CS 5/7320
Artificial Intelligence

Search with Uncertainty
AIMA Chapters 4.3-4.5

Slides by Michael Hahsler
with figures from the AIMA textbook

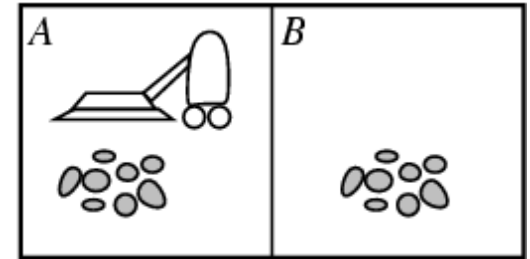Online Material

Recap: Solving Search Problems under Certainty

No Uncertainty

- **Full observability**: The agent always knows (=can observe) the state.

- **Deterministic environment** with a known transition model
  $$Result(s, a) = s'$$
  The agent can predict the outcome of its actions.

State space: A state completely describes the condition of the environment and the agent.

Initial state

Goal states
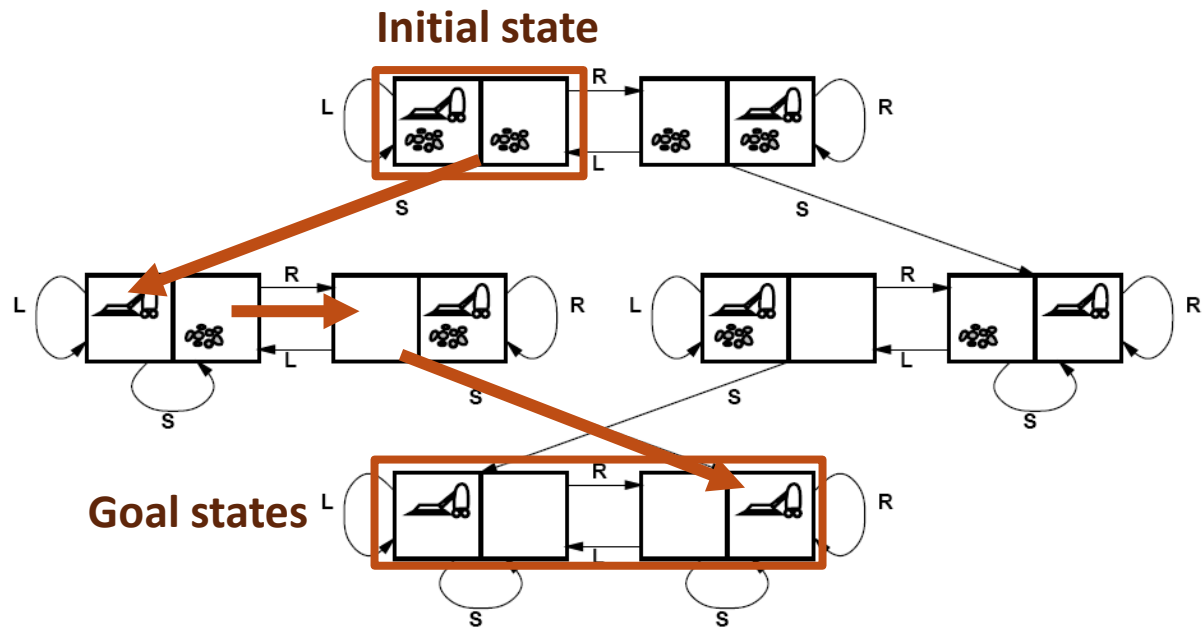
Solution: Use tree search in the planning phase to create a **sequence of actions** also called a **plan.** Then blindly execute the plan: **[Suck, Right, Suck]**

Sources and Consequence of Uncertainty

Sources: The environment may be

- **Not fully observable**: The agent may be uncertain about its current state.
- **Stochastic (transition function)**: The agent may not be able to perfectly predict the outcome of its actions.

Consequences:

1. The agent needs to keep track of all the states it could be in.
   This set is called a *belief state.*

2. A fixed precomputed plan (sequence of actions) does not work for stochastic transition functions, but a

   conditional plan (also called strategy or policy)

   that depends on percepts is needed.

Types of uncertainty in the environment*

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

No observations:
Sensorless problems.

Partially observable environments:
The agent cannot directly observe the state of the environment.

Exploration:
Unknown environments and online search.

* we will quantify uncertainty with probabilities later.

Nondeterministic Actions

Stochastic Environment (Stochastic Transition Model)

Definition: Nondeterministic Actions

The outcome of actions in the environment is nondeterministic = the **transition model needs to describe uncertainty.**
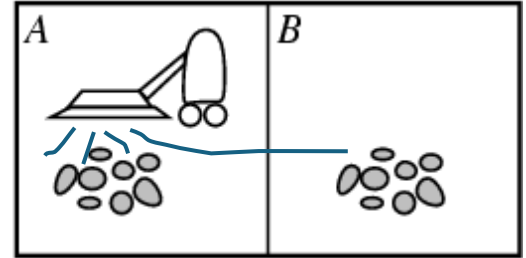
Example transition:

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.
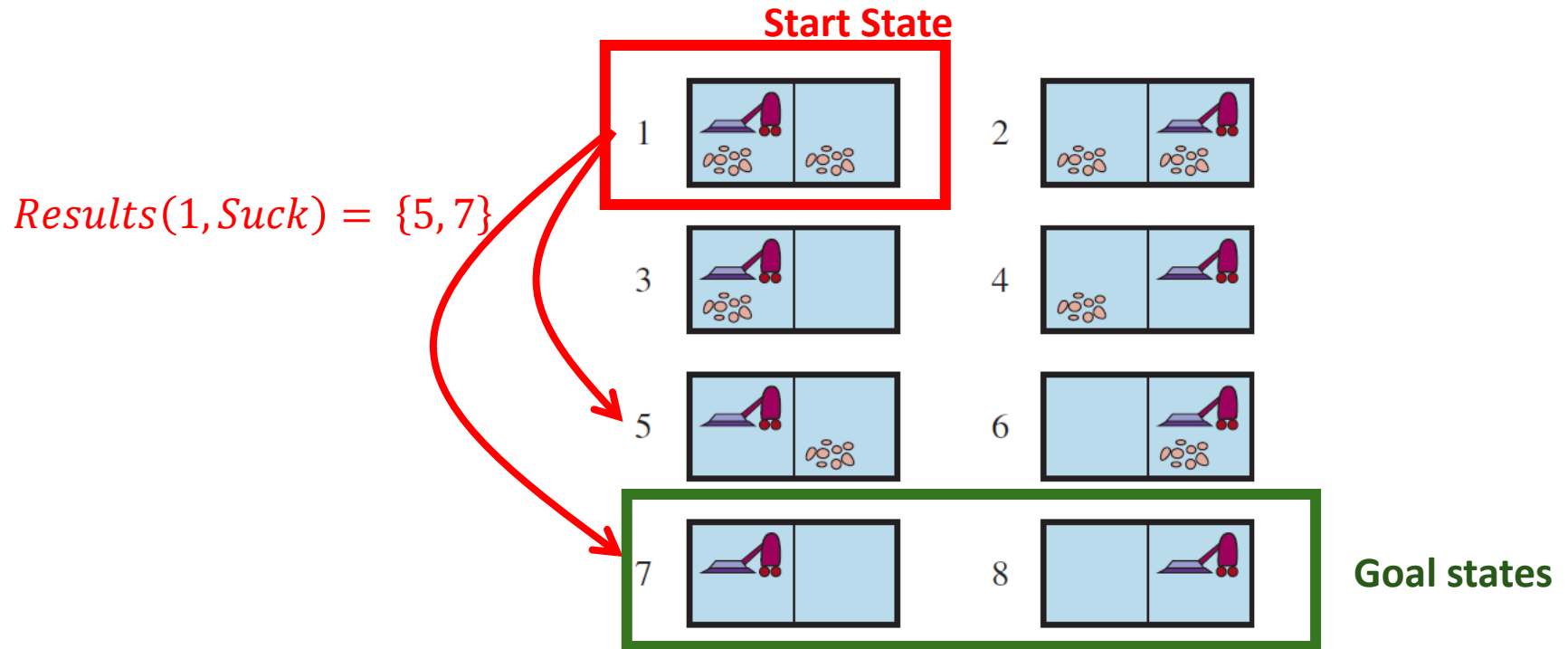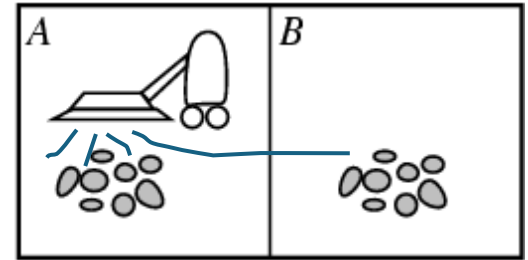
Example:
Erratic Vacuum World

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.

