# Principles of Intelligent Systems: Problem Solving*

## Lecture 3

*These slides are taken from the Chapter 3 slides of Russell and Norvig's *Artificial Intelligence: A modern approach* (http://aima.eecs.berkeley.edu/slides-pdf/)

# Outline

- ♢ Problem-solving agents
- ♢ Problem types
- ♢ Problem formulation
- ♢ Example problems

# Problem-solving agents

Restricted form of general 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
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← RECOMMENDATION(seq, state)
    seq ← REMAINDER(seq, state)
    return action
```

Note: this is *offline* problem solving; solution executed "eyes closed."
*Online* problem solving involves acting without complete knowledge.

# Example: Romania

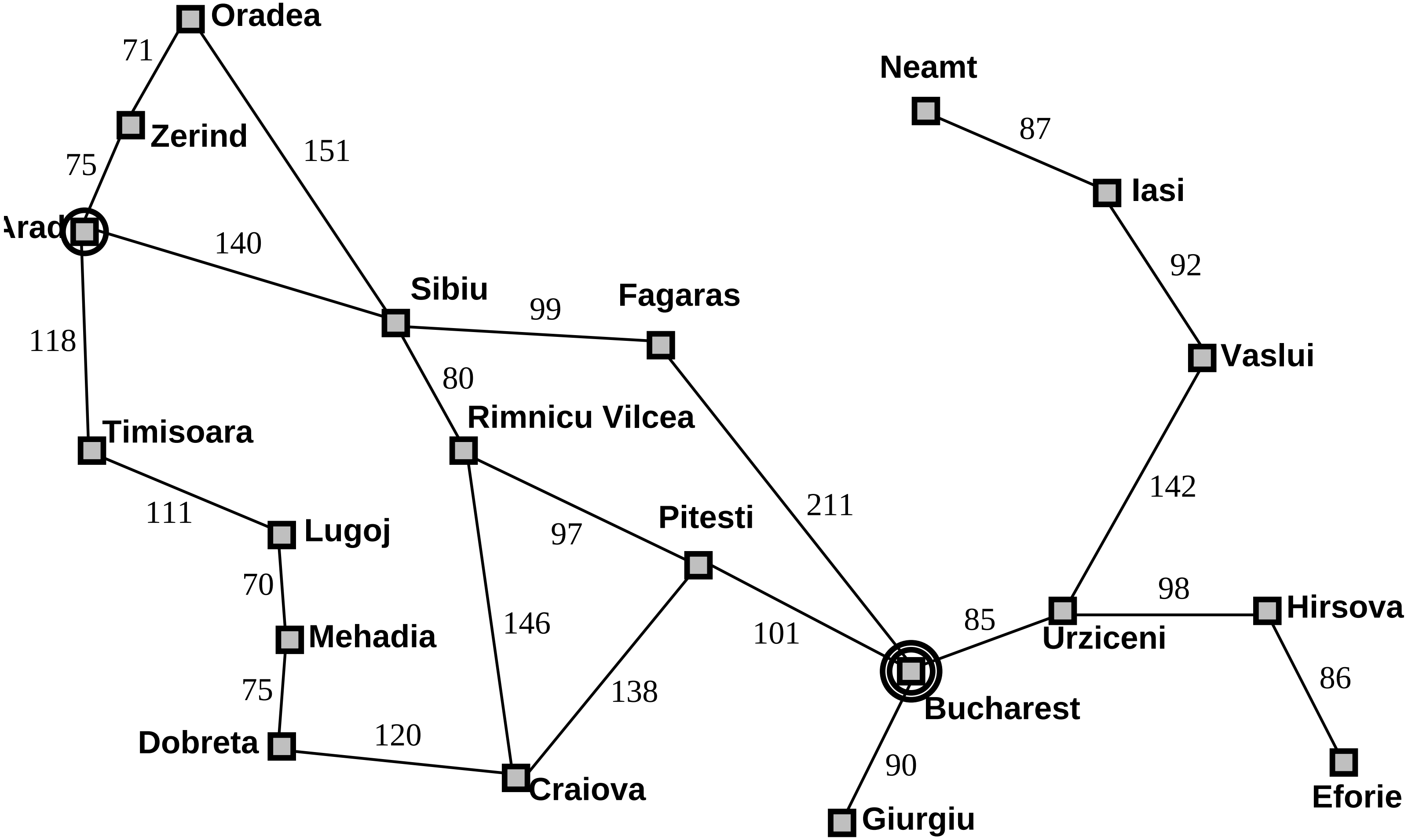
On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

