



Chapter 4:

BEYOND CLASSICAL SEARCH

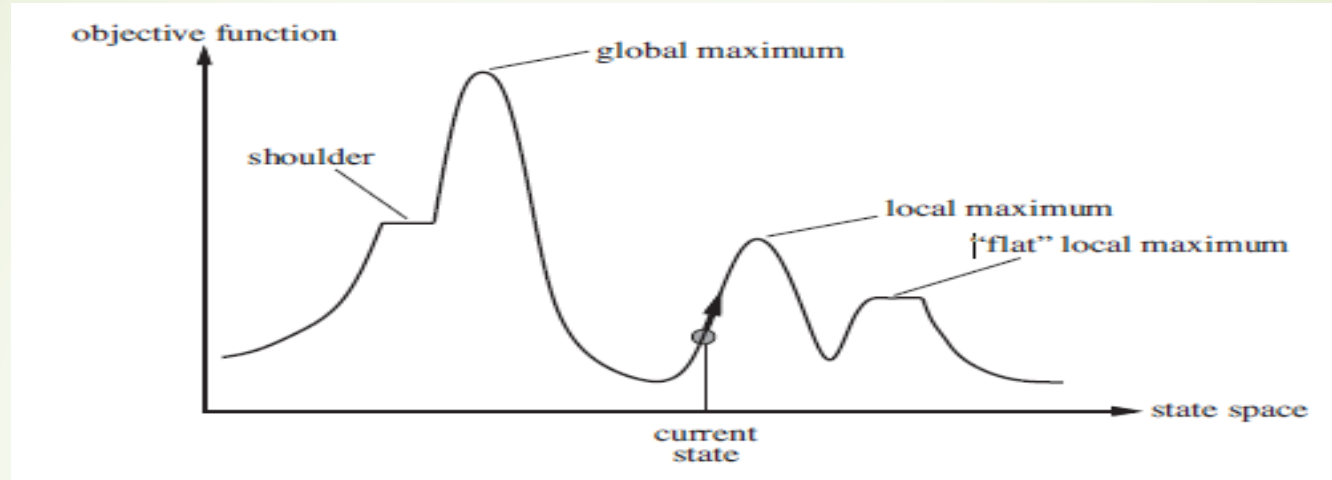
Prepared By:
Dipanita Saha
Assistant Professor
Institute of Information Technology(IIT)
Noakhali Science and Technology University

➤ LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

When a goal is found, the *path* to that goal also constitutes a *solution* to the problem.

- In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added.
- The same general property holds for many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming telecommunications network optimization, vehicle routing, and portfolio management.
- If the path to the goal does not matter, we might consider a different class of algorithms,
- **Local search** algorithms operate using a single **current node** (rather than multiple paths) and generally move only to neighbors of that node.
- Although local search algorithms are not systematic, they have two key advantages: (1) they use very little memory—usually a constant amount;
- and (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
- local search algorithms are useful for solving pure **optimization problems**, in which the aim is to find the best state according to an **objective function**.

- To understand local search, we find it useful to consider the **state-space landscape**.



- A landscape has both “location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function).
- If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**; if elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**.
- Local search algorithms explore this landscape.
- A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.

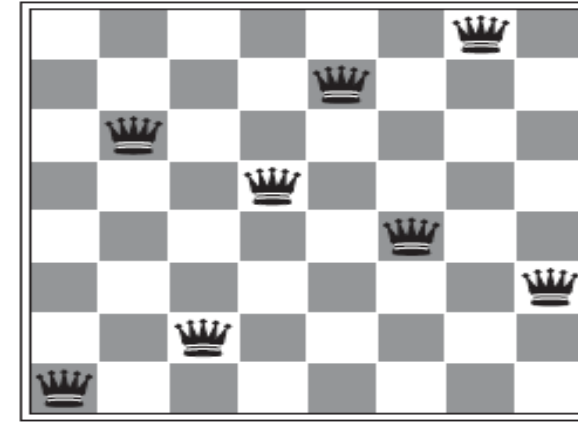
➤ Hill-climbing search

The **hill-climbing** search algorithm (**steepest-ascent** version) is simply a loop that continually moves in the direction of increasing value—that is, uphill.

- It terminates when it reaches a “peak” where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia.
- Local search algorithms typically use a **complete-state formulation**, where each state has 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. The global minimum of this function is zero, which occurs only at perfect solutions.

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 18 | 12 | 14 | 13 | 13 | 12 | 14 | 14 |
| 14 | 16 | 13 | 15 | 12 | 14 | 12 | 16 |
| 14 | 12 | 18 | 13 | 15 | 12 | 14 | 14 |
| 15 | 14 | 14 | ♙ | 13 | 16 | 13 | 16 |
| ♙ | 14 | 17 | 15 | ♙ | 14 | 16 | 16 |
| 17 | ♙ | 16 | 18 | 15 | ♙ | 15 | ♙ |
| 18 | 14 | ♙ | 15 | 15 | 14 | ♙ | 16 |
| 14 | 14 | 13 | 17 | 12 | 14 | 12 | 18 |

(a)



(b)

- Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.
- in Figure 4.3(a), it takes just five steps to reach the state in Figure 4.3(b), which has $h=1$ and is very nearly a solution. Unfortunately, hill climbing often gets stuck for the following reasons:
 - **Local maxima:** a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. More concretely, the state in Figure 4.3(b) is a local maximum every move of a single queen makes the situation worse.

