Artificial Intelligence

Search in Complex Environments

Dr. Bilgin Avenoğlu

Complex Environments

- We have already addressed problems in fully observable, deterministic, static, known environments where the solution is a sequence of actions.
- We will look at discrete states.
- We will relax the assumptions of determinism and observability.

Local Search and Optimization Problems

- Instead of finding paths through the search space (Arad to Bucharest)
  - We care only the final state
- 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.
  - They are not systematic - they might never explore a portion of the search space where a solution actually resides.
  - they use very little memory
  - they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

Hill-climbing

- Keeps track of one current state and on each iteration moves to the neighboring state with highest value
  - 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.

Steepest ascent

```
Function hill ()
{
     queue = [];     // initialize an empty queue
     state = root_node;   // initialize the start state
     while (true)
     {
          if is_goal (state)
               then return SUCCESS
          else
          {
               sort (successors (state));
               add_to_front_of_queue (successors (state));
          }
          if queue == []
               then report FAILURE;
          state = queue [0]; // state = first item in queue
          remove_first_item_from (queue);
     }
 }
```

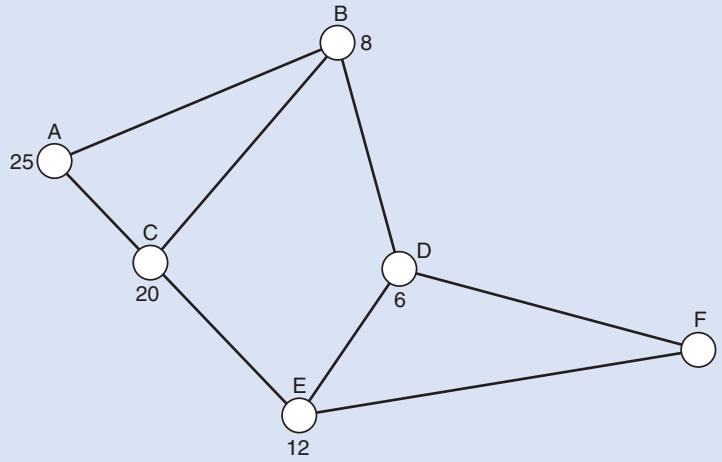


Figure 4.8
The map of five cities where the straight-line distance from each city to the goal city (F) is shown

Table 4.4 Analysis of hill climbing

Step	State	Queue	Notes
1	A	(empty)	The queue starts out empty, and the initial state is the root node, which is A.
2	A	B,C	The successors of A are sorted and placed on the queue. B is placed before C on the queue because it is closer to the goal state, F.
3	B	C	
4	B	D,C,C	
5	D	C,C	
6	D	F,E,C,C	F is placed first on the queue because it is closest to the goal. In fact, it is the goal, as will be discovered in the next step.
7	F	E,C,C	SUCCESS: Path is reported as A,B,D,F.

The 8-queens problem

- The heuristic cost function h is the number of pairs of queens that are attacking each other;
- This will be zero only for solutions.
- It counts as an attack if two pieces are in the same line, even if there is an intervening piece between them.

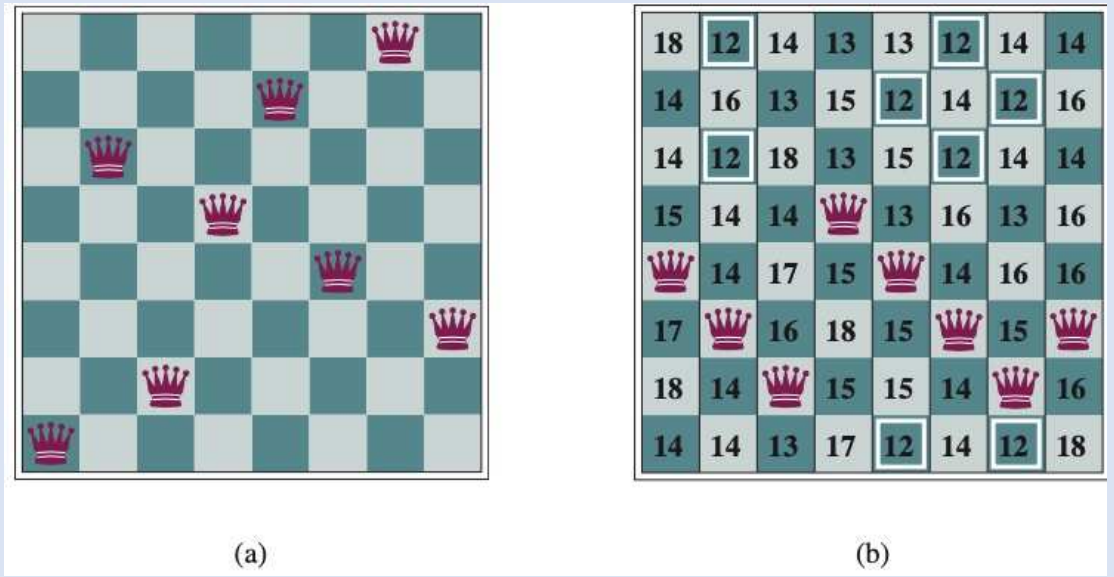
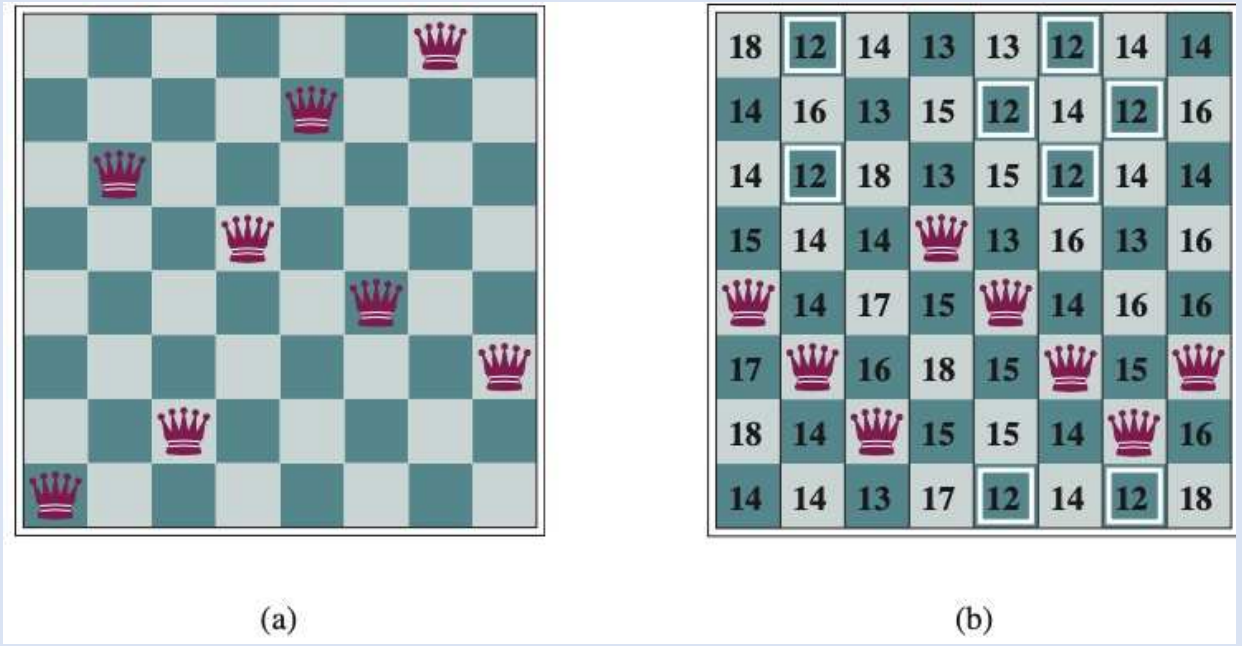
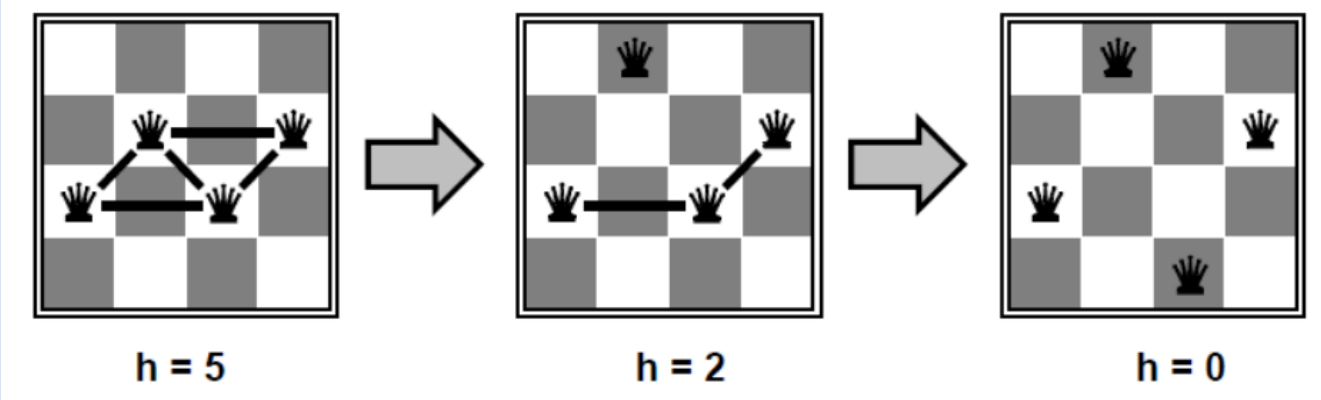


Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (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.

Hill climbing

- Sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.
- It can make rapid progress toward a solution because it is usually quite easy to improve a bad state.
- From (b), it takes just five steps to reach the state in (a), which has h = 1 and is very nearly a solution.

