Last time: Summary

- **Definition of AI?**
- **Turing Test?**
- **Intelligent Agents:**
  - Anything that can be *viewed as* **perceiving** its **environment** through **sensors** and **acting** upon that environment through its **effectors** to maximize progress towards its **goals**.
  - PAGE (Percepts, Actions, Goals, Environment)
  - Described as a Perception (sequence) to Action Mapping: $f: \mathcal{P}^* \rightarrow \mathcal{A}$
  - Using look-up-table, closed form, etc.

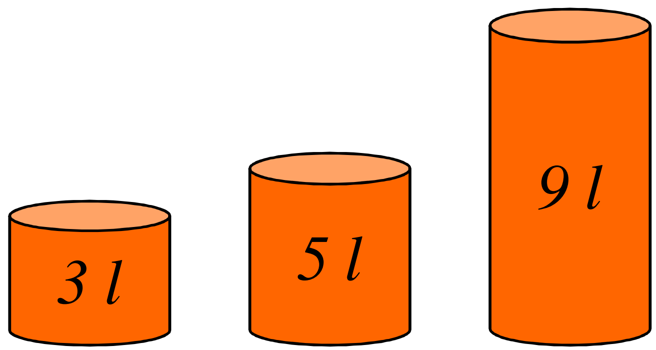
- **Agent Types:** Reflex, state-based, goal-based, utility-based

- **Rational Action:** The action that maximizes the expected value of the performance measure <u>given the percept sequence to date</u>

Outline: Problem solving and search

- **Introduction to Problem Solving**
- **Complexity**
- **Uninformed search**
  - Problem formulation
  - Search strategies: depth-first, breadth-first, uniform search
- **Informed search**
  - Search strategies: best-first, A*
  - Heuristic functions

Example: Measuring problem!

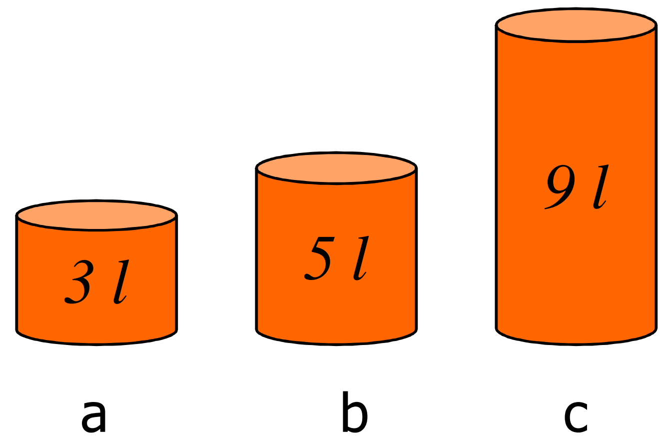


Problem: Using these three buckets,
measure 7 liters of water.

Example: Measuring problem!

- **(one possible) Solution:**

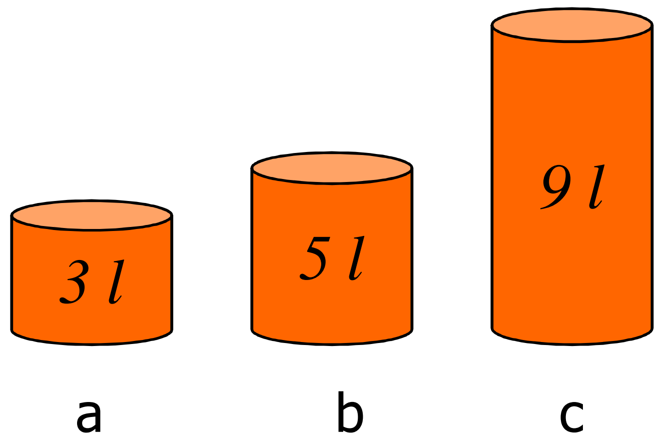
a	b	c	
0	0	0	start



Example: Measuring problem!

- **(one possible) Solution:**

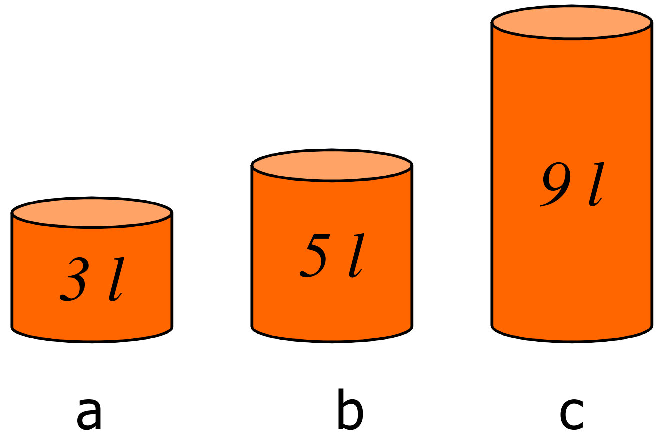
a	b	c	
0	0	0	start
3	0	0	



Example: Measuring problem!

- **(one possible) Solution:**

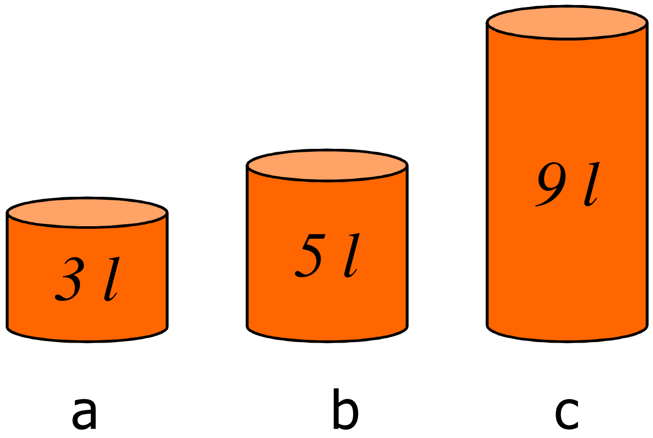
a	b	c	
0	0	0	start
3	0	0	
0	0	3	



Example: Measuring problem!

- **(one possible) Solution:**

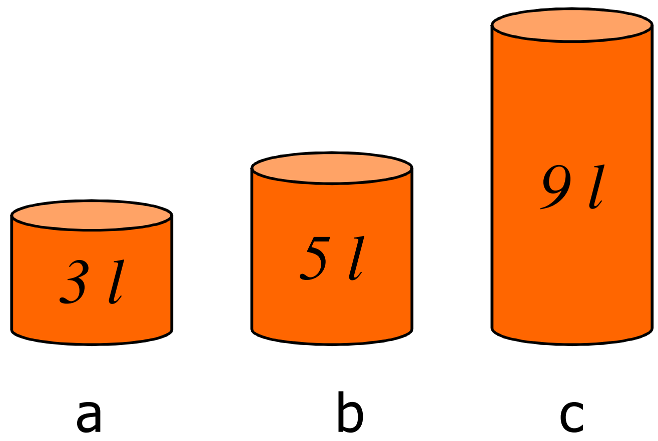
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	



Example: Measuring problem!

- **(one possible) Solution:**

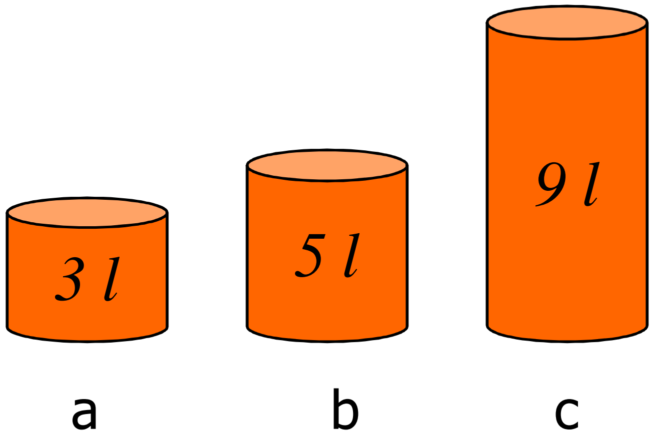
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	



Example: Measuring problem!