- **Ridges:** a ridge is shown in Figure. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaux:** a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder**, from which progress is possible. A hill-climbing search might get lost on the plateau.



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.

- 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.

➡ 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.

- ➡ To explain simulated annealing, we switch from hill climbing to **gradient descent** and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface.
- ➡ If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum.
- ➡ The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).
- ➡ The innermost loop of the simulated-annealing algorithm 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 and also decreases the “temperature” T goes down:
- ➡ If the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

➤ 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.

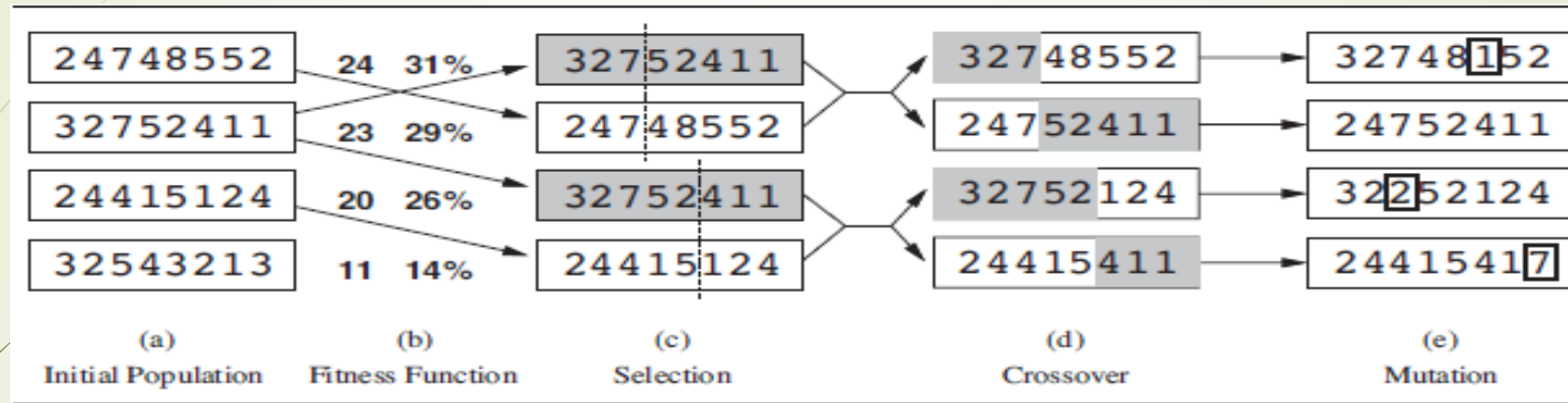


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

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

- In (b), each state is rated by the objective function, or (in GA terminology) the **fitness function**.
- A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of *nonattacking* pairs of queens, which has a value of 28 for a solution.
- The values of the four states are 24, 23, 20, and 11.
- In (c), two pairs are selected at random for reproduction, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all. 4 For each pair to be mated, a **crossover** point is chosen randomly from the positions in the string. In Figure the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.
- In (d), the offspring themselves are created by crossing over the parent strings at the crossover point. For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent.
- Finally, in (e), each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

- genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads.
- The primary advantage, if any, of genetic algorithms comes from the crossover operation.
- it can be shown mathematically that, if the positions of the genetic code are permuted initially in a random order, crossover conveys no advantage.

➤ LOCAL SEARCH IN CONTINUOUS SPACES

- None of the algorithms we have described (except for first-choice hill climbing and simulated annealing) can handle continuous state and action spaces, because they have infinite branching factors.
- This section provides a *very brief* introduction to some local search techniques for finding optimal solutions in continuous spaces.
- Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map (Figure 3.2) to its nearest airport is minimized.
- The state space is then defined by the coordinates of the airports: (x_1, y_1) , (x_2, y_2) , and (x_3, y_3) .

- This is a **six-dimensional space**; we also say that states are defined by six **variables**.
- Moving around in this space corresponds to moving one or more of the airports on the map.
- The objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state once we compute the closest cities.
- Let C_i be the set of cities whose closest airport (in the current state) is airport i . Then, *in the neighborhood of the current state*, where the C_i s remain constant, we have

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

- This expression is correct *locally*, but not globally because the sets C_i are (discontinuous) functions of the state.
- One way to avoid continuous problems is simply to **discretize** the neighborhood of each state. For example, we can move only one airport at a time in either the x or y direction by a fixed amount $\pm\delta$.
- With 6 variables, this gives 12 possible successors for each state.
- We 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

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

- we can find a maximum by solving the equation $\nabla f = 0$. 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,

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

- 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 STEP SIZE constant often called the **step size**.
- 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.

- 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.
- 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 $g(\mathbf{x})=0$. It works by computing a new estimate for the root \mathbf{x} according to Newton’s formula

$$\mathbf{x} \leftarrow \mathbf{x} - g(\mathbf{x})/g'(\mathbf{x}) .$$

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

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

- where $\mathbf{H}_f(\mathbf{x})$ is the **Hessian** matrix of second derivatives, whose elements H_{ij} HESSIAN 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. High-dimensional continuous spaces are, however, big places in which it is easy to get lost.