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: Romania



# Problem types

Deterministic, fully observable $\Longrightarrow$ *single-state 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

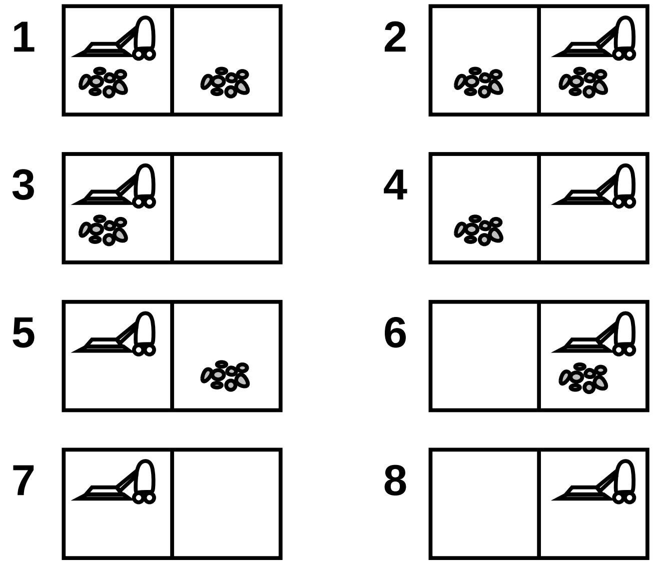
Nondeterministic and/or partially observable $\Longrightarrow$ *contingency problem*

percepts provide *new* information about current state
solution is a *tree* or *policy*
often *interleave* search, execution

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

# Example: vacuum world

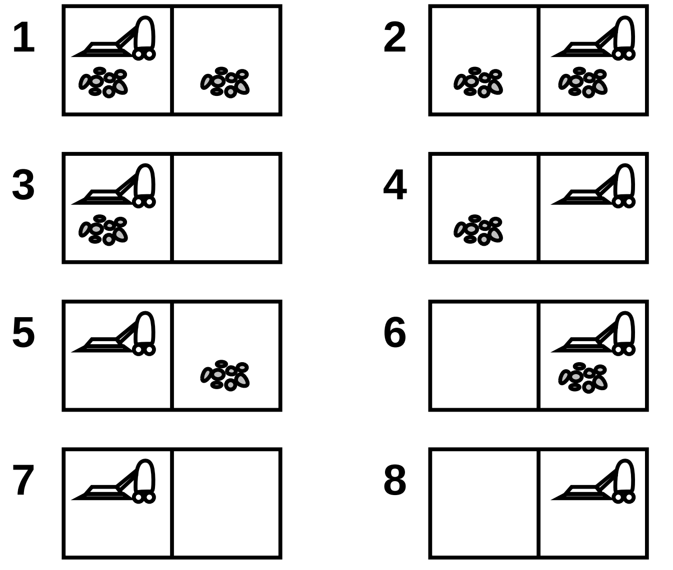
Single-state, start in #5. Solution??



# Example: vacuum world

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??

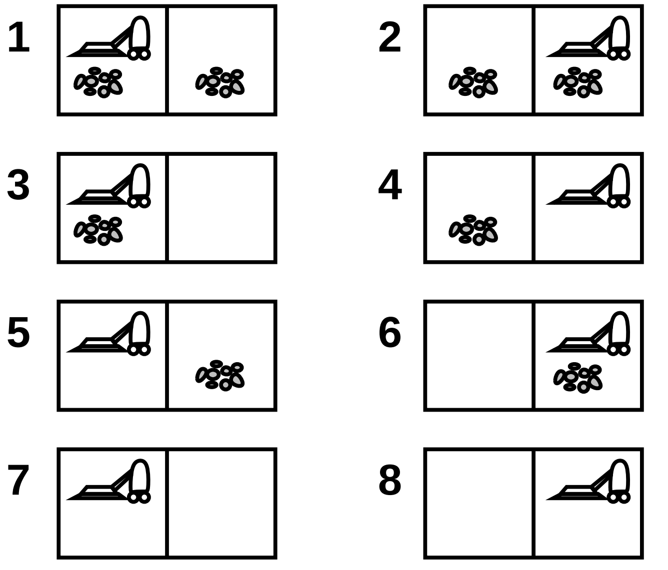


# Example: vacuum world

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??

