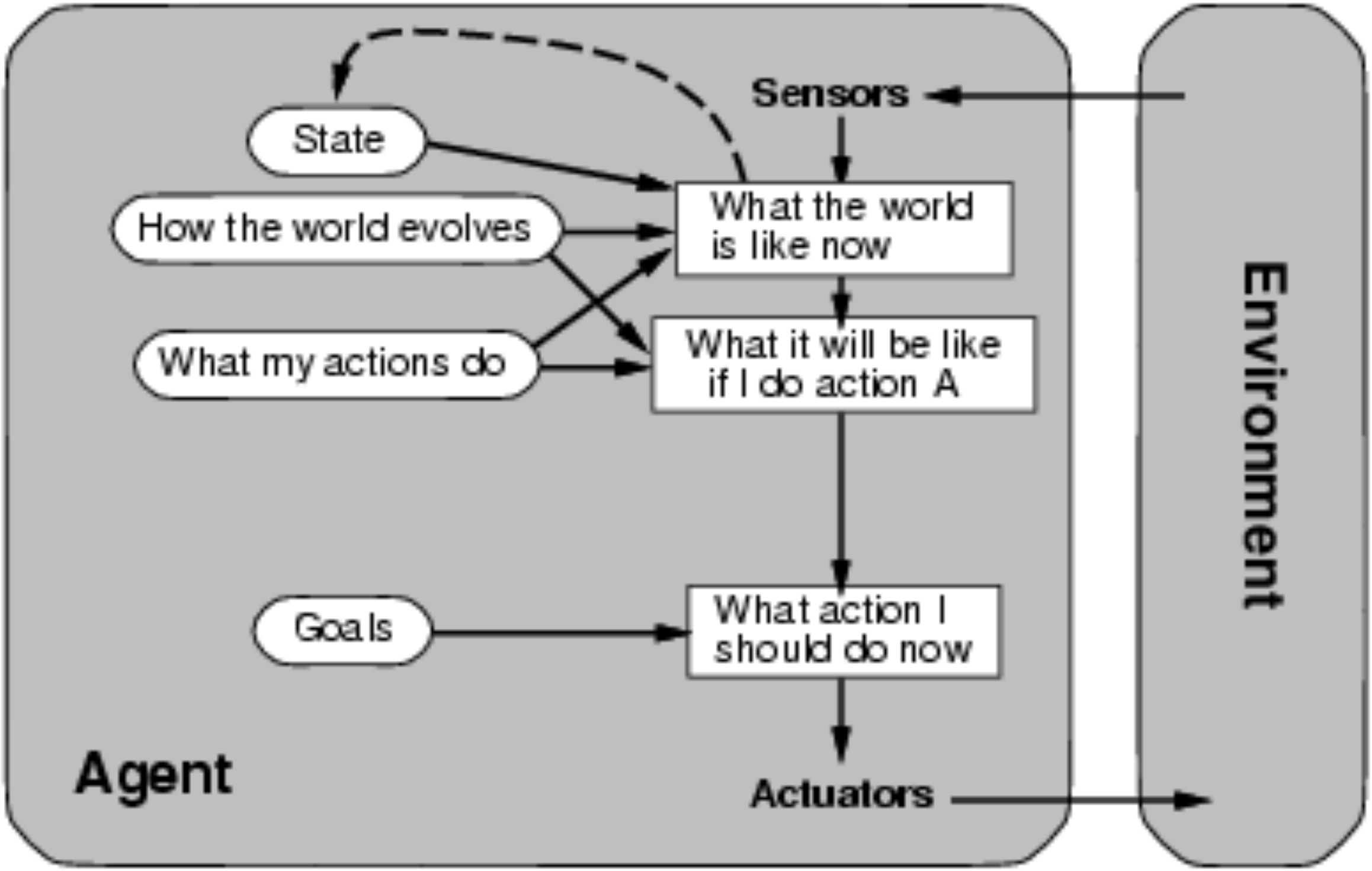
Chapter 2

Problem solving

Basanta Joshi, PhD
basanta@ioe.edu.np
Lecture notes can be downloaded from
www.basantajoshi.com.np

Goal Based Agents



Goal Based Agents

- Assumes the problem environment is:
  - Static
    - The plan remains the same
  - Observable
    - Agent knows the initial state
  - Discrete
    - Agent can enumerate the choices
  - Deterministic
    - Agent can plan a sequence of actions such that each will lead to an intermediate state

- The agent carries out its plans with its eyes closed
  - Certain of what's going on
  - Open loop system

Well Defined Problems and Solutions

A problem

- Initial state
- Actions and Successor Function
- Goal test
- Path cost

Problem Solving

- **Problem solving is fundamental to many AI-based applications.**

  There are two types of problems.

  - The Problems like, computation of the sine of an angle or the square root of a value. These can be solved through the use of deterministic procedure and the success is guaranteed.
  - In the real world, very few problems lend themselves to straightforward solutions.

  Most real world problems can be solved only by searching for a solution.

  AI is concerned with these type of problems solving.

- **Problem solving is a process** of generating solutions from observed data.
  - a problem is characterized by a set of goals,
  - a set of objects, and
  - a set of operations.

  These could be ill-defined and may evolve during problem solving.

Problem Solving

- **Problem space** is an abstract space.
  - A problem space encompasses all *valid states* that can be generated by the application of any combination of *operators* on any combination of *objects*.
  - The problem space may contain one or more *solutions*.

  Solution is a combination of operations and objects that achieve the goals.

- **Search** refers to the search for a solution in a problem space.
  - Search proceeds with different types of *search control strategies*.
  - The *depth-first search and breadth-first search* are the two common search strategies.

Problem Solving

To build a system to solve a particular problem, we need to

- Define the problem precisely – find input situations as well as final situations for acceptable solution to the problem.
- Analyze the problem – find few important features that may have impact on the appropriateness of various possible techniques for solving the problem.
- Isolate and represent task knowledge necessary to solve the problem
- Choose the best problem solving technique(s) and apply to the particular problem.

Problem Solving

Problem Definitions :

A *problem* is defined by its *elements* and their *relations*.

To provide a formal description of a problem, we need to do following:

a. Define a *state space* that contains all the possible configurations of the relevant objects, including some impossible ones.

b. Specify one or more states, that describe possible situations, from which the problem-solving process may start. These states are called *initial states*.

c. Specify one or more states that would be acceptable solution to the problem. These states are called *goal states*.

d. Specify a set of *rules* that describe the *actions* (*operators*) available.

Problem Solving

The problem can then be solved by using the *rules*, in combination with an appropriate *control strategy*, to move through the *problem space* until a *path* from an *initial state* to a *goal state* is found.

This process is known as ***search***.

- Search is fundamental to the problem-solving process.
- Search is a general mechanism that can be used when more direct method is not known.
- Search provides the framework into which more direct methods for solving subparts of a problem can be embedded.

A very large number of AI problems are formulated as search problems.

Problem Space

A *problem space* is represented by directed *graph*, where *nodes* represent *search state* and *paths* represent the *operators* applied to change the *state*.

To simplify a search algorithms, it is often convenient to logically and programmatically represent a problem space as a ***tree***. A tree usually decreases the complexity of a search at a ***cost***. Here, the cost is due to duplicating some nodes on the tree that were linked numerous times in the graph; e.g., node **B** and node **D** shown in example below.

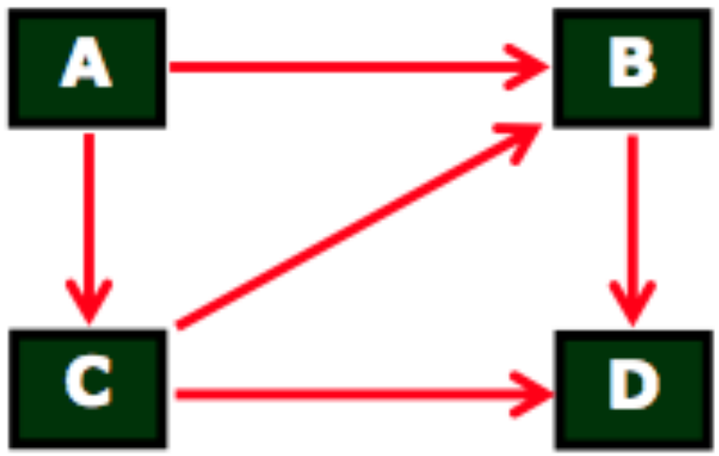
A **tree** is a graph in which any two vertices are connected by exactly one path. Alternatively, any connected graph with no cycles is a tree.

Problem Space

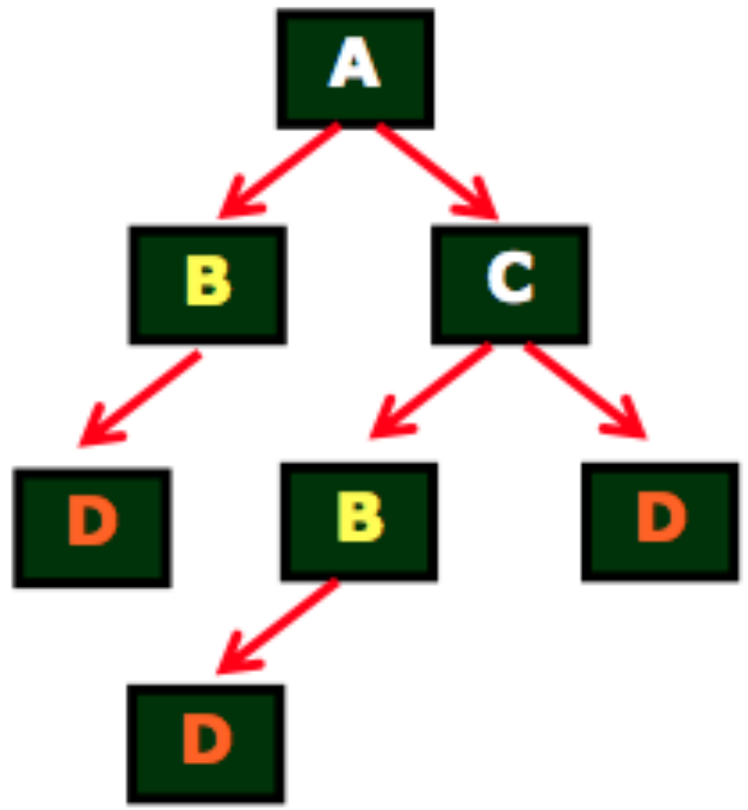
A **tree** is a graph in which any two vertices are connected by exactly one path. Alternatively, any connected graph with no cycles is a tree.

Examples

Graph



Trees



Problem Solving

The term Problem Solving relates analysis in AI. Problem solving may be characterized as a *systematic search* through a range of possible actions to reach some predefined goal or solution. Problem-solving methods are categorized as *special purpose* and *general purpose.*

- **Special-purpose method** is tailor-made for a particular problem, often exploits very specific features of the situation in which the problem is embedded.
- **General-purpose method** is applicable to a wide variety of problems. One general-purpose technique used in AI is "means-end analysis". It is a step-by-step, or incremental, reduction of the difference between current state and final goal.

Examples : Tower of Hanoi puzzle

Pr

- For a Robot this might consist of PICKUP, PUTDOWN, MOVEFORWARD, MOVEBACK, MOVELEFT, and MOVERIGHT—until the goal is reached.
- Puzzles and Games have explicit rules : e.g., the Tower of Hanoi puzzle.

Tower of Hanoi puzzle.

- This puzzle may involves a set of rings of different sizes that can be placed on three different pegs.
- The puzzle starts with the rings arranged as shown in Fig. (a)
- The goal of this puzzle is to move them all as to Fig. (b)
- Condition : Only the top ring on a peg can be moved, and it may only be placed on a smaller ring, or on an empty peg.

In this Tower of Hanoi puzzle : Situations encountered while solving the problem are described as *states*. The set of all possible configurations of rings on the pegs is called *problem space*.

State

A *state* is a representation of elements at a given moment. A problem is defined by its *elements* and their *relations*.

At each instant of a problem, the elements have specific descriptors and relations; the *descriptors* tell - how to select elements ?

Among all possible states, there are two special states called :

- *Initial state* is the start point
- *Final state* is the goal state

State

State Change: Successor Function

A *Successor Function* is needed for state change.

The successor function moves one state to another state.

Successor Function :

- Is a description of possible actions; a set of operators.
- Is a transformation function on a state representation, which converts that state into another state.
- Defines a relation of accessibility among states.
- Represents the conditions of applicability of a state and corresponding transformation function

State space

A *State space* is the set of all states reachable from the *initial state*.