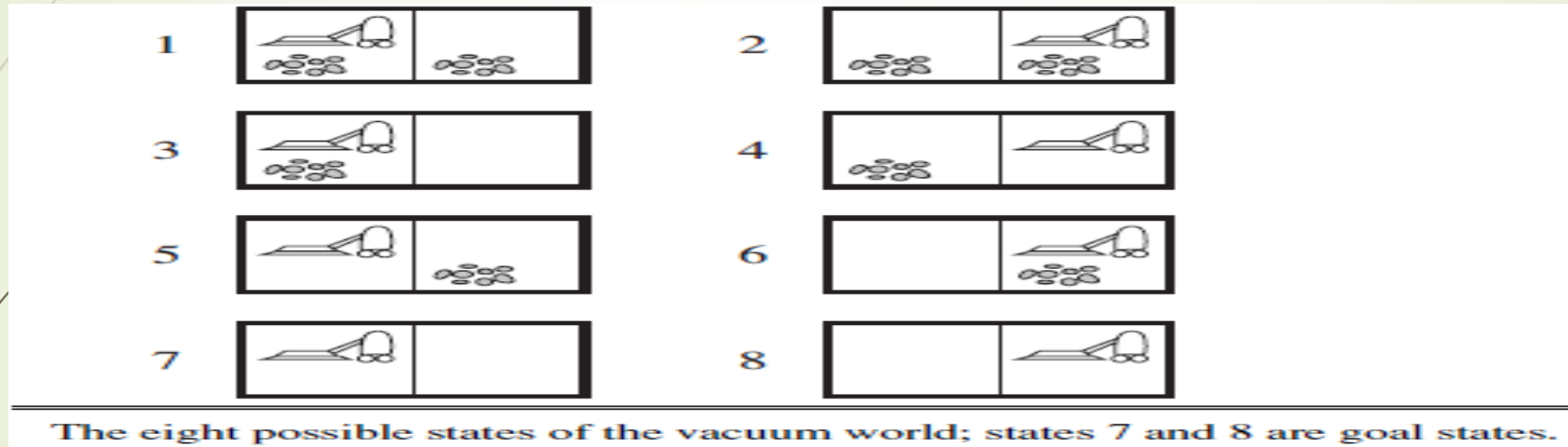
➤ **SEARCHING WITH NONDETERMINISTIC ACTIONS**

- When the environment is either partially observable or nondeterministic (or both), percepts become useful.
- 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** that **specifies** what to do depending on what percepts are received.

➤ The erratic vacuum world

Recall that the state space has eight states. There are three actions—*Left*, *Right*, and *Suck*—and the goal is to clean up all the dirt.

- If the environment is observable, deterministic, and completely known, 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

