# Local Search in Continuous Spaces

AIMA V3 – Chp. 4 Section 4.2

# Continuous state spaces

Suppose we want to find 3 locations to build three airports in Romania:

- 6-D state space defined by $(x_1, y_2)$, $(x_2, y_2)$, $(x_3, y_3)$
- objective function $f(x_1, y_2, x_2, y_2, x_3, y_3) =$
  sum of squared distances from each city to nearest airport

# Continuous state spaces

In continuous spaces, we have two options:

- ♢ Discretize the allowed values of the variables (modify by fixed amount)
- ♢ Use the gradient

# Continuous state spaces - Discretization

Discretization methods turn continuous space into discrete space,

Then we can consider $\pm\delta$ change in each coordinate

# Continuous state spaces - Gradient Approach

♢ Use the gradient

The standard way to solving continuous problems.

Gradient methods compute

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

to increase/reduce $f$, e.g., by $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$

Problems w/ choosing step size, slow convergence

# Searching with Non-Deterministic Actions

AIMA V3 – Chp. 4 Section 4.3

# Example: vacuum world

The rest of the chapter will not be covered in full, but we have here a summary of other types of problems, so that you know where to look if you face those types of problems.

So far, we dealt with problems where the environment was fully observable and deterministic, so the agent knows what the effects of each action are.

But in real life, the environment is more challenging: either **partially observable** or **non-deterministic**.

# Problem types

Deterministic, accessible/fully observable $\Longrightarrow$ *single-state problem*

Deterministic, inaccessible/partially observable $\Longrightarrow$ *multiple-state problem*

Nondeterministic, inaccessible/partially observable $\Longrightarrow$ *contingency problem*

- must use sensors during execution
- solution is a *tree* or *policy*
- often *interleave* search, execution

Unknown state space $\Longrightarrow$ *exploration problem* ("online search")

# The erratic vacuum world

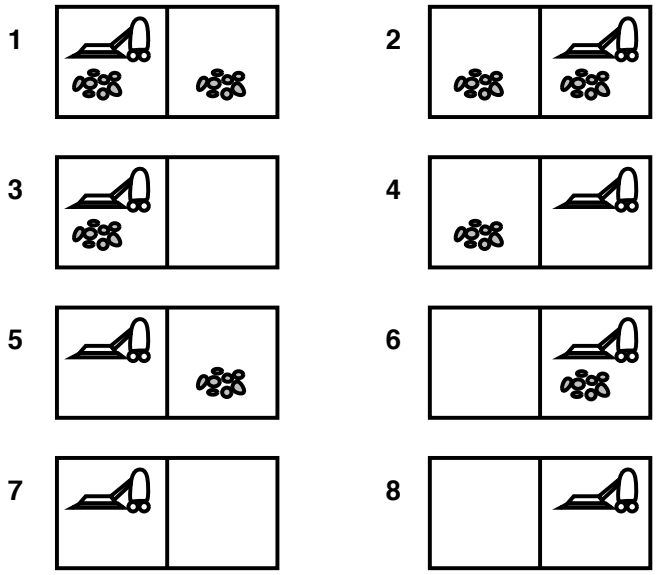
Suck action works as follows:

♢ On a dirty square, it cleans the square and sometimes the other square too

♢ On a clean square, it may dump dirt.

If we start for instance in State 1, there is no single sequence of actions that solves the problem.

