

Artificial Intelligence

# SOLVING PROBLEMS BY SEARCHING

Nguyễn Tiến Huy  
ntienhuy@fit.hcmus.edu.vn

# Outline

---

- Problem-solving agents
- Example problems
- Searching for solutions

# Problem-solving Agents

- *Well-defined problems and solutions*
- *Formulating problems*

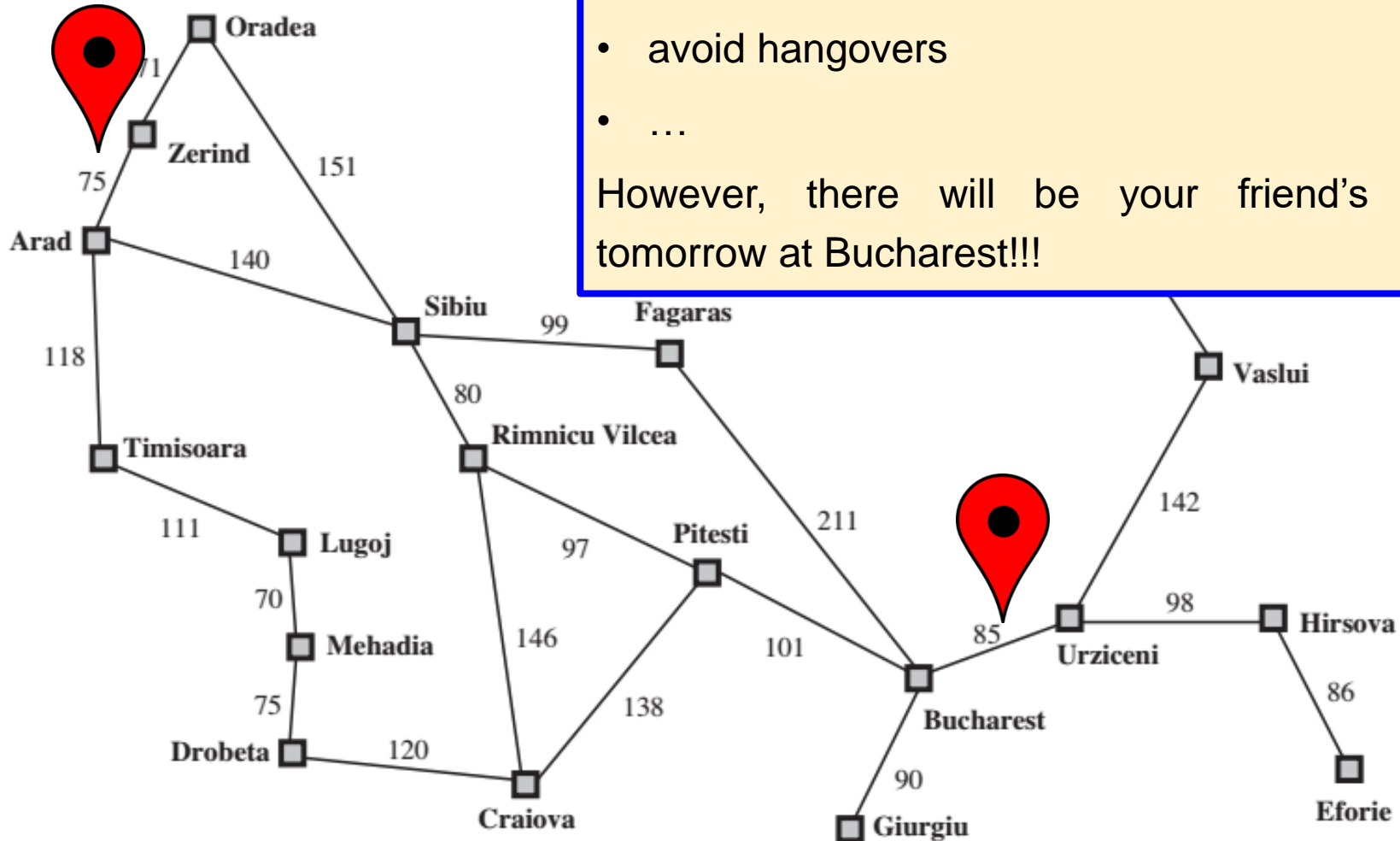


# A touring holiday in Romania

What makes a good trip at Arad?

- take in the sights, enjoy the foods, etc.
- avoid hangovers
- ...

However, there will be your friend's wedding tomorrow at Bucharest!!!



