CS 5/7320
Artificial Intelligence

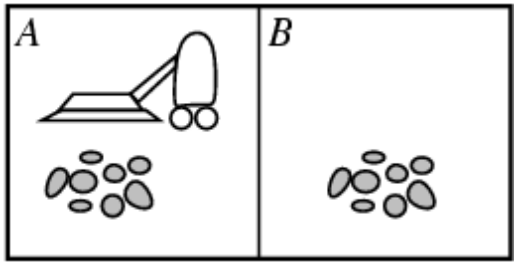
Search with Uncertainty

AIMA Chapters 4.3-4.5

Slides by Michael Hahsler
with figures from the AIMA textbook

Online Material

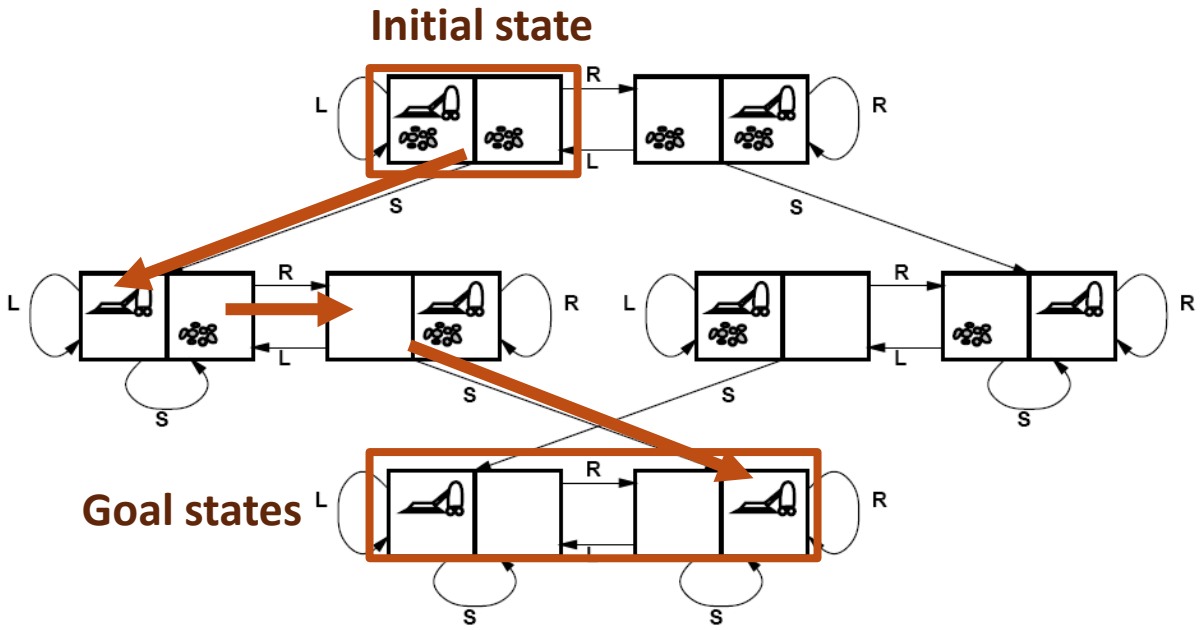
Remember: Solving Search Problems under Certainty



No Uncertainty

- Deterministic actions with known transition model $Result(s_1, a) = s_4$
- Full observability (we know where everything is while planning)

State space: A state completely describes the environment and agent



Solution of the planning phase is a **sequence of actions** also called a **plan** that can be blindly followed: **[Suck, Right, Suck]**

Consequence of Uncertainty

1. The agent may not know in what state it and the environment exactly is in.

   It needs to keep track of all the states it could be in. This set is called the ***believe state.***

2. The solution is typically not a fixed precomputed plan (sequence of actions), but a

   conditional plan (also called strategy or policy)

   that depends on percepts.

Types of uncertainty in the environment*

Nondeterministic Actions:
Outcome of an action in a state is uncertain.

No observations:
Sensorless problem

Partially observable environments:
The agent does not know in what state the environment is.

Exploration:
Unknown environments and Online search

* we will quantify uncertainty with probabilities later.


Nondeterministic Actions

Definition: Nondeterministic Actions

Outcome of actions in the environment is nondeterministic = **transition model need to describe uncertainty.**

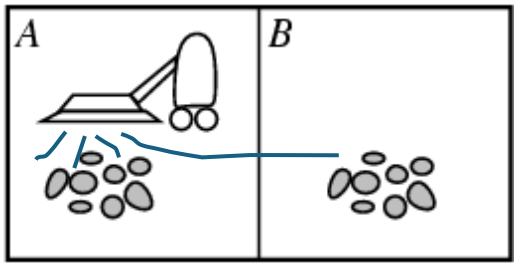
Example transition:

Note the ‘s’ here

$$Results(s_1, a) = \{s_2, s_4, s_5\}$$

i.e., action a in s_1 can lead to one of several states.

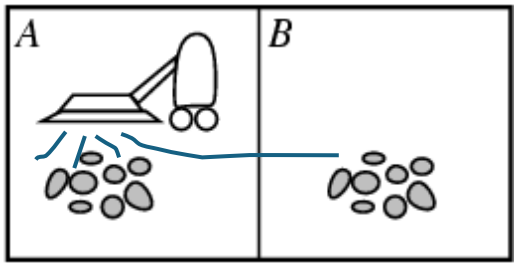
Example:
Erratic Vacuum World

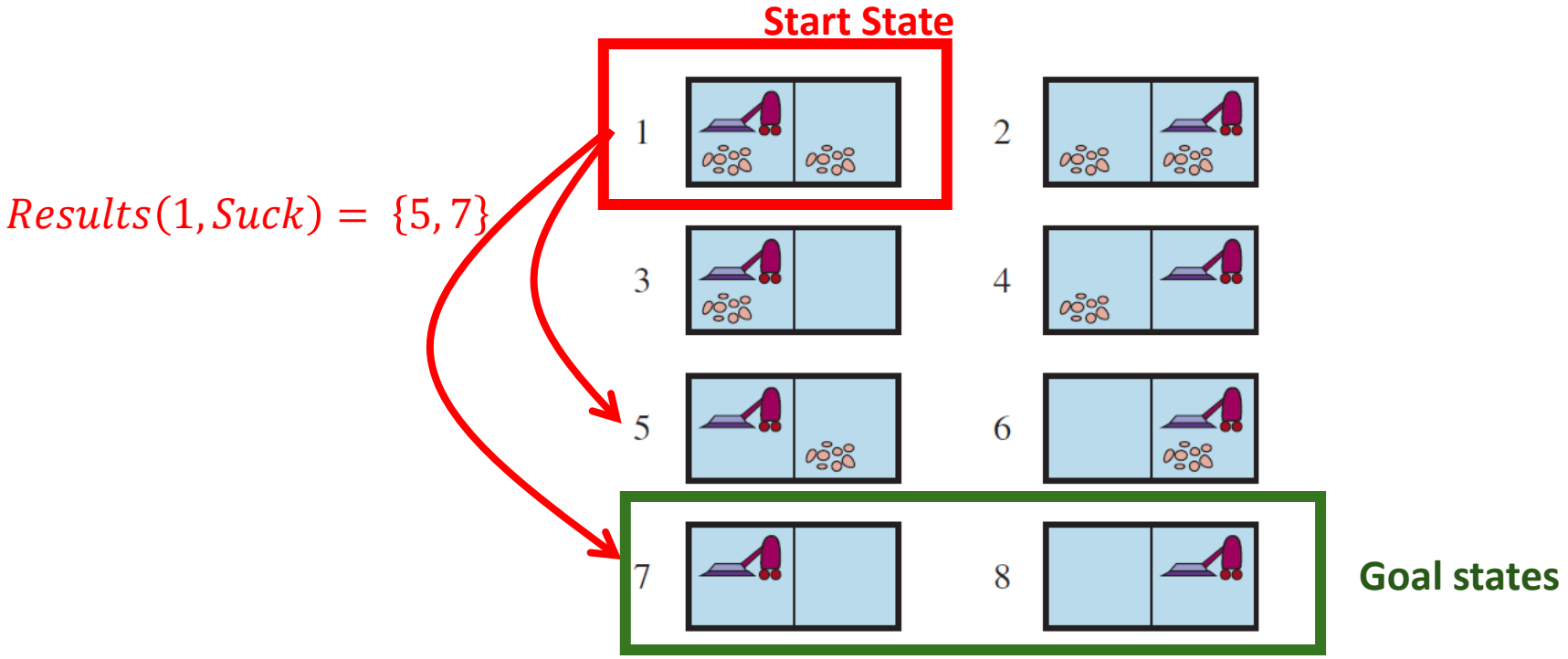


Regular deterministic vacuum world, but the action '**suck**' is more powerful and **nondeterministic**:

a) **On a dirty square**: cleans the square and sometimes cleans dirt on adjacent squares as well.
b) **On a clean square**: sometimes deposits some dirt on the square.