Definitions of terms :

- A *state space* forms a *graph* (or map) in which the *nodes* are states and the *arcs* between nodes are actions.
- In *state space*, a *path* is a sequence of states connected by a sequence of actions.
- The *solution* of a problem is part of the map formed by the *state space*.

Structure of State space

The *Structures* of *state space* are *trees* and *graphs*.

- Tree is a hierarchical structure in a graphical form; and
- Graph is a non-hierarchical structure.

◇ **Tree** has only one path to a given node;

i.e., a *tree* has one and only one path from any point to any other point.

◇ **Graph** consists of a set of nodes (vertices) and a set of edges (arcs). Arcs establish relationships (connections) between the nodes;

i.e., a graph has several paths to a given node.

◇ **operators** are directed *arcs* between nodes.

Search process explores the *state space*. In the worst case, the search explores all possible *paths* between the *initial state* and the *goal state*.

Problem solution

In the *state space*, a *solution is a path* from the *initial state* to a *goal state* or sometime just a *goal state*.

- A Solution cost function assigns a numeric cost to each path;
  It also gives the cost of applying the operators to the states.

- A Solution quality is measured by the path cost function; and
  An optimal solution has the lowest path cost among all solutions.

- The solution may be any or optimal or all.

- The importance of cost depends on the problem and the type of solution asked.

Problem solution

A problem consists of the description of :

- *current state* of the world,
- *actions* that can transform one state of the world into another,
- *desired state* of the world.

- **State space** is defined explicitly or implicitly

  A state space should describe everything that is needed to solve a problem and nothing that is not needed to the solve the problem.

- **Initial state** is start state

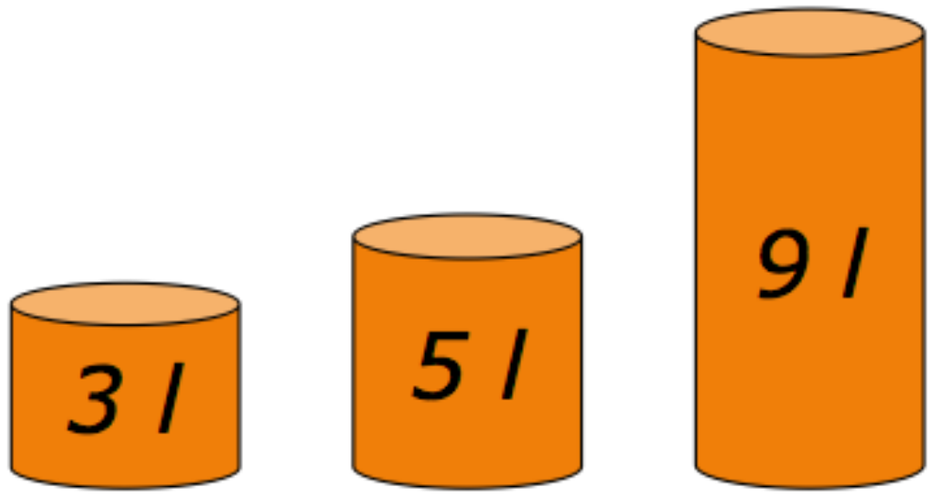
- **Goal state** is the conditions it has to fulfill
  - A description of a desired state of the world;
  - The description may be complete or partial.

- **Operators** are to change state
  - Operators do actions that can transform one state into another.
  - Operators consist of : Preconditions and Instructions;
    - Preconditions provide partial description of the state of the world that must be true in order to perform the action,
    - Instructions tell on how to create next state.
  - Operators should be as general as possible, to reduce their number.

- **Elements of the domain** has relevance to the problem
  - Knowledge of the starting point.
- **Problem solving** is finding solution
  - Finding an ordered sequence of operators that transform the current (start) state into a goal state;

- **Restrictions** are solution quality any, optimal, or all
  - Finding the shortest sequence, or
  - Finding the least expensive sequence defining cost , or
  - Finding any sequence as quickly as possible.

Example: Measuring Problem



Problem: Using these three buckets,
measure 7 liters of water.

Example: Measuring Problem

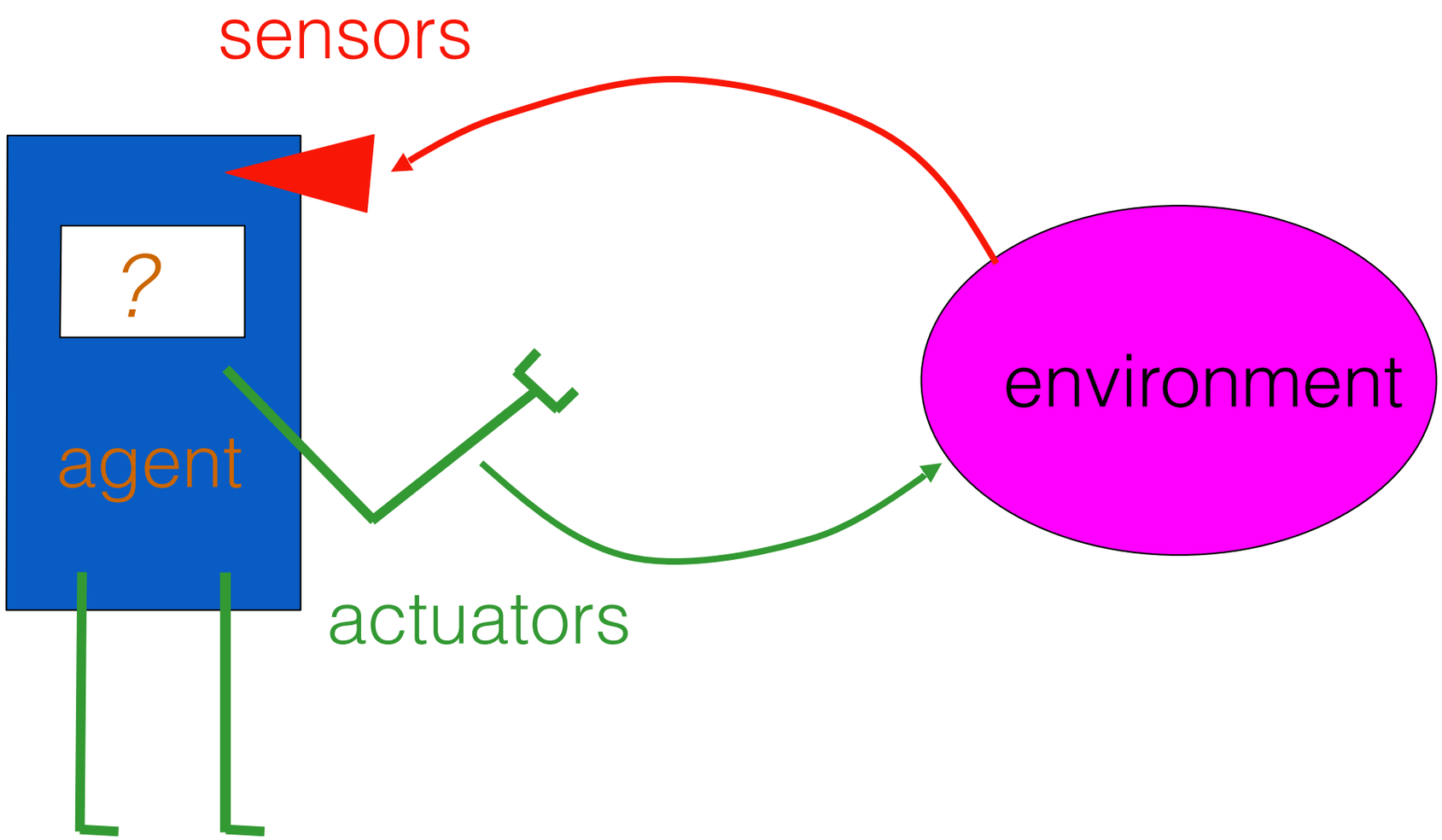
- **Solution 1:**

a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	7	**goal**

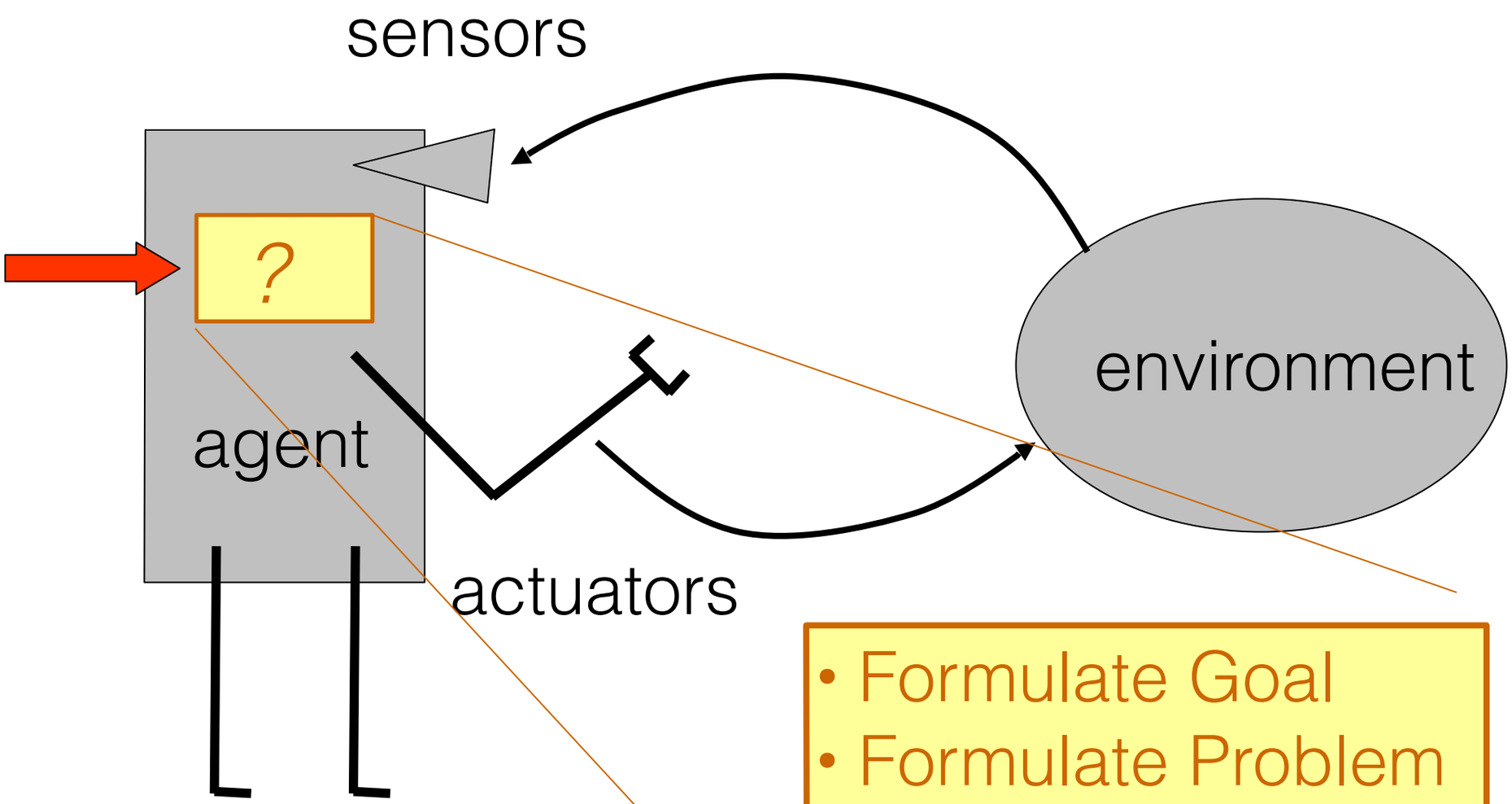
- **Solution 2:**

a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	
3	5	2	
3	0	7	**goal**

Problem-Solving Agent



Problem-Solving Agent



- Formulate Goal
- Formulate Problem
  - States
  - Actions
- Find Solution

Assumptions

- World States
- Actions as transitions between states
- Goal Formulation: A set of states
- Problem Formulation:

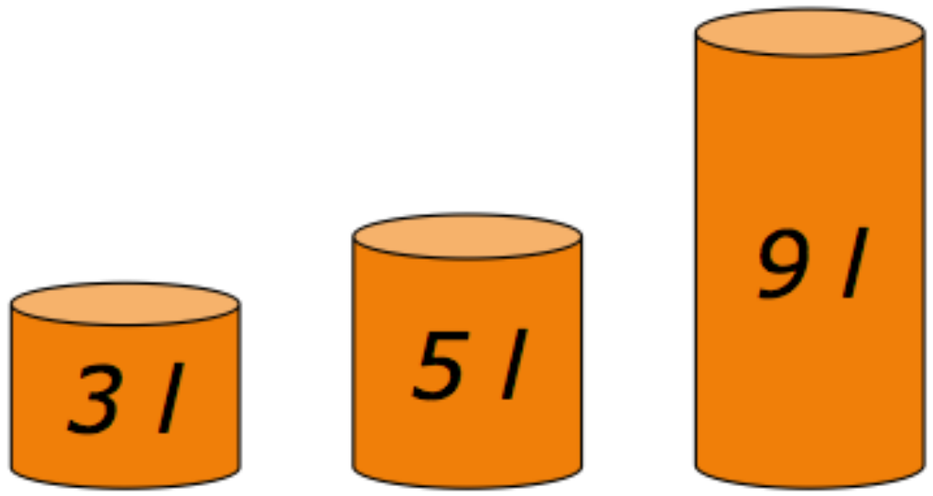
  The sequence of required actions to move from current state to a goal state