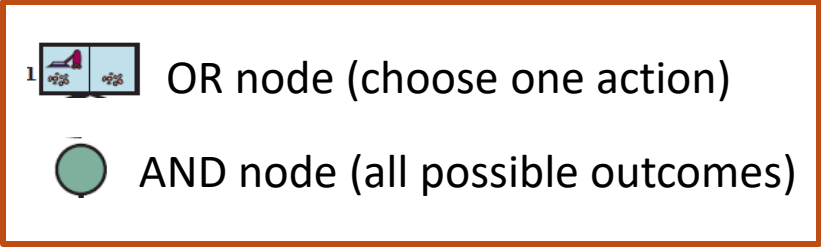# Example: Erratic Vacuum World

Start State

1

2

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

3

4

5

6

Goal states

7

8

Suck can 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

OR node (choose one action)

AND node (all possible outcomes)

OR

AND

Suck

Right

7 GOAL

5 Suck Right

2 Left Suck

5 LOOP

1 LOOP

6 Suck Left

1 LOOP

8 GOAL

4

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

8 GOAL

5 LOOP

...

Search the AND-OR Tree

OR node

AND node

Suck

Right

Suck
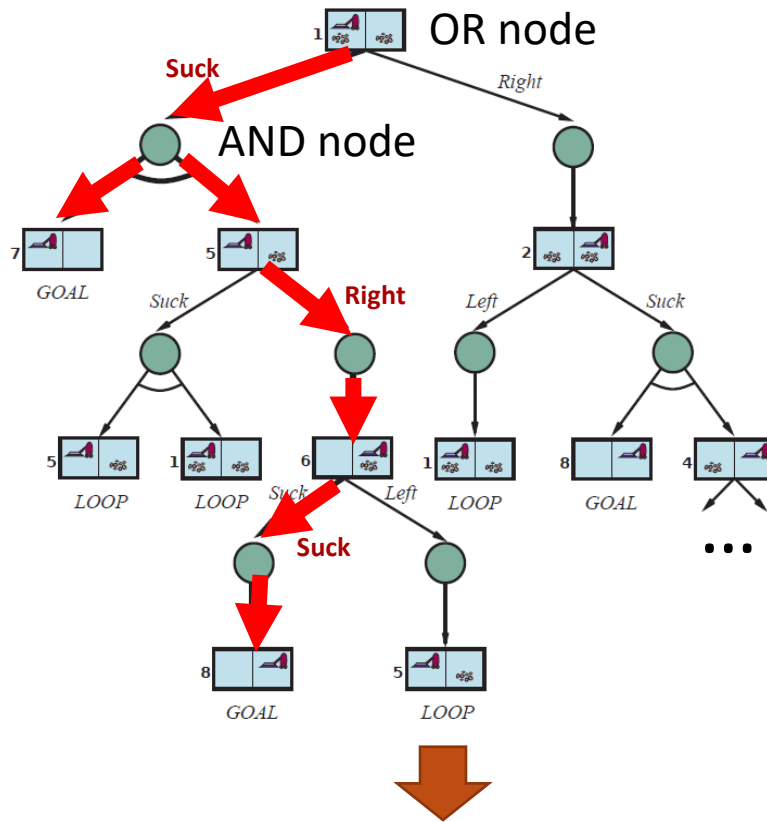
- **Goal**: Find a subtree with one action for each OR node and considering all outcomes of the AND nodes that has only goal leaf nodes.

- Descend the tree depth-first:
  - OR node: trying one action at a time.
  - AND node: consider all outcomes and check recursively.
  - Ignore cycles.
  - Abandon a subtree if not all leaf nodes are the desired goal nodes.
  - Stop when **the first complete subtree with only goal leaf nodes is found.**

- Construct the conditional plan that represents the subtree starting at the root node.

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

Recursive AND-OR Tree Search (DFS)

= nested If-then-else statements

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

path is used for cycle checking!

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**
      $plan \leftarrow$ AND-SEARCH($problem$, RESULTS($state$, $action$), [$state$] + $path$])
      if $plan \neq failure$ **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**   **// consider all possible outcomes, none can fail!**
      $plan_i \leftarrow$ 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** $plan_n$]
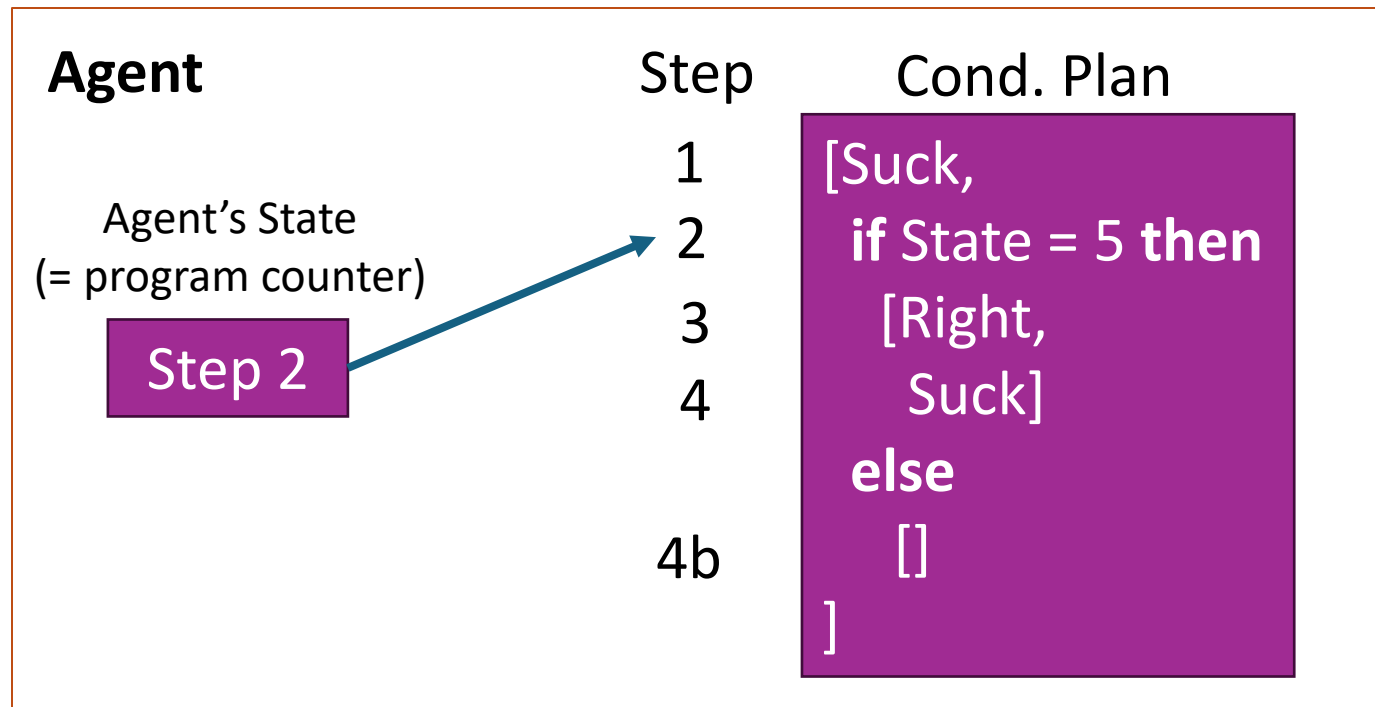
Notes:
- The DFS search tree is implicitly created using the call stack (recursive algorithm).
- DFS is **not optimal**! It returns the first valid plan (subtree) that it finds.

An Agent using the Conditional Plan

- Planning uses search to find a conditional plan that always 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 conditions in the if-statements.