Example:
Erratic Vacuum World



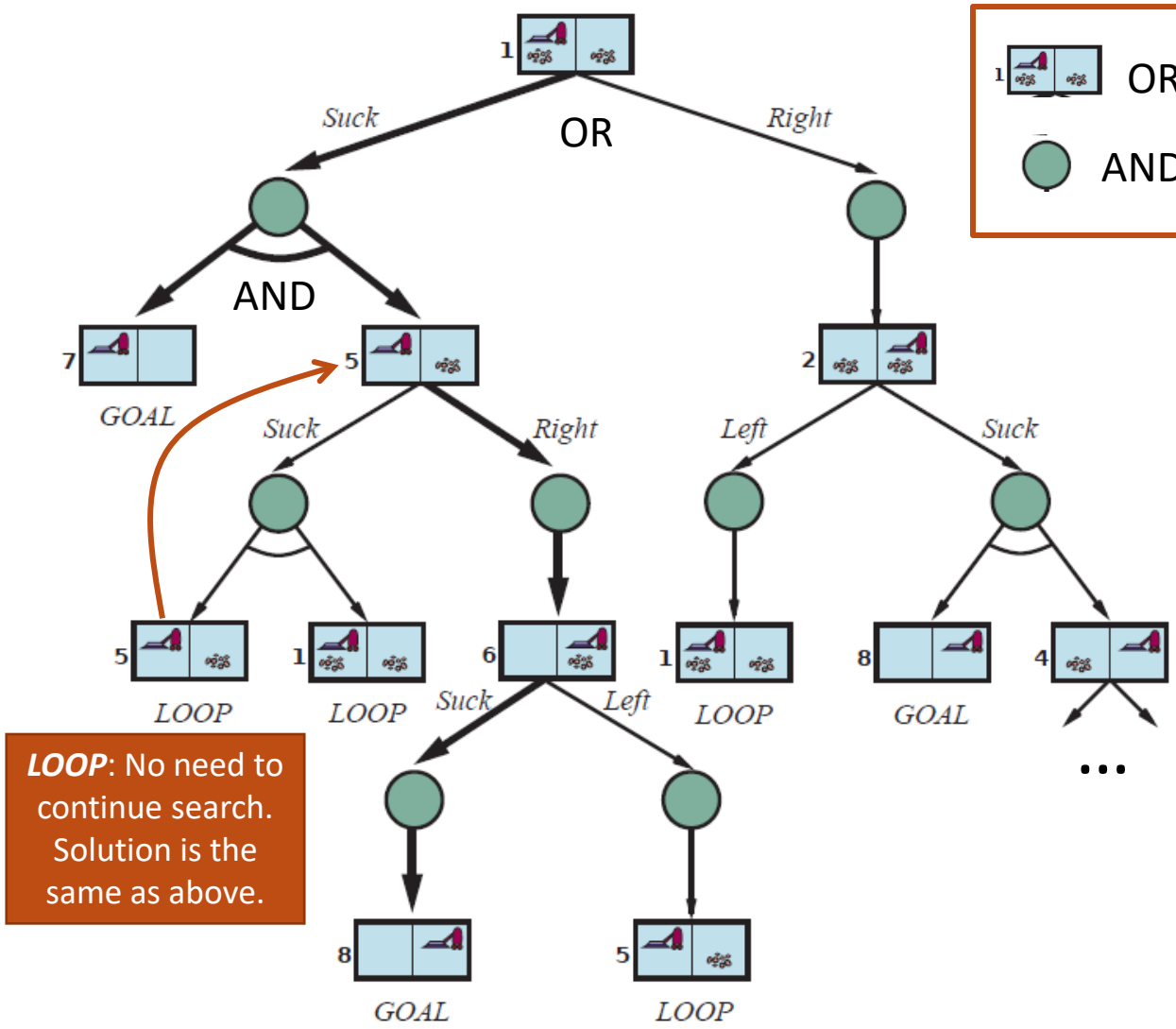


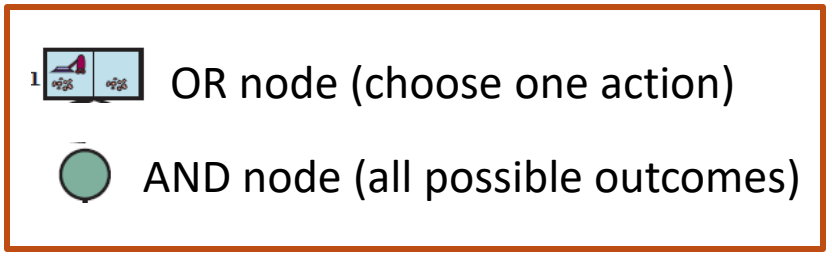
Suck can also lead to two different states!

We need a conditional plan

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

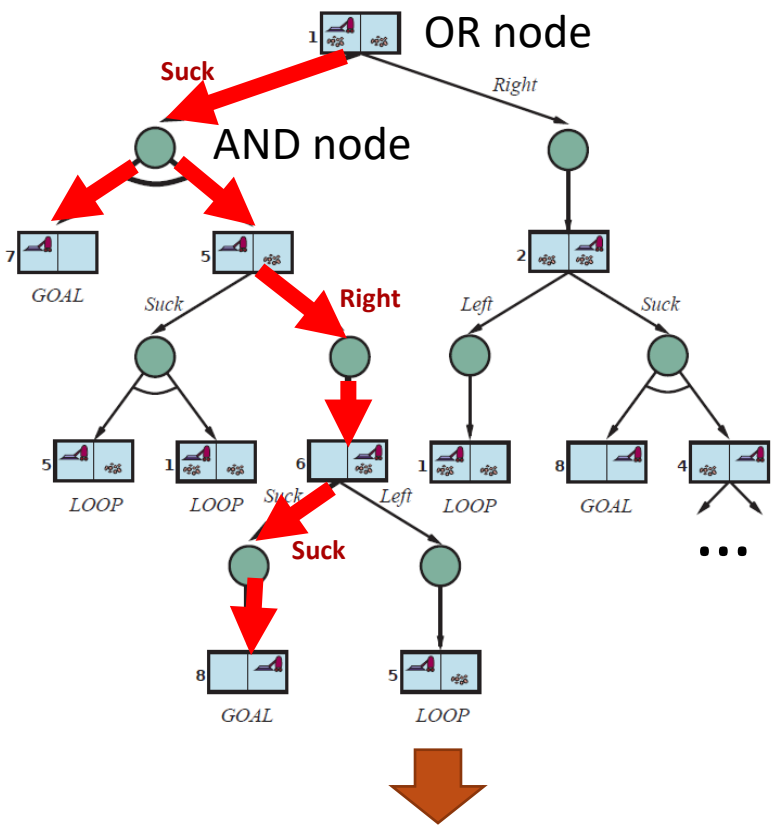
Transition Model as an AND-OR Search Tree





LOOP: No need to continue search. Solution is the same as above.

Search the AND-OR Tree



- Descend the tree depth-first by trying an action in each OR node and considering all resulting states of the AND nodes.
- Loop nodes are ignored.
- Remove branches (actions) if we cannot find a subtree below that only leads to goal nodes. (see failure in the code on the next slide).
- Stop when we find a subtree that only has goal states in all leaf nodes.
- Construct the conditional plan that represents the subtree starting at the root node.

Conditional Plan:
[Suck, **if** State = 5 **then** [Right, Suck] **else** []]

AND-OR Recursive DFS Algorithm

= nested If-then-else statements

path is only maintained for cycle checking!

```
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(path) then return failure                    // don't follow loops using path.
   for each action in problem.ACTIONS(state) do             // try all possible actions
       plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
       if plan ≠ failure then return [action] + plan]
   return failure                                           // fail means we found no action that leads to
                                                            // a goal-only subtree
function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
   for each s_i in states do                                // try all possible outcomes, none can fail!
       plan_i ← OR-SEARCH(problem, s_i, path)
       if plan_i = failure then return failure              // fail if we find any non-goal subtree
   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]
```

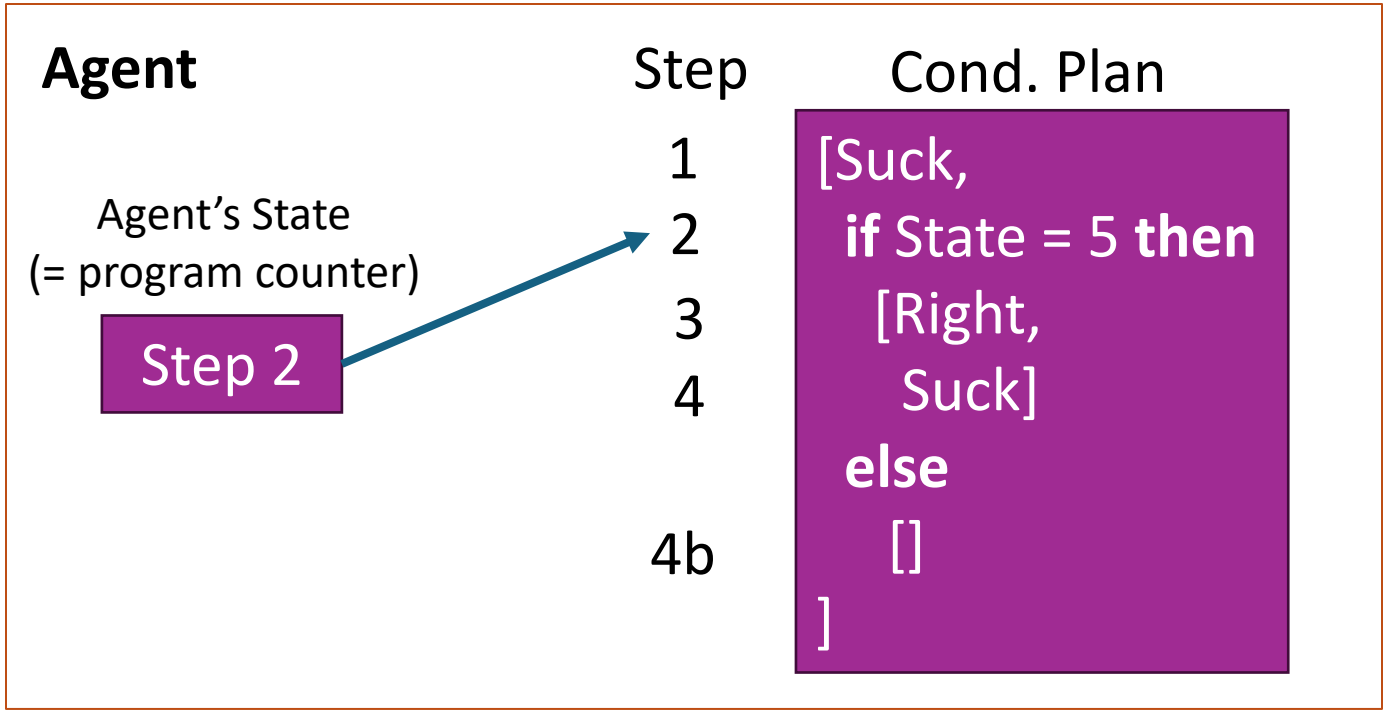
Notes:

- The DFS search tree is implicitly created using the call stack (recursive algorithm).
- DFS is **not optimal**! BFS and A* search can be used to find better solutions (e.g., smallest subtree).