# Goal-based agents

---

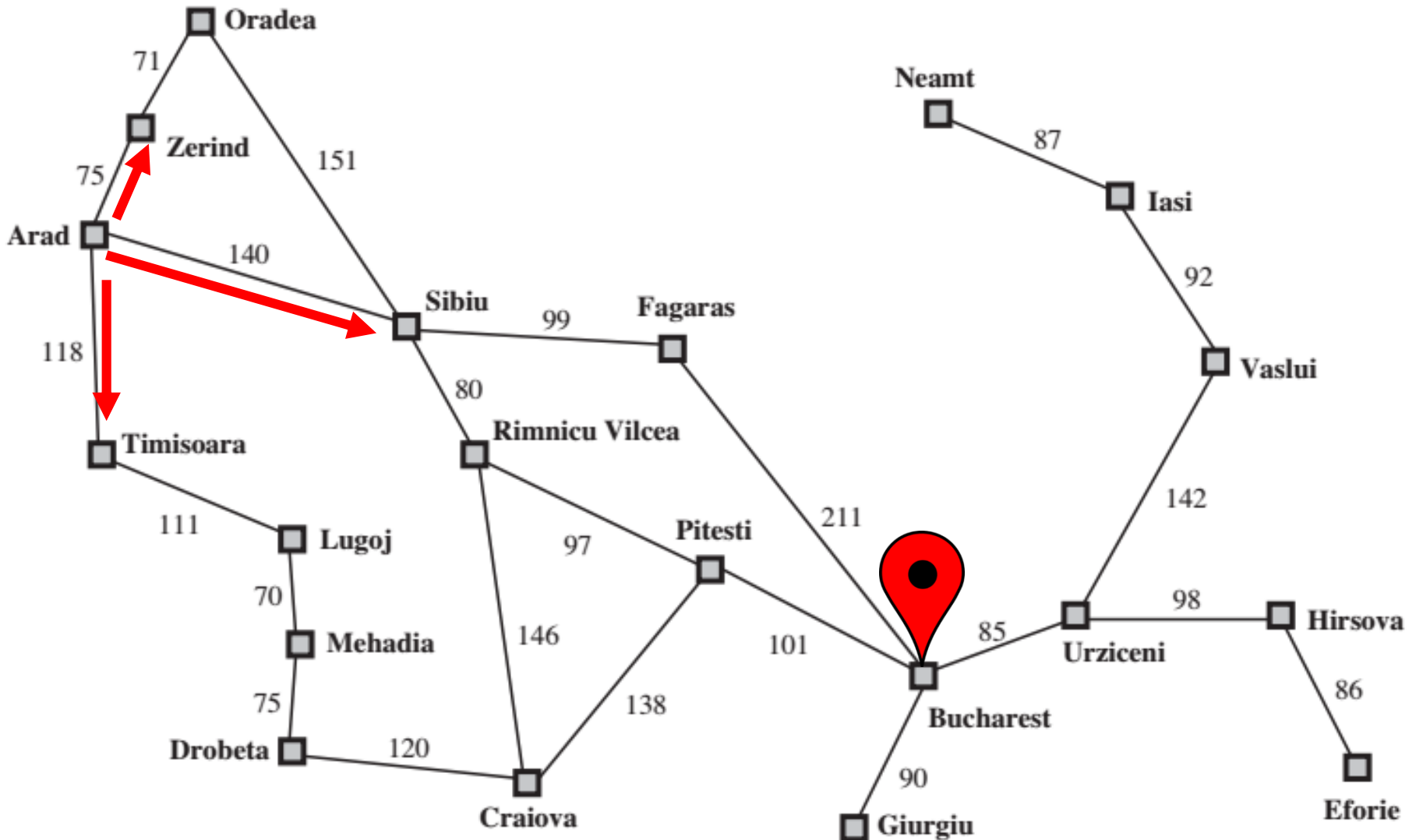
- Intelligent agents **maximize their performance measure**.
  - Performance measure includes different factors.
  - Making decisions involves many tradeoffs.
- **Goals** help organize behavior by **limiting the objectives** that the agent is trying to achieve and **the actions** it considers.

# Problem formulation

- Consider a **goal** to be a set of world states in which the objective is satisfied.
- **Problem formulation** is the **process of deciding what actions and states to consider, given a goal.**
  - E.g., state: being in a particular town, actions: driving from one town to another → goal state: being in Bucharest



# Traveling from Arad to Bucharest





# Goal-based agents in Romania

---

- The agent initially does not know which road to follow  
→ **unknown environment**, try an action randomly
- Suppose the agent has a map of Romania.
- The agent discovers many hypothetical journey and finds a journey that eventually gets to Bucharest.





# Properties of the Romania environment

---

- **Observable**

- Each city has a sign indicating its presence for arriving drivers.
- The agent always knows the current state.

- **Discrete**

- Each city is connected to a small number of other cities.
- There are only finitely many actions to choose from any given state.

- **Known**

- The agent knows which states are reached by each action.

- **Deterministic**

- Each action has exactly one outcome.

# Solving problem by searching

---

- **Search:** the process of looking for a **sequence of actions** that reaches the goal
- A **search algorithm** takes a problem as input and returns a **solution** in the form of an action sequence.
- **Execution phase:** once a solution is found, the recommended actions are carried out.