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)   // What is the current state?
    if seq is empty then
            goal ← FORMULATE-GOAL(state)   // From LA to San Diego (given curr. state)
            problem ← FORMULATE-PROBLEM(state, goal)   // e.g., Gas usage
            seq ← SEARCH( problem)
    action ← RECOMMENDATION(seq, state)
    seq ← REMAINDER(seq, state)   // If fails to reach goal, update
    return action
```

Note: This is *offline* problem-solving. *Online* problem-solving involves acting w/o complete knowledge of the problem and environment

Example: Measuring Problem



Problem: Using these three buckets,
measure 7 liters of water.

Example: Measuring Problem

Measure 7 liters of water using a 3-liter, a 5-liter, and a 9-liter buckets.

- **Formulate goal:** Have 7 liters of water in 9-liter bucket
- **Formulate problem:**
  - States: amount of water in the buckets
  - Operators: Fill bucket from source, empty bucket
- **Find solution:** sequence of operators that bring you from current state to the goal state

Environment types

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Operating System	Yes	Yes	No	No	Yes
Virtual Reality	Yes	Yes	Yes/no	No	Yes/no
Office Environment	No	No	No	No	No
Mars	No	Semi	No	Semi	No

The environment types largely determine the agent design.

Problem types

- **Single-state problem:** deterministic, accessible
  - Agent knows everything about world (the exact state),
  - Can calculate optimal action sequence to reach goal state.
  - E.g., playing chess. Any action will result in an exact state

Problem types

- **Multiple-state problem:** deterministic, inaccessible
  - Agent does not know the exact state (could be in any of the possible states)
    - May not have sensor at all
  - Assume states while working towards goal state.
  - E.g., walking in a dark room
    - If you are at the door, going straight will lead you to the kitchen
    - If you are at the kitchen, turning left leads you to the bedroom
    - …

Problem types

- **Contingency problem:** nondeterministic, inaccessible
  - Must use sensors during execution
  - Solution is a tree or policy
  - Often interleave search and execution

  - E.g., a new skater in an arena
    - Sliding problem.
    - Many skaters around

Problem types

- **Exploration problem:** unknown state space

  Discover and learn about environment while taking actions.

  - *E.g., Maze*

Problem types

Deterministic, fully observable, known, discrete $\Longrightarrow$ state space problem

Agent knows exactly which state it will be in; solution is a sequence

Non-observable $\Longrightarrow$ conformant problem

Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable $\Longrightarrow$ contingency problem

percepts provide **new** information about current state

solution is a contingent plan or a policy

often **interleave** search, execution

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

Components of well-defined problems

- **Initial state**
- Available **actions** given by **successor function**
  - Initial state + successor function define **state space**
    - **Path**: sequence of states connected by actions
- **Goal test** (i.e. is current state a goal state?)
- **Path costs** (here, sum of **step costs**)

Components of well-defined problems

Solutions

- A **solution** to a problem is a path from the initial state to a goal state
- An **optimal solution** has the lowest path cost among all solutions
  - I.e. solution quality is measured by the path cost
- If path cost sum of step costs, and step costs equal, then optimum solution is shortest path

Example: Romania

On holiday in Romania; currently in Arad.

Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

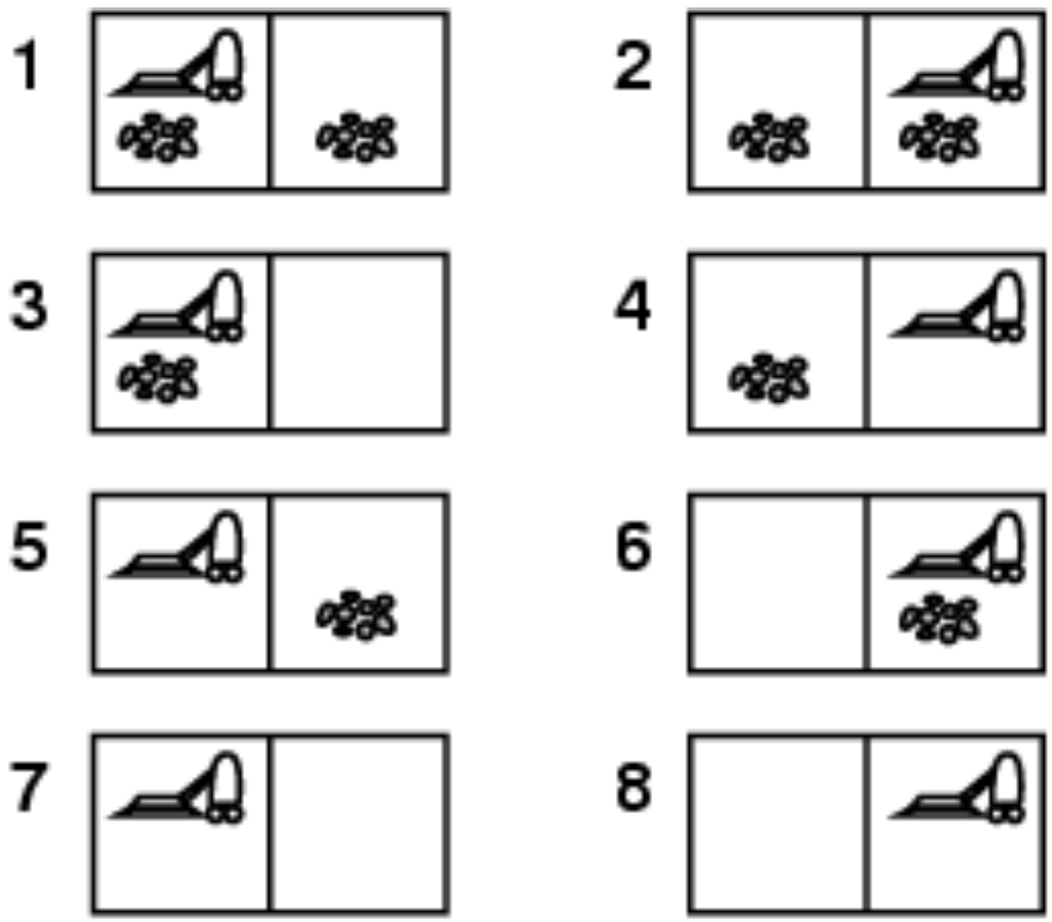
Example: vacuum world

Simplified world: 2 locations, each may or not contain dirt, each may or not contain vacuuming agent.
Goal of agent: clean up the dirt.

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

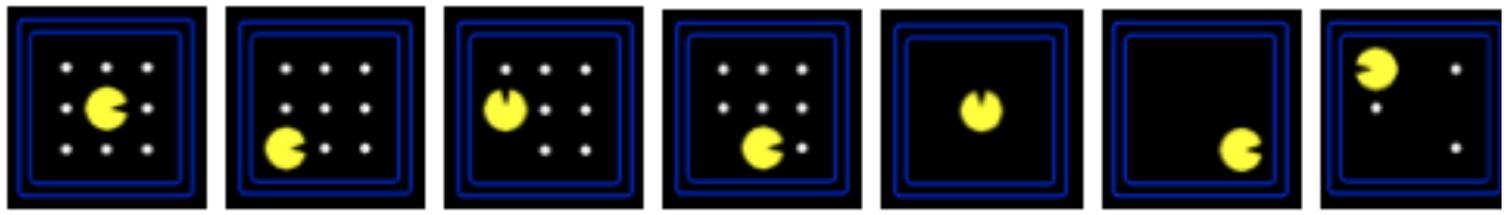
Conformant, start in {1; 2; 3; 4; 5; 6; 7; 8}
e.g., *Right* goes to {2; 4; 6; 8}. Solution??
[*Right; Suck; Left; Suck*]

Contingency, start in #5
Murphy's Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
[*Right*; if *dirt* then *Suck*]

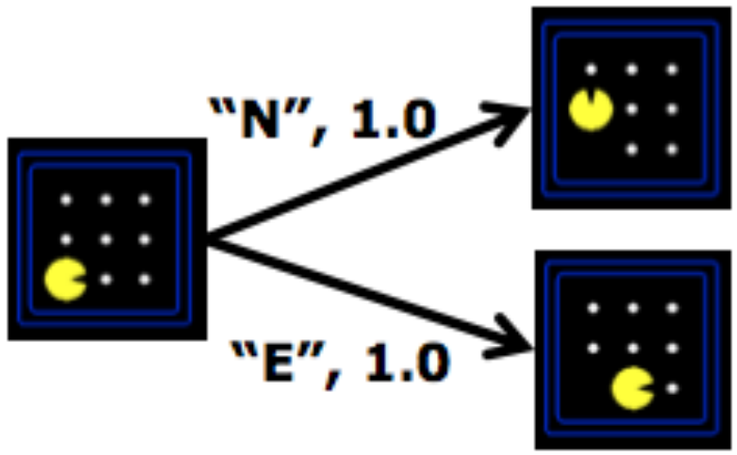


Problem Formulation

- A search problem consists of:
  - A state space



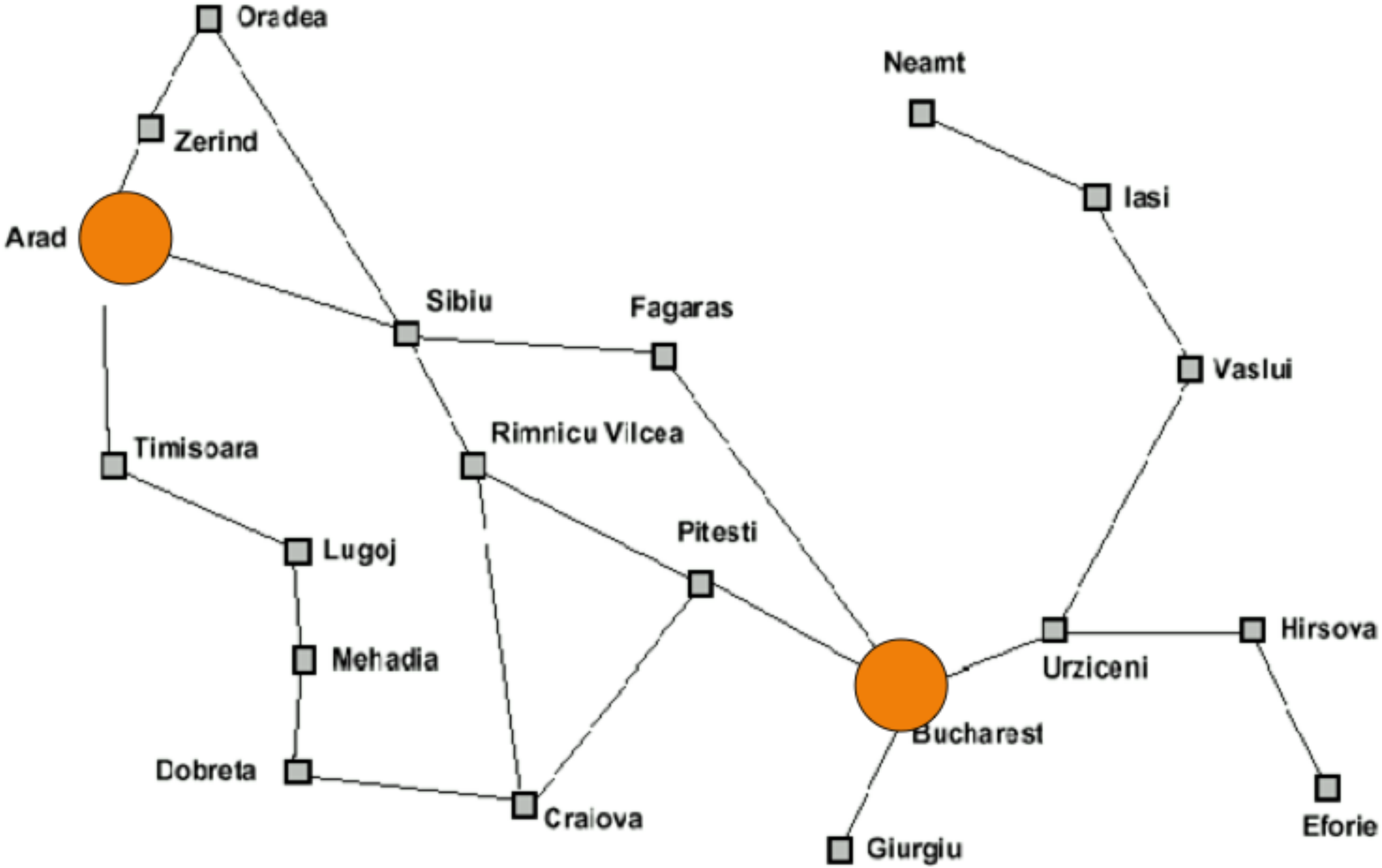
  - A successor function



  - A start state and a goal test

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

Example: Travelling from Arad to Bucharest



Single-state problem formulation

- A problem is defined by four items:

- initial state e.g., "at Arad”

- actions or successor function *S(x)* = set of action–state pairs
  e.g., *S(Arad)* = {*<Arad -->Zerind, Zerind>*, *<Arad -->Timisoara, Timisoara>*, *…* }

- goal test, can be
  explicit, e.g., *x* = "at Bucharest"
  implicit, e.g., *Checkmate(x)*

- path cost (additive)
  e.g., sum of distances, number of actions executed, etc.
  c(x,a,y) is the step cost, assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state

