CS W4701 Artificial Intelligence

Fall 2013

Chapter 3:

Problem Solving Agents

Jonathan Voris

(based on slides by Sal Stolfo)

Assignment 1

- Due in one week!
  - Tuesday October 1st @ 11:59:59 PM EDT
- Please follow submission instructions
  - https://www.cs.columbia.edu/~jvoris/AI/notes/Assignment%20submission%20guideline-Spring11.pdf
- Submit:
  - Code
  - Test Input/Output File
  - README Documentation File
- Both CLIC machines and LispWorks are acceptable platforms

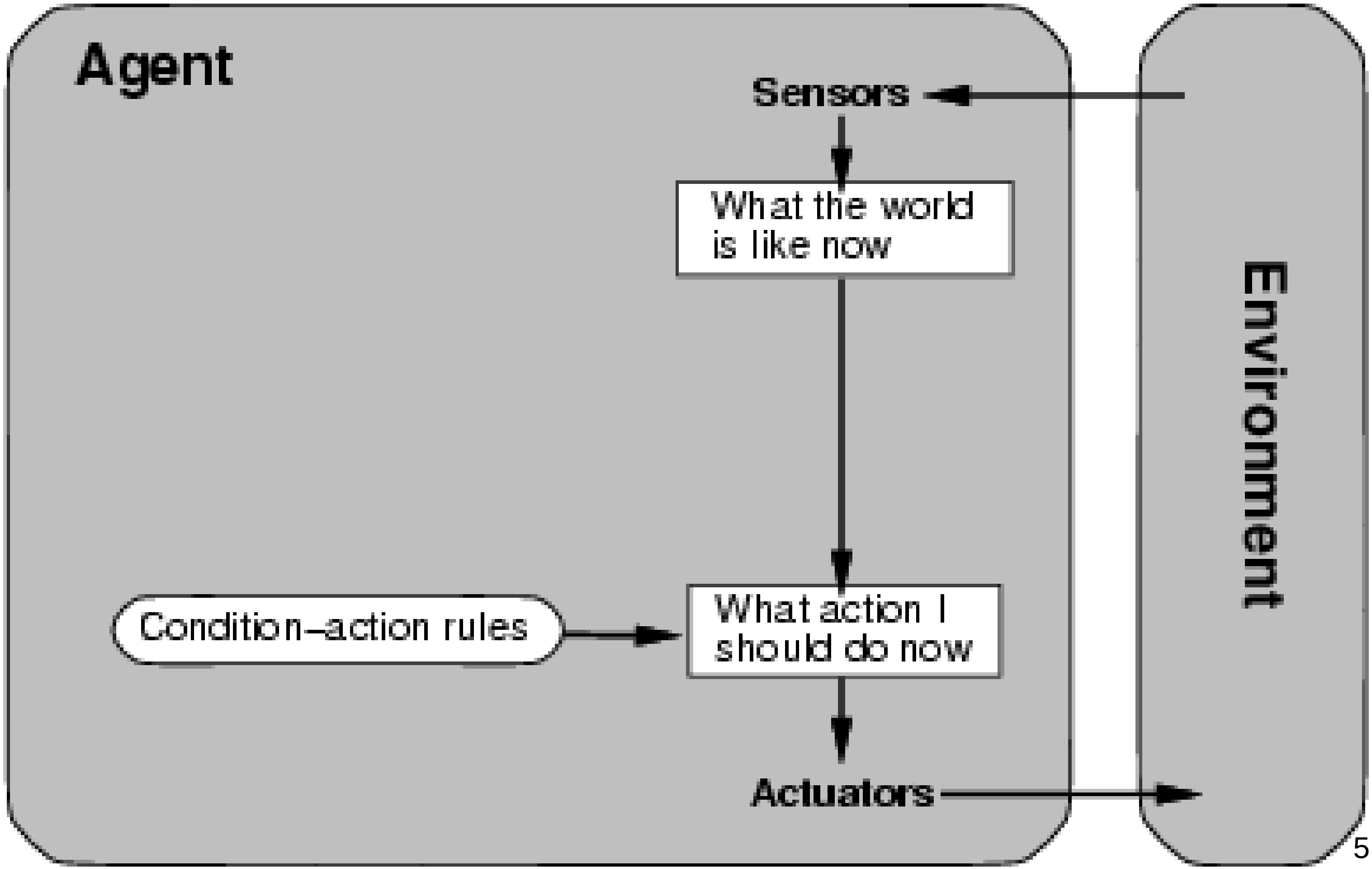
Recap

- Covered AI history
- Defined AI as…?
- Described intelligent agents
  - But how do you build them?