# Solving problem by searching

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: 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)
    if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
return action
```

# Well-define problems and solutions

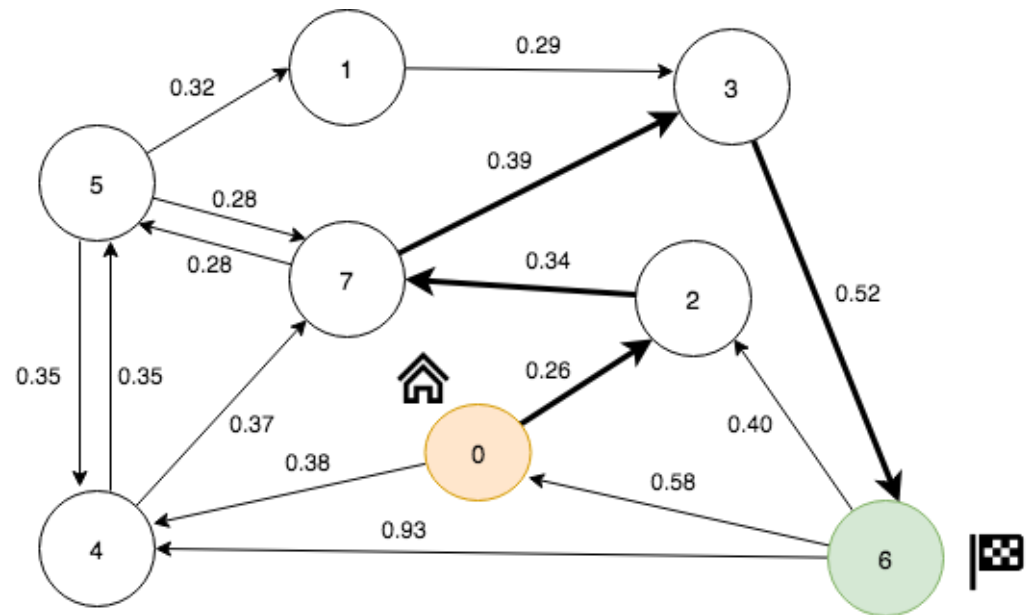
---

- A problem can be defined formally by five components.
- **Initial state:** in which the agent starts
  - E.g., the agent in Romania has its initial state described as  $In(Arad)$
- **Actions:** the possible actions available to the agent
  - E.g.,  $ACTION(Arad) = \{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$
- **Transition model:** what each action does
  - E.g.,  $Result(In(Arad), Go(Zerind)) = In(Zerind)$
  - **Successor:** a state reachable from a given state by a single action

# The state space

- The set of states that are **reachable from the initial state** by any sequence
  - It is implicitly defined by the initial state, actions, and transition model
- **Directed graph** – nodes are states and the links between nodes are actions.

A **path** in the state space is a sequence of states connected by a sequence of actions.



# Well-define problems and solutions

---

- **Goal test:** determine whether a given state is a goal state
  - The goal is specified by either an explicit set of possible goal states or an abstract property.
  - E.g.,  $In(Bucharest)$ , checkmate
- **Path cost:** a function that sets a numeric cost to each path
  - Nonnegative, reflecting the agent's performance measure
  - E.g.,  $c(In(Arad), Go(Zerind), In(Zerind)) = 75$
- An **optimal solution** has the lowest path cost.

# Formulating problems by abstraction

- The process of removing detail from a representation
- Navigation example: how do we define states and actions?
  - First abstract “the big picture”, i.e., solve a map problem
    - Nodes = cities, links = roads connecting cities (a high-level description)



VS.



- Later worry about details
  - E.g., traveling companions, road condition, weather, etc. Similarly for driving action – time, fuel consumption, pollution, etc.



