Chapter 4:
Beyond Classical Search

CS-4365 Artificial Intelligence

Chris Irwin Davis

Overview

- 4.1 – Local Search Algorithms and Optimization Problems
- 4.2 – Local Search in Continuous Spaces
- 4.3 – Searching with Non-deterministic Actions
- 4.4 – Searching with Partial Observations
- ~~4.5 – Online Search Agents and Unknown Environments~~

Introduction

- Chapter 3 addressed a single category of problems: observable, deterministic, known environments where the solution is a *sequence of actions*.
- In this chapter, we look at what happens when these assumptions are relaxed.

Introduction

- Sections 4.1 and 4.2 cover algorithms that perform purely local search in the state space, evaluating and modifying one or more current states rather than systematically exploring *paths from an initial state*.
- In sections 4.3 and 4.4 we examine what happens when we relax the assumptions of determinism and observability.
  - The key idea is that if an agent cannot predict exactly what percept it will receive, then it will need to consider what to do under each contingency that its percepts may reveal.
  - With partial observability, the agent will also need to keep track of the states it might be in.

4.1 – Local Search Algorithms and Optimization Problems

Relevance of the Path to the Goal

- In many problems the *path to the goal is irrelevant*, e.g.
  - 8-Queens Problem
  - Integrated circuit design
  - Factory-floor layout
  - Vehicle routing
  - Portfolio Management
- If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all → "Local" Search

Local Search Algorithms

- The family of **local search algorithms** includes methods inspired by statistical physics (**simulated annealing**) and evolutionary biology (**genetic algorithms**).
- The search algorithms that we have seen so far are designed to explore search spaces systematically:
  - In those, systematicity was achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path and which have not.
  - When a goal is found, the *path* to that goal also constitutes a *solution* to the problem.

Local Search Algorithms

- **Local search** algorithms operate using a single **current node** (rather than multiple paths) and generally move only to neighbors of that node. Typically, the paths followed by the search are not retained.
- Although local search algorithms are not systematic, they have two key advantages:
  - (1) they use very little memory—usually a constant amount; and
  - (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

Optimization Problems

- In addition to finding goals, local search algorithms are useful for solving pure **optimization problems**, in which the aim is to find the best state according to an **objective function**.
- Many optimization problems do not fit the "standard" search model introduced in Chapter 3.
  - For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no "goal test" and no "path cost" for this problem.

State-space Landscape

- To understand local search, we will find it very useful to consider the state-space landscape (Figure 4.1 below).
- A landscape has both “location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function).

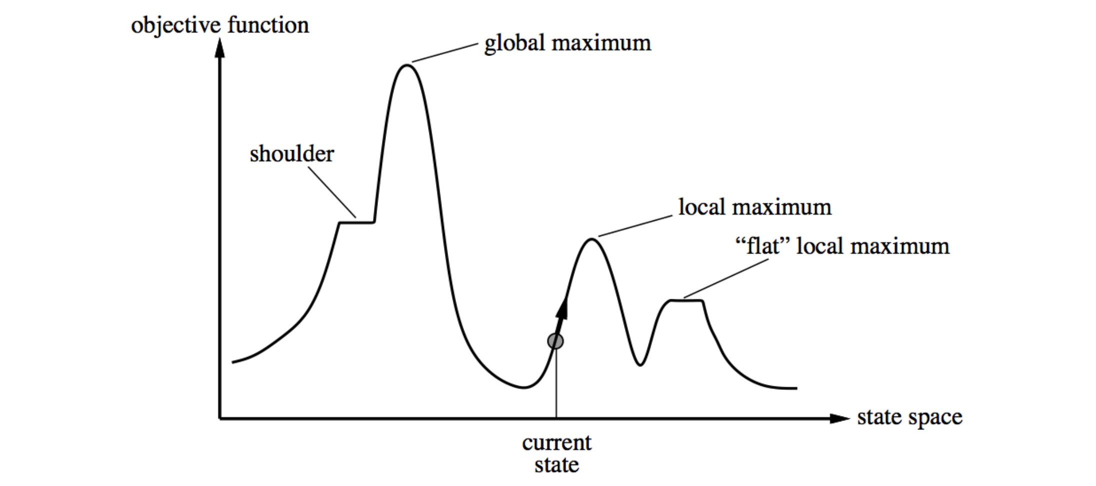
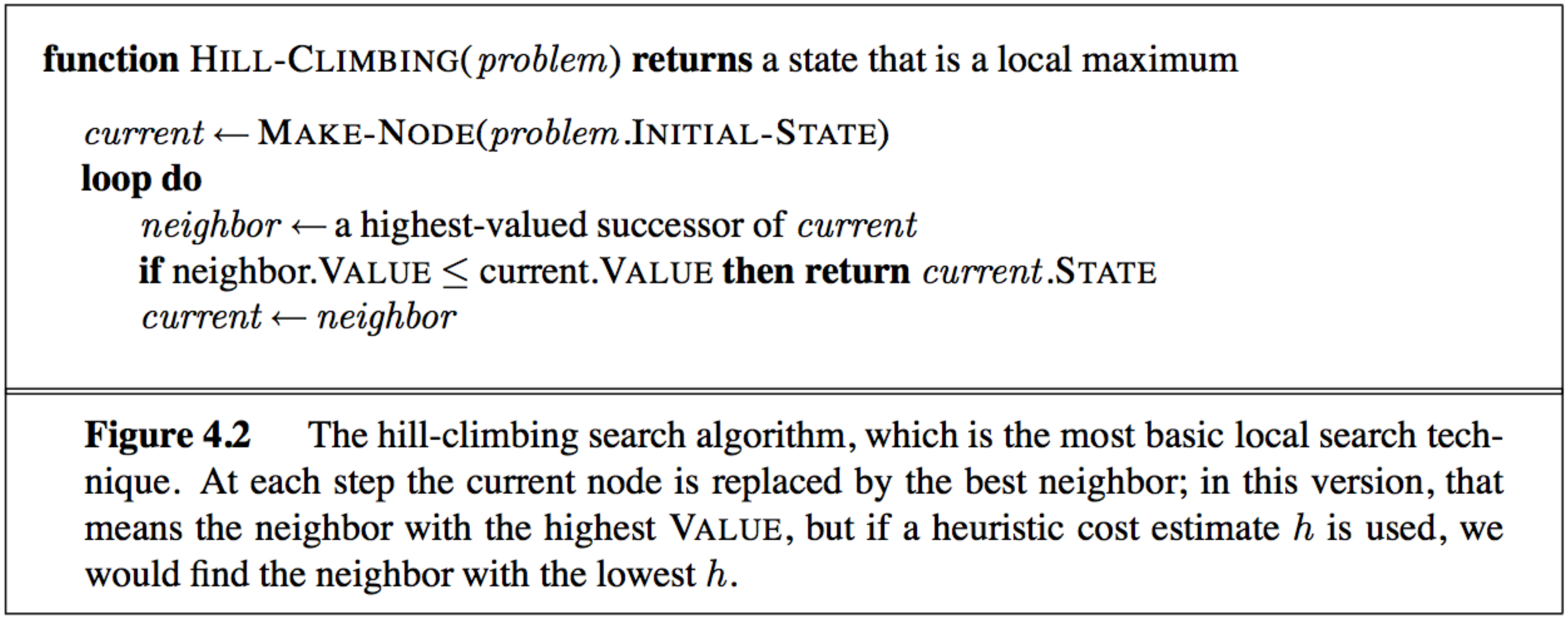


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill-climbing Search

- A loop that continually moves in the direction of increasing value—that is, uphill.
  - Or downhill if searching for minimal cost
- It terminates when it reaches a “peak” where no neighbor has a higher value.
-

