國立成功大學
National Cheng Kung University
1931

Introduction to

Artificial Intelligence

Chapter 4
Search in Complex Environments

Wei-Ta Chu (朱威達)

Local Search Algorithms and Optimization Algorithms

- The search algorithms that we have seen so far are designed to explore search spaces systematically. 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. In the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added.
    - We need algorithms not worrying about paths at all.

Local Search Algorithms and Optimization Algorithms

- **Local search** algorithms operate using a single current node and generally move only to neighbors of that node.
  - They use very little memory—usually a constant amount
  - They can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
- 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**.

Local Search Algorithms and Optimization Algorithms

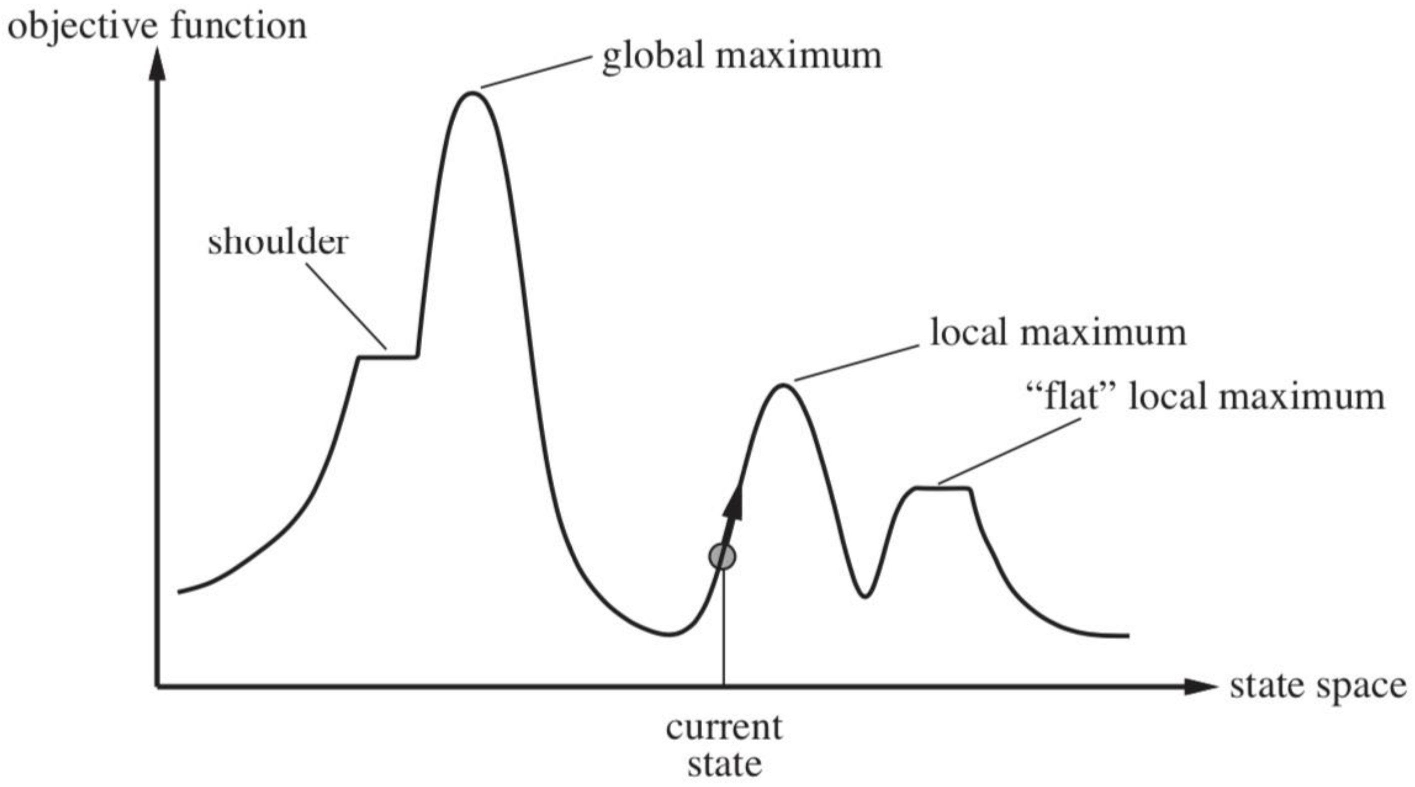
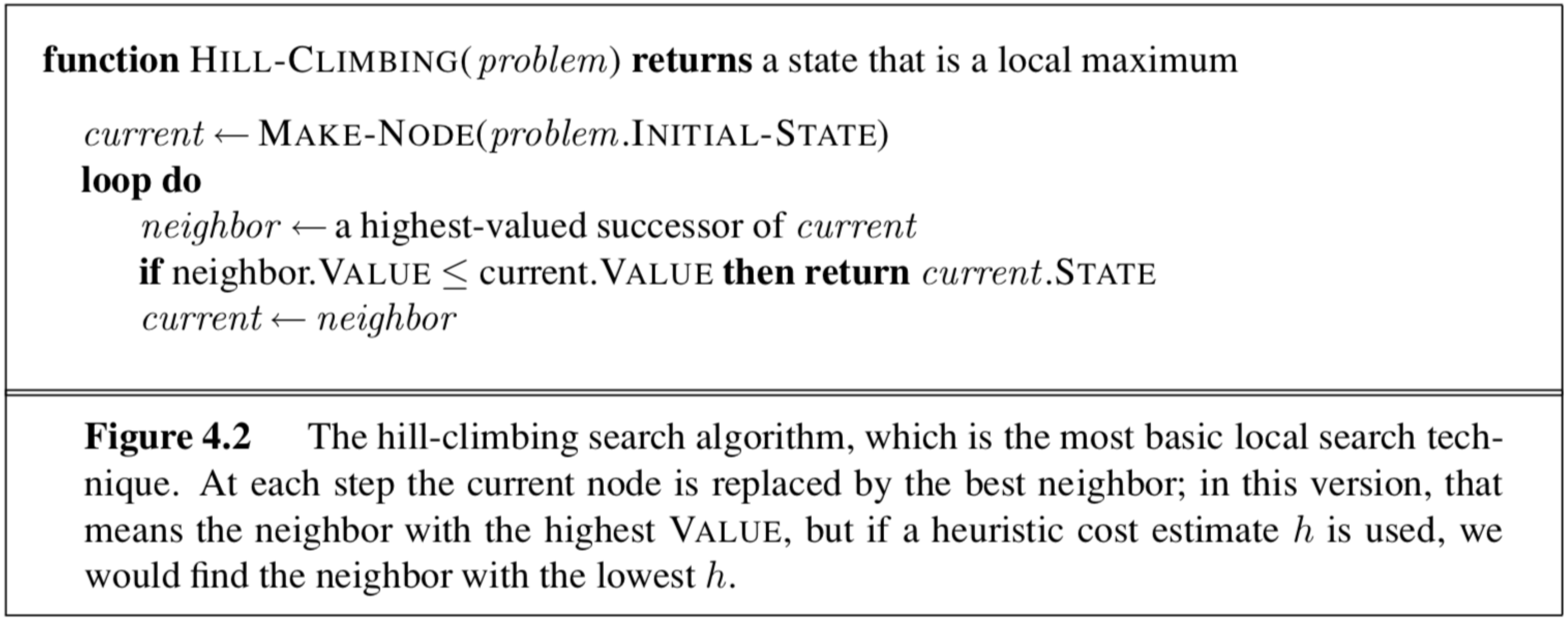


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill Climbing Search

- The hill-climbing search algorithm (steepest-ascent version) is simply a loop that continually moves in the direction of increasing value. It terminates when it reaches a “peak”.
- 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 ≤ 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.

Hill Climbing Search

- 8-queens problem
- The successors of a state are all possible states generated by moving a single queen to another square in the same column. The heuristic cost function *h* is the number of pairs of queens that are attacking each other.
- Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.