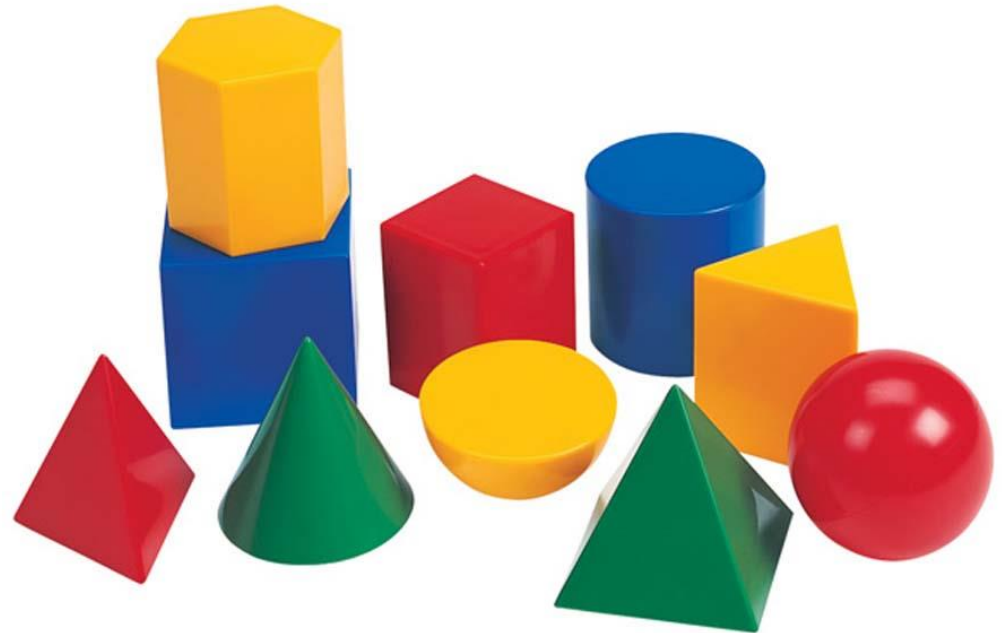
# Formulating problems by abstraction

- Abstraction creates an approximate and simplified model of the world, which is **critical for automated problem solving**.
- A good abstraction **removes as much details as possible**, while **retaining validity**, and ensure that the abstract actions are **easy to be carried out**.



# Example problems

- *Toy problems*
- *Real-world problems*



# Toy problems vs. Real-world problems

## Toy problems

Exercise problem-solving methods

Compare performance of methods

Concise, exact description

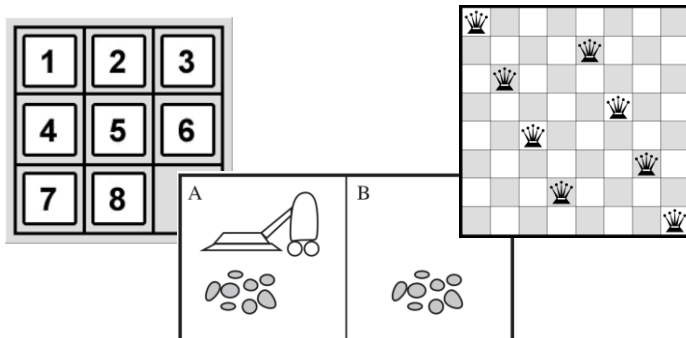
E.g., 8-puzzle, 8-queens problem, cryptarithmic, vacuum world, missionaries and cannibals, simple route finding

## Real-world problems

Bring solutions to practical issues

No single, agreed-upon description

E.g., route finding, touring and traveling salesperson problems, VLSI layout, robot navigation, assembly sequencing

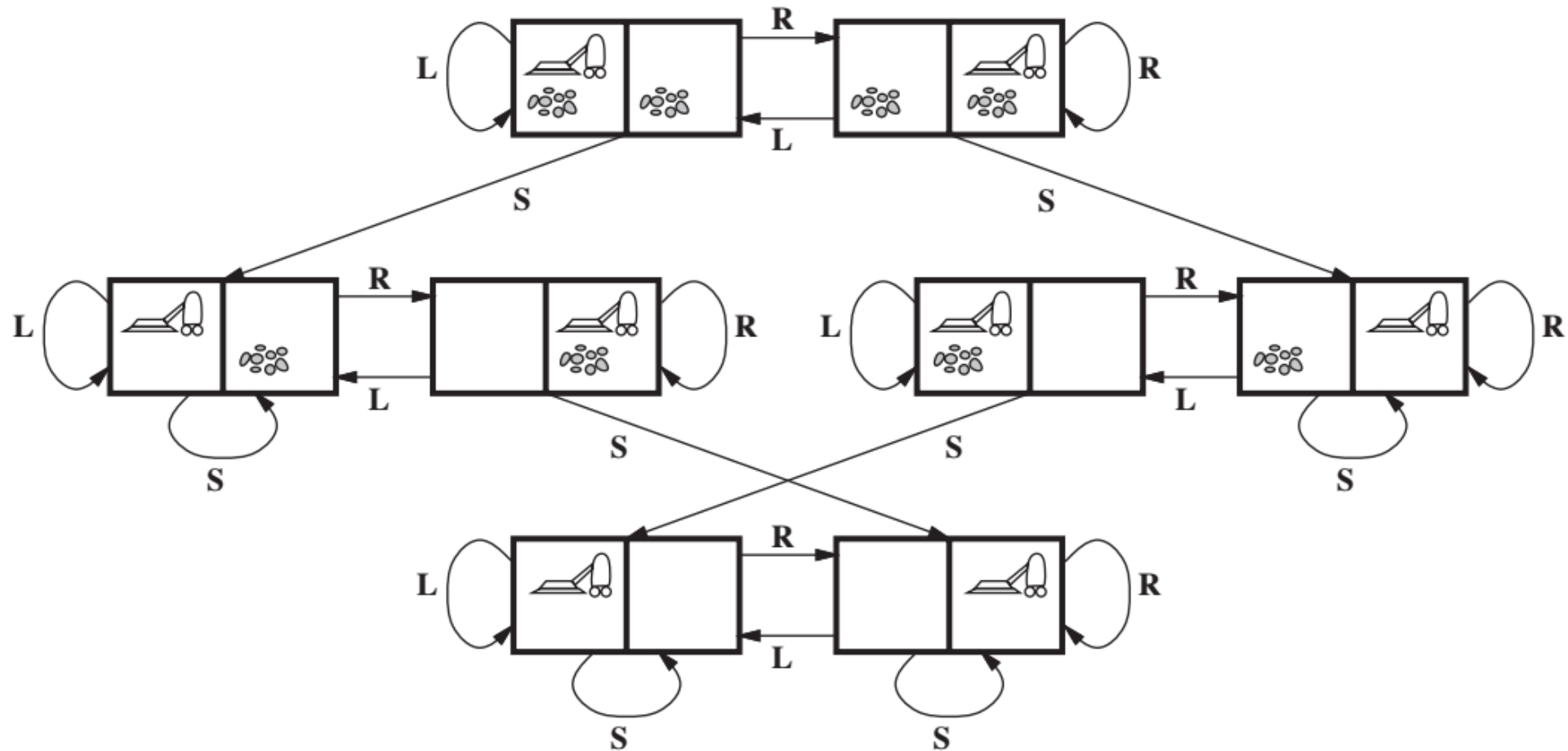


# The Vacuum-cleaner world

---

- **States:** determined by the agent location and the dirt locations
  - $2 \times 2^2 = 8$  possible world states ( $n \times 2^n$  in general)
- **Initial state:** Any state can be designated as the initial state.
- **Actions:** Left, Right, Suck, and Do nothing
  - Larger model may include Up and Down, etc.
- **Transition model:** The actions have their expected effects.
  - Except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
- **Goal test:** whether all the squares are clean
- **Path cost:** each step costs 1

# The Vacuum-cleaner world



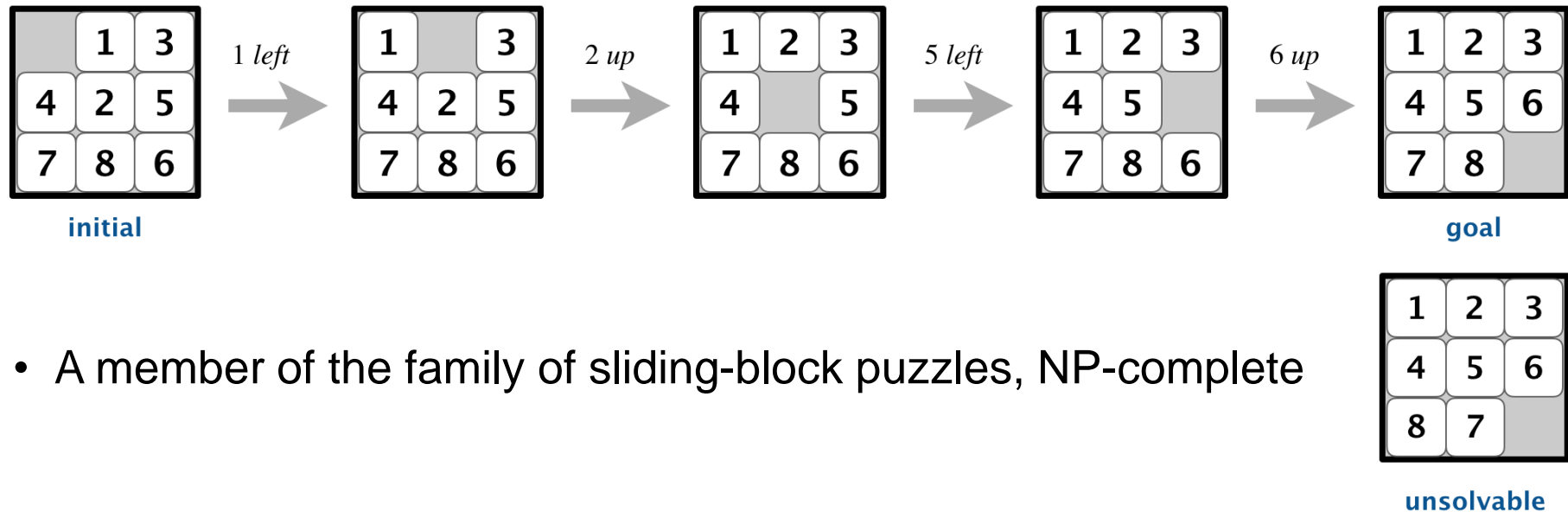
The state space for the vacuum world  
Links denote actions: L = Left, R = Right, S = Suck.

# The 8-puzzle

---

- **States:** the location of each of the eight tiles and the blank
- **Initial state:** any state can be designated as the initial state
- **Actions:** movements of the blank space
  - Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is
- **Transition model:** return a resulting state given a state and an action
- **Goal test:** check whether the state matches the goal configuration