Hill Climbing Search Algorithm



```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current $\leftarrow$ MAKE-NODE(problem.INITIAL-STATE)
  loop do
      neighbor $\leftarrow$ a highest-valued successor of current
      if neighbor.VALUE $\leq$ current.VALUE then return current.STATE
      current $\leftarrow$ neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Hill-climbing Search

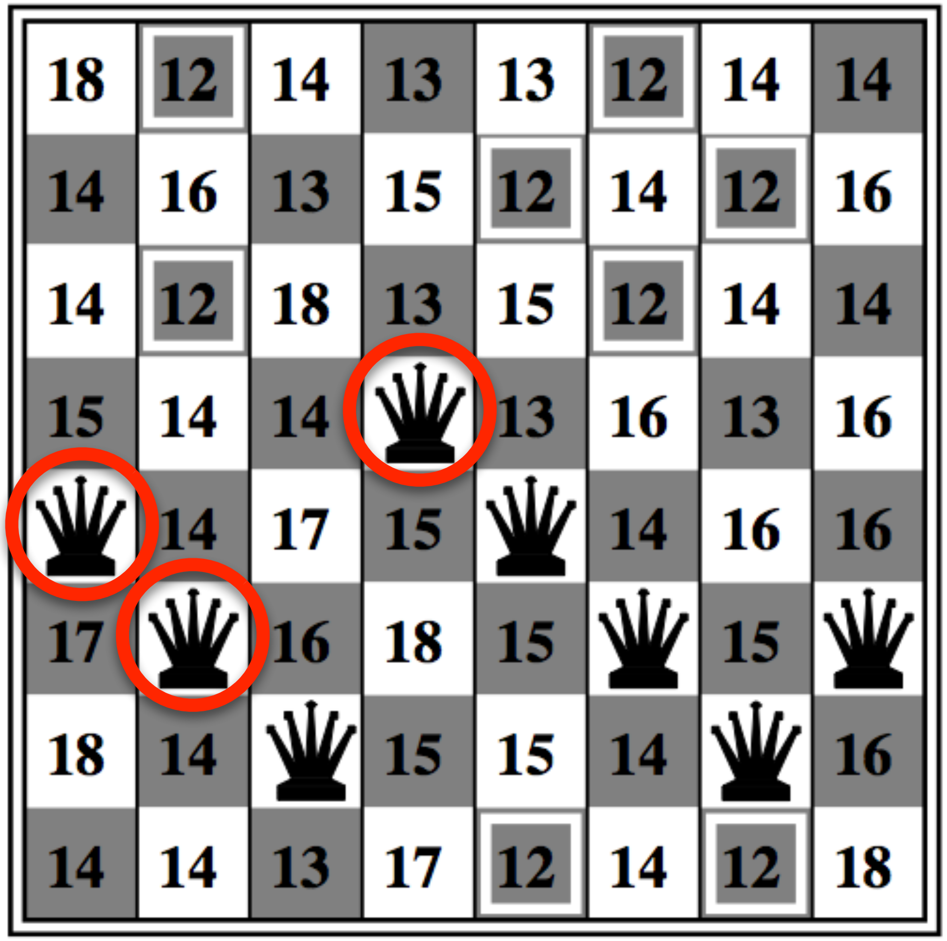
- Problems?
  - Local maxima
  - Ridges
  - Plateaus or shoulders
    - Allow sideways moves?
    - How many?

Hill Climbing: 8-Queens

- Local search algorithms typically use a complete-state formulation, i.e. each state would have 8 queens already on the board, one per column
- The successors of a state are all possible states generated by moving a single queen to another square in the same column
- The **heuristic cost function** *h* is the number of pairs of queens that are attacking each other, either directly or indirectly.
  - The global minimum of this function is zero, which occurs only at perfect solutions.

Hill Climbing: 8-Queens

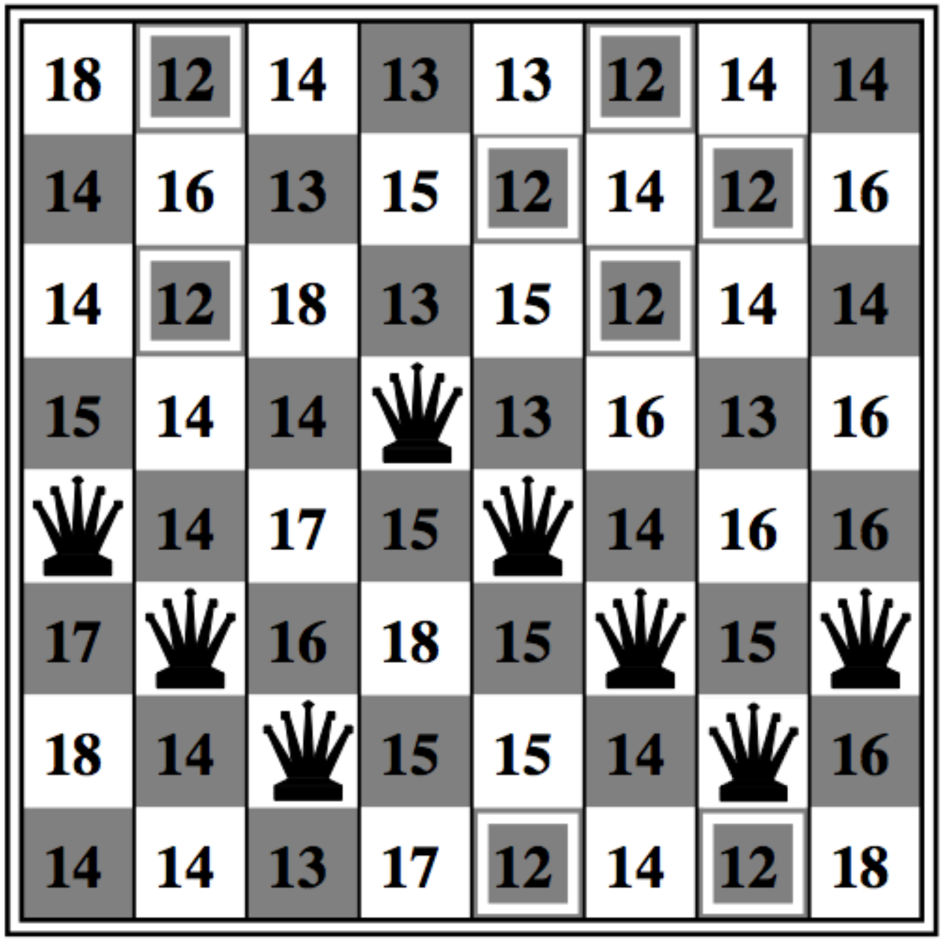
$h = 17$



18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

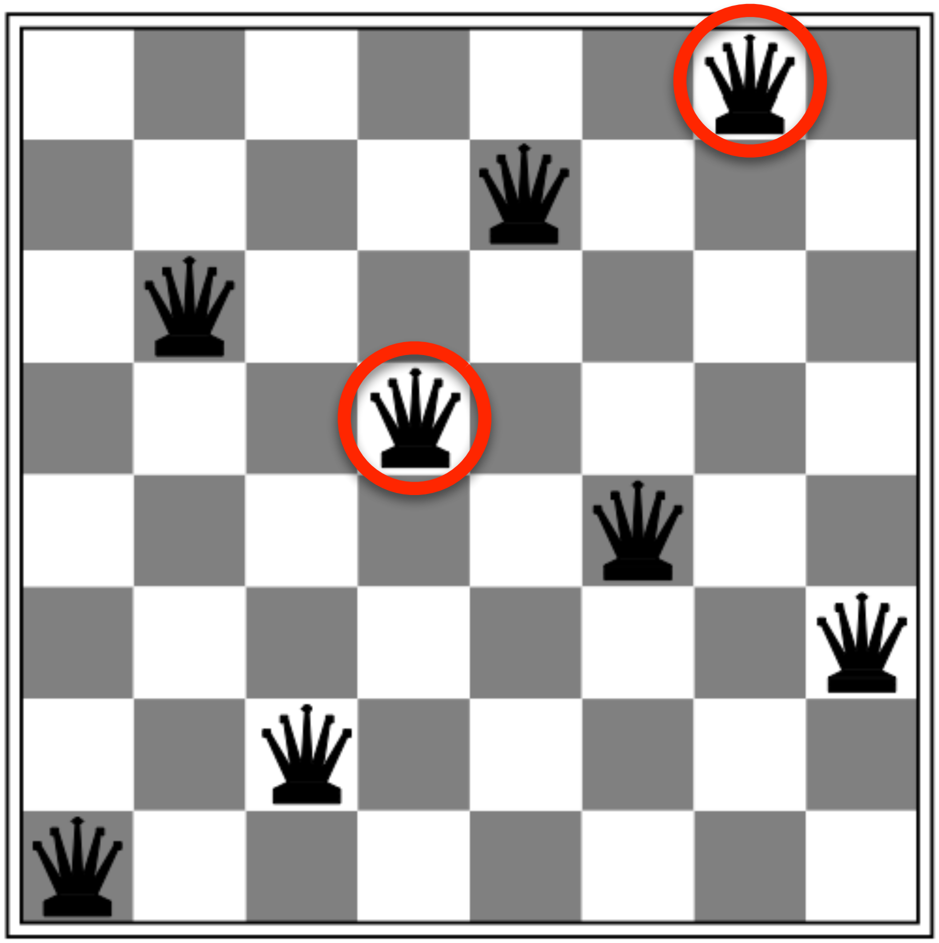
Hill Climbing: 8-Queens

$h = 12$



Hill-Climbing: Local Minimum

$h = 1$



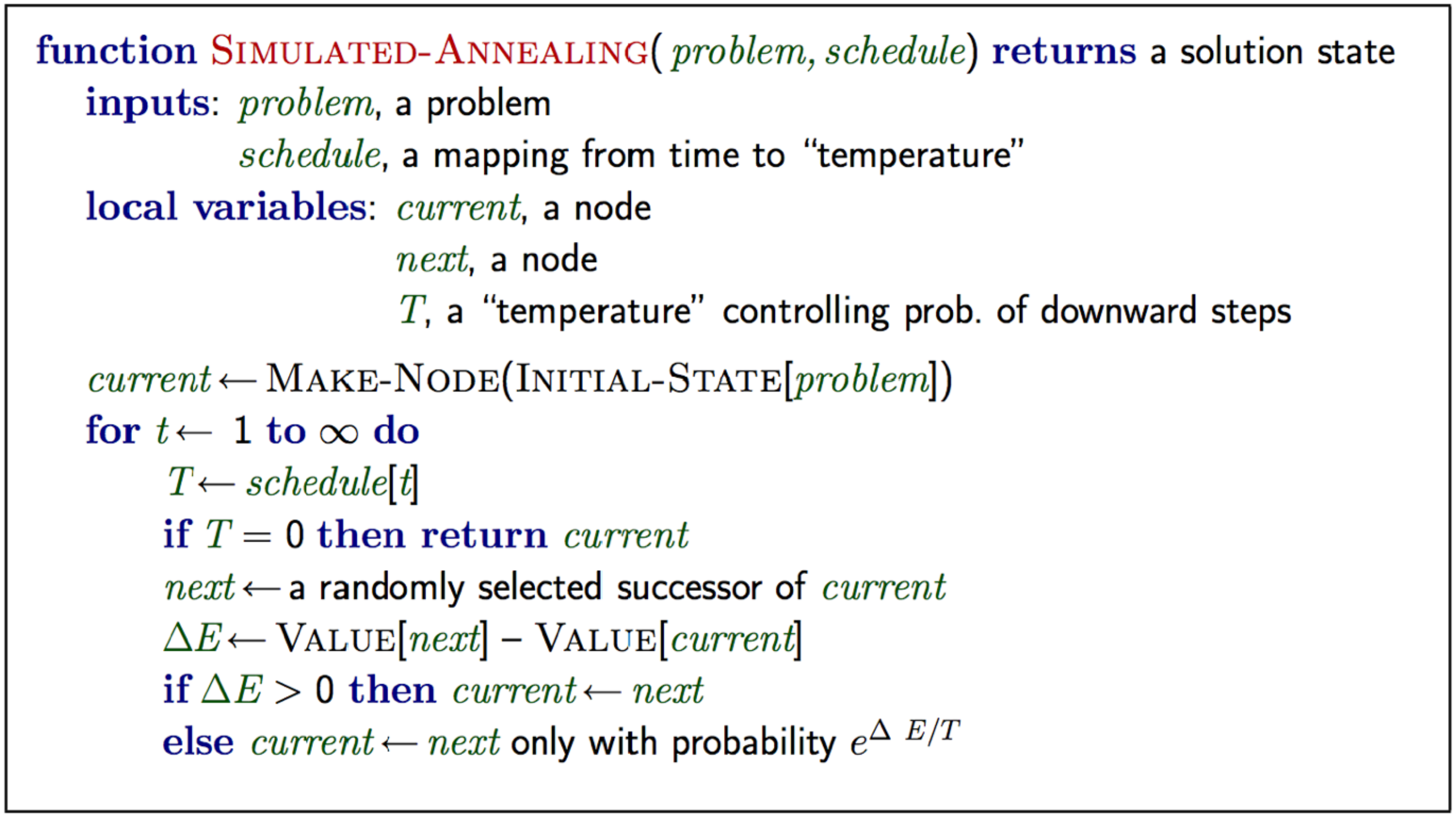
Simulated Annealing Search

- A hill-climbing algorithm that never makes “downhill” moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum.
- In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete, but extremely inefficient.
- Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness → **Simulated Annealing**

Simulated Annealing Search

- Imagine a ping pong ball on a bumpy surface
- If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local minima, but not hard enough to dislodge it from the global minimum.
- The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

Simulated Annealing Search



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     $T$, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for $t \leftarrow$ 1 to $\infty$ do
        $T \leftarrow$ schedule[$t$]
        if $T = 0$ then return current
        next ← a randomly selected successor of current
        $\Delta E \leftarrow$ VALUE[next] – VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

Local Beam Search

- Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations.