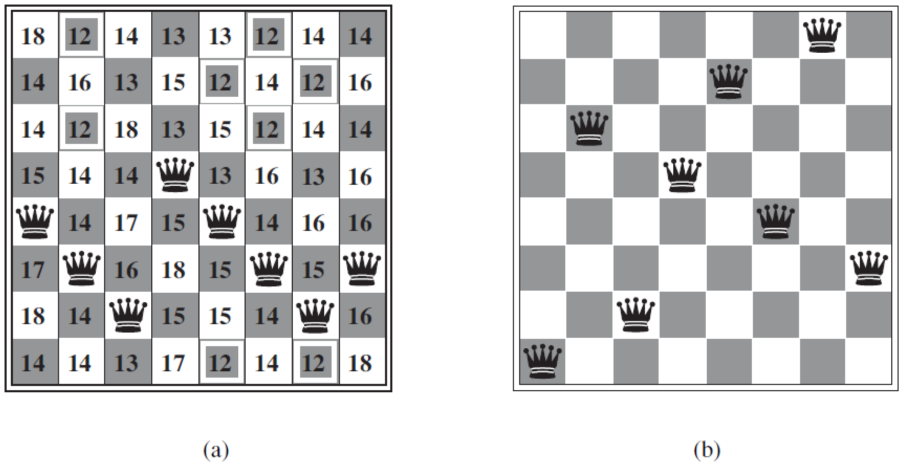


Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

Hill Climbing Search

- Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.
  - It turns out that greedy algorithms often perform quite well
- Hill climbing often gets stuck for the following reasons:
  - Local maxima
  - Ridges (屋脊)
  - Plateaux (can be flat local maximum or a **shoulder**)

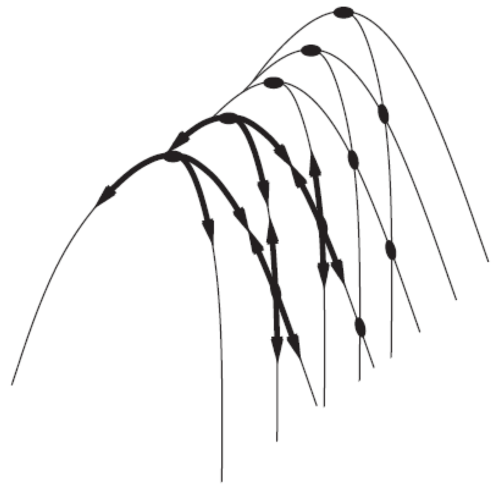


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point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

Hill Climbing Search

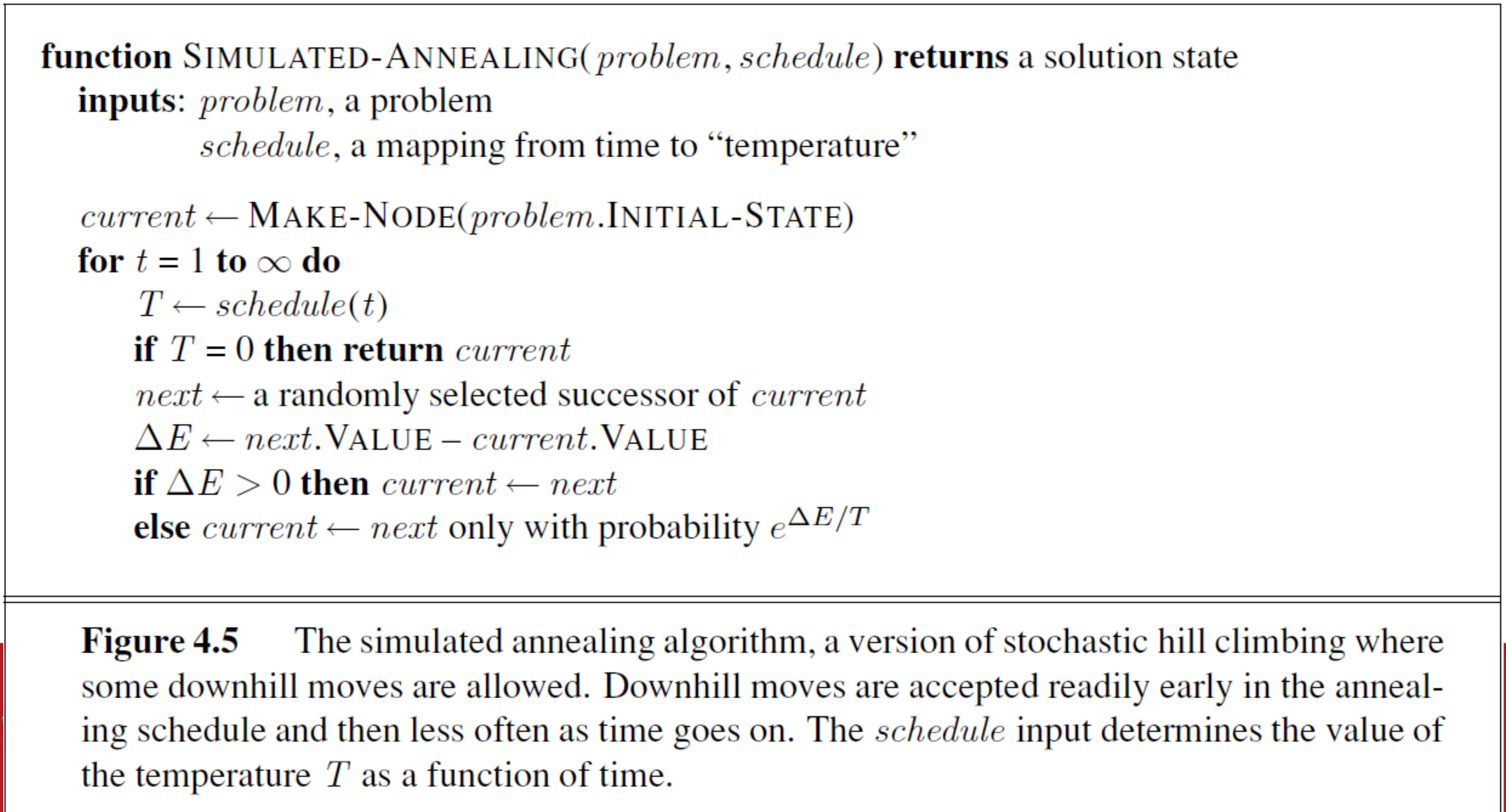
- For the 8-queens problem, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck—not bad for a state space with $8^8 \approx 17$ million states.
- Might it not be a good idea to keep going—to allow a sideways move in the hope that the plateau is really a shoulder? The answer is usually yes. 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.

Hill Climbing Search

- **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
- **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
- **Random-restart hill climbing** conducts a series of hill-climbing searches from randomly generated initial states until a goal is found.
- The success of hill climbing depends very much on the shape of the state-space landscape.

Simulated Annealing

- **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.



```
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 $e^{\Delta E/T}$
```

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

- 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 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.

Local Beam Search

- The **local beam search** algorithm keeps track of *k* states rather than just one.
- It begins with *k* randomly generated states. At each step, all the successors of all *k* states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the *k* best successors from the complete list and repeats.
- A local beam search seem to be nothing more than running *k* random restarts in parallel instead of in sequence. 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.
- **Stochastic beam search** chooses *k* successors at random, with the probability of choosing a given successor being an increasing function of its value.

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.
- 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.

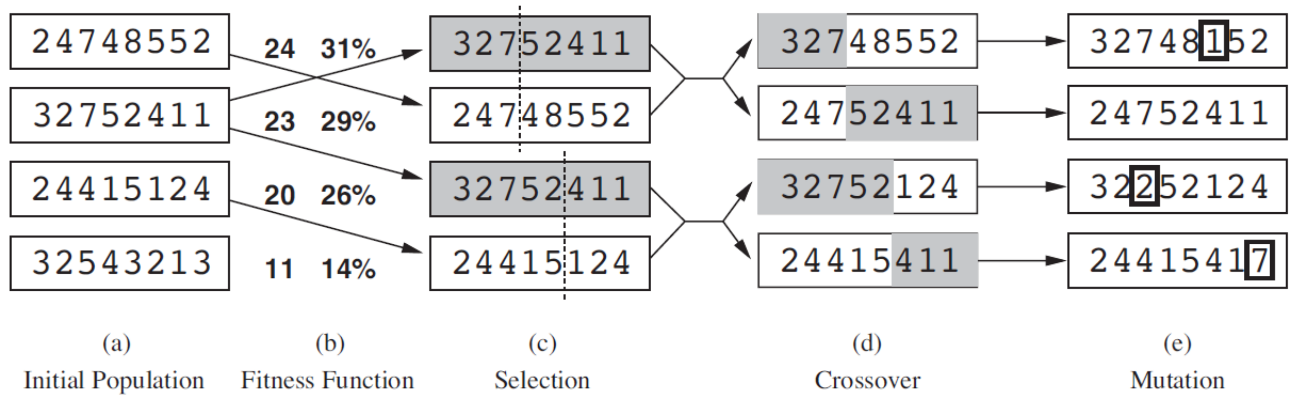


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Genetic Algorithms

- A **fitness** function should return higher values for better states. The probability of being chosen for reproducing is directly proportional to the fitness score.
- For each pair to be mated, a **crossover** point is chosen randomly from the positions in the string.