An Agent using the Conditional Plans

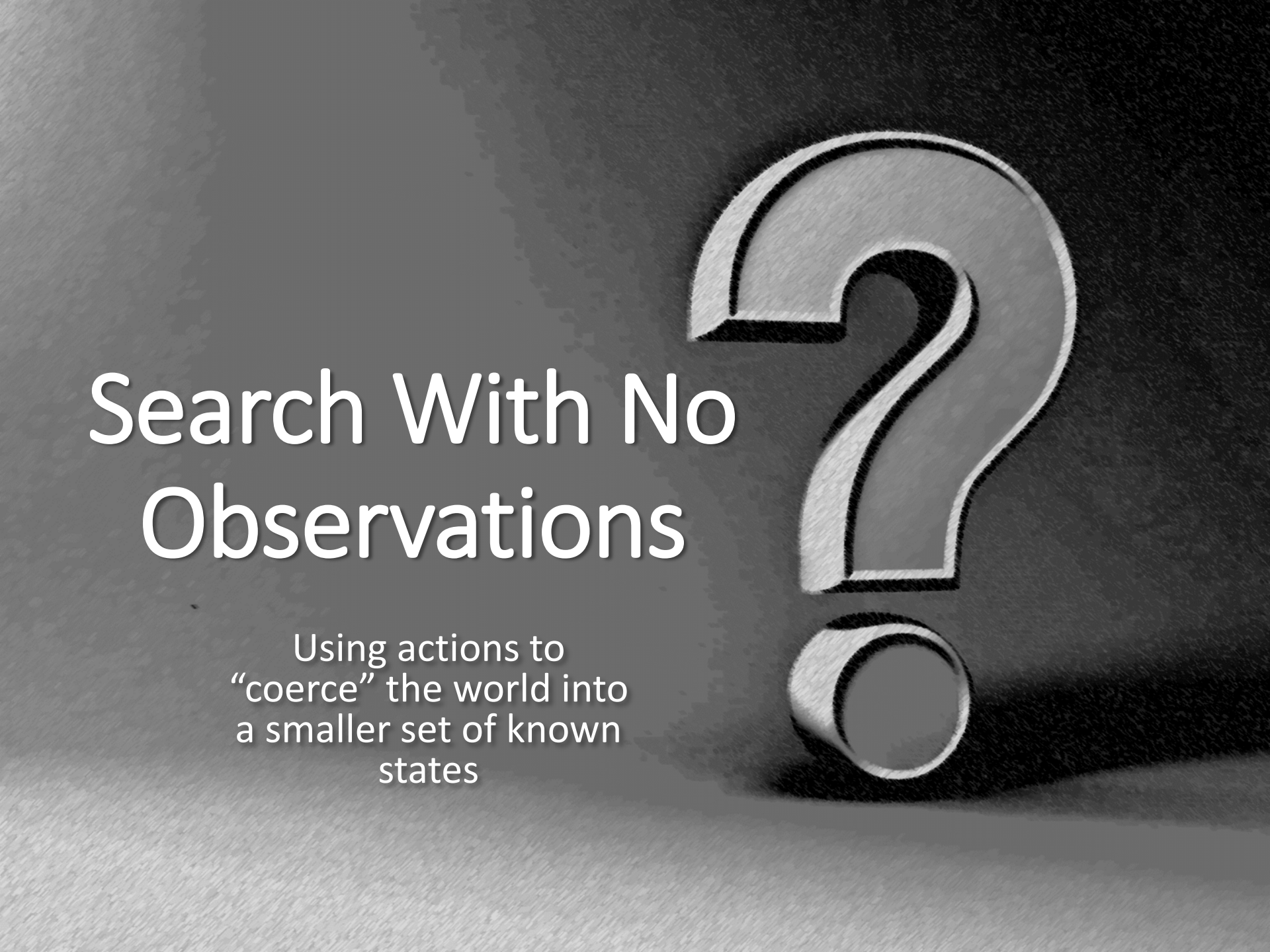
- Planning uses search to find a conditional plan that leads to a goal state.
- The conditional plan can be executed by a **model-based reflex agent** that uses a program counter to execute the plan and **percepts** for the if statements.

Example: After the initial action "suck"



Search With No Observations

Using actions to "coerce" the world into a smaller set of known states



Sensorless Problems

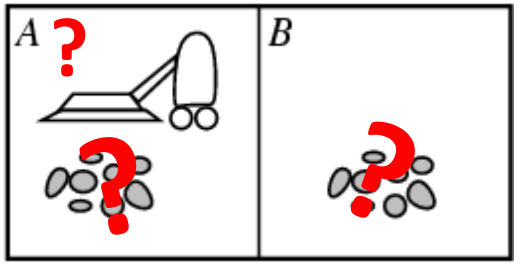
Conformant problem: The agent has no sensors so the environment is not observable.

Why is this useful?

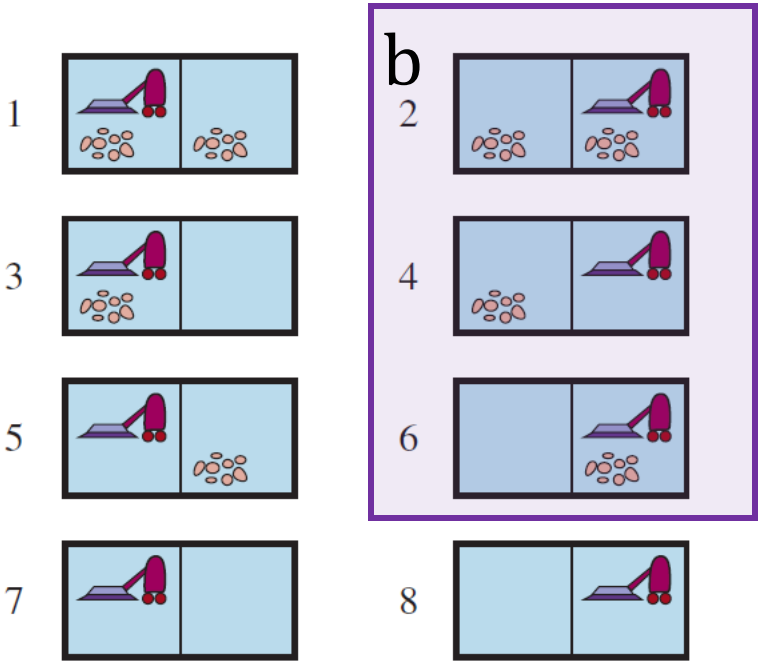
- **Example**: Doctor prescribes a broad-band antibiotic instead of performing time-consuming blood work to find a more targeted antibiotic. This saves time and money.

- **Basic idea**: Find a solution (a plan) that **works (reasonably well) from any state** and then just blindly execute it.

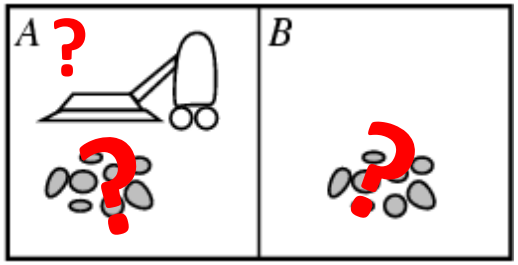
Definition: Belief State



- The agent does not know in which state it is exactly in.
- However, it may know that it is in one of a set of possible states. This set is called a **belief state** of the agent.
- Example: $\mathrm{b} = \{s_2, s_4, s_6\}$

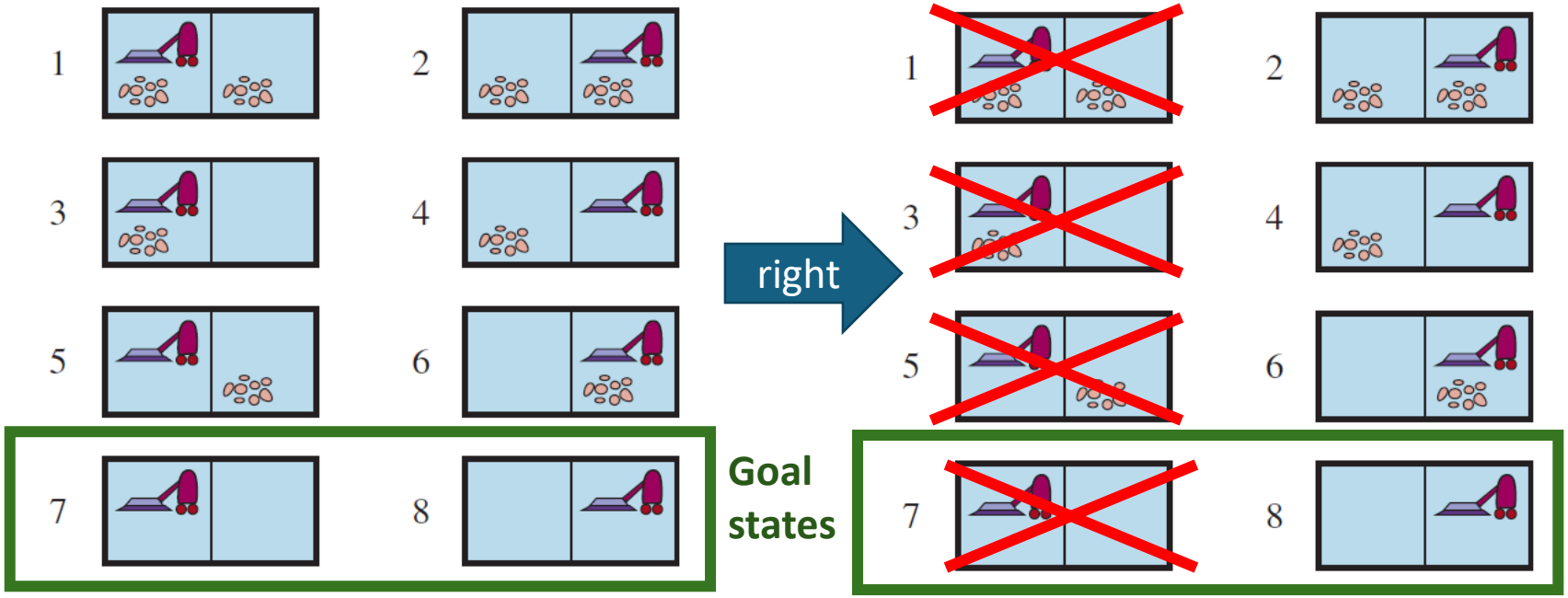


Actions to Coerce the World into States

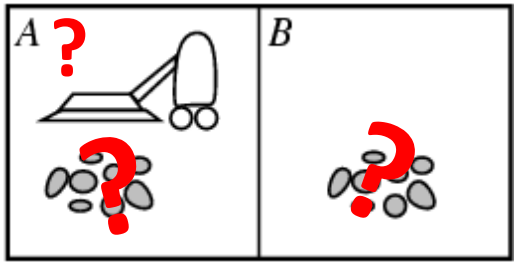


- Actions can reduce the number of possible states.
- **Example**: Deterministic vacuum world. Agent does not know its position and the dirt distribution.