- Keep track of *k* states rather than just one
- Begin 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.

Local Beam Search

- At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence.
- In fact, the two algorithms are quite different. In a random-restart search, each search process runs *independently* of the others. In a local beam search, useful information is passed among the k parallel search threads.
  - For example, if one state generates several good successors and the other $k - 1$ states all generate bad successors, then the effect is that the first state says to the others, “Come over here, the grass is greener!” The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

Local Beam Search

- Problems
  - Can suffer from a lack of diversity among the *k* states—they can quickly become concentrated in a small region of the state space, i.e., effectively an expensive version of Hill-Climbing
- Solution?
- *Stochastic* Beam Search
  - A variant, analogous to stochastic hill climbing, helps to alleviate this problem
  - Instead of choosing the best *k* from the the pool of candidate successors, stochastic beam search chooses *k* successors at random, with the probability of choosing a given successor being an increasing function of its value.

Stochastic Beam Search

- Stochastic beam search bears some resemblance to the process of natural selection, whereby the “successors” (offspring) of a “state” (organism) populate the next generation according to its “value” (fitness).
- Difference?
  - Single parent vs. Two parents
- Genetic Algorithms

Genetic Algorithms

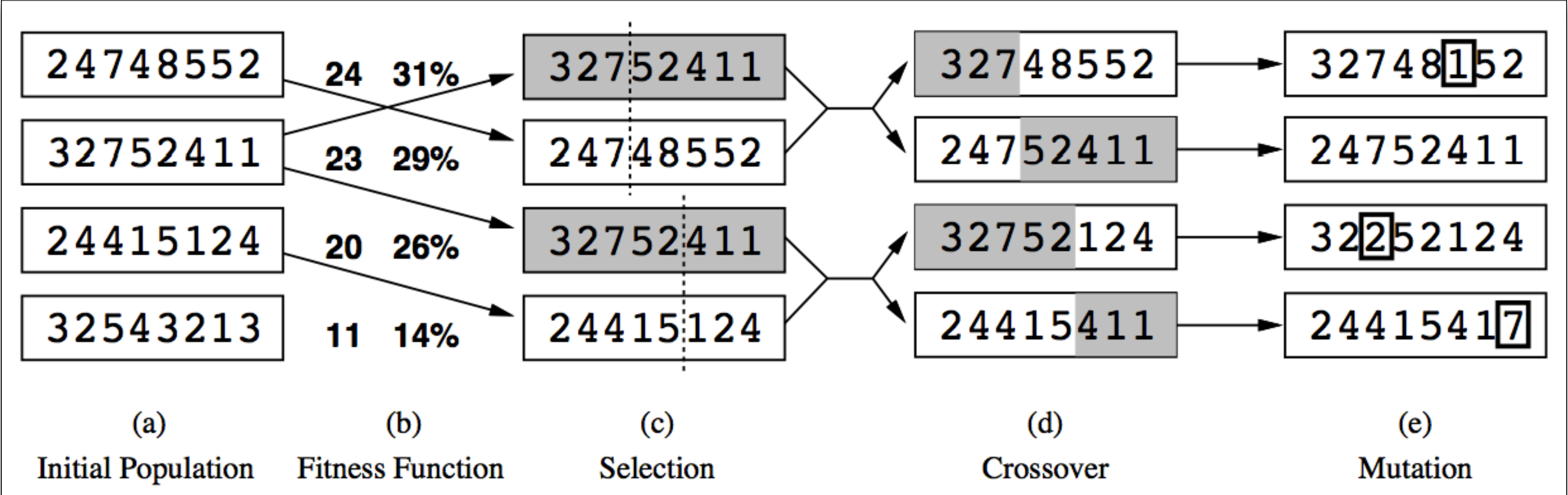
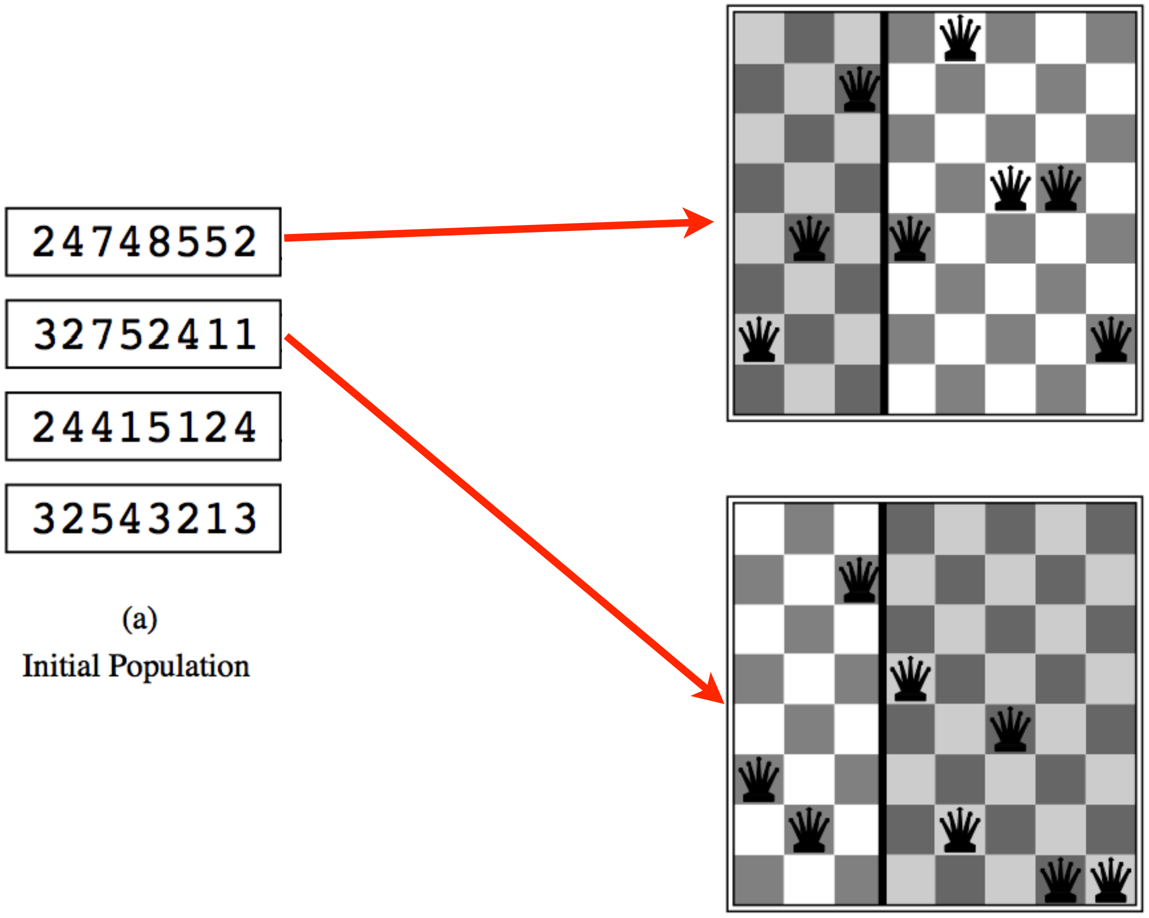


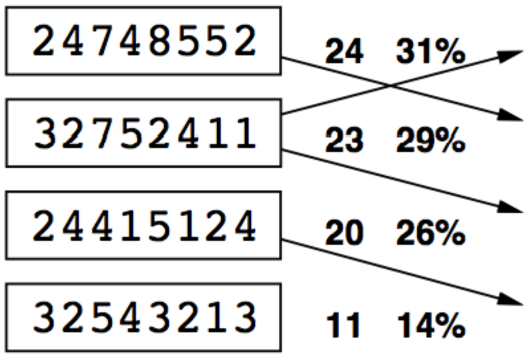
Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Genetic Algorithms



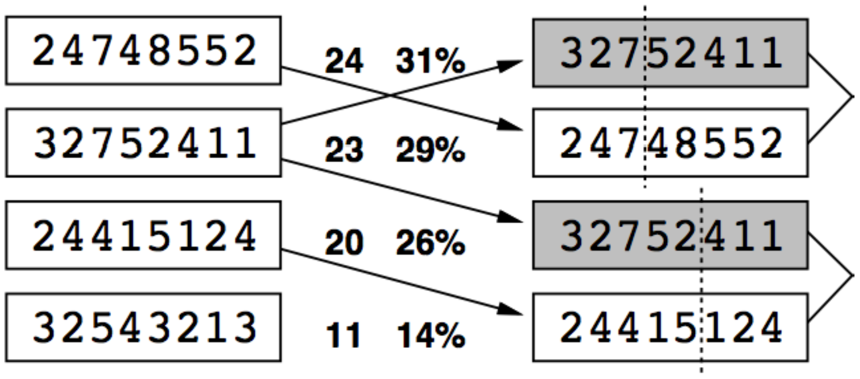
(a)
Initial Population

Genetic Algorithms



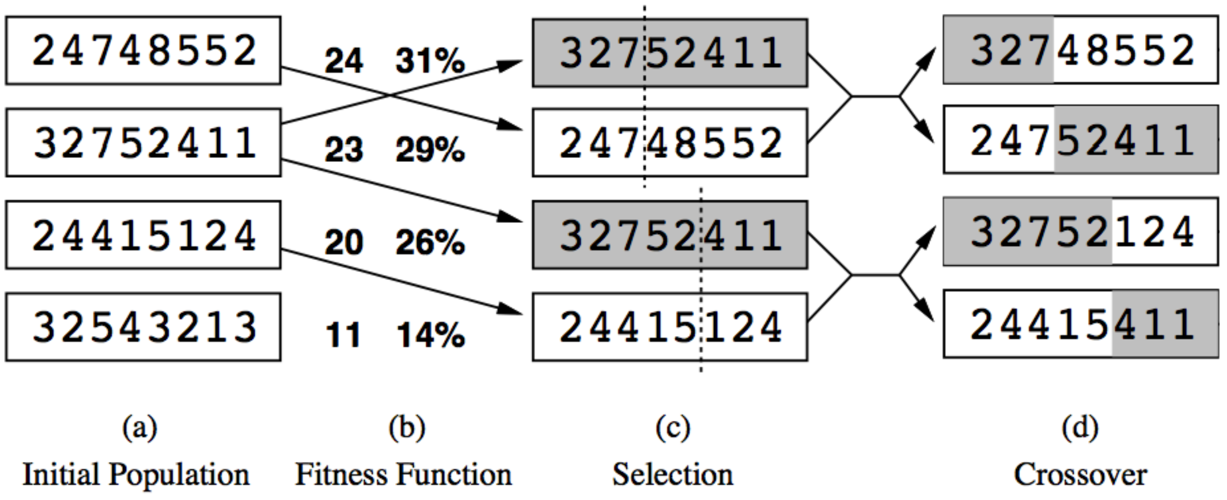
(a) Initial Population (b) Fitness Function

Genetic Algorithms

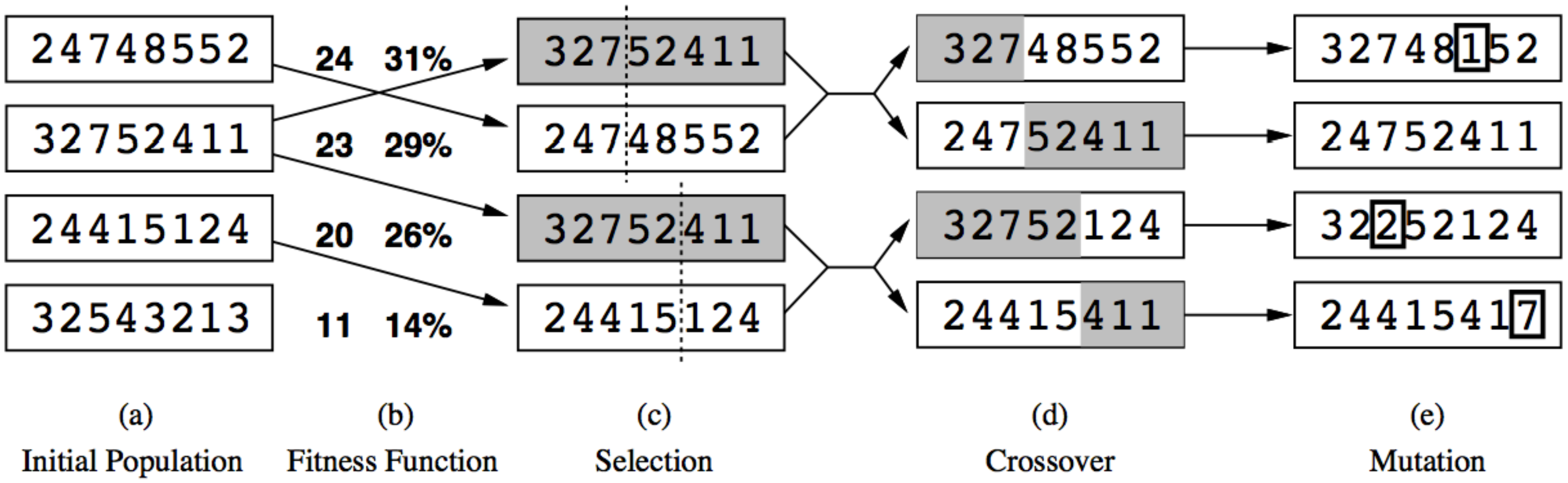


(a) Initial Population (b) Fitness Function (c) Selection

Genetic Algorithms



Genetic Algorithms



Genetic Algorithms

- Evaluation
  - Uphill tendency with random exploration* and exchange of information among parallel search threads
  - The primary advantage, if any, of genetic algorithms comes from the crossover operation.
- Schemata