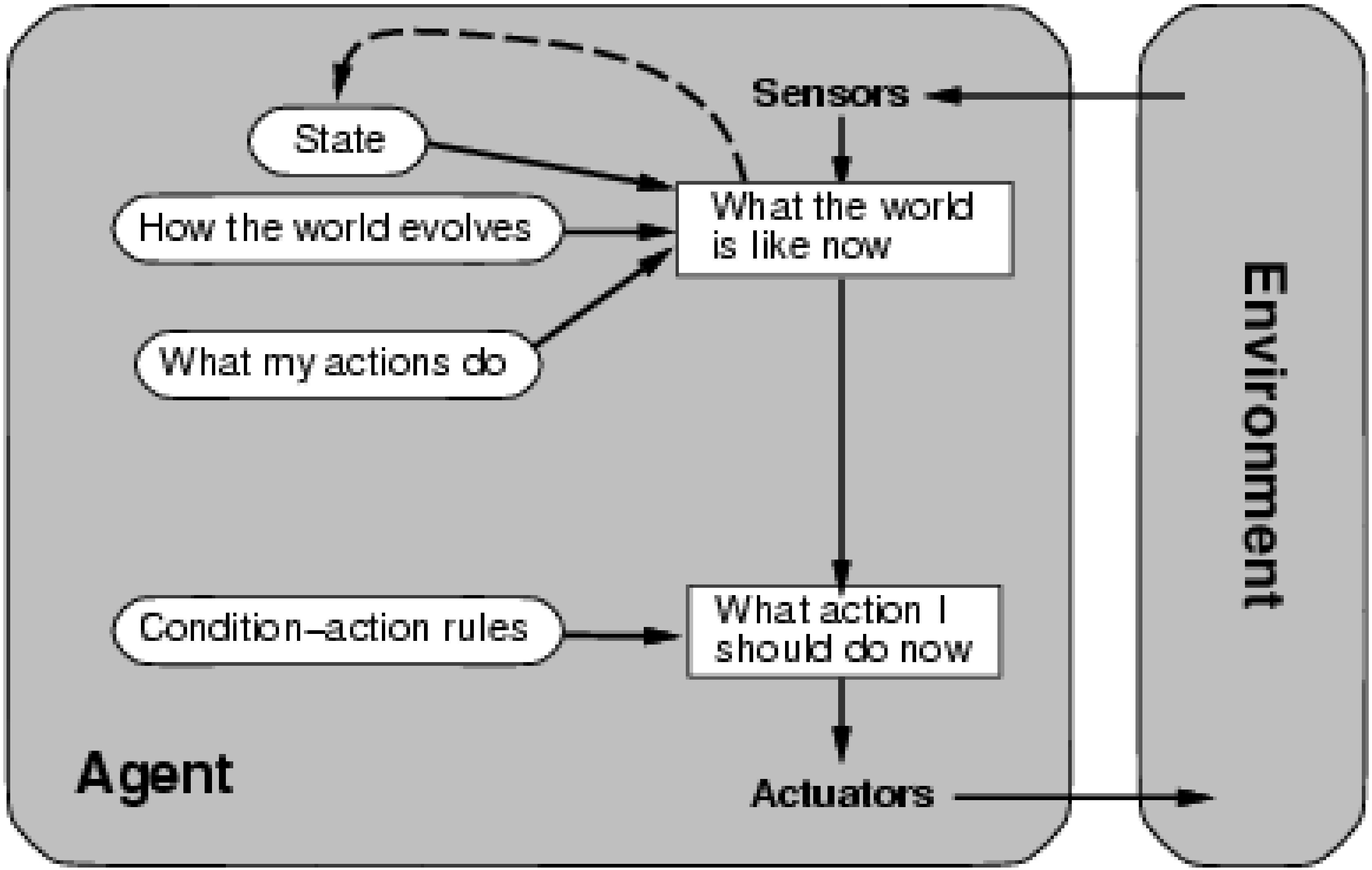
Reflex Agents

- Essentially a function f(s) = a
  - Accepts a state
  - Outputs an action

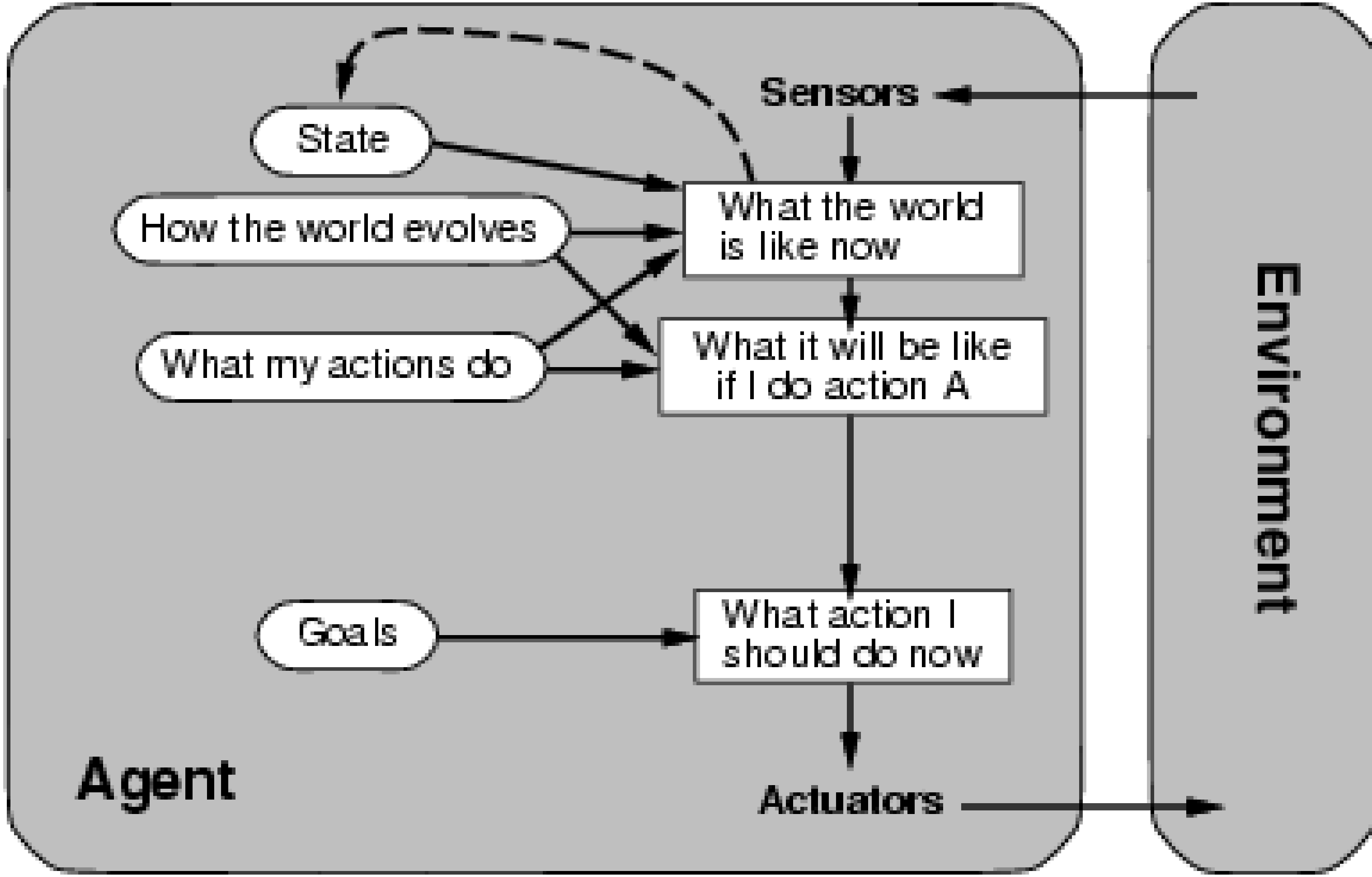
Simple Reflex Agents



Model-based Reflex Agents



Goal-based agents



Goal-based Agents

- Have a concept of **the future**
- Can consider impact of action on future states
- Capable of comparing desirability of states relative to a **goal**
- Agent’s job: identify best course of actions to reach goal
- Can be accomplished by **searching** through possible states and actions

A Problematic Perspective

- Think of agent as looking for a **solution** to a specific **problem**
- **Problem** consists of:
  - Current state
  - A **goal**
  - Possible courses of action
- **Solution** consists of:
  - Ordered list of actions

Current Assumptions

- States are **atomic**
  - Indivisible black boxes
  - As opposed to factored or structured
- Future observations will not alter agent's actions
  - Solution does not change over time

General Problem Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH( problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Crafting a Goal

- Agent creates goal based on:
  - Current environment
  - Evaluation metrics
  - Where do these come from?
- How does a goal help?
  - Guidance when state is ambiguous
  - Narrows down potential choices

Crafting a Problem

- Current state – We're here
- Goal state(s) – Over there
- How do you transition from A to B?
- **Problem**: Actions and states to consider en route to goal
- Set of all possible states is known as the **state space**