- For example, in the erratic vacuum world, the *Suck* action in state 1 leads to a state in the set {5, 7}—the dirt in the right-hand square may or may not be vacuumed up.
- We also need to generalize the notion of a **solution** to the problem.
- For example, if we start in state 1, there is no single *sequence* of actions that solves the problem.
- Instead, we need a contingency plan such as the following:

```
[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.
- Many problems in the real, physical world are contingency problems because exact prediction is impossible.

➤ AND–OR search trees

The next question is how to find contingent solutions to nondeterministic problems.

- In a deterministic environment, the only branching is introduced by the agent's own choices in each state. We call these nodes **OR nodes**.
- In the vacuum world, for example, at an OR node the agent chooses *Left or Right or Suck*.
- In a nondeterministic environment, branching is also introduced by the *environment's* choice of outcome for each action. We call these nodes **AND nodes**.
- For example, the *Suck* action in state 1 leads to a state in the set {5, 7}, so the agent would need to find a plan for state 5 *and* for state 7.
- A solution for an AND–OR search problem is a subtree that (1) has a goal node at every leaf, (2) specifies one action at each of its OR nodes, and (3) includes every outcome branch at each of its AND nodes.
- The solution is shown in bold lines in the figure; it corresponds to the plan (The plan uses if–then–else notation to handle the AND branches, but when there are more than two branches at a node, it might be better to use a **case** construct.)

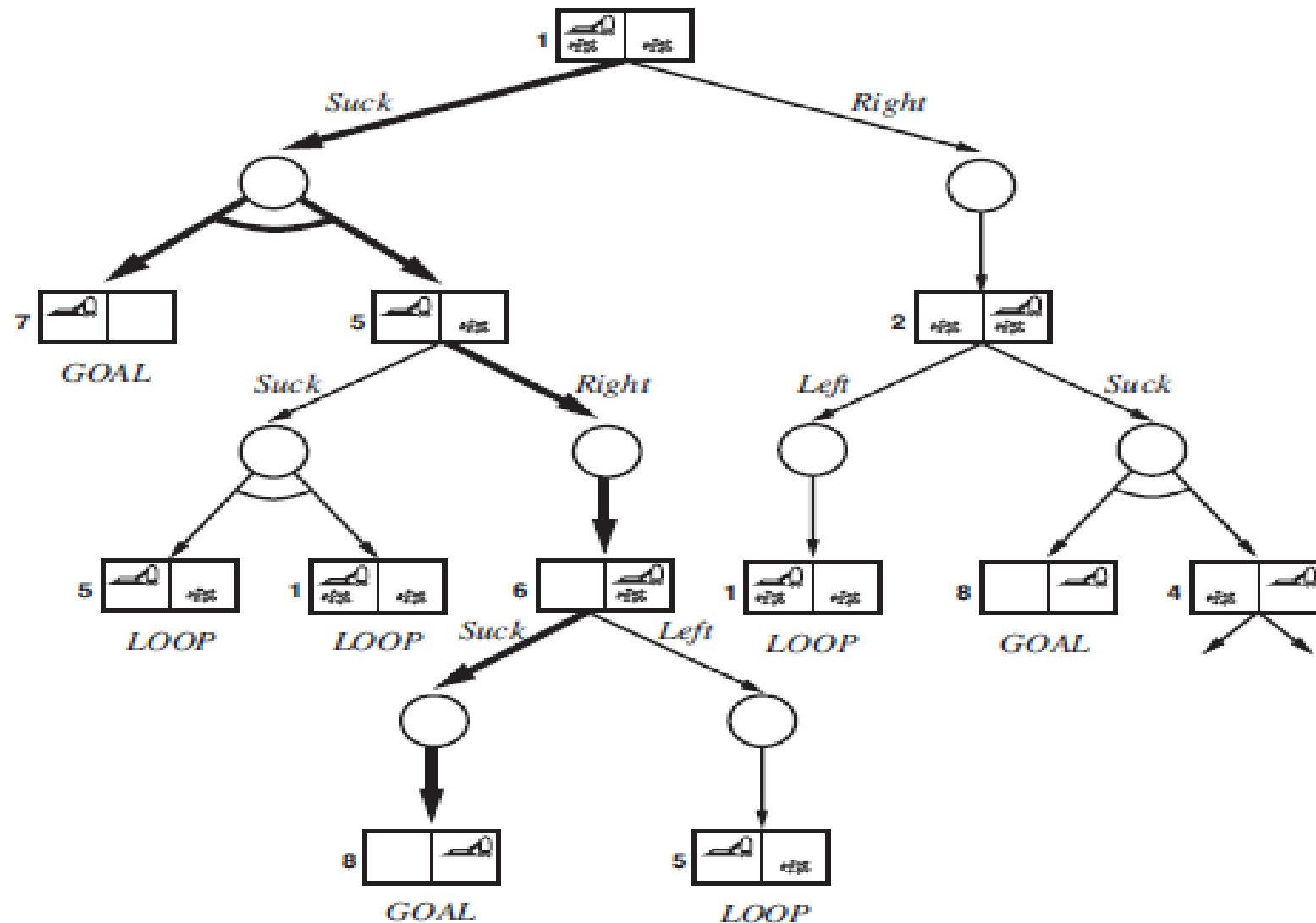

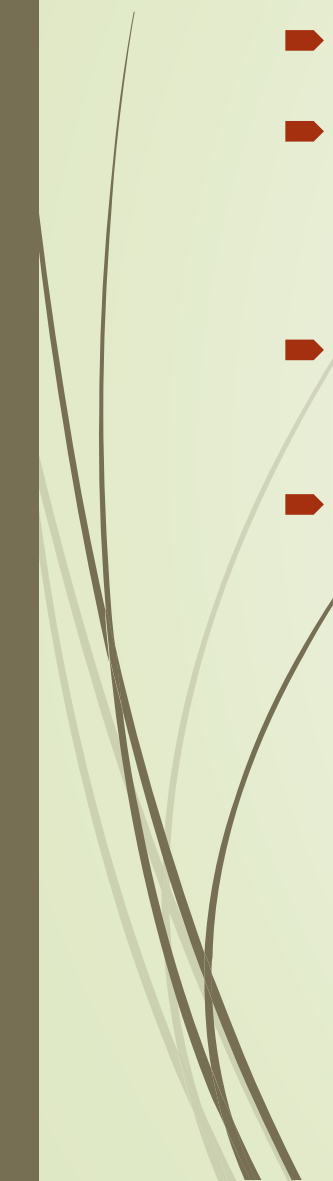


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

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

➤ 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.

- if the agent is in one of several possible states, then an action may lead to one of several possible outcomes—even *if the environment is deterministic*.
- The key concept required for solving partially observable problems is the **belief state**, representing the agent's current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point.

➤ Searching with no observation

- When the agent's percepts provide *no information at all*, we have what is called a **sensor less** problem or sometimes a **conformant** problem.
- At first, one might think the sensor less agent has no hope of solving a problem if it has no idea what state it's in; in fact, sensor less problems are quite often solvable.