Example: After the initial action "suck"

Agent	Step	Cond. Plan
	1	[Suck,
Agent's State (= program counter)	2	**if** State = 5 **then**
Step 2	3	[Right,
	4	Suck]
		else
	4b	[]
		]

Search With No Observations

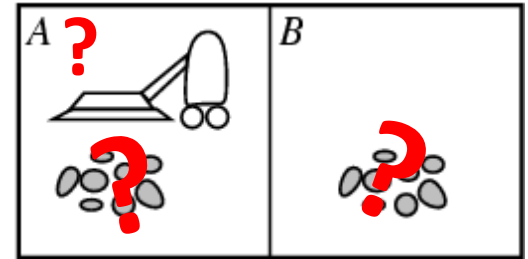Using actions to "coerce" the world into a smaller set of known states

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.
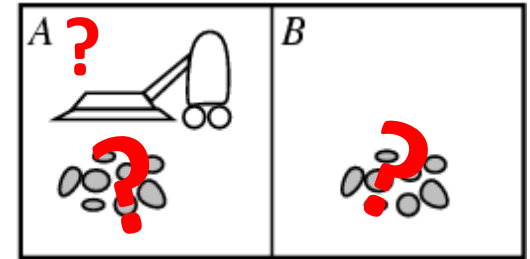
Definition: Belief State

- The agent does not know exactly what state it is 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.
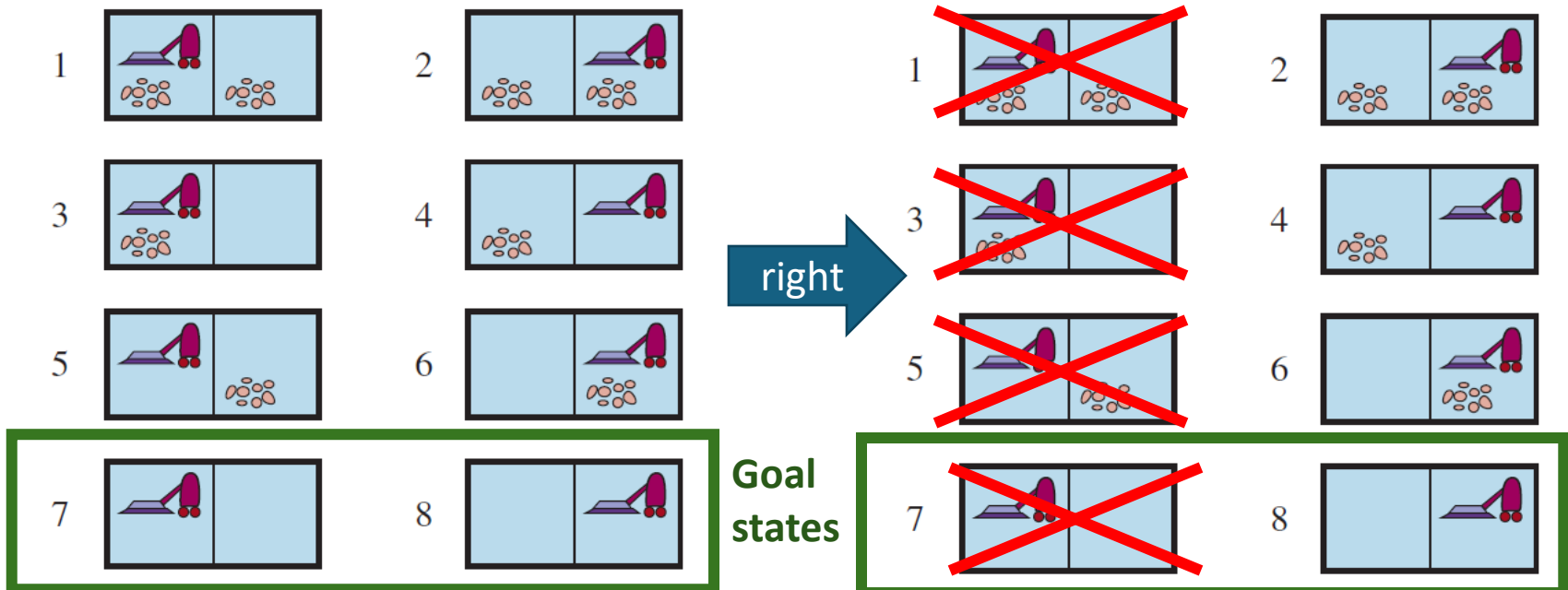- Example: b = $\{s_2, s_4, s_6\}$
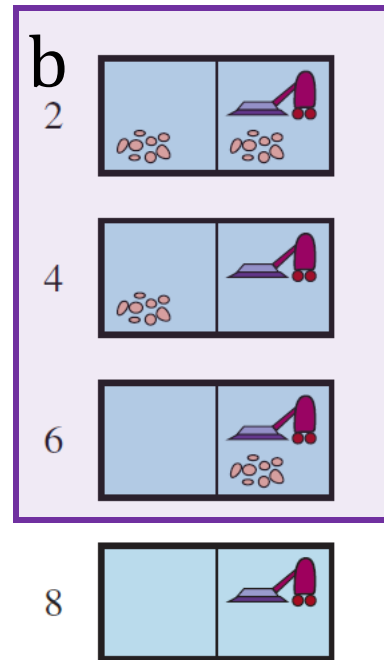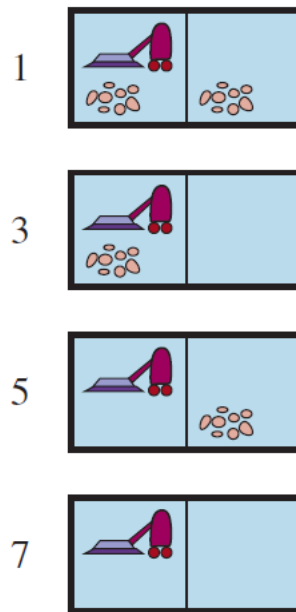
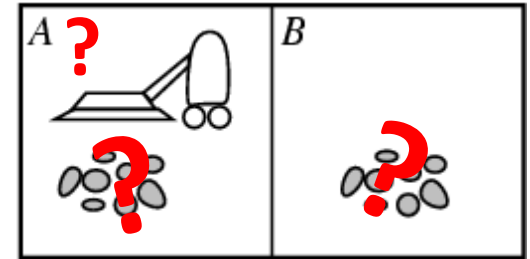# Actions to Coerce the World into Known States

- Actions can reduce the number of possible states.
- **Example**: Deterministic but unobservable vacuum world. The agent does not know its position or the dirt distribution.