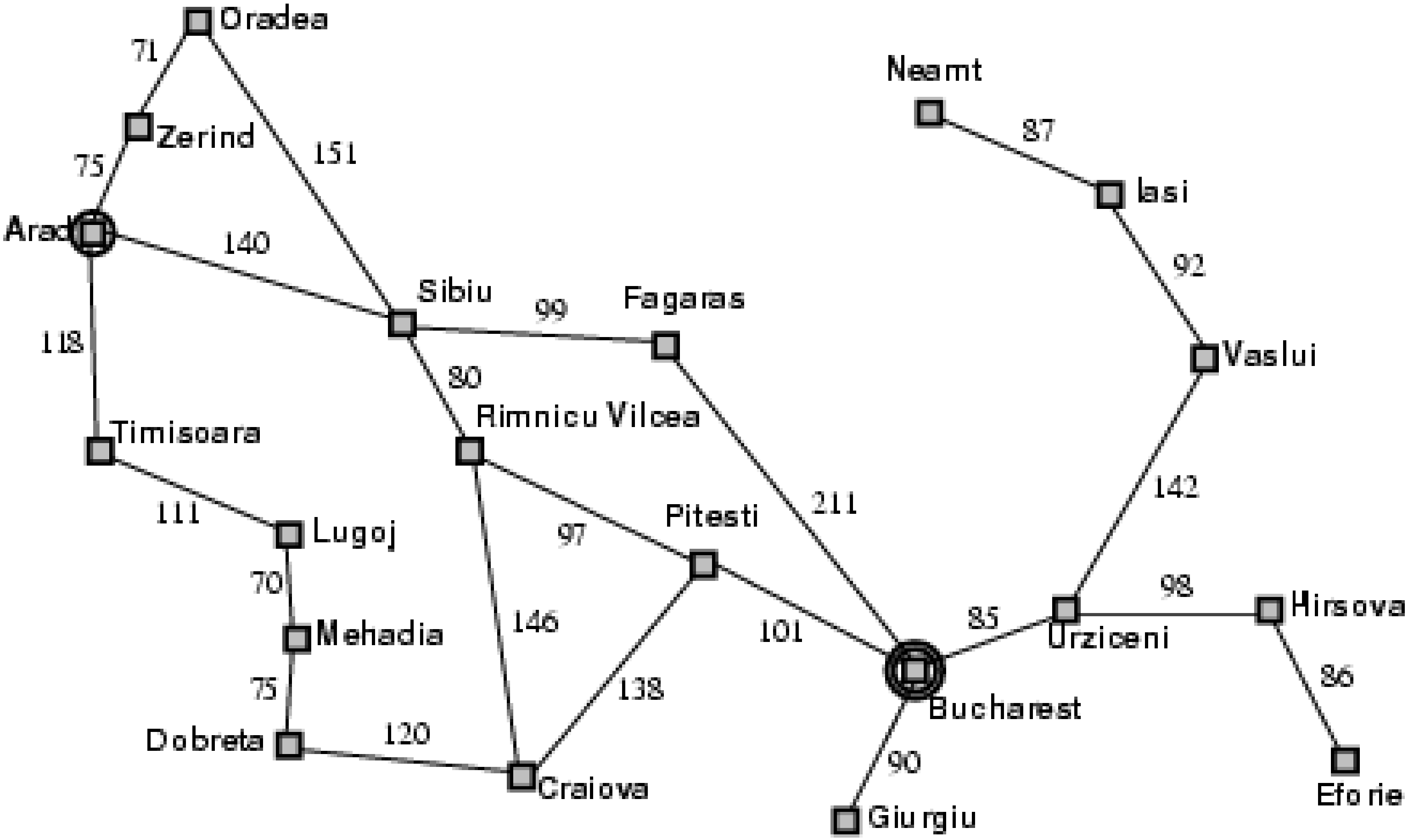
Crafting a Problem

- Actions should be of suitable granularity
  - "Take a step"
  - "Walk down block"
  - "Drive to city"
  - "Travel to star system"
- Actions should pertain to goal
- Problem must be well defined for successful agents

Romanian Vacation Example

- On vacation in Romania
  - Currently in Arad
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - Want to be in Bucharest
- Formulate problem:
  - States: various cities
  - Actions: drive between cities
- Find solution:
  - Sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



AI World Problems

- Five parts:
- Initial state
  - (in arad)
- Applicable actions (given state)
  - (go sibiu) (go Timisoara) (go zerind)
- Transition model: state + action = new state
  - (result (in arad) (go zerind)) = (in zerind)

AI World Problems

- Five parts:
- Goal test – Did I win yet?
  - Condition (implicit) or set of states (explicit)
  - {(in bucharest)}
- Path cost
  - Agent assigns to action based on **performance measure**
  - (cost (in arad) (go zerind) (in zerind)) = 75 kilometers

The Devil is in the Details

- Isn’t Simand between Zerind and Arad?
- Did you have the air conditioner on?
- Restroom stops?
- Personal growth during trip

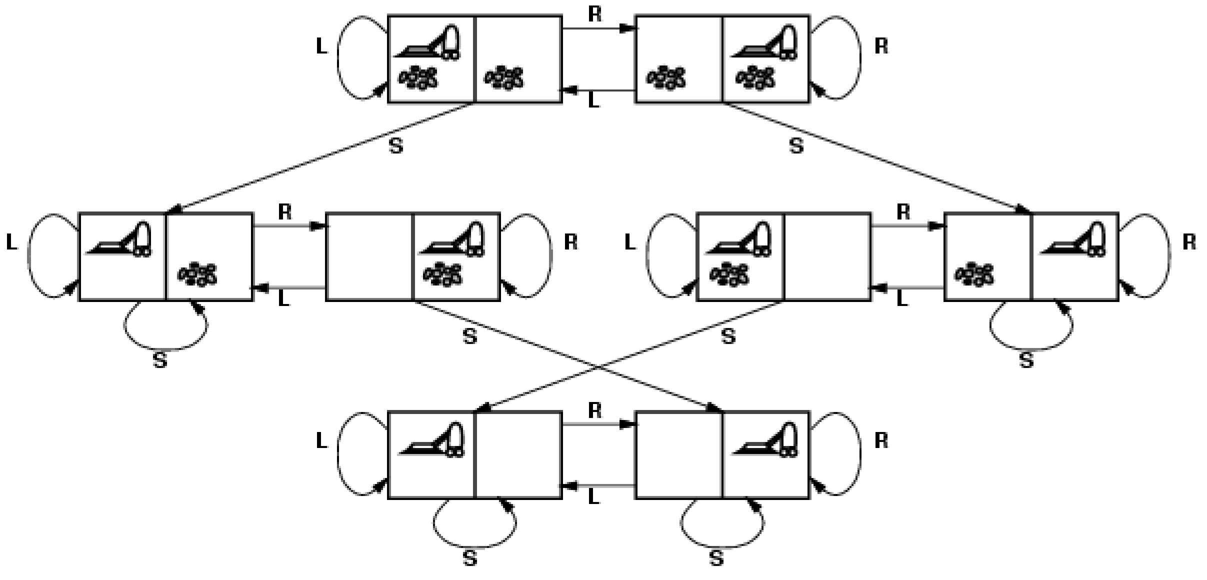
Abstraction is your Friend

- The real world is absurdly complex
  - State space must be **abstracted** for problem solving
- **Abstract** away things which:
  - Are irrelevant to problem at hand
  - Don’t affect validity of solution

Selecting a State Space

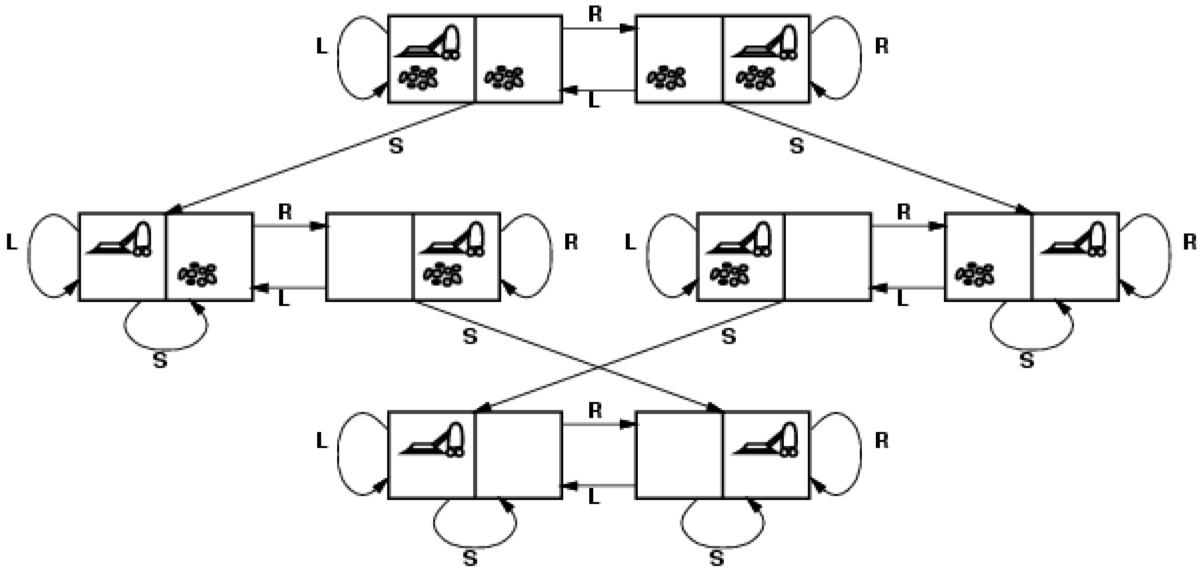
- (Abstract) state = Set of real states
- (Abstract) action = Complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state “in Arad” must get to some real state “in Zerind”
- Abstract solution will represent a set of detailed solutions
  - Set of real paths that are solutions in the real world
- Good abstraction makes problems “easier”