  - It could be that putting the first three queens in positions 2, 4, and 6 (where they do not attack each other) constitutes a useful block that can be combined with other blocks to construct a solution.
  - 246*****

*** Like Stochastic Beam Search**

4.2 – Local Search in Continuous Spaces

Local Search in Continuous Spaces

- Chapter 2, we explained the distinction between discrete and continuous environments, pointing out that most real-world environments are continuous.
- None of the algorithms described so far (except for first-choice hill climbing and simulated annealing) can handle continuous state and action spaces, because they have infinite branching factors.
- This section provides a very brief introduction to some local search techniques for finding optimal solutions in continuous spaces.

Local Search in Continuous Spaces

- Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map (Figure 3.2) to its nearest airport is minimized.
- The state space is then defined by the coordinates of the airports: (x_1, y_1), (x_2, y_2), (x_3, y_3).
  - This is a six dimensional space, i.e. six variables. In general, states are defined by an n-dimensional vector of variables, x.
  - Moving around the state space corresponds to moving one or more of the airports on the map.
- The object function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ is easy to compute for any particular state once we compute the closest cities.

Local Search in Continuous Spaces

- One way to avoid continuous problems is simply to discretize the neighborhood of each state. For example, we can move only one airport at a time in either the x or y direction by a fixed amount $\pm \delta$.
- How many successor states?
  - With 6 variables, there are 12 possible successors for each state.
- How to search?
  - We can apply any of the previously described search algorithms.
  - We could also apply stochastic hill climbing and simulated annealing directly without having to discretize the space.
  - Many methods use the gradient of the landscape to find a maximum, ∇f.

Local Search in Continuous Spaces

- What are some applications of continuous search?

4.3 – Searching with Non-deterministic Actions

Searching with Non-deterministic Actions

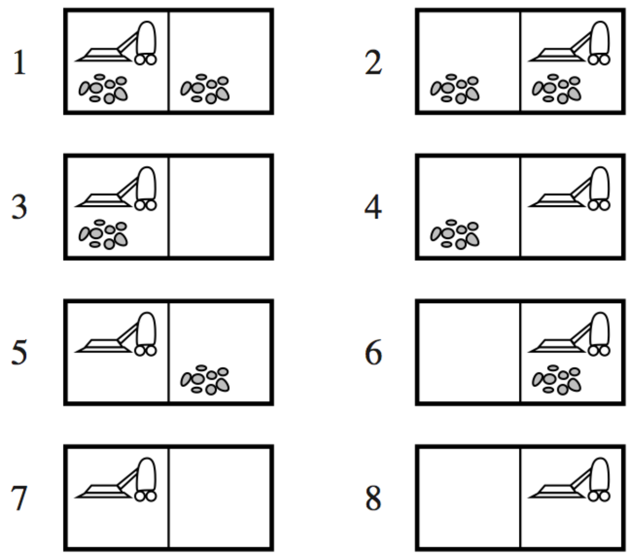
- In Chapter 3, we assumed that the environment is fully **observable** and **deterministic** and that the agent knows what the effects of each action are.