- Moreover, sensor less agents can be surprisingly useful, primarily because they *don't* rely on sensors working properly.
- In manufacturing systems, for example, many ingenious methods have been developed for orienting parts correctly from an unknown initial position by using a sequence of actions with no sensing at all.

- The high cost of sensing is another reason to avoid it: for example, doctors often prescribe a broad spectrum antibiotic rather than using the contingent plan of doing an expensive blood test, then waiting for the results to come back, and then prescribing a more specific antibiotic and perhaps hospitalization because the infection has progressed too far.
- in belief-state space, the problem is *fully observable* because the agent always knows its own belief state. Furthermore, the solution (if any) is always a sequence of actions.
- It is instructive to see how the belief-state search problem is constructed.
- Suppose the underlying physical problem P is defined by $ACTIONSP$, $RESULTP$, $GOAL-TESTP$, and $STEP-COSTP$. Then we can define the corresponding sensor less problem as follows:
- **Belief states:** The entire belief-state space contains every possible set of physical states. If P has N states, then the sensor less 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 :

$$ACTIONS(b) = \bigcup_{s \in b} ACTIONS_P(s)$$

- **Transition model:** The agent doesn't know which state in the belief state is the right one; so as far as it knows, it might get to any of the states resulting from applying the action to one of the physical states in the belief state. For deterministic actions, the set of states that might be reached is

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

With deterministic actions, b' is never larger than b . With nondeterminism, we have

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

- The process of generating the new belief state after the action is called the **prediction** step; the notation $b = \text{PREDICT}_P(b, a)$ will come in handy.
- **Goal test:** The agent wants a plan that is sure to work, which means that a belief state satisfies the goal only if *all* the physical states in it satisfy GOAL-TEST_P . The agent may *accidentally* achieve the goal earlier, but it won't *know* that it has done so.
- **Path cost:** This is also tricky. If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of several values. Figure shows the reachable belief-state space for the deterministic, sensor less vacuum world. There are only 12 reachable belief states out of $2^8 = 256$ possible belief states.