Back to Vacuum World



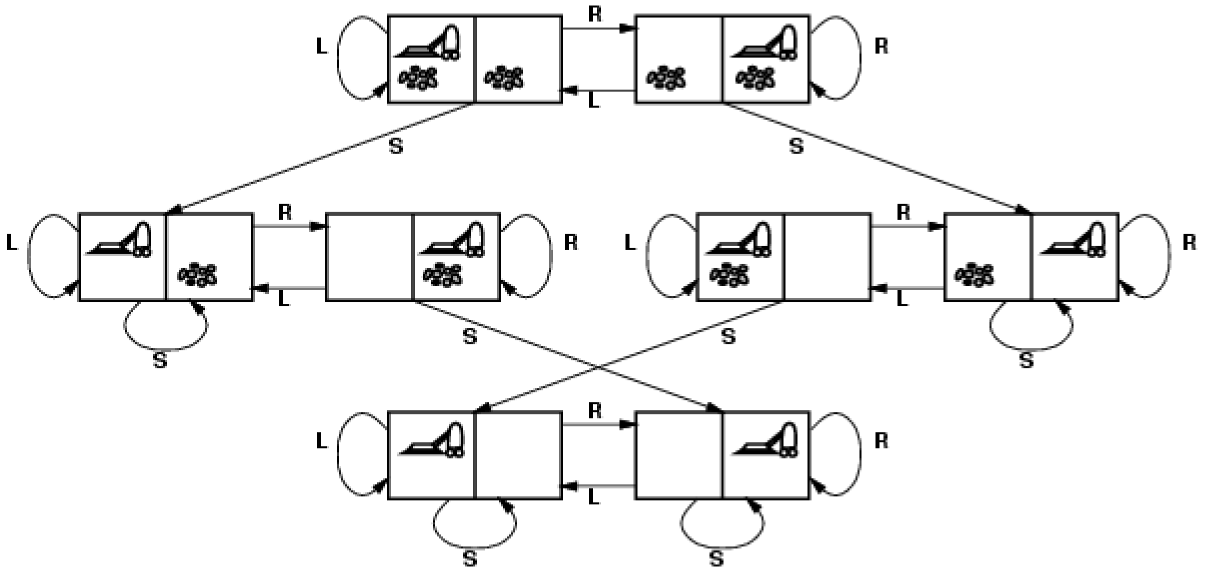
- States?
- Actions?
- Goal test?
- Path cost?

Back to Vacuum World



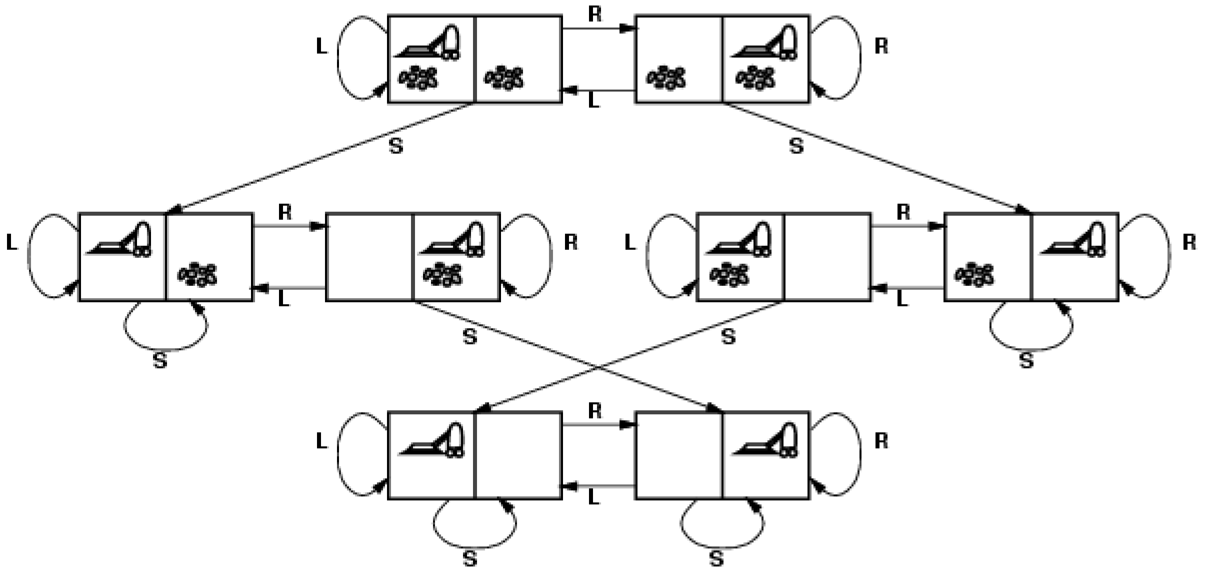
- States? Location of agent, location(s) of dirt
- Actions?
- Goal test?
- Path cost?

Back to Vacuum World



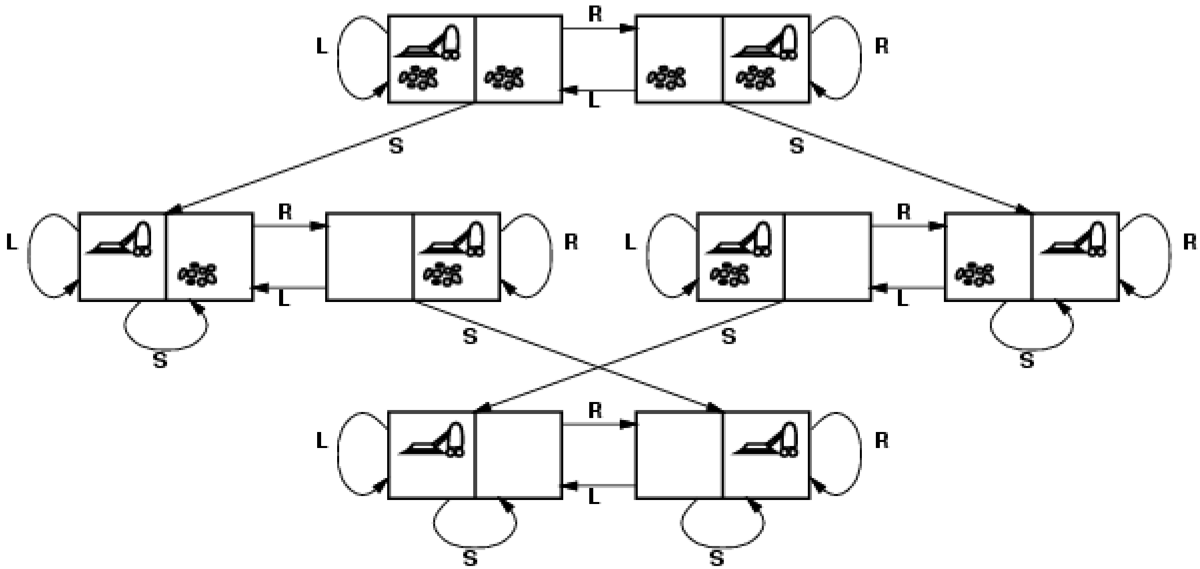
- States? Location of agent, location(s) of dirt
- Actions? Move in direction, suck
- Goal test?
- Path cost?

Back to Vacuum World



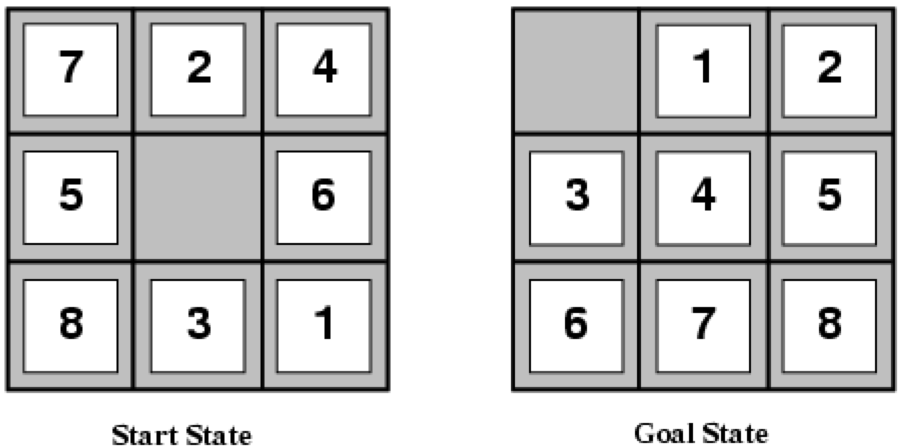
- States? Location of agent, location(s) of dirt
- Actions? Move in direction, suck
- Goal test? All clean?
- Path cost?