- Therefore, the agent can
  - calculate exactly which state results from any sequence of actions
  - always knows which state it is in
- Its percepts provide no new information after each action, although of course they tell the agent the initial state.

Searching with Non-deterministic Actions

- When the environment is either partially observable or nondeterministic (or both), percepts become useful:
  - In a **partially observable** environment, every percept helps to narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals.
  - When the environment is **nondeterministic**, percepts tell the agent which of the possible outcomes of its actions has actually occurred.
- In both cases, future percepts can't be determined in advance; and the agent's future actions will depend on those future percepts; so the solution to a problem is not a sequence, but a contingency plan (aka **strategy**) that specifies what to do depending on what percepts are received.

Erratic Vacuum World

- Recall that the state space has eight states, and
- Three actions—*Left*, *Right*, and *Suck*
- The goal is to clean up all the dirt (states 7 and 8).
- If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms in Chapter 3 and the solution is an action sequence.

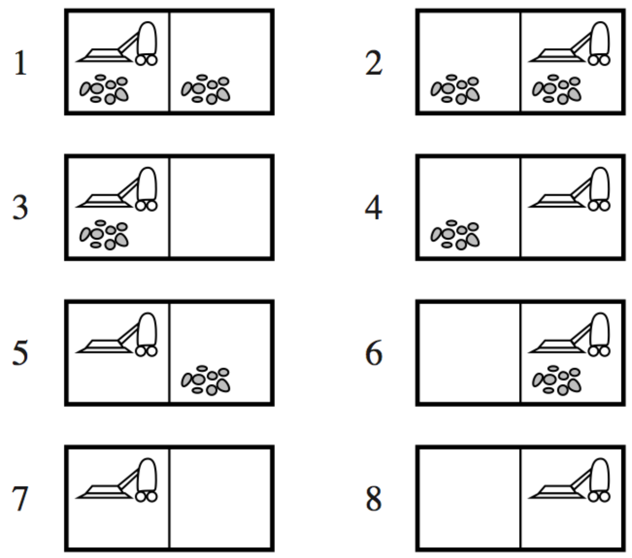


Erratic Vacuum World

- Now suppose that we introduce *nondeterminism* in the form of a powerful but erratic vacuum cleaner. In the erratic vacuum world, the Suck action works as follows:
  - When applied to a *dirty* square it cleans it and sometimes cleans up dirt in an adjacent square too.
  - When applied to a *clean* square it sometimes deposits dirt on the carpet.

Erratic Vacuum World

- Instead of defining the transition model by a `RESULT()` function that returns a single state, we use a `RESULTS()` function that returns a <u>*set*</u> of possible outcome states.
- For example, in the Erratic Vacuum World, the *Suck* action in state 1 leads to a state in the set {5, 7}—the dirt in the right-hand square may or may not be vacuumed up.



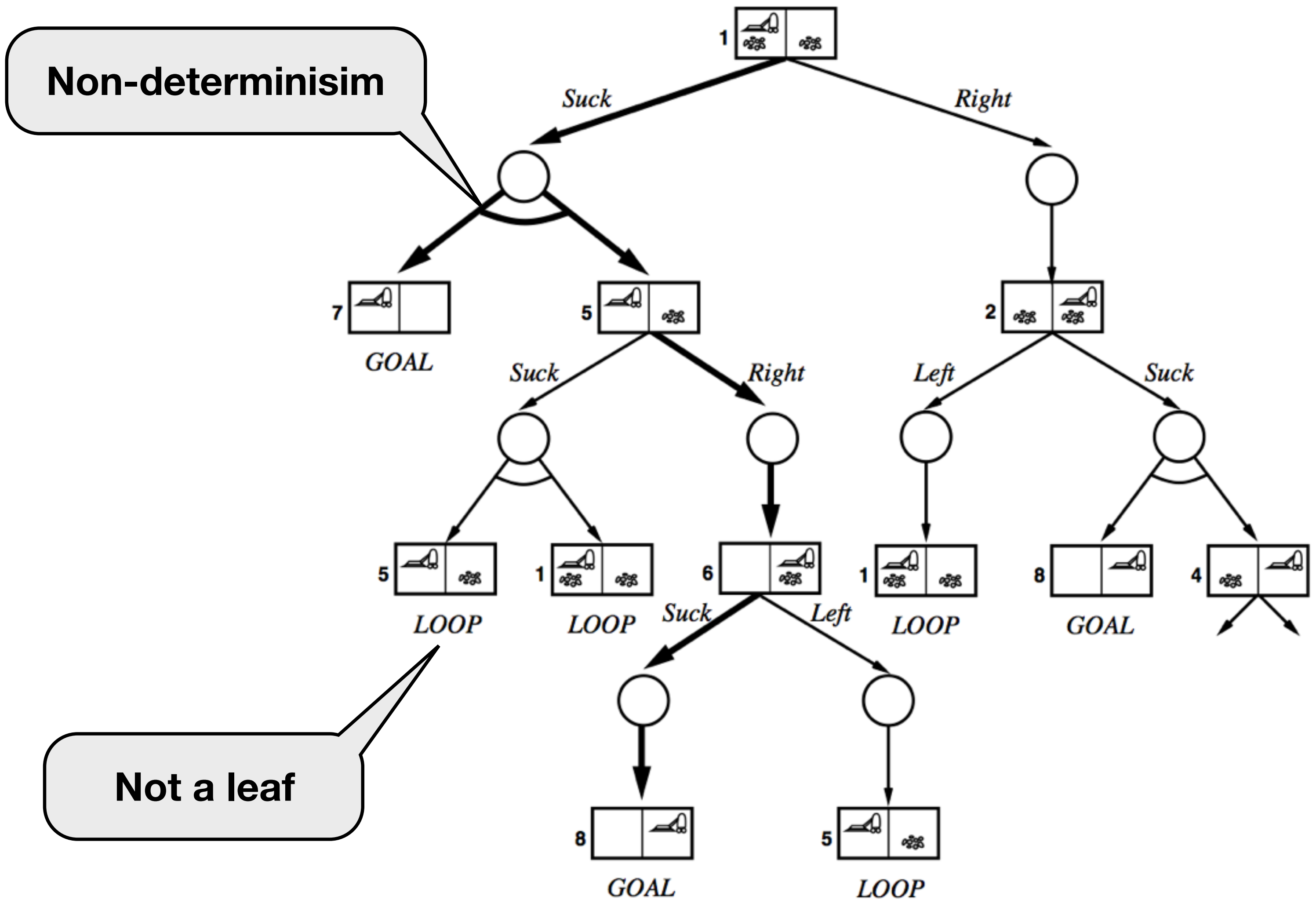
AND-OR Search Trees

- The next question is how to find contingent solutions to **nondeterministic** problems.
- In a **deterministic** environment (Chapter 3), the only tree branching is introduced by the *agent's own choices* in each state. We will 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 will call these nodes **AND nodes**.

