Chapter 4 - Search in Complex Environments

Russell, S., & Norvig, P. (2022). *Artificial Intelligence - A Modern Approach* (4th global ed.). Pearson.

Contents

- 1. Local Search and Optimization Problems
- 2. Local Search in Continuous Spaces
- 3. Search with Nondeterministic Actions
- 4. Search in Partially Observable Environments
- 5. Online Search Agents and Unknown Environments (Reading the textbook)

Search in Complex Environments

- Chapter 3 addressed problems in fully observable, deterministic, static, known environments where the solution is a sequence of actions.
- In this chapter, we relax those constraints.
  - We begin with the problem of finding a good state without worrying about the path to get there, covering both discrete and continuous states.
  - Then we relax the assumptions of determinism and observability.
    - In a nondeterministic world, the agent will need a conditional plan and carry out different actions depending on what it observes—for example, stopping if the light is red and going if it is green.
    - With partial observability, the agent will also need to keep track of the possible states it might be in.

1. Local Search and Optimization Problems

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

1. Local Search and Optimization Problems

- To understand local search, consider the states of a problem laid out in a **state-space landscape**.
- Each point (state) in the landscape has an "elevation," defined by the value of the objective function.
- If elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**—and we call the process **hill climbing**.
- If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**—and we call it **gradient descent**.

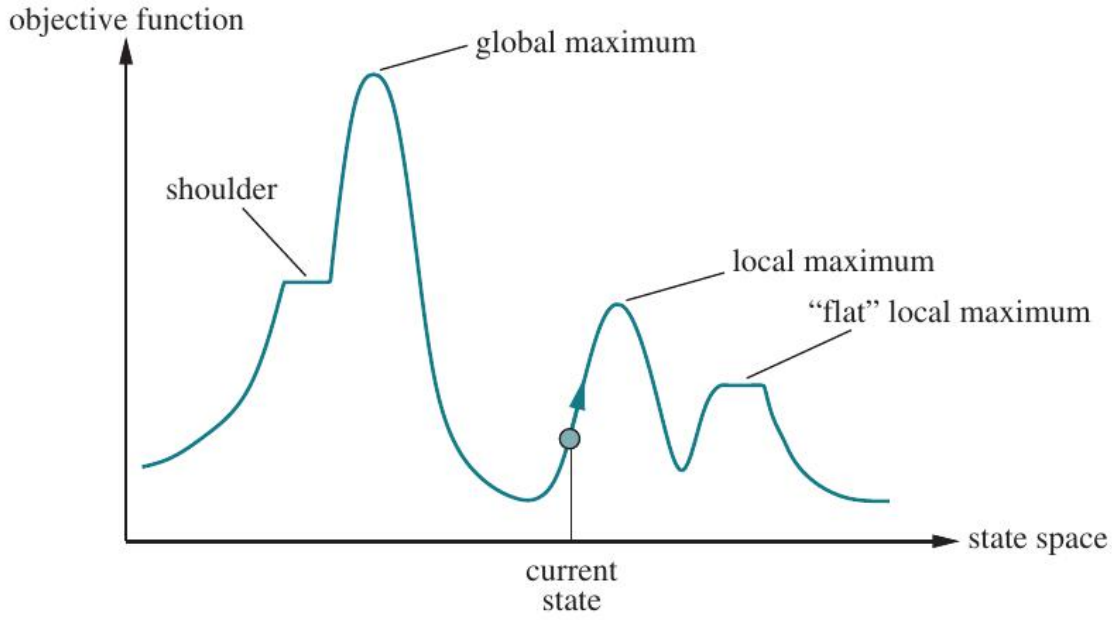


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

1. Local Search and Optimization Problems

Hill-Climbing Search

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

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

- It keeps track of one current state and on each iteration moves to the neighboring state with highest value—that is, it heads in the direction that provides the **steepest ascent**.
- It terminates when it reaches a "peak" where no neighbor has a higher value.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state.

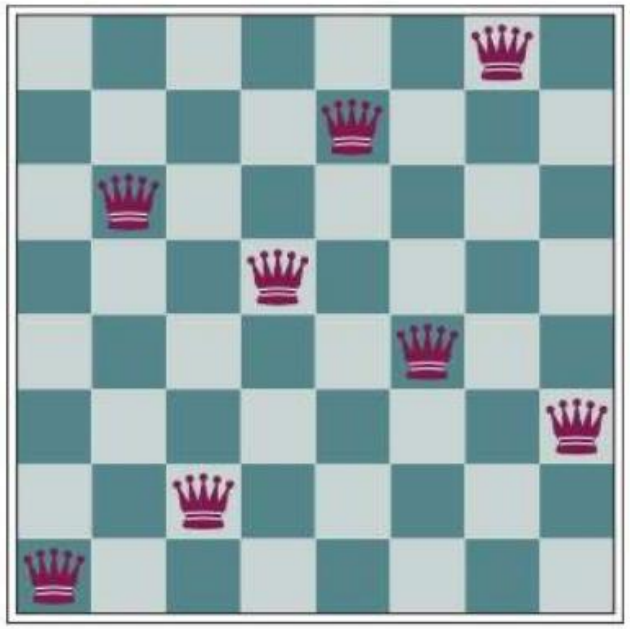
1. Local Search and Optimization Problems

Hill-Climbing Search

- The **8-queens problem**
  - **Complete-state formulation:** Every state has all the components of a solution, but they might not all be in the right place.
  - In this case every state has 8 queens on the board, one per column.
  - The initial state is chosen at random, and the successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has $8 \times 7 = 56$ successors).
  - The heuristic cost function h is the number of pairs of queens that are attacking each other; this will be zero only for solutions.

1. Local Search and Optimization Problems

Hill-Climbing Search



(a)

(b)

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

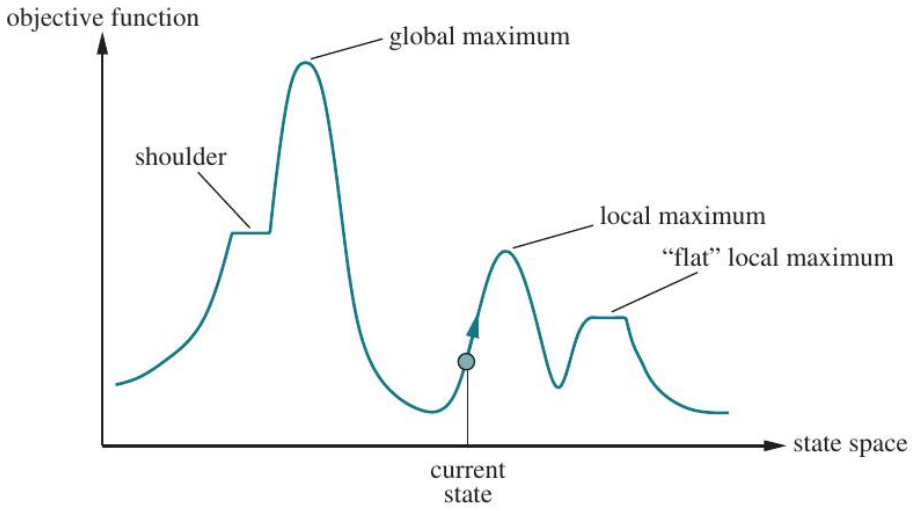
1. Local Search and Optimization Problems

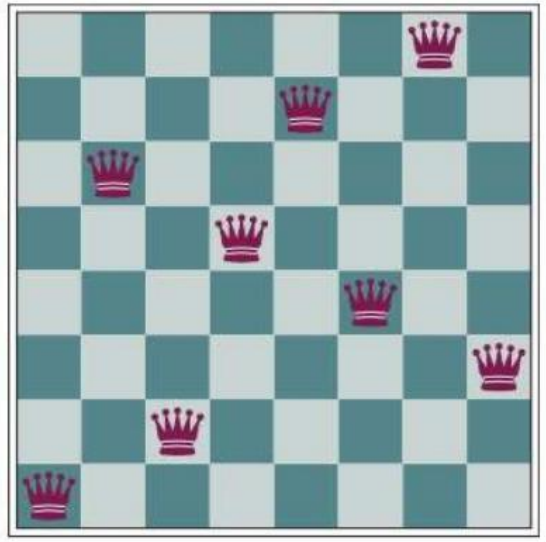
Hill-Climbing Search

- Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.
- Although greedy is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well.
  - Hill climbing can make rapid progress toward a solution because it is usually quite easy to improve a bad state.

1. Local Search and Optimization Problems
Hill-Climbing Search

- Unfortunately, hill climbing can get stuck for any of 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.
    - Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.



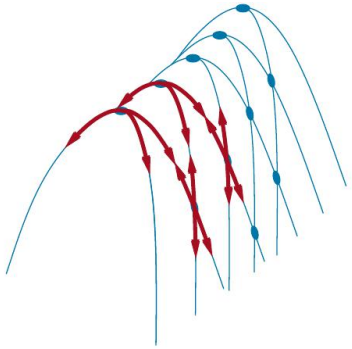


- The state in the figure is a local maximum (i.e., a local minimum for the cost *h*); every move of a single queen makes the situation worse.