Back to Vacuum World



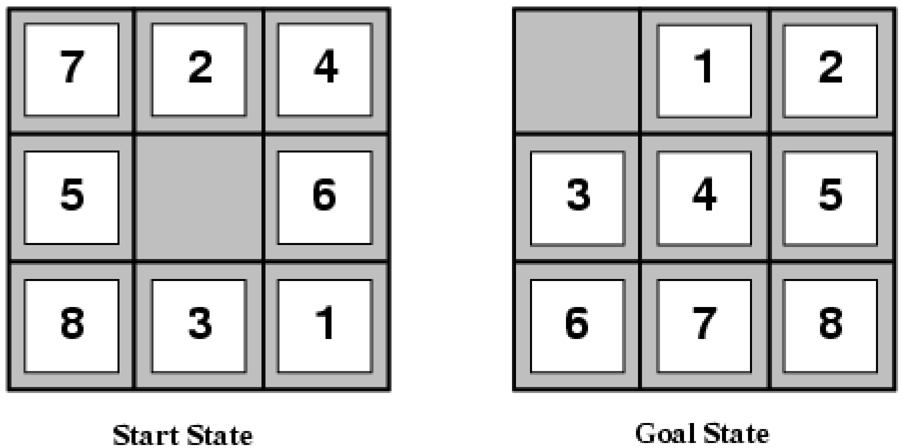
- States? Location of agent, location(s) of dirt
- Actions? Move in direction, suck
- Goal test? All clean?
- Path cost? 1/action

Example: The 8-puzzle



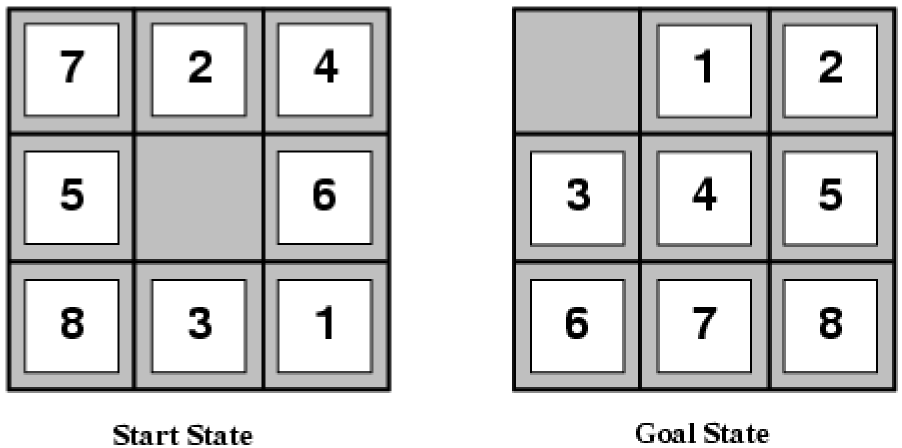
- States?
- Actions?
- Goal test?
- Path cost?

Example: The 8-puzzle



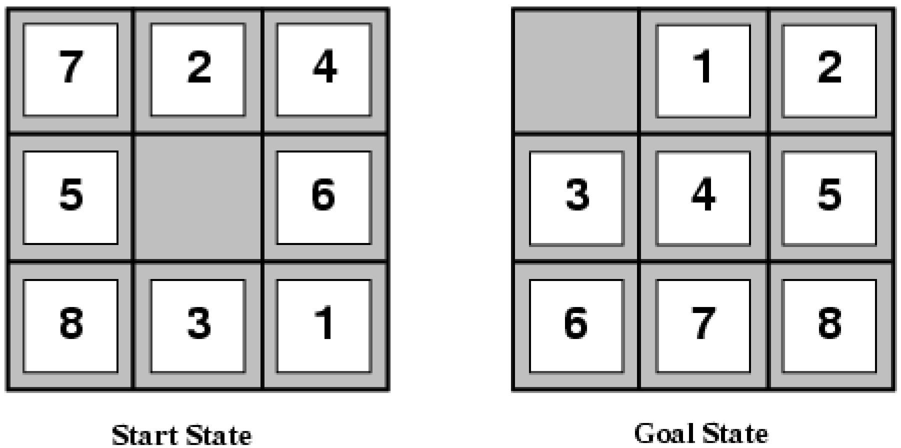
- States? Tile locations
- Actions?
- Goal test?
- Path cost?

Example: The 8-puzzle



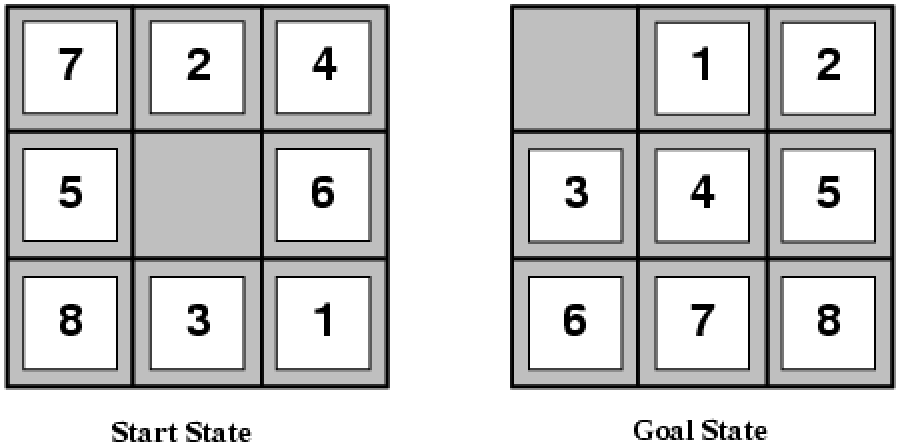
- States? Tile locations
- Actions? Move blank
- Goal test?
- Path cost?

Example: The 8-puzzle



- States? Tile locations
- Actions? Move blank
- Goal test? Tiles in (blank, 1, 2,3,…) order
- Path cost?

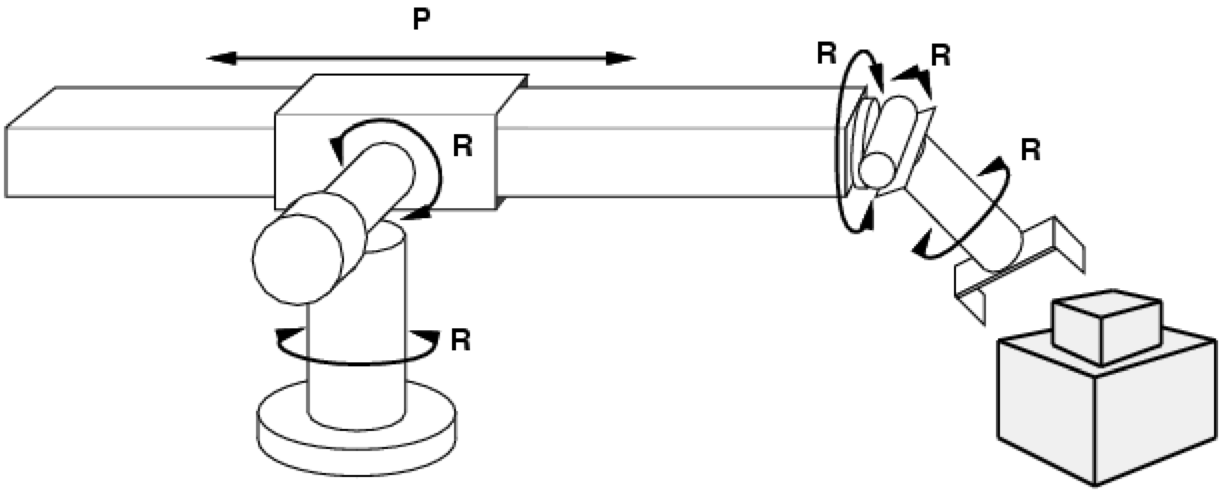
Example: The 8-puzzle



- States? Tile locations
- Actions? Move blank
- Goal test? Tiles in (blank, 1, 2,3,…) order
- Path cost? 1/move

[Note: optimal solution of *n*-Puzzle family is NP-hard]

Example: robotic assembly



- States?: real-valued coordinates of robot joint angles parts of the object to be assembled
- Actions?: continuous motions of robot joints
- Goal test?: complete assembly
- Path cost?: time to execute

What Does This Have To Do with Search?

