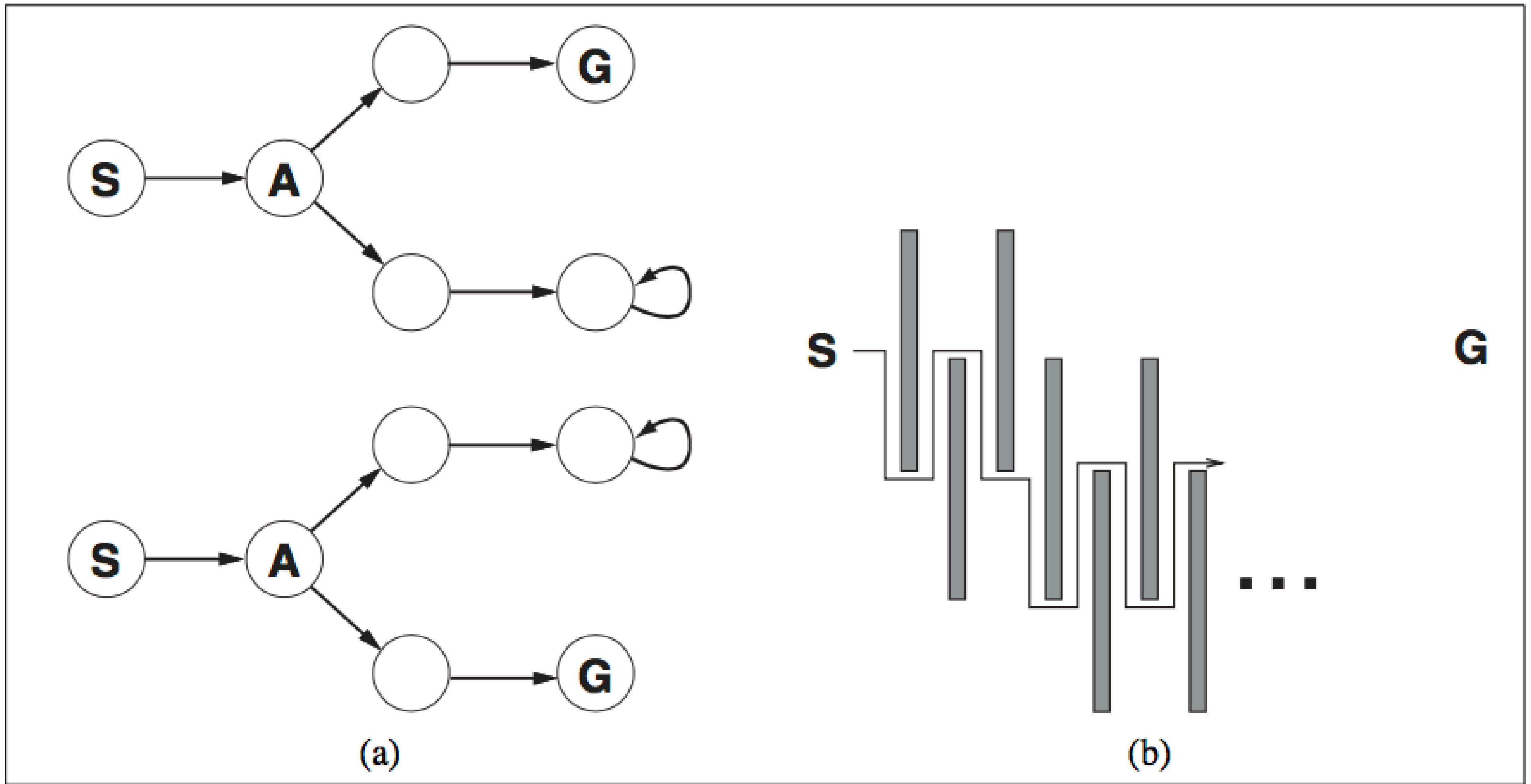