1. Local Search and Optimization Problems

Hill-Climbing Search

– **Ridges**: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

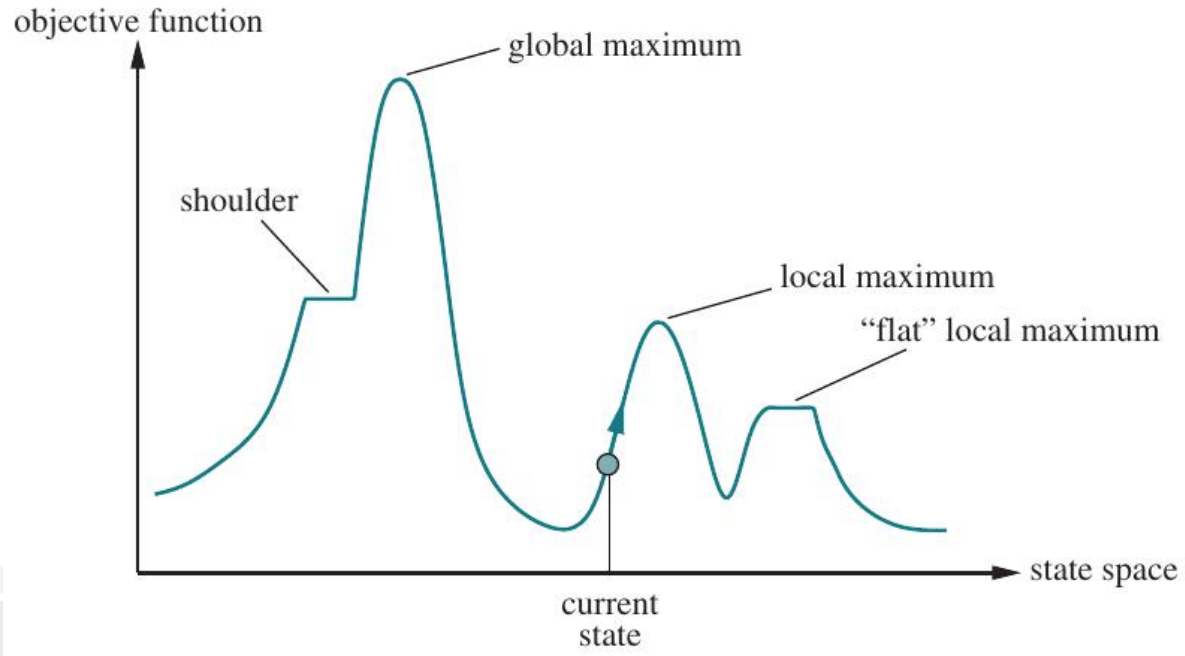


- This graph is a surface, maybe containing hill-like and valley-like shapes.
- One such shape is a ridge, much like you would find on a mountain.
  - If you walk along the top of the ridge, you ascend the hill.
  - If you walk perpendicular to the top of the ridge, then you will climb to the top and descend down the other side.
- Now, remember that in hill climbing, we aren't allowed to go in every direction.
  - The case of a two-variable function $f(x, y)$: So for a given point (x_0, y_0), we either make a small change in the x direction or a small change in the y direction to obtain either a new function value $f(x_0 + d, y_0)$ or $f(x_0, y_0 +$ d). This means we can only travel in directions parallel to the x and y axes. But what if we have a ridge that runs diagonally to both axes?

1. Local Search and Optimization Problems

Hill-Climbing Search

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



1. Local Search and Optimization Problems
Hill-Climbing Search

- Variants of hill climbing
  - **Stochastic hill climbing**
    - **Stochastic hill climbing** randomly selects a neighboring solution to move to, rather than always picking the steepest ascent.
    - It addresses the problem of getting stuck in local optima by allowing for non-greedy moves.
    - This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
    - **First-choice hill climbing:** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.
  - **Random-restart hill climbing**
    - "If at first you don't succeed, try, try again."
    - It 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: if there are few local maxima and plateaus, random-restart hill climbing will find a good solution very quickly.

1. Local Search and Optimization Problems

Simulated Annealing

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

1. Local Search and Optimization Problems

Simulated Annealing

- 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.
- The overall structure of the simulated-annealing algorithm is similar to hill climbing.
  - Instead of picking the best move, however, it picks a random move.
  - If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.
  - The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down.
    - Prob(ΔE) ~ $exp(\Delta E/T)$ (Boltzman distribution)
  - Temperature schedule: The temperature schedule determines how the temperature of the system changes over time. In the beginning, the temperature is high so that the algorithm can explore a wide range of solutions, even if they are worse than the current solution. As the iterations increase, the temperature gradually decreases. Hence, the algorithm becomes more selective and accepts better solutions with higher probability.
    $T \leftarrow \alpha T \quad (\alpha < 1)$

1. Local Search and Optimization Problems

Simulated Annealing

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

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

1. Local Search and Optimization Problems

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 variant called **stochastic beam search**, analogous to stochastic hill climbing. Instead of choosing the top k successors, stochastic beam search chooses successors with probability proportional to the successor's value, thus increasing diversity.

1. Local Search and Optimization Problems

Evolutionary Algorithms

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

1. Local Search and Optimization Problems
Evolutionary Algorithms

- There are endless forms of evolutionary algorithms, varying in the following ways:
  - The size of the population
  - The representation of each individual. In genetic algorithms, each individual is a string over a finite alphabet (often a Boolean string), just as DNA is a string over the alphabet ACGT.
  - The mixing number, ρ, which is the number of parents that come together to form offspring.
    - The most common case is $\rho = 2$: two parents combine their "genes" (parts of their representation) to form offspring.
    - When $\rho = 1$ we have stochastic beam search (which can be seen as asexual reproduction).
    - It is possible to have $\rho > 2$, which occurs only rarely in nature but is easy enough to simulate on computers.

1. Local Search and Optimization Problems
Evolutionary Algorithms

- The **selection** process for selecting the individuals who will become the parents of the next generation: one possibility is to select from all individuals with probability proportional to their fitness score. Another possibility is to randomly select n individuals ($n > \rho$), and then select the ρ most fit ones as parents.
- The **recombination** procedure. One common approach (assuming $\rho = 2$), is to randomly select a **crossover point** to split each of the parent strings, and recombine the parts to form two children, one with the first part of parent 1 and the second part of parent 2; the other with the second part of parent 1 and the first part of parent 2.

1. Local Search and Optimization Problems

Evolutionary Algorithms

- The **mutation rate**, which determines how often offspring have random mutations to their representation. Once an offspring has been generated, every bit in its composition is flipped with probability equal to the mutation rate.
- The makeup of the next generation. This can be just the newly formed offspring, or it can include a few top-scoring parents from the previous generation (a practice called **elitism**, which guarantees that overall fitness will never decrease over time). The practice of **culling**, in which all individuals below a given threshold are discarded, can lead to a speedup.

1. Local Search and Optimization Problems
Evolutionary Algorithms

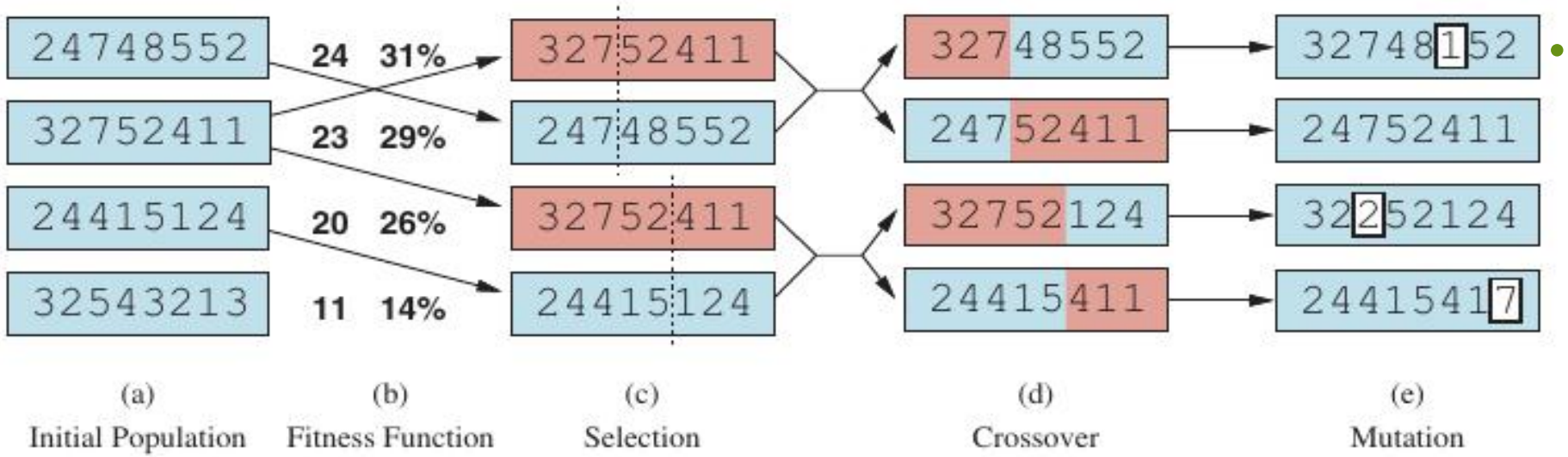


Figure 4.6 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

- Figure 4.6(a) shows a population of four 8-digit strings, each representing a state of the 8-queens puzzle:
  - The *c*-th digit represents the row number of the queen in column *c*.

- In (b), each state is rated by the fitness function.
  - Higher fitness values are better, so for the 8-queens problem we use the number of *nonattacking* pairs of queens, which has a value of 8 × 7/2 =28 for a solution. The values of the four states in (b) are 24, 23, 20, and 11. The fitness scores are then normalized to probabilities, and the resulting values are shown next to the fitness values in (b).