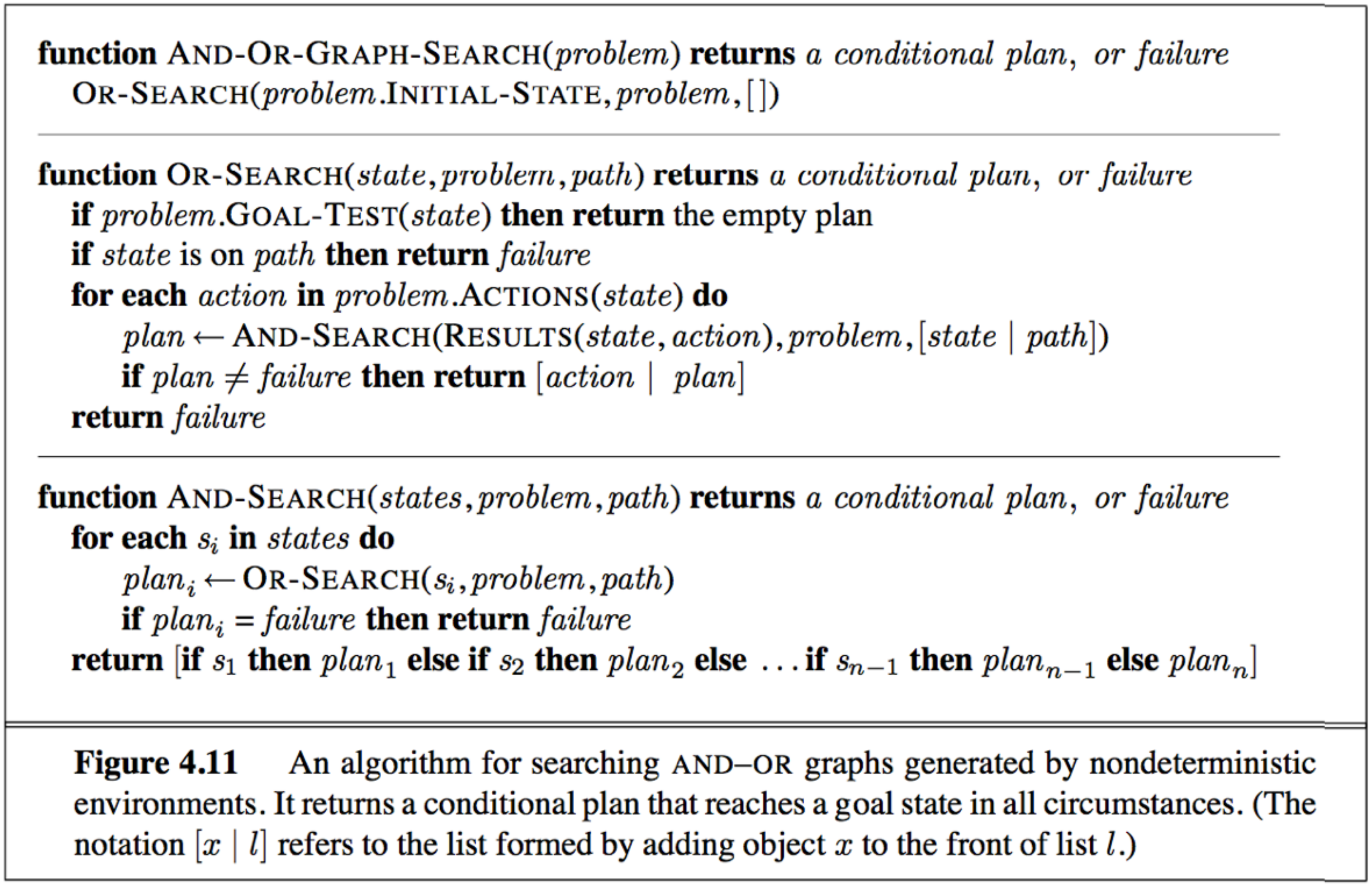
AND-OR Search Trees

- For example, the Suck action in state 1 leads to a state in the set {5, 7}, so the agent would need to find a plan for state 5 and for state 7.
- These two kinds of nodes alternate, leading to an AND-OR tree
- A solution for an AND-OR search problem is a subtree that
  - (1) has a goal node at every leaf,
  - (2) specifies one action at each of its OR nodes, and
  - (3) includes every outcome branch at each of its AND nodes.
- The solution is shown in bold lines in the figure

AND-OR Search Trees



AND-OR Search Trees



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

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
    if problem.GOAL-TEST(state) then return the empty plan
    if state is on path then return failure
    for each action in problem.ACTIONS(state) do
        plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
        if plan $\neq$ failure then return [action | plan]
    return failure

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

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

Try, Try Again

- Consider the **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 state set {1, 2}.
  - Figure 4.12 shows part of the the search graph; clearly, there are no longer any acyclic solutions from state 1, and AND-OR-GRAPH-SEARCH would return with failure.
  - There is, however, a cyclic solution, which is to keep trying Right until it works.
  - We can express this solution by adding a label to denote some portion of the plan and using that label later instead of repeating the plan itself.

Slippery Vacuum World

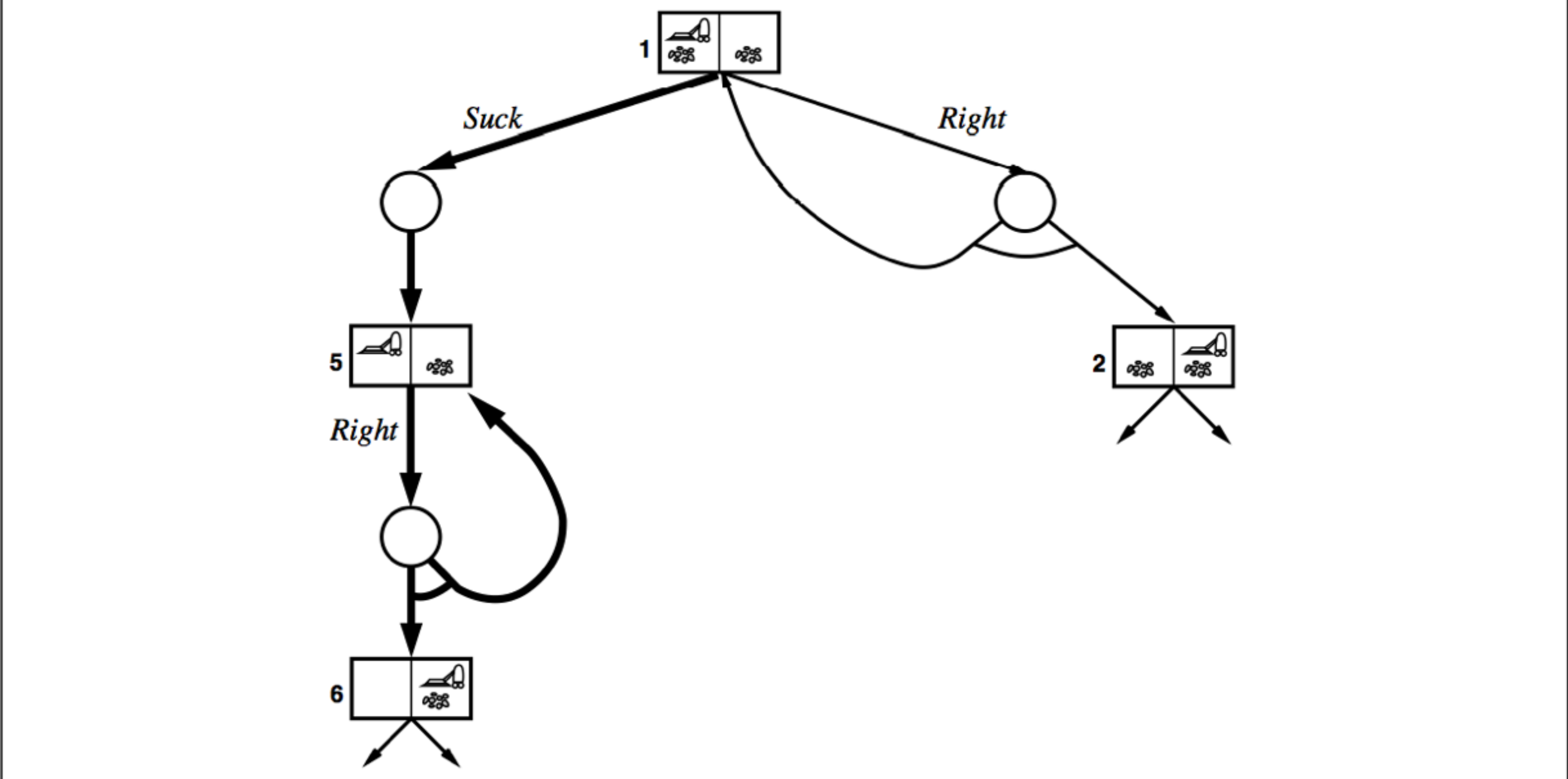


Figure 4.12 Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably

Slippery Vacuum World

- In general a cyclic plan may be considered a solution, provided that every leaf is a goal state and a leaf is reachable from every point in the plan.
- The key realization is that a loop in the state space back to a state *L* translates to a loop in the plan back to the point where the sub-plan for state *L* is executed.*
- Given the definition of a cyclic solution, an agent executing such a solution will eventually reach the goal provided that each outcome of a nondeterministic action *eventually* occurs.

* Modifications needed to AND-OR-GRAPH-SEARCH are covered in Exercise 4.4.