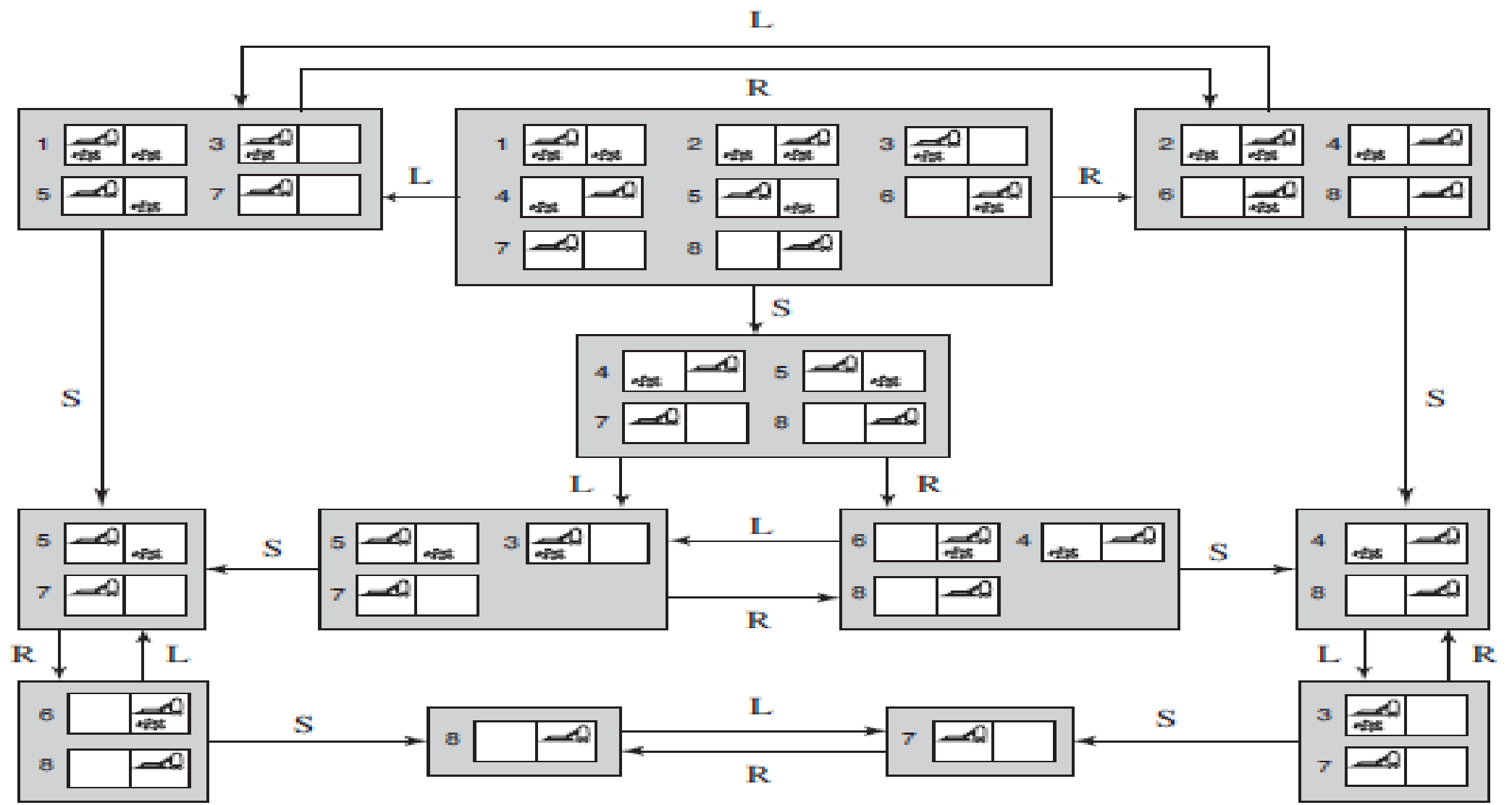


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete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

- in Figure the action sequence [*Suck*, *Left*, *Suck*] starting at the initial state reaches the same belief state as [*Right*, *Left*, *Suck*], namely, {5, 7}.

Now, consider the belief state reached by [*Left*], namely, {1, 3, 5, 7}. Obviously, this is not identical to {5, 7}, but it is a *superset*.

- It is easy to prove that if an action sequence is a solution for a belief state b , it is also a solution for any subset of b . Hence, we can discard a path reaching {1, 3, 5, 7} if {5, 7} has already been generated.
- Conversely, if {1, 3, 5, 7} has already been generated and found to be solvable, then any *subset*, such as {5, 7}, is guaranteed to be solvable.
- This extra level of pruning may dramatically improve the efficiency of sensor less problem solving.
- we can look *inside* the belief states and develop **incremental belief-state search** algorithms that build up the solution one physical state at a time.
- For example, in the sensor less vacuum world, the initial belief state is {1, 2, 3, 4, 5, 6, 7, 8}, and we have to find an action sequence that works in all 8 states.
- We can do this by first finding a solution that works for state 1; then we check if it works for state 2; if not, go back and find a different solution for state 1, and so on.

➤ Searching with observations

For a general partially observable problem, we have to specify how the environment generates percepts for the agent.

- For example, we might define the local-sensing vacuum world to be one in which the agent has a position sensor and a local dirt sensor but has no sensor capable of detecting dirt in other squares.
- The formal problem specification includes a $\text{PERCEPT}(s)$ 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 $\text{PERCEPT}(s)=s$ for every state s , while sensor less problems are a special case in which $\text{PERCEPT}(s)=\text{null}$.
- When observations are partial, it will usually be the case that several states could have produced any given percept.
- For example, the percept [A, Dirty] is produced by state 3 as well as by state 1. Hence, given this as the initial percept, the initial belief state for the local-sensing vacuum world will be {1, 3}.
- The ACTIONS , STEP-COST , and GOAL-TEST are constructed from the underlying physical problem just as for sensor less problems, but the transition model is a bit more complicated.