1. Local Search and Optimization Problems

Evolutionary Algorithms

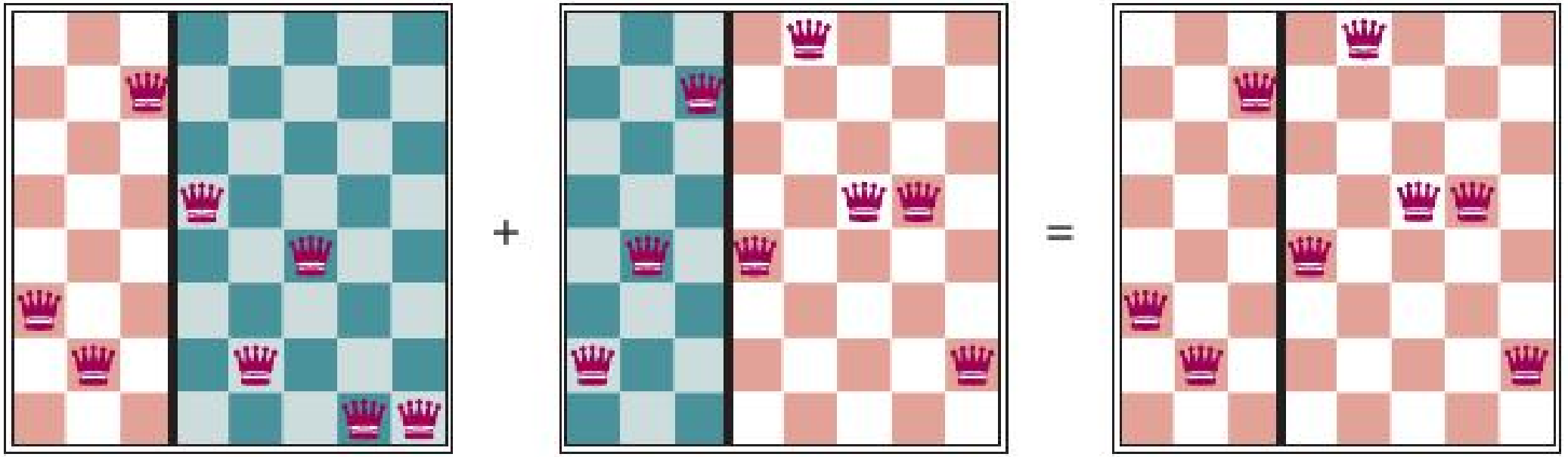


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The green columns are lost in the crossover step and the red columns are retained. (To interpret the numbers in Figure 4.6: row 1 is the bottom row, and 8 is the top row.)

1. Local Search and Optimization Problems
Evolutionary Algorithms

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

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

- *population* is an ordered list of individuals.
- *weights* is a list of corresponding fitness values for each individual.
- *fitness* is a function to compute these values.

2. Local Search in Continuous Spaces

- A continuous action space has an infinite branching factor, and thus can't be handled by most of the algorithms we have covered so far (with the exception of first-choice hill climbing and simulated annealing).
- A very brief introduction to some local search techniques for continuous spaces:
  - Example: Place three new airports anywhere in Romania, such that the sum of squared straight-line distances from each city on the map to its nearest airport is minimized.
  - The state space is then defined by the coordinates of the three airports: (x_1, y_1), (x_2, y_2), and (x_3, y_3).
  - This is a six-dimensional space; we also say that states are defined by six **variables**.
- In general, states are defined by an n-dimensional vector of variables, **x**.

2. Local Search in Continuous Spaces

- The objective function $f(\mathbf{x}) = f(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state once we compute the closest cities.
- Let C_i be the set of cities whose closest airport (in the state $\mathbf{x}$) is airport i.

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

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

2. Local Search in Continuous Spaces

- One way to deal with a continuous state space is to **discretize** it.
  - For example, instead of allowing the (x_i, y_i) locations to be any point in continuous two-dimensional space, we could limit them to fixed points on a rectangular grid with spacing of size δ (delta).
  - Then instead of having an infinite number of successors, each state in the space would have only 12 successors, corresponding to incrementing one of the 6 variables by $\pm\delta$.
  - We can then apply any of our local search algorithms to this discrete space.

2. Local Search in Continuous Spaces

- Often we have an objective function expressed in a mathematical form such that we can use calculus to solve the problem analytically rather than empirically. Many methods attempt to use the **gradient** of the landscape to find a maximum.
- The gradient of the objective function is a vector ∇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.
  - 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*)

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

2. Local Search in Continuous Spaces

- 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 (alpha) is a small constant often called the **step size**.

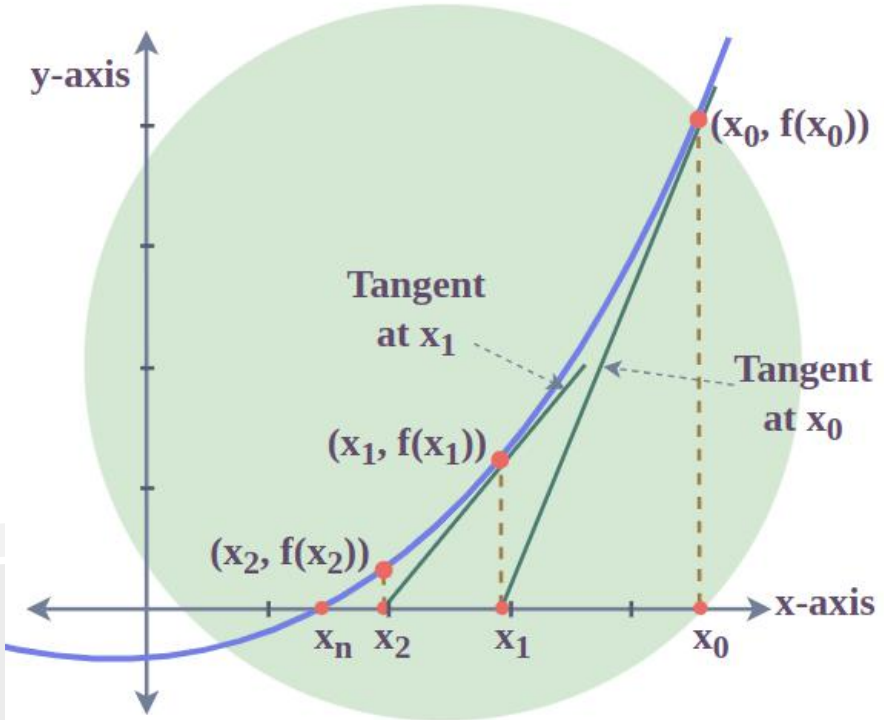
  - The basic problem is that if α is too small, too many steps are needed; if α is too large, the search could overshoot the maximum.

- There exist a huge variety of methods for adjusting α.

2. Local Search in Continuous Spaces

- **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.
- **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

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



2. Local Search in Continuous Spaces

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

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

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

3. Search with Nondeterministic Actions

- A fully observable, deterministic, known environment
  - An agent can observe the initial state, calculate a sequence of actions that reach the goal, and execute the actions with its "eyes closed," never having to use its percepts.
- A partially observable environment
  - The agent doesn't know for sure what state it is in.
- A nondeterministic environment
  - The agent doesn't know what state it transitions to after taking an action.
- A partially observable and nondeterministic environment
  - That means that rather than thinking "I'm in state s_1 and if I do action a I'll end up in state s_2," an agent will now be thinking "I'm either in state s_1 or s_3, and if I do action a I'll end up in state s_2, s_4 or s_5." We call a set of physical states that the agent believes are possible a **belief state**.
  - The solution to a problem is no longer a sequence, but rather a **conditional plan** (sometimes called a contingency plan or a strategy) that specifies what to do depending on what percepts agent receives while executing the plan.

3. Search with Nondeterministic Actions

- The vacuum world
  - The environment is fully observable, deterministic, and completely known → 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.