- **Path cost:** each step costs 1

# The 8-puzzle

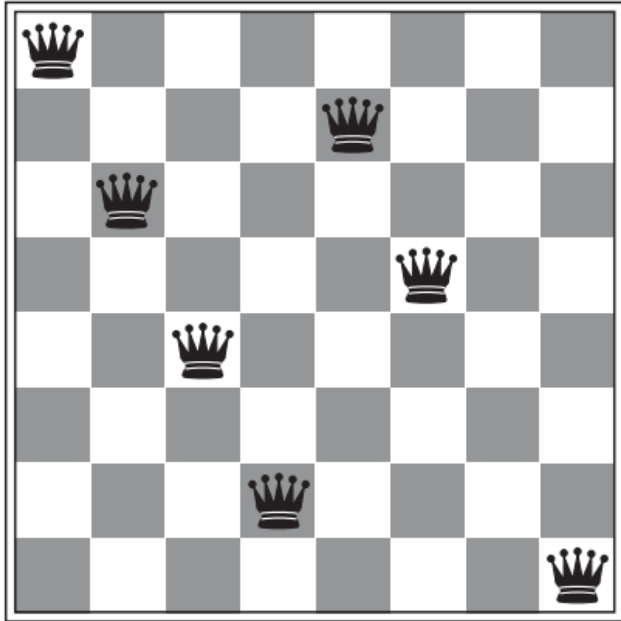


- A member of the family of sliding-block puzzles, NP-complete
- 8-puzzle:  $9!/2 = 181,440$  reachable states → easily solved.
- 15-puzzle: 1.3 trillion ( $10^{12}$ ) states → optimally solved in a few millisecs
- 24-puzzle: around  $10^{25}$  states → optimally solved in several hours



# The 8-queens

---



- **Incremental formulation:** add a queen step-by-step to the empty initial state
- **Complete-state formulation:** start with all 8 queens on the board and move them around
- The path cost is trivial because only the final state counts

# The 8-queens: Incremental formulation

---

- **States:** any arrangement of 0 to 8 queens on the board
- **Initial state:** no queens on the board
- **Actions:** add a queen to any empty square
- **Transition model:** returns the board with a queen added to the specified square
- **Goal test:** 8 queens are on the board, none attacked
- $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$  possible sequences to investigate

# The 8-queens: Incremental formulation

---

- A better formulation would prohibit placing a queen in any square that is already attacked.
- **States:** All possible arrangements of  $n$  queens ( $0 \leq n \leq 8$ ), one per column in the leftmost  $n$  columns, with no queen attacking another
- **Actions:** Add a queen to any square in the leftmost empty column such that it is not attacked by other queens
- 8-queens: from  $1.8 \times 10^{14}$  states to just 2,057 states
- 100-queens: from  $10^{400}$  states to about  $10^{52}$  states

# Knuth's 4 problem

- Devised by Donald Knuth (1964)
- Illustration of how infinite state spaces can arise
- **Knuth's conjecture:** Starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

- **States:** positive numbers.
- **Initial state:** 4
- **Actions:** apply factorial, square root, or floor operation (factorial for integers only)
- **Transition model:** given by the operations' mathematical definitions