# The erratic vacuum world

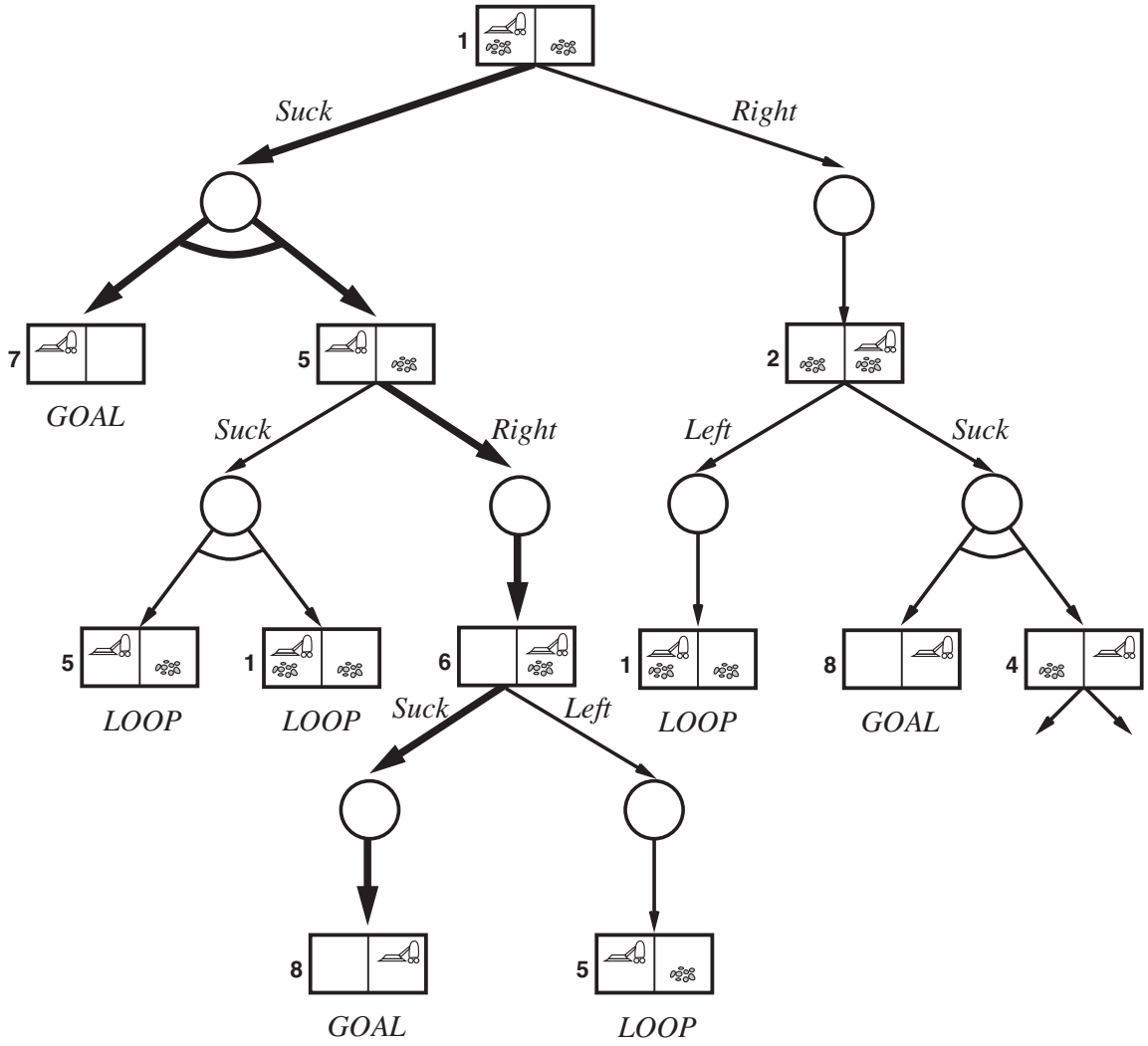



♢ Result of an action is replaced by possible outcomes

♢ Solution becomes a contingency plan (if ... then ...)

Starting at State 1, Suck action may take the agent to State 5 or 7.

# The erratic vacuum world - AND-OR Search Tree



# The erratic vacuum world - AND-OR Search Tree

AND-OR Search Trees:

♢ OR node: the agent chooses among its available actions (Left, Right, Suck)

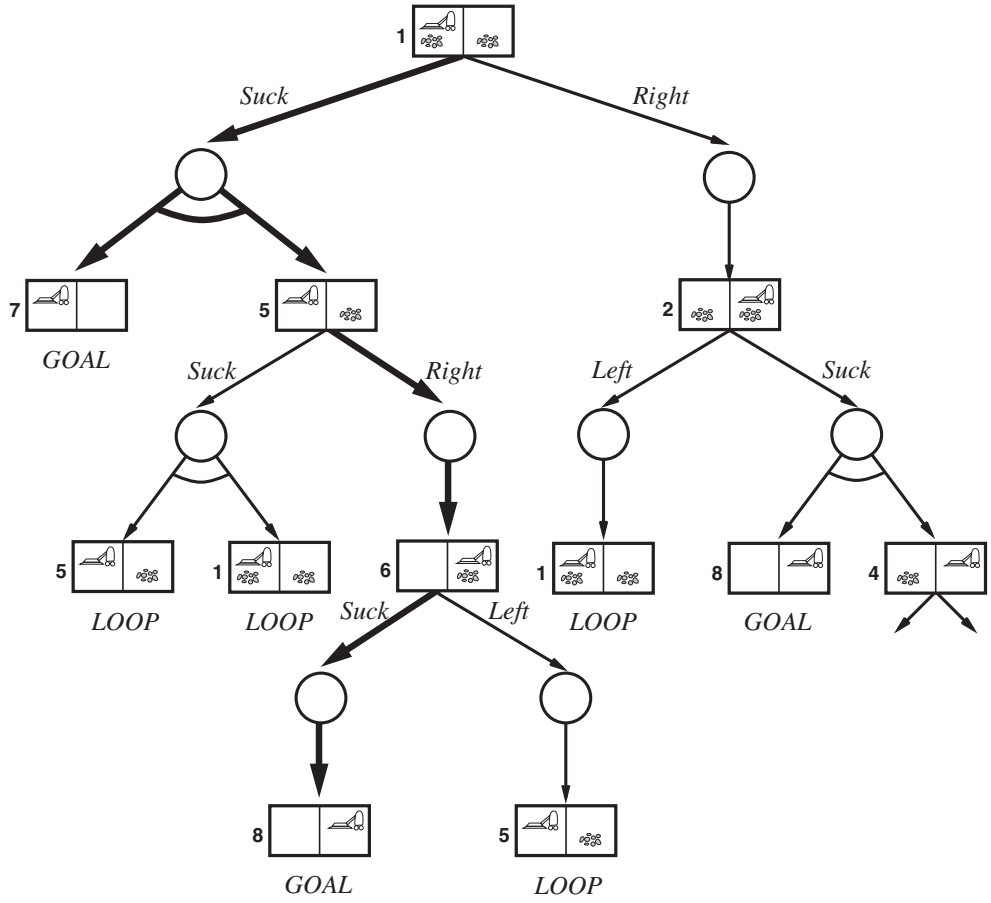
♢ AND node: the environment's choice of outcome are given

# The erratic vacuum world - AND-OR Search Tree

Solution:

[Suck,
if State = 7 then []
else [Right, Suck]
]

Solution becomes a **tree** rather than a sequence.