Selecting a state space

- Real world is absurdly complex; some abstraction is necessary to allow us to reason on it…
- Selecting the correct abstraction and resulting state space is a difficult problem!
- Abstract states ⇔ real-world states
- Abstract operators ⇔ sequences or real-world actions
  (e.g., going from city i to city j costs Lij ⇔ actually drive from city i to j)
- Abstract solution ⇔ set of real actions to take in the real world such as to solve problem

Example Problems

- Toy problems
  - vacuum cleaner agent
  - 8-puzzle
  - 8-queens
  - cryptarithmetic
  - missionaries and cannibals
- Real-world problems
  - route finding
  - traveling salesperson
  - VLSI layout
  - robot navigation
  - assembly sequencing

Example: vacuum world

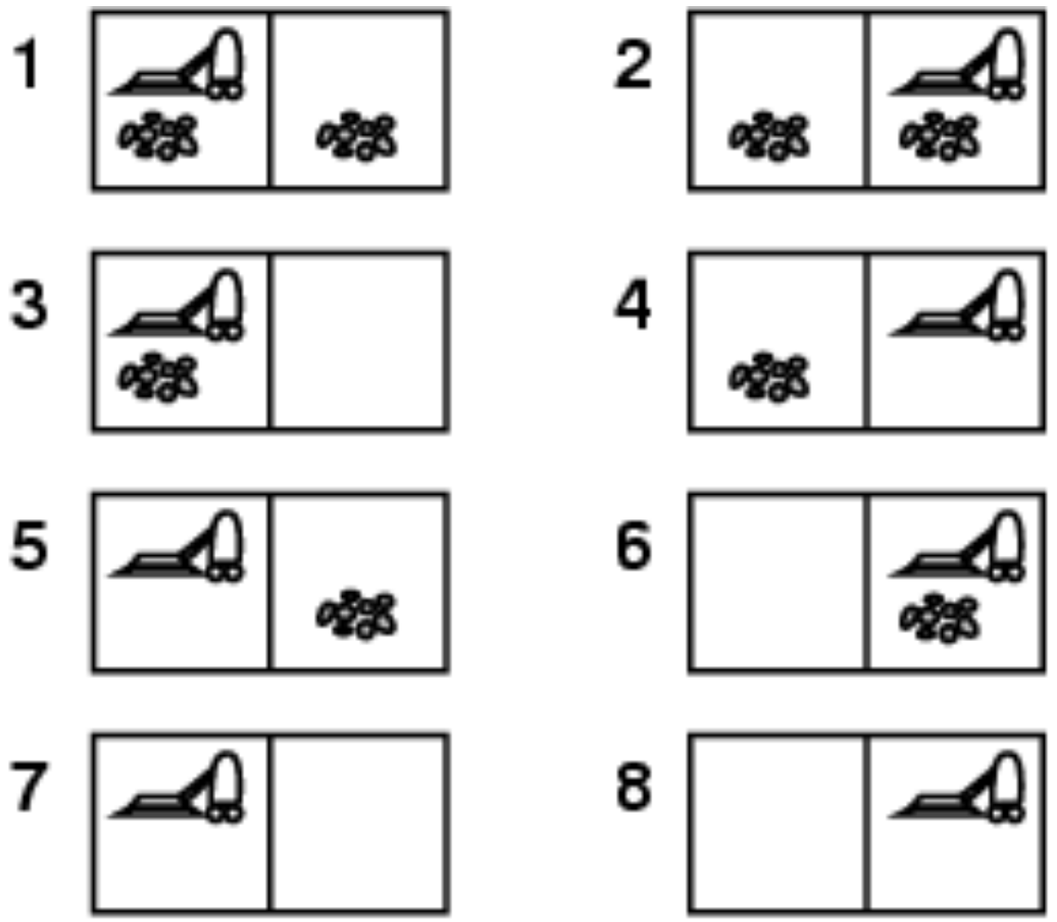
Multiple State Problem

Sensorless

Start in {*1,2,3,4,5,6,7,8*}

Solution?

[Right, Clean, Left, Clean]



Example: vacuum world

Contingency

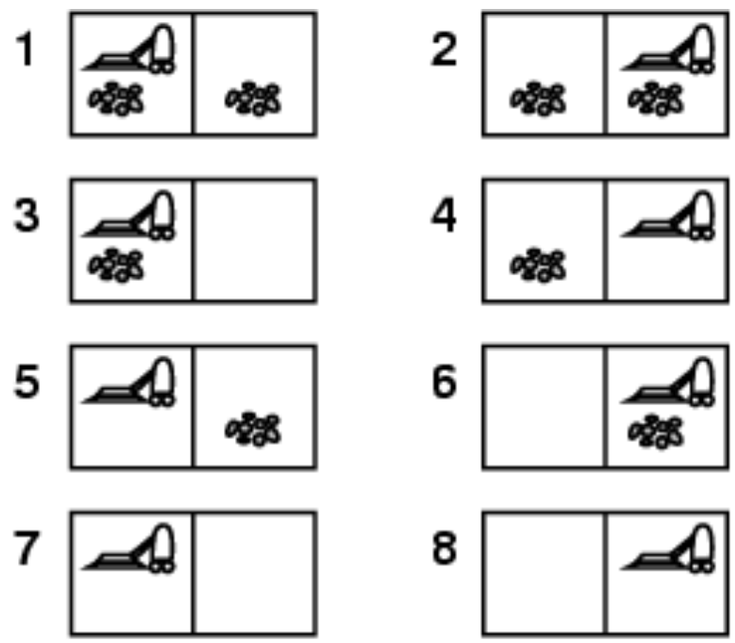
Nondeterminism: *Cleaning* may dirty a clean carpet.

Partially observable: Location, dirt at current location.

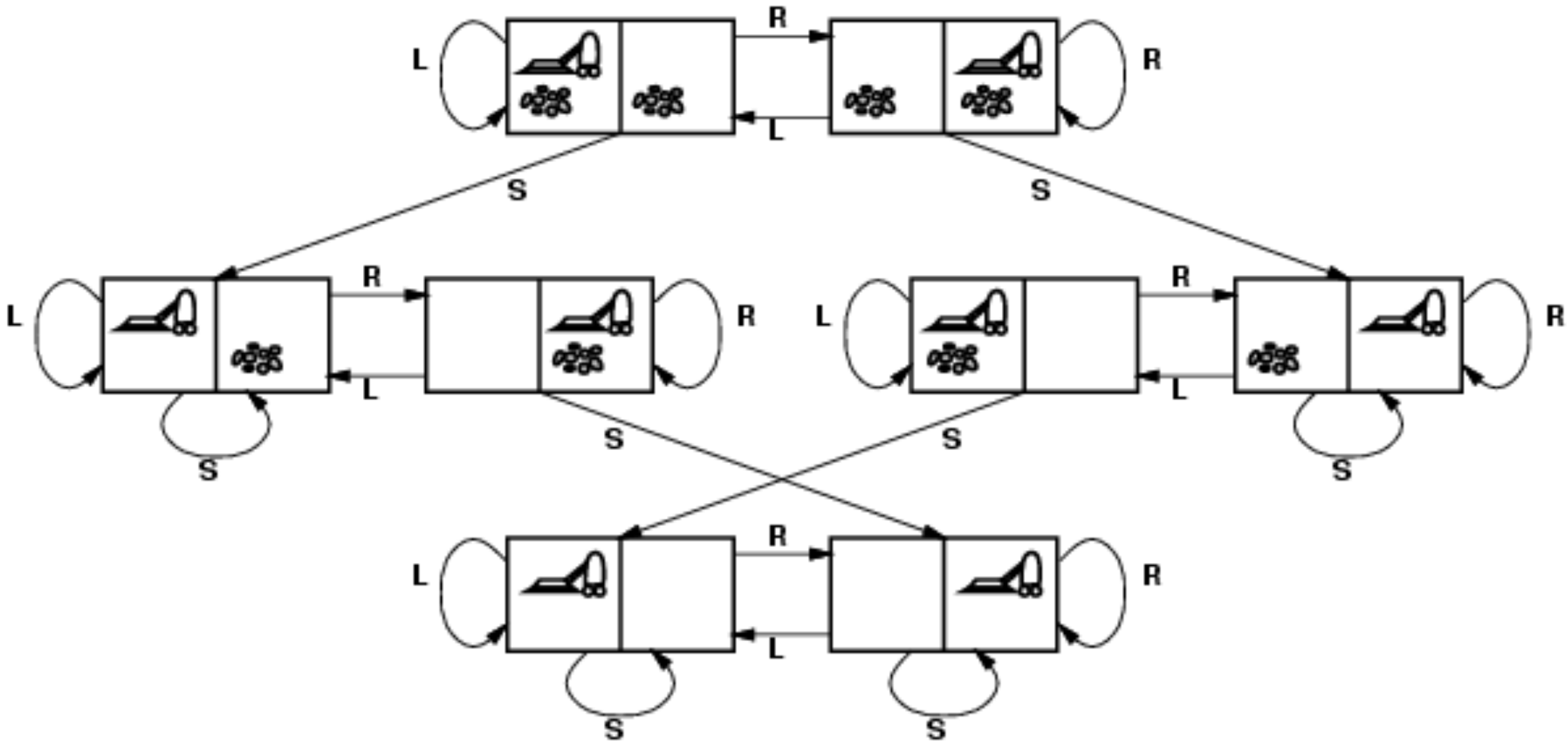
Percept: *[L, Clean],* i.e., start in #5 or #7

Solution?

[Right, if dirt then Clean]



Vacuum world state space graph



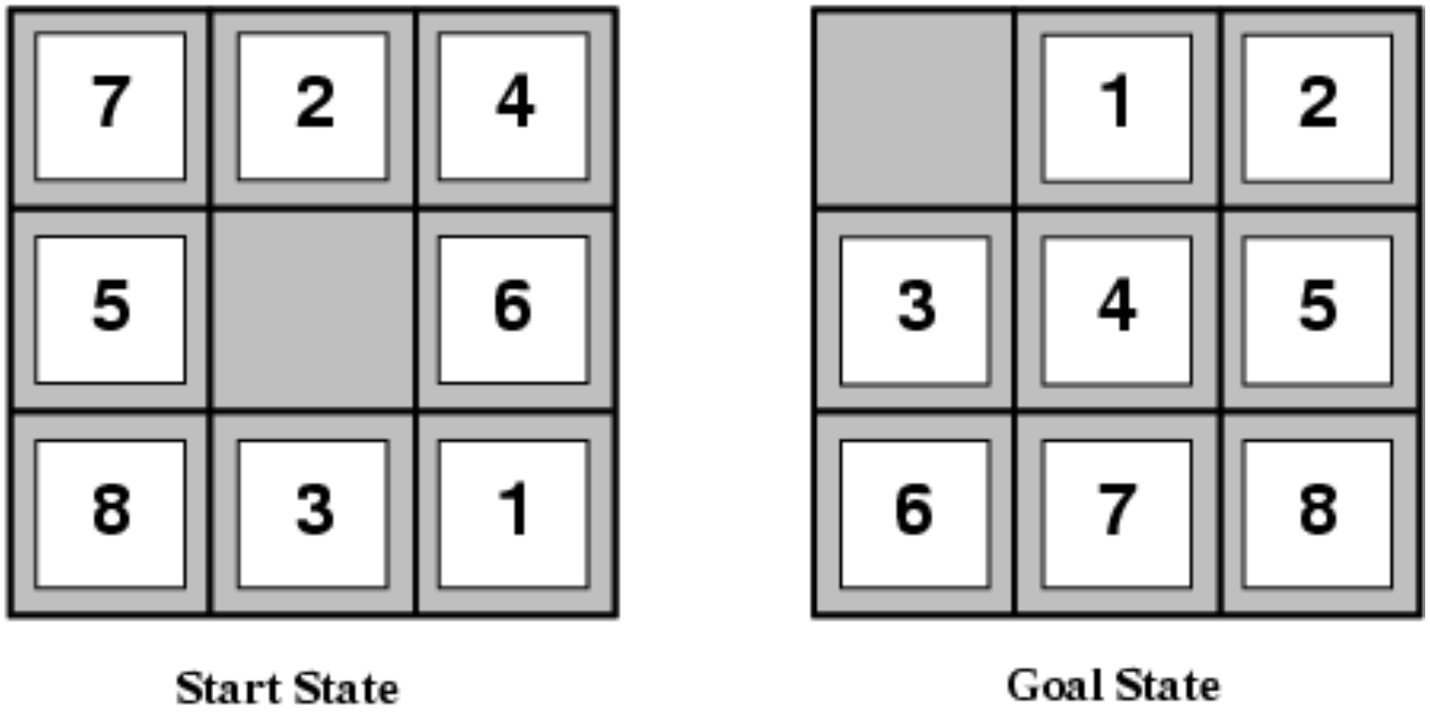
States? Dirt and robot location

Actions? *Left*, *Right*, *Clean*

Goal test? No dirt at all locations

Path cost? 1 per action

Example: The 8-puzzle



Start State Goal State

States? Locations of tiles

Actions? Move blank left, right, up, down

Goal test? Given

Path cost? 1 per move

Example: Eight Puzzle

Eight puzzle is from a family of “sliding –block puzzles”

NP Complete

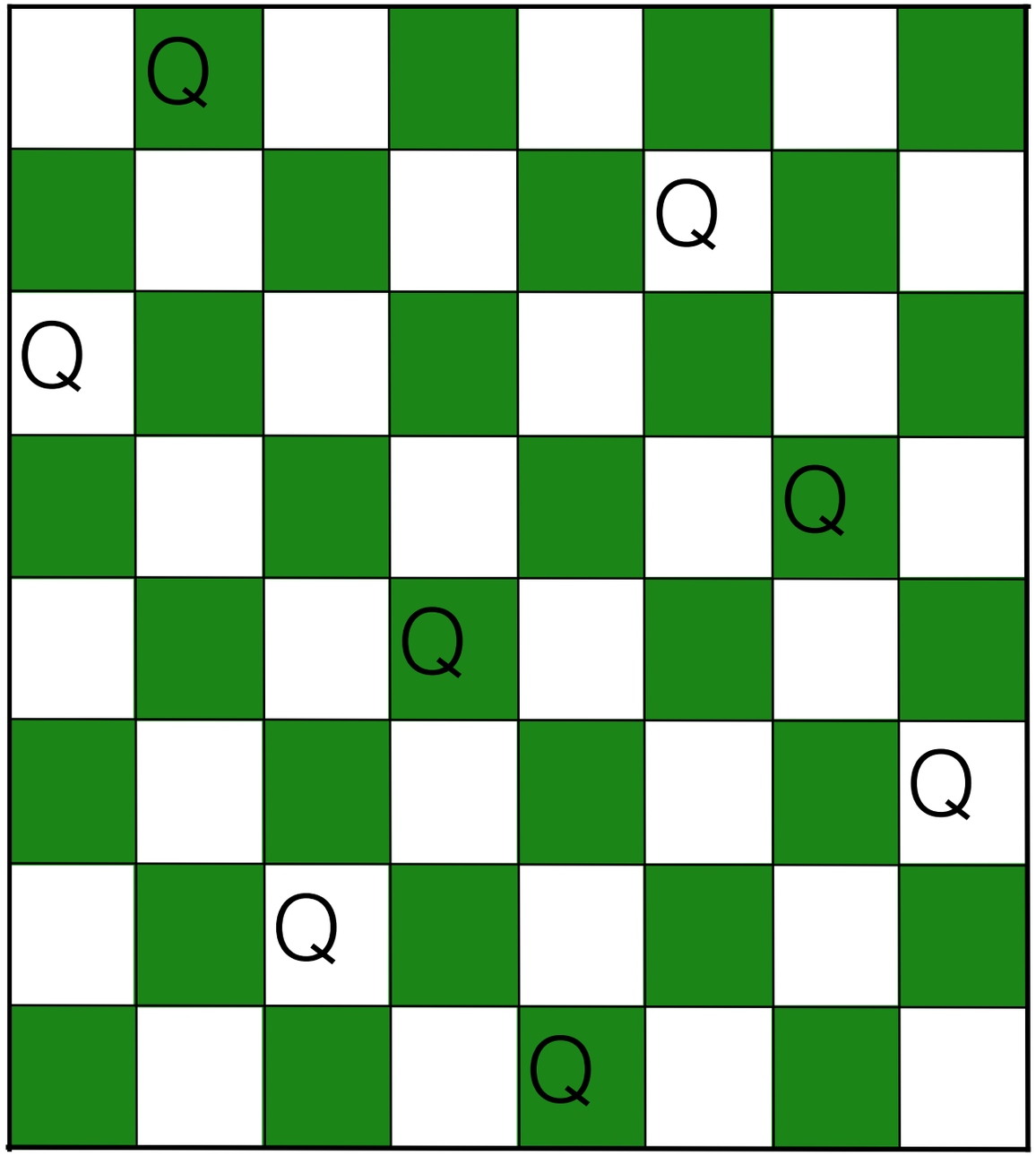
8 puzzle has 9!/2 = 181440 states

15 puzzle has approx. $1.3*10^{12}$ states

24 puzzle has approx. $1*10^{25}$ states

Example: Eight Queens

Place eight queens on a chess board such that no queen can attack another queen



No path cost because only the final state counts!

Incremental formulations

Complete state formulations

Example: Eight Queens

States: Any arrangement of 0 to 8 queens on the board

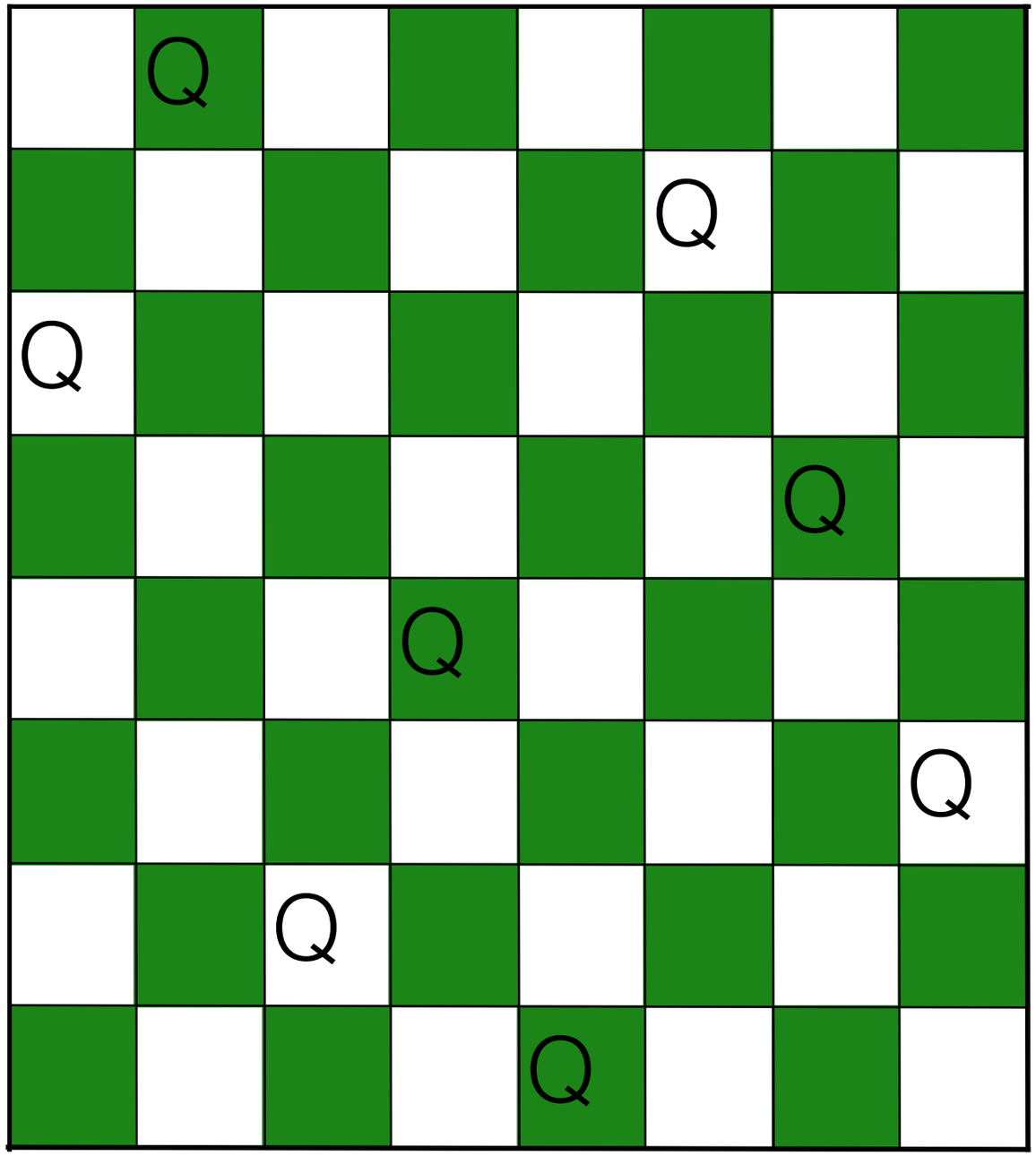
Initial state: No queens on the board

Successor function: Add a queen to an empty square

Goal Test: 8 queens on the board and none are attacked

$64*63*\ldots*57 = 1.8*10^{14}$ possible sequences

Ouch!



Example: Eight Queens

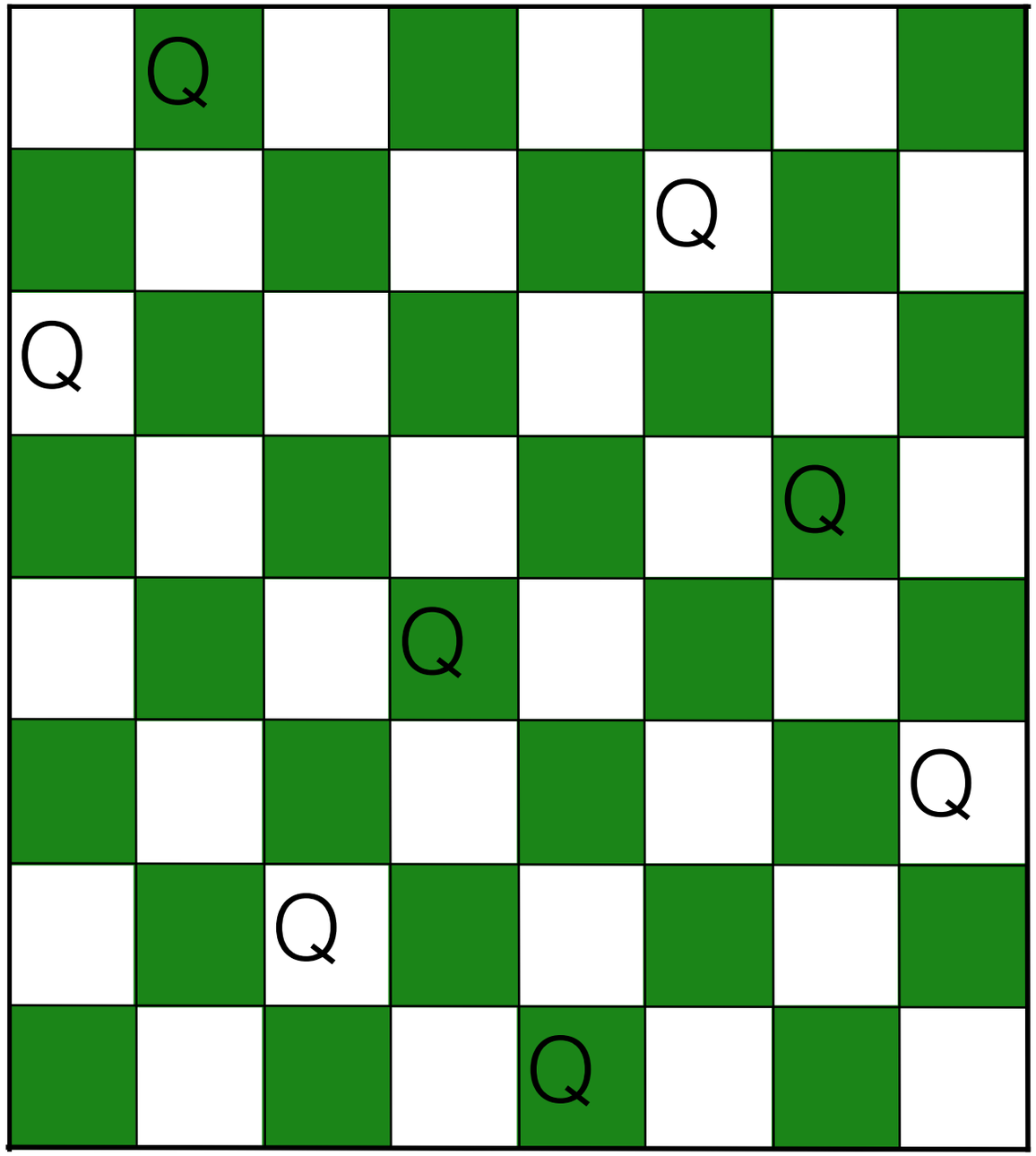
States:

Arrangements of n queens, one per column in the leftmost n columns, with no queen attacking another are states

Successor function:

Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

2057 sequences to investigate



Example: Cryptarithmatic

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

The solution to this puzzle is O = 0, M = 1, Y = 2, E = 5, N = 6, D = 7, R = 8, and S = 9.

States? A cryptharithmetic puzzle w/ some letters replaced with digits.

Actions? Replacing a letter with an unused digit.

Goal test? Puzzle contains only digits.

Path cost? ZERO. All solutions equally valid.

Example: Missionaries and cannibals

problem formulation

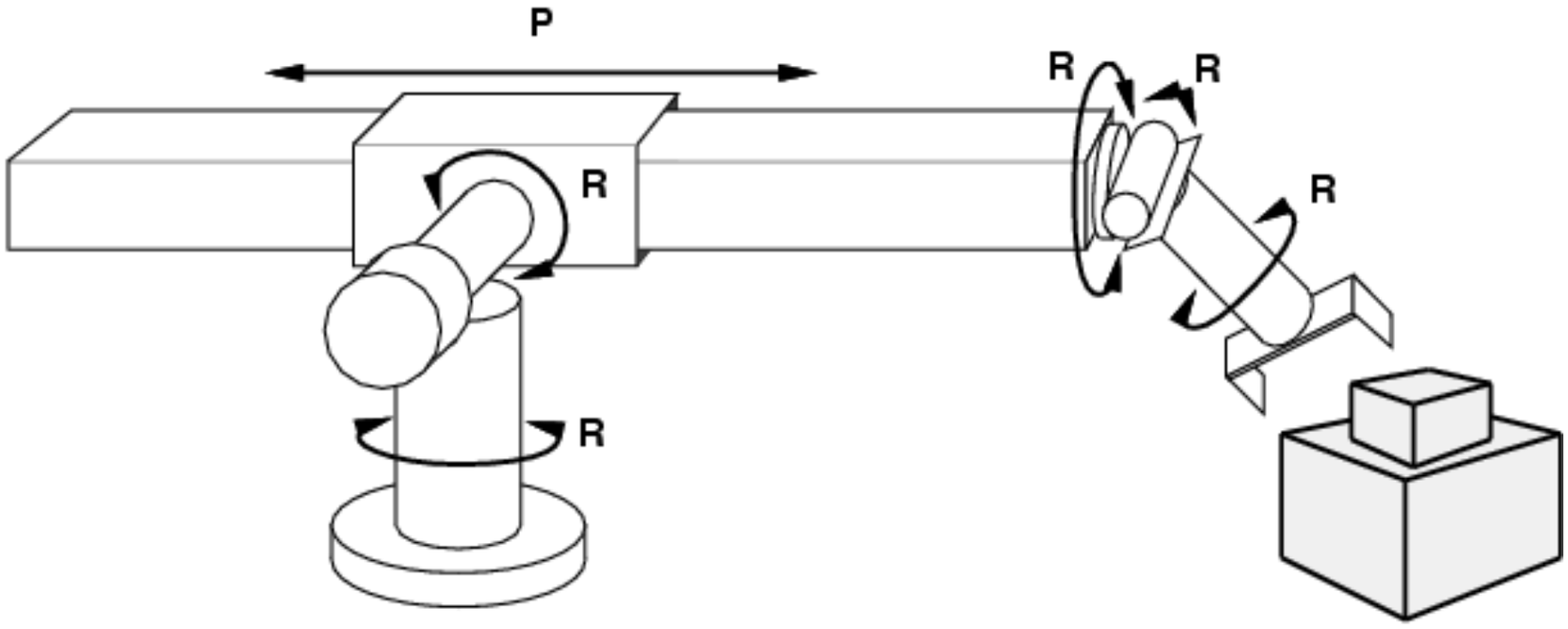
State : 3 missionaries and 3 cannibals in the either side of the river

Operators: either 1 missionary, 1 cannibal, 2 missionaries, 2 cannibals, or one of each across in the boat.

Goal test: 3 missionaries and 3 cannibals in the other side of the river

Path cost: the number of crossing

Example: robotic assembly



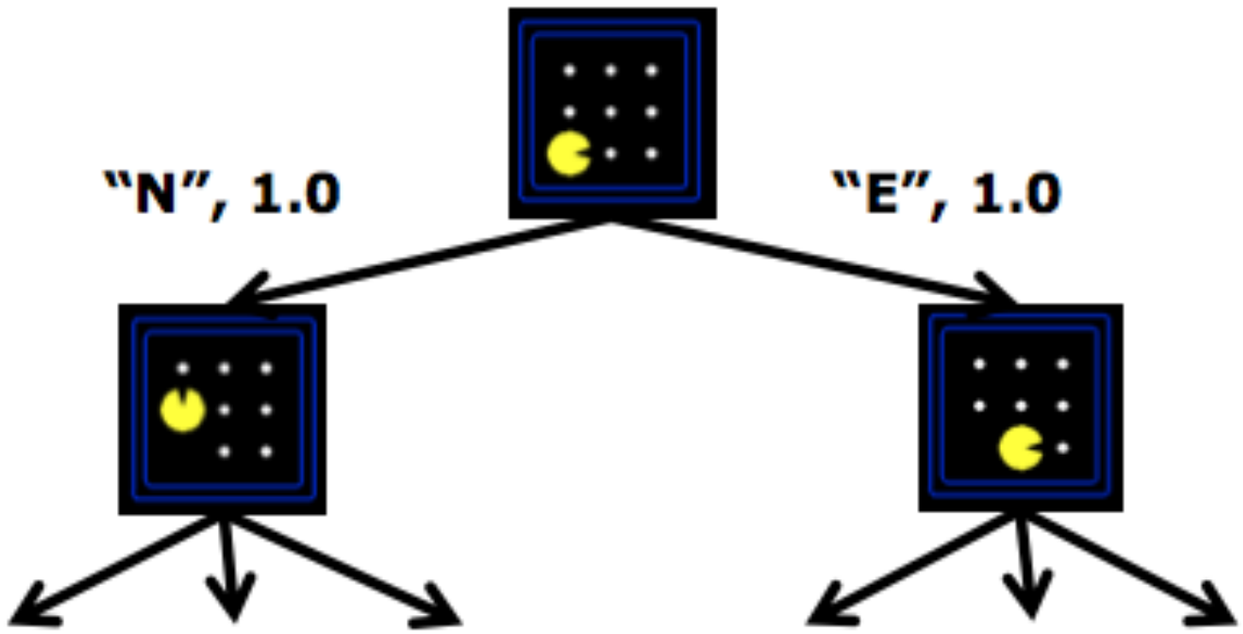
States? real-valued coordinates of robot joint angles parts of the object to be assembled

Actions? continuous motions of robot joints

Goal test? complete assembly

Path cost? time to execute

Search Trees



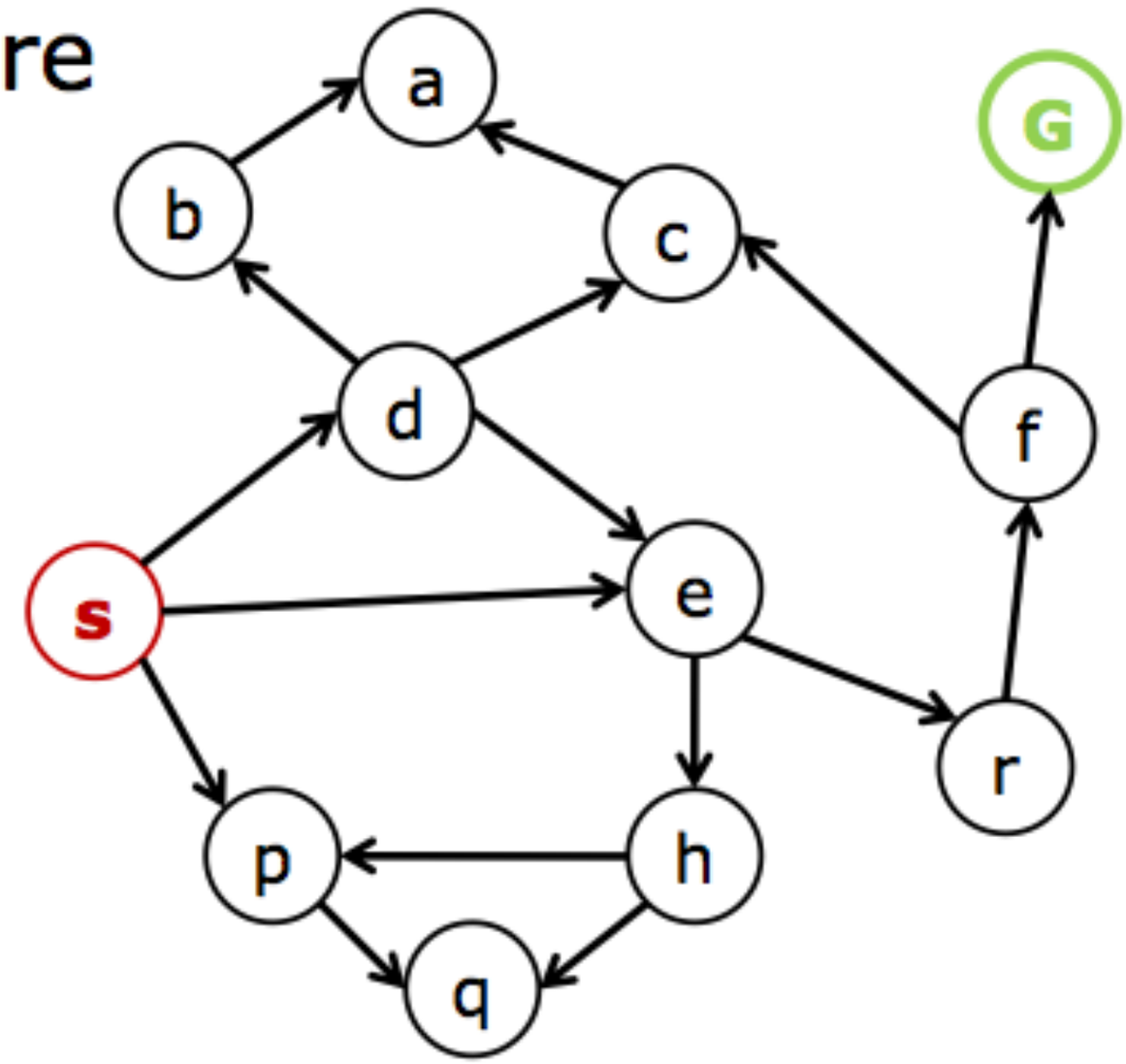
- A search tree:
  - This is a "what if" tree of plans and outcomes
  - Start state at the root node
  - Children correspond to successors
  - Nodes labeled with states, correspond to PLANS to those states
  - For most problems, can never build the whole tree
    - So, have to find ways to use only the important parts!

Tree search algorithms

- There's a big graph where
  - Each state is a node
  - Each successor is an outgoing arc

- Important: For most problems we could never actually build this graph

- How many states in Pacman?

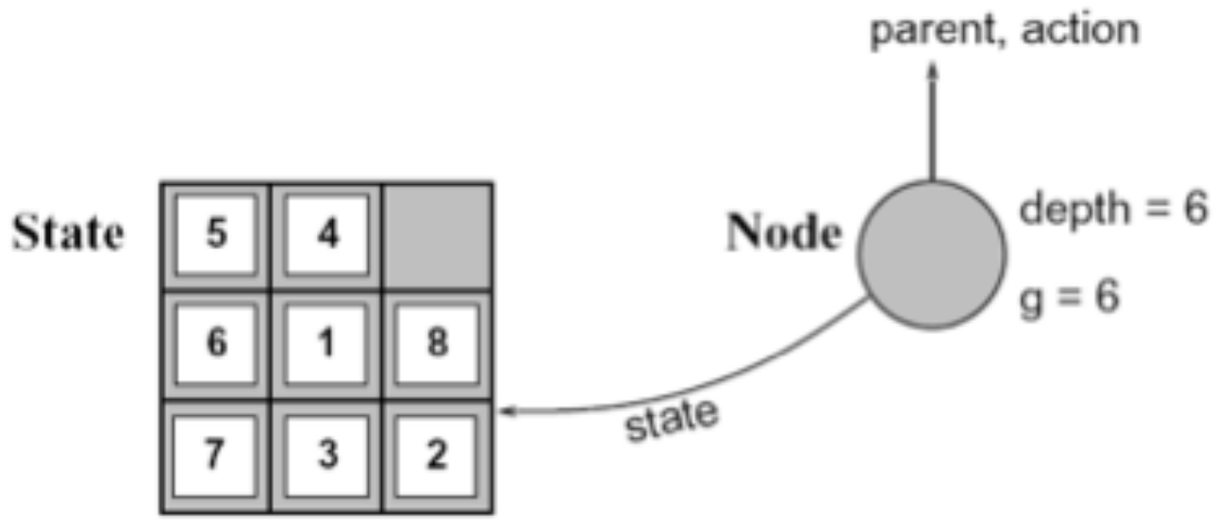


Laughably tiny search graph for a tiny search problem

States Vs. Nodes

- Problem graphs have problem states
  - Represent an abstracted state of the world
  - Have successors, predecessors, can be goal / non-goal

- Search trees have search nodes
  - Represent a plan (path) which results in the node's state
  - Have 1 parent, a length and cost, point to a problem state
  - Expand uses successor function to create new tree nodes
  - The same problem state in multiple search tree nodes



Tree search algorithms

Basic idea:

offline, simulated 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

Finding a solution

Solution: is a sequence of operators that bring you from current state to the goal state.