- **Goal test:** whether it is the desired positive integer

# The route-finding problem

---

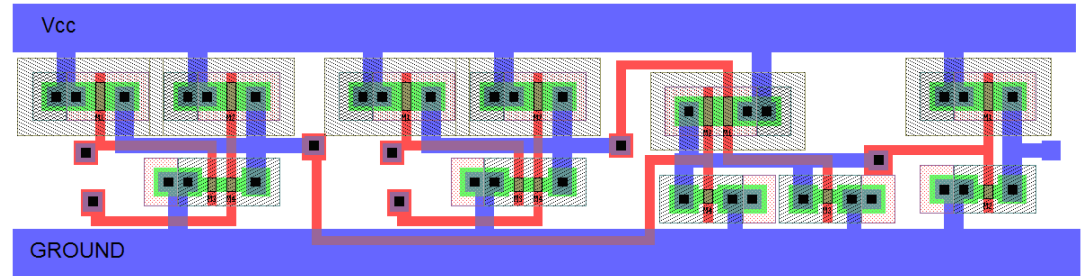
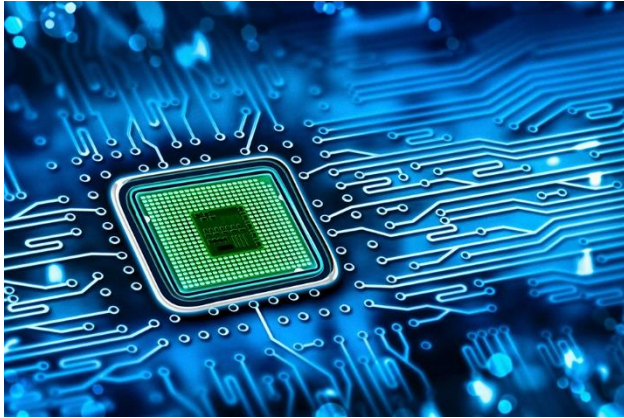
- Consider the airline travel problems solved by a travel-planning Web site.
- **States:** a location (e.g., an airport) and the current time
  - Extra information about “historical” aspects, e.g., previous segments, fare bases, statuses as domestic or international, are needed.
- **Initial state:** specified by the user’s query
- **Actions**
  - Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed
- **Transition model**
  - Current location: the flight’s destination, current time: the flight’s arrival time
- **Goal test:** whether the agent is at the destination specified by the user.
- **Path cost:** depend on different factors of the performance measure

# The touring problems

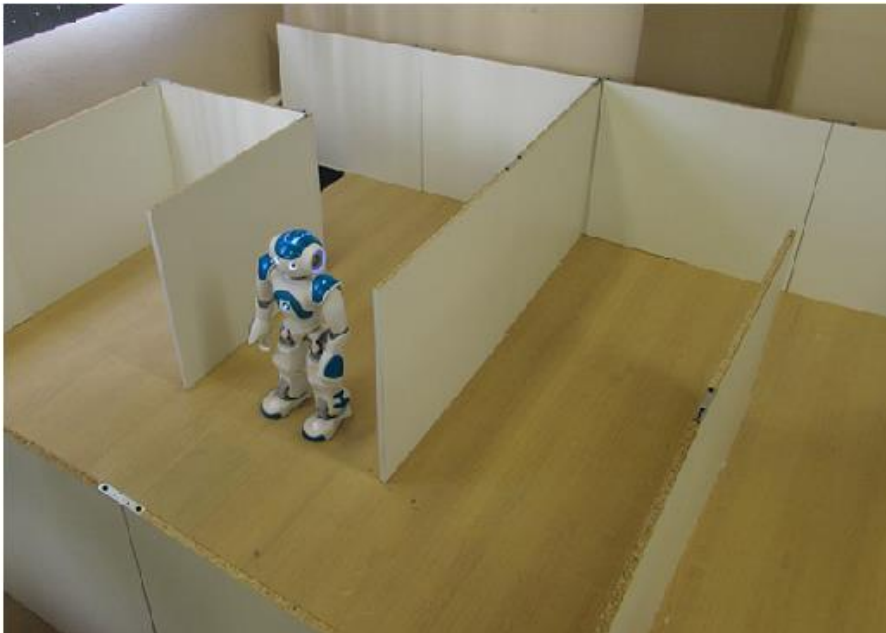
---

- **Actions:** correspond to trips between adjacent cities
- Each **state** must include not just the current location but also the set of cities the agent has visited.
- For example, the touring holiday in Romania
  - In(Bucharest), Visited({Bucharest}): initial state
  - In(Vaslui), Visited({Bucharest, Urziceni, Vaslui}): intermediate state
  - Goal test: whether in Bucharest and all 20 cities have been visited.
- **Traveling salesperson problem (TSP):** NP-hard
  - Every city must be visited exactly once, and the tour is shortest.
  - Plan movements of automatic circuit-board drills or stocking machines on shop floors, etc.

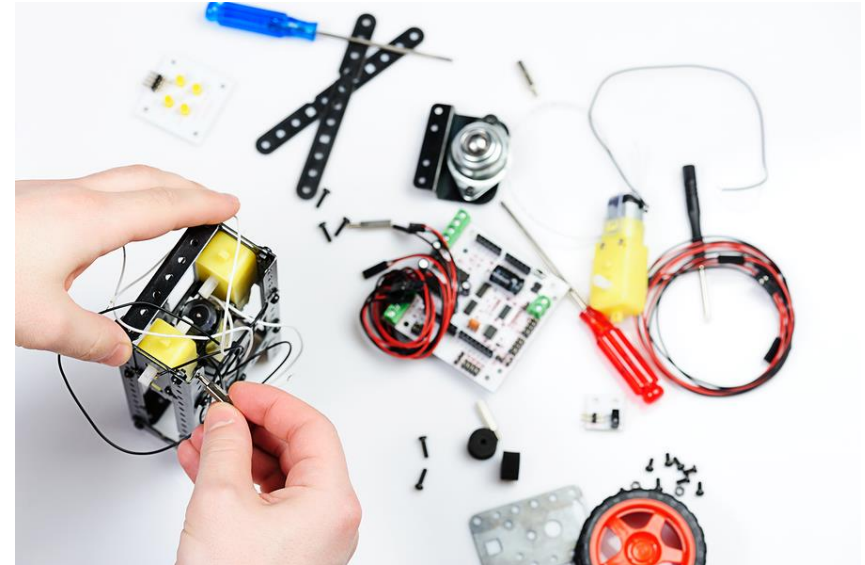
# Other real-world problems



VLSI layout problem



Robot navigation



Automatic assembly sequencing of complex objects by a robot



# Quiz 01: The Towers of Hanoi

---

- Formulate the Towers of Hanoi problem with three pegs and three disks.

# Searching for solutions

- *Infrastructure for search algorithms*
- *Measuring problem-solving performance*



# Search tree

---

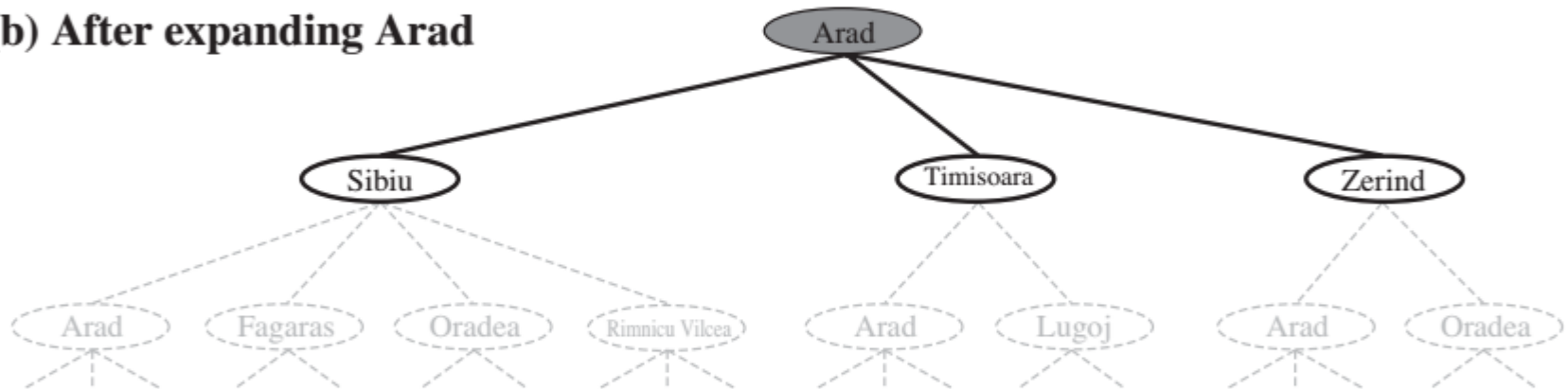
- Search algorithms consider many possible action sequences to find the solution sequence.