4.4 – Searching with Partial Observations

Searching with No Observation

- When the agent's percepts provide no information at all, we have what is called called a **sensorless** or sometimes a **conformant** problem
- Solvable?
  - At first, one might think the sensorless agent has no hope of solving a problem if it has no idea what state it's in
  - Sensorless problems are often solvable

Searching with No Observation

- In the sensorless vacuum world, the agent knows only that its initial state is one of the set {1, 2, 3, 4, 5, 6, 7, 8}
- Now, consider what happens if it tries the action Right. This will cause it to be in one of the states {2,4,6,8}—the agent now has more information!
- Furthermore, the action sequence [Right,Suck] will always end up in one of the states {4, 8}
- Finally, the sequence [Right,Suck,Left,Suck] is guaranteed to reach the goal state 7 no matter what the start state. We say that the agent can **coerce** the world into state 7

Searching with No Observation

- Avoid the standard search algorithms, which treat belief states as black boxes just like any other problem state.
- Instead, we can look inside the belief states and develop incremental belief-state search algorithms that build up the solution one physical state at a time.

Searching with No Observation

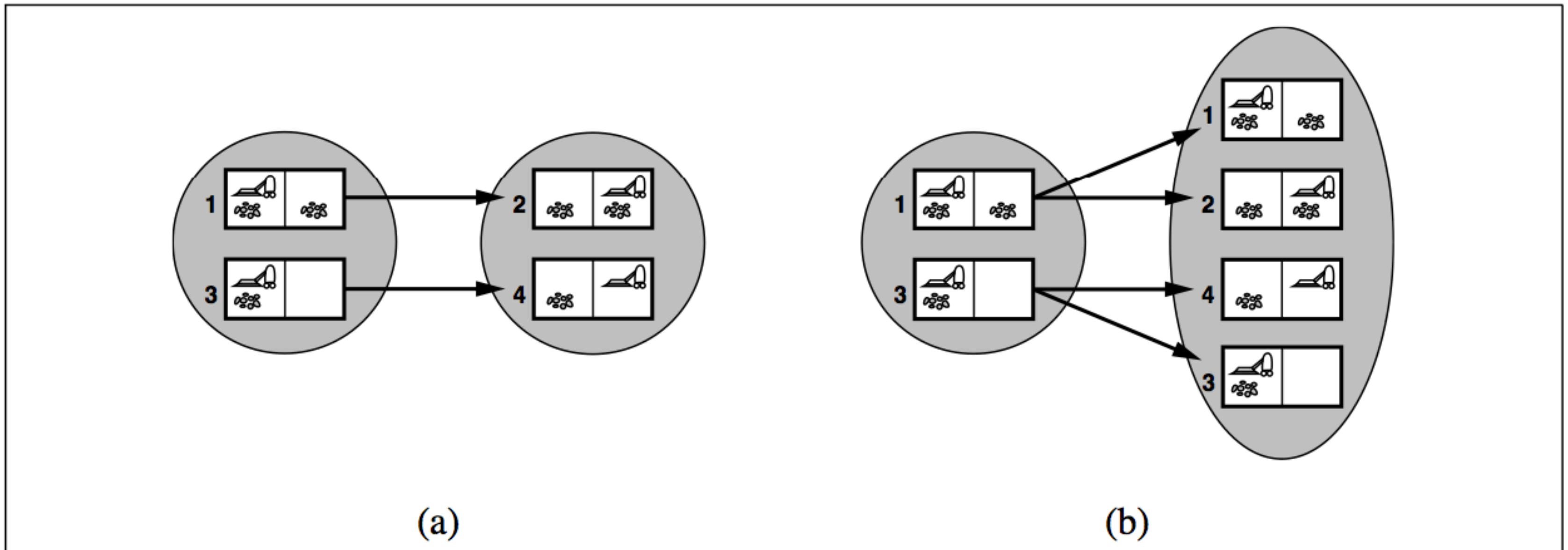


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

Searching with No Observation

- For example, in the sensorless vacuum world, the initial belief state is {1, 2, 3, 4, 5, 6, 7, 8}, and we have to find an action sequence that works in all 8 states.
- Find a solution that works for state 1; then check if it works for state 2; if not, find a different solution for state 1, etc.
- Just as an AND-OR search has to find a solution for every branch at an AND node, this algorithm has to find a solution for every state in the belief state; the difference is that
  - **AND-OR search** can find a different solution for each branch, whereas an
  - **Incremental belief-state search** has to find one solution that works for all the states.

Searching with No Observation

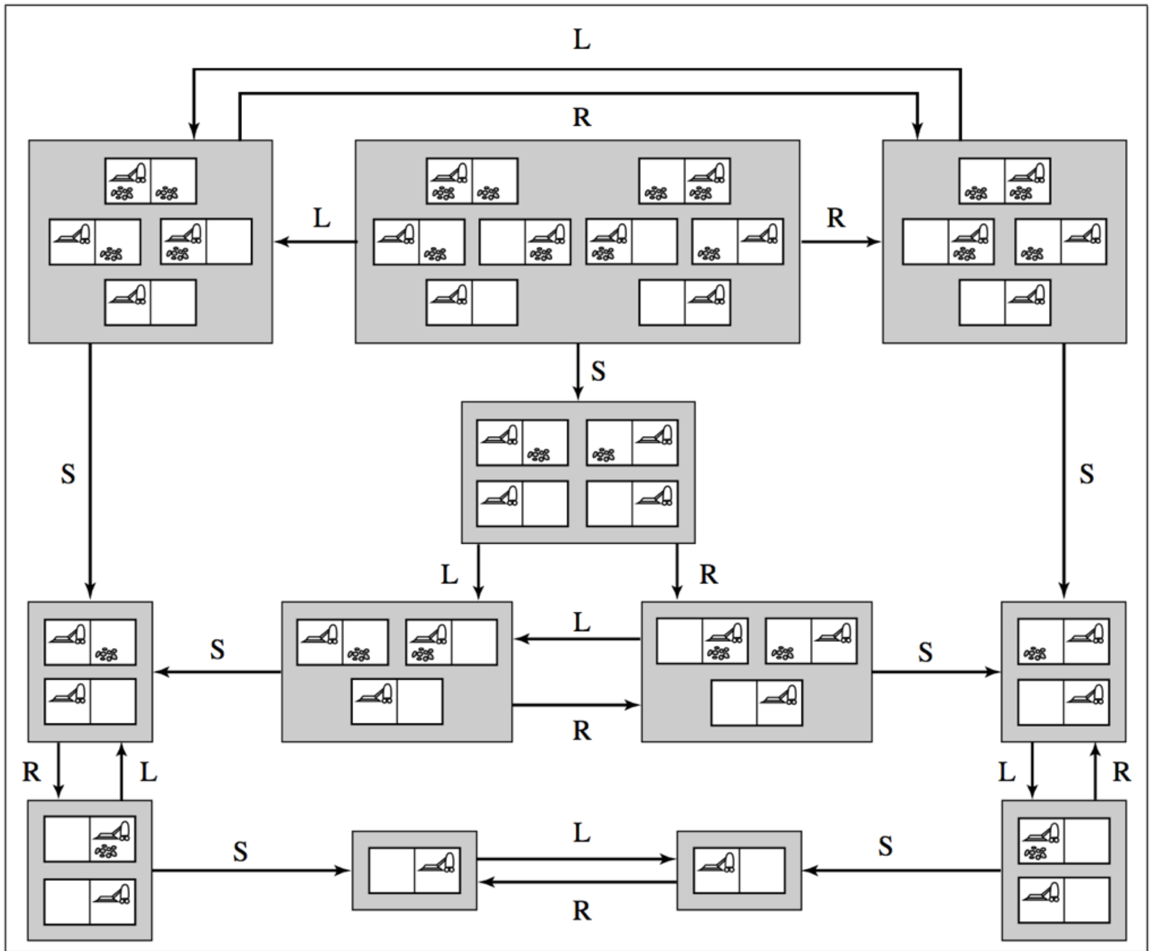


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete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

How Observations Supply Information

- For a general *partially* observable problem, we have to specify how the environment generates percepts for the agent.