# The erratic vacuum world

A solution for an AND-OR search problem is a subtree that:

1) has a goal node at every leaf

2) specifies an action at each of its OR nodes

3) includes every outcome branch at each of its AND nodes

♢ The solution is shown in **bold** in previous figure.

♢ Solution may be found using a modified DFS (Fig. 4.11) or BFS or Best First Search.

### 4.3.2 AND–OR search trees

The next question is how to find contingent solutions to nondeterministic problems. As in Chapter 3, 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. We call these nodes **OR nodes**. In the vacuum world, for example, at an OR node the agent chooses *Left or Right or Suck*. In a nondeterministic environment, branching is also introduced by the *environment's* choice of outcome for each action. We call these nodes **AND nodes**. For example, the *Suck* action in state 1 leads to a state in the set $\{5, 7\}$, so the agent would need to find a plan for state 5 *and* for state 7. These two kinds of nodes alternate, leading to an AND–OR **tree** as illustrated in Figure 4.10.

OR NODE

AND NODE

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; it corresponds to the plan given in Equation (4.3). (The plan uses if–then–else notation to handle the AND branches, but when there are more than two branches at a node, it might be better to use a **case**

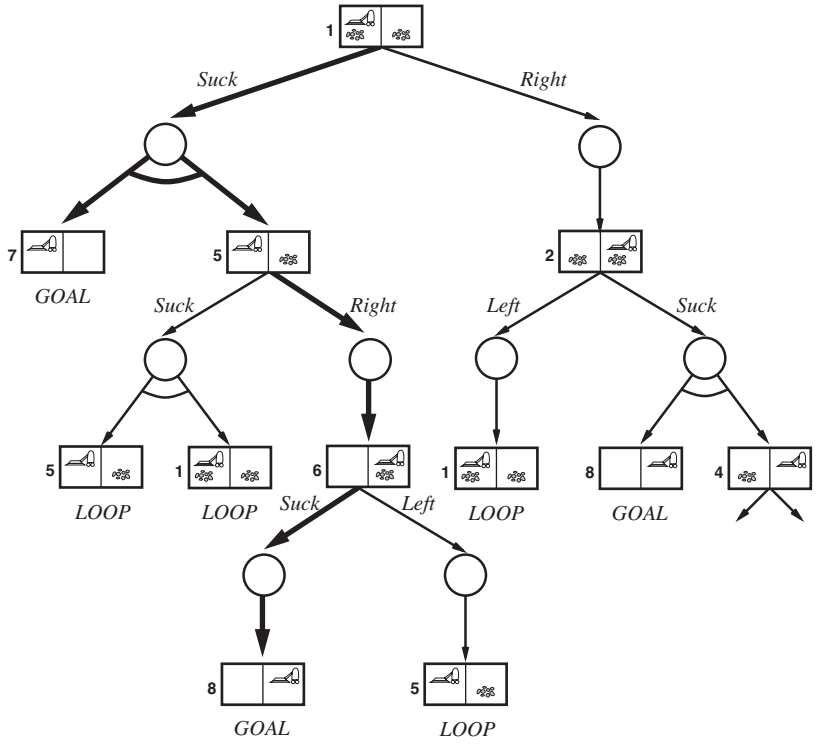
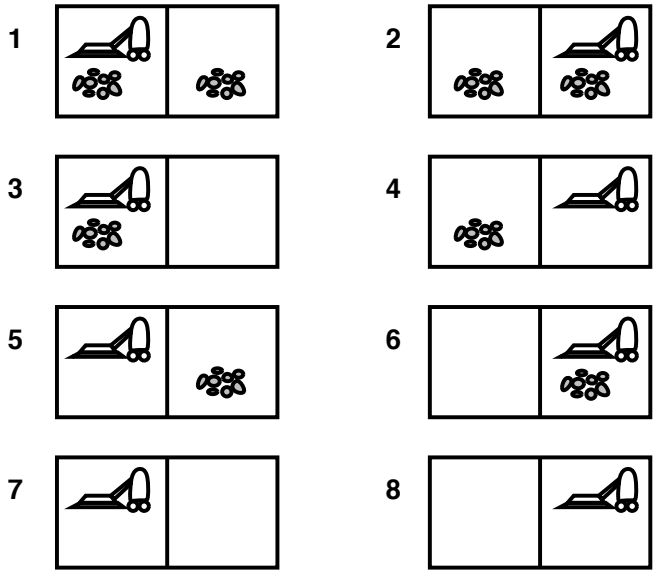



**Figure 4.10** The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold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