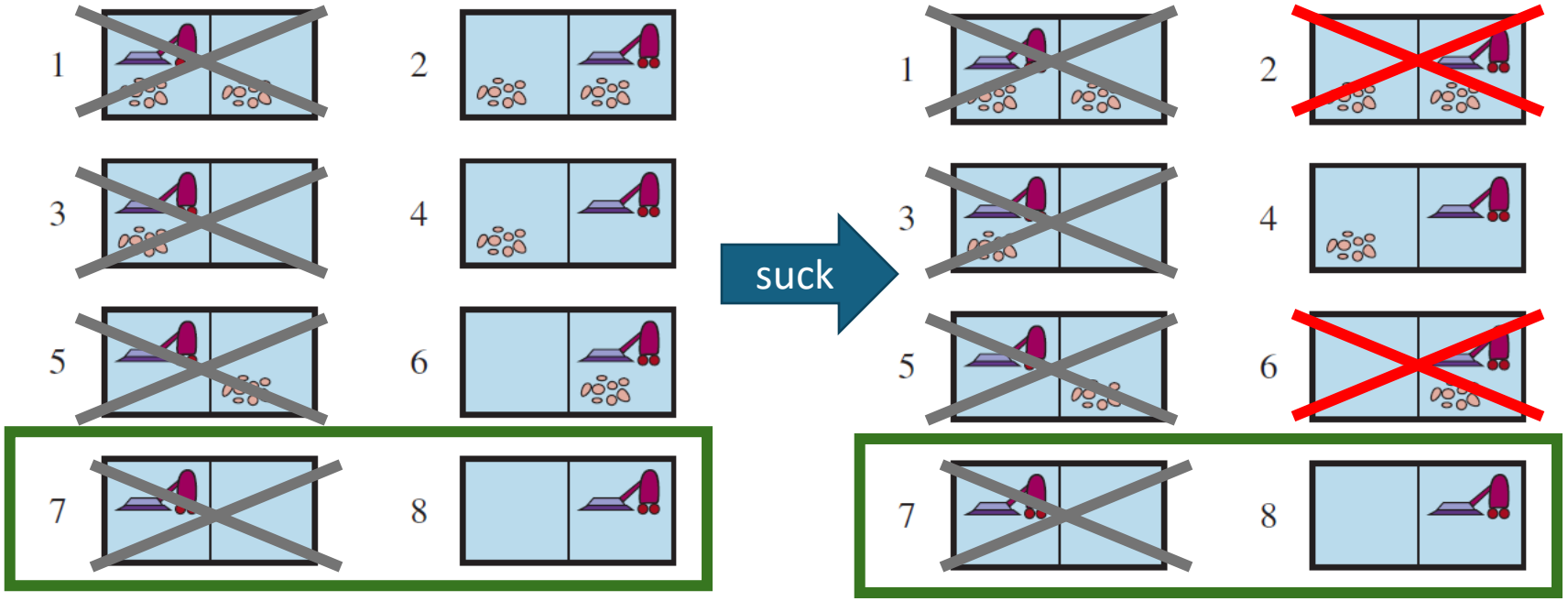
Initial belief state {1,2,3,4,5,6,7,8}



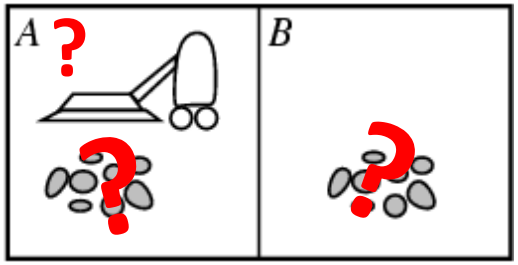
Actions to Coerce the World into States



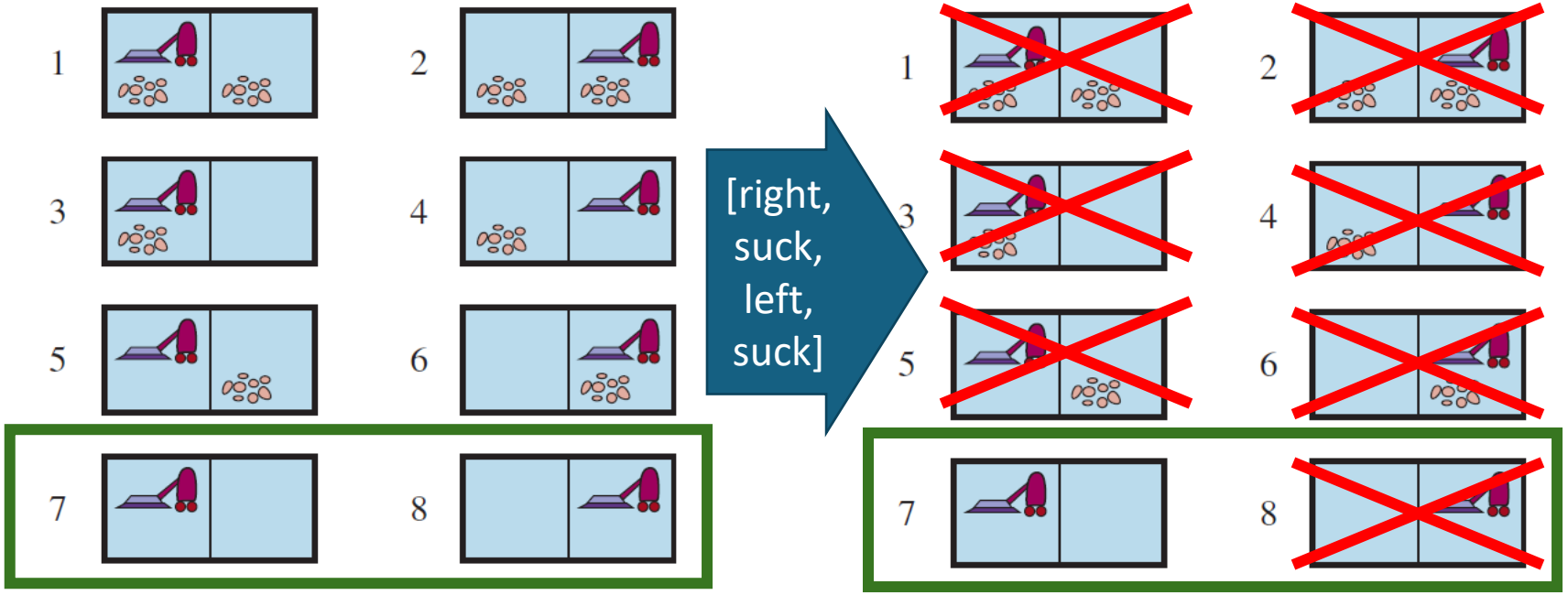
- Actions can reduce the number of possible states.
- **Example**: Deterministic vacuum world. Agent does not know its position and the dirt distribution.



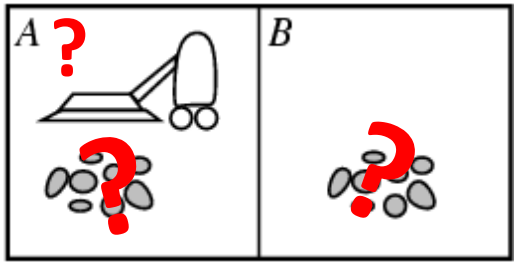
Actions to Coerce the World into States 2