- **(one possible) Solution:**

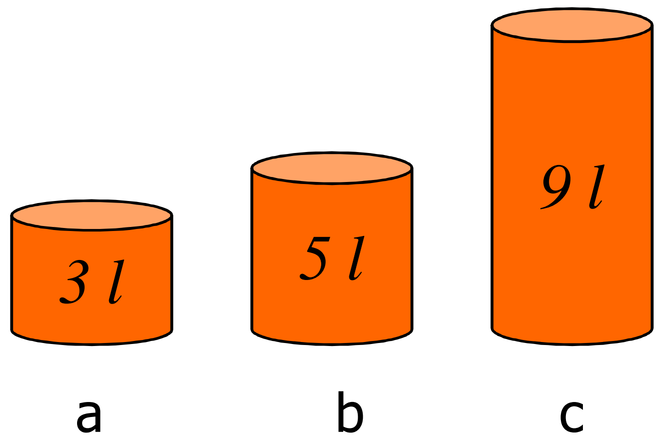
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	



Example: Measuring problem!

- **(one possible) Solution:**

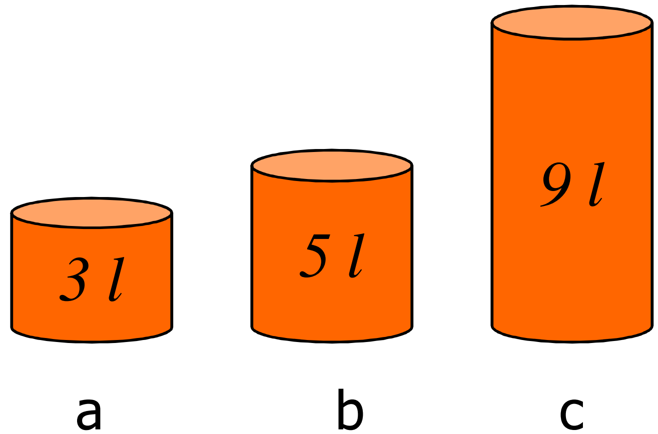
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	



Example: Measuring problem!

- **(one possible) Solution:**

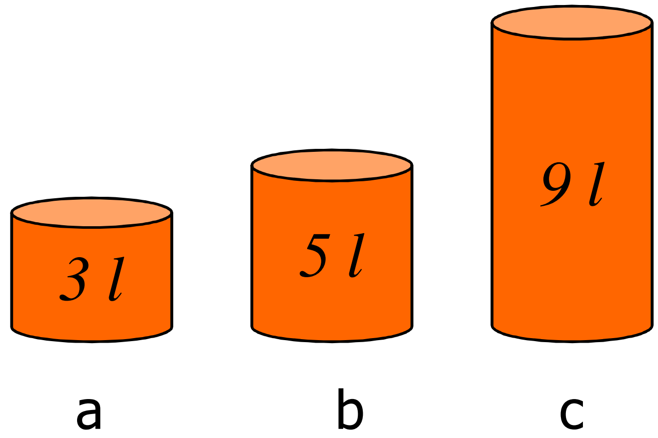
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	



Example: Measuring problem!

- **(one possible) Solution:**

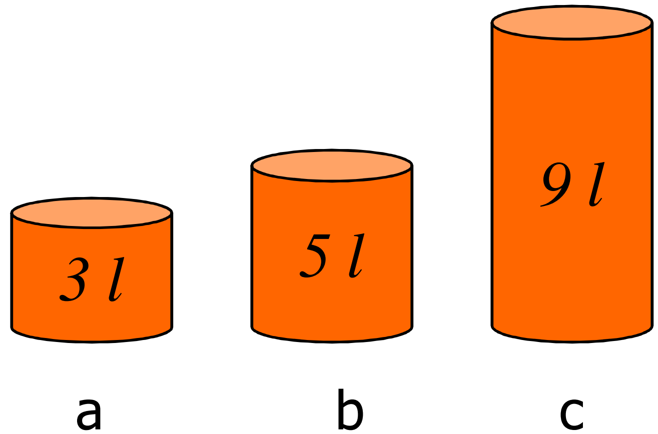
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	



Example: Measuring problem!

- **(one possible) Solution:**

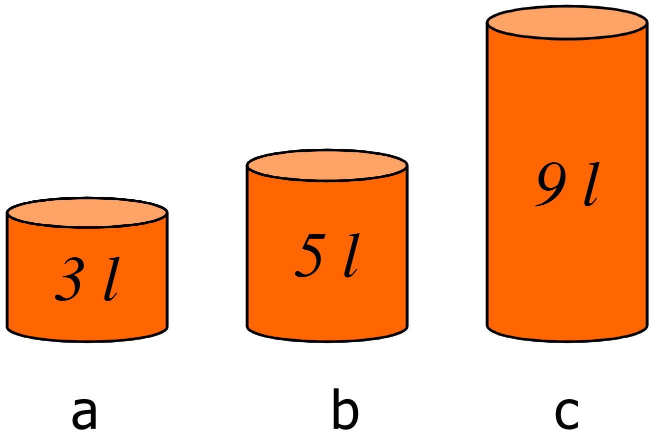
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	**7**	**goal**



Example: Measuring problem!

- **Another Solution:**

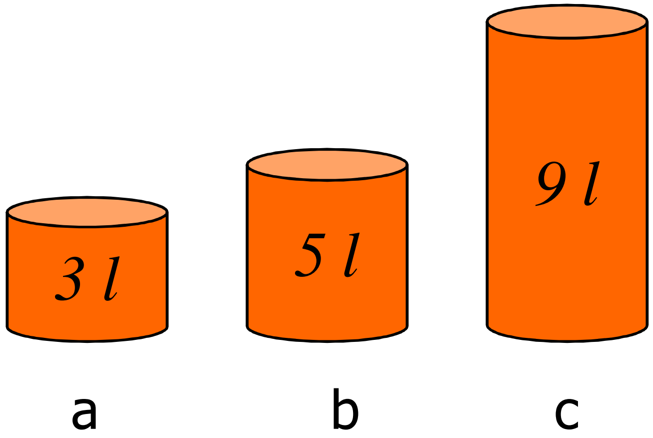
a	b	c	
0	0	0	start
0	5	0	



Example: Measuring problem!

- **Another Solution:**

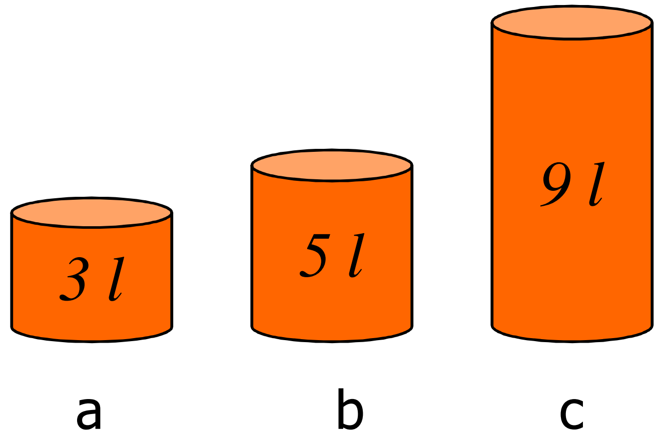
a	b	c	
0	0	0	start
0	5	0	
3	2	0	



Example: Measuring problem!

- **Another Solution:**

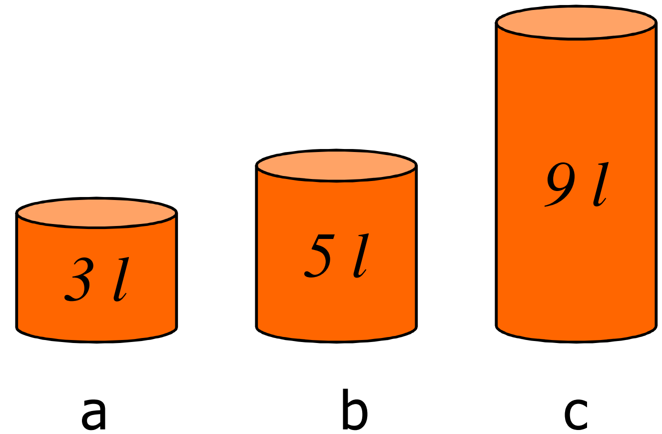
a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	



Example: Measuring problem!

- **Another Solution:**

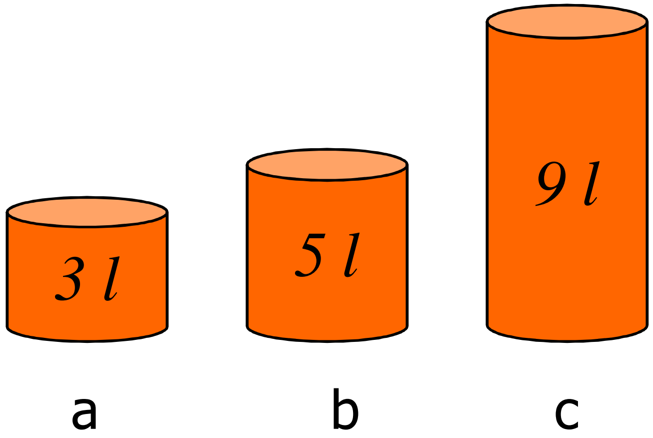
a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	
3	5	2	



Example: Measuring problem!