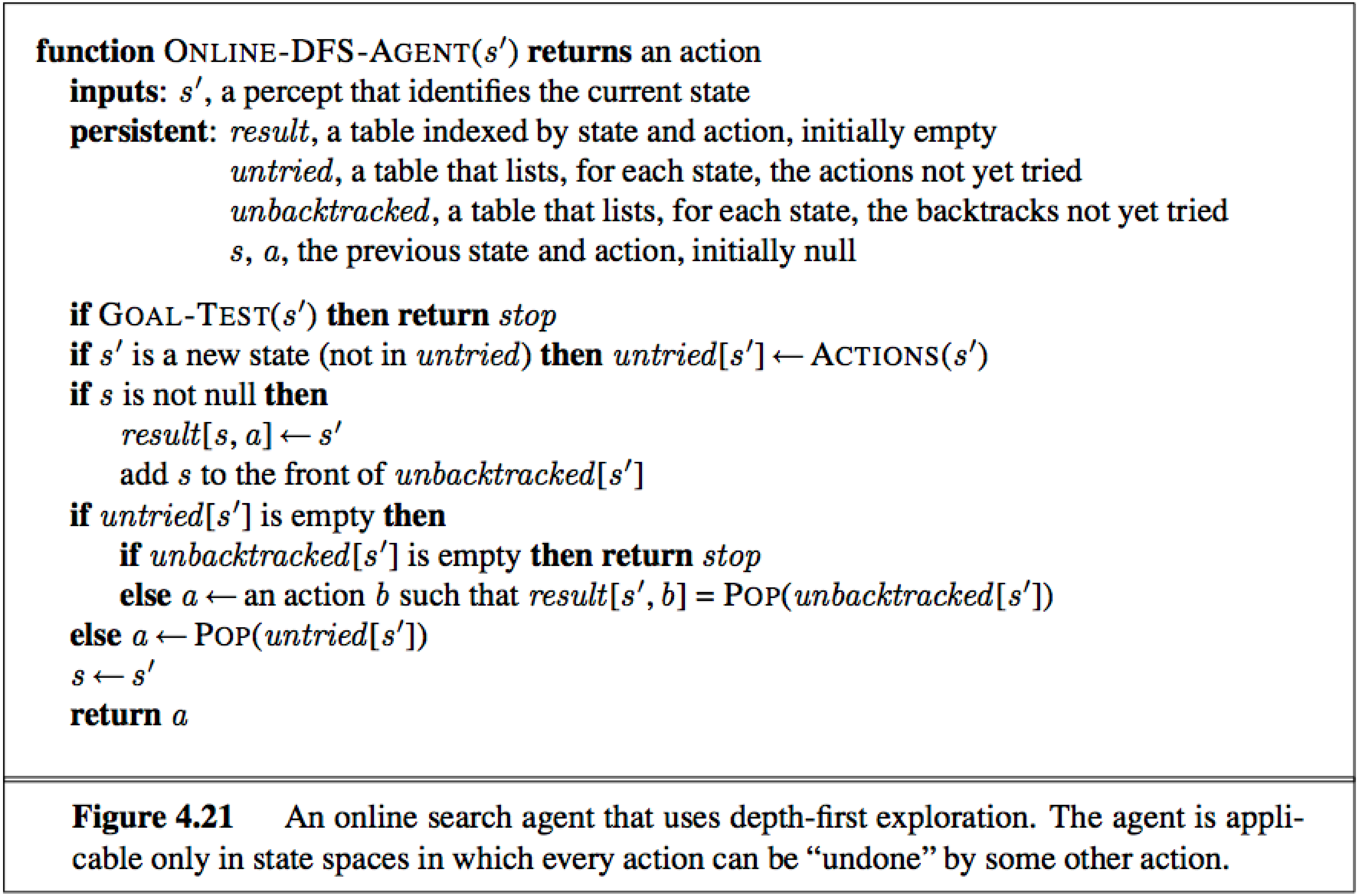