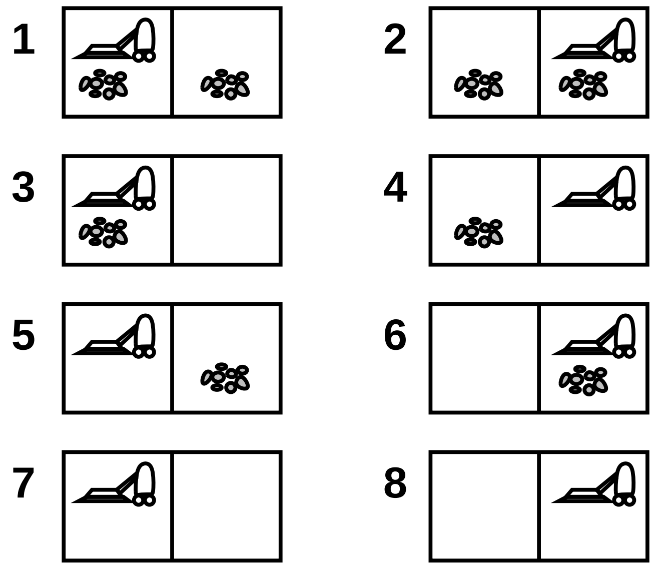


# Example: vacuum world

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??
$[Right, \textbf{if } dirt \textbf{ then } Suck]$



# Single-state problem formulation

A *problem* is defined by four items:

*initial state* e.g., "at Arad"

*successor function* $S(x)$ = set of action–state pairs
e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind \rangle, \ldots\}$

*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 actions executed, etc.
$c(x, a, y)$ is the step cost, assumed to be $\geq 0$

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
$\Rightarrow$ state space must be *abstracted* for problem solving

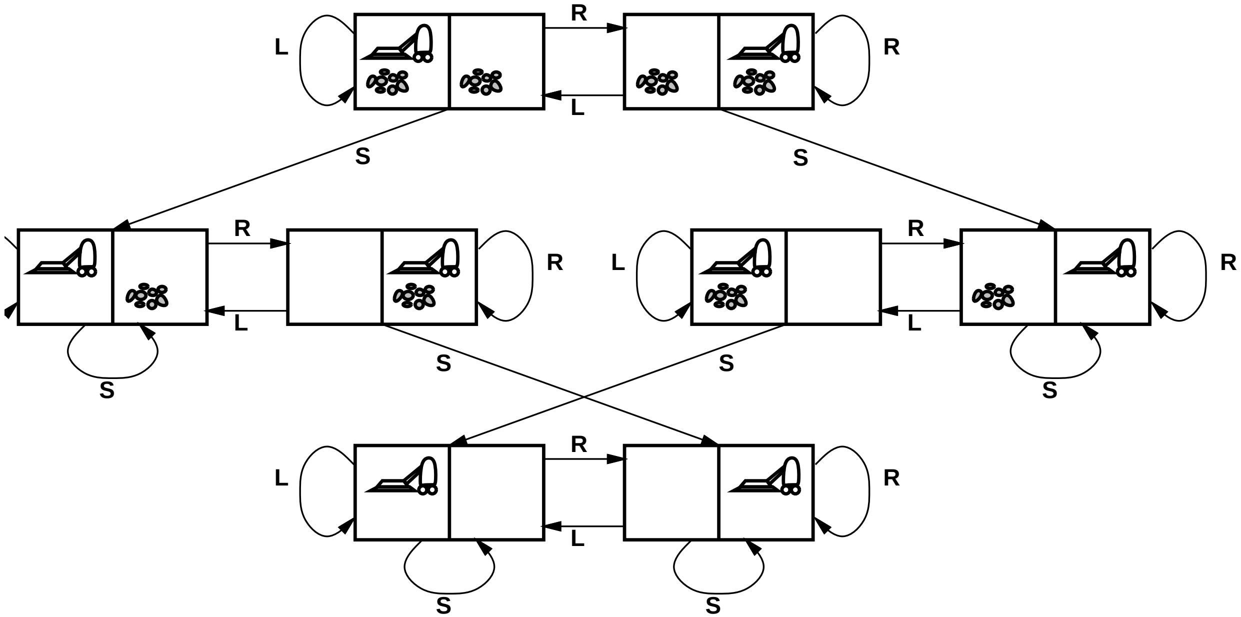
(Abstract) state = set of real states

(Abstract) action = complex combination of real actions
e.g., "Arad $\rightarrow$ 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 =
set of real paths that are solutions in the real world

Each abstract action should be "easier" than the original problem!