- **Another Solution:**

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



Which solution do we prefer?

- **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

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

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

Assumes environment is Static, Observable, Discrete, and Deterministic – in other words, this kind of agent can handle the simplest kind of problems…

Example: Buckets

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

Remember (lecture 2): Environment types

Environment	Observable	Deterministic	Episodic	Static	Discrete
Crossword Puzzle **[S]**	Fully	Deterministic	Non-Episodic	Yes	Yes
Chess with a Clock **[M]**	Fully	Deterministic	Non-Episodic	Semi-Static	Yes
Driving a Taxi **[S]**	Partial	Non-Deterministic	Non-Episodic	Dynamic	Continuous
Interactive English Tutor **[M]**	Partial	Non-Deterministic	Non-Episodic	Dynamic	Yes
Part Sorting Robot **[S]**	Partial	Non-Deterministic	Yes	Dynamic	Continuous

The environment types largely determine the agent design.

Problem types

- **Single-state problem:** deterministic, accessible
  Agent knows everything about world, thus can calculate optimal action sequence to reach goal state.

- **Multiple-state problem:** deterministic, inaccessible
  Agent must reason about sequences of actions and states assumed while working towards goal state.

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

- **Exploration problem:** unknown state space
  Discover and learn about environment while taking actions.

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*

Example: Vacuum world

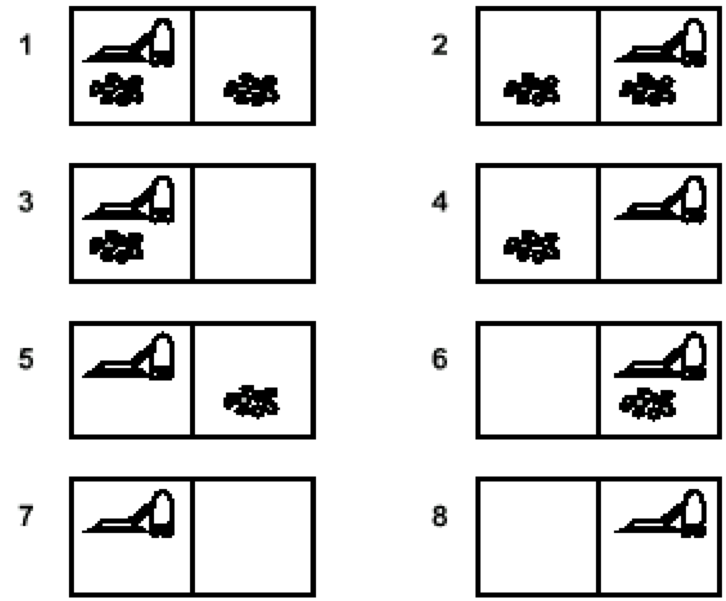
Simplified world: 2 locations, each may or not contain dirt, each may or not contain vacuuming agent.

Goal of agent: clean up the dirt.

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

Multiple-state, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??

Contingency, start in #5
Murphy's Law: $Suck$ can dirty a clean carpet
Local sensing: dirt, location only.
Solution??



Example: Romania

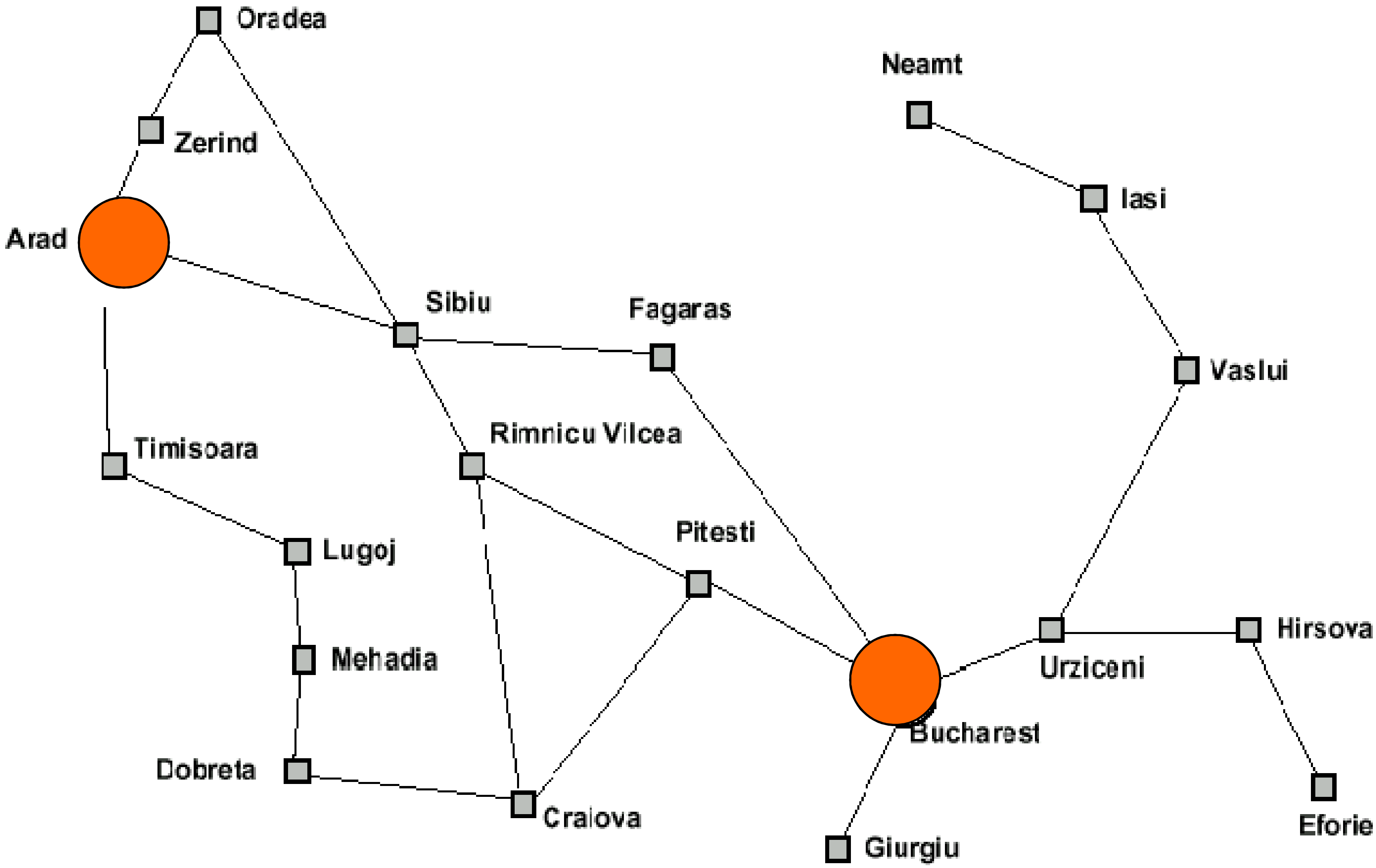
- In Romania, on vacation. Currently in Arad.
- Flight leaves tomorrow from Bucharest.

- **Formulate goal:**
  - be in Bucharest

- **Formulate problem:**
  - states: various cities
  - operators: drive between cities

- **Find solution:**
  - sequence of cities, such that total driving distance is minimized.

Example: Traveling from Arad To Bucharest



Problem formulation

A *problem* is defined by four items:

initial state e.g., "at Arad"

operators (or *successor function* $S(x)$)
e.g., Arad $\rightarrow$ Zerind Arad $\rightarrow$ Sibiu etc.

goal test, can be
explicit, e.g., $x =$ "at Bucharest"
implicit, e.g., $NoDirt(x)$

path cost (additive)
e.g., sum of distances, number of operators executed, etc.