- We can think of transitions from one belief state to the next for a particular action as occurring in three stages, as shown in Figure

- ❖ The **prediction** stage is the same as for sensor less problems: given the action a in belief state b , the predicted belief state is $b' = \text{PREDICT}(b, a)$.

- ❖ The **observation prediction** stage determines the set of percepts o that could be observed in the predicted belief state: $\text{POSSIBLE-PERCEPTS}(b') = \{o : o = \text{PERCEPT}(s) \text{ and } s \in b'\}$.

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

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

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

$$\text{RESULTS}(b, a) = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b, a), o) \text{ and } o \in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\}.$$

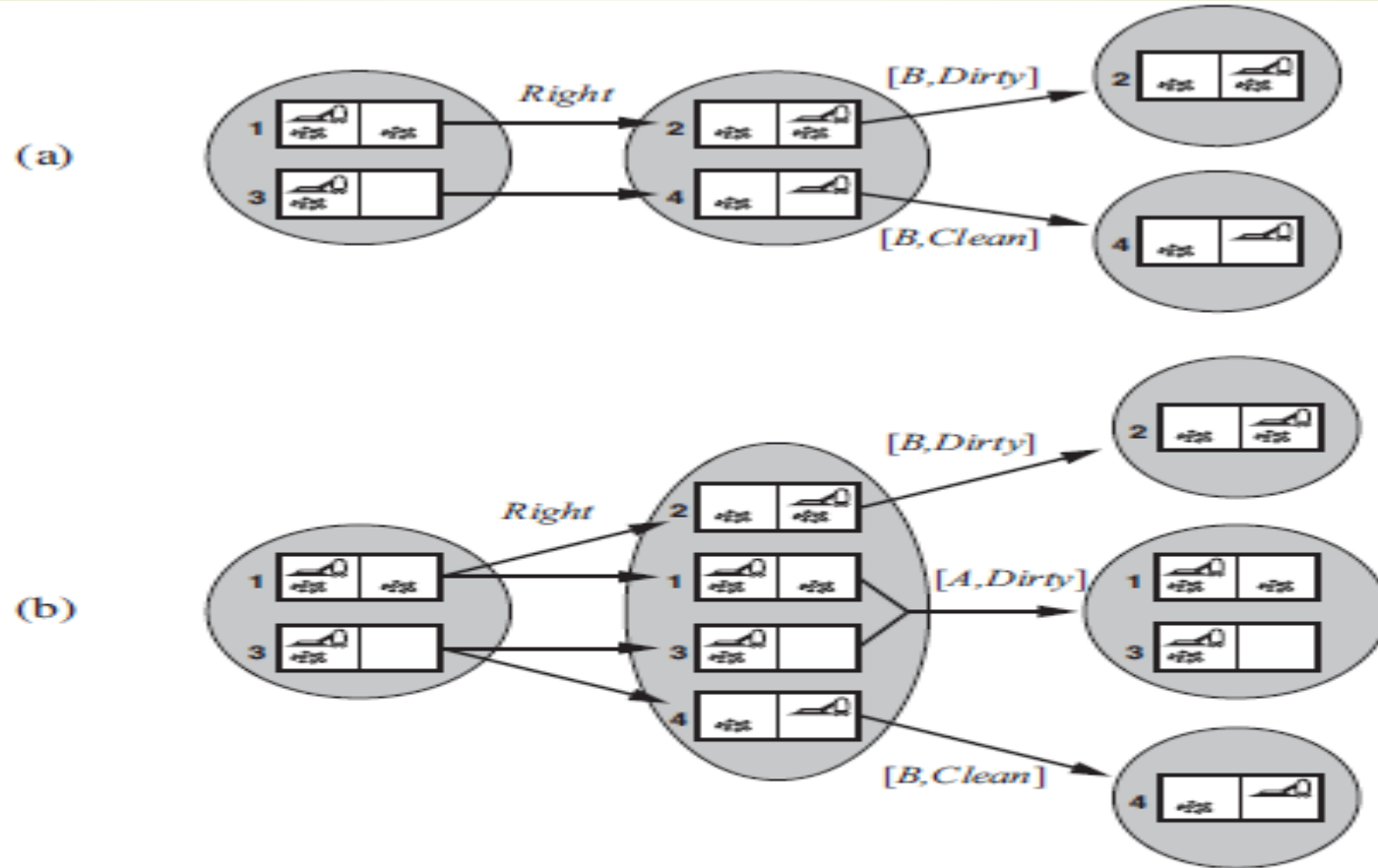


Figure 4.15 Two example of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are $[B, Dirty]$ and $[B, Clean]$, leading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

➤ Online search problems

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

- • **ACTIONS**(s), which returns a list of actions allowed in state s;
 - • The step-cost function $c(s, a, s')$ —note that this cannot be used until the agent knows that s' is the outcome;
 - • and **GOAL-TEST**(s).
- 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, the agent **does not know** that going Up from (1,1) leads to (1,2); nor, having done that, does it know that going Down will take it back to (1,1).
- This degree of ignorance can be reduced in some applications—for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

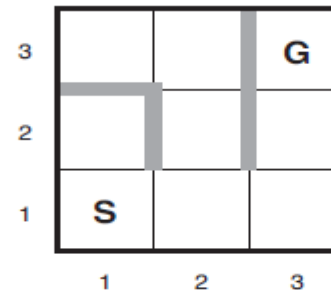


Figure 4.19 A simple maze problem. The agent starts at *S* and must reach *G* but knows nothing of the environment.

- For example, in Figure, the agent might know the location of the goal and be able to use the Manhattan-distance heuristic.

Typically, the agent's objective is to reach a goal state while minimizing cost.

- 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.
- In the language of online algorithms, this is called the **competitive ratio**; we would like it to be as small as possible.
- it is easy to see that the best achievable competitive ratio is infinite in some cases.
- For example, if some actions are **irreversible**— i.e., they lead to a state from which no action leads back to the previous state—the online search might accidentally reach a **dead-end state from which no goal state is reachable**.