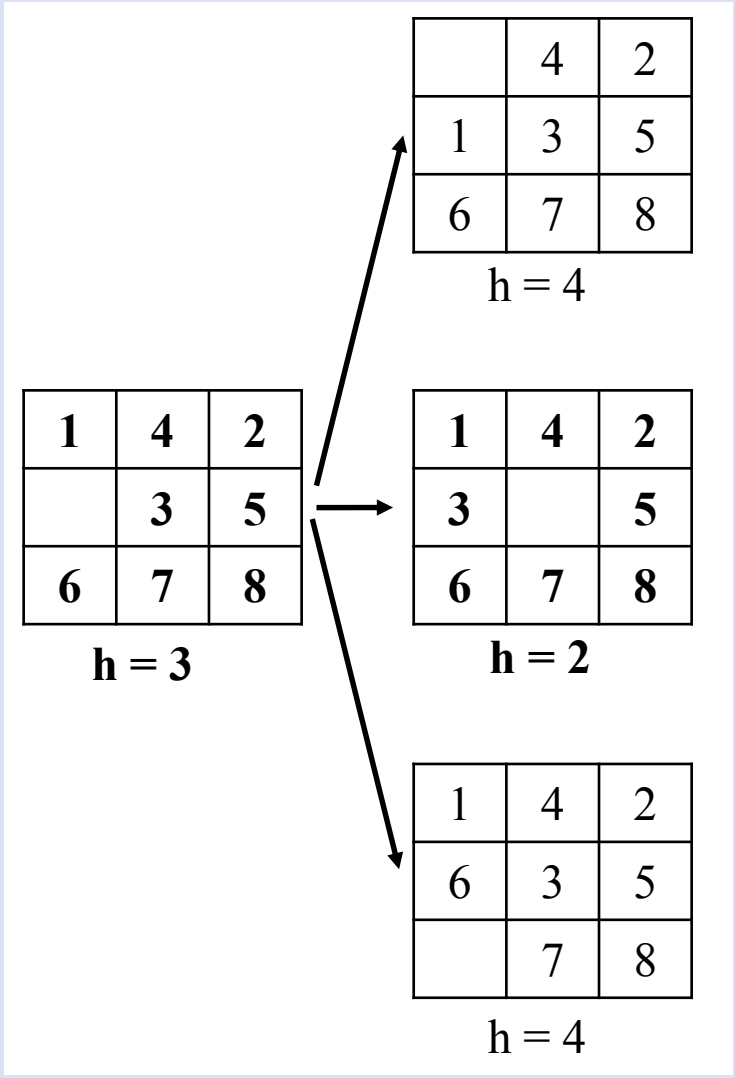




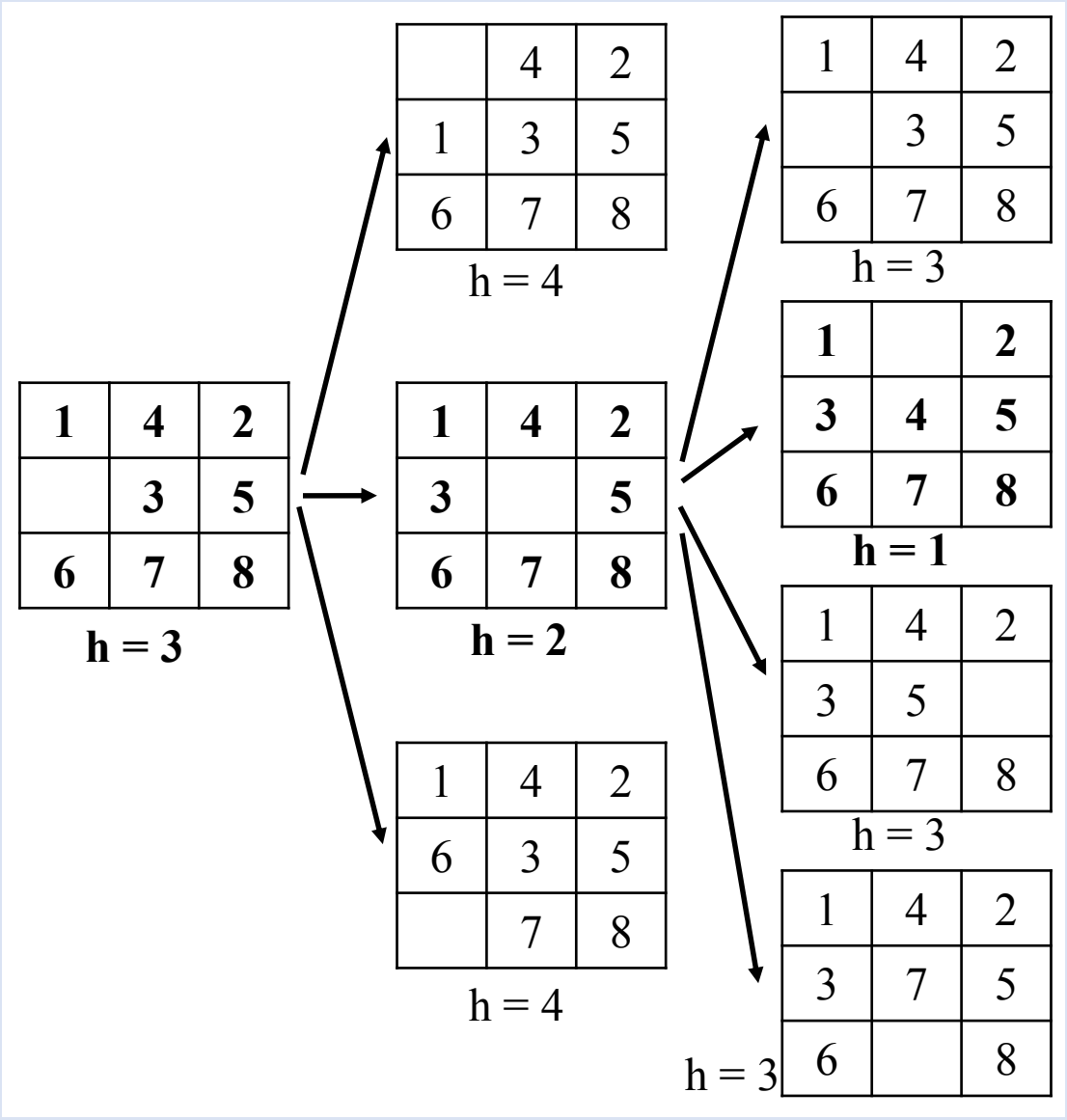
Example

1	**4**	**2**
	3	**5**
6	**7**	**8**

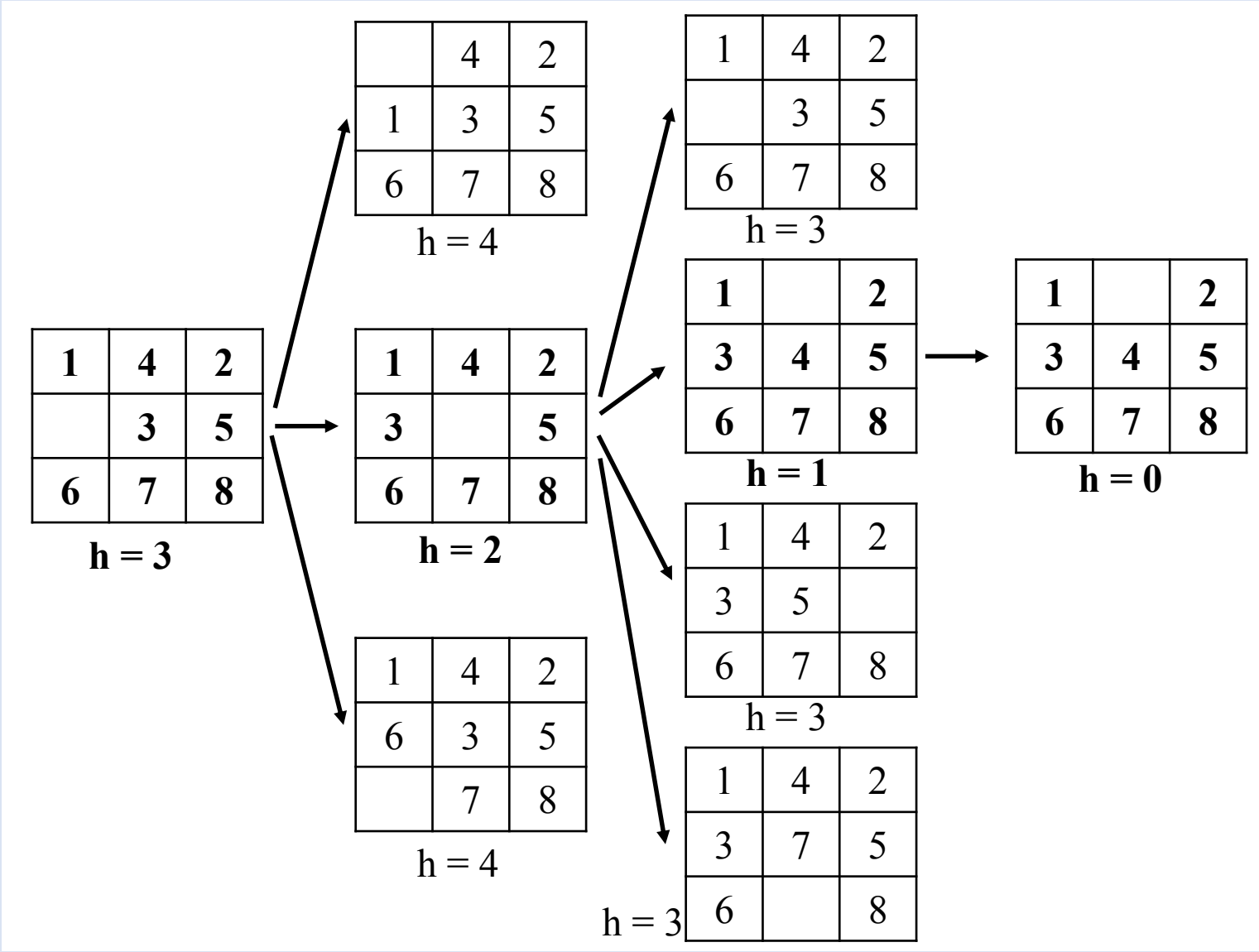
h = 3



Example

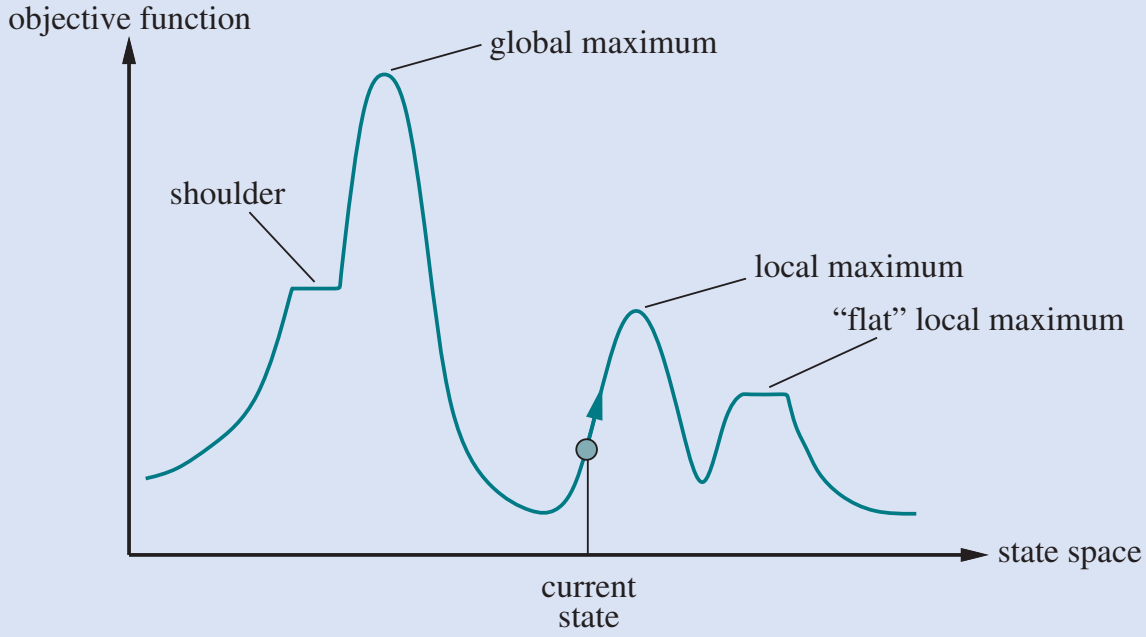

1 4 2
3 5
6 7 8
h = 3
4 2
1 3 5
6 7 8
h = 4
1 4 2
3 5
6 7 8
h = 2
1 4 2
6 3 5
7 8
h = 4
1 4 2
3 5
6 7 8
h = 3
1 2
3 4 5
6 7 8
h = 1
1 4 2
3 5
6 7 8
h = 3
1 4 2
3 7 5
6 8
h = 3

Example


1 4 2
3 5
6 7 8
h = 3
4 2
1 3 5
6 7 8
h = 4
1 4 2
3 5
6 7 8
h = 2
1 4 2
6 3 5
7 8
h = 4
1 4 2
3 5
6 7 8
h = 3
1 2
3 4 5
6 7 8
h = 1
1 2
3 4 5
6 7 8
h = 0
1 4 2
3 5
6 7 8
h = 3
1 4 2
3 7 5
6 8
h = 3

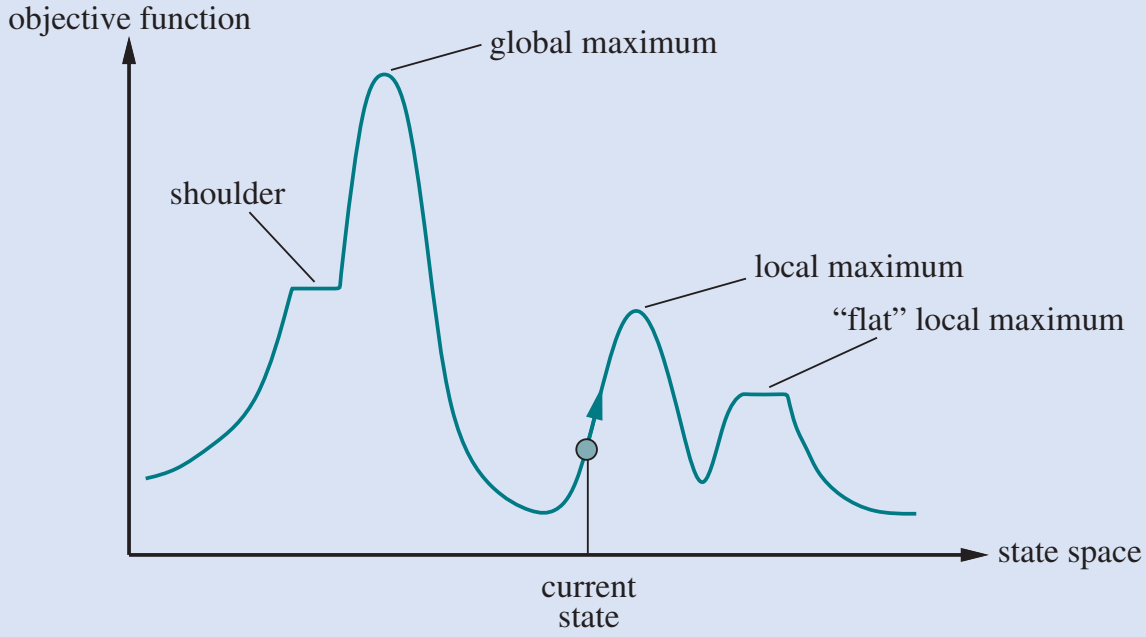
Hill climbing – Local Maxima Problem

- A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
- Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.
- Figure is a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.