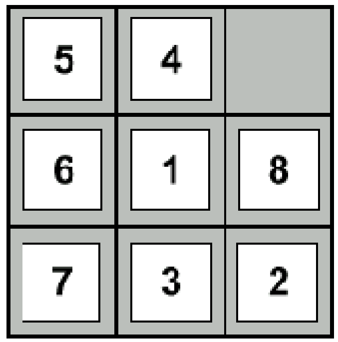
A *solution* is a sequence of operators
leading from the initial state to a goal state

Selecting a state space

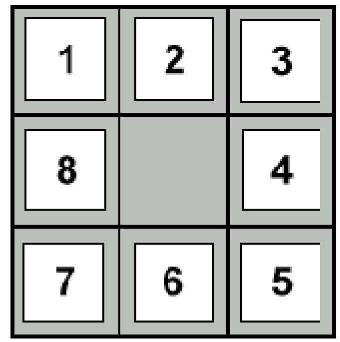
- Real world is absurdly complex; some abstraction is necessary to allow us to reason on it…
- Selecting the correct abstraction and resulting state space is a difficult problem!
- Abstract states ⇔ real-world states
- Abstract operators ⇔ sequences or real-world actions

  (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: 8-puzzle



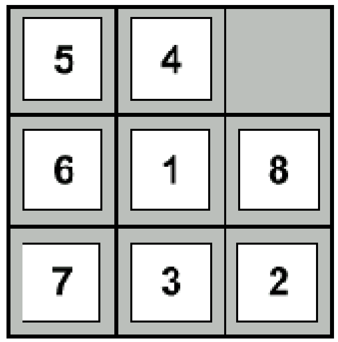
start state



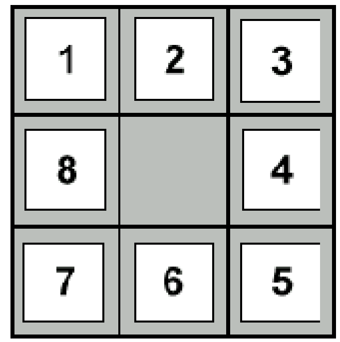
goal state

- State:
- Operators:
- Goal test:
- Path cost:

Example: 8-puzzle



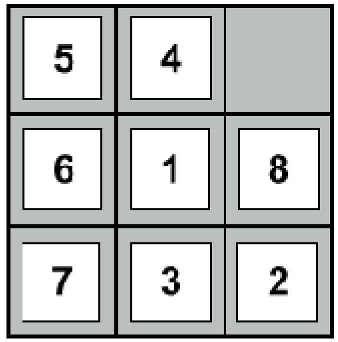
start state



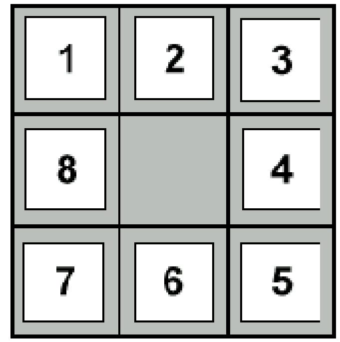
goal state

- State: integer location of tiles (ignore intermediate locations)
  - Size of the State Space: Number of unique states in problem
  - What is the State Space size for the problem above?
  - Think of it as a “counting” problem: 9! Distinct states
- Operators: moving blank left, right, up, down (ignore jamming)
- Goal test: does state match goal state?
- Path cost: 1 per move

Example: 8-puzzle



start state



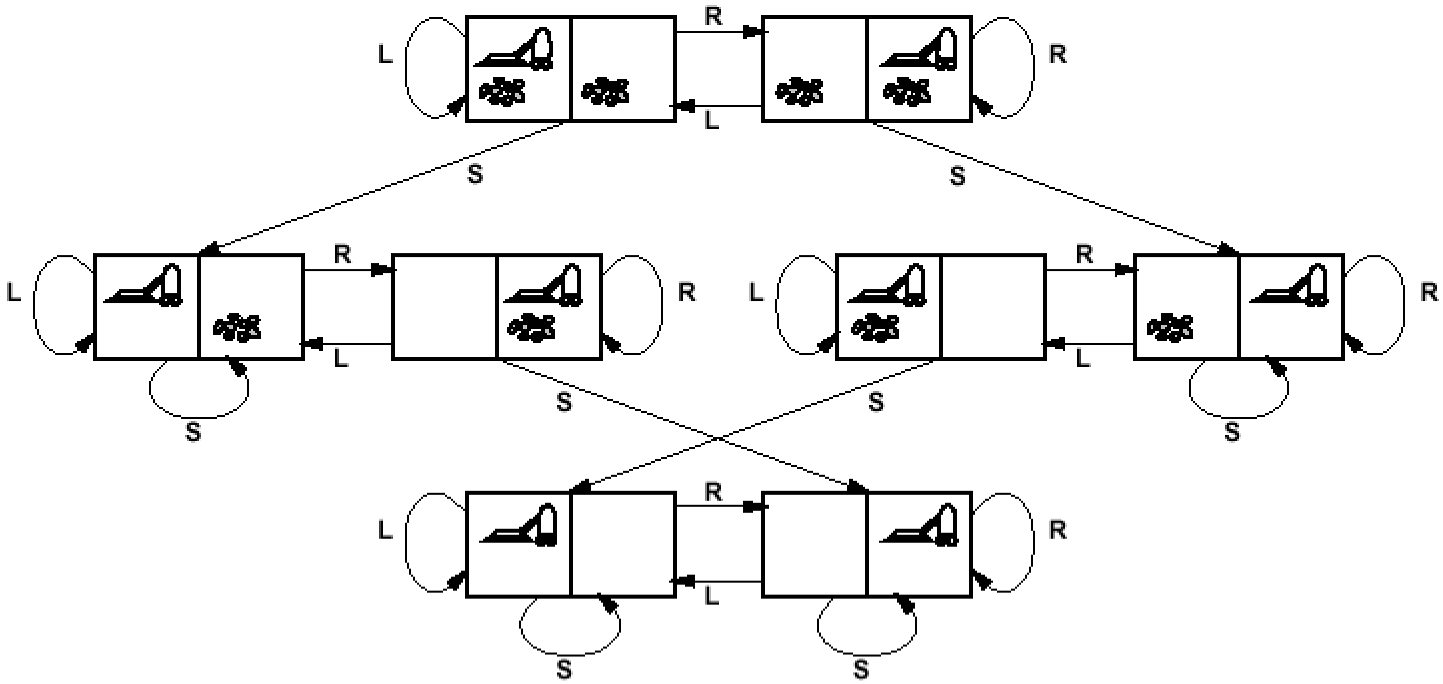
goal state

Why search algorithms? State space explodes in size!!

- 8-puzzle has 362,800 states (9!)
- 15-puzzle has 10^12 states (16!)
- 24-puzzle has 10^25 states (25!)

So, we need a principled way to look for a solution in these huge search spaces…

Back to Vacuum World



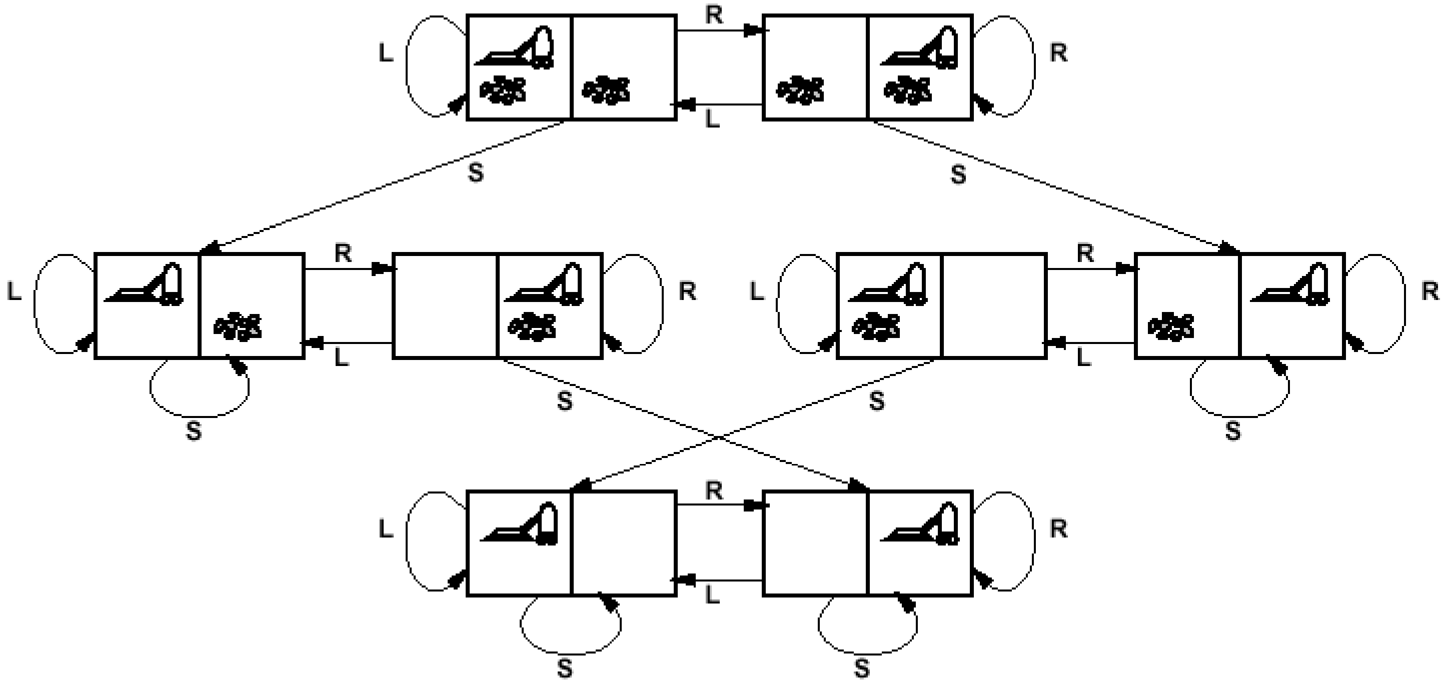
states??

operators??

goal test??