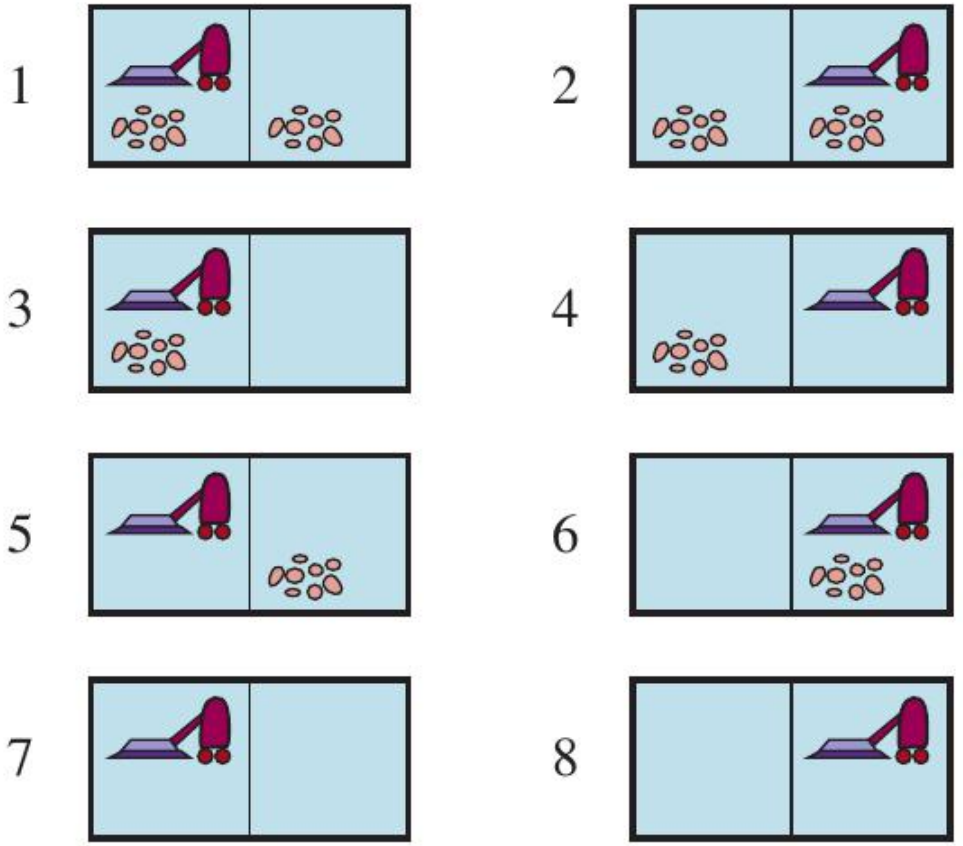
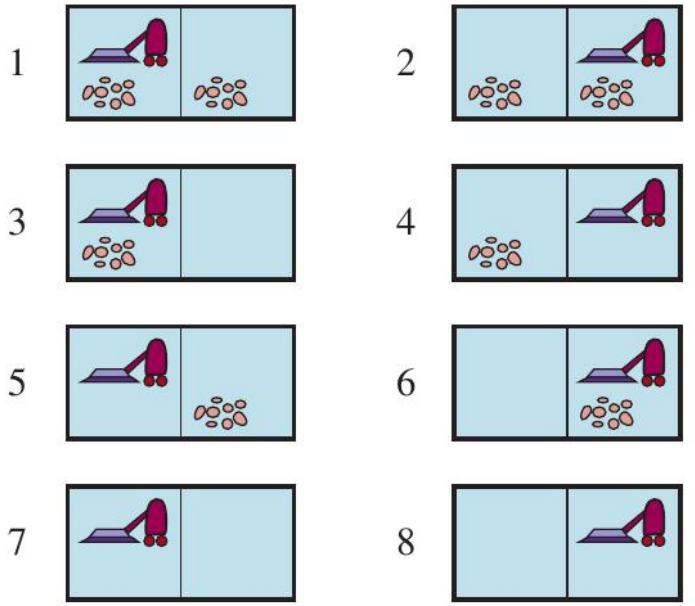


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

3. Search with Nondeterministic Actions

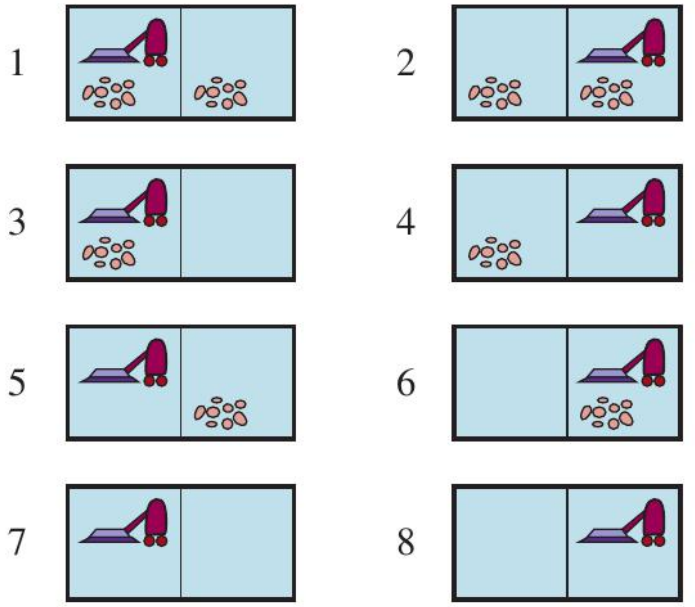
The Erratic Vacuum World



- *Nondeterministic* environment: 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.
- The RESULTS function that returns a set of possible outcome states.
  - For example, in the erratic vacuum world, the *Suck* action in state 1 cleans up either just the current location, or both locations:
    - RESULTS(1, *Suck*) = {5, 7}

3. Search with Nondeterministic Actions

The Erratic Vacuum World



- If we start in state 1, no single *sequence* of actions solves the problem, but the following **conditional plan** does:
  - [*Suck*, **if** *state* = 5 **then** [*Right*, *Suck*] **else** []]
  - Here we see that a conditional plan can contain **if–then–else** steps; this means that solutions are *trees* rather than sequences.
  - Here the conditional in the **if** statement tests to see what the current state is; this is something *the agent will be able to observe at runtime*, but doesn't know at planning time. Alternatively, we could have had a formulation that tests the percept rather than the state.
- Many problems in the real, physical world are contingency problems, because exact prediction of the future is impossible. For this reason, many people keep their eyes open while walking around.