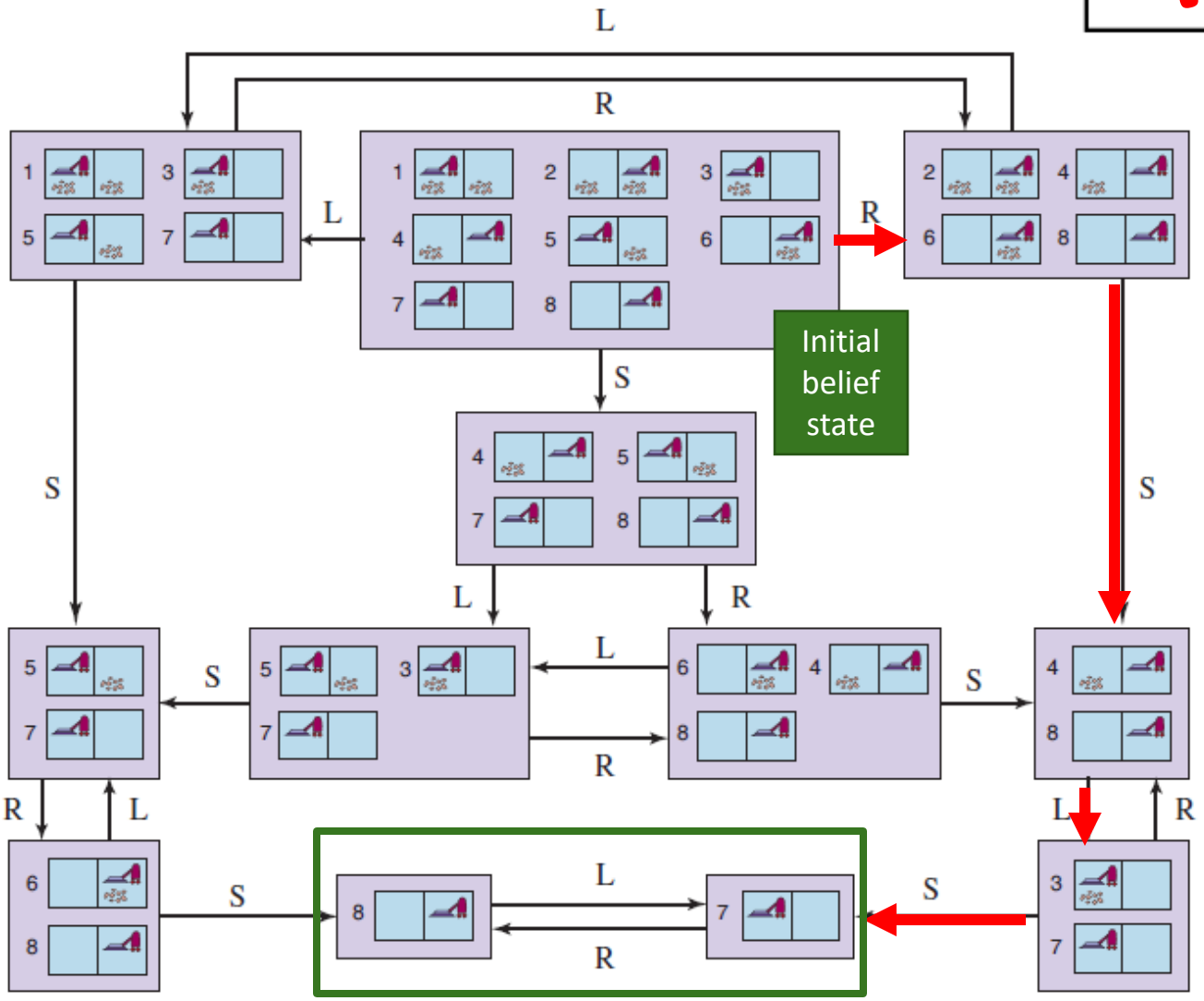


- The action sequence [right, suck, left, suck] coerces the world into the goal state 7. This plan works from any initial state!
- There are no observations so there is no need for a conditional plan.



The Reachable Belief State Space





Size of the belief state space depends on the number of states N:

$$\mathcal{P}_s = 2^N = 2^8 = 256$$

Only a small fraction (12 states) are reachable by actions.

No observations, so we get a solution sequence from an initial belief state:
[Right, Suck, Left, Suck]

Finding a Plan

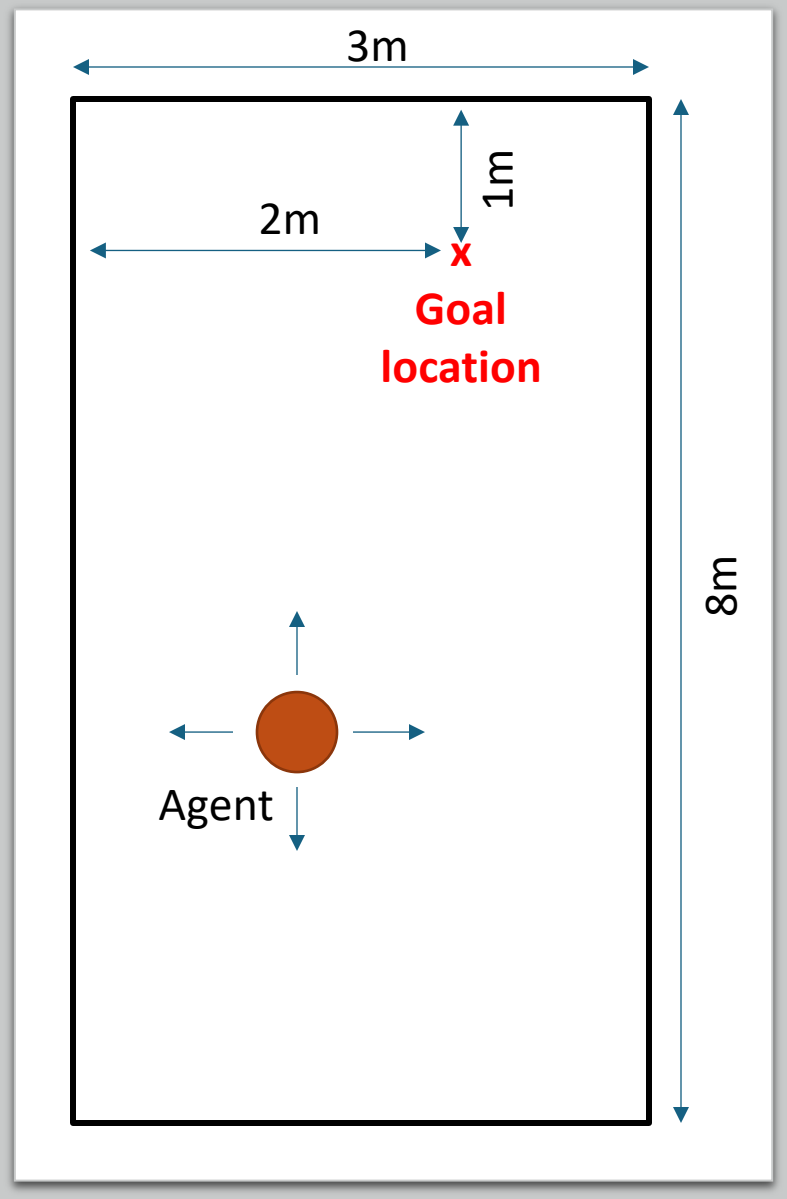
Note: State space size makes this impractical for larger problems!

Formulate as a regular search problem and solve with DFS, BFS or A*:

- **States**: All belief states (=powerset $\mathcal{P}_s$ of the set of N states has size 2^N)
- **Initial state:** Often the belief state containing all states.
- **Actions:** Available actions of a belief state are the union of the possible actions for all the states it contains.
- **Transition model:** $b' = Results(b, a) = \{s' : s' = Result(s, a)\ and\ s \in b\}$
- **Goal test:** Does the belief state only contain goal states?
- **Simplifying property:** If a belief state (e.g., $b_1 = \{1,2,3,4,5\}$) is solvable (i.e., there is a sequence of actions that coerce all states to only goal states), then belief states that are subsets (e.g., $b_2 = \{2,5\}$) are also solved using the same action sequence. This can be used to prune the search tree.

Other approach:

- **Incremental belief-state search**. Generate a solution that works for one state and check if it also works for all other states. If it does not, then modify the solution slightly. This is similar to local search.



Case Study

The agent can move up, down right, and left.

The agent has **no sensors** and does not know its current location.

1. Can you navigate to the goal location? How?

2. What would you need to know about the environment?

3. What type of agent can do this?

Partially Observable Environments

Using Observations to Learn About the State

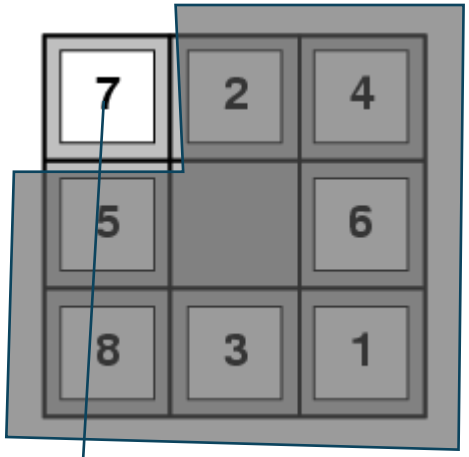
Percepts and Observability

- Many problems cannot be solved efficiently without sensing (e.g., 8-puzzle).
- We need to see at least one square.

Percept function: $Percept(s)$

…s is the state

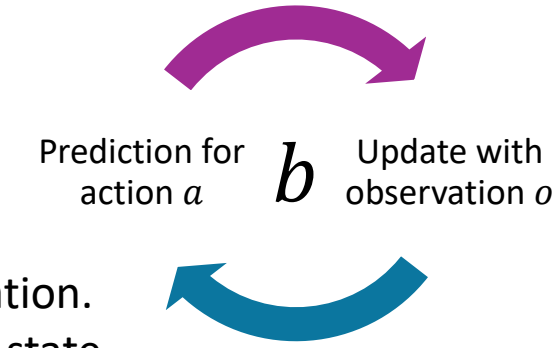
- **Fully observable**: $Percept(s) = s$
- **Sensorless**: $Percept(s) = None$
- **Partially observable**: $Percept(s) = o$
  o is called an observation and tells us something about s



$Percept(s) = Tile7$

Problem: Many states (different order of the hidden tiles) can produce the same observation!

Use Observations to Learn About the State



The agents chooses an action and then receive an observation.

Idea: Observations can be used to learn about the agent's state.

Assume we have a current belief state b (i.e., the set of states we could be in).

1. Prediction for action: Choose an action a and compute a new belief state that results from the action using the transition model.

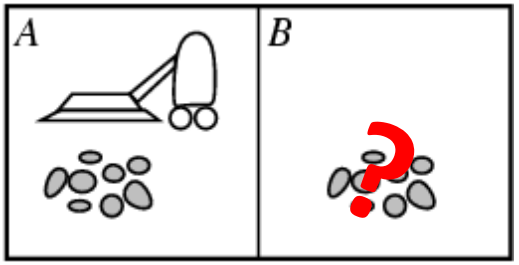
$$\hat{b} = Predict(b, a) = \bigcup_{s \in b} Result(s, a)$$

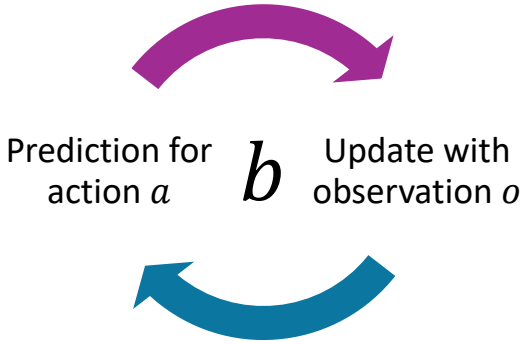
2. Update with observation: You receive an observation o and only keep states that are consistent with the new observation. The filtered belief after observing o is:

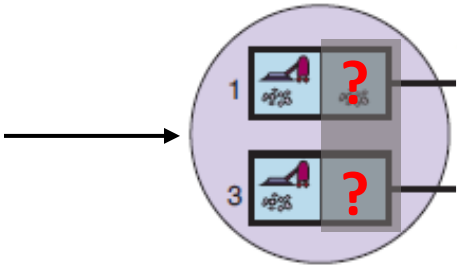
$$b_o = Update(\hat{b}, o) = \{s : \ s \in \hat{b} \ \wedge \ Percept(s) = o\}$$