- Perhaps the term “accidentally” is unconvincing—after all, there might be an algorithm that happens not to take the dead-end path as it explores.

- Consider the two dead-end state spaces in Figure 4.20(a).
- To an online search algorithm that has visited states S and A, the two state spaces look *identical*, so it must make the same decision in both. Therefore, it will fail in one of them.
- This is an example of an **adversary argument**—we can imagine an adversary constructing the state space while the agent explores it and putting the goals and dead ends wherever it chooses.

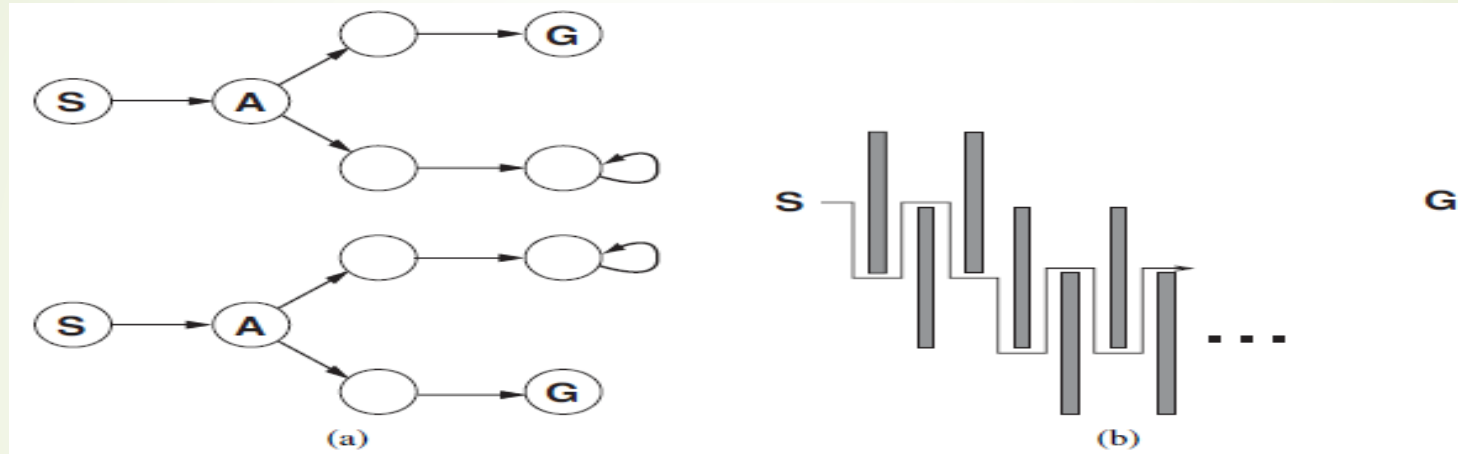


Figure 4.20 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent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

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

- Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost.

This is easy to show in environments with irreversible actions, but in fact it remains true for the reversible case as well, as Figure 4.20(b) shows.

- For this reason, it is common to describe the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

➤ Online local search

- Like depth-first search, **hill-climbing search** has the property of locality in its node expansions.
- because it keeps just one current state in memory, hill-climbing search is *already* an online search algorithm!
- Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go.
- Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.
- Instead of random restarts, one might consider using a **random walk** to explore the environment.
- A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried

- Figure 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.

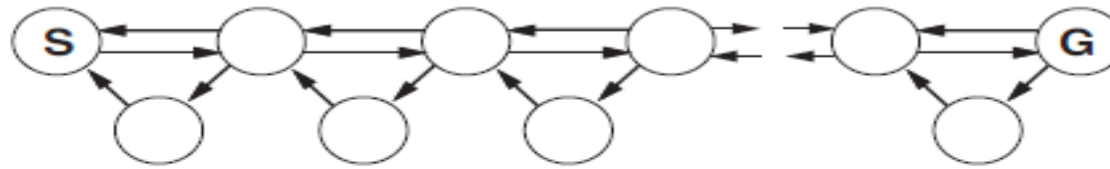
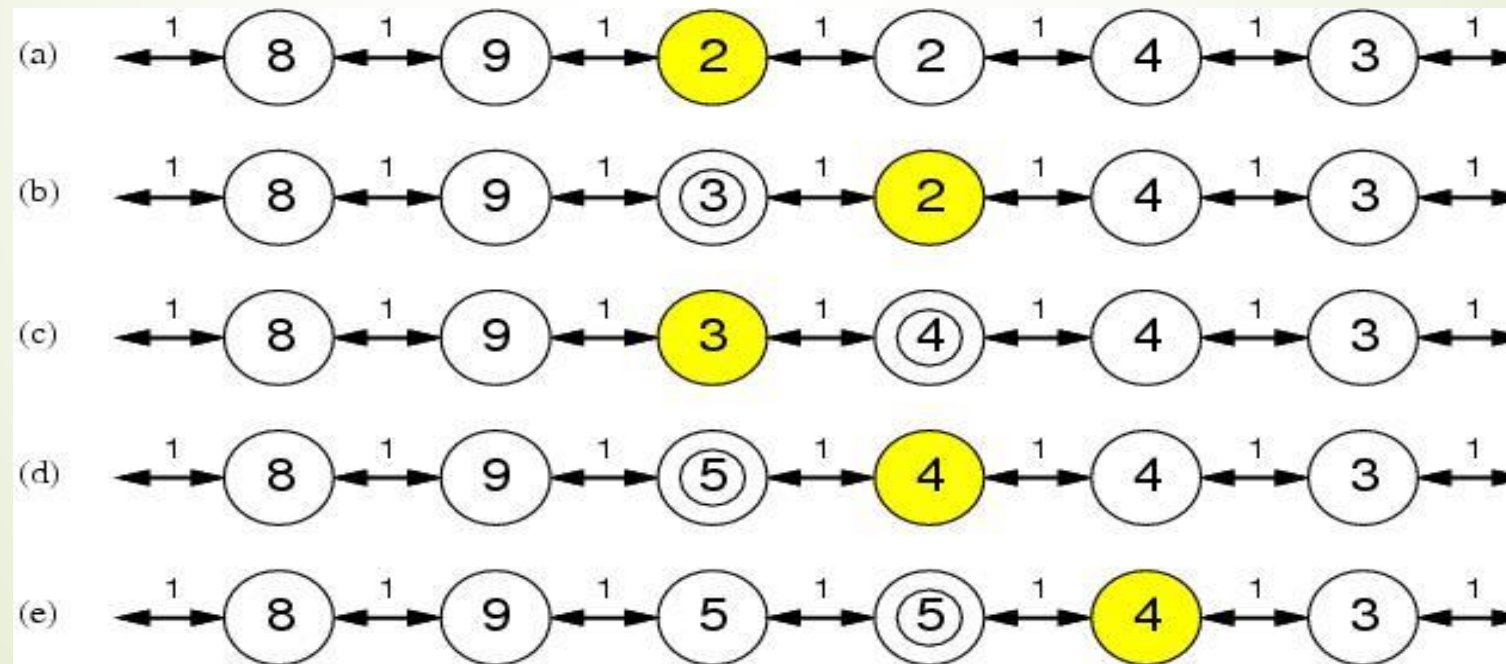


Figure 4.22 An environment in which a random walk will take exponentially many steps to find the goal.

- Learning real-time A* (LRTA*)



function LRTA*-COST(s, a, s', H) **return** an cost estimate

if s' is undefined the return $h(s)$

else return $c(s, a, s') + H[s']$

function LRTA*-AGENT(s') **return** an action

input: s' , a percept identifying current state

static: *result*, a table indexed by action and state, 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** $H[s'] \leftarrow h(s')$

unless s is null

$result[a, s] \leftarrow s'$

$H[s] \leftarrow \min_{b \in ACTIONS(s)} LRTA^*-COST(s, b, result[b, s], H)$

$a \leftarrow$ an action b in $ACTIONS(s')$ that minimizes $LRTA^*-COST(s', b, result[b, s'], H)$

$s \leftarrow s'$

return a