3. Search with Nondeterministic Actions
AND-OR Search Trees

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

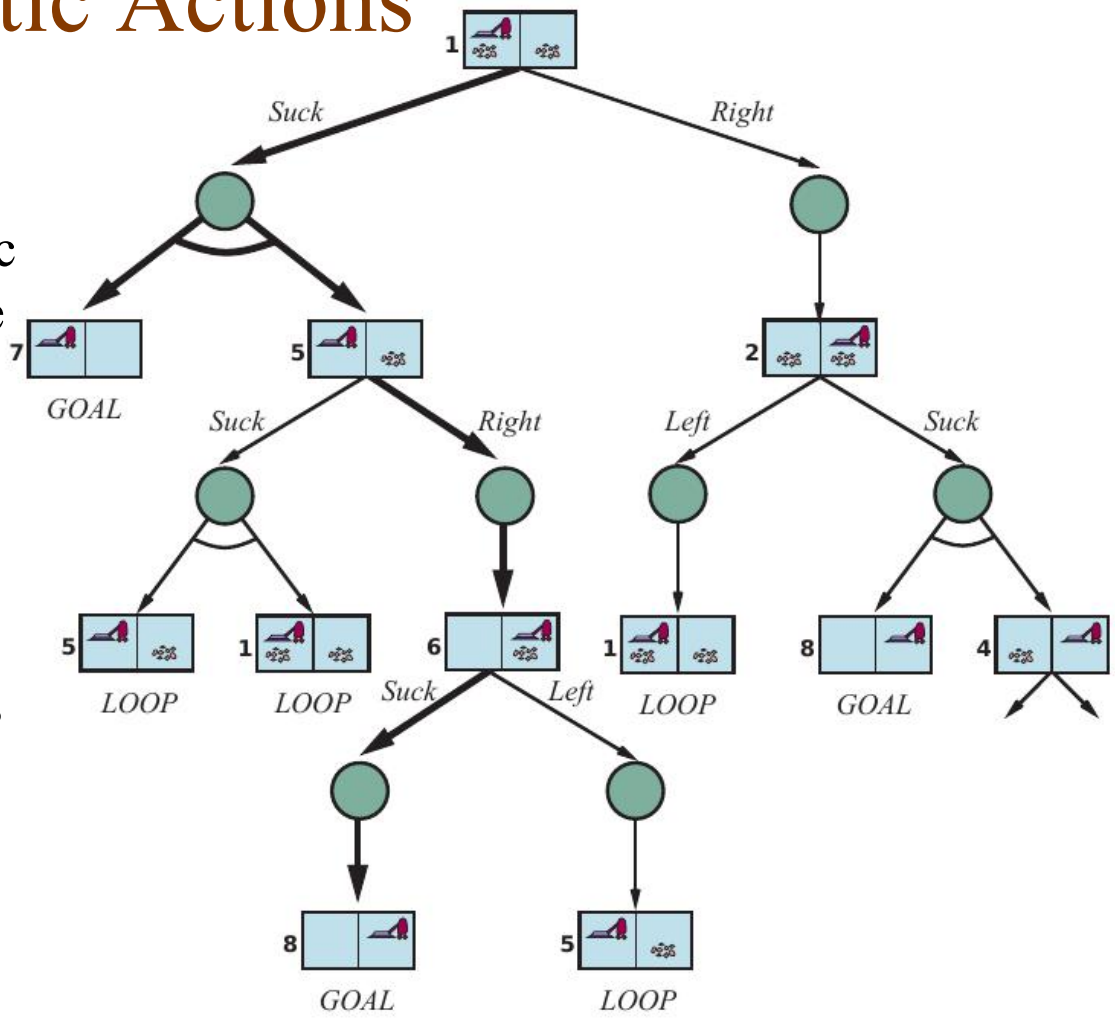


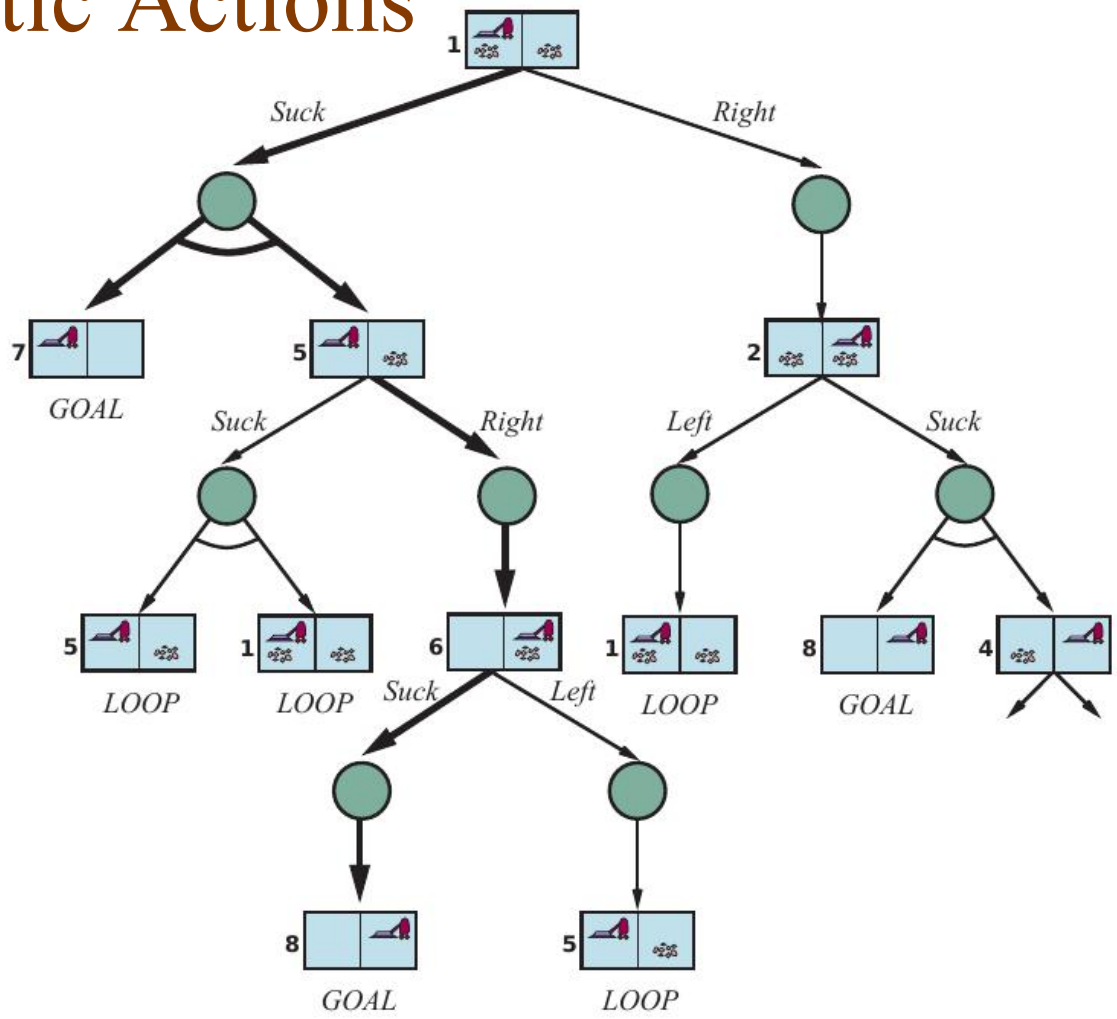
Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

3. Search with Nondeterministic Actions

AND-OR Search Trees

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

[*Suck*, **if** *state* = 5 **then** [*Right*, *Suck*] **else** []]



3. Search with Nondeterministic Actions

AND-OR Search Trees

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

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

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

Figure 4.11 An algorithm for searching AND –OR graphs generated by nondeterministic environments.

A recursive, depth-first algorithm for AND –OR graph search

A solution is a conditional plan that considers every nondeterministic outcome and makes a plan for each one.

3. Search with Nondeterministic Actions

Try, Try Again

- Consider a *slippery* vacuum world, which is identical to the ordinary (non-erratic) vacuum world except that movement actions sometimes fail, leaving the agent in the same location.
- For example, moving *Right* in state 1 leads to the belief state {1, 2}.
- Figure 4.12 shows part of the search graph; clearly, there are no longer any acyclic solutions from state 1, and AND-OR-SEARCH would return with failure.

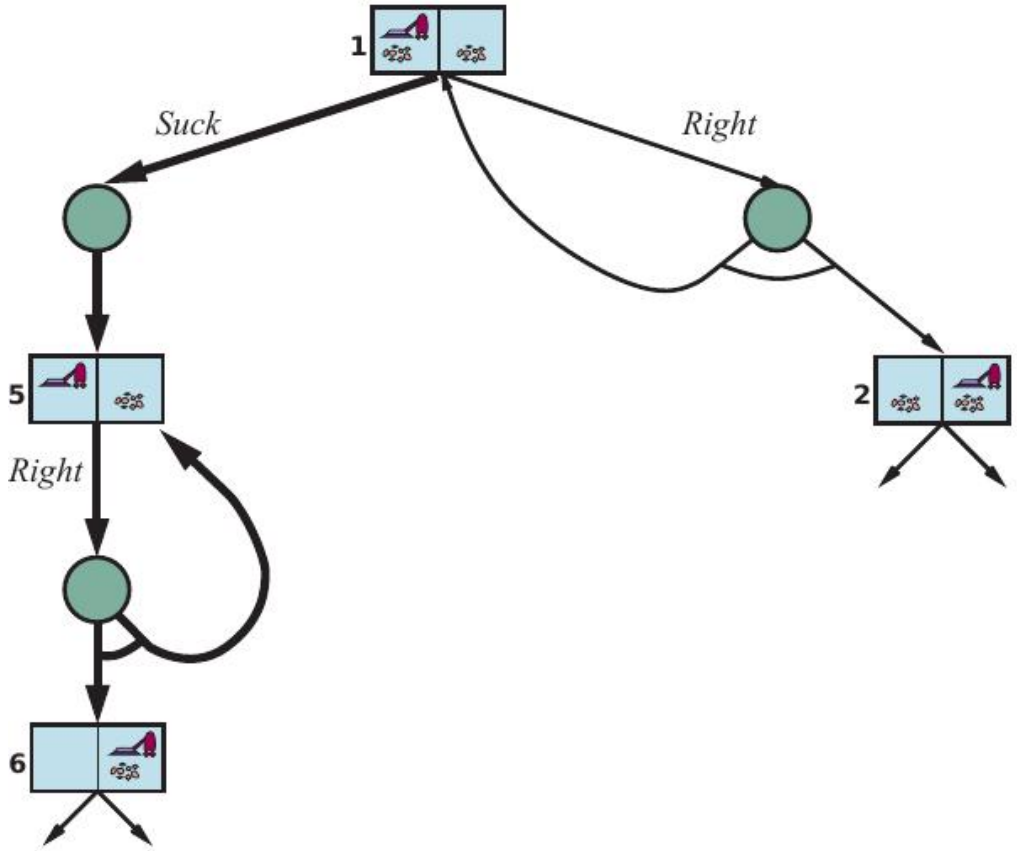
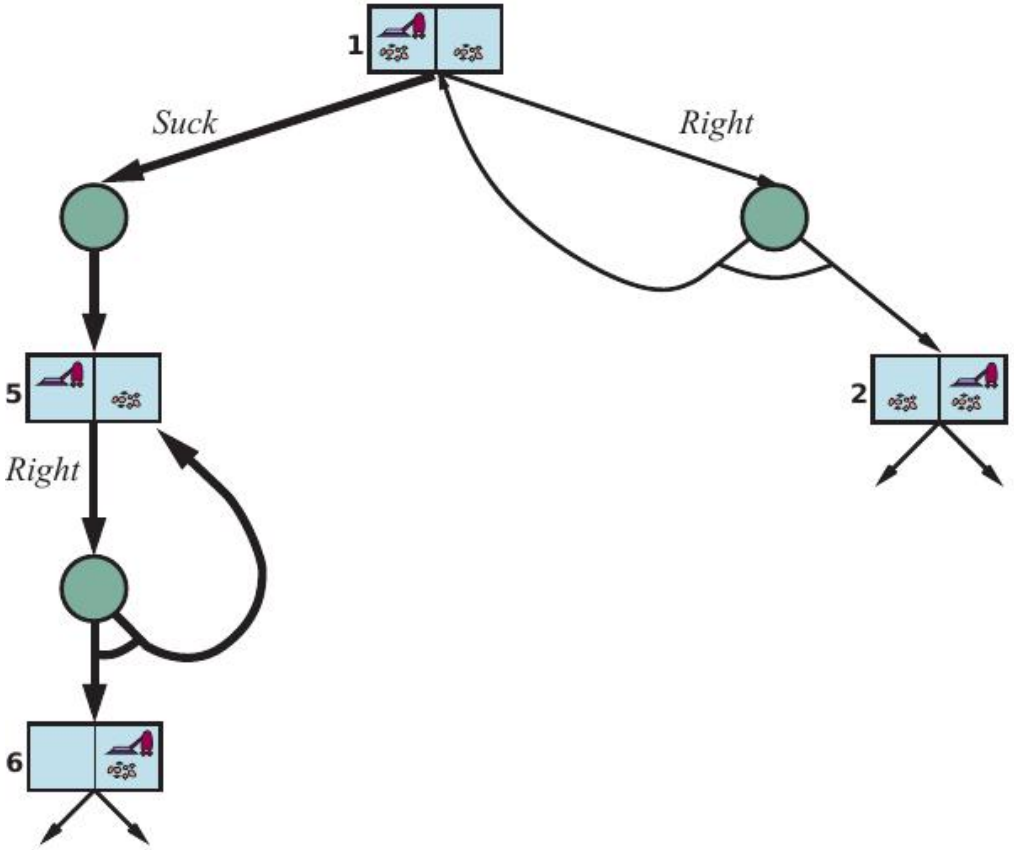


Figure 4.12 Part of the search graph for a slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

3. Search with Nondeterministic Actions

Try, Try Again

- There is, however, a **cyclic solution**, which is to keep trying *Right* until it works.
- **while** construct:
  - [*Suck*, **while** *State* = 5 **do** *Right, Suck*]
- or by adding a **label** to denote some portion of the plan and referring to that label later:
  - [*Suck*, L_1 : *Right*, **if** *State* = 5 **then** L_1 **else** *Suck*].