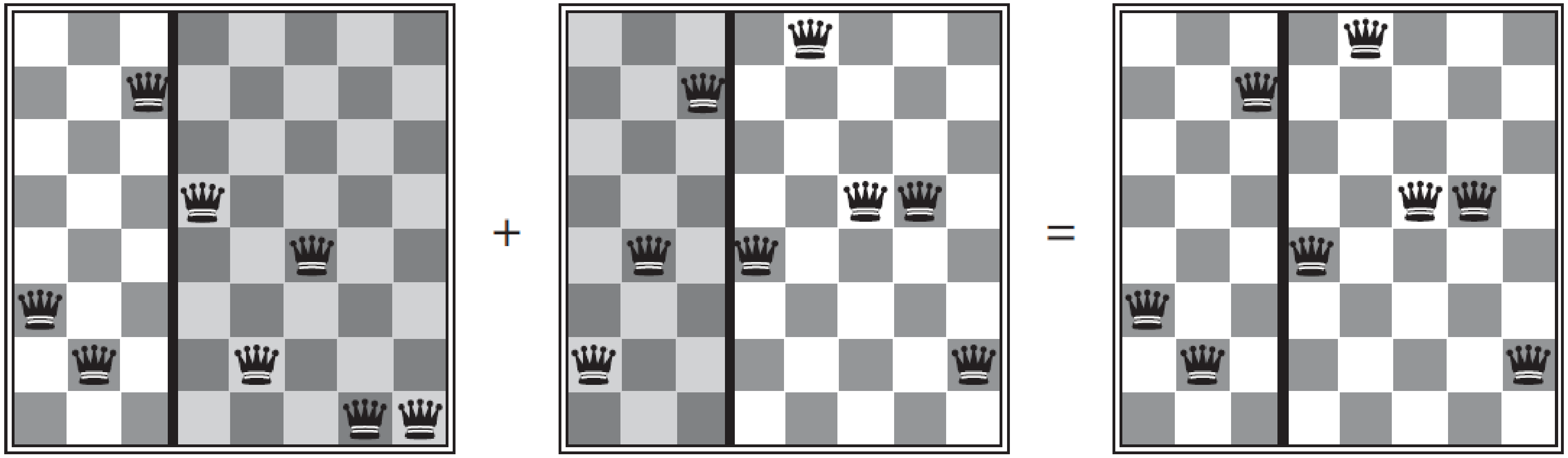
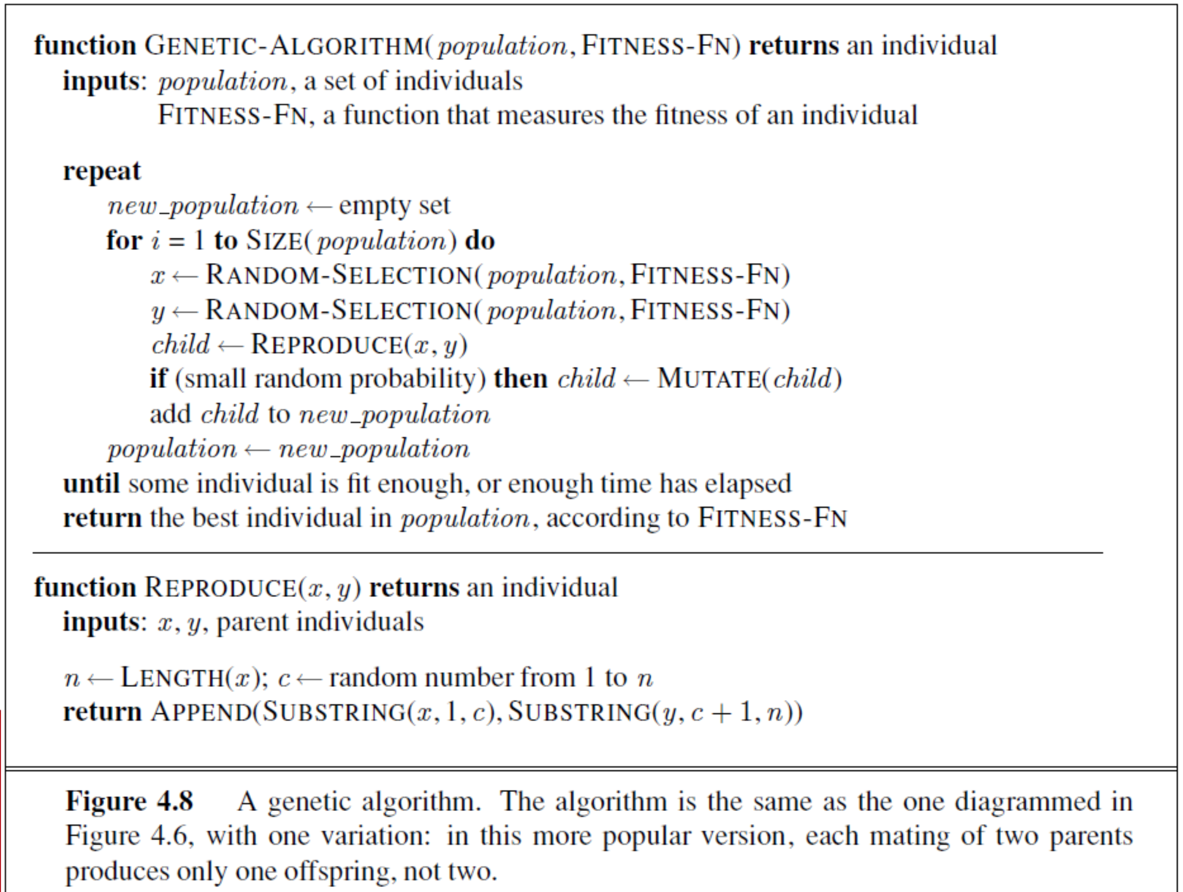