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

right

Goal states

Actions to Coerce the World into Known States

- Actions can reduce the number of possible states.
- **Example**: Deterministic but unobservable vacuum world. The agent does not know its position or the dirt distribution.
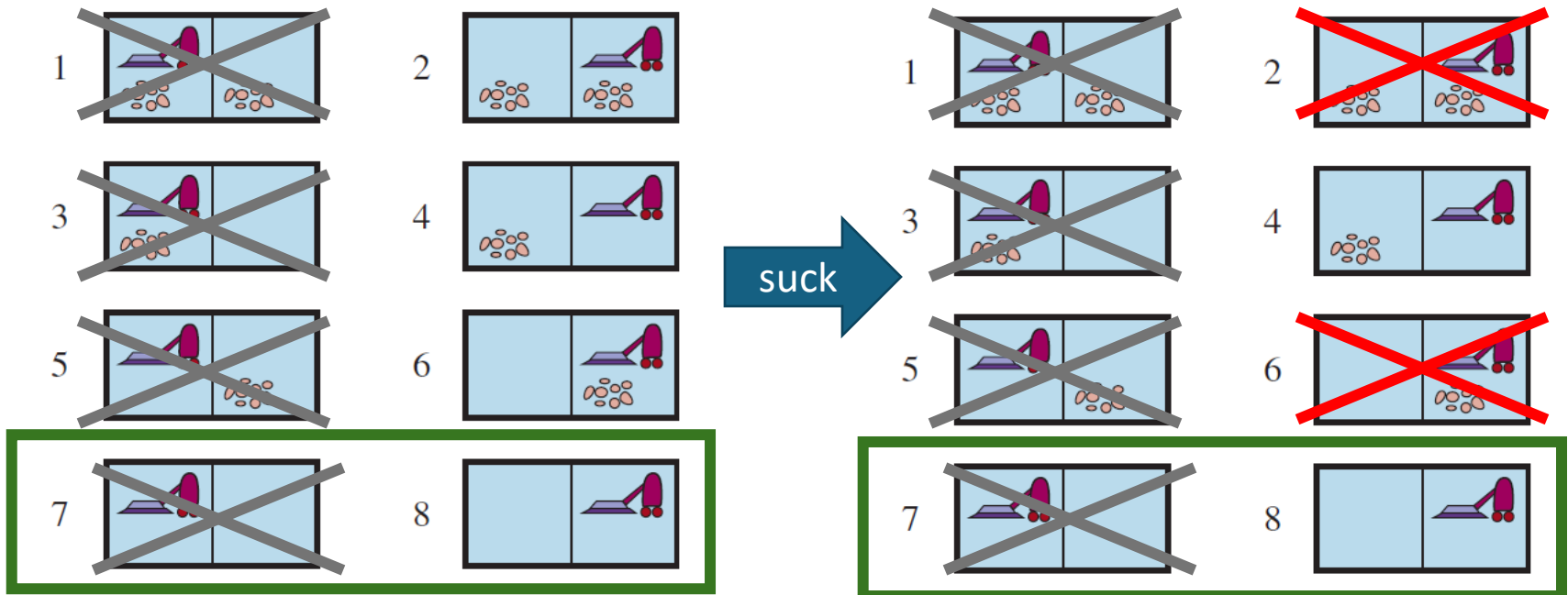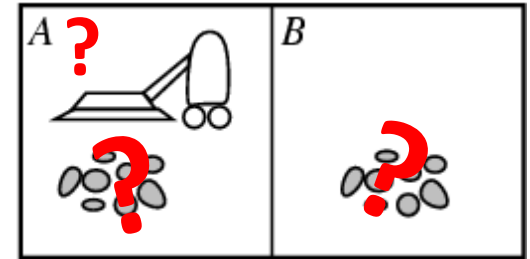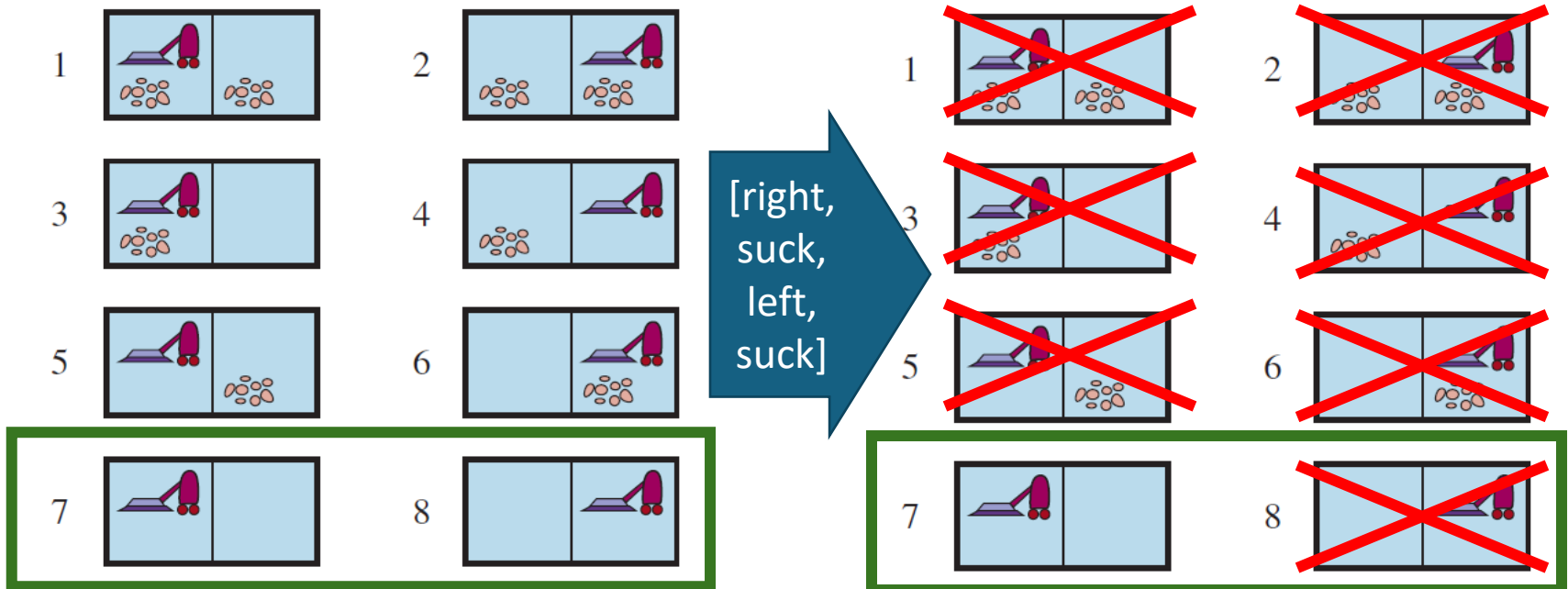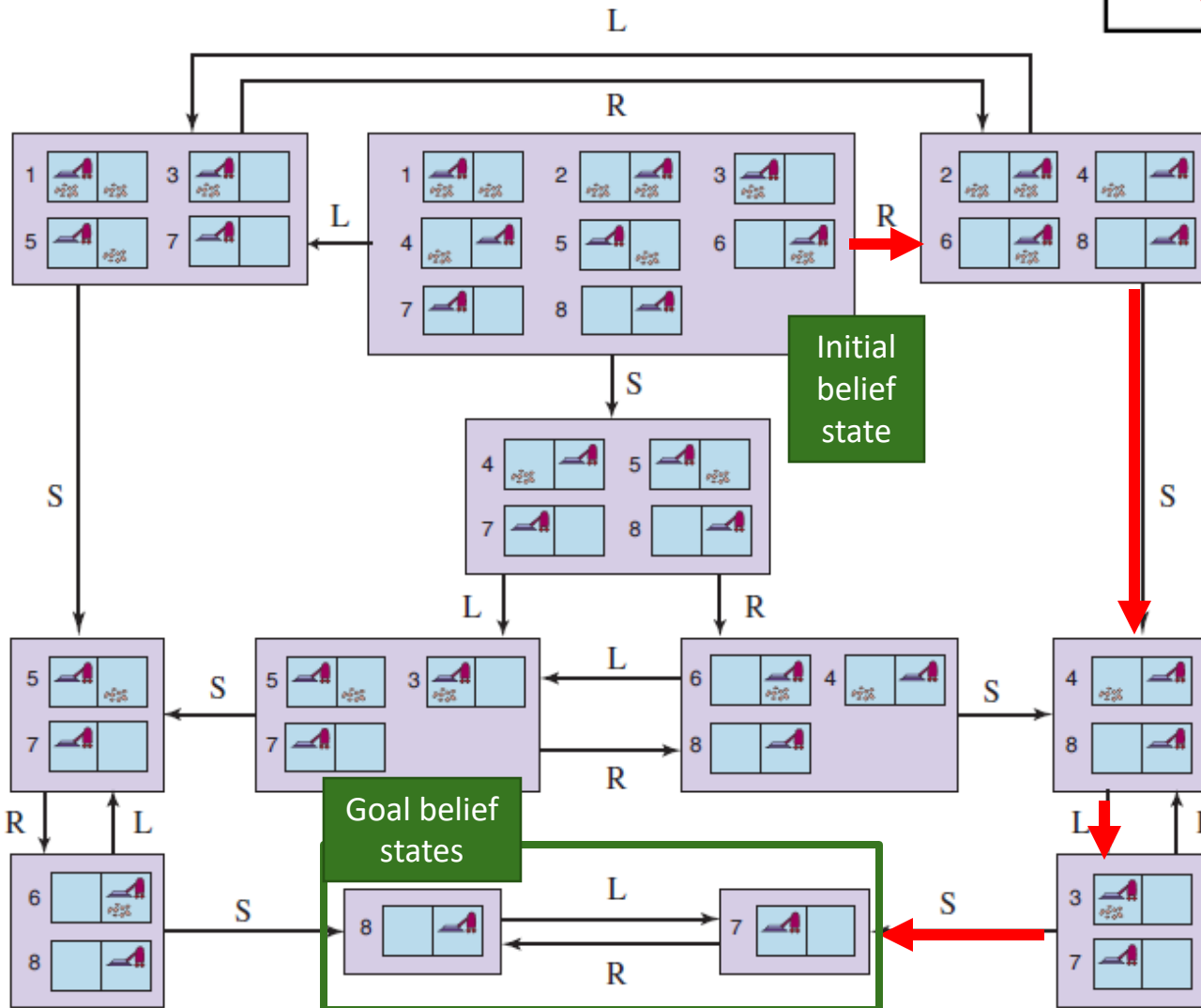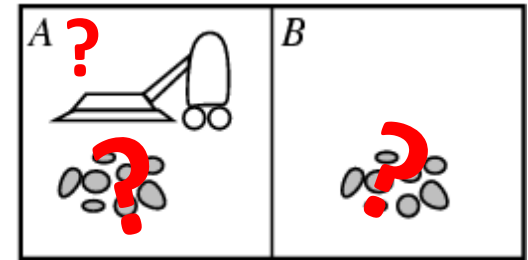

suck

Actions to Coerce the World into Known States

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

[right, suck, left, suck]

The Reachable Belief State Space

The size of the belief state space is the powerset of the original N states:

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

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

Initial belief state

Goal belief states

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

Finding a Plan

Formulate as a regular search problem and solve with DFS, IDS, 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.
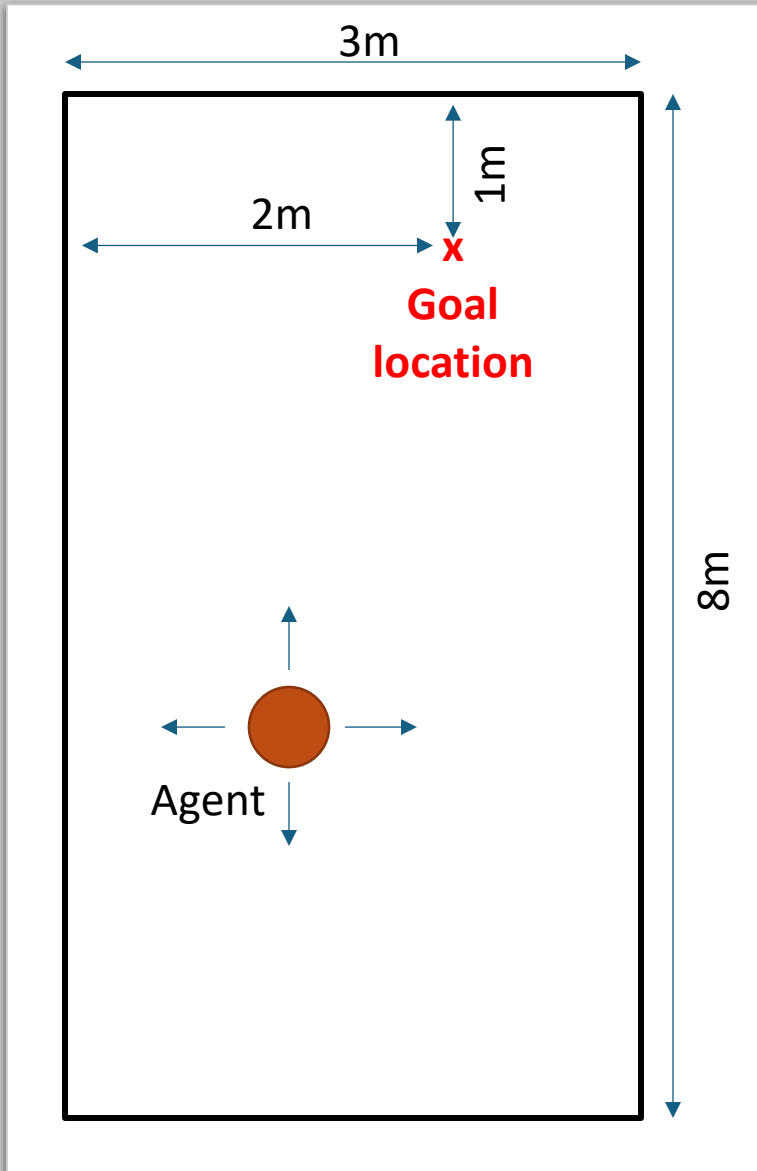
Case Study

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

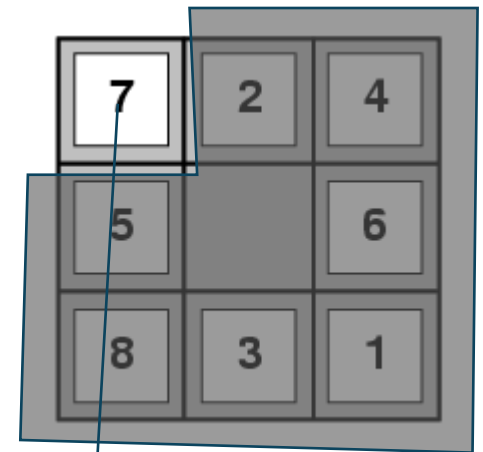Using Observations to Learn About the State

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

Prediction for action a b Update with observation o
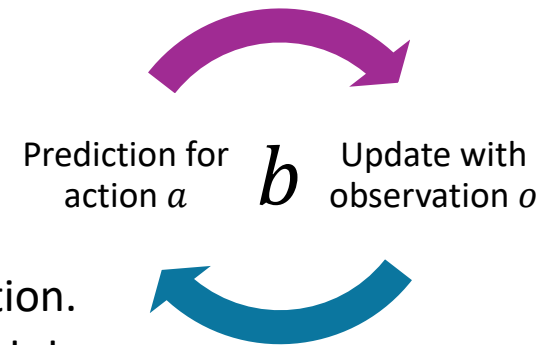
The agents chooses an action and then receive an observation.

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:

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

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

Example: Deterministic local sensing vacuum world

Prediction for action a b Update with observation o

Predict for action a → Update with observation o

Right   [B,Dirty]

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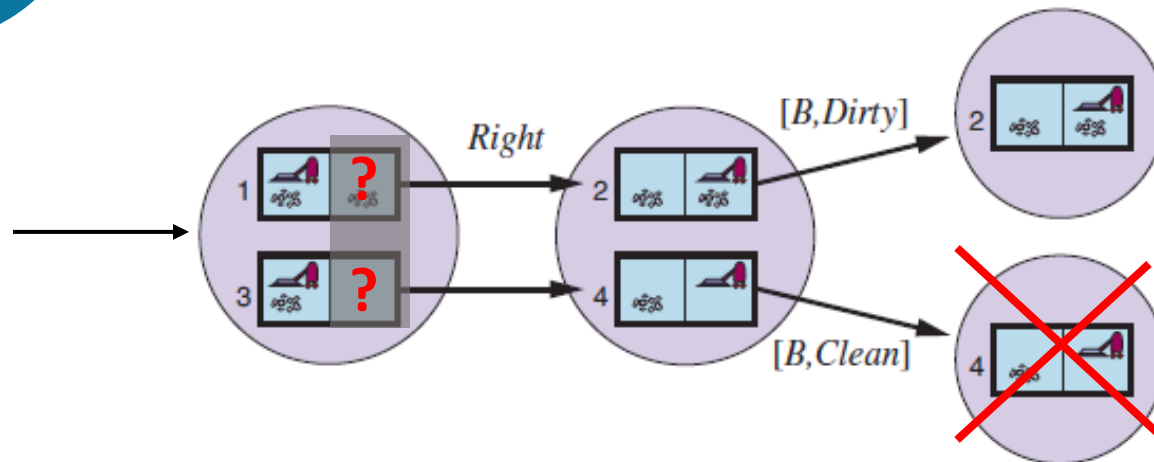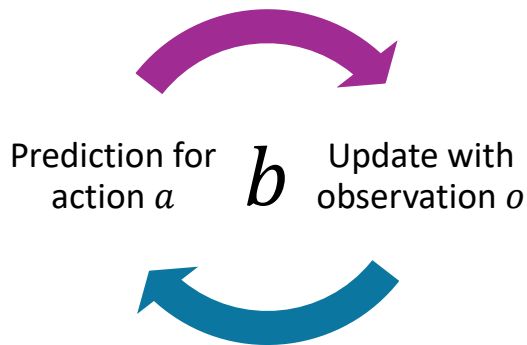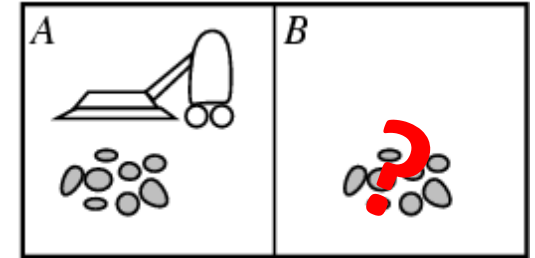
$$b \leftarrow Update(Predict(\ \ b\ ,\ \ \ a\ ),\ \ \ o\ \ \ )$$
$$Update(Predict(\{1,3\}, Right), [B, Dirty]) = \{2\}$$