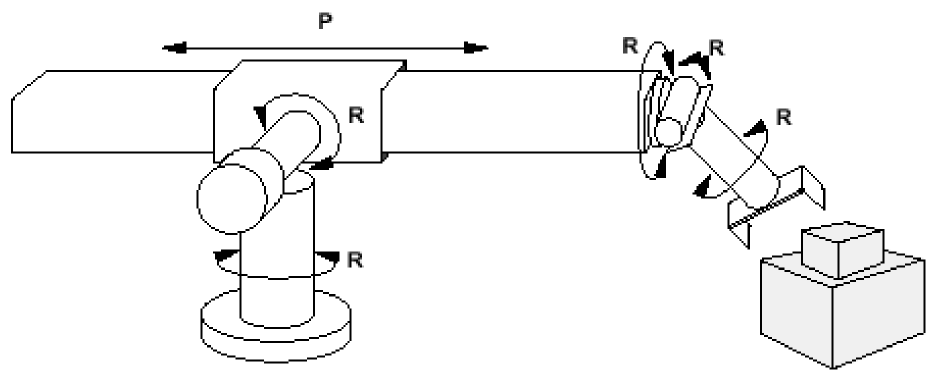
path cost??

Back to Vacuum World



<u>states</u>??: integer dirt and robot locations (ignore dirt *amounts*)
<u>operators</u>??: *Left*, *Right*, *Suck*
<u>goal test</u>??: no dirt
<u>path cost</u>??: 1 per operator

Example: Robotic Assembly



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

operators??: continuous motions of robot joints

goal test??: complete assembly *with no robot included!*

path cost??: time to execute

Real-life example: VLSI Layout

- Given schematic diagram comprising components (chips, resistors, capacitors, etc) and interconnections (wires), find optimal way to place components on a printed circuit board, under the constraint that only a small number of wire layers are available (and wires on a given layer cannot cross!)

- “optimal way”??

- minimize surface area
- minimize number of signal layers
- minimize number of vias (connections from one layer to another)
- minimize length of some signal lines (e.g., clock line)
- distribute heat throughout board
- etc.

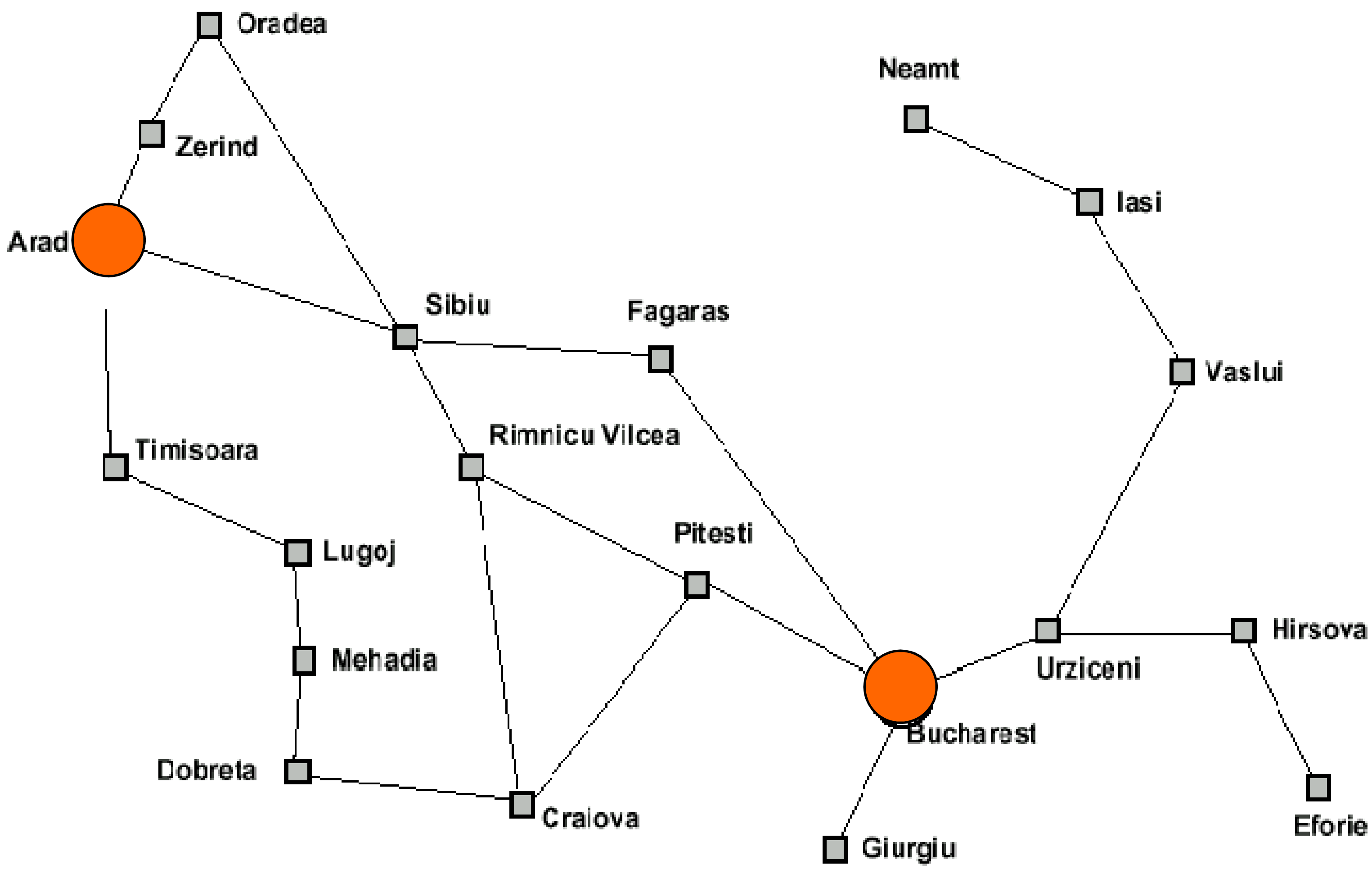
Search algorithms

Basic idea:

offline, systematic exploration of simulated state-space by generating **successors** of explored states (**expanding**)