# Example: vacuum world state space graph



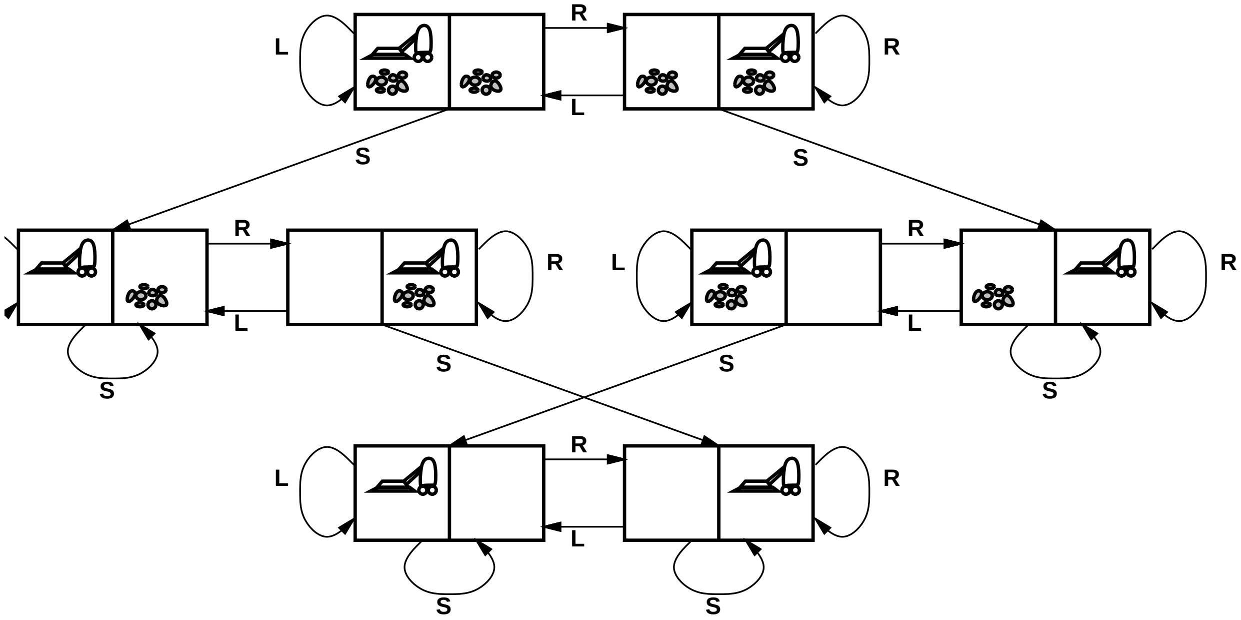

states??

actions??

goal test??