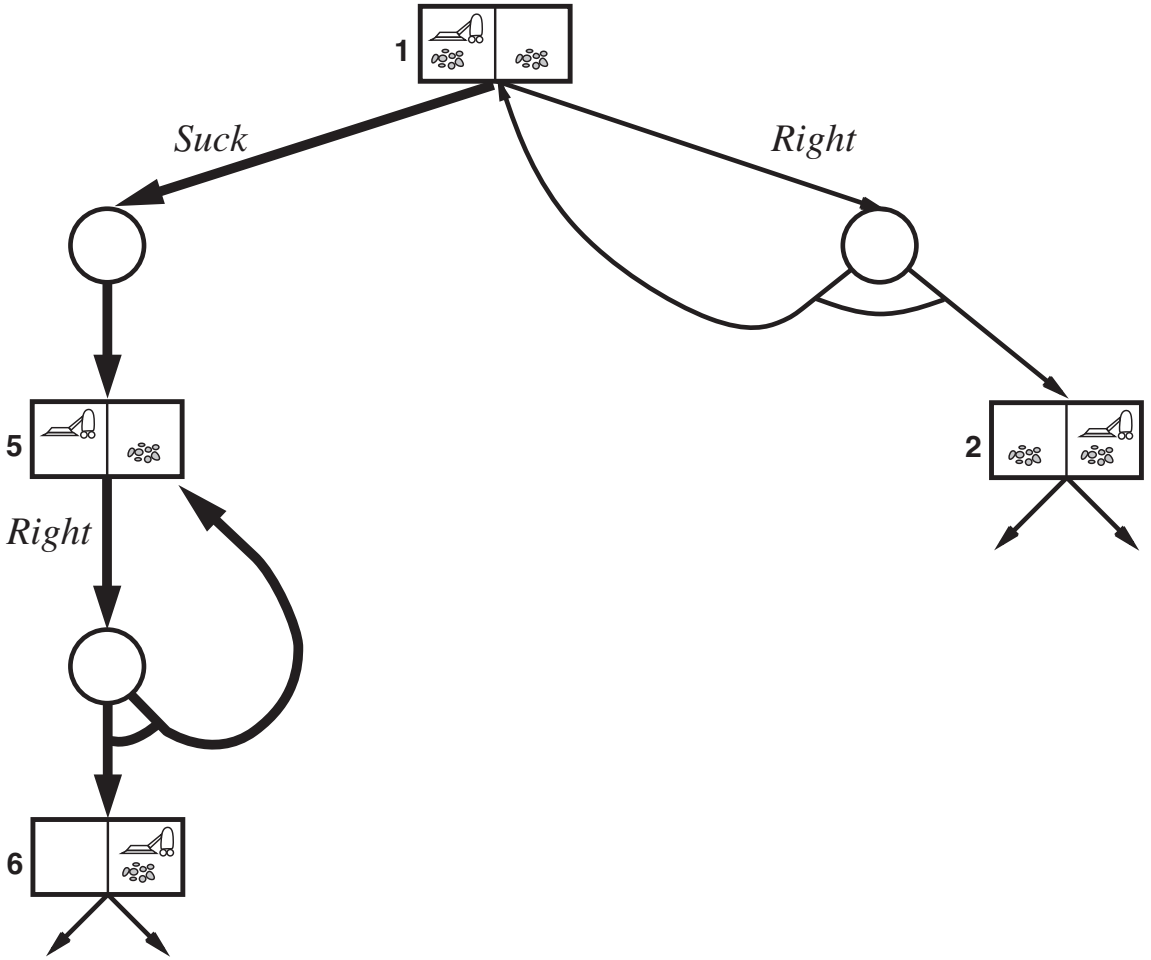
# Another non-deterministic world: Slippery Vacuum World

Consider a slippery vacuum world which is identical to the original (non-erratic) vacuum world except that movement actions may sometimes fail, leaving the agent in its spot.

E.g. Moving Right in State 1 leads to 1 or 2



# Slippery Vacuum World - Solution



# Slippery Vacuum World

♢ There is no longer any acyclic solution from state 1.

♢ There is however a **cyclic** solution which is to keep trying Right until it works.

[Suck, While State = 5 do Right]

An agent executing such a solution will eventually reach the goal state provided that each outcome of a non-deterministic action eventually occurs.

- ♢ rolling a dice vs
- ♢ having the wrong key card (will never succeed)