Hill climbing – Plateaus Problem

- 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.



Hill climbing – Ridges Problem

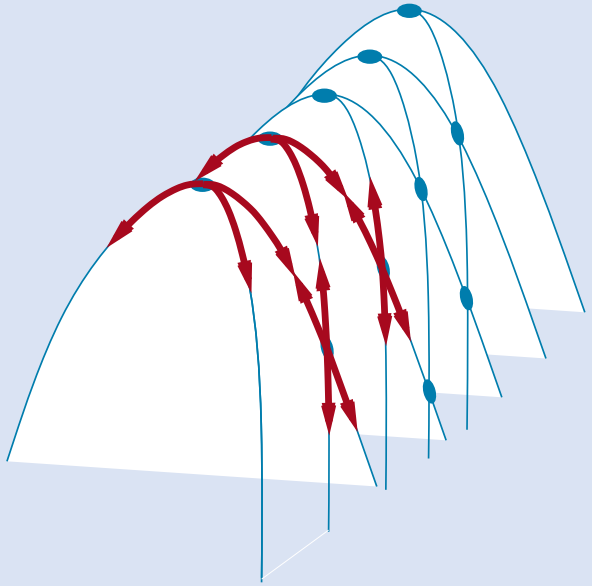
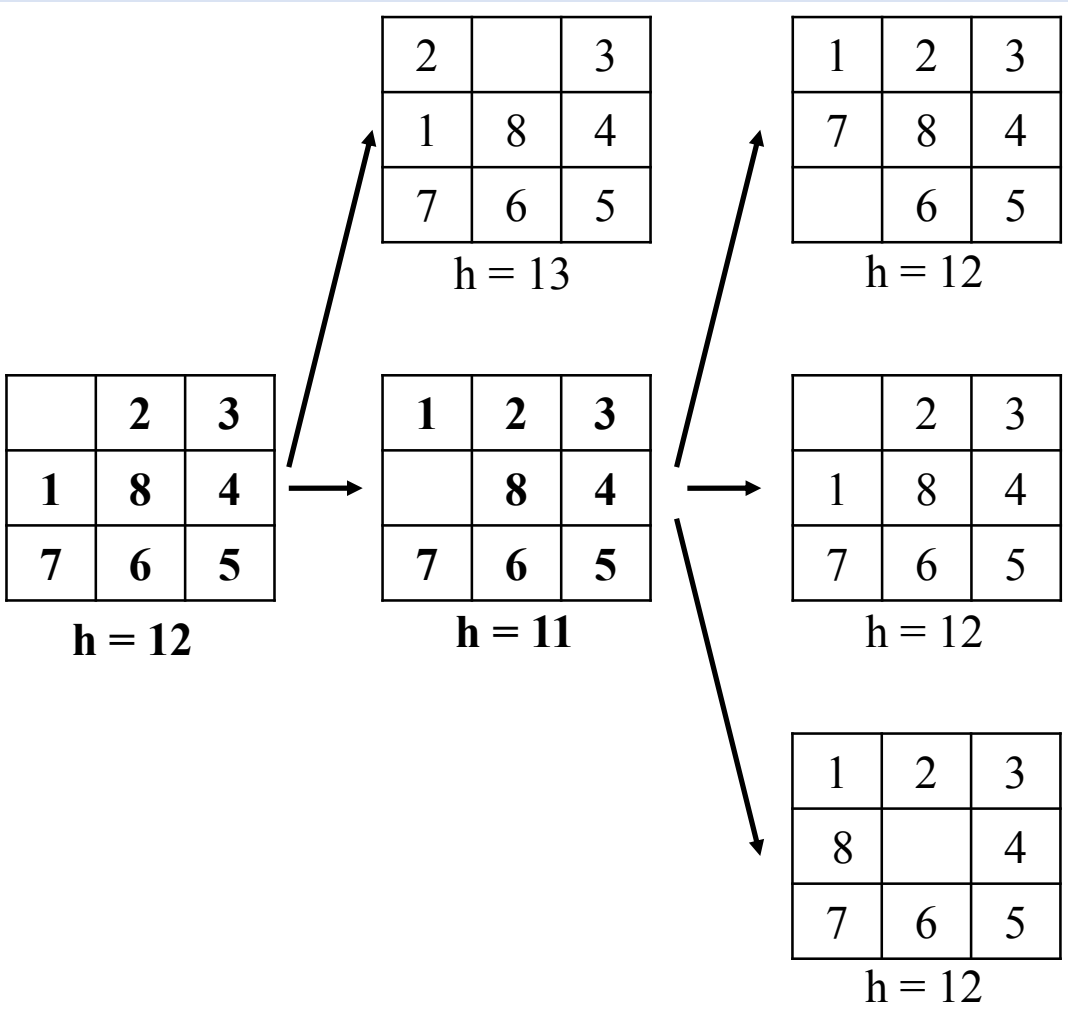


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill. Topologies like this are common in low-dimensional state spaces, such as points in a two-dimensional plane. But in state spaces with hundreds or thousands of dimensions, this intuitive picture does not hold, and there are usually at least a few dimensions that make it possible to escape from ridges and plateaus.

Hill climbing

- Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances.
- On the other hand, 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.

Example

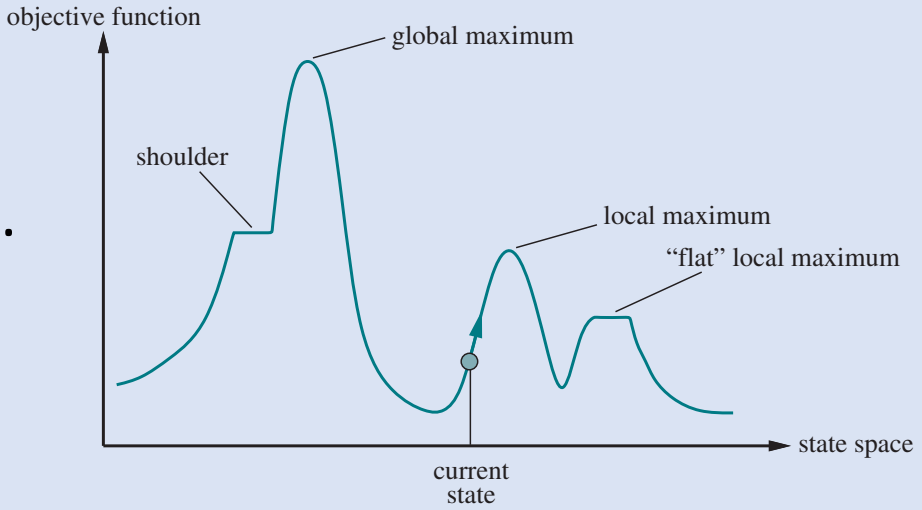


Heuristic function is Manhattan Distance

We are stuck with a local maximum.

Hill climbing - Solutions

- Keep going when we reach a plateau - to allow a sideways move in the hope that the plateau is really a shoulder.
- If it is actually a flat local maximum, it will wander on the plateau forever.
- We can limit the number of consecutive sideways moves, stopping after, say, 100 consecutive sideways moves.
- This raises the percentage of problem instances solved 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 Properties

- Hill Climbing is **NOT complete**.
- Hill Climbing is **NOT optimal**.

- **Why use local search?**
  - Low memory requirements – usually constant
  - Effective – Can often find good solutions in extremely large state spaces
  - Randomized variants of hill climbing can solve many of the drawbacks in practice.

Hill climbing variants

- **Stochastic hill climbing** chooses at random from among the uphill moves;
- The **probability of selection** can vary with the ***steepness of the uphill move***.
- **Stochastic hill climbing** usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.

- **Stochastic hill climbing** is NOT complete, but it may be less likely to get stuck.

Hill climbing variants

- 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.

First-choice hill climbing is also NOT complete,

Hill climbing variants

- Random-restart hill climbing, which adopts the adage, “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.

- Random-Restart Hill Climbing is **complete** if ***infinite (or sufficiently many tries) are allowed***.
- If each hill-climbing search has a ***probability p*** of success, then the ***expected number of restarts required*** is **1/p**.
- For 8-queens instances with no sideways moves allowed, $p \approx 0.14$, so we need roughly 7 iterations to find a goal (6 failures and 1 success).
  - For 8-queens, then, random-restart hill climbing is very effective indeed.
  - Even for three million queens, the approach can find solutions in under a minute.
- The ***success of hill climbing*** depends very much on the ***shape of the state-space landscape***:
  - If there are few local maxima and plateau, ***random-restart hill climbing*** will find a good solution very quickly.
  - On the other hand, many real problems have ***many local maxima*** to get stuck on.
  - NP-hard problems typically have an ***exponential number of local maxima*** to get stuck on.

Simulated annealing

- A hill-climbing algorithm never makes “downhill” moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum.
- 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.
- It seems reasonable to try to combine hill climbing with a random walk in a way that yields both efficiency and completeness.

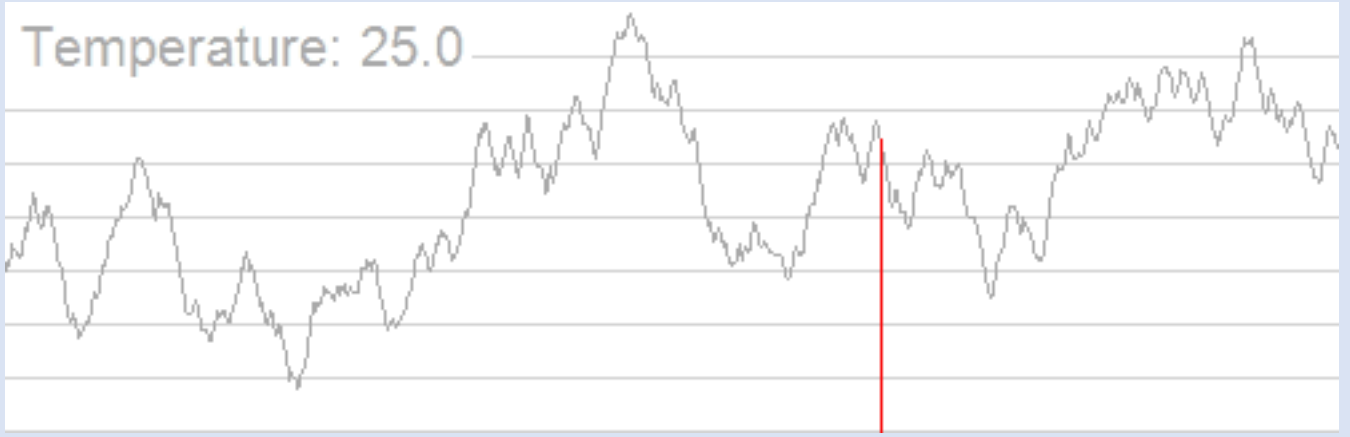
Simulated annealing

- 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 amount ΔE by which the evaluation is worsened.
- The probability also decreases as the "temperature" T goes down:
  - "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.
- If the schedule lowers T to 0 slowly enough, then a property of the Boltzmann distribution, $e^{\Delta E/T}$,
  - is that all the probability is concentrated on the global maxima,
  - which the algorithm will find with probability approaching 1.

Simulated annealing

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

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



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.
- In a local beam search, useful information is passed among the parallel search threads.
- Quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

Stochastic beam search

- Local beam search can suffer from a lack of diversity among the k states
  - they can become clustered in a small region of the state space, making the search little more than a k-times-slower version of hill climbing.
- A variant called stochastic beam search, analogous to stochastic hill climbing, helps alleviate this problem.
- Instead of choosing the top k successors, stochastic beam search chooses successors with probability proportional to the successor's value, thus increasing diversity.

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

- (a) The c-*th* digit represents the row number of the queen in column c.
- 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.