Solving Partially Observable Problems
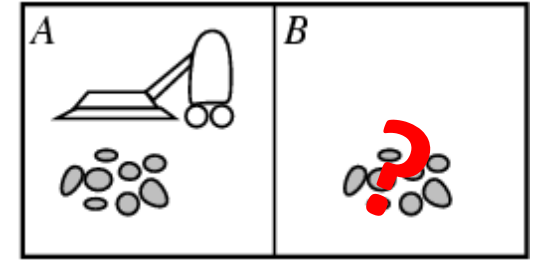
Use an AND-OR tree of belief states to create a **conditional plan.**
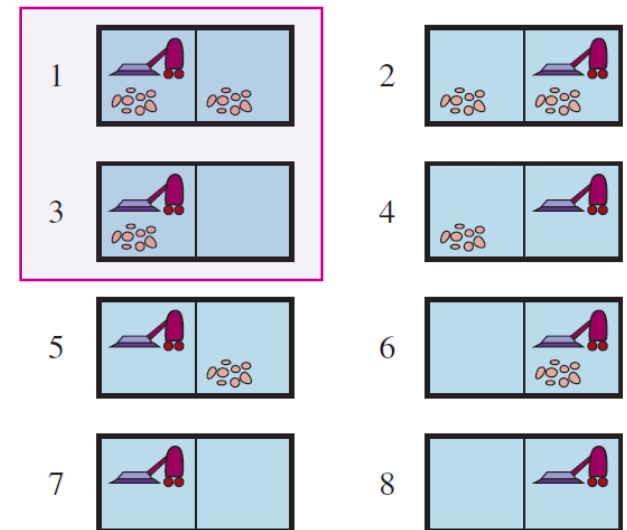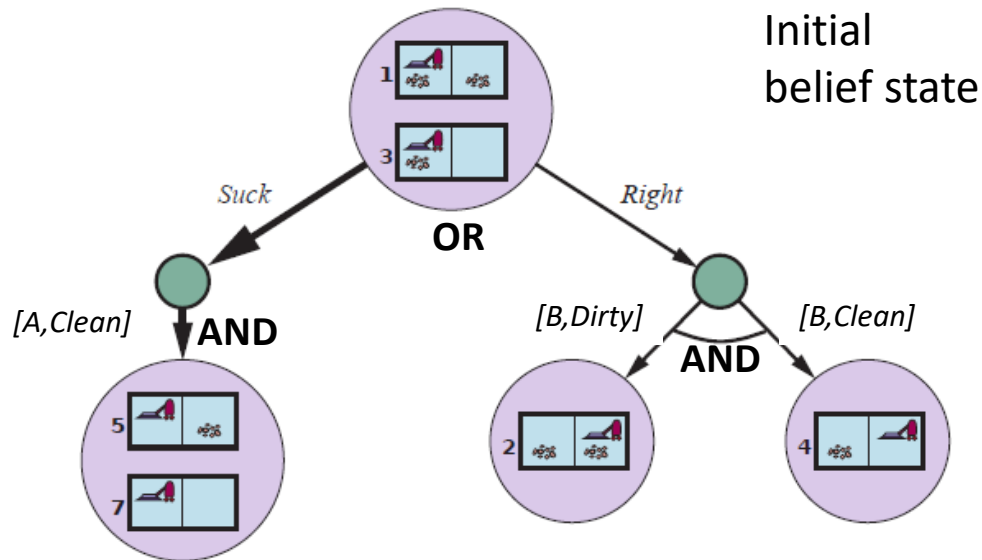
Solving Partially Observable Problems 2

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

predict

update

Suck

Suck

OR

AND

[A,Clean]   **AND**

[B,Dirty]   [B,Clean]

Plan: *[Suck, …]*

Solving Partially Observable Problems 3

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

predict

update

Suck

Right

OR

[A,Clean] **AND**

[B,Dirty] **AND** [B,Clean]

Right
...

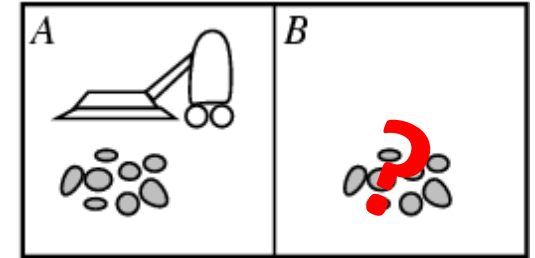**Plan**: *[Suck, Right, ...]*

Right
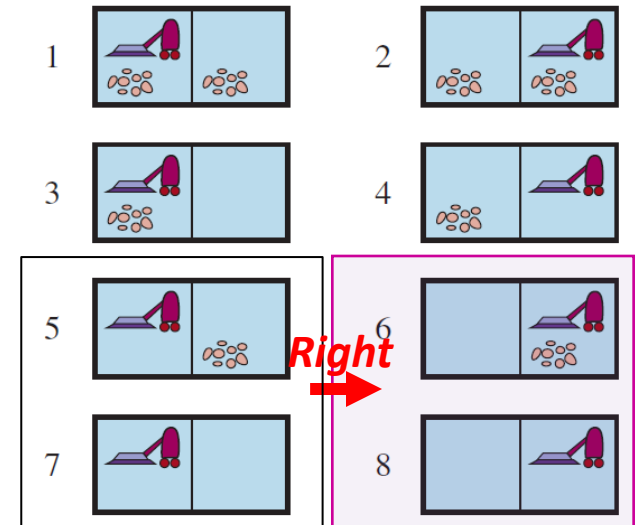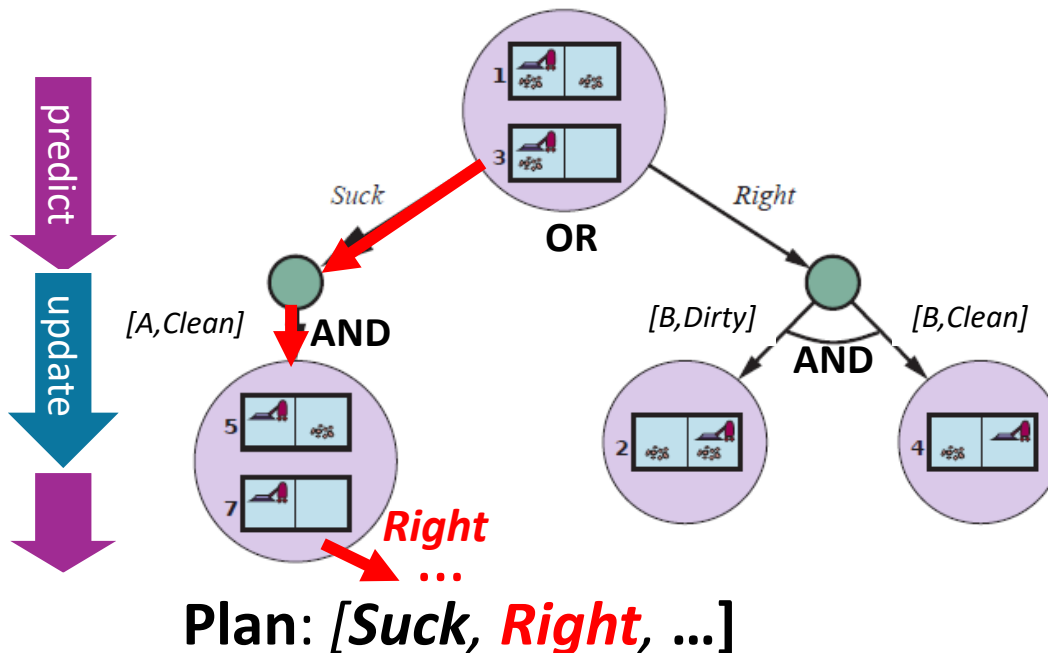
Solving Partially Observable Problems 4

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

predict

update

OR

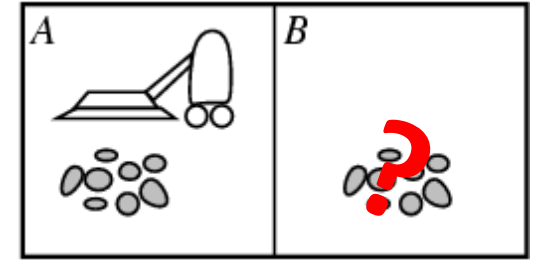*Suck*

Right

[A,Clean] **AND**

[B,Dirty] **AND** [B,Clean]

...