Figure 4.20 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent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

内容提要 – 联机搜索Agent



```
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[$s'$] $\leftarrow$ ACTIONS($s'$)
    if $s$ is not null then
        result[$s$, $a$] $\leftarrow s'$
        add $s$ to the front of unbacktracked[$s'$]
    if untried[$s'$] is empty then
        if unbacktracked[$s'$] is empty then return stop
        else $a \leftarrow$ an action $b$ such that result[$s'$, $b$] = POP(unbacktracked[$s'$])
    else $a \leftarrow$ POP(untried[$s'$])
    $s \leftarrow s'$
    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.

内容提要 – 联机搜索Agent

- 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.

内容提要 – 联机局部搜索

- **random walk**

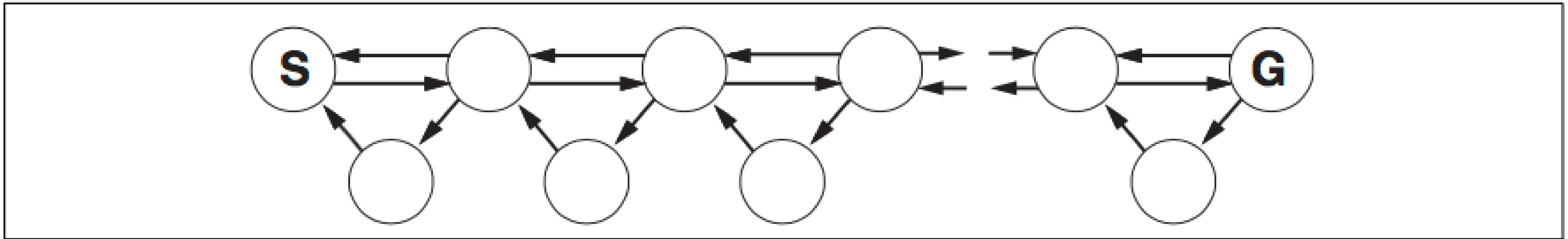


Figure 4.22 An environment in which a random walk will take exponentially many steps to find the goal.

- 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.

learning real-time A* (LRTA*)

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.