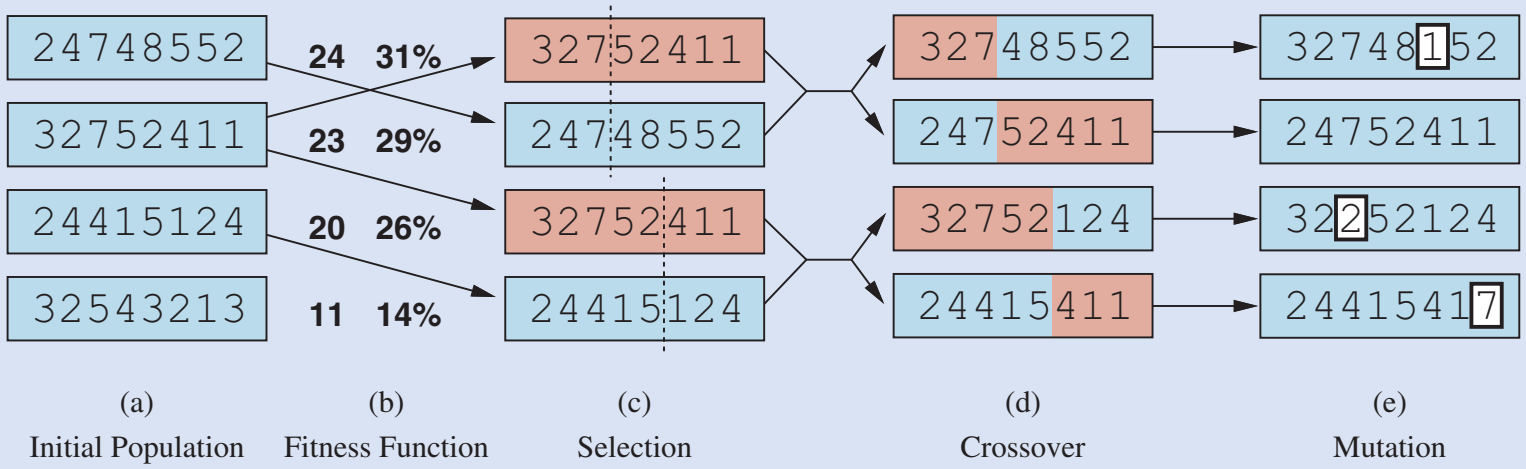


Figure 4.6 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

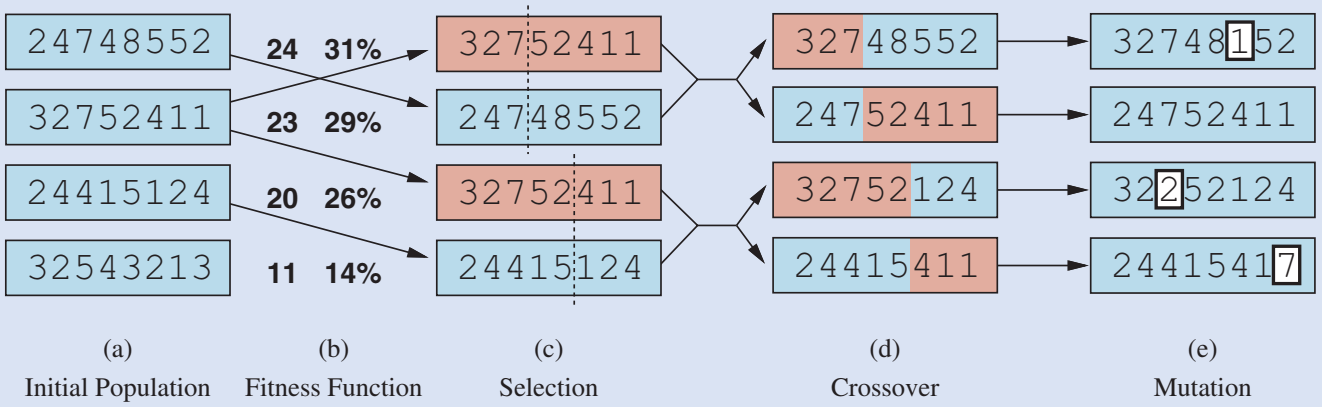


Figure 4.6 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

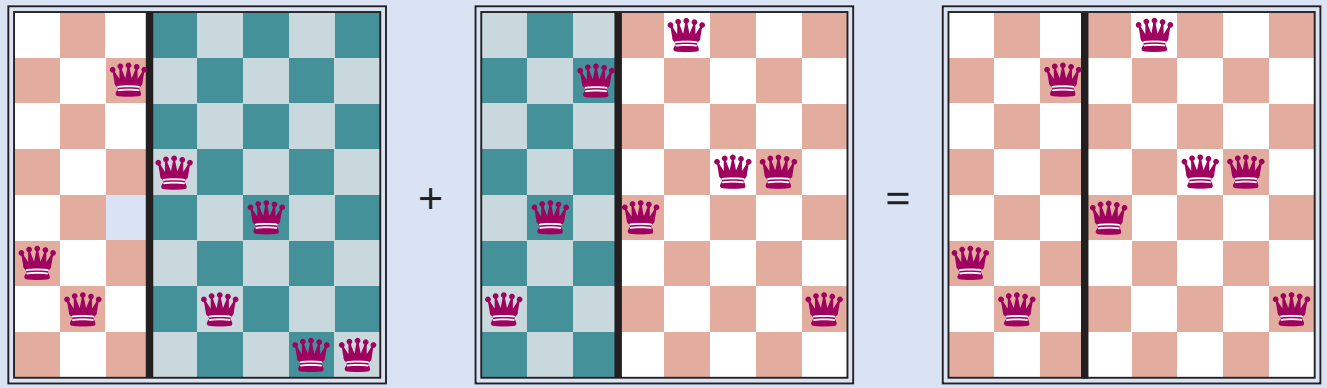


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The green columns are lost in the crossover step and the red columns are retained. (To interpret the numbers in Figure 4.6: row 1 is the bottom row, and 8 is the top row.)

Evolutionary algorithms

- The population is diverse early on in the process, so crossover frequently takes large steps in the state space early in the search process (as in simulated annealing).
- After many generations of selection towards higher fitness, the population becomes less diverse, and smaller steps are typical.

```
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))
```

Figure 4.8 A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

Search with Nondeterministic Actions

- When the environment is partially observable,
  - the agent doesn't know for sure what state it is in;
- When the environment is nondeterministic,
  - the agent doesn't know what state it transitions to after taking an action.
- That means that rather than thinking
  - "I'm in *state s1* and if I *do action a*, I'll end up in state s2"
  - Think like: "I'm either in *state s1 or s3*, and if I do *action a*, I'll end up in *state s2, s4 or s5*"
- We call a set of physical states that the agent believes are possible a belief state.

The erratic vacuum world

- Let's 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.

The erratic vacuum world

- A conditional plan can contain
  - if–then–else steps;
- 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.

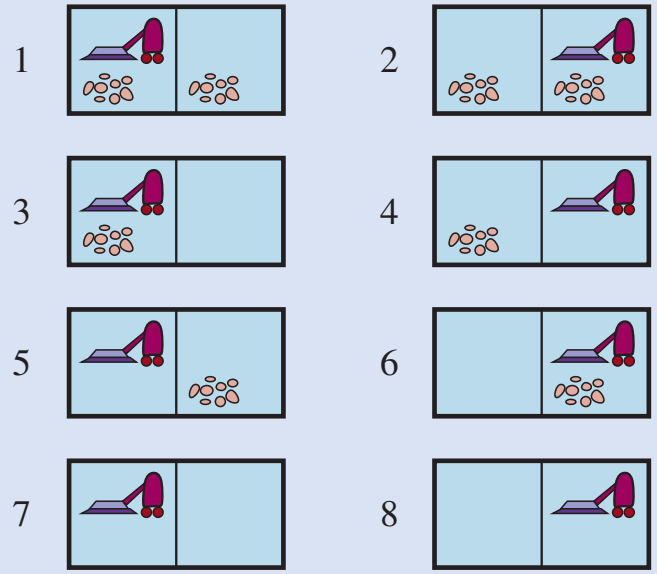


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

$$\textsc{Results}(1, Suck) = \{5, 7\}$$

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

AND-OR search trees

- 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.

AND-OR search trees

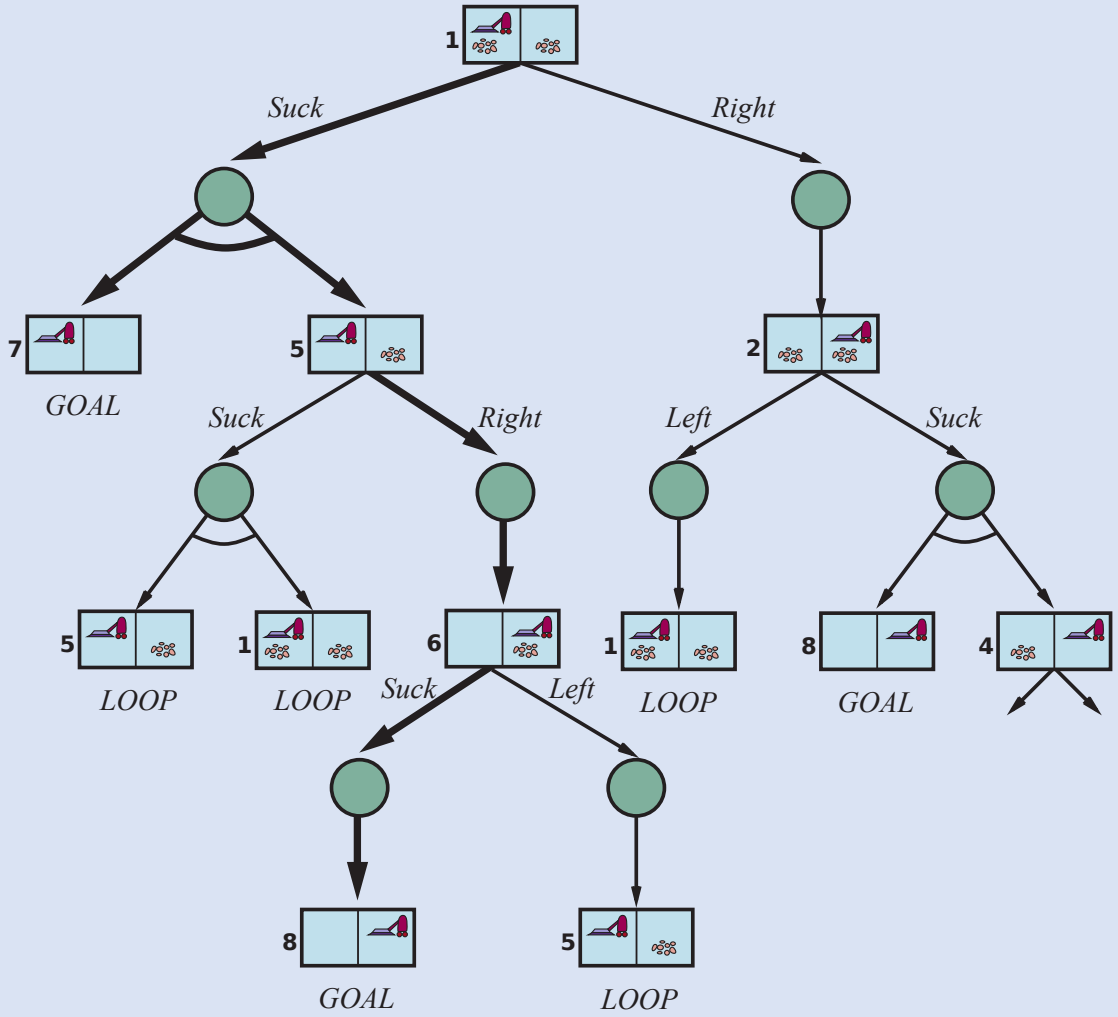


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

AND-OR search trees

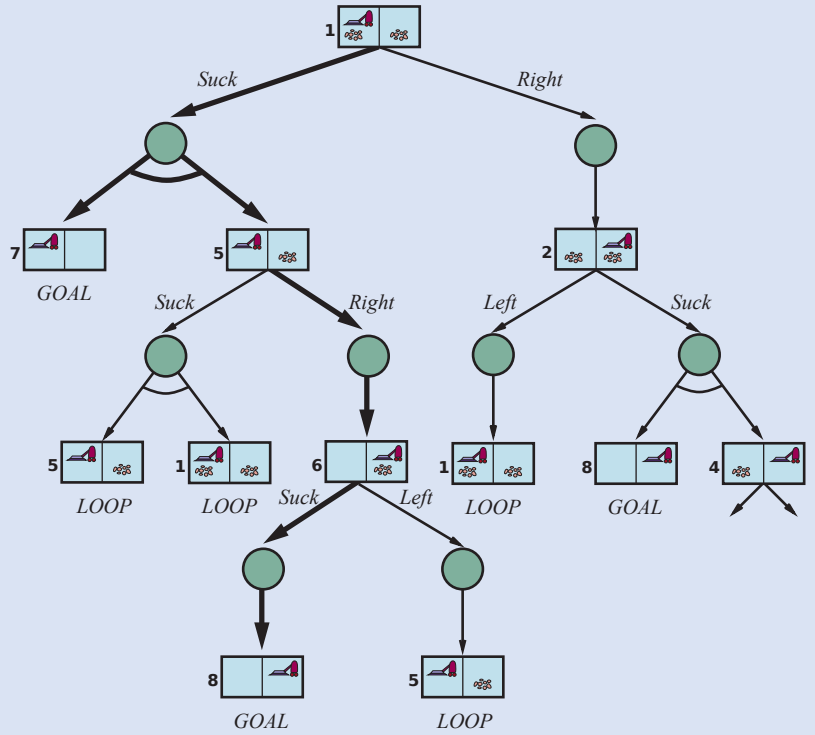


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

function AND-OR-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**
    $plan \leftarrow$ AND-SEARCH(*problem*, RESULTS(*state*, *action*), [*state*] + [*path*])
    if $plan \neq failure$ **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**
    $plan_i \leftarrow$ 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 solution is a conditional plan that considers every nondeterministic outcome and makes a plan for each one.