```
function TREE-SEARCH(problem, strategy)
                                        returns a solution, or failure
  initialize the search tree using the initial state 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 resulting nodes to the search
tree
end
```

Strategy: The search strategy is determined by ???

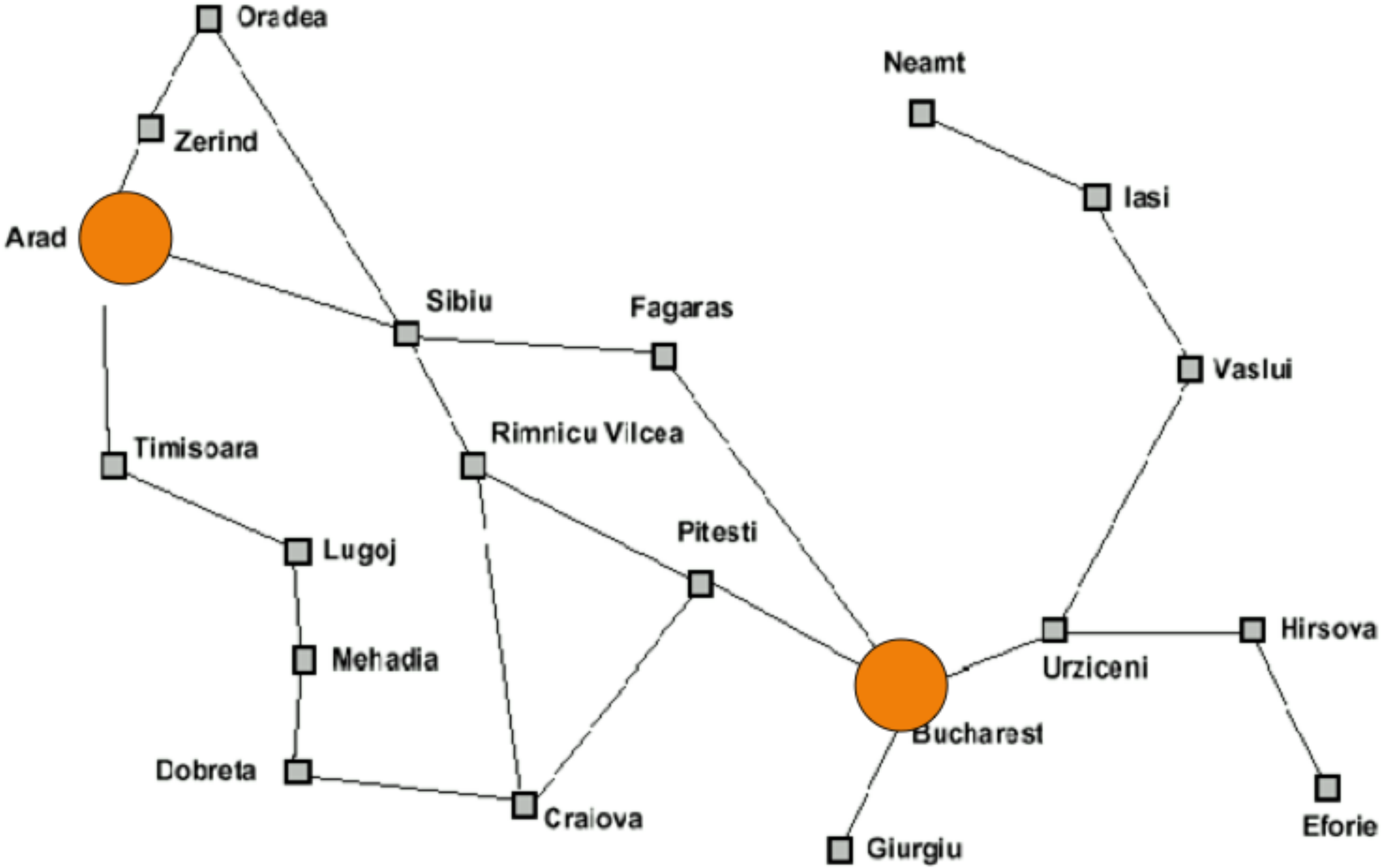
Finding a solution

Solution: is a sequence of operators that bring you from current state to the goal state.

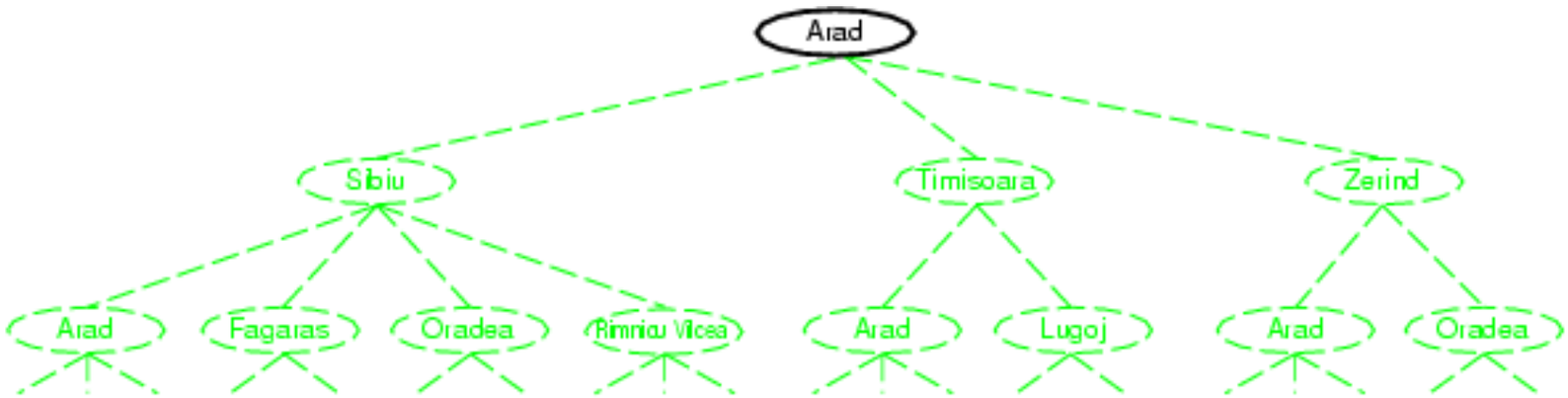
```
function TREE-SEARCH(problem, strategy)
                                    returns a solution, or failure
  initialize the search tree using the initial state problem
  loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according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