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

Genetic Algorithms

- Finally, each location is subject to random mutation with a small independent probability. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.



```
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.

Local Search in Continuous Spaces

- Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map 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**.

Local Search in Continuous Spaces

- Let C_i be the set of cities whose closest airport (in the current state) is airport i. The objective function is

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

- To avoid continuous problems: **discretize** the neighborhood of each state. 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.

Local Search in Continuous Spaces

- 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.

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

- In some cases, we can find a maximum by solving the equation $\nabla f = 0$. In many cases, however, this equation cannot be solved in closed form.

Local Search in Continuous Spaces

- 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; for example,

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

- 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**.

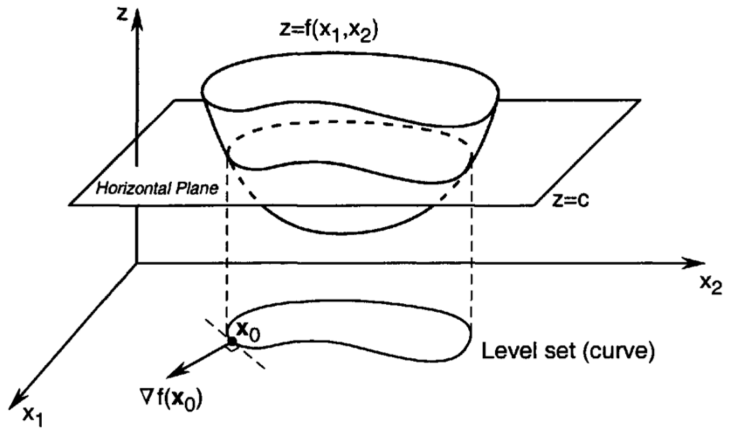
Local Search in Continuous Spaces

- 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 until *f* starts to decrease again.
- 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

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

Introduction

- The gradient of f at $\boldsymbol{x}_0$, denoted by $\triangledown f(\boldsymbol{x}_0)$, is orthogonal to the tangent vector to an arbitrary smooth curve passing through $\boldsymbol{x}_0$ on the level set $f(\boldsymbol{x}) = c$
- The direction of maximum rate of increase of a real-valued differentiable function at a point is orthogonal to the level set of the function through that point.
- The gradient acts in such a direction that for a given small displacement, the function f increases more in the direction of the gradient than in any other directio

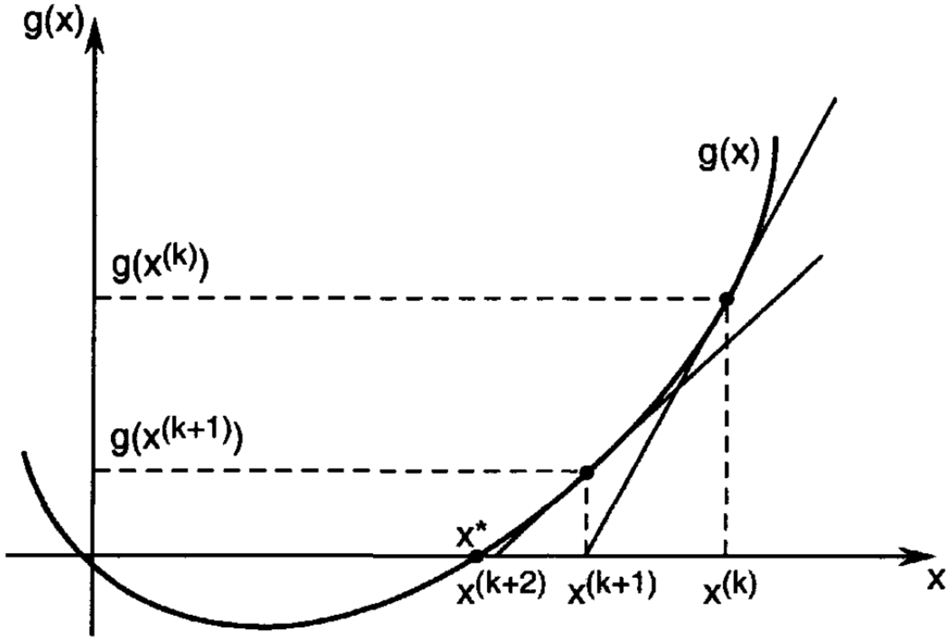


Newton's Method

- Newton's method for solving equations of the form $g(x) = 0$ is also referred to as ***Newton's method of tangents***.
- If we draw a tangent to $g(x)$ at the given point $x^{(k)}$, then the tangent line intersects the *x*-axis at the point $x^{(k+1)}$, which we expect to be closer to the root x^* of $g(x) = 0$.
- Note that the slope of $g(x)$ at

$$g'(x^{(k)}) = \frac{g(x^{(k)})}{x^{(k)} - x^{(k+1)}}$$

$$\Rightarrow \quad x^{(k+1)} = x^{(k)} - \frac{g(x^{(k)})}{g'(x^{(k)})}$$



Local Search in Continuous Spaces

- To find a maximum or minimum of *f*, we need to find *x* such that the gradient is zero (i.e., $\nabla f(x) = 0$). Thus, $g(x)$ in Newton's formula becomes $\nabla f(x)$, 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 $\partial^2 f/\partial x_i \partial x_j$.
- 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.

Local Search in Continuous Spaces

- A **constrained optimization** problem is constrained if solutions must satisfy some hard constraints on the values of the variables.
- The best-known category is that of **linear programming** problems, in which constraints must be linear inequalities forming a **convex set** and the objective function is also linear.
- 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.

Simple Examples of Linear Programs

- Formally, a linear program is an optimization problem of the form

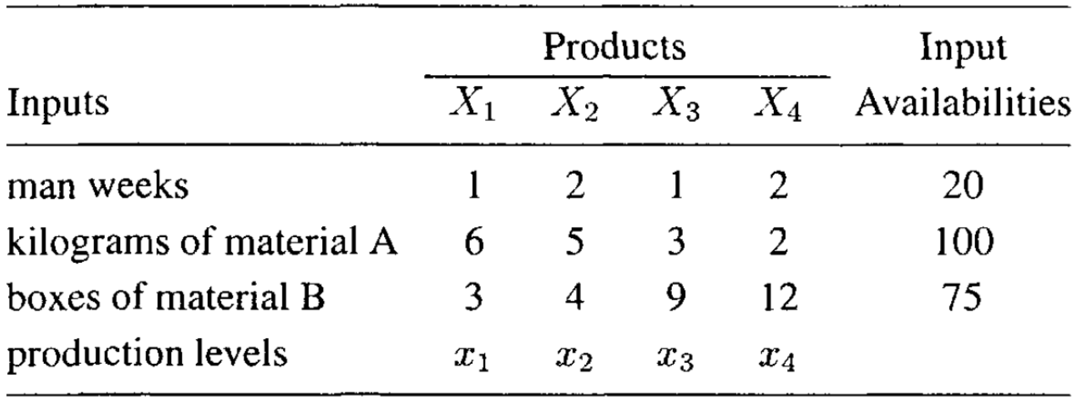
$$\begin{array}{ll} \text{minimize} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \quad \boldsymbol{x} \geq \boldsymbol{0} \end{array}$$

  where $\boldsymbol{c} \in R^n, \boldsymbol{b} \in R^m, \boldsymbol{A} \in R^{m\times n}$. The vector inequality $\boldsymbol{x} \geq \boldsymbol{0}$ means that each component of $\boldsymbol{x}$ is nonnegative.
- Several variations of this problem are possible. For example, we can maximize, or the constraints may be in the form of inequalities, such as $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$ or $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$. In fact, these variations can all be rewritten into the standard form shown above.

Example

- A manufacturer produces four different products: X_1, X_2, X_3, X_4 there are three inputs to this production process: labor in person-weeks, kilograms of raw material A, and boxes of raw material B. Each product has different input requirements. In determining each week's production schedule, the manufacturer cannot use more than the available amounts of labor and the two raw materials. The relevant information is presented in this table. Every production decision must satisfy the restrictions on the availability of inputs. These constraints can be written using the data in this table.

$$x_1 + 2x_2 + x_3 + 2x_4 \leq 20$$
$$6x_1 + 5x_2 + 3x_3 + 2x_4 \leq 100$$
$$3x_1 + 4x_2 + 9x_3 + 12x_4 \leq 75$$



	Products				Input
Inputs	X_1	X_2	X_3	X_4	Availabilities
man weeks	1	2	1	2	20
kilograms of material A	6	5	3	2	100
boxes of material B	3	4	9	12	75
production levels	x_1	x_2	x_3	x_4	

Searching with Nondeterministic Actions

- The erratic vacuum world
  - The state space has eight states. 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.

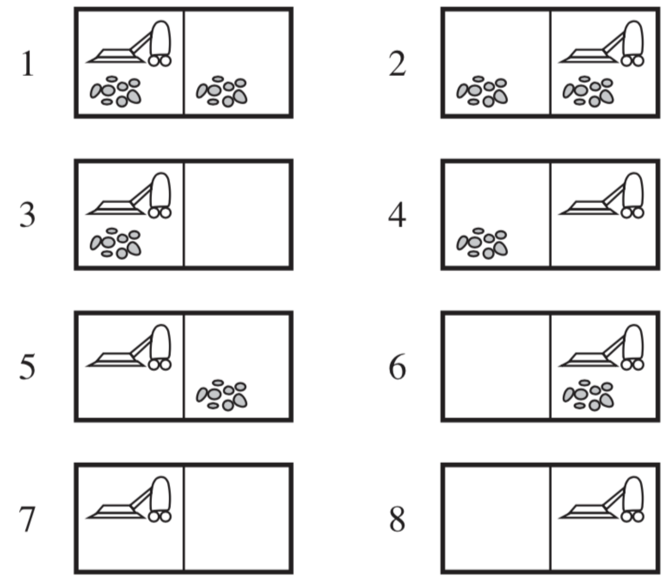


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

Searching with Nondeterministic Actions

- Suppose that we introduce nondeterminism. 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.
- 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}

Searching with Nondeterministic Actions

- 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 (偶發、可能性) plan such as the following:

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

- Thus, solutions for nondeterministic problems can contain nested **if–then–else** statements; this means that they are **trees** rather than sequences. Many problems in the real, physical world are contingency problems because exact prediction is impossible.

Searching with Nondeterministic Actions

- AND-OR search trees
  - 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 a nondeterministic environment, branching is 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**.

Searching with Nondeterministic Actions

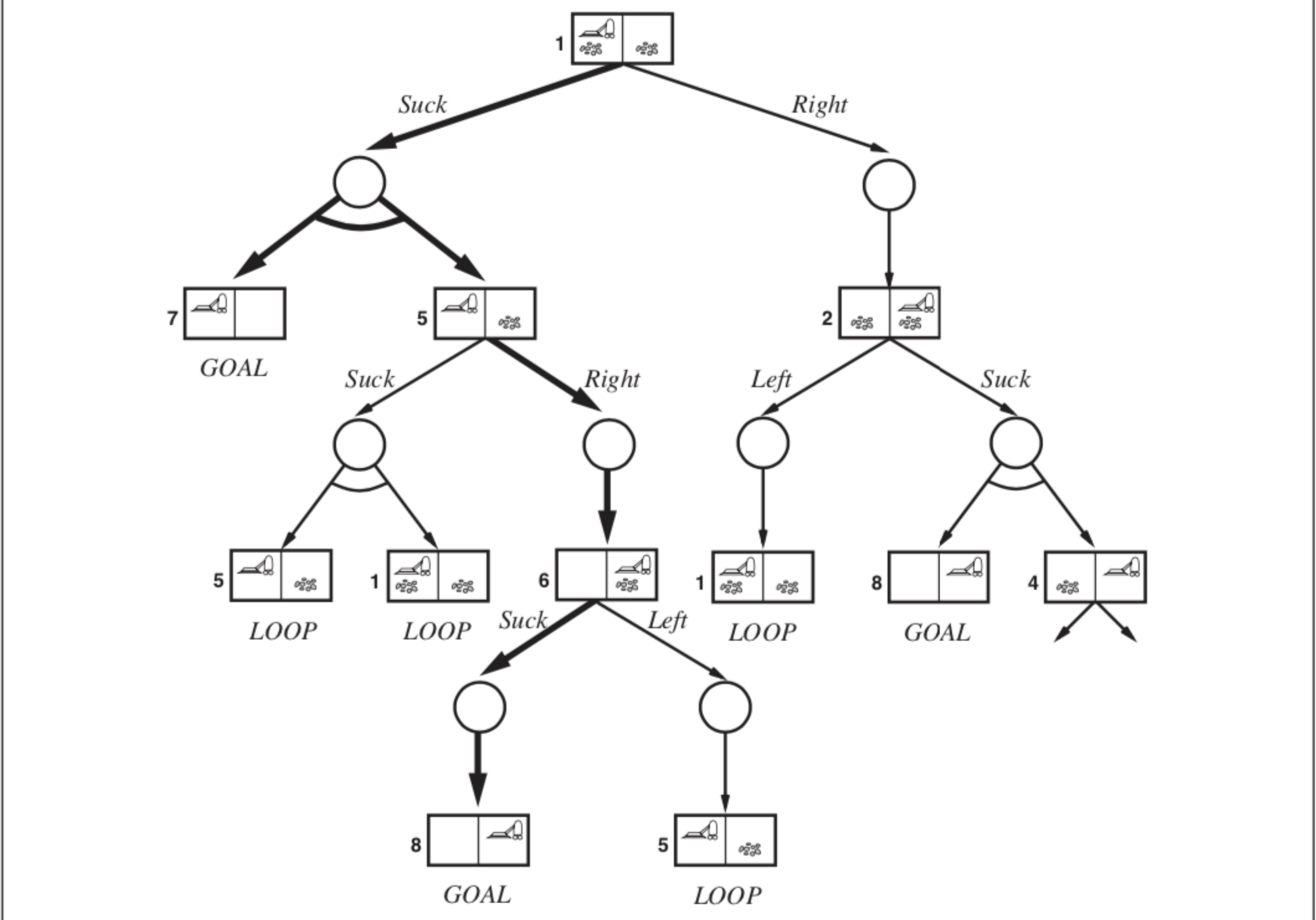
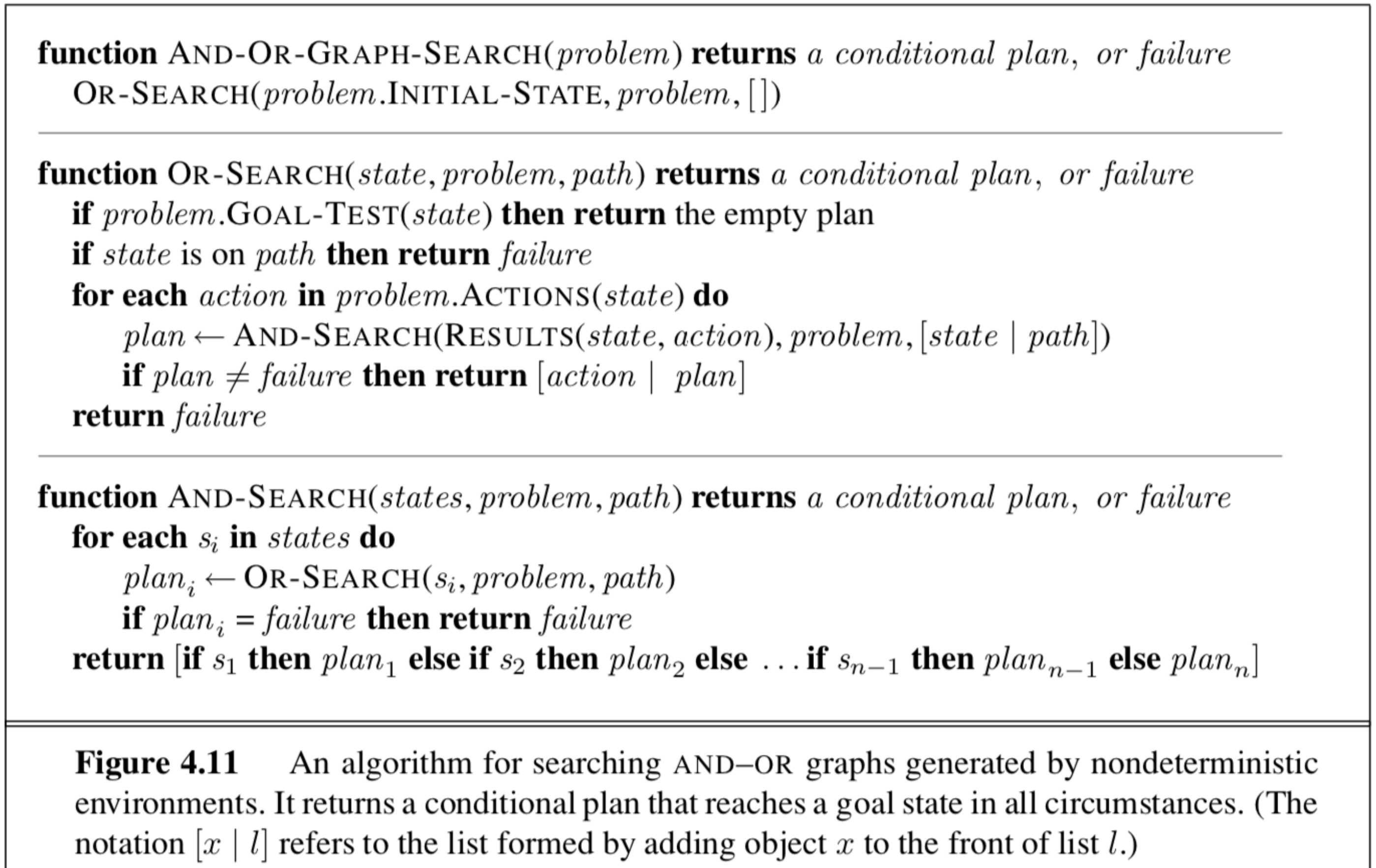


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

Searching with Nondeterministic Actions

- 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.