Slippery vacuum world

- Identical to the ordinary (non-erratic) vacuum world except that movement actions sometimes fail, leaving the agent in the same location.
- If the vacuum robot's drive mechanism works some of the time, but randomly and independently slips on other occasions, then the agent can be confident that if the action is repeated enough times, eventually it will work and the plan will succeed.

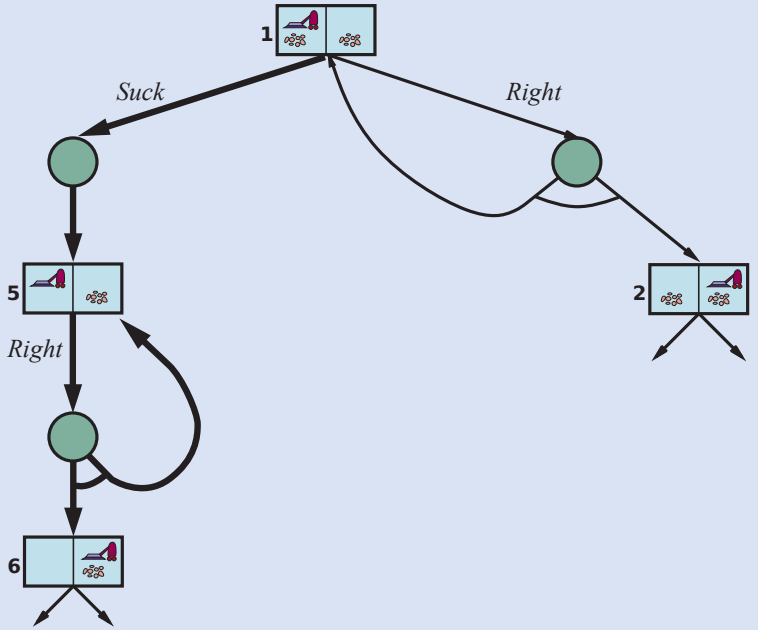


Figure 4.12 Part of the search graph for a slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

$$[Suck, \textbf{while } State = 5 \textbf{ do } Right, Suck]$$

or by adding a **label** to denote some portion of the plan and referring to that label later:

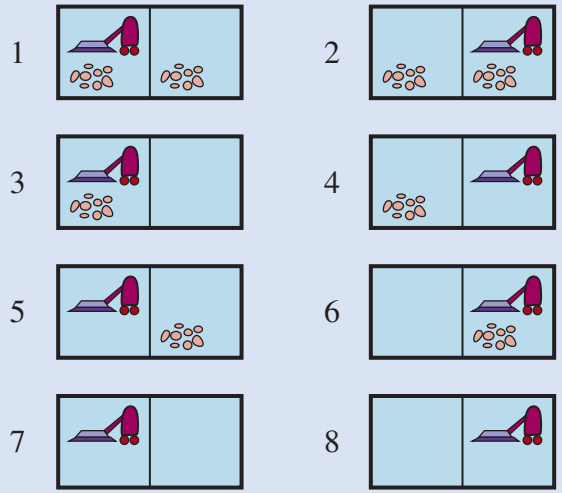
$$[Suck, L_1 : \ Right, \textbf{if } State = 5 \textbf{ then } L_1 \textbf{ else } Suck] .$$

Searching with no observation

- When the agent's percepts provide no information at all, we have what is called a sensorless problem.
- 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.
  - its initial belief state is {1,2,3,4,5,6,7,8}
  - if the agent moves Right it will be in one of the states {2,4,6,8}
  - 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

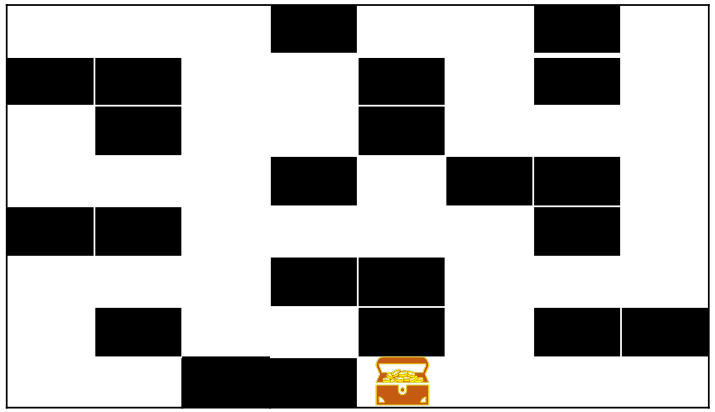
  to reach the goal state 7, no matter what the start state.



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 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.
- As in the ordinary problems of Chapter 3, the percepts received after each action are completely predictable - they're always empty! So there are no contingencies to plan for.

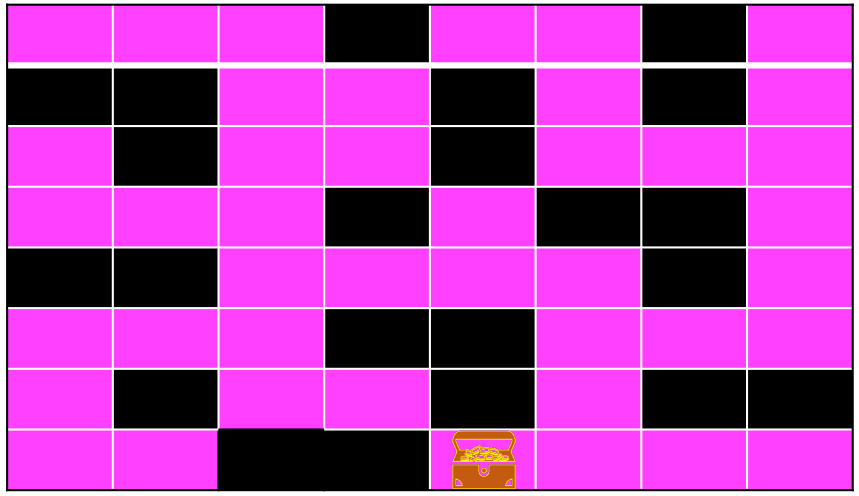
Unobservable environments



But suppose this environment is unobservable.

We know the layout of the map (it's a known environment), but we don't know which square we're starting from. Once we move, we can't tell whether we moved successfully, or ran into a wall.

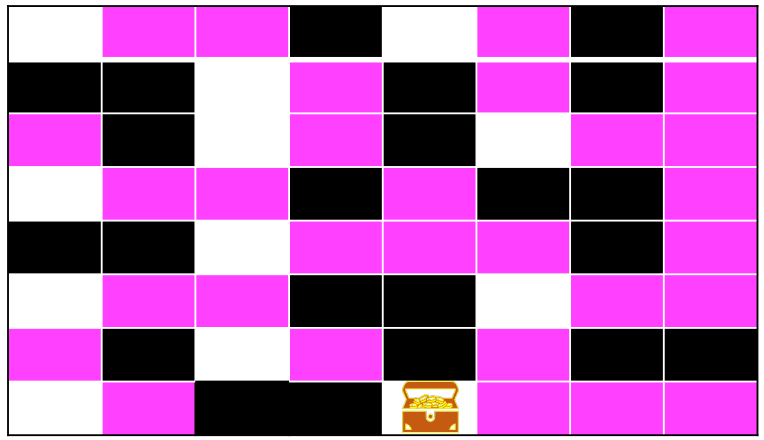
Isn't that a hopeless problem?



Let's use pink to color in all the squares where we might be.

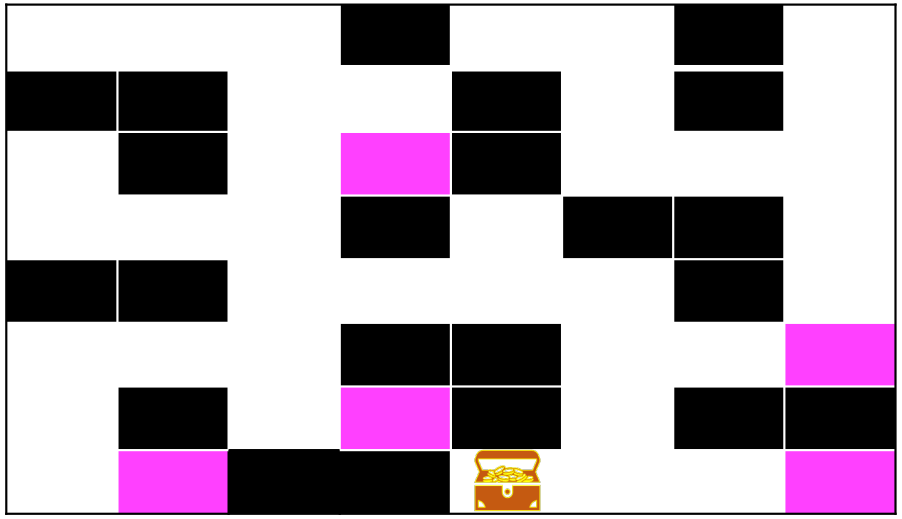
At the beginning of the search, we might be in any square, so all squares are pink.

Unobservable environments



Now take 1 step to the right.

If the environment is deterministic, then we can now guarantee that we are in one of the pink squares shown here. We know we're not in any of the white squares.



Take 4 steps right, then 4 steps down, then 4 steps right, then 4 steps down, then 2 steps right.

Now we know that we are in one of these five squares.

Searching with no observation

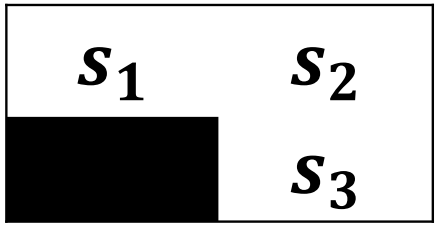
Belief state

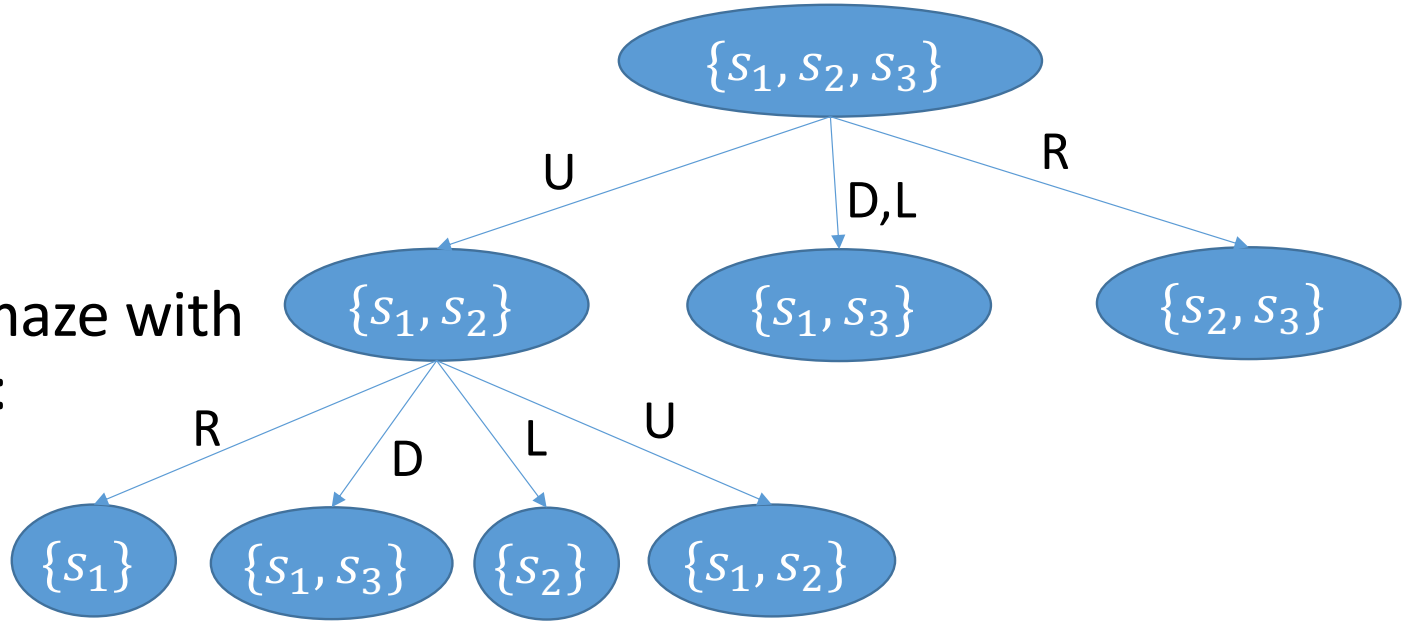
- A **belief state** is a set of physical states (by "physical state," we mean a state of the environment).
  - In a maze, a physical state might be $s_1 = (x_1, y_1)$, the agent position.
  - The belief state is a set of physical states: $b = \{s_1, s_2, s_3, \dots\}$
- If the environment is unobservable, then we can't perform search using physical states.
- Instead, we create a search tree using belief states.