Writing both steps as one update: $b \leftarrow Update(Predict(b, a), o)$

Example: Deterministic local sensing vacuum world

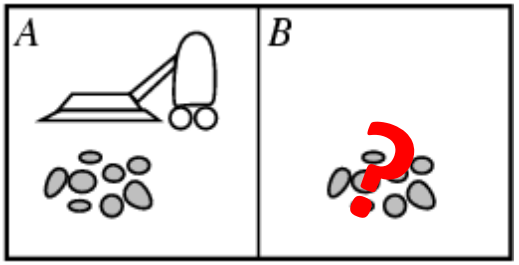




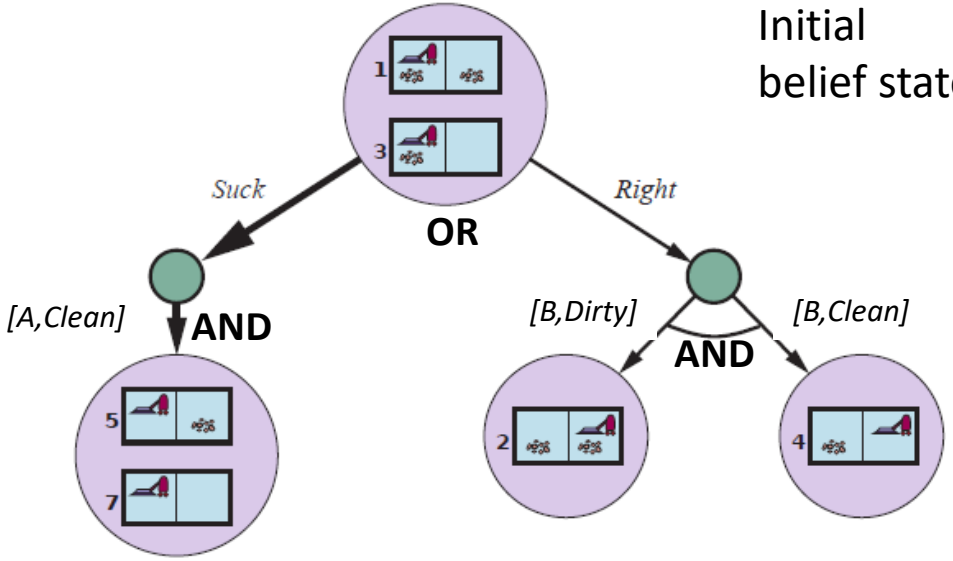


$$b \leftarrow Update(Predict(\quad b \quad , \quad a\ ), \quad o \quad)$$
$$Update(Predict(\{1,3\}, Right), [B, Dirty]) = \{2\}$$

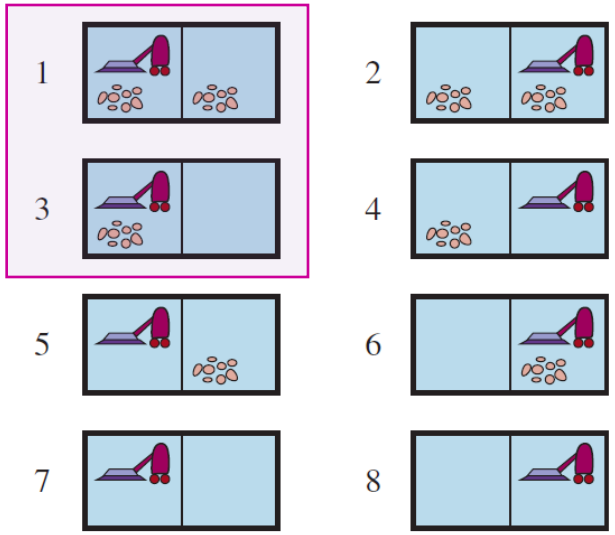
Solving Partially Observable Problems



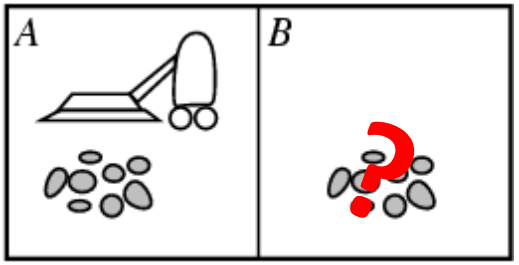
Use an AND-OR tree of belief states to create a conditional plan



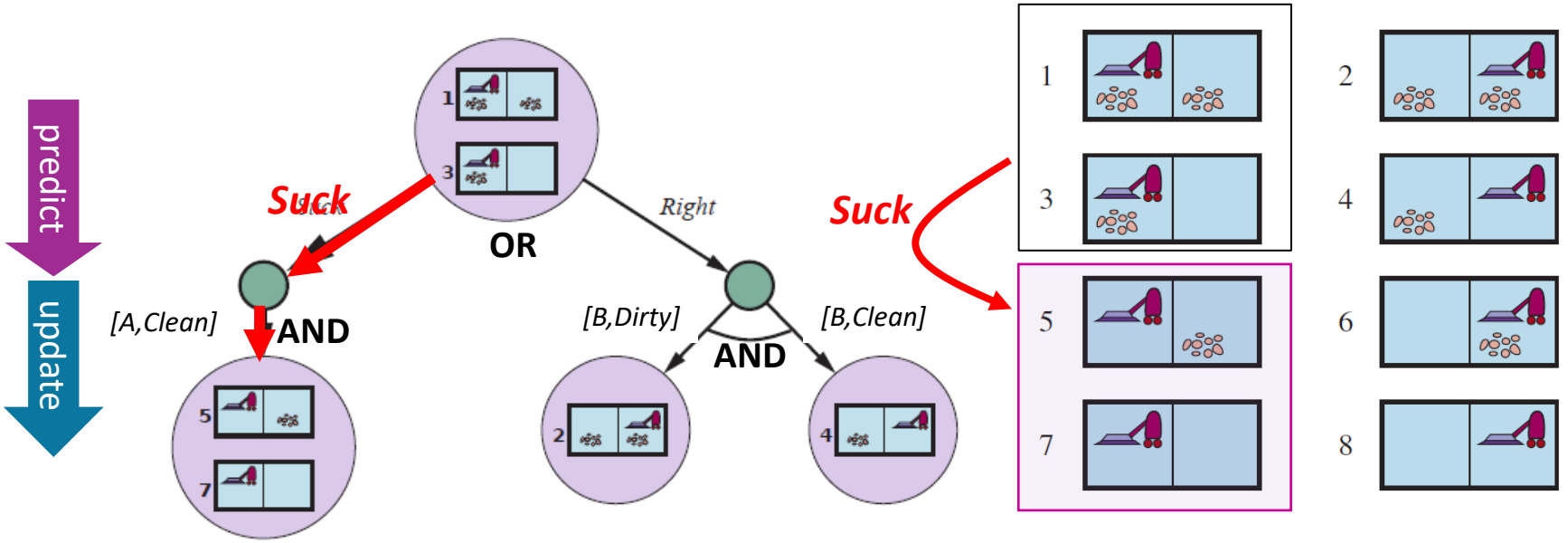
Initial belief state



Solving Partially Observable Problems 2

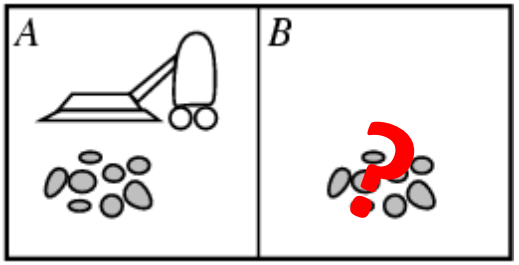


Use an AND-OR tree to create a conditional plan

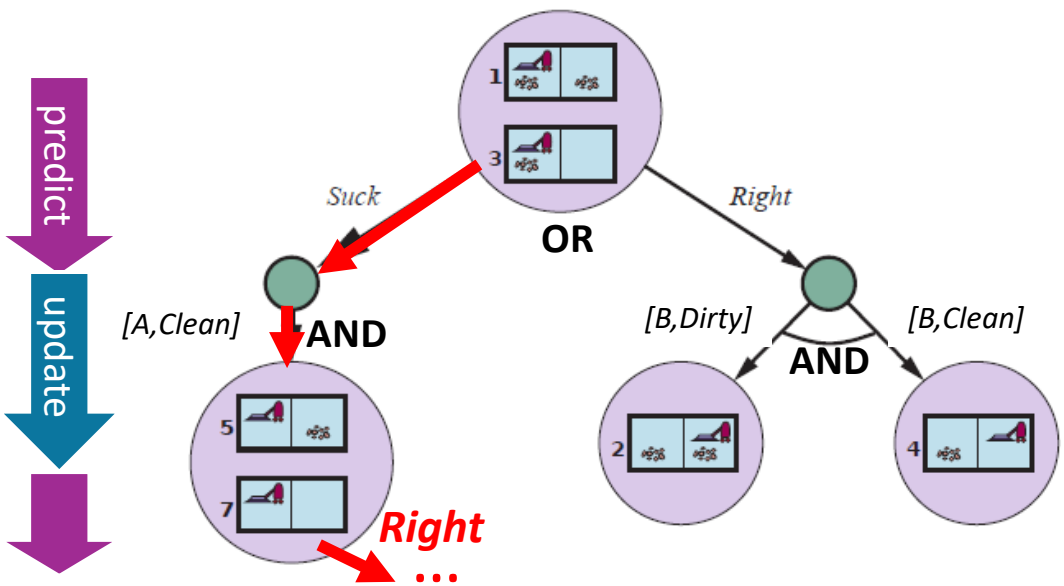


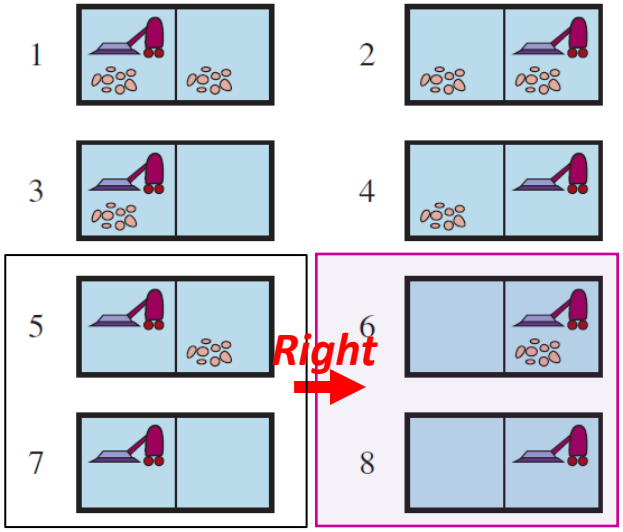
Solution: *[Suck, …]*

Solving Partially Observable Problems 3



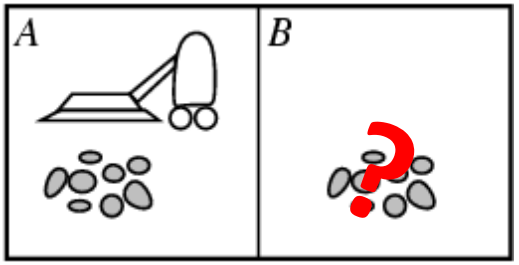
Use an AND-OR tree to create a conditional plan