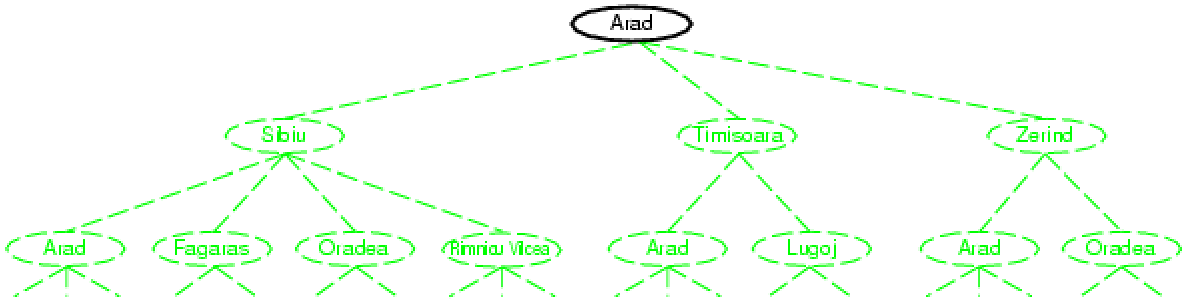
- Created a problem, need to create a solution
  - Recall: A solution is a sequence of actions
- Form a **search tree**
  - Root: Start state
  - Branches: Actions
  - Nodes: Resultant actions
- General algorithm:
  - Are we in goal state?
  - Expand current state by exploring each potential action
  - Choose which state to explore further
    - Easier said than done!

Tree Search Algorithms

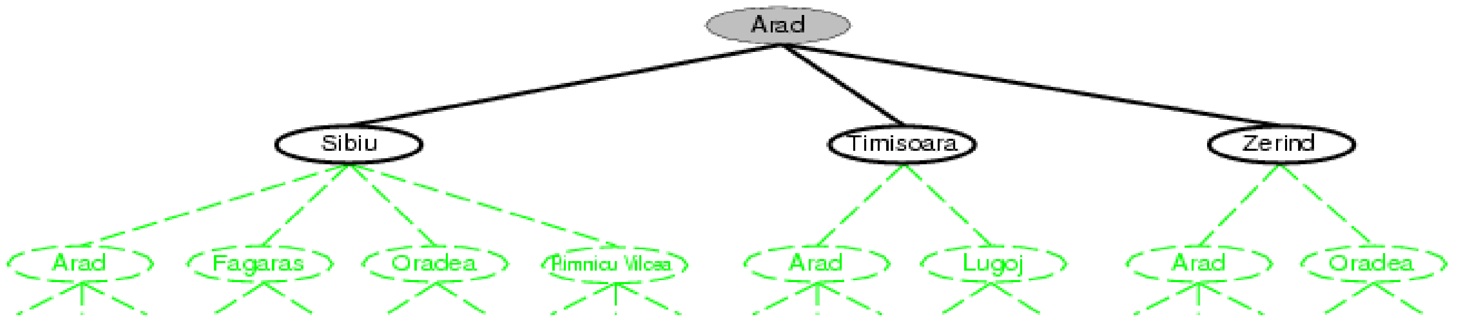
- Core concept:
  - Exploration of state space by generating successors of already-explored states (a.k.a. **expanding** states)

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

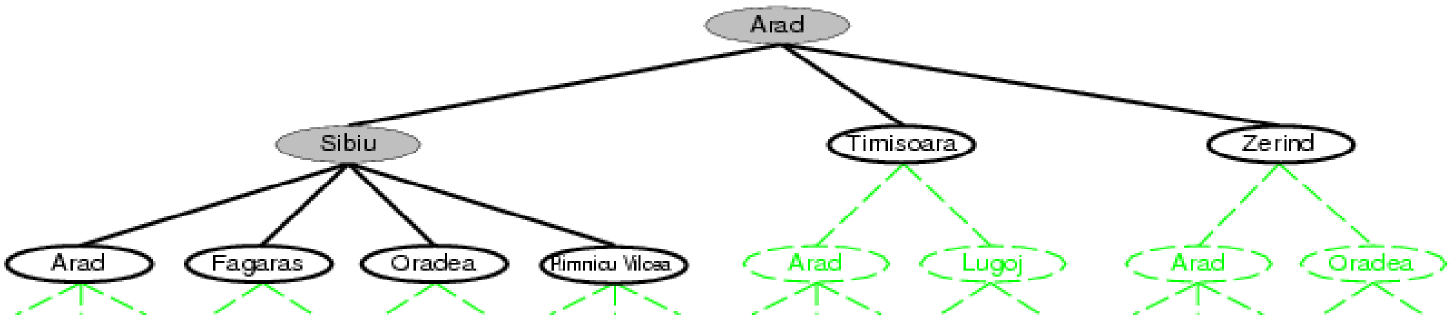
Tree Search Example



Tree Search Example



Tree Search Example



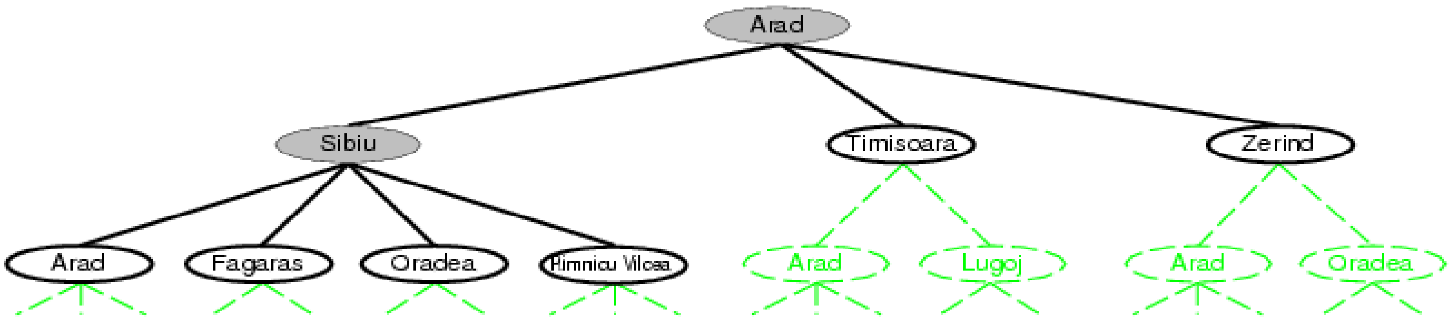
Implementation: General Tree Search

function TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
  loop do
    if *fringe* is empty **then return** failure
    node ← REMOVE-FRONT(*fringe*)
    if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
    fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

---

function EXPAND(*node*, *problem*) **returns** a set of nodes
  successors ← the empty set
  for each *action*, *result* **in** SUCCESSOR-FN[*problem*](STATE[*node*]) **do**
    s ← a new NODE
    PARENT-NODE[*s*] ← *node*; ACTION[*s*] ← *action*; STATE[*s*] ← *result*
    PATH-COST[*s*] ← PATH-COST[*node*] + STEP-COST(*node*, *action*, *s*)
    DEPTH[*s*] ← DEPTH[*node*] + 1
    add *s* to *successors*
  return *successors*

Tree Search Example



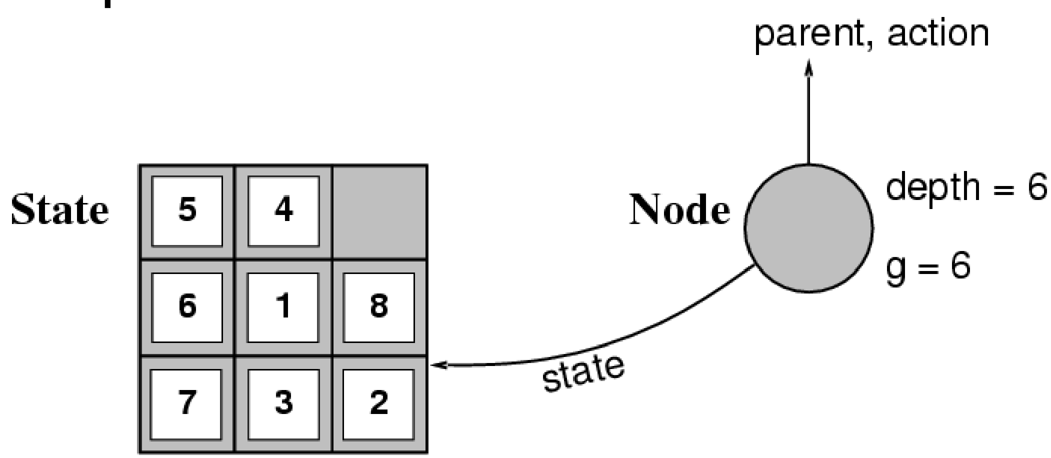
Anything odd here?