Function General-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

Example: Traveling from Arad To Bucharest

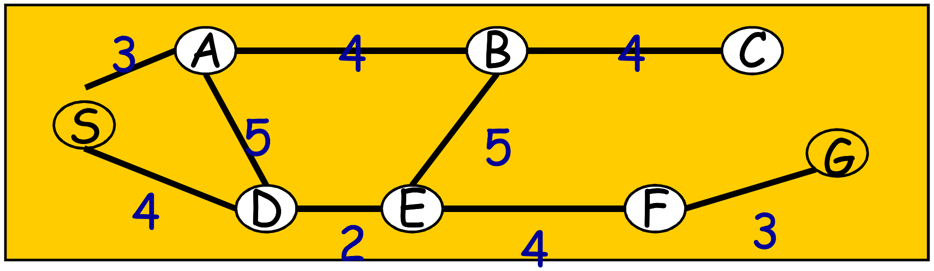


From problem space to search tree

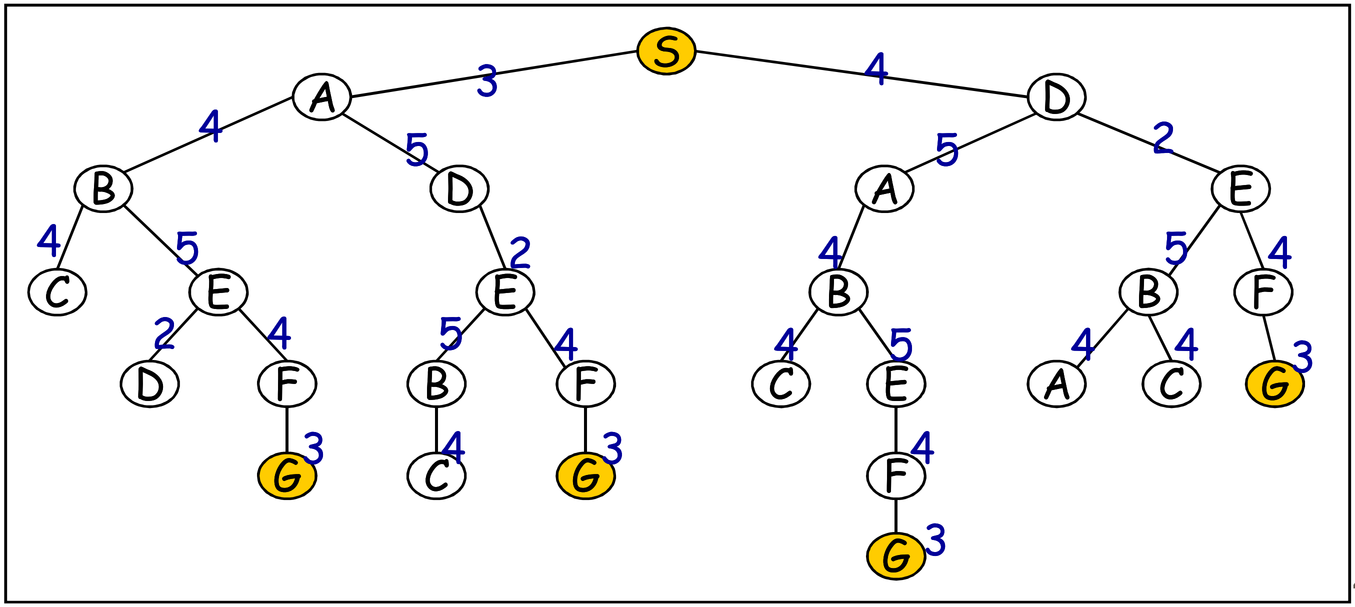
- Some material in this and following slides is from

http://www.cs.kuleuven.ac.be/~dannyd/FAI/ check it out!