Strategy: The search strategy is determined by the **order** in which the nodes are expanded.

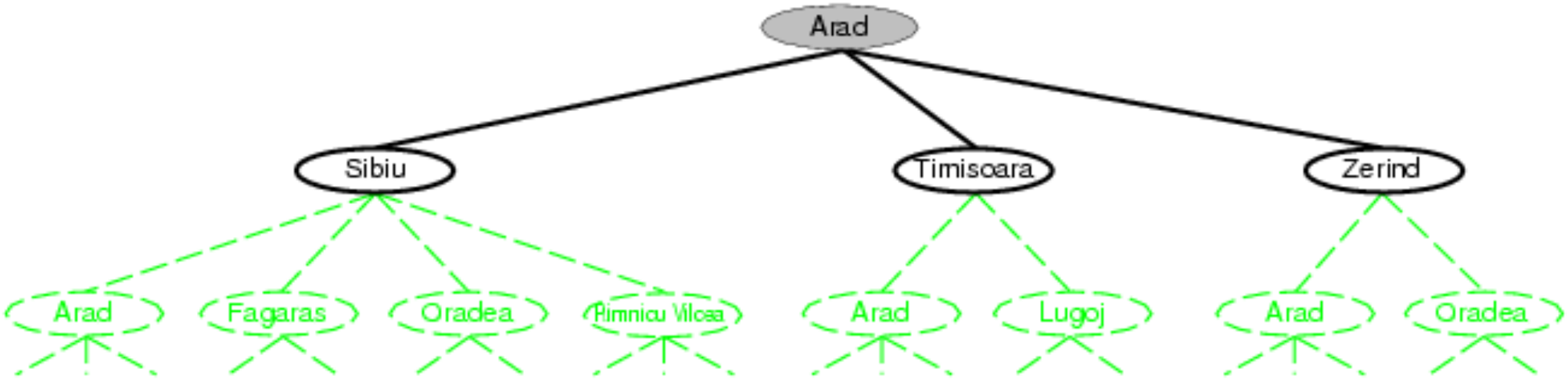
Example: Travelling from Arad to Bucharest