- For example, we might define the local-sensing vacuum world to be one in which the agent has a position sensor and a local dirt sensor, but no sensor capable of detecting dirt in other squares.

How Observations Supply Information

- The formal problem specification includes a PERCEPT(*s*) function that returns the percept received in a given state
  - If sensing is nondeterministic, then we use a PERCEPTS function that returns a *set* of percepts
- For example, in the local-sensing vacuum world, the PERCEPT in state 1 is [A,Dirty]
  - Fully observable problems are a special case in which PERCEPT(*s*) = *s* for every state *s*, while
  - sensorless problems are a special case in which PERCEPT(*s*)=null

How Observations Supply Information

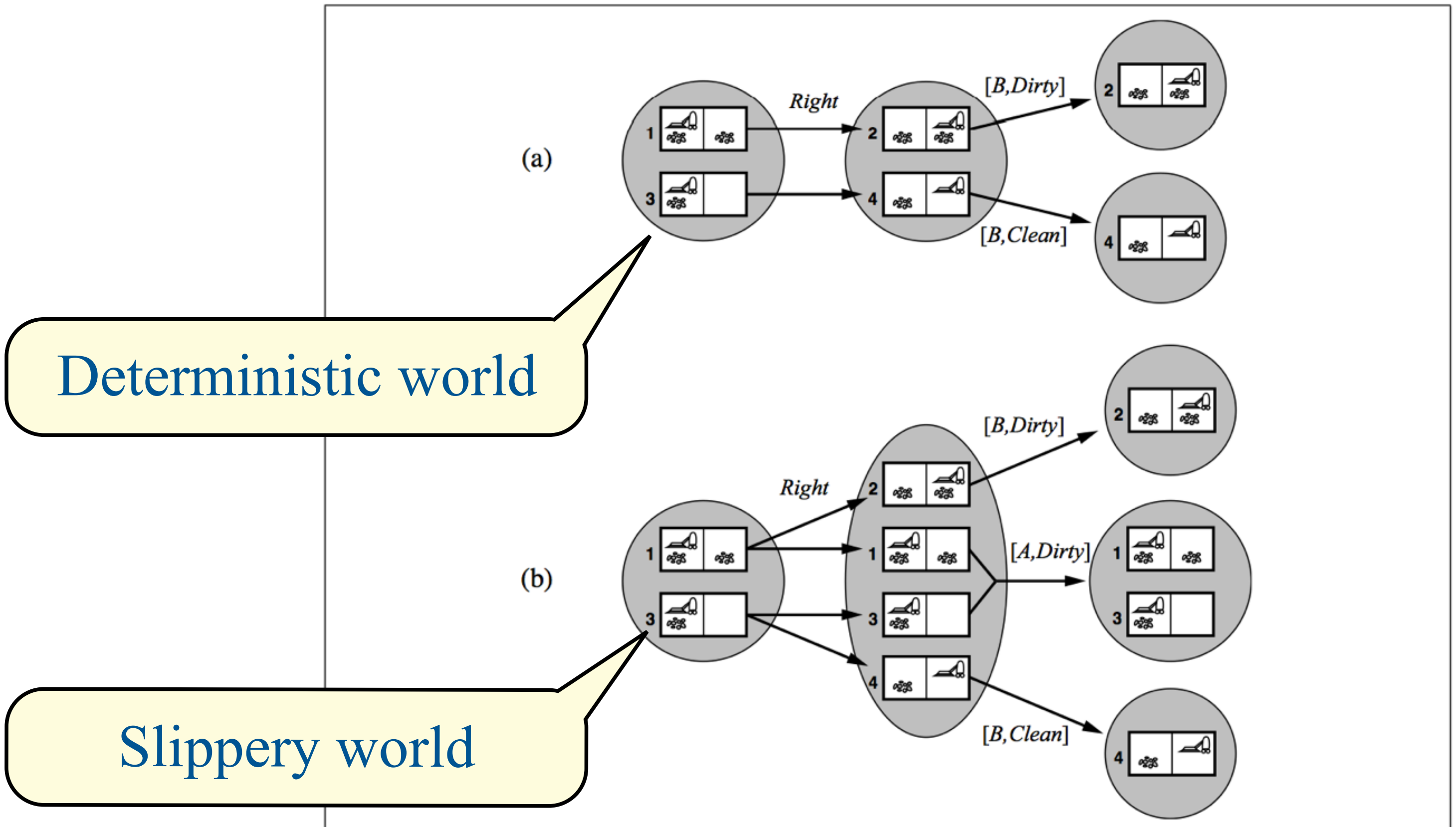
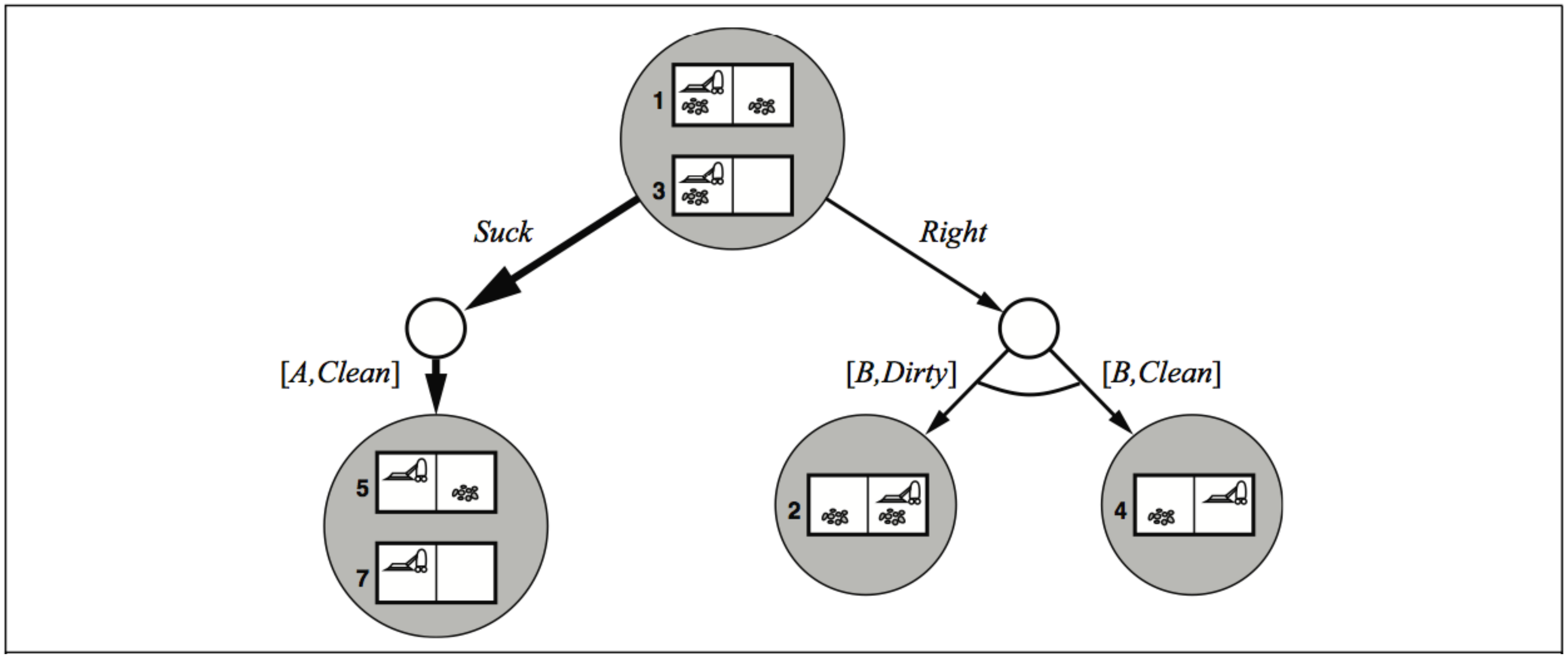


Figure 4.15 Two example of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are $[B, Dirty]$ and $[B, Clean]$, leading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

Solving Partially Observable Problems

- The preceding section showed how to formulate a nondeterministic belief-state problem—in particular, the RESULTS function—from
  - an underlying physical problem, and
  - the PERCEPT function
- Given such a formulation, the AND-OR search algorithm of Figure 4.11 can be applied directly to *derive a solution*…

Solving Partially Observable Problems



- Because we supplied a belief-state problem to the AND-OR search algorithm, it returned a conditional plan that tests the belief state rather than the actual state
- *Suck* is the first step of the solution

An Agent for Partially Observable Environments

- The design of a problem-solving agent for *partially observable environments* is similar to the simple problem-solving agent in Figure 3.1:
  - The agent formulates a problem, calls a search algorithm (such as AND-OR-GRAPH-SEARCH) to solve it, and executes the solution.
- There are two main differences:
  - First, the solution to a problem will be a conditional plan rather than a sequence; if the first step is an if–then–else expression, the agent will need to test the condition in the if-part and execute the then-part or the else-part accordingly.
  - Second, the agent will need to maintain its belief state as it performs actions and receives percepts.

~~4.5 – Online Search Agents and Unknown Environments~~