- **Search tree:** the possible action sequences starting at the initial state (**root**)
  - Branches are actions and nodes are states in the state space
- **Frontier:** the set of all leaf nodes **available for expansion** at any given point

*Search algorithms all share the basic structure while vary according to how they choose which state to expand next  
-- called **search strategy**.*

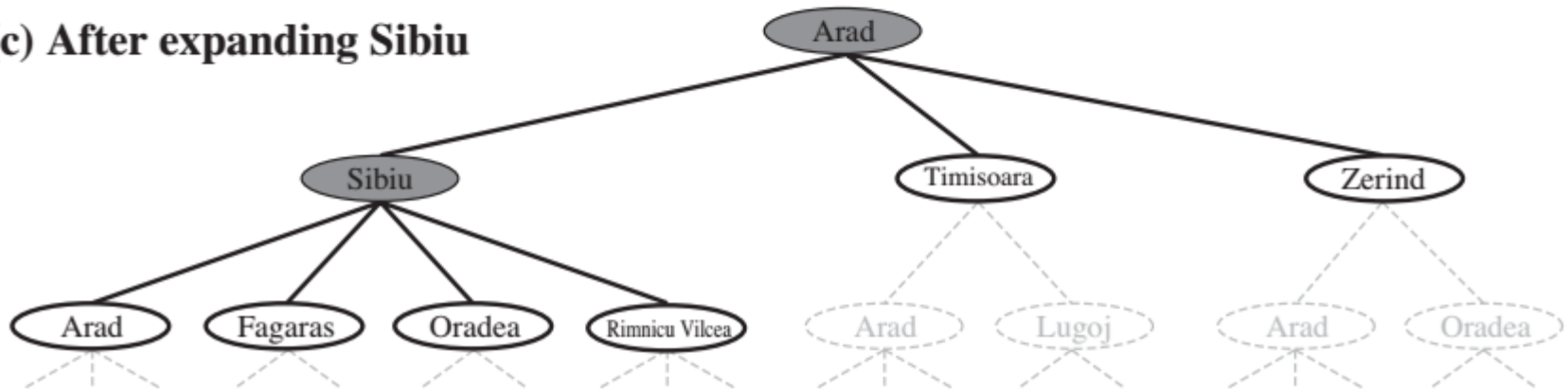
**(a) The initial state**



**(b) After expanding Arad**



**(c) After expanding Sibiu**



# TREE-SEARCH algorithms

**function** TREE-SEARCH(*problem*) **returns** a solution, or failure

    initialize the frontier using the initial state of *problem*

**loop do**

**if** the frontier is empty **then return** failure

        choose a leaf node and remove it from the frontier

**if** the node contains a goal state **then return** the corresponding solution

        expand the chosen node, adding the resulting nodes to the frontier

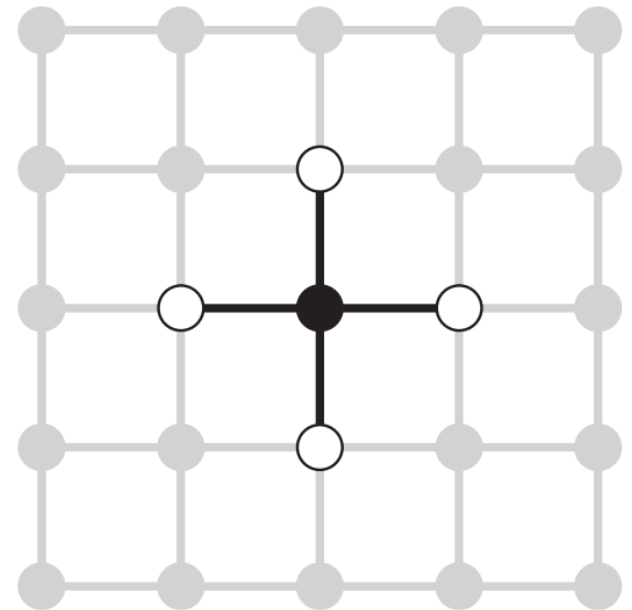
# Redundant paths

- Redundant paths exist whenever there is more than one way to get from one state to another.

Each state has four successors.

A search tree of depth  $d$ :  $4^d$  leaves with repeated states, about  $2d^2$  distinct states within  $d$  steps of any given state

For  $d = 20$ : a trillion nodes but only about 800 distinct states.



# GRAPH-SEARCH algorithms

**function** GRAPH-SEARCH(problem) **returns** a solution, or failure

initialize the frontier using the initial state of problem

*initialize the explored set to be empty*

**loop do**

if the frontier is empty **then return** failure

choose a leaf node and remove it from the frontier

if the node contains a goal state **then return** the corresponding solution

*add the node to the explored set*

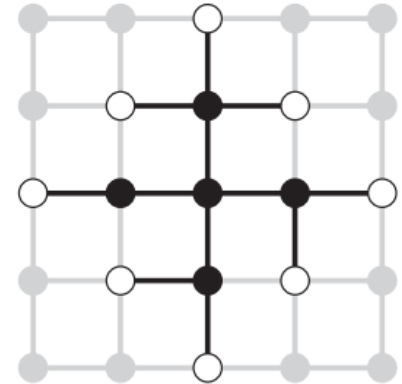
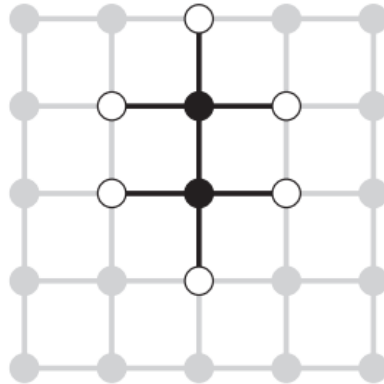
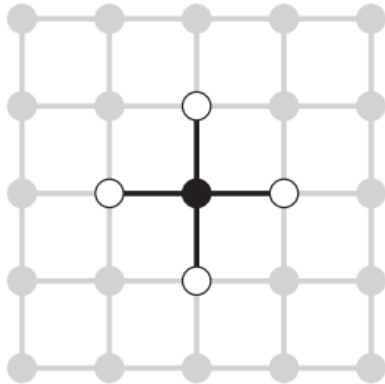
expand the chosen node, adding the resulting nodes to the frontier

*only if not in the frontier or explored set*

- The **explored set** remembers every expanded node.