# Searching with Partial Observations

AIMA V3 – Chp. 4 Section 4.4

# Searching with Partial Observations

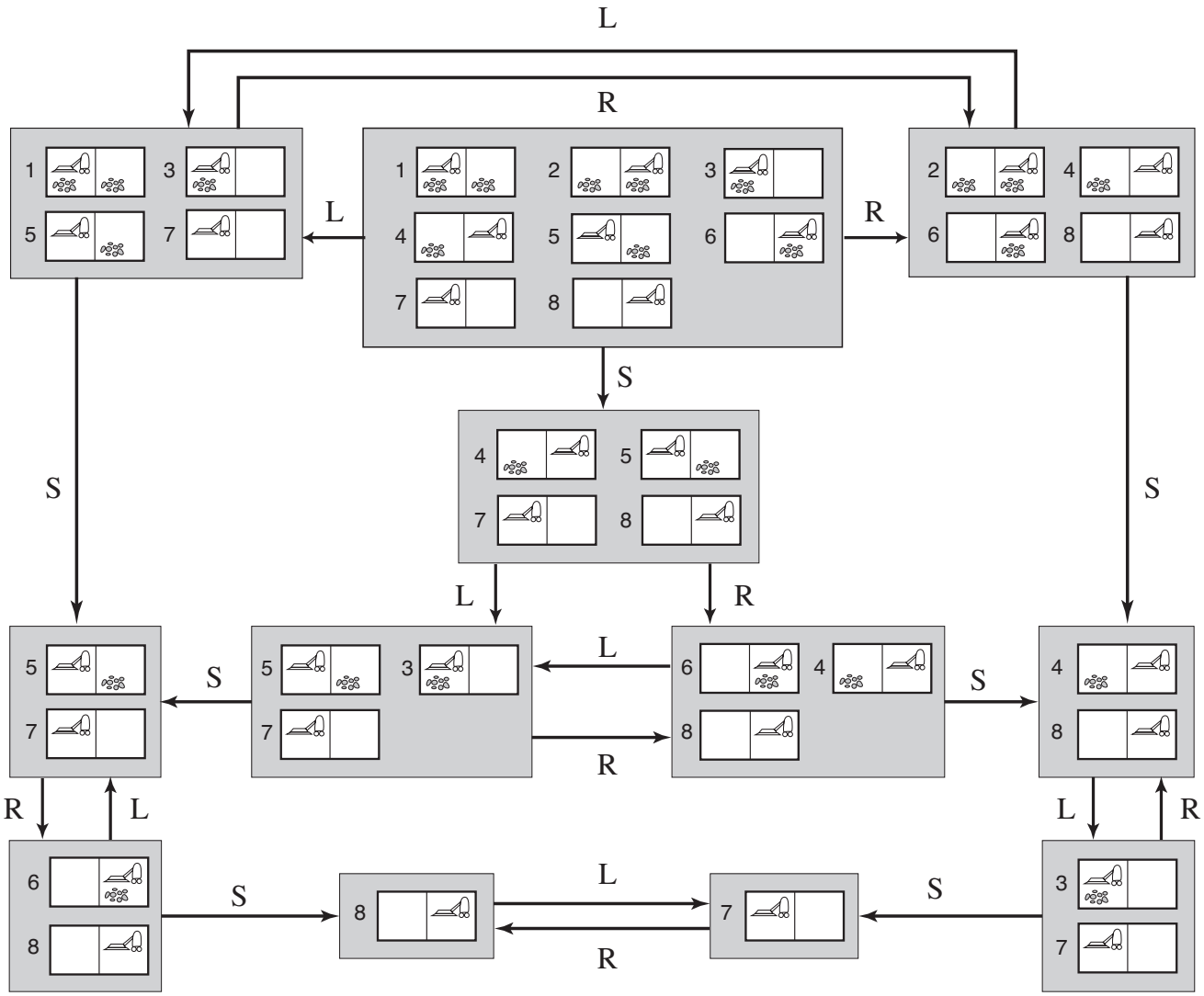
Agent's percepts do not suffice to pin down the exact state.

♢ No observation (sensorless problem):
no sensor cost

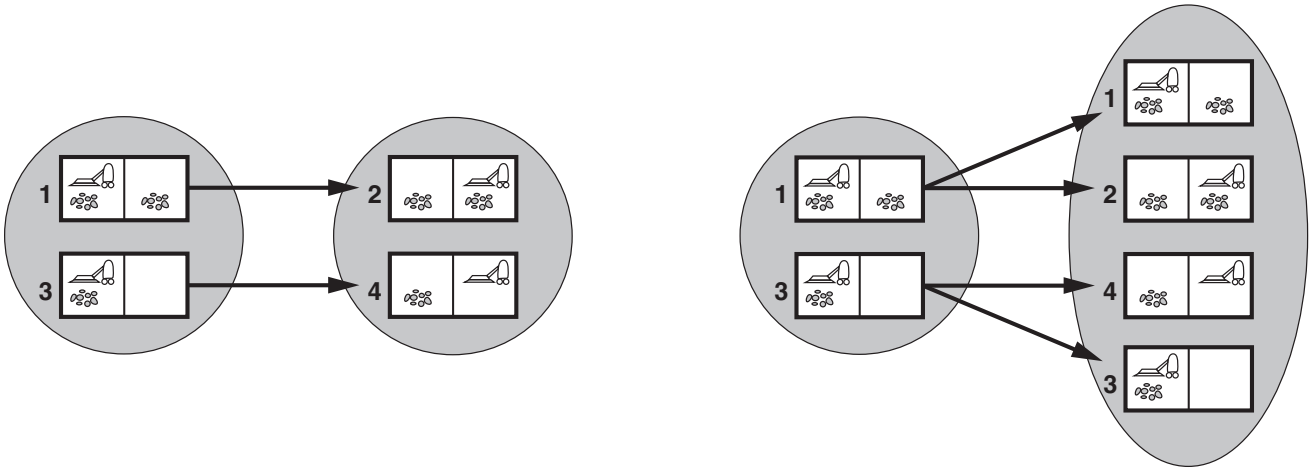
♢ Partial observation:
E.g. only a local dirt sensor which does not know about the dirt in the other room
[A, Dirty] may come from S1 or S3.