```
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
        $plan \leftarrow$ AND-SEARCH(RESULTS($state$, $action$), $problem$, [$state$ | $path$])
        if $plan \neq failure$ 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
        $plan_i \leftarrow$ 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.)

Searching with Nondeterministic Actions

- Try, try again
  - Consider the slippery (須小心對待的) 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}.

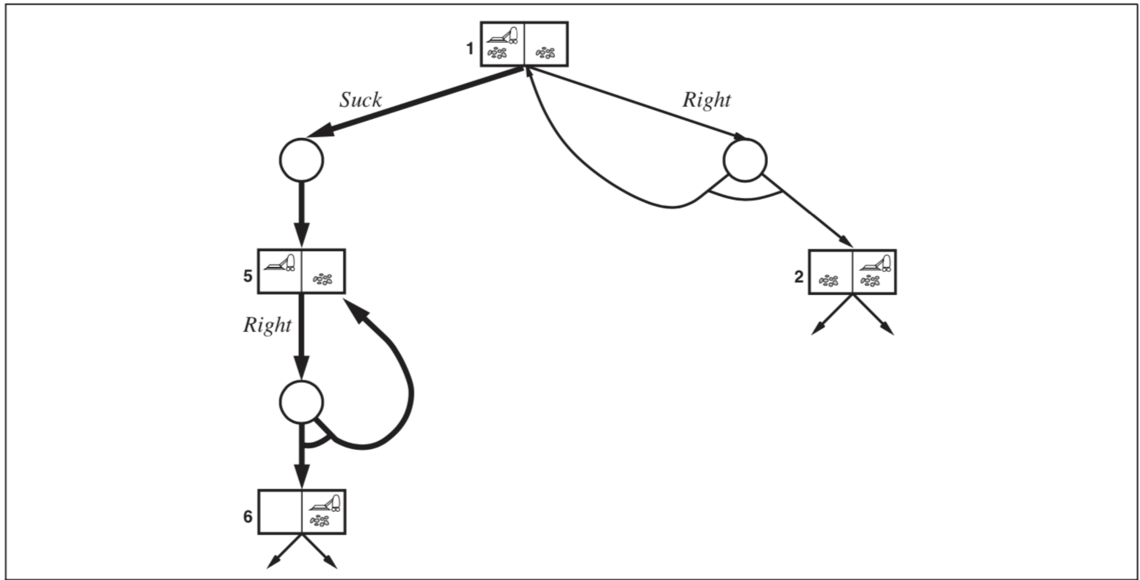


Figure 4.12 Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

Searching with Nondeterministic Actions

- 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

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

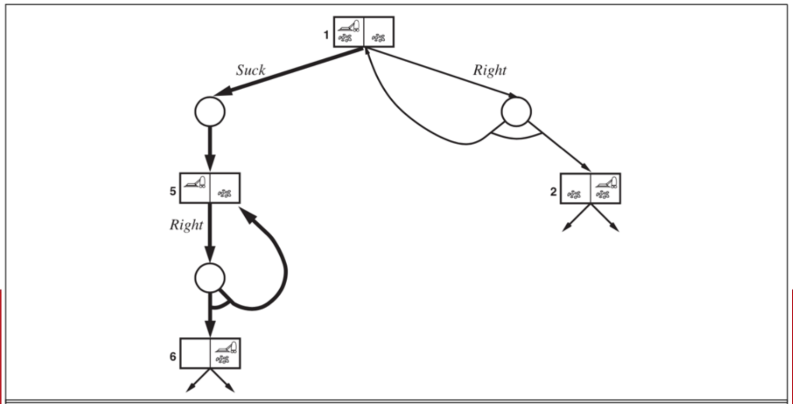


Figure 4.12 Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

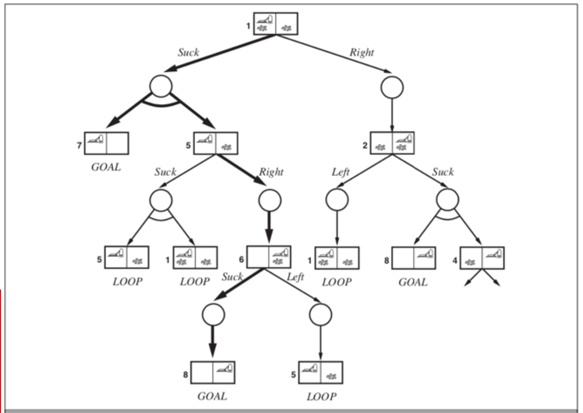


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The

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.
- 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.