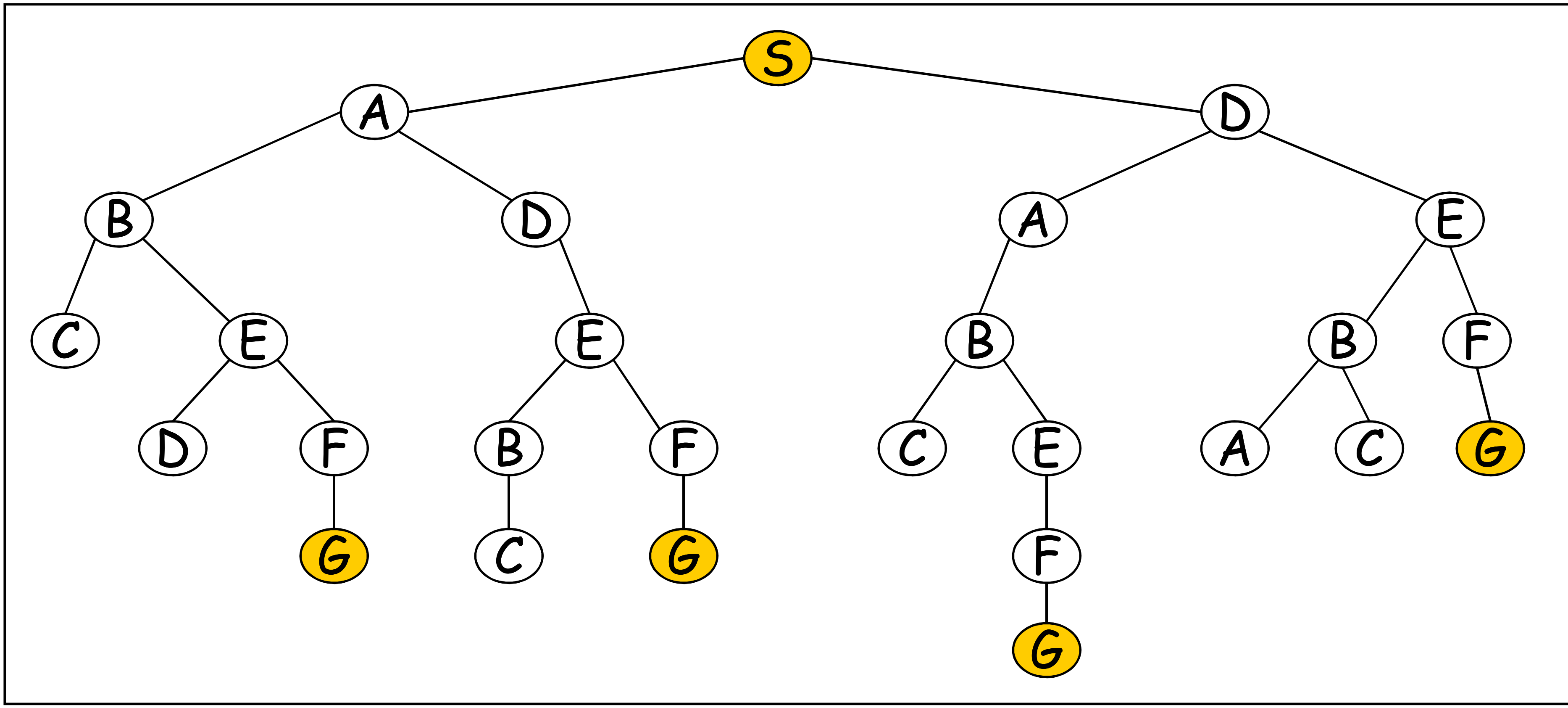
Problem space



Associated **loop-free** search **tree**

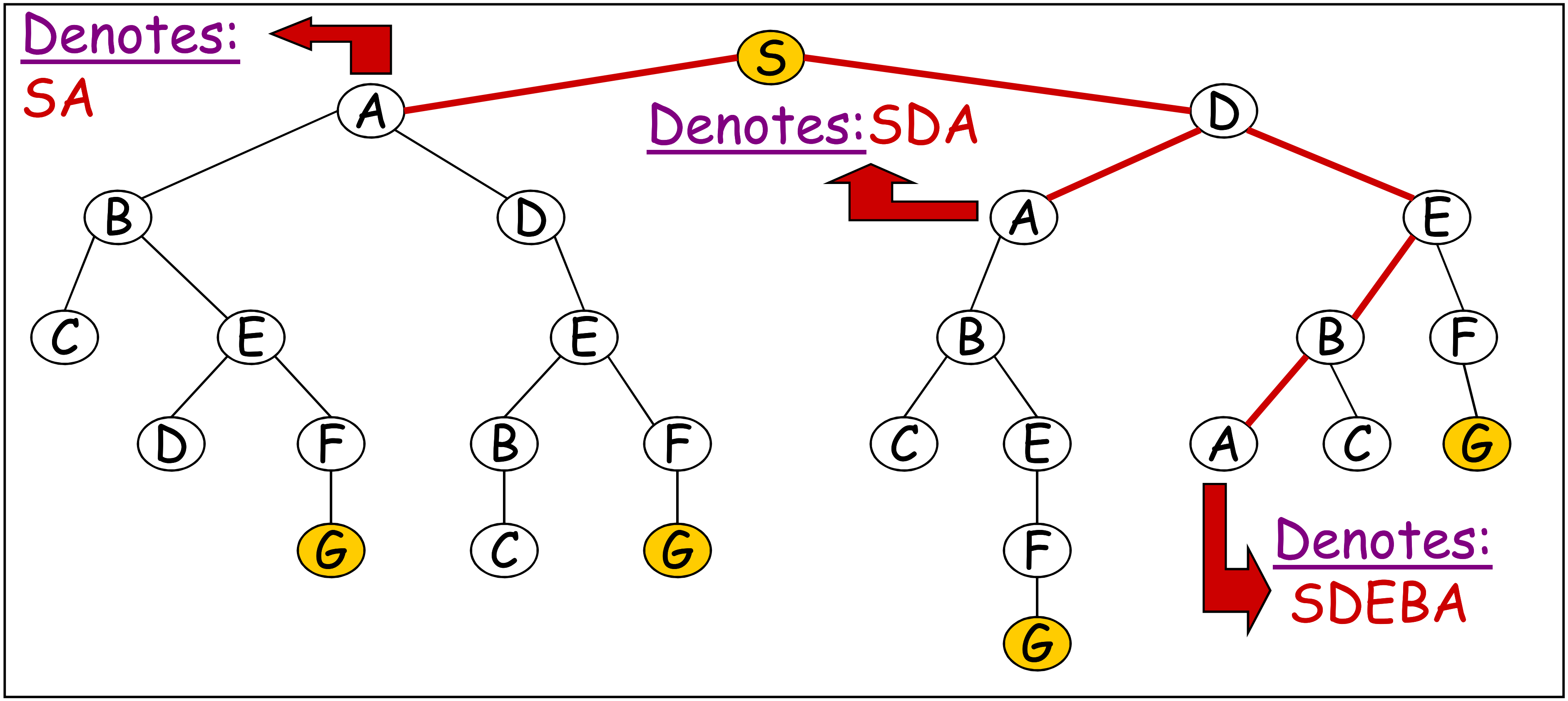


Terminology:



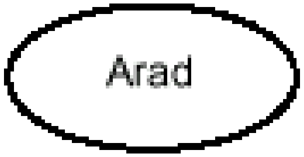
- Node, link (or edge), branch
- Parent, child, ancestor, descendant
- Root node, goal node
- Expand / Open node / Closed node / Branching factor