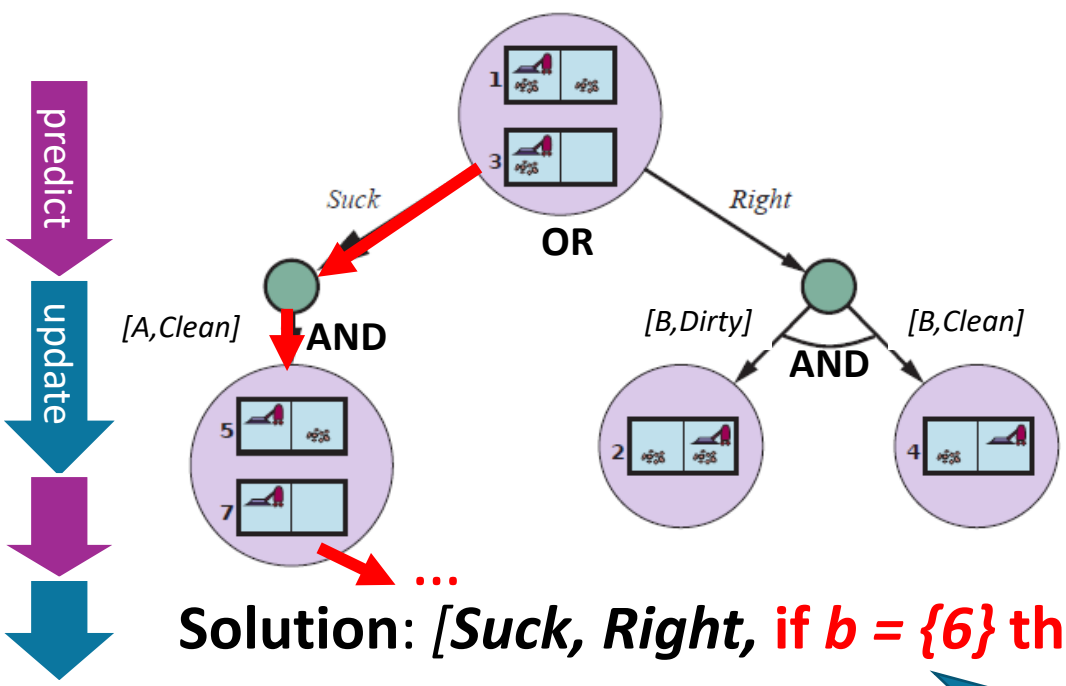


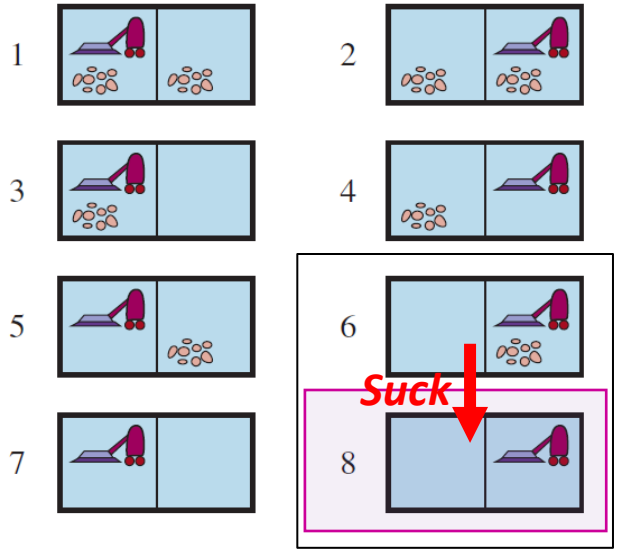
Solution: *[**Suck**, **Right**, …]*

Solving Partially Observable Problems 4



Use an AND-OR tree to create a conditional plan

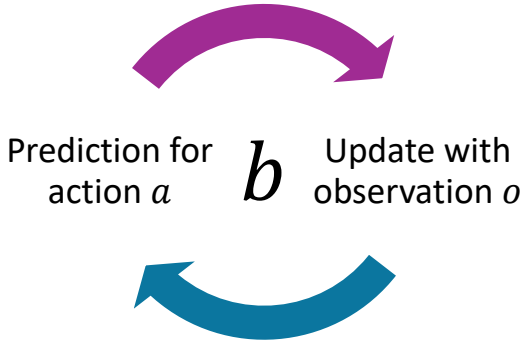




Solution: *[Suck, Right,* **if *b = {6}* then *Suck* else *[]*** *]*

b = {6} is the result of the update with *o = [B, Dirty]*

State Estimation and Approximate Belief States

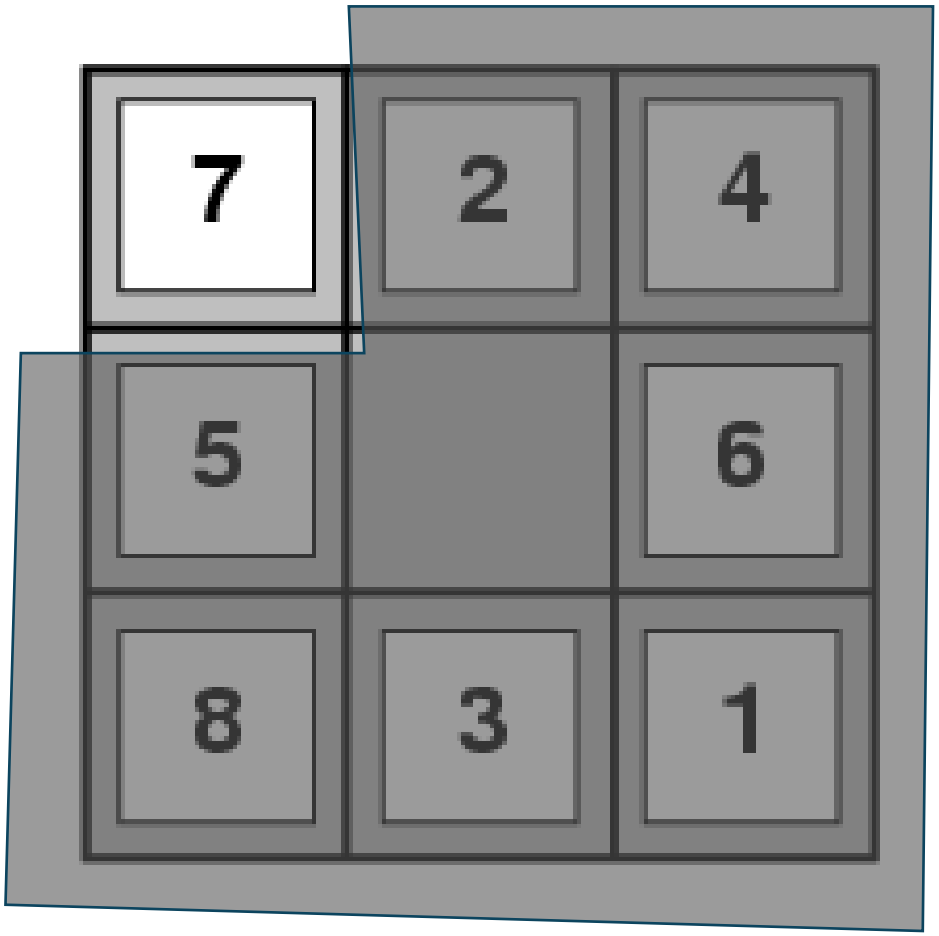


- Agents choose an **action** and then receive an **observation** from the environment.
- The agent keep track of its belief state using the following update:

$$b \leftarrow Update(Predict(b, a), o)$$

- This process is often called
  - **monitoring**,
  - **filtering**, or
  - **state estimation**.

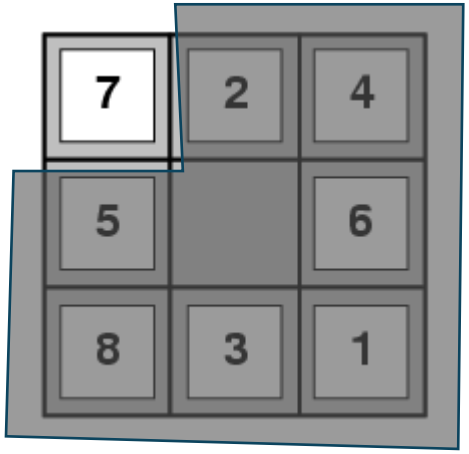
- **Issue:** The agent needs to be able to update its belief state following observations in **real time**! For many practical application, there is only time to compute an **approximate belief state!** Such approximations are commonly used in control theory and reinforcement learning.



Case Study:

Partially Observable 8-Puzzle

Partially Observable 8-Puzzle



Give a problem description for this problem.

- States:
- Initial state:
- Actions:
- Transition model:
- Goal test:
- Percept function:

This problem can be solved using an AND-OR Tree, but is there an easier solution?

a. What type of agent do we use?
b. What algorithms can be used?


Exploration
Unknown Environments and
Online Search

Online Search

- **Recall offline search**: Create a plan using the state space and the transition model before taking any action. The **plan** can be a **sequence of actions** or a **conditional plan** to account for uncertainty.