Searching with no observation

Belief states

Suppose that we have a maze with only three physical states:

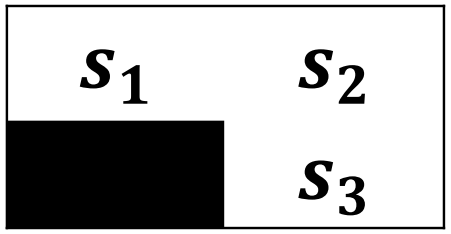




- At the beginning of search, the belief state is $b = \{s_1, s_2, s_3\}$.
- After one step to the right, the belief state is $b = \{s_2, s_3\}$

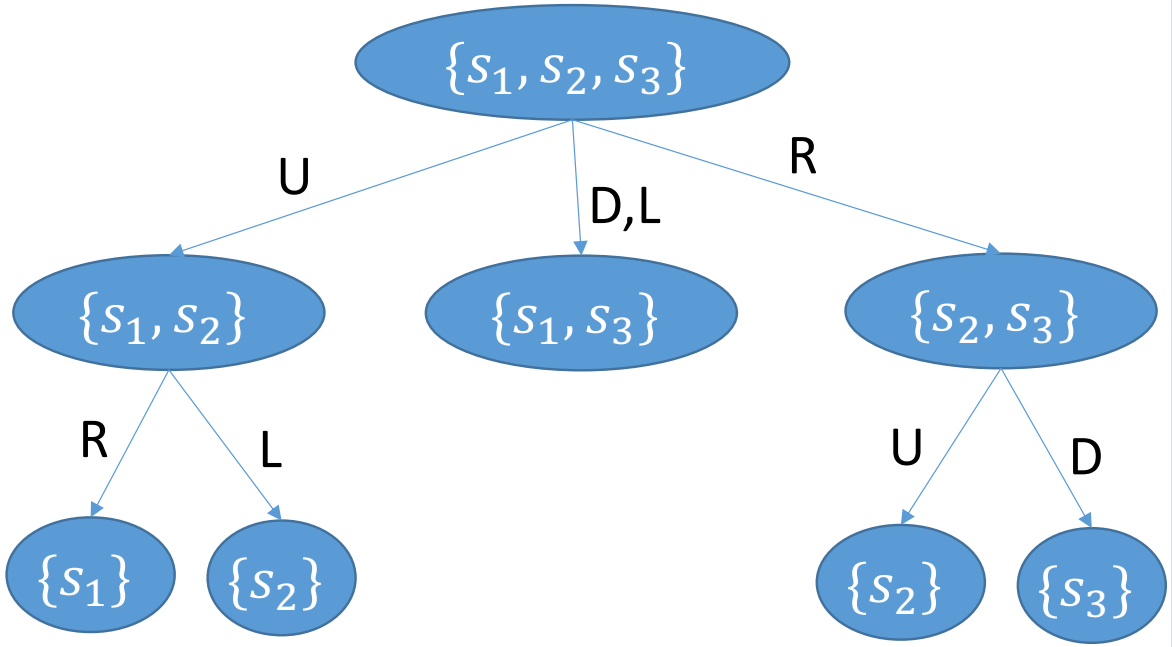
Searching with no observation

Belief states



A full breadth-first search on this maze can reach any desired physical state in just 2 steps, even if we have no idea where we started from.

(Shown here: the tree without any repeated states).



Searching with no observation

Computational Complexity

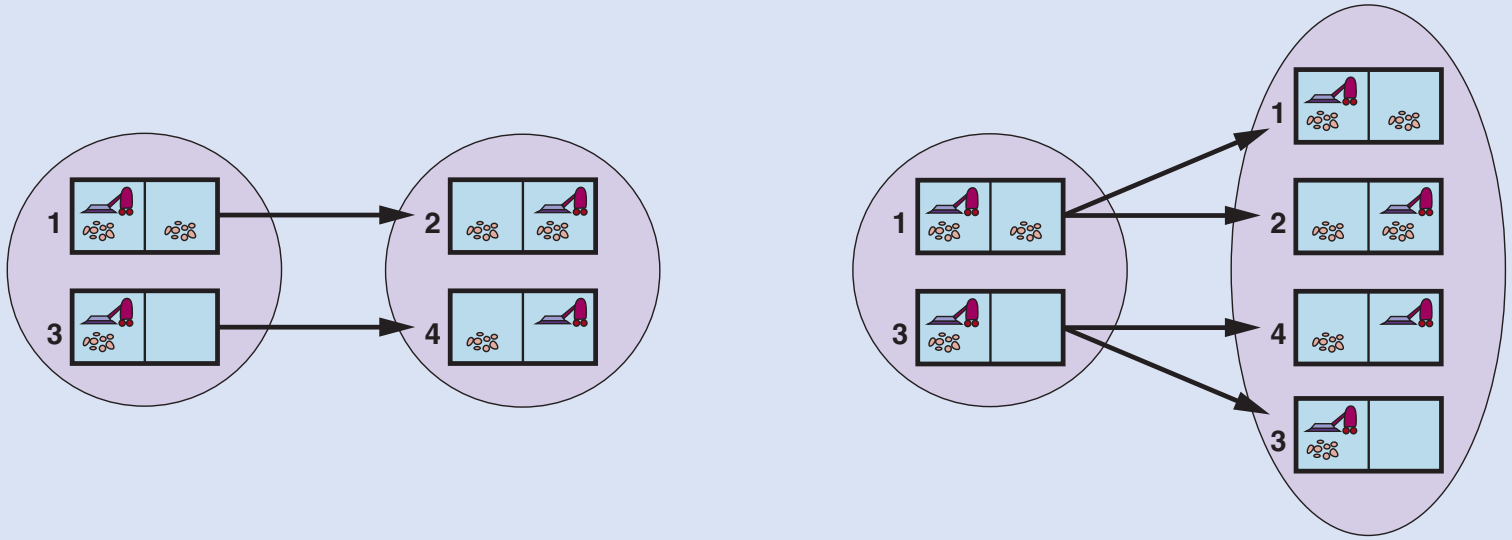
Remember that the time-complexity of BFS for an observable state space is $O\{\min(b^d, N)\}$, where b=branching factor, d=length of best path, N=# distinct states (exhaustive search). But…

- If N is the # distinct physical states, then 2^N is the number of distinct belief states. Exhaustive search is much more exhausting!!!
- The shortest path is also longer: call it d'.

Total:

$$O\{\min(b^{d'}, 2^N)\}$$

Searching with no observation



(a) (b)

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

Searching with no observation

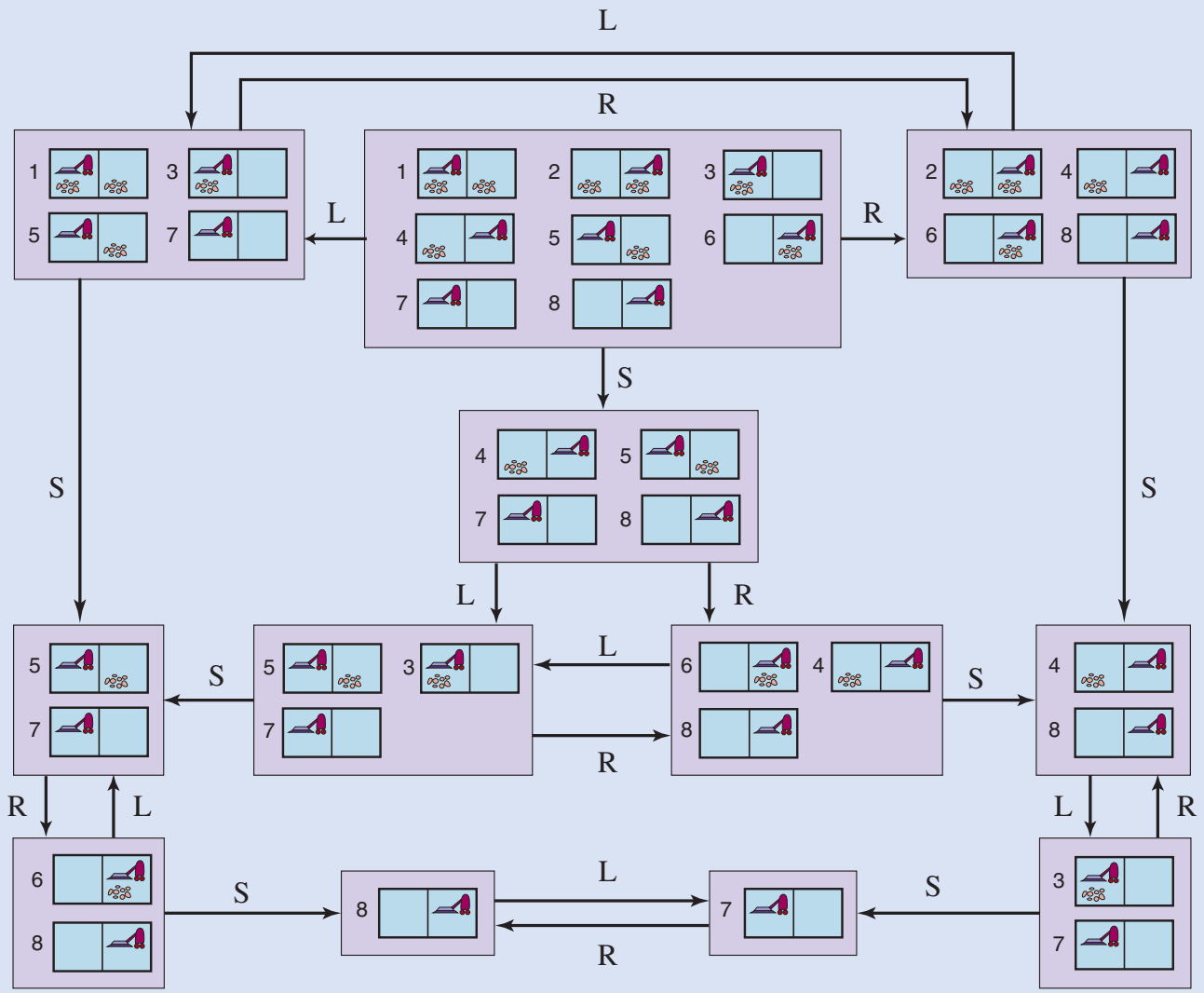


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each rectangular box corresponds to a single belief state. At any given point, the agent has a belief state but does not know which physical state it is in. The initial belief state (complete ignorance)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

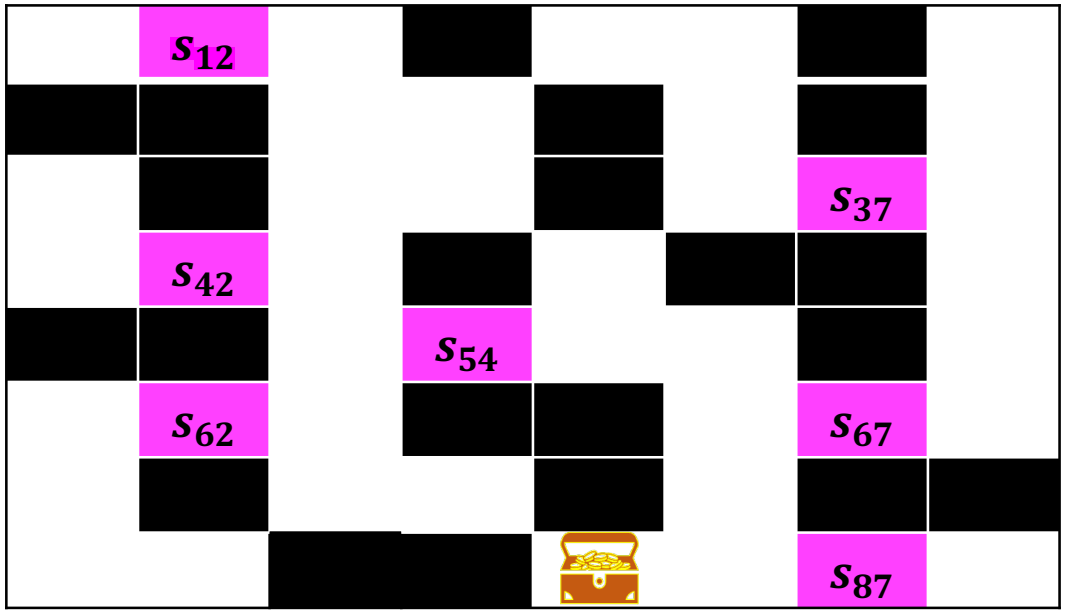
Searching in partially observable environments

Partially observable environments

- Observable environment: the agent knows its state.
- Unobservable environment: the agent doesn’t know its state.
- Partially observable environment: the agent can observe something, but not everything.