Suck

Plan: *[Suck, Right,* **if *b = {6}* then *Suck* else *[]]***
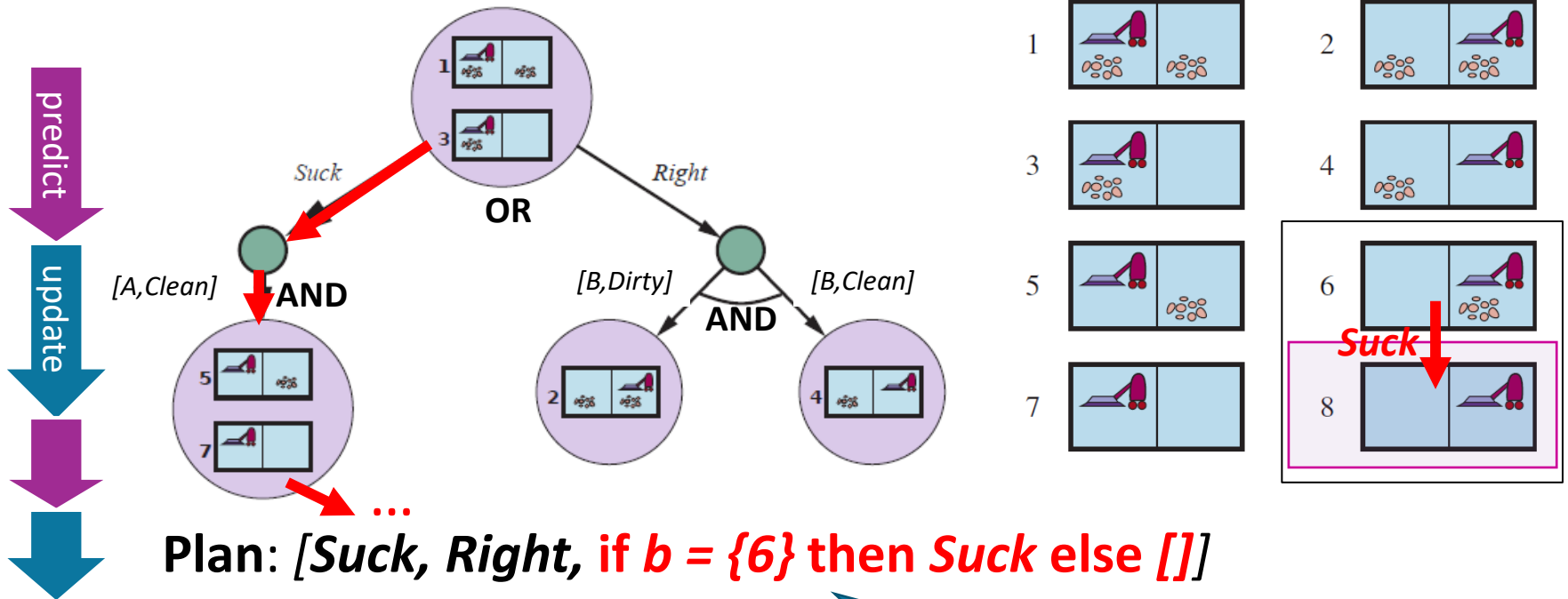
b = {6} is the result of the update with *o = [B, Dirty]*

State Estimation and Approximate Belief States

Prediction for action a b Update with observation o
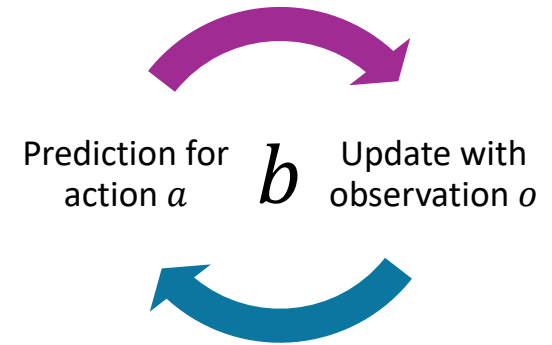
- Agents choose an **action** and then receive an **observation** from the environment.

- The agent keeps 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 applications, 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
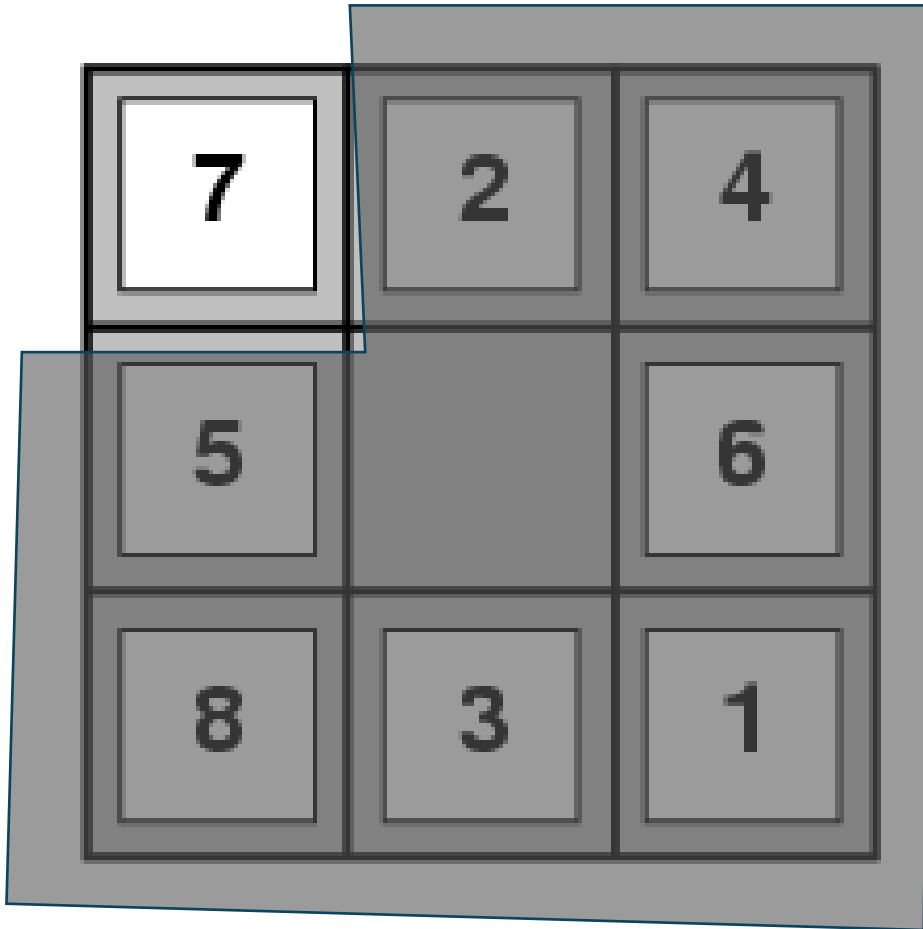
Partially Observable 8-Puzzle

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 agents would we use?
b.   What algorithms can be used?
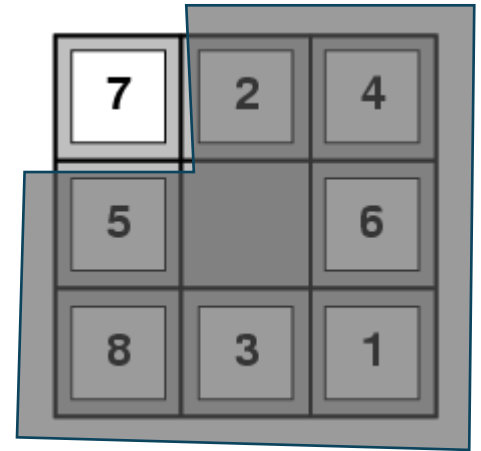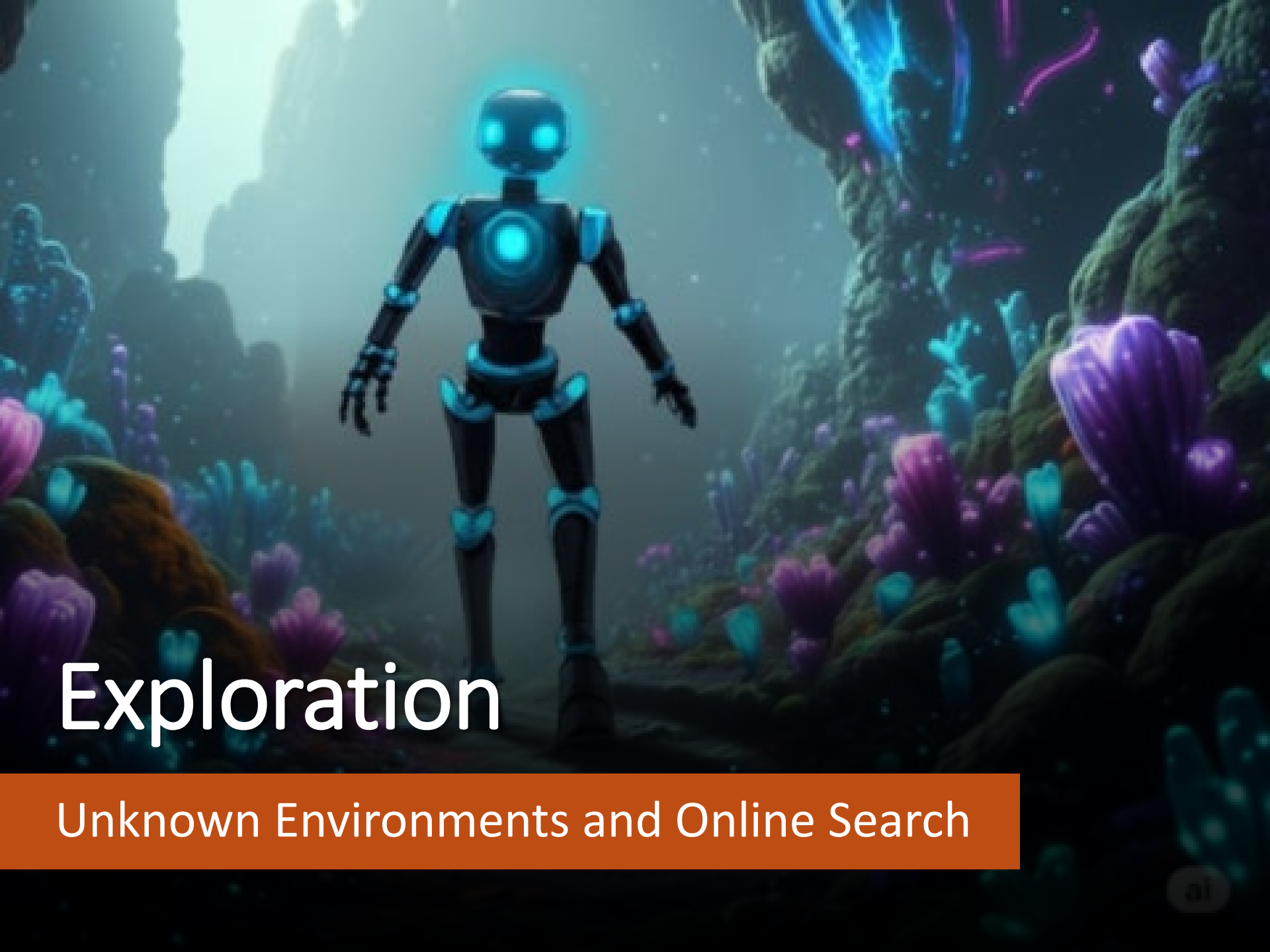
Exploration