4. Search in Partially Observable Environments

- The agent's percepts are not enough to pin down the exact state. That means that some of the agent's actions will be aimed at reducing uncertainty about the current state.

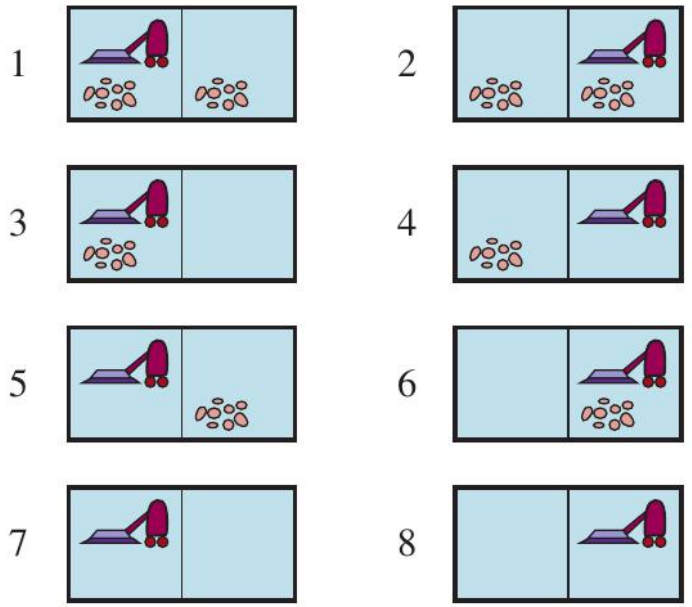
4. Search in Partially Observable Environments

Searching With No Observation

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

4. Search in Partially Observable Environments

Searching With No Observation



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

4. Search in Partially Observable Environments

Searching With No Observation

- The solution to a sensorless problem is a sequence of actions, not a conditional plan (because there is no perceiving). But we search in the space of belief states rather than physical states.
  - In a fully observable environment, each belief state contains one physical state. Thus, we can view the algorithms in Chapter 3 as searching in a belief-state space of singleton belief states.
- In belief-state space, the problem is *fully observable* because the agent always knows its own belief state.
- Furthermore, the solution (if any) for a sensorless problem is always a sequence of actions. This is true *even if the environment is nondeterministic*.

4. Search in Partially Observable Environments

Searching With No Observation

- We can use the existing algorithms from Chapter 3 if we transform the underlying physical problem into a belief-state problem, in which we search over belief states rather than physical states.
- The original problem, P, has components $Actions_P$, $Result_P$ etc., and the belief-state problem has the following components:
  - **States**: The belief-state space contains every possible subset of the physical states. If P has N states, then the belief-state problem has 2^N belief states, although many of those may be unreachable from the initial state.
  - **Initial state**: Typically the belief state consisting of all states in P, although in some cases the agent will have more knowledge than this.

4. Search in Partially Observable Environments

Searching With No Observation

- **Actions**: This is slightly tricky. Suppose the agent is in belief state $b = \{s_1, s_2\}$, but $\text{ACTIONS}_P(s_1) \neq \text{ACTIONS}_P(s_2)$; then the agent is unsure of which actions are legal.
  - If we assume that illegal actions have *no effect on the environment*, then it is safe to take the *union* of all the actions in any of the physical states in the current belief state b:

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

  - On the other hand, if an illegal action might lead to catastrophe, it is safer to allow only the *intersection*, that is, the set of actions legal in all the states.
  - For the vacuum world, every state has the same legal actions, so both methods give the same result.

4. Search in Partially Observable Environments

Searching With No Observation

- **Transition model**:
  - For deterministic actions, the new belief state has one result state for each of the current possible states:

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

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

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

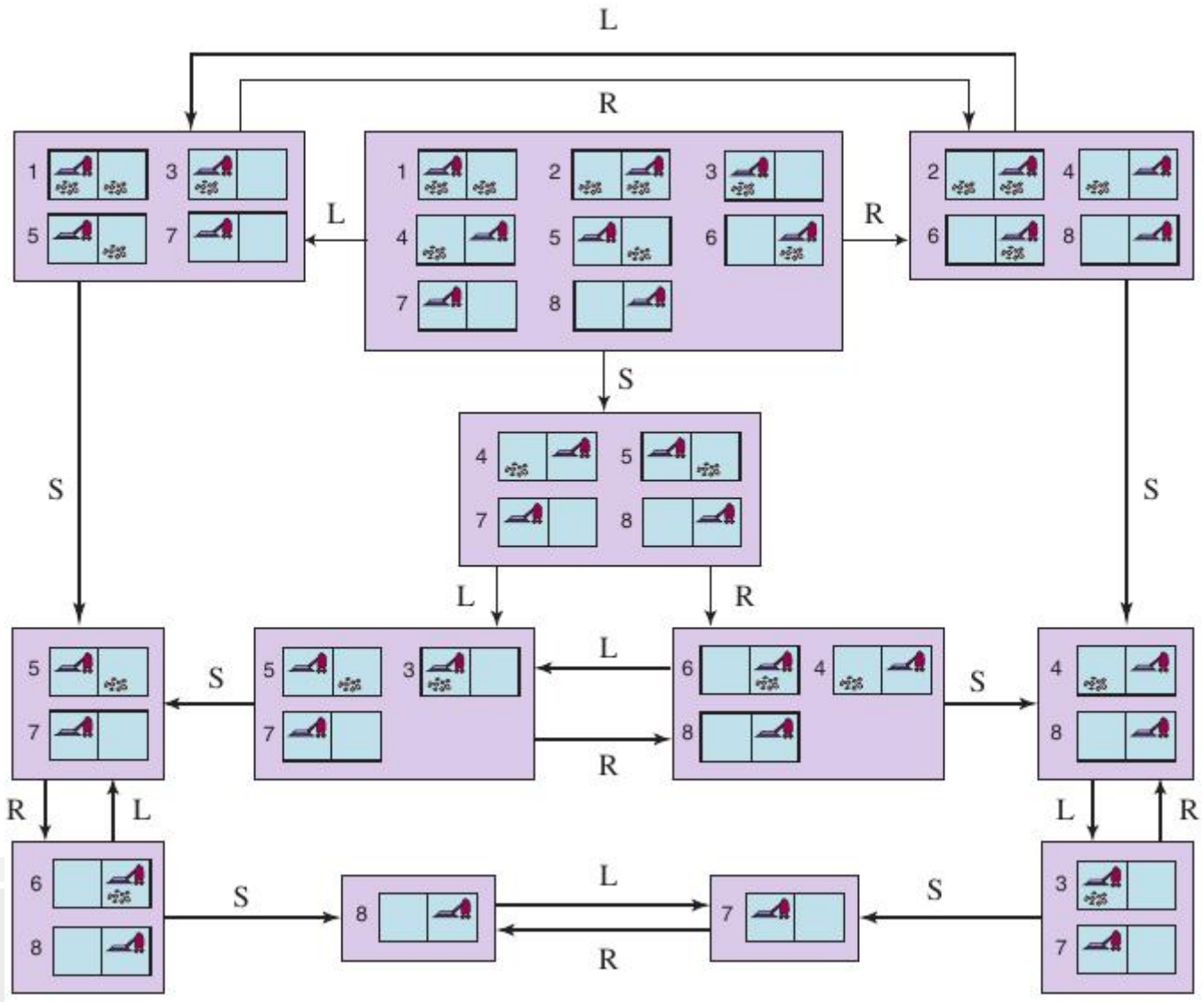
4. Search in Partially Observable Environments

Searching With No Observation

- **Goal test**:
  - The agent *possibly* achieves the *goal* if any state s in the belief state satisfies the goal test of the underlying problem, I S-GOAL$_P(s)$.
  - The agent *necessarily* achieves the goal if every state satisfies IS-GOAL$_P(s)$. **We aim to necessarily achieve the goal**.
- **Action cost**:
  - This is also tricky. If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of 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.

4. Search in Partially Observable Environments

Searching With No Observation



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

4. Search in Partially Observable Environments

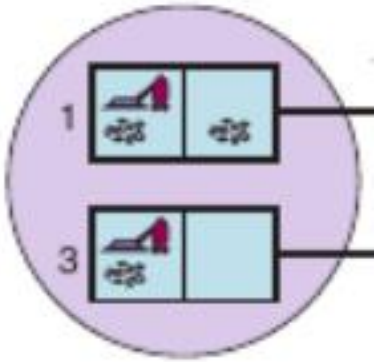
Searching in Partially Observable Environments

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

4. Search in Partially Observable Environments

Searching in Partially Observable Environments

- Consider a local-sensing vacuum world, in which the agent has



  - A position sensor that yields the percept *L* in the left square, and *R* in the right square
  - A dirt sensor that yields *Dirty* when the current square is dirty and *Clean* when it is clean.
  - Thus, the PERCEPT in state 1 is [*L, Dirty*].

- With partial observability, it will usually be the case that several states produce the same percept;
  - State 3 will also produce [*L, Dirty*].
  - Hence, given this initial percept, the initial belief state will be {1, 3}.