- The agent uses the transition function to predict the consequence of actions. But what if the **transition function is unknown** and we cannot predict outcomes of actions? This is called an **unknown environment.**

- **Online search** explores the real world one action at a time. Prediction is replaced by "act" and update by "observe."

- Useful for
  - **Unknown environment**: The agent has no complete model of how the environment works. It needs to explore an unknown state space and/or what actions do. I.e., it needs to **learn the transition function** $f : S \times A \rightarrow S$
  - **Real-time problems**: When offline computation takes too long and there is a penalty for sitting around and thinking.
  - **Nondeterministic domain**: Only focus on what happens instead of planning for everything!

Design Considerations for Online Search

- **Knowledge**: What does the agent already know about the outcome of actions? E.g.,
  - Does go north and then south lead to the same location?
  - Where are the walls in the maze?

  Transition function

  Often a part or all of the transition function is unknown!

- **We need a safely explorable state space/world**: There are **no irreversible actions** (e.g., traps, cliffs) or the agent needs to be able to avoid these actions using percepts.

- **Exploration order is important**: Expanding nodes in **local order** is more efficient if you must execute the actions to get observations: Depth-first search with backtracking instead of BFS or A* Search.

Online Search: A Model-based Reflex Agent to Learn the Transition Model

Setting: Environment is deterministic and fully observable (= the percept is the full state) but the transition model (function $result$) is unknown.
Approach: The agent builds the map $result(s, a) \rightarrow s'$ by trying all actions and backtracks when all actions in a state have been explored.

```
function ONLINE-DFS-AGENT(problem, s') returns an action
    s, a, the previous state and action, initially null
  persistent: result, a table mapping (s, a) to s', initially empty
              untried, a table mapping s to a list of untried actions
              unbacktracked, a table mapping s to a list of states never backtracked to

  if problem.IS-GOAL(s') then return stop
  if s' is a new state (not in untried) then untried[s'] ← problem.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
```

Learns results function (= transition function)

Untried is the "frontier"

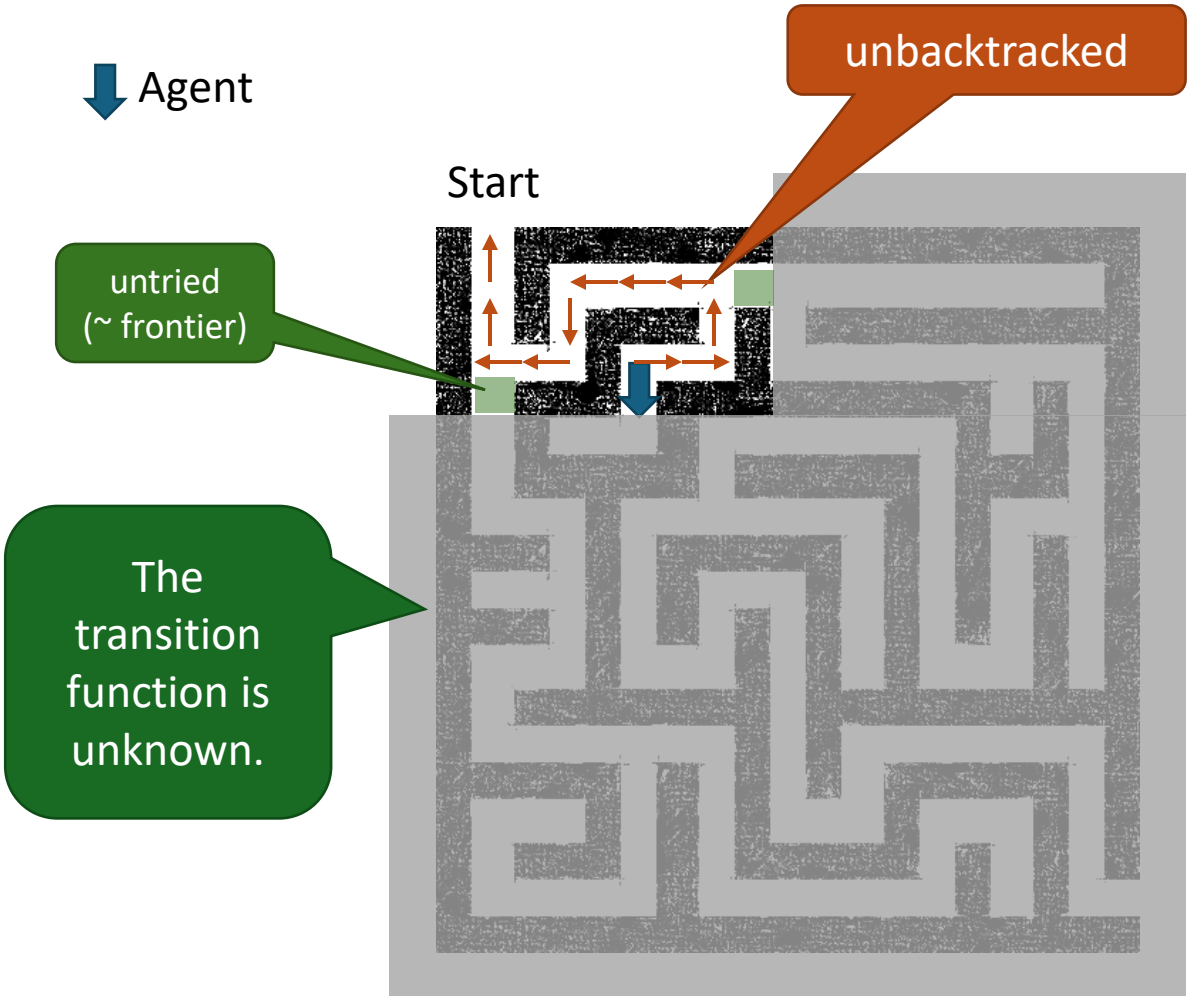
unbacktracked stores the current path

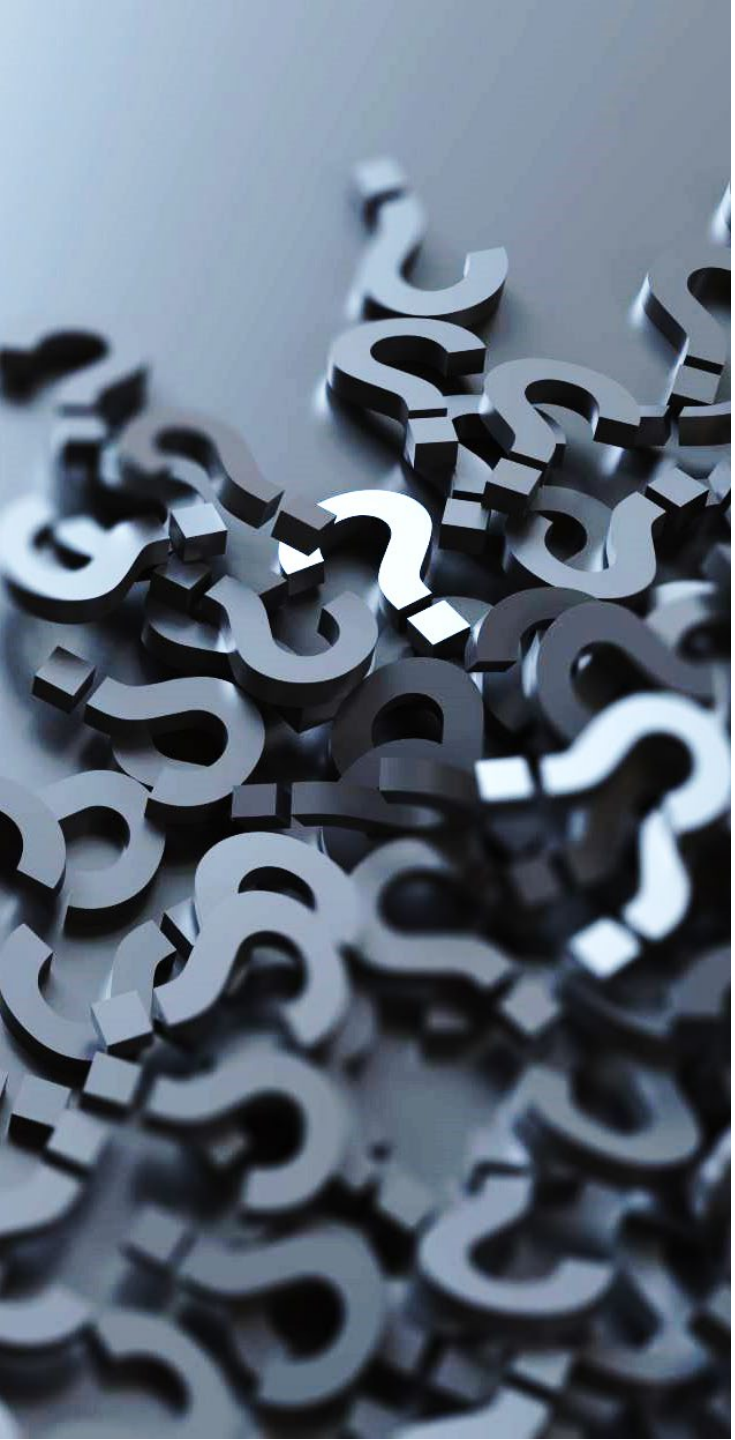
Record the found transition

Keep breadcrumbs to go back

Case Study: DFS with Backtracking for an unknown Maze

- We can only see adjacent squares and don't know the location of the goal!
- We cannot plan so we must explore by walking around!
- A simple method is to store the path for backtracking to get back to untied actions when we run into a dead end (think breadcrumbs or a string).
- This is an iterative implementation of DFS without a reached data structure. Unbacktaced is the currently explored tree branch and the frontier is untried. DFS memory management applies.





Important concepts that you should be able to explain and use now…

- Difference between solution types:
  a. a fixed actions sequence (a plan),
  b. a conditional plan (also called a strategy or policy), and
  c. exploration.
- What are belief states?
- How actions can be used to coerce the world into known states.
- How observations can be used to learn about the state: State estimation with repeated predict and update steps.
- The use of AND-OR trees to solve small problems.