Unknown Environments and Online Search

Recap: Offline Search

- **Offline search aka planning**: 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** that uses observations to account for uncertainty or imperfect observability.
- The agent plans using search with the known transition function to predict the consequence of actions.

- **Issue**: In an **unknown environment,** we do not know the transition function.
- We cannot predict outcomes of actions; therefore, we cannot plan using offline search!

Online Search

- **Online search** does not use planning! It explores the real world one action at a time. Offline prediction and update are replaced by "act" and "observe."

Act → Observe → Act → Observe → Act → **...**

- 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**
  $$f : S \times A \to S$$
  - **Real-time problems**: When offline computation takes too long, and there is a penalty for sitting around and thinking.
  - **Nondeterministic domain**: Conditional plans become very large. 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?

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

  Transition function

- **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 during exploration using percepts.

- **Exploration order is important**: Expanding nodes in **local order** (= close by) is more efficient if you must execute the actions to get observations: Use 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 $result(s, a) \to s'$ by trying all actions and backtracks when all actions in a state have been explored (this is a form of iterative DFS called backtracking DFS).

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 stack with the current path

if $problem$.IS-GOAL(s') **then return** $stop$
if s' is a new state (not in $untried$) **then** $untried[s'] \leftarrow problem$.ACTIONS($s'$)
if s is not null **then**
     $result[s, a] \leftarrow s'$
     add s to the front of $unbacktracked[s']$
if $untried[s']$ is empty **then**
     if $unbacktracked[s']$ is empty **then return** $stop$
     else $a \leftarrow$ an action b such that $result[s', b] = $ POP($unbacktracked[s']$)
else $a \leftarrow$ POP($untried[s']$)
$s \leftarrow s'$
return a

Learn the result function (= transition function)

Untried is the "frontier"

Unbacktracked stores the current path

Record found transitions

Keep breadcrumbs to go back later

Use breadcrumbs to walk back

Case Study: DFS with Backtracking for an unknown Maze

- We don't have a map (transition function) of the maze. We can only see adjacent squares.

- We cannot plan, so we must explore by walking around!

- A simple method is **backtracking DFS** that only stores the current path for backtracking (on a stack) to get back to untied actions when we run into a dead end (think leaving breadcrumbs or a string).

- This is an iterative implementation of DFS without a reached data structure. Unbacktaced represents the currently explored path, and untried represents the frontier. DFS memory management applies.

Agent

Start

Unbacktracked (= current path)

untried (~ frontier)

The transition function is unknown.
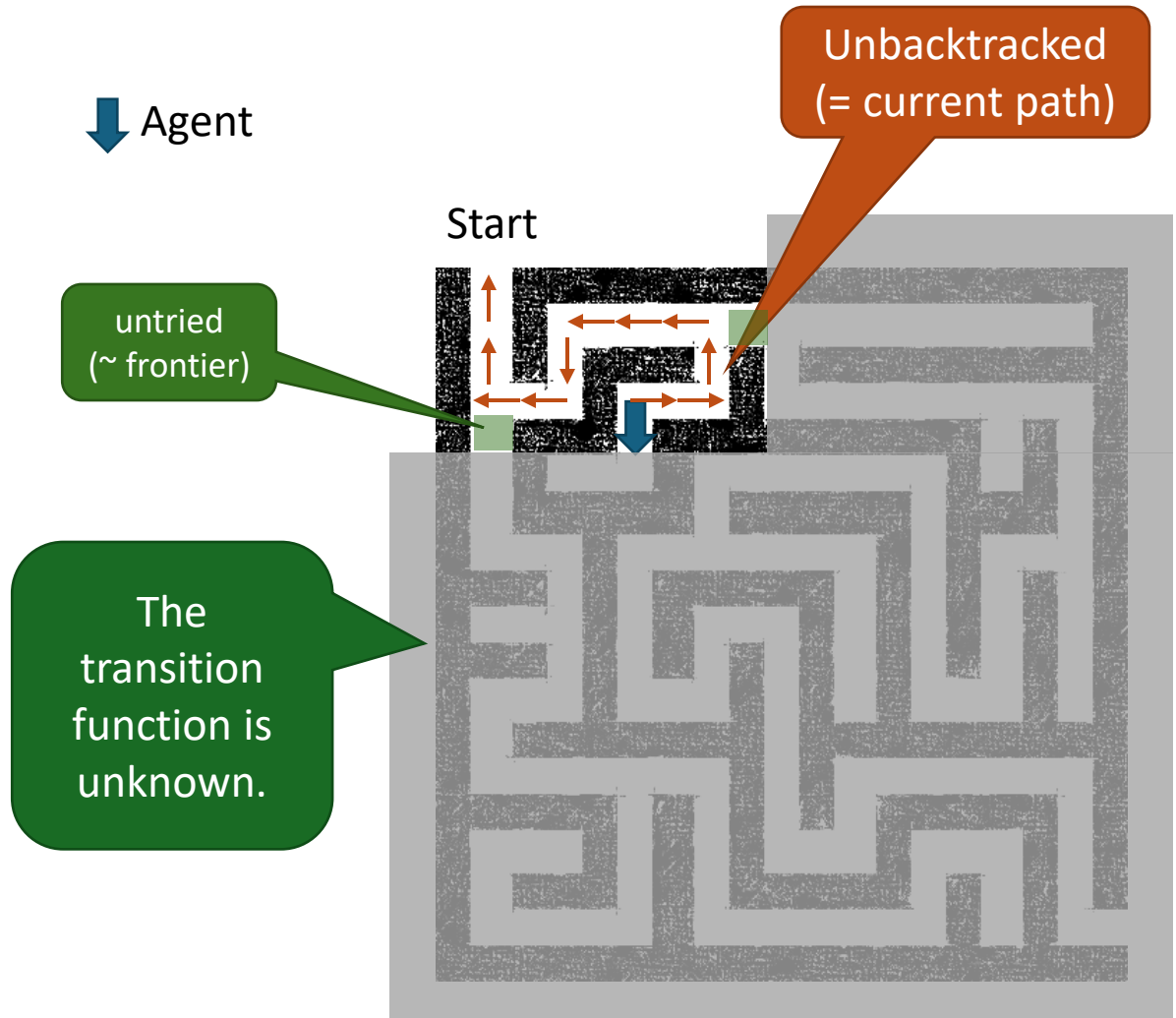
Important concepts that you should be able to explain and use now...

- Difference between solution types:
    a. a fixed action 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 actions and observations (from **percept functions**) 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.
- Large problems are hard!