⇒ Belief states: Represent the agent's current belief about the possible physical states it might be in.

# Vacuum world with No Observation



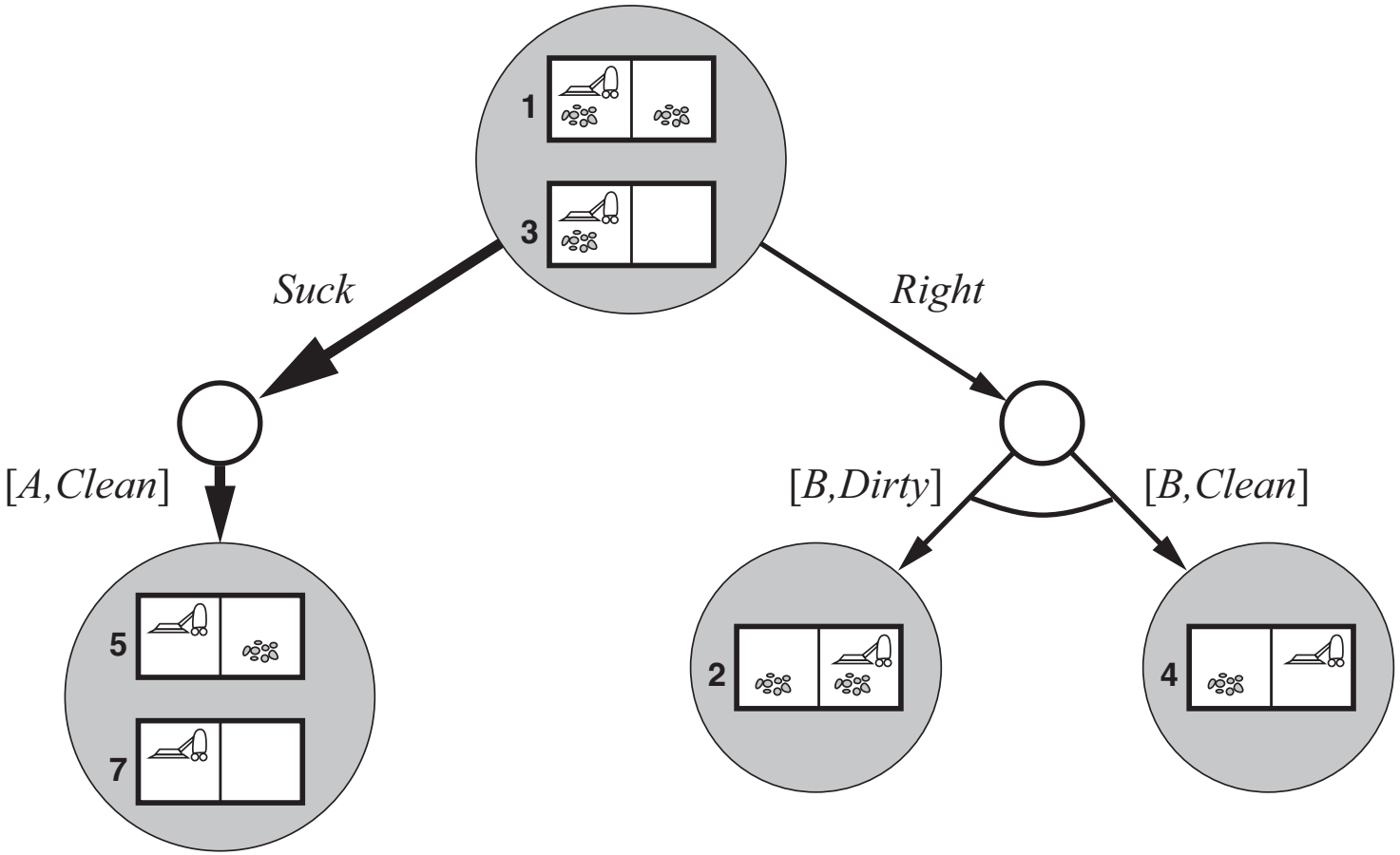
# Vacuum world with Partial Observation




(a) (b)

a) Deterministic world b) Slippery world

# Vacuum world with Partial Observation



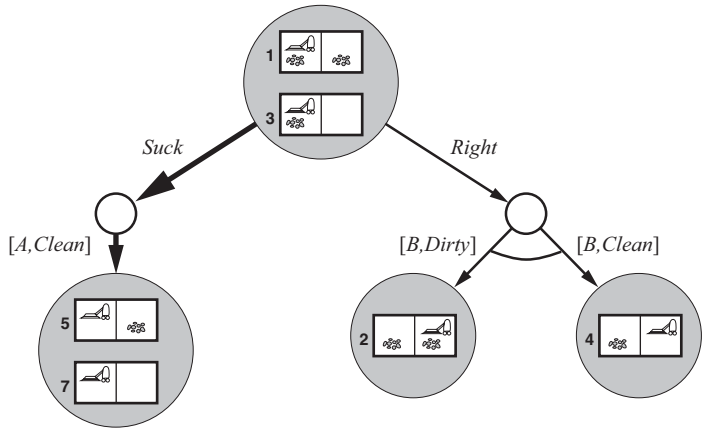



**Figure 4.16** The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first step of the solution.

such a formulation, the AND–OR search algorithm of Figure 4.11 can be applied directly to derive a solution. Figure 4.16 shows part of the search tree for the local-sensing vacuum world, assuming an initial percept $[A, Dirty]$. The solution is the conditional plan

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

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

As in the case of standard search algorithms applied to sensorless problems, the AND–OR search algorithm treats belief states as black boxes, just like any other states. One can improve on this by checking for previously generated belief states that are subsets or supersets of the current state, just as for sensorless problems. One can also derive incremental search algorithms, analogous to those described for sensorless problems, that provide substantial speedups over the black-box approach.