Searching with Partial Observations

- **Searching with no observation (sensorless problem)**
- Assume that the agent knows the geography of its world, but doesn't know its location or the distribution of dirt. Its initial state could be any element of the set {1, 2, 3, 4, 5, 6, 7, 8}.
- Consider what happens if it tries the action *Right*. This will cause it to be in one of the states {2, 4, 6, 8}. 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** (強制) the world into state 7.

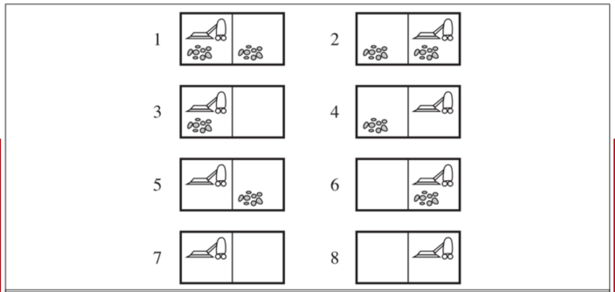


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

National Cheng Kung University

Searching with Partial Observations

- To solve sensorless problems, we search in the space of *belief states* rather than *physical states*.
- 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.

Searching with Partial Observations

- The sensorless problem:
  - **Actions**: Suppose the agent is in belief state $b = \{s_1, s_2\}$, but $\text{ACTIONS}_P(s_1) \neq \text{ACTIONS}_P(s_2)$. 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:

$$\textsc{Actions}(b) = \bigcup_{s \in b} \textsc{Actions}_P(s)$$

    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.

Searching with Partial Observations

- The sensorless problem:
  - **Transition model**: The agent doesn't know which state in the belief state is the right one. For deterministic actions, the set of states that might be reached is $b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\}$ 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*. The process of generating the new belief state after the action is called the **prediction** step; *b*' = PREDICT$_P$(*b*, *a*).

Searching with Partial Observations

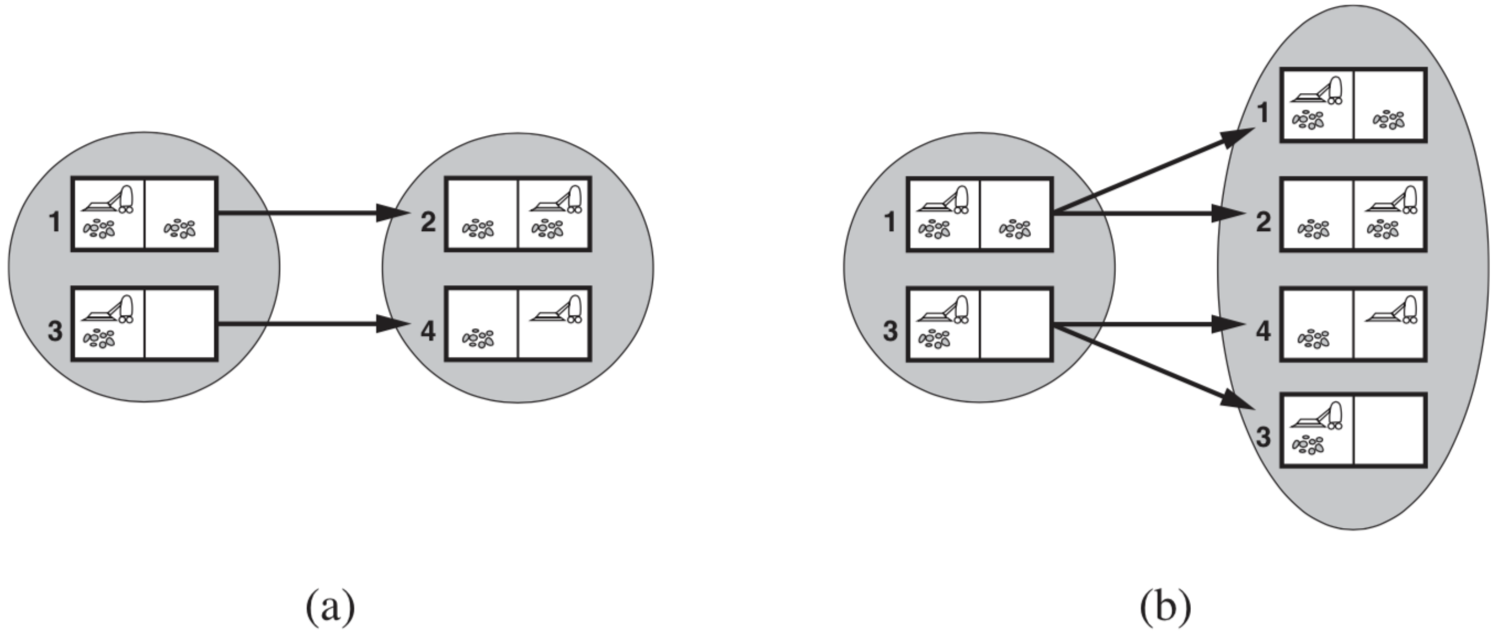


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

Searching with Partial Observations

- The sensorless problem:
  - **Goal test**: 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**: 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.

Searching with Partial Observations

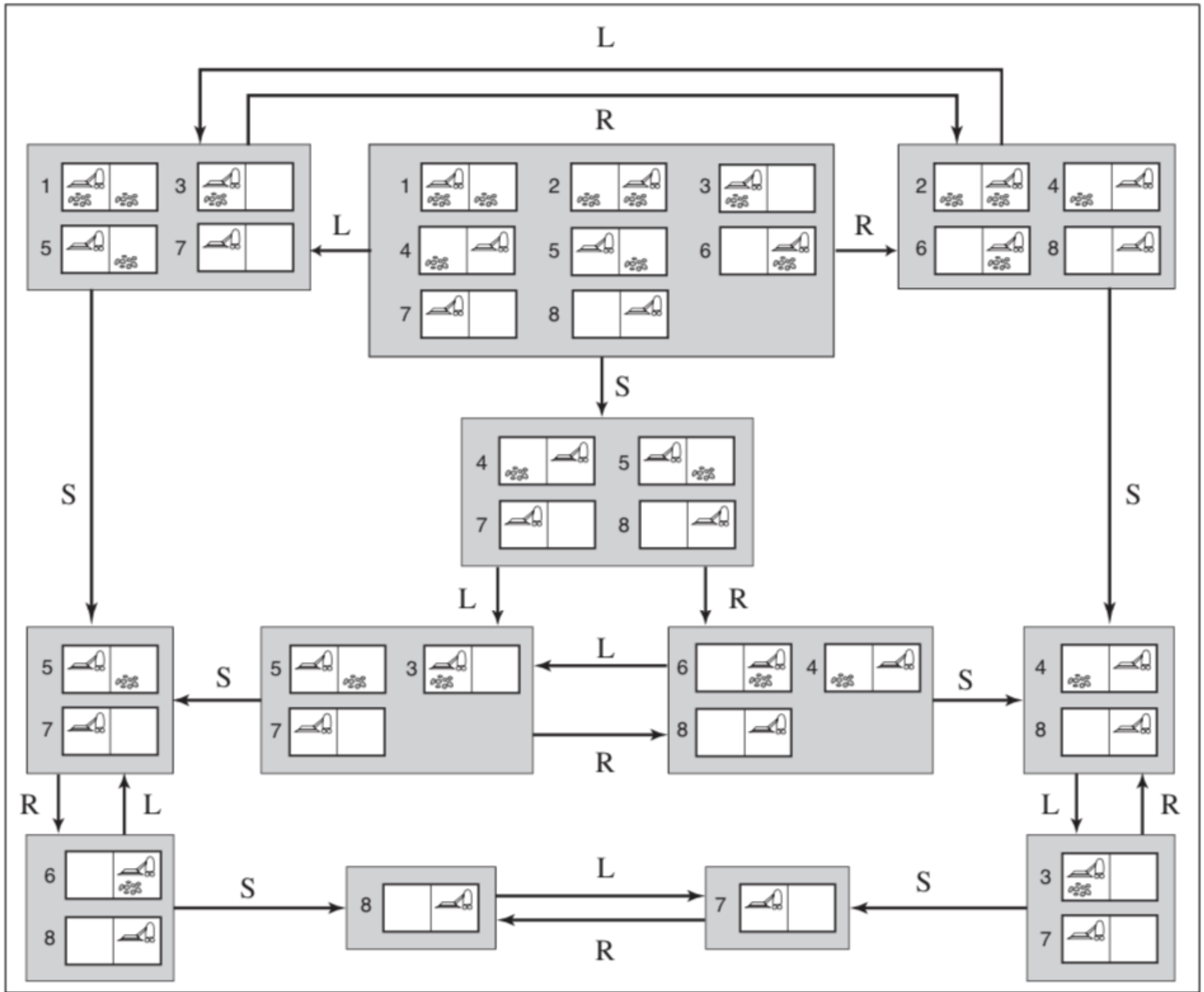


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete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

Searching with Partial Observations

- The sensorless problem:
    - 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.