4. Search in Partially Observable Environments

Searching in Partially Observable Environments

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

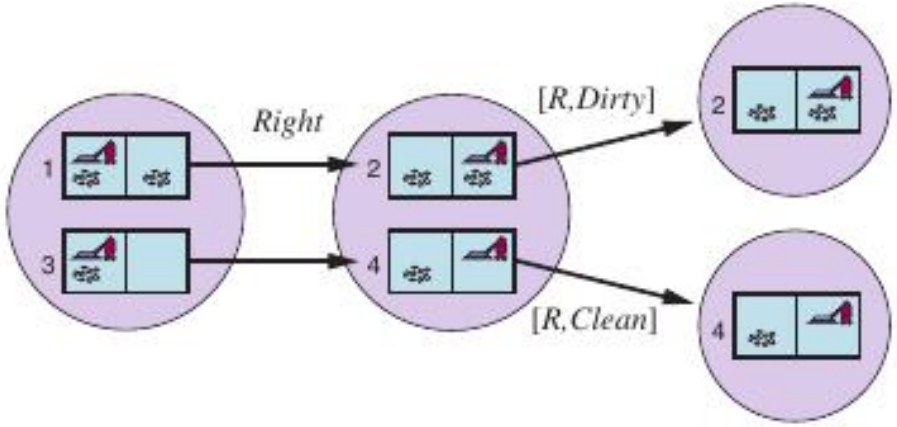
$$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}.$$

4. Search in Partially Observable Environments

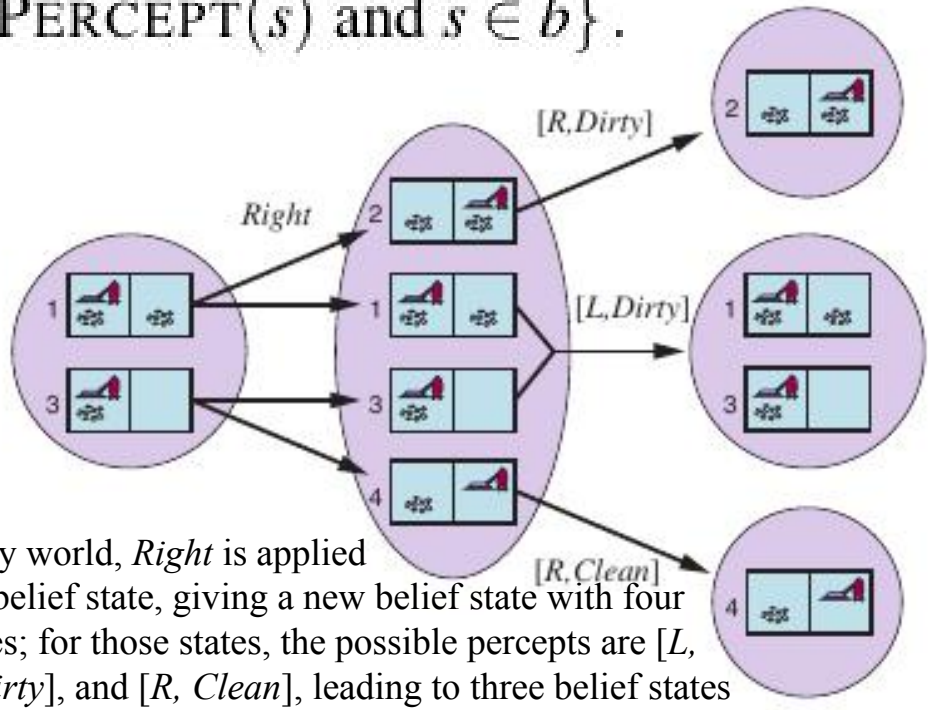
Searching in Partially Observable Environments

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

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



In the deterministic world, *Right* is applied in the initial belief state, resulting in a new predicted belief state with two possible physical states; for those states, the possible percepts are [*R, Dirty*] and [*R, Clean*], leading to two belief states, each of which is a singleton.



In the slippery world, *Right* is applied in the initial belief state, giving a new belief state with four physical states; for those states, the possible percepts are [*L, Dirty*], [*R, Dirty*], and [*R, Clean*], leading to three belief states as shown.

4. Search in Partially Observable Environments

Searching in Partially Observable Environments

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

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

4. Search in Partially Observable Environments

Solving Partially Observable Problems

- The RESULTS function for a nondeterministic belief-state problem from an underlying physical problem, given the PERCEPT function:

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

- With this formulation, the AND–OR search algorithm can be applied directly to derive a solution.

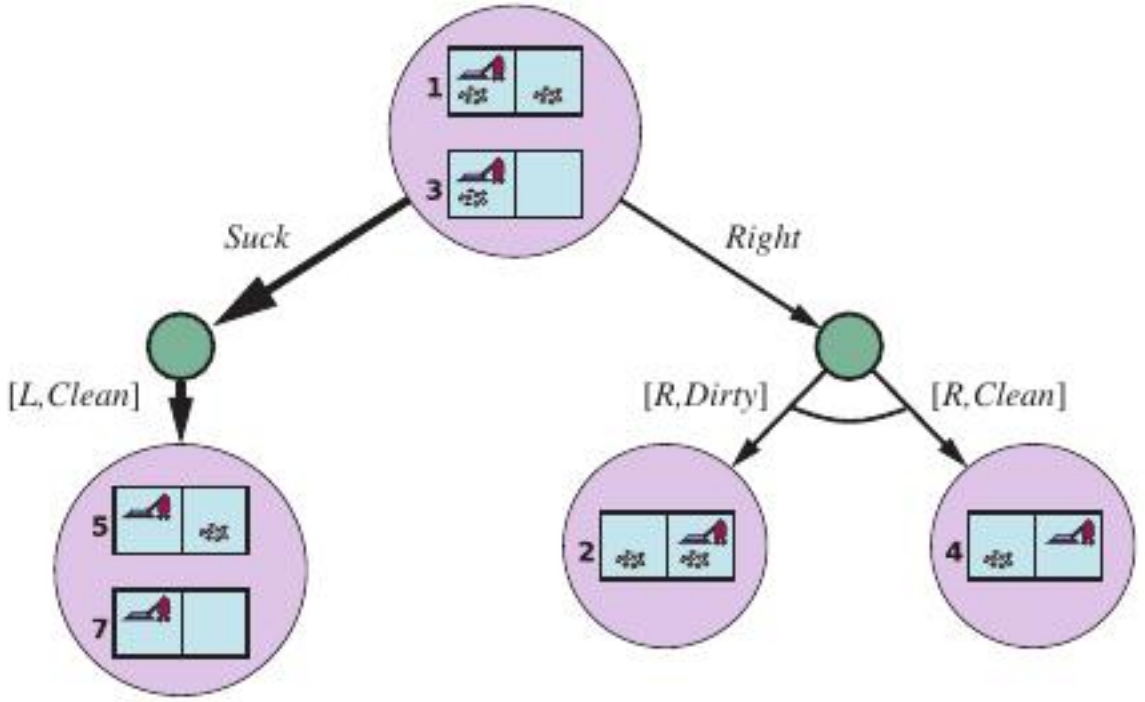
4. Search in Partially Observable Environments

Solving Partially Observable Problems

- The figure shows part of the search tree for the local-sensing vacuum world, assuming an initial percept [*L, Dirty*]. The solution is the conditional plan:

  [*Suck, Right*, **if** *Rstate* = {6} **then** *Suck* **else** [ ]]

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



4. Search in Partially Observable Environments

An Agent for Partially Observable Environments

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

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

  - In partially observable environments—which include the vast majority of real-world environments—maintaining one's belief state is a core function of any intelligent system. This function goes under various names, including **monitoring**, **filtering**, and **state estimation**. Equation (4.6) is called a recursive state estimator because it computes the new belief state from the previous one rather than by examining the entire percept sequence.

4. Search in Partially Observable Environments

An Agent for Partially Observable Environments

- Consider a kindergarten vacuum world wherein agents sense only the state of their current square, and *any square may become dirty at any time unless the agent is actively cleaning it at that moment.*