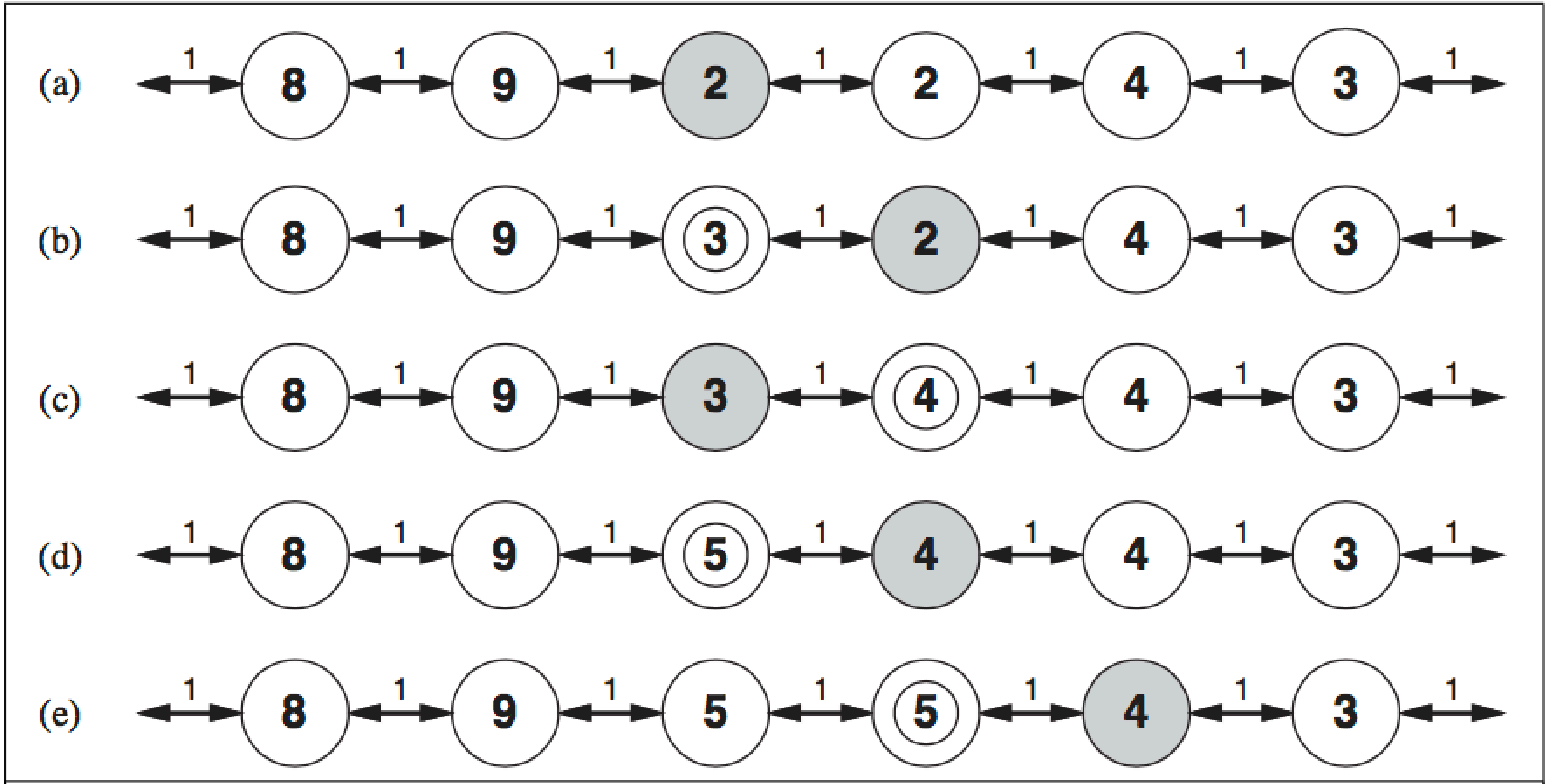
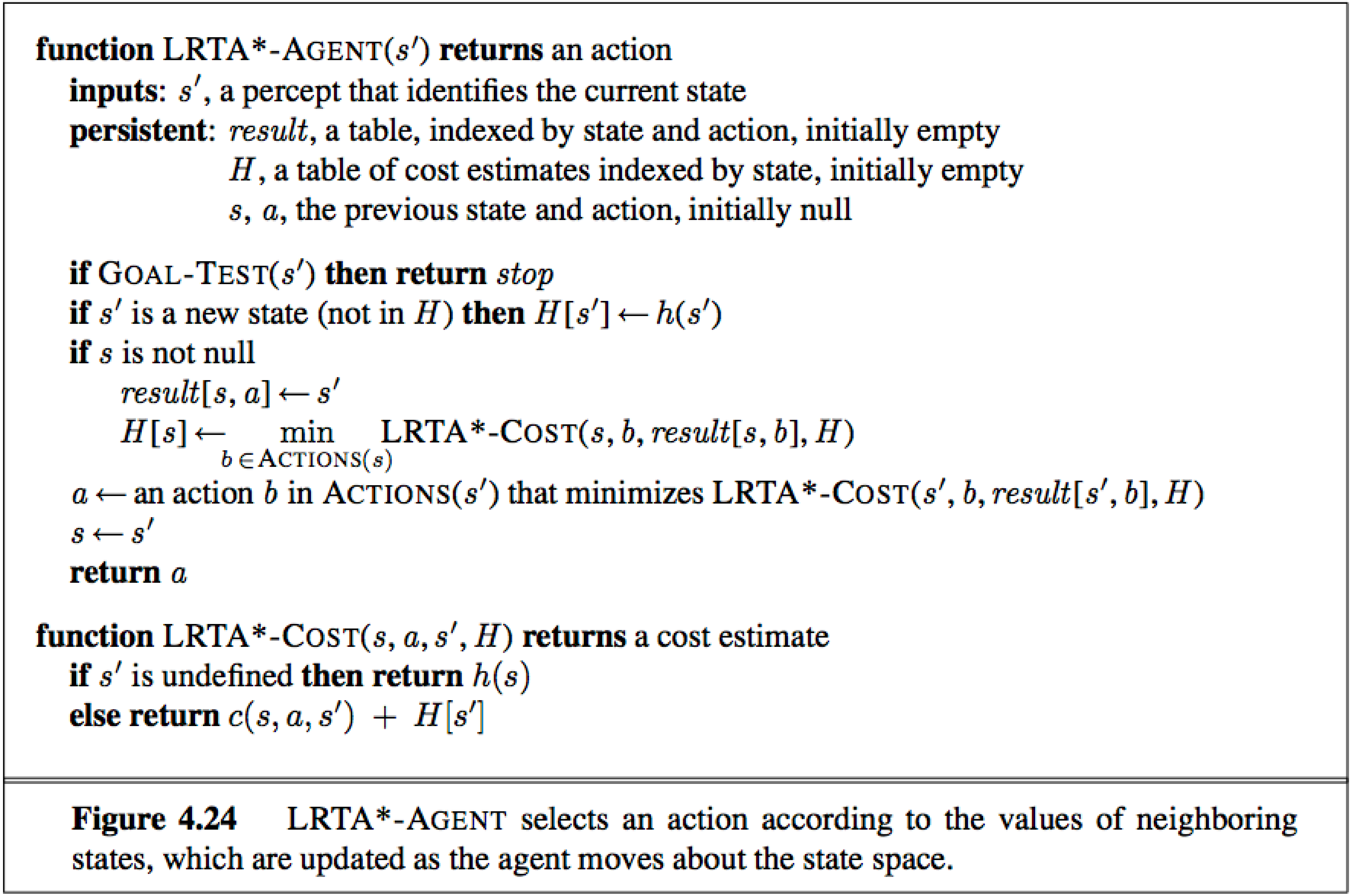


Figure 4.23 Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and each link is labeled with its step cost. The shaded state marks the location of the agent, and the updated cost estimates at each iteration are circled.

learning real-time A* (LRTA*)



```
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
        $result[s, a] \leftarrow s'$
        $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)}$ 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.

内容提要 – 联机搜索中的学习

- 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.

内容提要 – 联机搜索中的学习

- 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.

内容提要 – 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 pro- gramming** 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.

内容提要 – SUMMARY

- In **nondeterministic** environments, agents can apply AND–OR search to generate **con-**
- **tingent** 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 **sensor- less 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.