- Generated nodes that match previously generated nodes, i.e., those in the explored set or the frontier — can be discarded

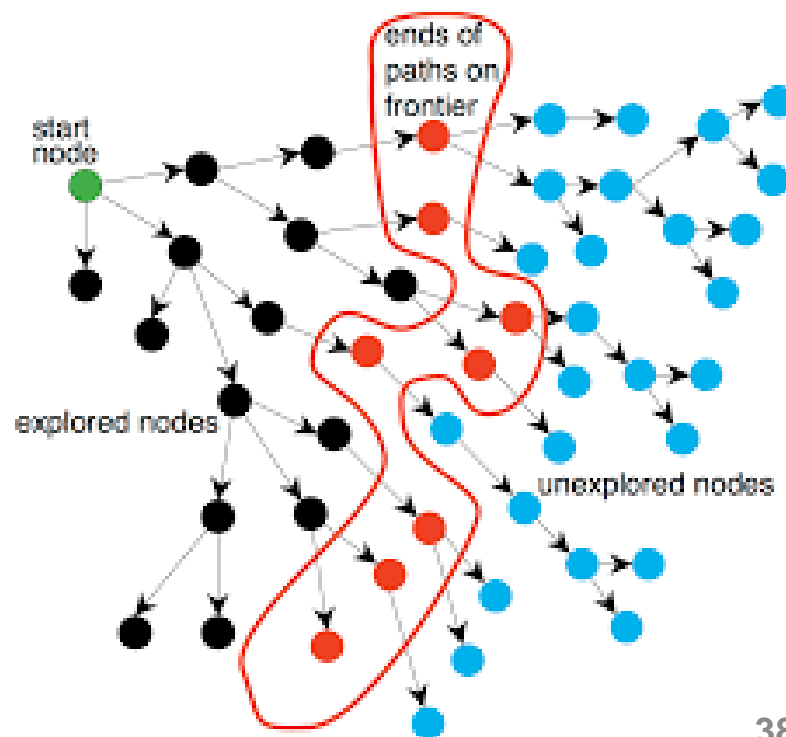


# GRAPH-SEARCH examples



# GRAPH-SEARCH separation property

- The frontier separates the state-space graph into two components: **explored region** vs. **unexplored region**.
  - Every path from the initial state to an unexplored state must pass through a state in the frontier.
- The algorithm systematically examines the states in the state space, one by one, until it finds a solution.

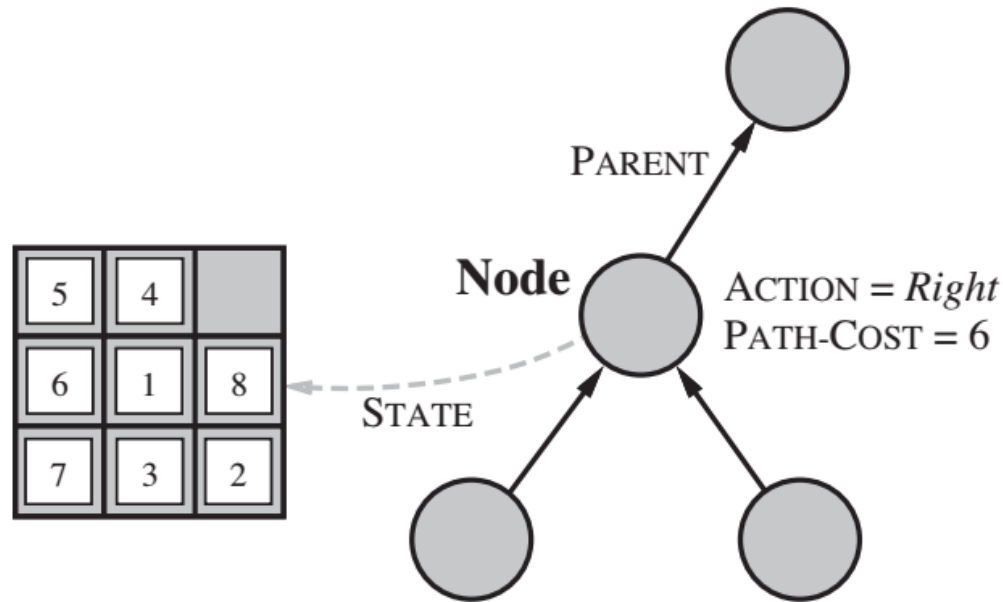


# Infrastructure for search algorithms

---

- Each node  $n$  is structuralized by four components.
  - **$n$ .STATE:** the state in the state space to which the node corresponds
  - **$n$ .PARENT:** the node in the search tree that generated the node  $n$
  - **$n$ .ACTION:** the action applied to the parent to generate  $n$
  - **$n$ .PATH – COST :** the cost, denoted by  $g(n)$ , of the path from the initial state to the node, as indicated by the parent pointer
- Frontier can be implemented with a (priority) queue or stack.
- Explored set can be a hash table that allows for efficient checking of repeated states
  - **Canonical form:** logically equivalent states should map to the same data structure

# Infrastructure for search algorithms



**function** CHILD-NODE(problem, parent, action) **returns** a node  
**return** a node with

STATE = problem.RESULT(parent.STATE, action),  
PARENT = parent, ACTION = action,  
PATH-COST = parent.PATH-COST  
+ problem.STEP-COST(parent.STATE, action)

# Measuring problem-solving performance

---

- **Completeness:** does it always find a solution if one exists?
- **Time complexity:** how long does it take to find a solution?
- **Space complexity:** how much memory is needed to perform the search?
- **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  $\infty$ )

# Quiz 02: The Towers of Hanoi

---

- Draw the first two levels of the search tree for the Towers of Hanoi problem with three pegs and two disks (identical repeated states on a path can be ignored).



**THE END**