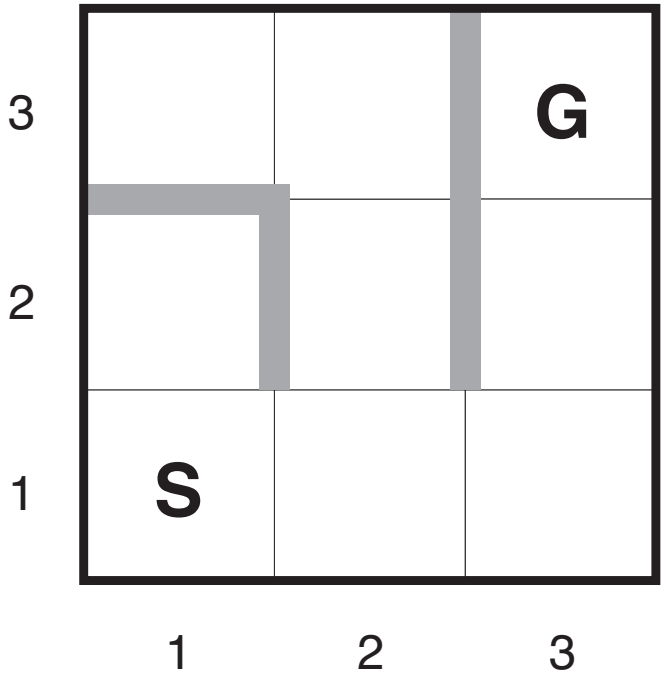
### 4.4.4 An agent for partially observable environments

The design of a problem-solving agent for partially observable environments is quite similar to the simple problem-solving agent 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. This process resembles the prediction–observation–update process in Equation (4.5) but is actually simpler because the percept is given by the environment rather than calculated by the

# Online Search

AIMA V3 – Chp. 4 Section 4.5

# Chp 4: Online Search



# Chp 4, Section 4.5: Online Search Algorithms

♢ Offline search: complete solution before setting foot in the real world and then executes the solution.

♢ Online search: **interleaves** computation and execution (e.g. factory robot, rescue robot,...) - exploration

# Chp 4: Online Search

We assume a deterministic environment, where the agent knows only the following:

- ♢ Actions(s) - list of actions available in state s
- ♢ Step cost function c(s,a,s') - this cannot be used until the agent knows that s' is the outcome
- ♢ Goal-Test(s)

After each action, the agent receives a percept telling what state it has reached; from this information, it can **augment its map of the environment**.

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

# Chp 4: Online Search

♢ Good in dynamic environments

♢ Good in non-deterministic environments: reduce computation of all contingencies

♢ Necessary for exploration problems, where the environment (states) and possibly the actions are not known to the agent

Examples: robot exploring Mars, baby exploring the world...

# Chp 4: Online Search

Online agents;

♢ can typically recognize a previously visited state

♢ minimizes the cost of reaching the goal (e.g. number of steps)

♢ may have a heuristic stating how close to the goal

# Chp 4: Online Search

♢ **Competitive ratio**: ratio of the cost of online solution to that of offline one for the same problem.

We would like to minimize the competitive ratio, but in cases involving infinite path costs or irreversible states, the competitive ratio will be infinite.

♢ *Safely explorable* spaces

Since no agent can avoid dead-ends in all state-spaces, let's assume that we have safely explorable spaces.

# Online Search Algorithms - DFS

♢ A* is offline: expands a node in one part of the space, then a node in another part of the space...

♢ The online search algorithm can expand only those states that it physically occupies..

$\Rightarrow$ Search algorithms that expand in local order are more suitable:

Depth-first search works! But the agent needs to physically do the backtracking (keep track of predecessors)? (see algorithm in 4.21)

# Online Search Algorithms - Hill Climbing

Already an online search! However, basic version is not very useful due to local maxima.

How to do random restarts?

# Online Search Algorithms

♢ **Random walk**: select at random one of the available actions (possibly preferring actions not tried before)

If the space is finite, a random walk will eventually find a goal or complete its exploration.