Search Tree Nuanaces

- States in search tree may repeat themselves
- Loopy paths
  - State A -> State B -> State A
- Redundant Paths
  - State A -> State Z
  - State A -> State B -> State C -> State D -> … -> State Z
- Solution: turn tree search into graph search by tracking redundant paths via an explored list
  - Starts out empty
  - Add node after goal test
  - Only expand node if not explored

Implementation: States vs. Search Tree Nodes

- A **state** is a (representation of) a physical configuration
- A **node** is a data structure constituting part of a search tree which includes **state**, parent node, action, path cost *g(x)*, and depth



- The expand function creates new nodes, filling in the various fields and using the successor function of the problem to create the corresponding state

Search Strategies

- A search strategy is defined by picking the **order** of **node expansion**
- Strategies are evaluated along the following dimensions:
  - completeness: Always find a solution (if one exists)?
  - time complexity: Number of nodes generated
  - space complexity: Maximum number of nodes in memory
  - optimality: Always find a least-cost solution?
- Time and space complexity are measured in terms of
  - ***b**:* Maximum branching factor of the search tree
  - ***d**:* Depth of the least-cost solution
  - ***m***: Maximum depth of the state space (may be ∞)
- Total cost: search cost + path cost
  - How to add apples and oranges?

Uninformed Search Strategies

- **Uninformed** search strategies use only the information available in the problem definition
- No analysis or knowledge of states, only:
  - Generate successor nodes
  - Check for goal state
- Specifically, no comparison of states

Uninformed Search Strategies

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Uninformed Search Strategies

- Breadth-first search
  - Expand shallowest node
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Uninformed Search Strategies

- Breadth-first search
  – Expand shallowest node
- Uniform-cost search
  – Expand least cost node
- Depth-first search
- Depth-limited search
- Iterative deepening search

Uninformed Search Strategies

- Breadth-first search
  - Expand shallowest node
- Uniform-cost search
  - Expand least cost node
- Depth-first search
  - Expand deepest node
- Depth-limited search
- Iterative deepening search

Uninformed Search Strategies

- Breadth-first search
  - Expand shallowest node
- Uniform-cost search
  - Expand least cost node
- Depth-first search
  - Expand deepest node
- Depth-limited search
  - Depth-first with depth limit
- Iterative deepening search

Uninformed Search Strategies

- Breadth-first search
  - Expand shallowest node
- Uniform-cost search
  - Expand least cost node
- Depth-first search
  - Expand deepest node
- Depth-limited search
  - Depth-first with depth limit
- Iterative deepening search
  - Depth-limited with increasing limit

Summary of Uninformed Search Algorithms

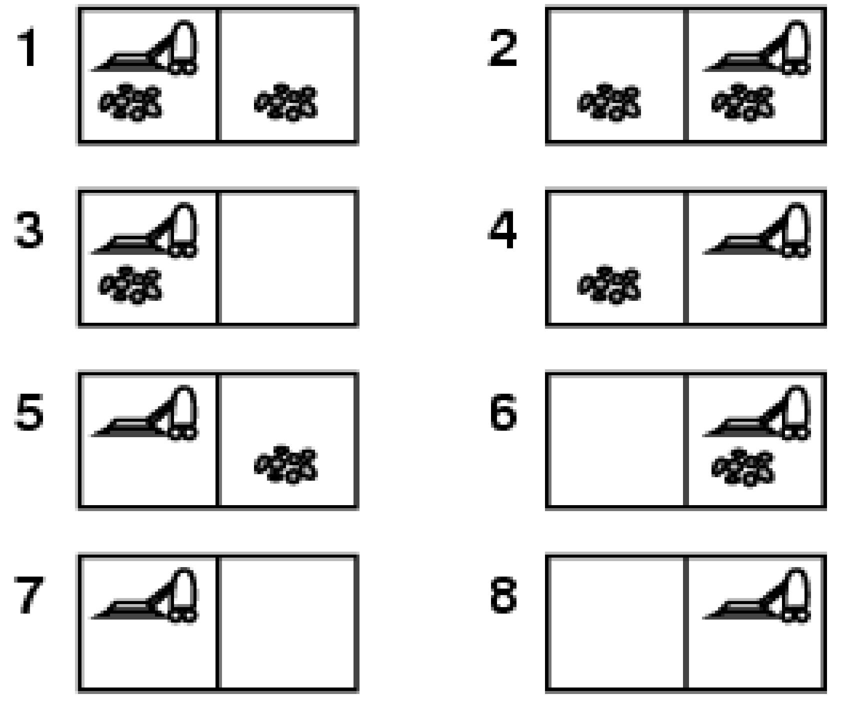
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Problem Types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in
  - Solution is a sequence
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is
  - Solution remains a sequence
- Nondeterministic and/or partially observable → contingency problem
  - Percepts provide new information about current state
  - Often interleave search and execution
  - Solution may require conditionals
- Unknown state space → exploration problem