Searching in partially observable environments

Partially observable environment

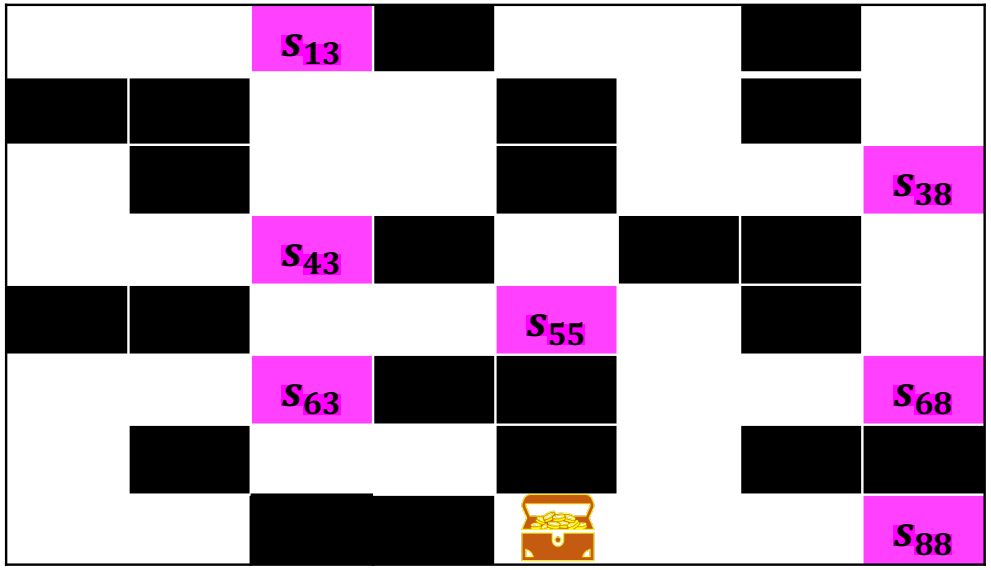


Suppose we begin by sensing the bit vector $\vec{o} = [0,1,0,1]^T$.

Then we know that our initial position must be one of these squares.

Searching in partially observable environments

Partially observable environment

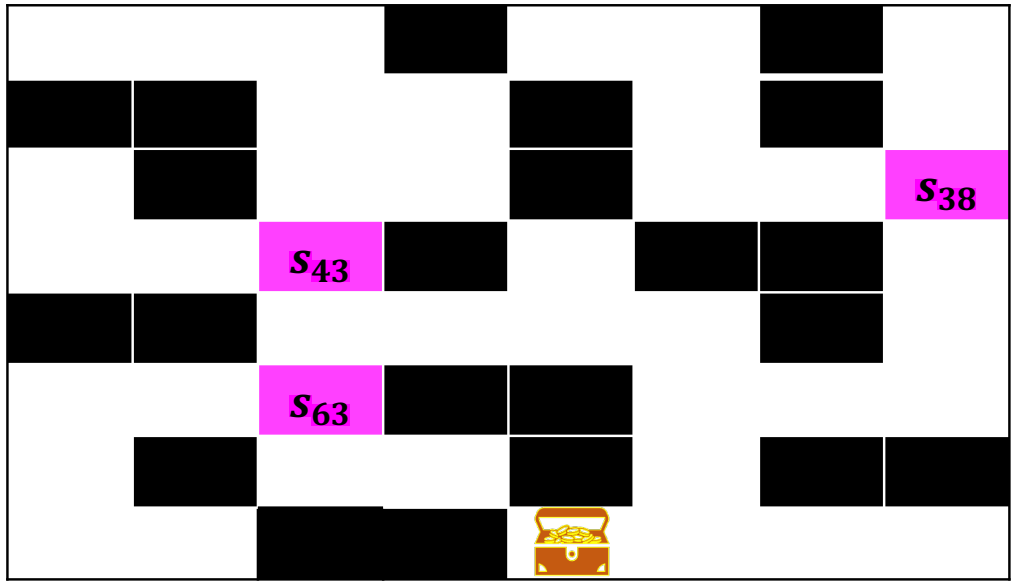


Move one step to the right.

Now we know that our position must be one of these squares.

Searching in partially observable environments

Partially observable environment



Now suppose we sense again, and measure the bit vector $\vec{o} = [1,0,0,0]^T$.

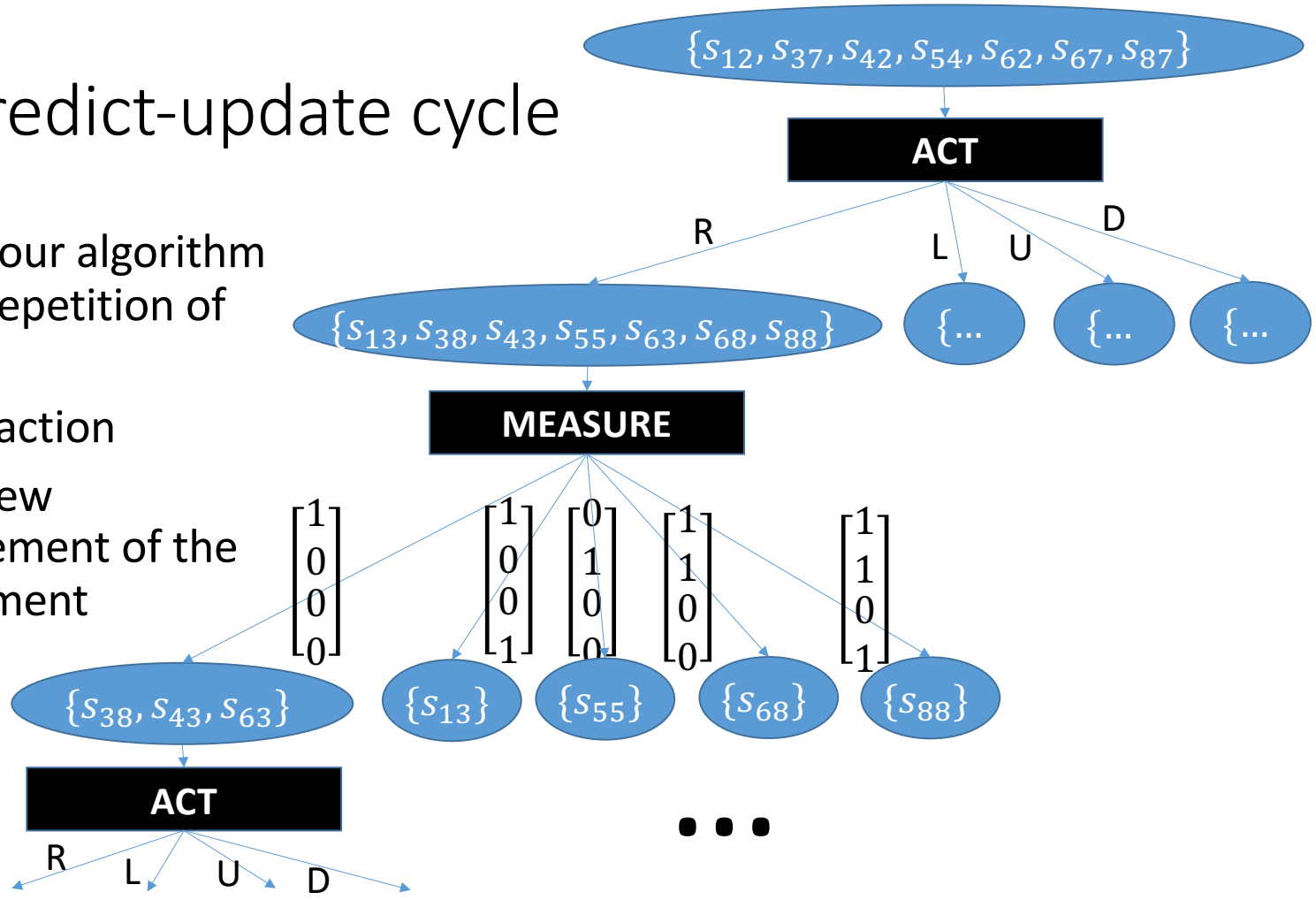
Then we know that our new position must be one of these squares.

Searching in partially observable environments

The predict-update cycle

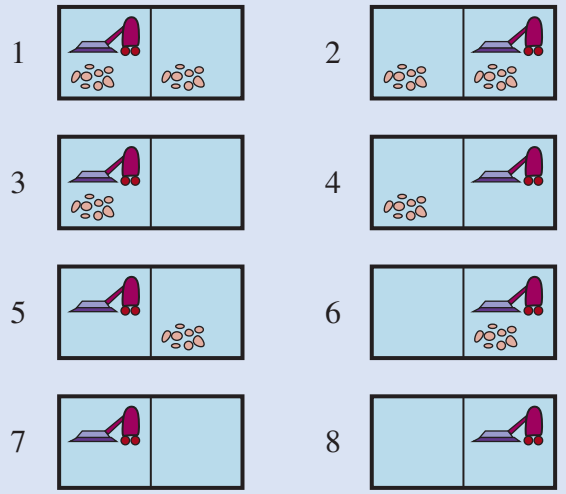
Notice that our algorithm is now the repetition of two steps:

1. Take an action
2. Take a new measurement of the environment



Searching in partially observable environments

- Consider a local-sensing vacuum world, agent has
  - a position sensor that yields the percept L in the left square, 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}.