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table indexed by state and action, initially empty
              untried, a table that lists, for each state, the actions not yet tried
              unbacktracked, a table that lists, for each state, the backtracks not yet tried
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in untried) then untried[s'] ← ACTIONS(s')
  if s is not null then
      result[s, a] ← s'
      add s to the front of unbacktracked[s']
  if untried[s'] is empty then
      if unbacktracked[s'] is empty then return stop
      else a ← an action b such that result[s', b] = POP(unbacktracked[s'])
  else a ← POP(untried[s'])
  s ← s'
  return a
```

**Figure 4.21** An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be "undone" by some other action.

lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

We recommend that the reader trace through the progress of ONLINE-DFS-AGENT when applied to the maze given in Figure 4.19. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

### 4.5.3 Online local search

Like depth-first search, **hill-climbing search** has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is *already* an online search algorithm! Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

RANDOM WALK

Instead of random restarts, one might consider using a **random walk** to explore the environment. A random walk simply selects at random one of the available actions from the

# Online Search Algorithms- LRTA*:

Augmenting hill-climbing with memory (rather than randomness): store the **current best estimate $H(s)$ of reaching the goal from each state that has been visited.**

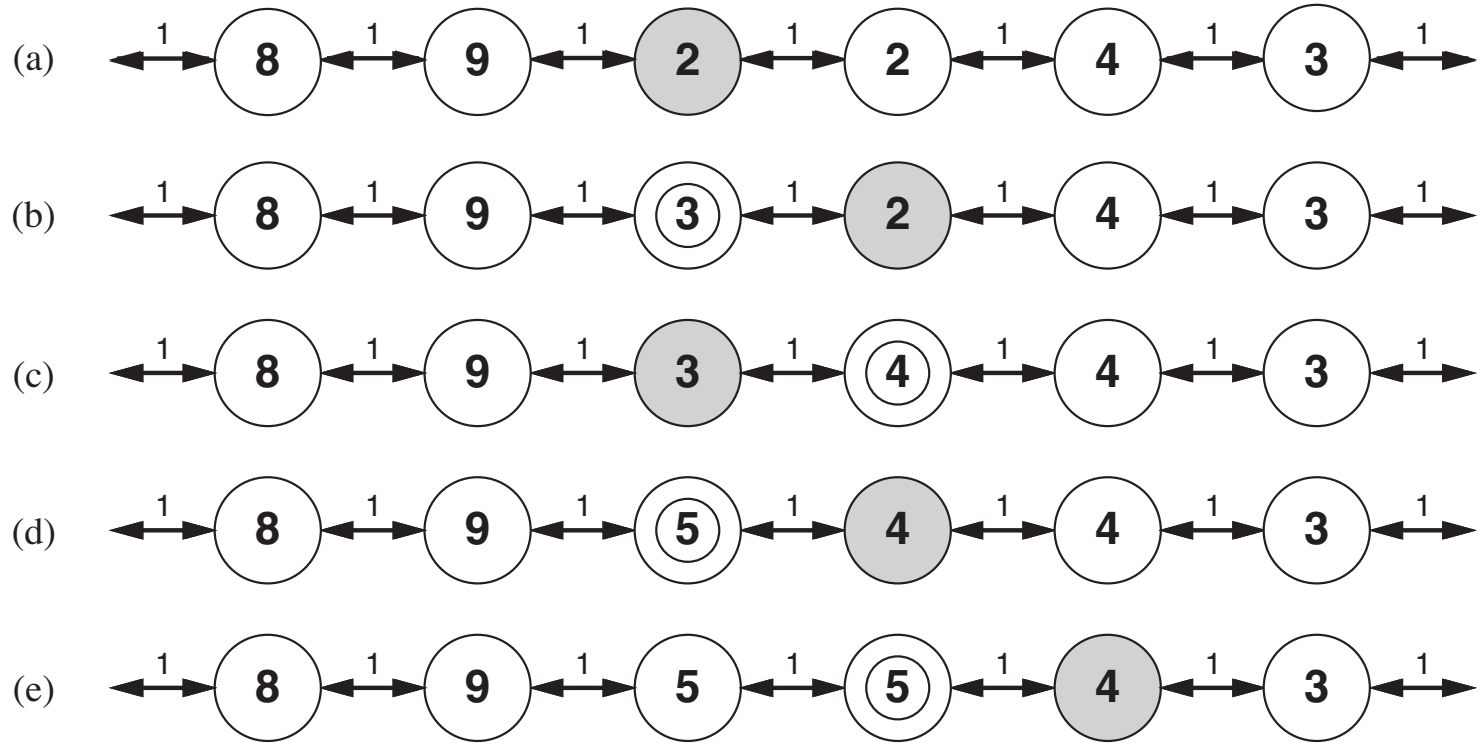
♢ $H(s) = h(s)$ initially, but is updated as the agent gains experience.

♢ Agent updates the H() estimate for the state it has just left.

$$H(s) = c(s, s') + H(s')$$

Assumes that h(s') is (more) reliable, improvement comes from the actual cost being used.

# Online Search Algorithms - LRTA*




Notice that the updated H values are more correct (9-2-2-4 was not meaningful in a 1D environment)

# Online Search Algorithms - LRTA*

♢ An LRTA* gent is guaranteed to find a goal in any finite, safely explorable environment

$O(n^2)$ steps in the worst-case in an environment with $n$ states.

♢ Unlike $A*$, it is not complete in infinite spaces

# Online Search Algorithms - Learning

Learning is important and already used in the previous slides in two ways:

♢ Building a map - recording results of actions taken from each state

♢ Updating heuristic estimates (as in LRTA*)

♢ But there can be more to learn. E.g. agent does not know that UP action is the reverse of Down...

Logical rules, as we will see in the following chapters, will be useful in specifying and infering how actions may be related or what they do...