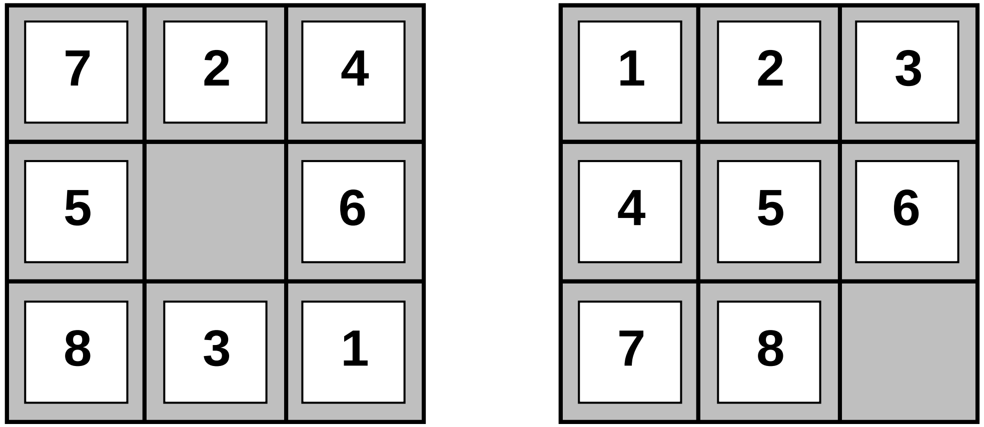
path cost??

# Example: vacuum world state space graph




states??: integer dirt and robot locations (ignore dirt *amounts*)
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal test??: no dirt
path cost??: 1 per action (0 for $NoOp$)

# Example: The 8-puzzle




**Start State** **Goal State**

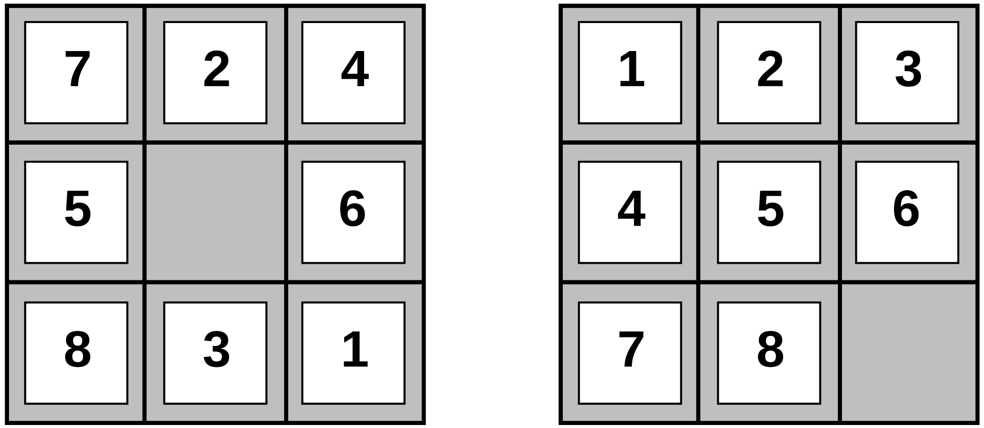
states??

actions??

goal test??

path cost??

# Example: The 8-puzzle

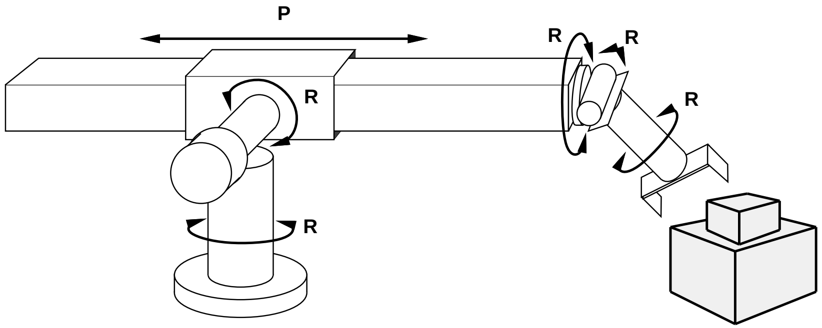



**Start State** **Goal State**

states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??: 1 per 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 *with no robot included!*