Paths in search trees

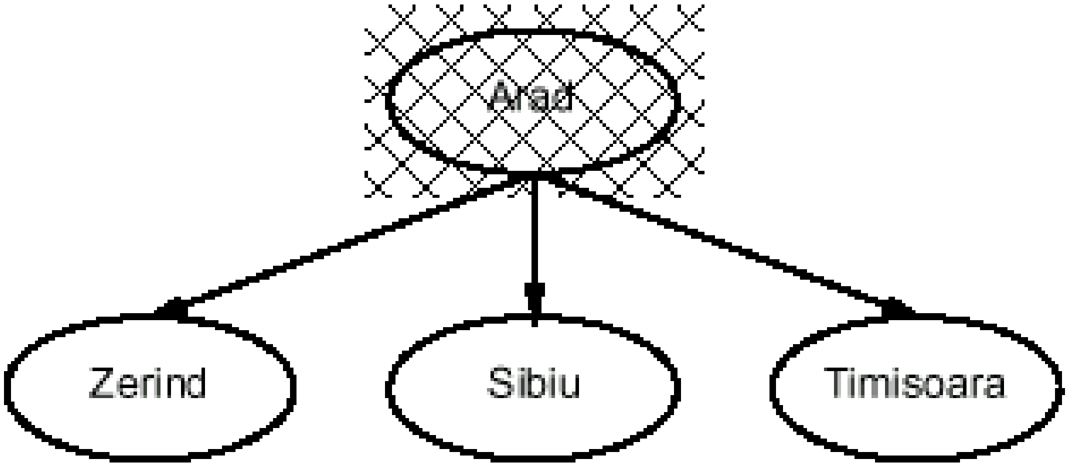


Each node really represents a unique path to the root node

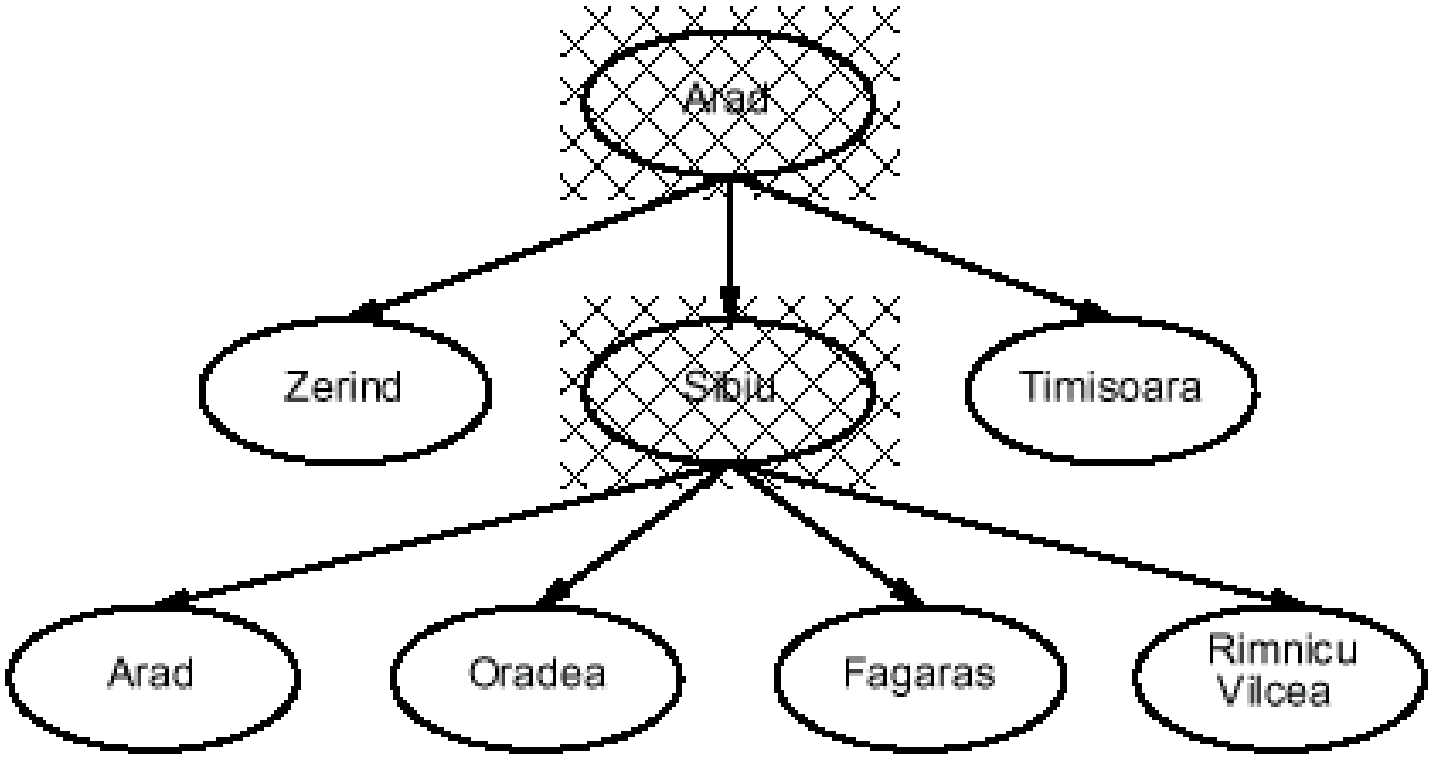
General search example



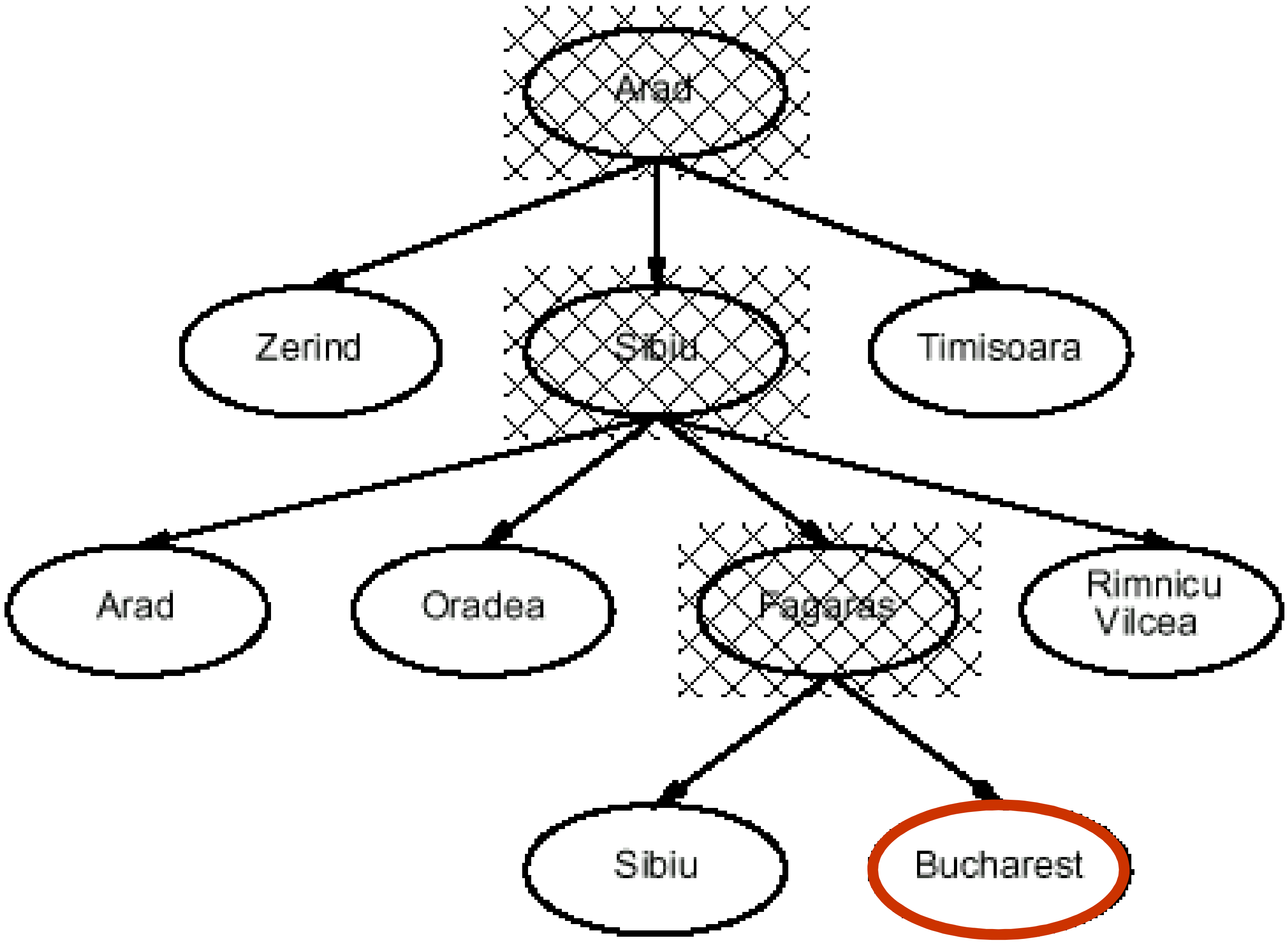
General search example



General search example



General search example



Implementation of search algorithms

```
Function General-Search(problem, Queuing-Fn) returns a solution, or failure
    nodes ← make-queue(make-node(initial-state[problem]))
    loop do
        if nodes is empty then return failure
        node ← Remove-Front(nodes)
        if Goal-Test[problem] applied to State(node) succeeds then return node
        nodes ← Queuing-Fn(nodes, Expand(node, Operators[problem]))
    end
```

Queuing-Fn(*queue, elements*) is a queuing function that inserts a set of elements into the queue and <u>determines the order of node expansion</u>. Varieties of the queuing function produce varieties of the search algorithm.

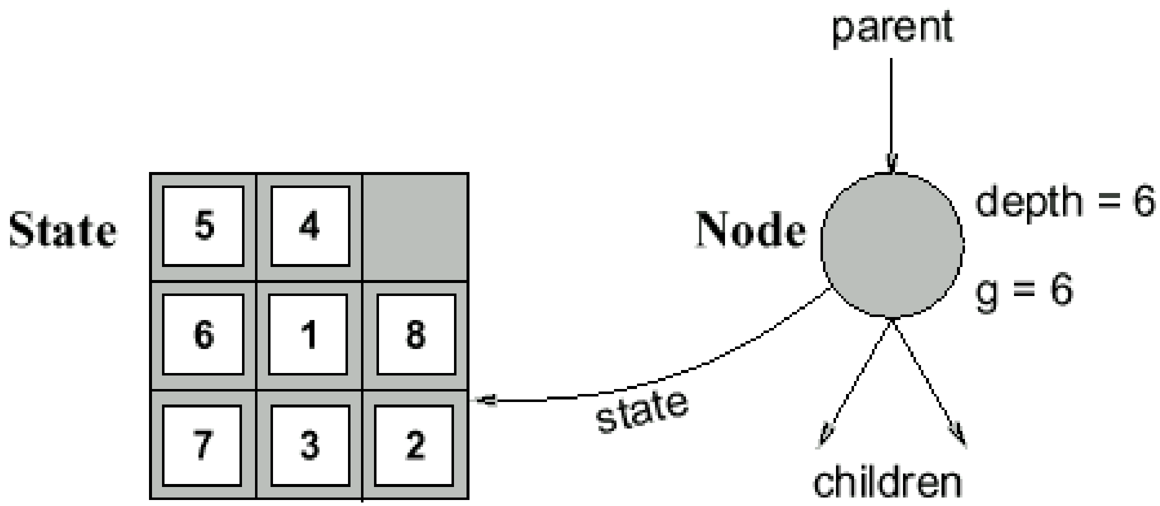
Encapsulating *state* information in *nodes*

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 OPERATORS (or SUCCESSORFN) of the problem to create the corresponding states.

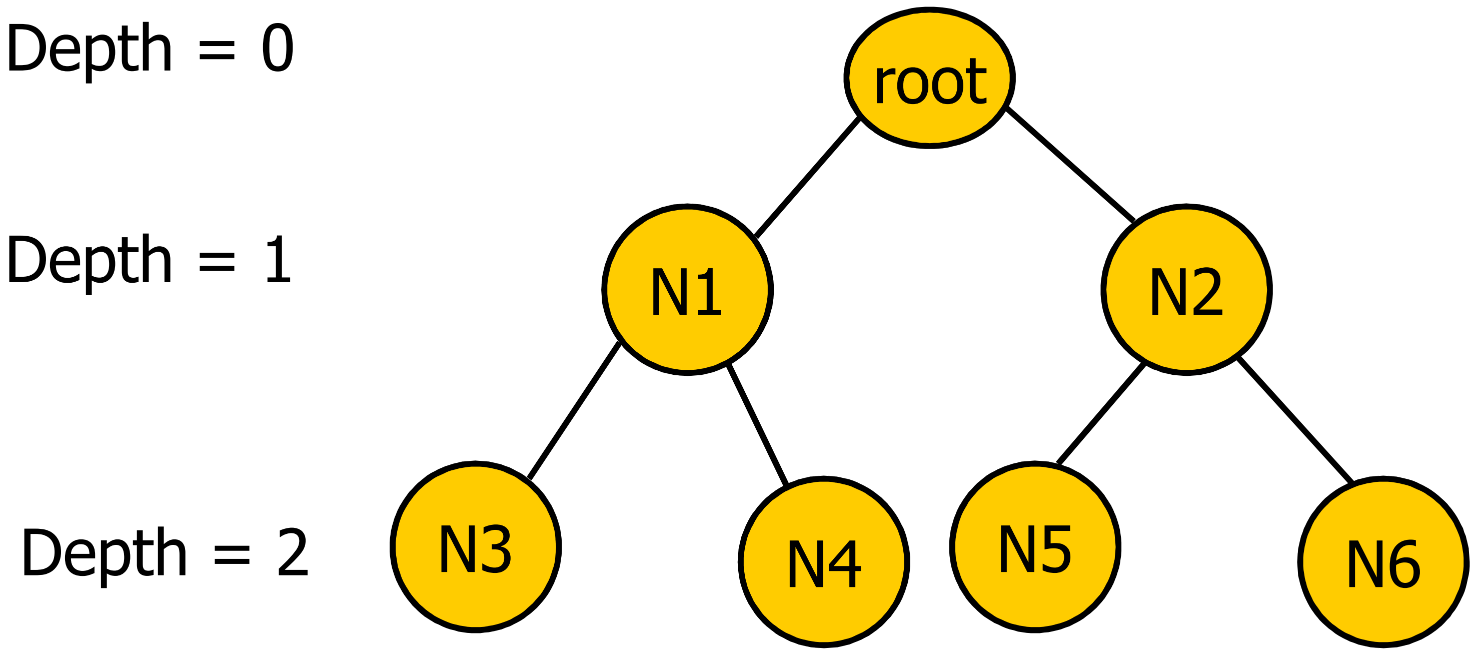
Evaluation of search strategies

- 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)

Binary Tree Example



Number of nodes: $n = 2^{\text{max depth}}$

Number of levels (max depth) = log(n) (could be n)