Searching with Partial Observations

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

$$\textsc{Possible-Percepts}(\hat{b}) = \{o : o = \textsc{Percept}(s) \text{ and } s \in \hat{b}\}$$

Searching with Partial Observations

- 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

$$b_o = \textsc{Update}(\hat{b}, o) = \{s : o = \textsc{Percept}(s) \text{ and } s \in \hat{b}\}$$

  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.

Searching with Partial Observations

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

$$\begin{aligned} \textsc{Results}(b,a) = \{b_o : b_o &= \textsc{Update}(\textsc{Predict}(b,a), o) \text{ and} \\ o &\in \textsc{Possible-Percepts}(\textsc{Predict}(b,a))\} \end{aligned}$$

Searching with Partial Observations

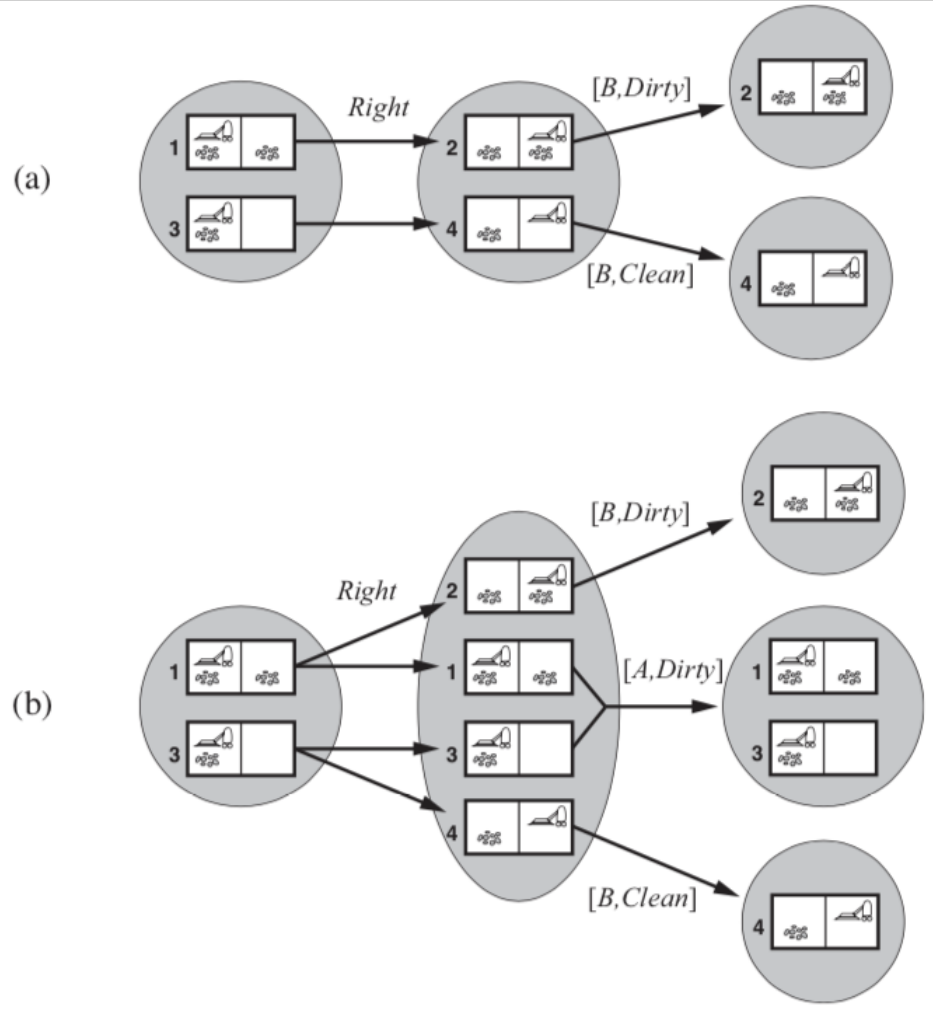


Figure 4.15 Two example of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are $[B, Dirty]$ and $[B, Clean]$, leading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

Searching with Partial Observations

- **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.
  - Two main differences: 1) 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. 2) the agent will need to maintain its belief state as it performs actions and receives percepts.

Searching with Partial Observations

- **An agent for partially observable environments**
  - Given an initial belief state *b*, an action *a*, and a percept *o*, the new belief state is: $b' = \text{UPDATE}(\text{PREDICT}(b, a), o)$

    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.

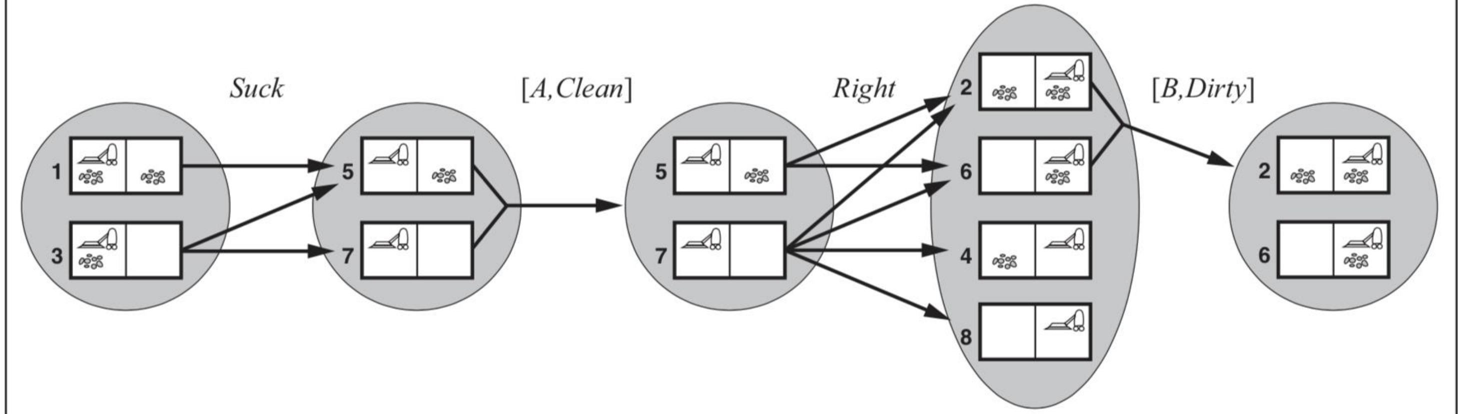
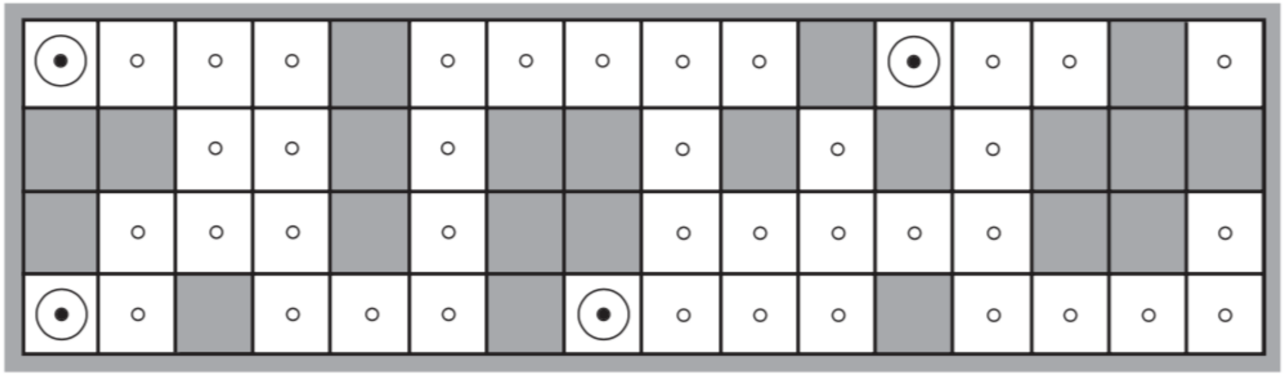


Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

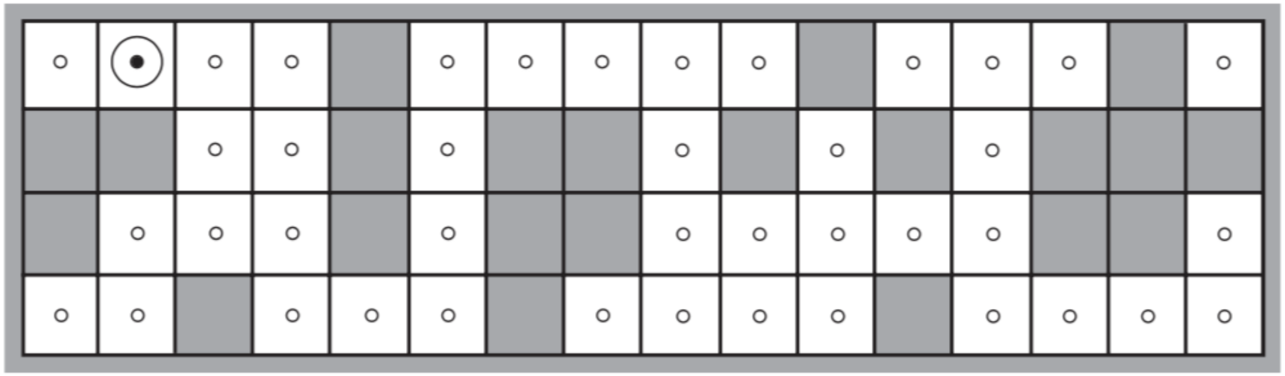
Searching with Partial Observations

- A robot is placed in the maze-like environment. It is equipped with four sonar sensors that tell whether there is an obstacle in each of the four compass directions.
- Assume that the sensors give perfectly correct data, and 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**.

Searching with Partial Observations



(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, $\odot$, (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.

Searching with Partial Observations

- Suppose the robot has just been switched on. Thus its initial belief state b consists of the set of all locations. The the robot receives the percept *NSW*, meaning there are obstacles to the north, west, and south, and does an update b_o = UPDATE(b), as shown in Figure 4.18(a). You can inspect the maze to see that those are the only four locations that yield the percept *NSW* .
- Next the robot executes a *Move* action, but the result is nondeterministic. The new belief state, b_a = 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 UPDATE(b_a,*NS*) and finds that the belief state has collapsed down to the single location shown in Figure 4.18(b).

Searching with Partial Observations

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

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

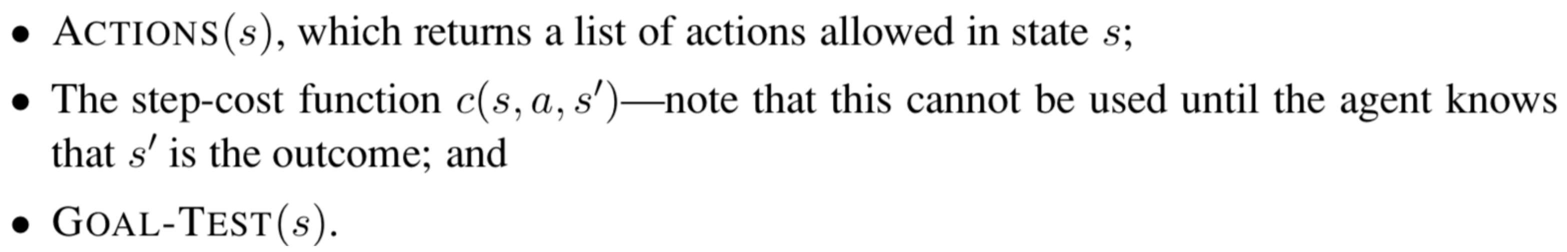
- With nondetermnistic actions the PREDICT step grows the belief state, but the UPDATE step shrinks it back down—as long as the percepts provide some useful identifying information.
- Sometimes the percepts don't help much for localization: If there were one or more long east-west corridors, then a robot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

Online Searching Agents with Unknown Environments

- 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** agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action.
- The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from *A* to *B*.

Online Searching Agents with Unknown Environments

- Online search algorithms
  - We stipulate (規定) 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).

  - The agent cannot determine RESULT(s,a) except by actually being in s and doing a.

Online Searching Agents with Unknown Environments

- Online search algorithms
  - 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).

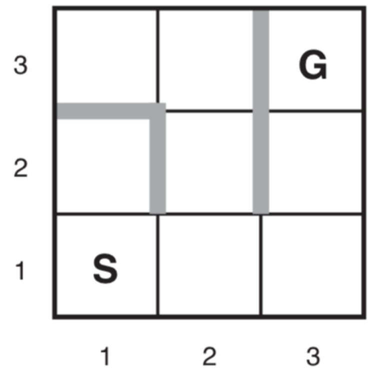


Figure 4.19 A simple maze problem. The agent starts at S and must reach G but knows nothing of the environment.

Online Searching Agents with Unknown Environments

- Online search algorithms
  - 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.

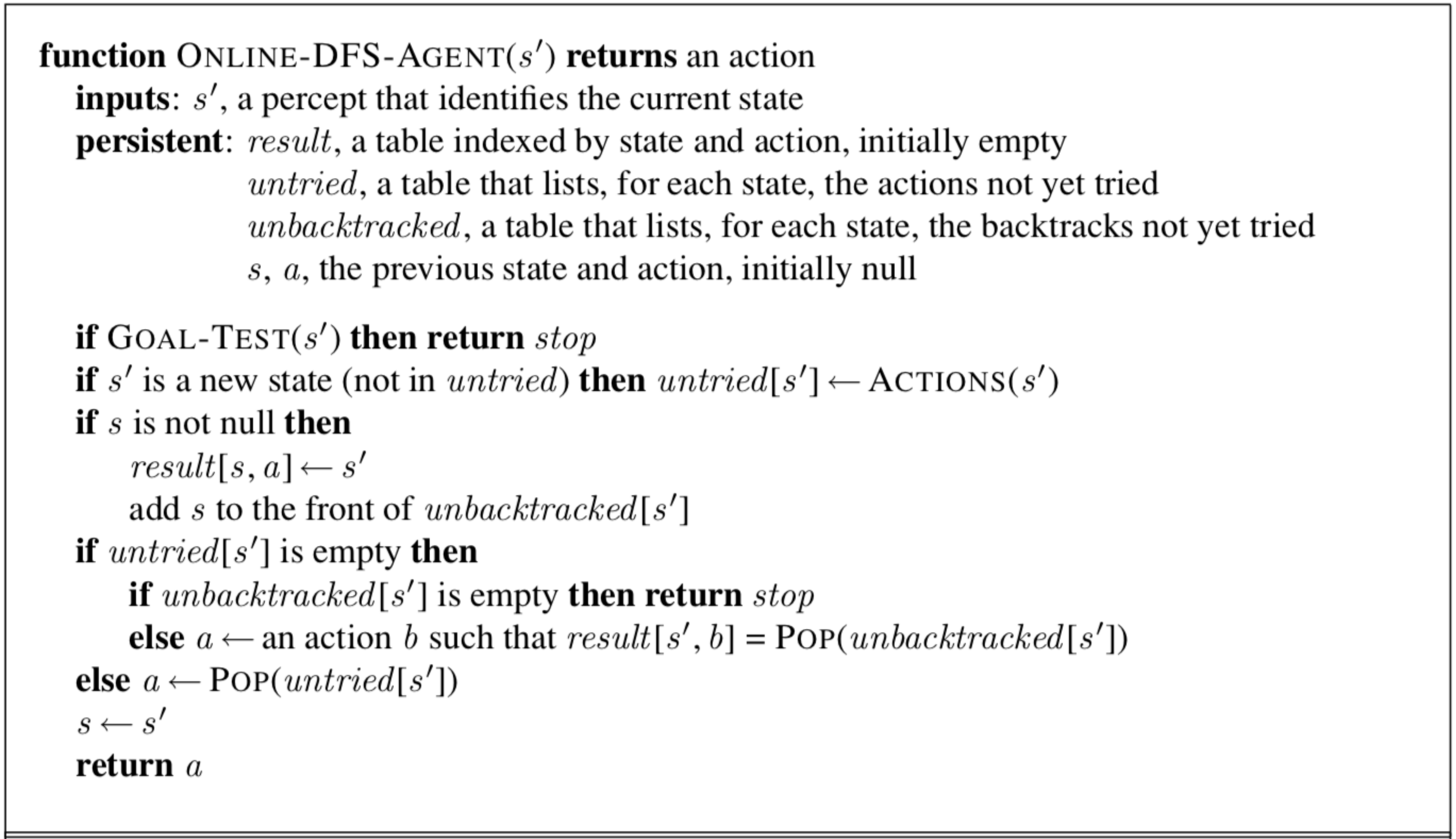
Online Searching Agents with Unknown Environments

- Online search algorithms
  - Typically, the agent’s objective is to reach a goal state while minimizing cost. 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. This is called the **competitive ratio**; we would like it to be as small as possible.

Online Searching Agents with Unknown Environments

- Online search agents
  - After each action, an online agent receives a percept telling it what state it has reached; from this info., 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.
  - To avoid traveling all the way across the tree to expand the next node, an online algorithm better expands nodes in a local order. DFS has exactly this property.

Online Searching Agents with Unknown Environments



```
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.

Online Searching Agents with Unknown Environments

- 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.