path cost??: time to execute

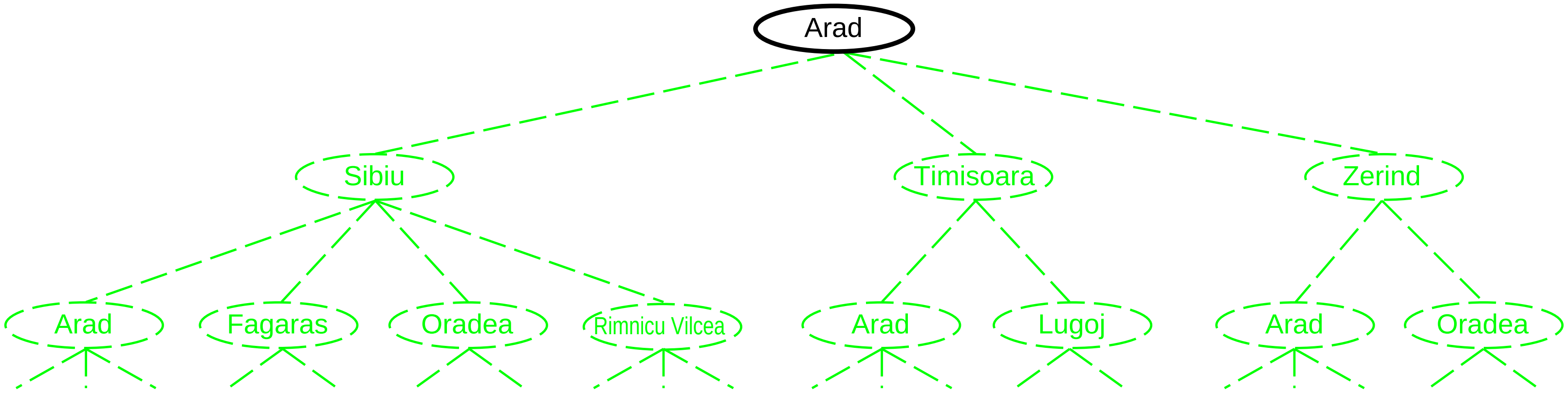
# Tree search algorithms

Basic idea:

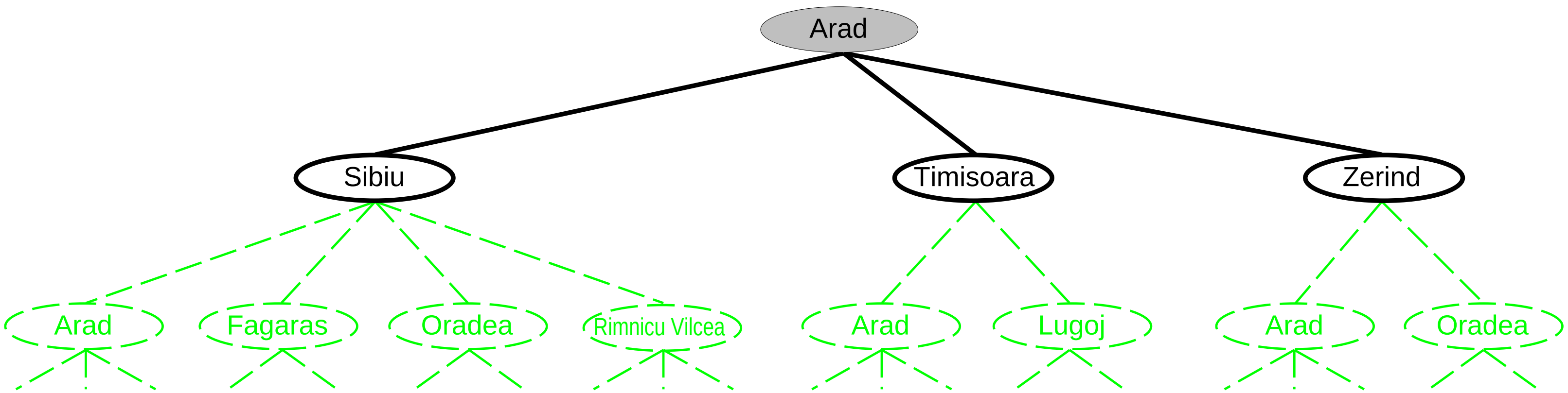
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. *expanding* states)

**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
  **end**

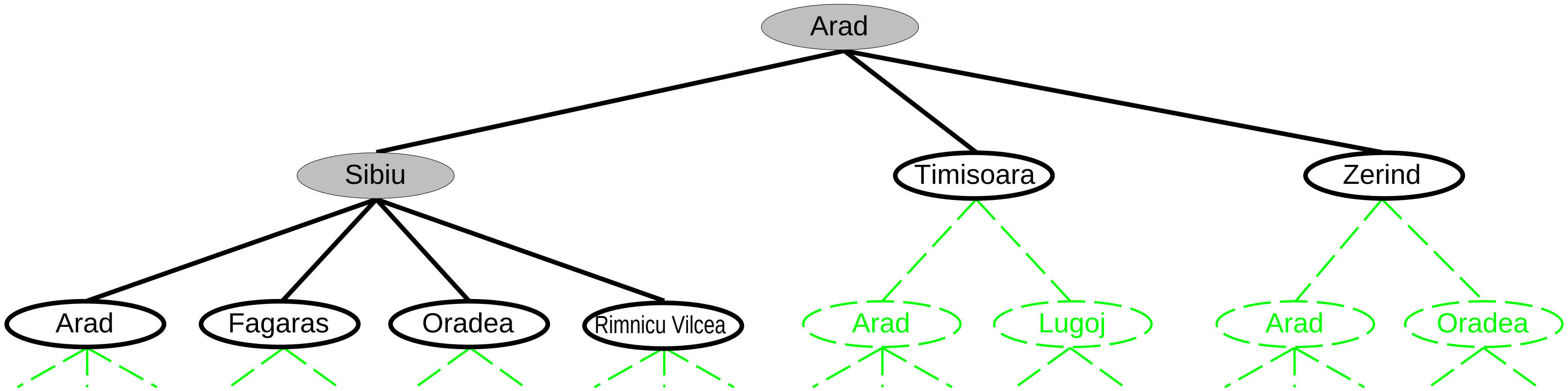
# Tree search example



# Tree search example

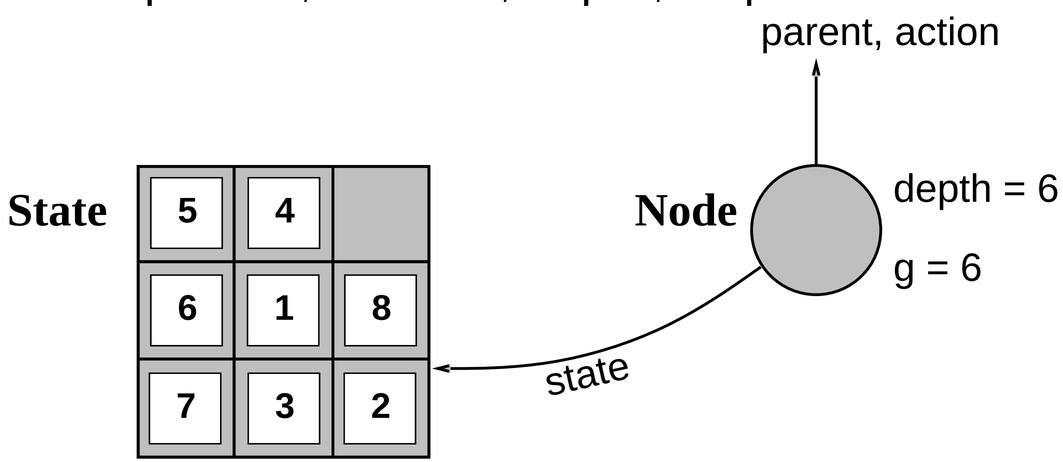


# Tree search example



# Implementation: states vs. 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.

# 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*] applied to State(*node*) succeeds **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*

# Search strategies

A strategy is defined by picking the *order of node expansion*

Strategies are evaluated along the following dimensions:

- completeness—does it always find a solution if one exists?
- time complexity—number of nodes generated/expanded
- space complexity—maximum number of nodes in memory
- optimality—does it 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 $\infty$)

# Summary

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

Given a suitable state space, problem solving can be performed using a tree search strategy