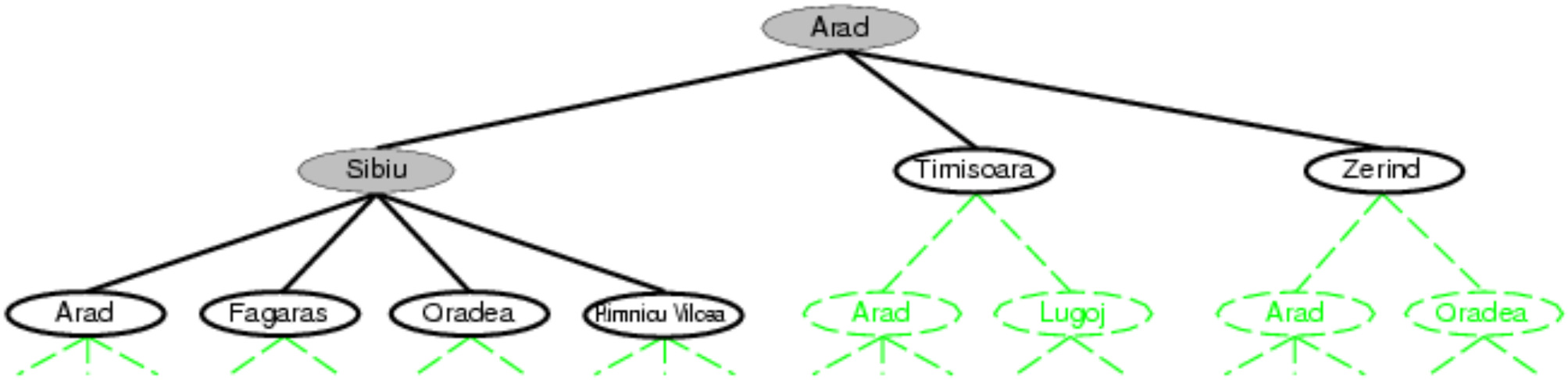
Tree search example



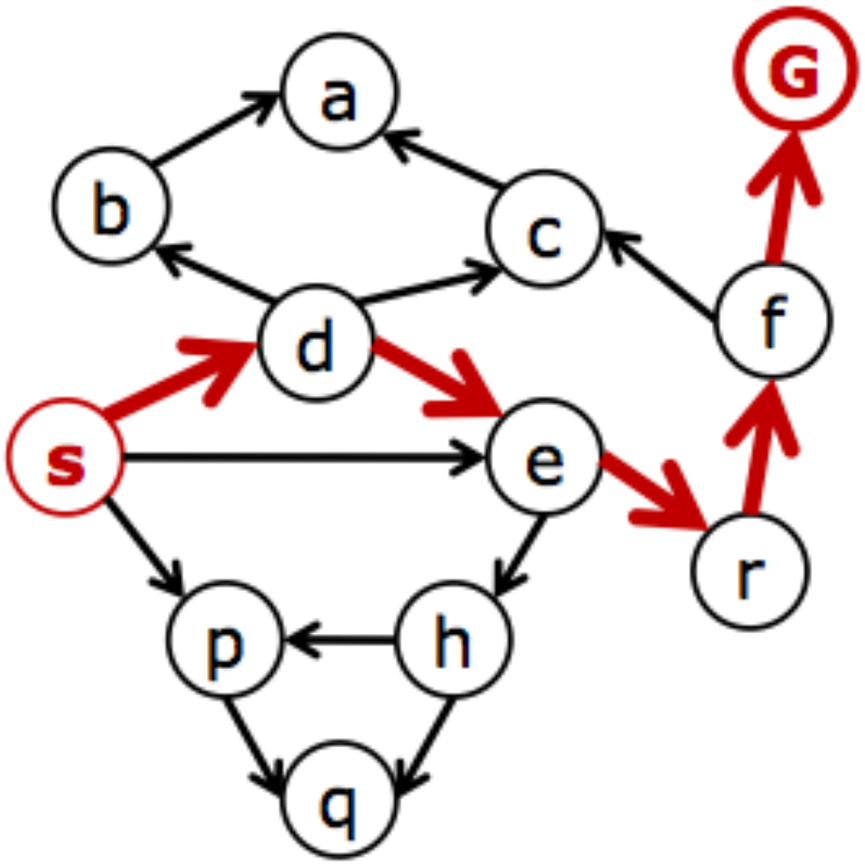
Tree search example



Tree search example

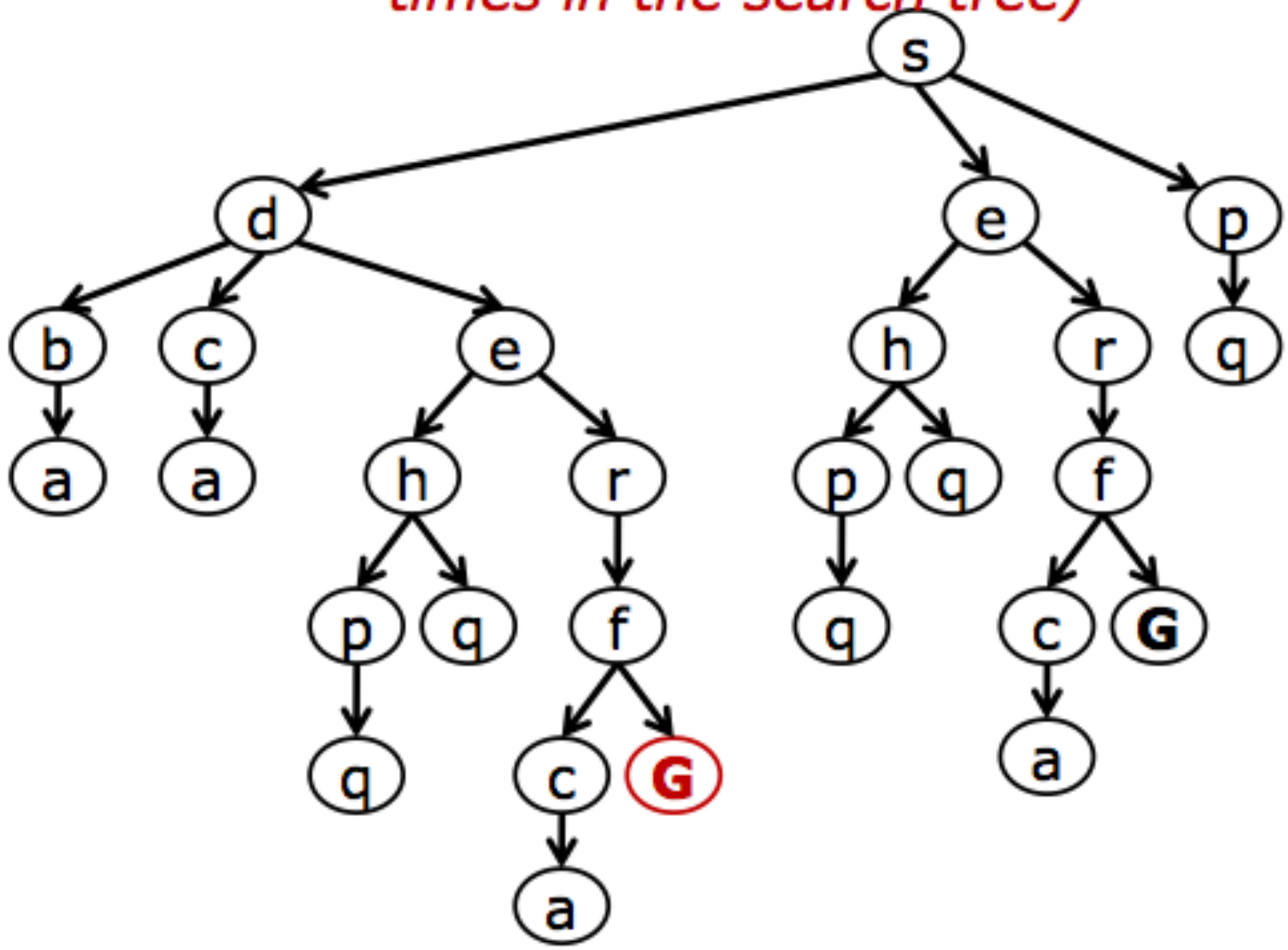


State Graphs vs. search Trees



We almost always construct both on demand – and we construct as little as possible

Each NODE in the search tree is an entire PATH in the state graph (note how many nodes in the graph appear multiple times in the search tree)



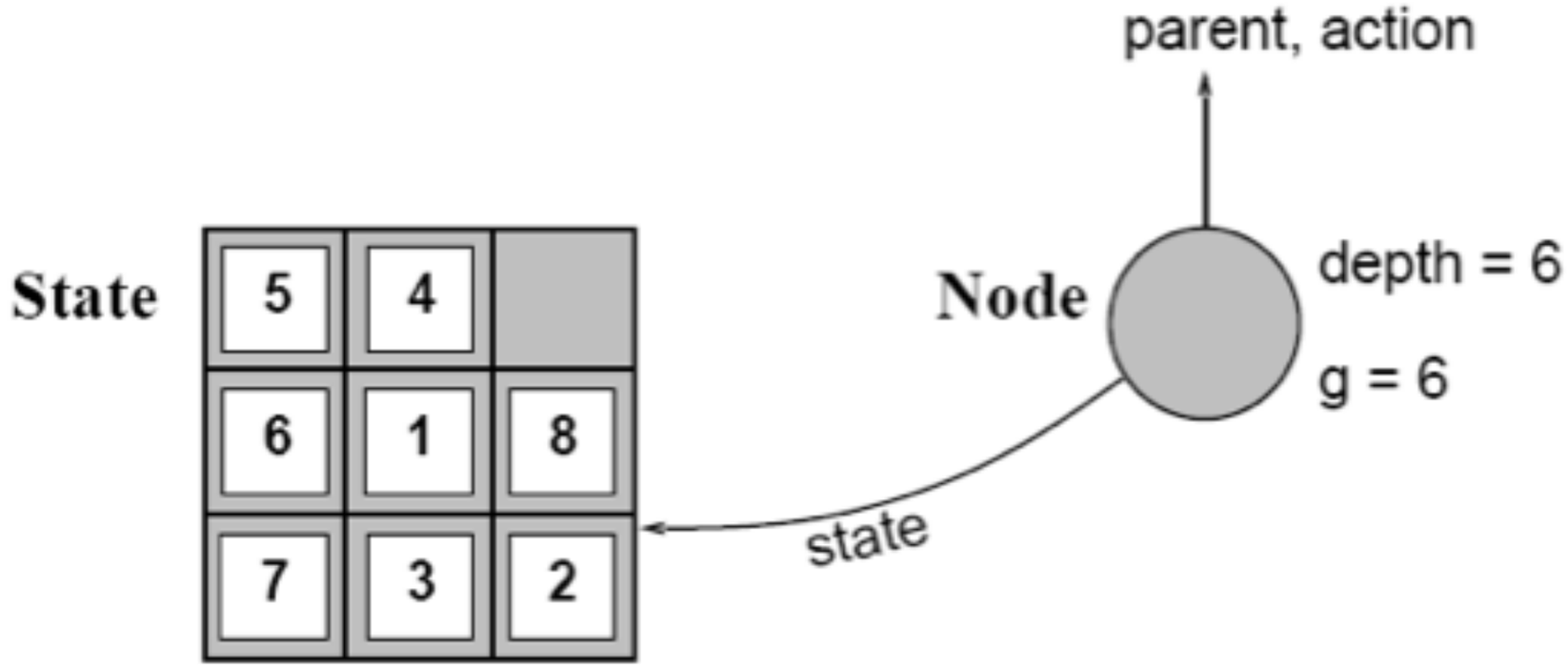
Encapsulating state information in nodes

A state is a (representation of) a physical configuration

A node is a data structure constituting part of a search tree

includes parent, children, depth, path cost $g(x)$

States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFN of the problem to create the corresponding states.

Implementing general 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 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
```

Implementing general 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 node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
                                  blem, STATE[node]) do
                                  ction; STATE[s] ← result
                                  EP-COST(node, action, s)
    return successors
```

The operations on ***fringe*** are queuing functions which inserts and removes elements into the ***fringe*** and determines the order of node expansion. Varieties of the queuing functions produce varieties of the search algorithm.

Evaluation of search strategies

- A search strategy is defined by picking the order of node expansion.
- Search algorithms are commonly evaluated according to the following four criteria:
  - **Completeness:** does it always find a solution if one exists?
  - **Time complexity:** how long does it take as function of num. of nodes?
  - **Space complexity:** how much memory does it require?
  - **Optimality:** does it guarantee the least-cost solution?
- Time and space complexity are measured in terms of:
  - *b* – max branching factor of the search tree
  - *d* – depth of the least-cost solution
  - *m* – max depth of the search tree (may be infinity)