Searching in partially observable environments

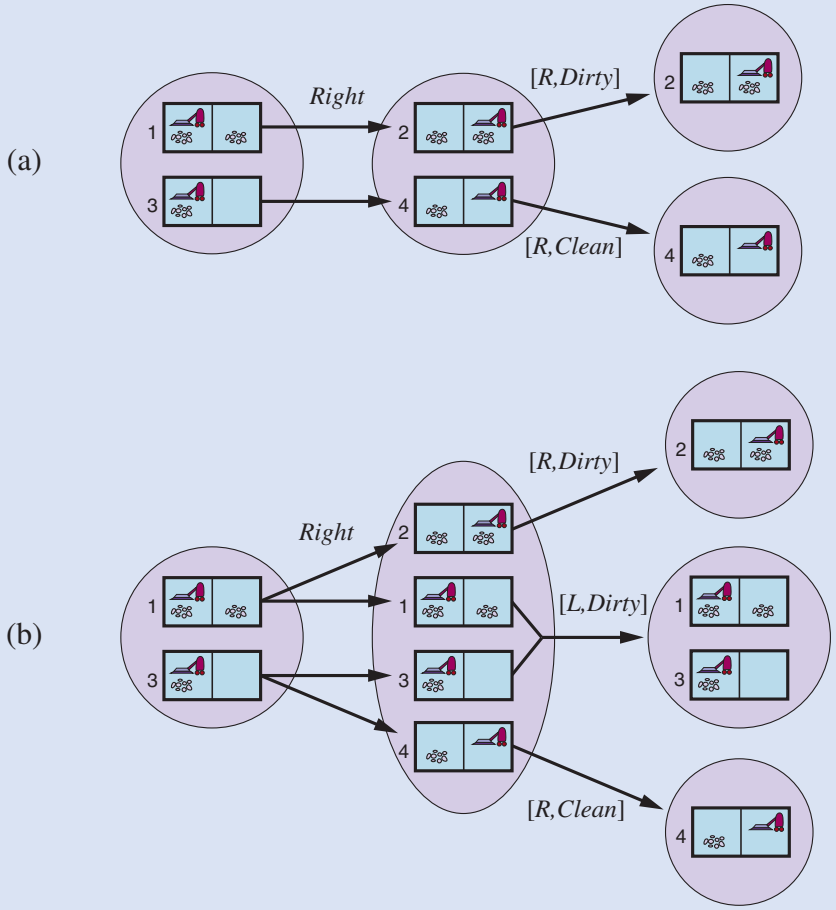


Figure 4.15 Two examples of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new predicted belief state with two possible physical states; for those states, the possible percepts are $[R, Dirty]$ and $[R, Clean]$,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

Searching in partially observable environments

- The prediction stage computes the belief state resulting from the action
  - $\hat{b}$ = RESULT(b, a) $\hat{b}$ means estimated belief state
  - To emphasize that this is a prediction 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):
  - POSSIBLE-PERCEPTS($\hat{b}$) = {o : o=PERCEPT(s) and s ∈ $\hat{b}$}.
- The update stage computes, for each possible percept, the belief state that would result from the percept.
  - b_o = UPDATE($\hat{b}$,o) = {s : o=PERCEPT(s) and s ∈ $\hat{b}$}.

Solving partially observable problems

- With this formulation, the AND-OR search algorithm can be applied directly to derive a solution.
- The solution is the conditional plan

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

An agent for partially observable environments

- An agent for partially observable environments
  - formulates a problem,
  - calls a search algorithm (such as AND-OR-SEARCH) to solve it,
  - executes the solution.
- There are two main differences between this agent and the one for fully observable deterministic environments
  - 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.
  - the agent will need to maintain its belief state as it performs actions and receives percepts.
    - prediction–observation–update process
    - 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:

$$b' = \textsc{Update}(\textsc{Predict}(b, a), o) .$$

An agent for partially observable environments

- Consider a modified 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.

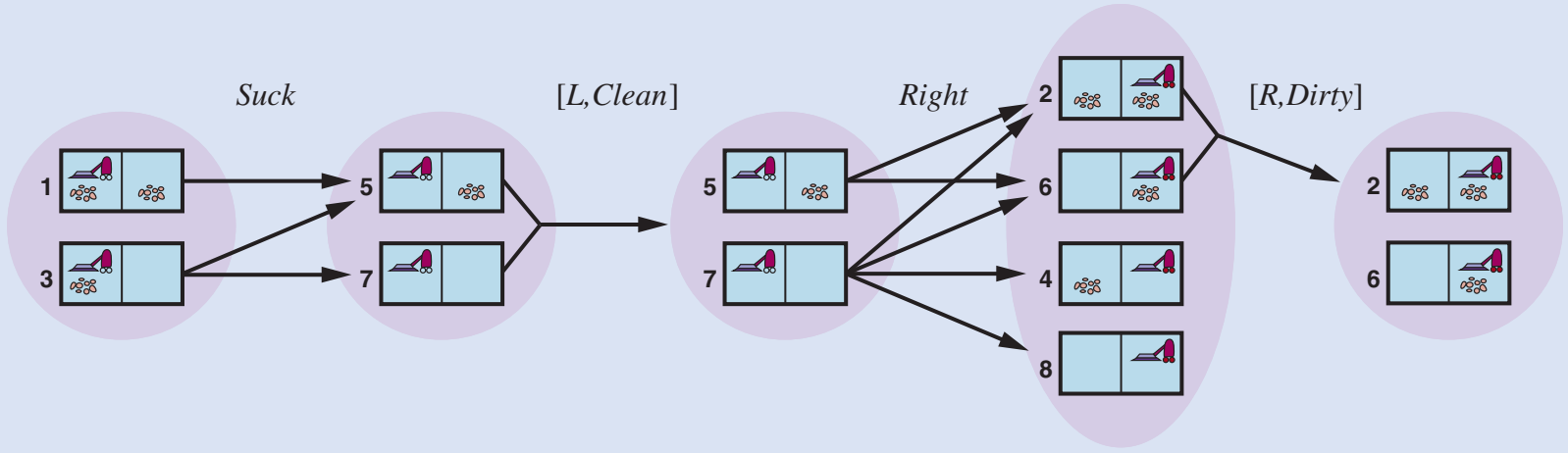


Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

An agent for partially observable environments

- The equation $b' = \text{UPDATE}(\text{PREDICT}(b,a),o)$. is called a recursive state estimator
- It computes the new belief state from the previous one rather than by examining the entire percept sequence.
- The computation has to happen as fast as percepts are coming in.
- As the environment becomes more complex, the agent will only have time to compute an approximate belief state,

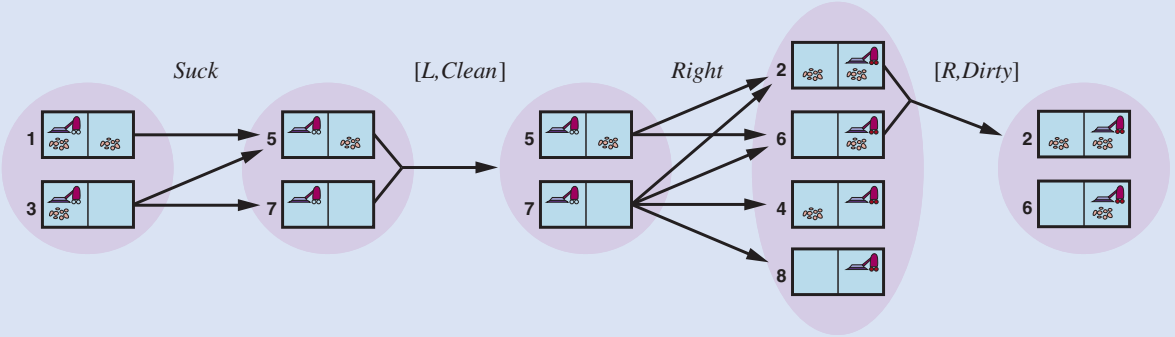
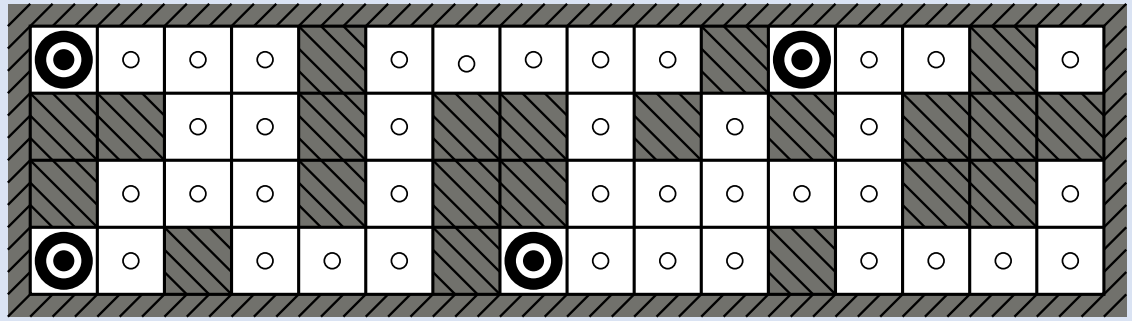


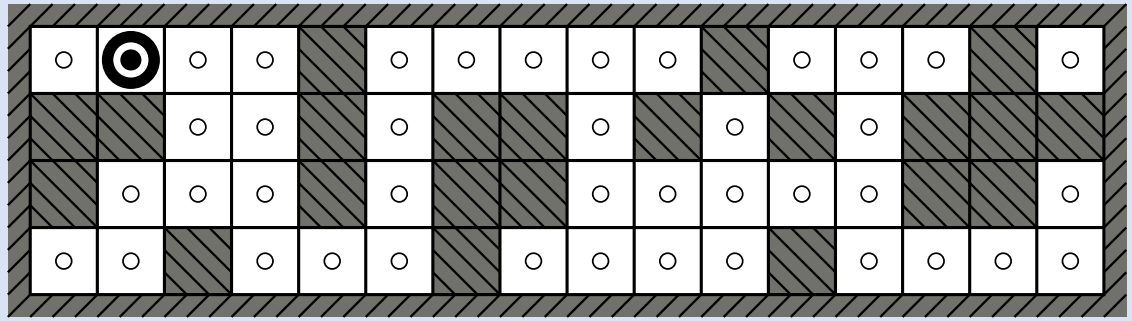
Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

An example in a discrete environment with deterministic sensors and nondeterministic actions.

- Sensors give perfectly correct data,
- The robot has a correct map of the environment.
- 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.



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



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

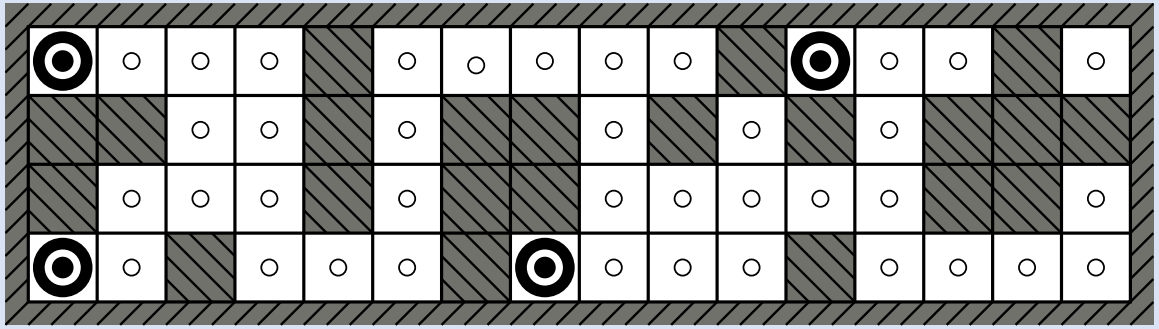
Figure 4.18 Possible positions of the robot, ⊙, (a) after one observation, $E_1 = 1011$, and (b) after moving one square and making a second observation, $E_2 = 1010$. When sensors are noiseless and the transition model is accurate, there is only one possible location for the robot consistent with this sequence of two observations.

Example

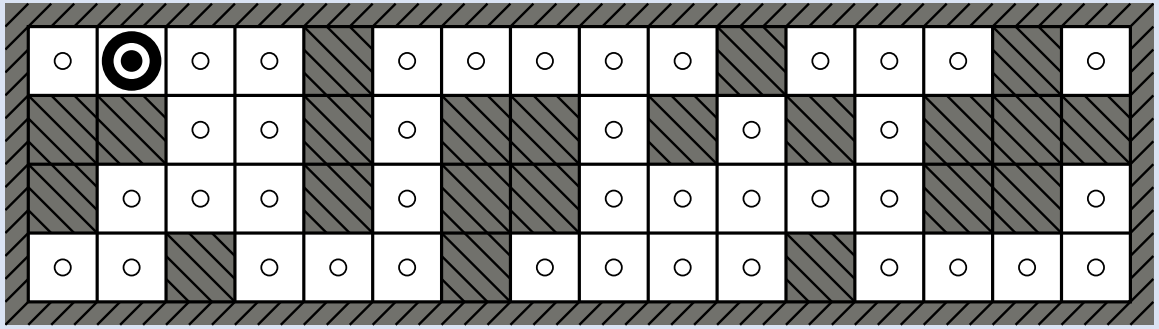
- The robot has just been switched on, and it does not know where it is
  - its initial belief state *b* consists of the set of all locations.
- Receives the percept 1011 and does an update using the equation

$$b_o = \text{UPDATE}(1011)$$

- Robot executes a Right action, but the result is nondeterministic.
- The new belief state $b_a = \text{PREDICT}(b_o, Right)$ contains all the locations that are one step away from the locations in b_o.
- When the second percept, 1010, arrives, the robot does $\text{UPDATE}(b_a, 1010)$



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



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

Figure 4.18 Possible positions of the robot, ⊙, (a) after one observation, $E_1 = 1011$, and (b) after moving one square and making a second observation, $E_2 = 1010$. When sensors are noiseless and the transition model is accurate, there is only one possible location for the robot consistent with this sequence of two observations.

$$\text{UPDATE}(\text{PREDICT}(\text{UPDATE}(b, 1011), Right), 1010).$$

The End!