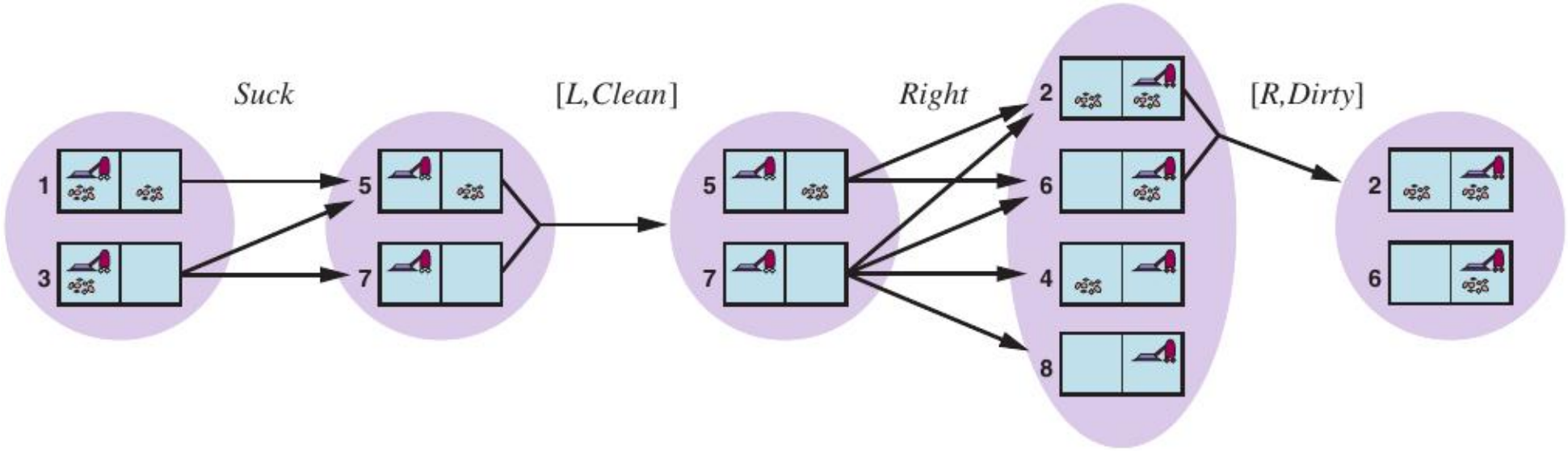


Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

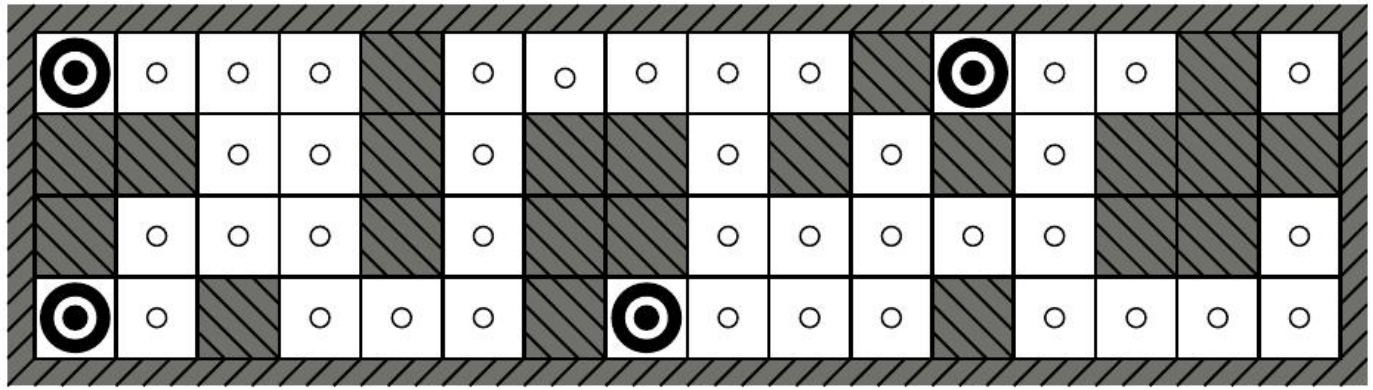
4. Search in Partially Observable Environments

An Agent for Partially Observable Environments

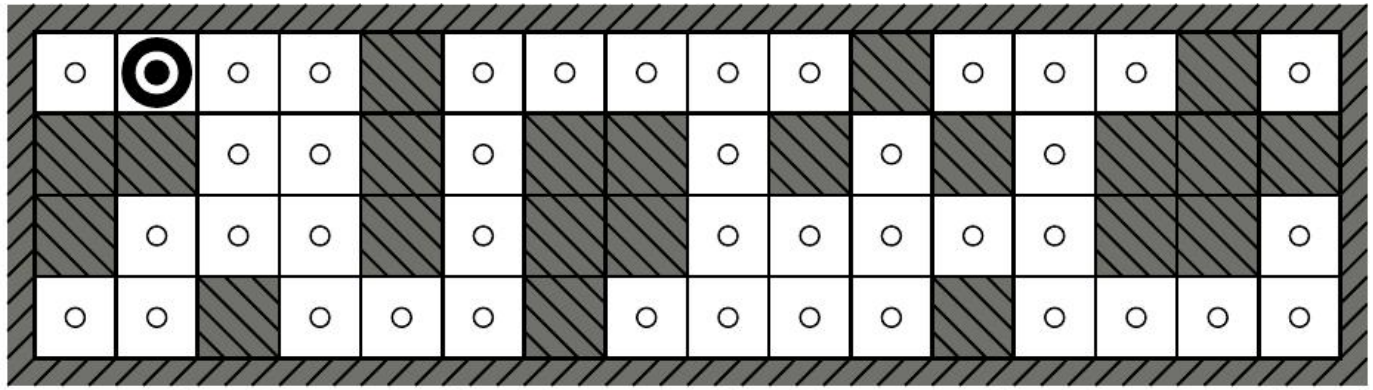
- **An example in a discrete environment with deterministic sensors and nondeterministic actions**
  - The example concerns a robot with a particular state estimation task called **localization**: working out where it is, given a map of the world and a sequence of percepts and actions.
- Our robot is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

4. Search in Partially Observable Environments

An Agent for Partially Observable Environments



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

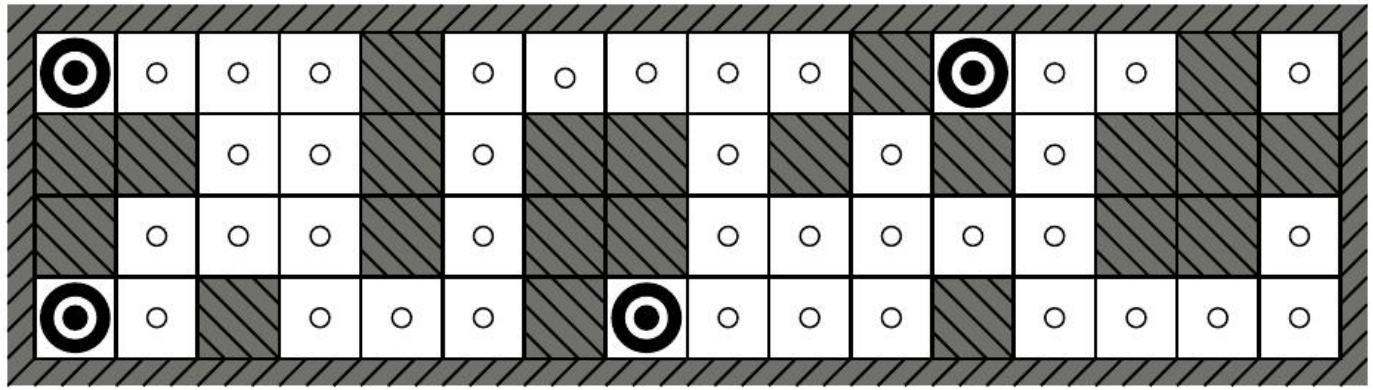


(b) Possible locations of robot after $E_1 = 1011, E_2 = 1010$

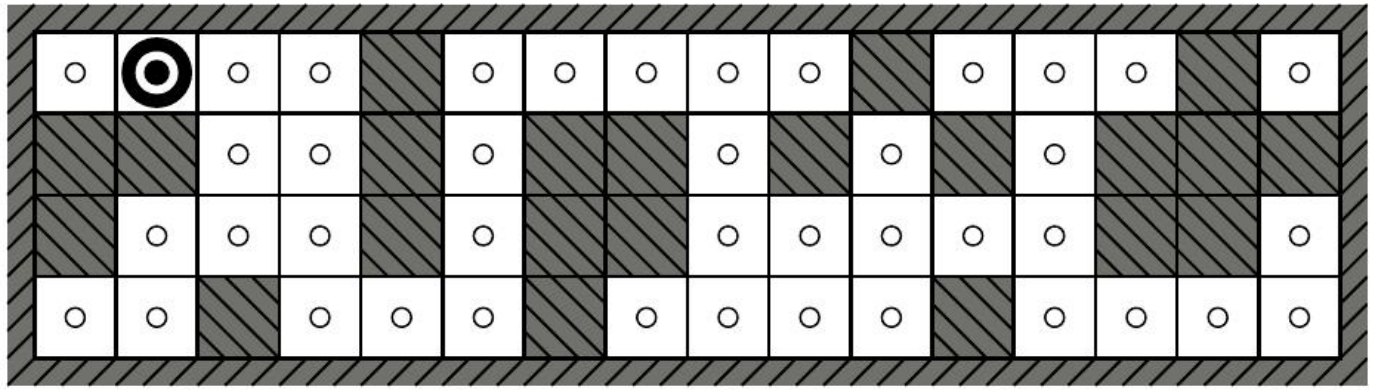
- We assume that the sensors give perfectly correct data, and that the robot has a correct map of the environment.
- But unfortunately, the robot's navigational system is broken, so when it executes a *Right* action, it moves randomly to one of the adjacent squares.
- The robot's task is to determine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

4. Search in Partially Observable Environments

An Agent for Partially Observable Environments



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



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

- Initial state
  - The robot has just been switched on, and it does not know where it is—its initial belief state b_o consists of the set of all locations.
  - The robot then receives the percept 1011 and does an update using the equation b_o = UPDATE(1011), yielding the 4 locations shown in Figure 4.18(a).
- Next the robot executes a *Right* action, but the result is nondeterministic.
  - The new belief state, b_a = PREDICT(b_o, *Right*), contains all the locations that are one step away from the locations in b_o.
  - When the second percept, 1010, arrives, the robot does UPDATE (b_a, 1010) and finds that the belief state has collapsed down to the single location shown in Figure 4.18(b). That's the only location that could be the result of UPDATE(PREDICT(UPDATE(*b*, 1011), *Right*), 1010) .