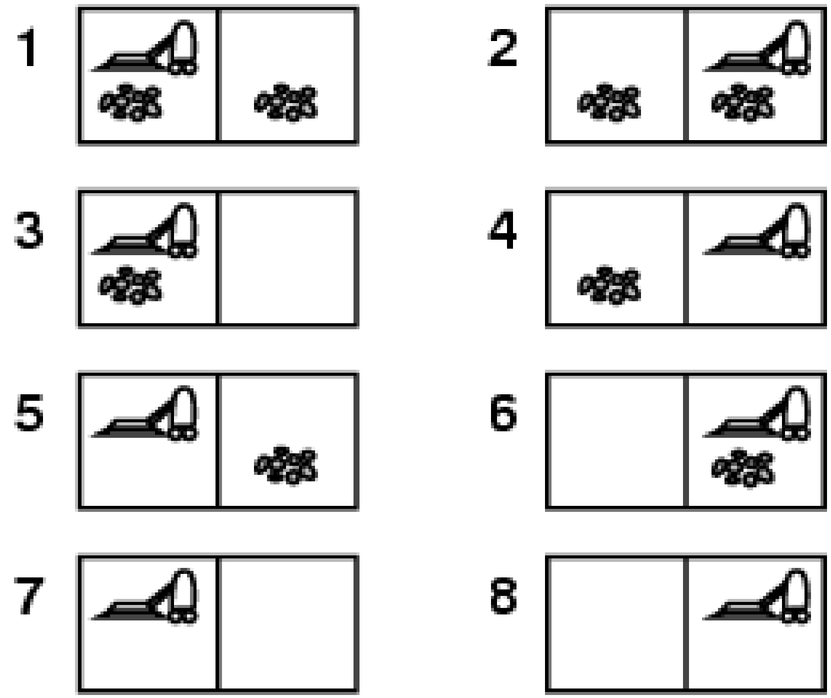
Example: Vacuum World

- Single-state, start in #5
- **Solution?**



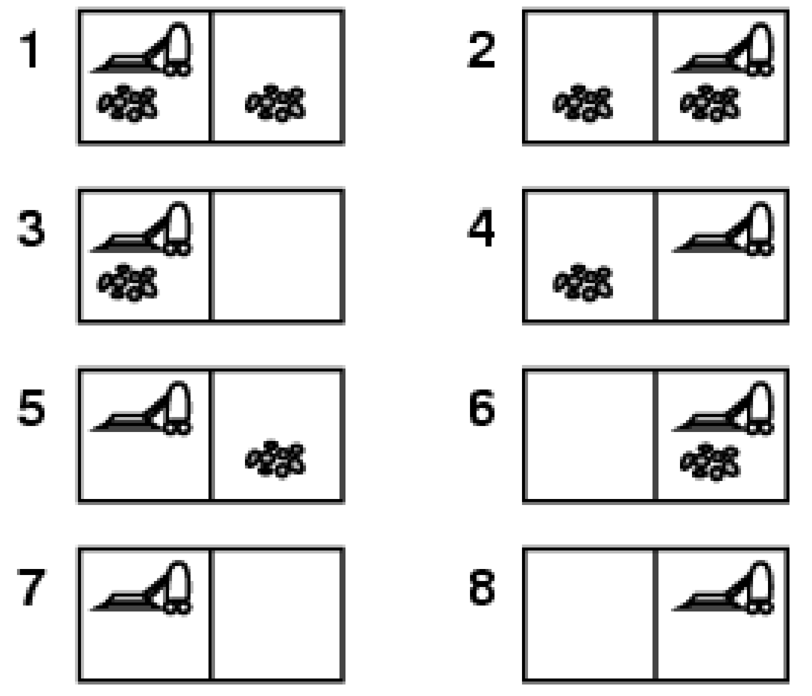
Example: Vacuum World

- Single-state, start in #5
- **Solution?**
- *[Right, Suck]*



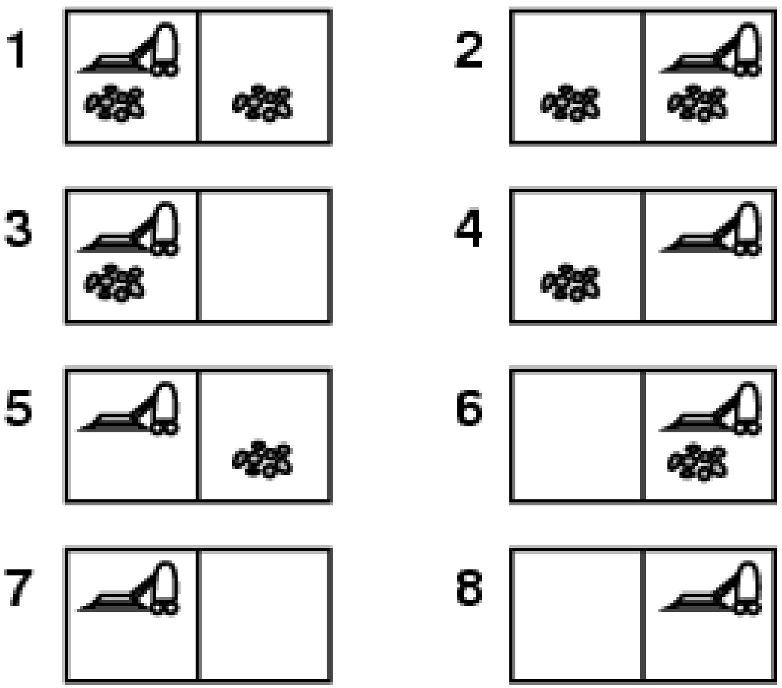
Example: Vacuum World

- Sensorless, start in {*1,2,3,4,5,6,7,8*}
- e.g., *Right* goes to {*2,4,6,8*}
- **Solution?**



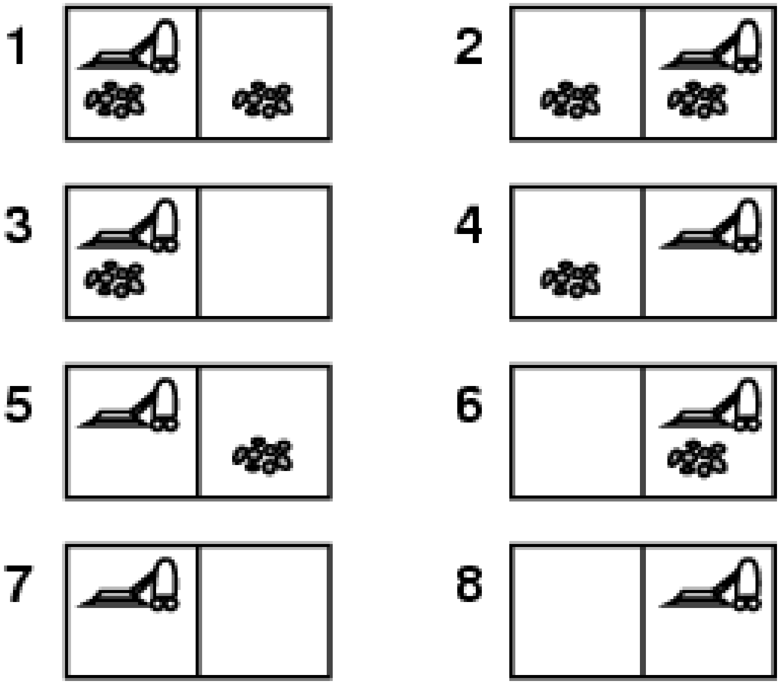
Example: Vacuum World

- Sensorless, start in {*1,2,3,4,5,6,7,8*}
- e.g., *Right* goes to {*2,4,6,8*}
- **Solution?**
- *[Right,Suck,Left,Suck]*



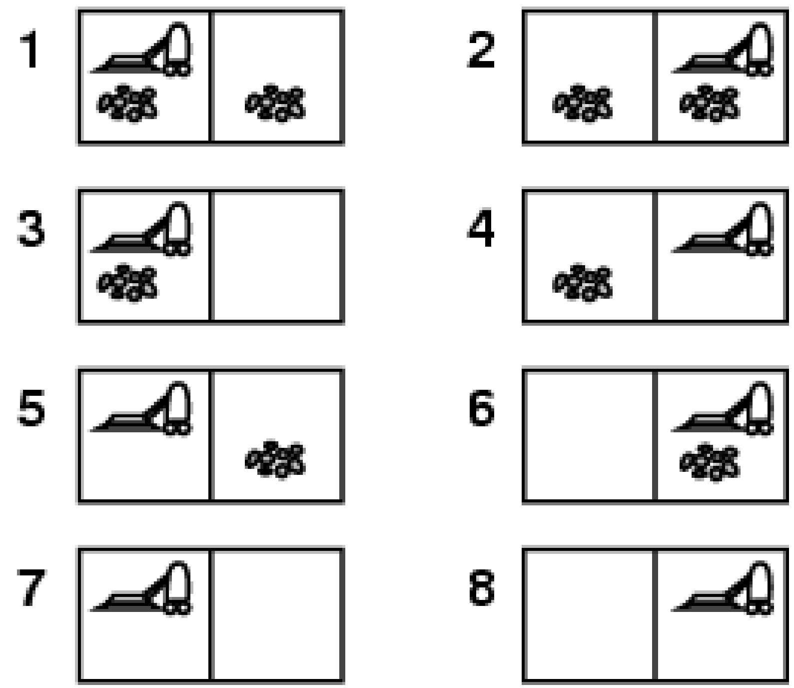
Example: Vacuum World

- Nondeterministic: *Suck* may dirty a clean carpet
- Partially observable:
  - Location
  - dirt at current location.
- Percept: *[L, Clean],*
- i.e., start in #5 or #7
- **Solution?**



Example: Vacuum World

- Nondeterministic: *Suck* may dirty a clean carpet
- Partially observable:
  - – Location
  - – dirt at current location.
- Percept: *[L, Clean],*
- i.e., start in #5 or #7
- **Solution?**
  - *[Right, **if** dirt **then** Suck]*



Summary

- Goals help agents solve problems
- Helpful to think of state space as a searchable tree
- General problem solving agent algorithm:
  - Observe environment
  - Construct goal
  - Construct problem (= start + options + goal)
  - Search problem for solution (= set of actions)
- Need to ignore details to turn an overwhelming real set of states into a manageable abstract state
- Order in which options are searched is crucial
  - Variety of simple methods

Up Next

- Order in which options are searched is crucial
- Variety of uninformed methods
  - Simple
  - Perform horribly on problems with exponential complexity
- What if we had a way to compare nodes that didn't contain the goal state…?
  - How would it be useful?
  - How would you go about that?
  - Stay tuned!