CS420 – Artificial Intelligence

SOLVING PROBLEMS BY SEARCHING

Nguyễn Ngọc Thảo
nnthao@fit.hcmus.edu.vn

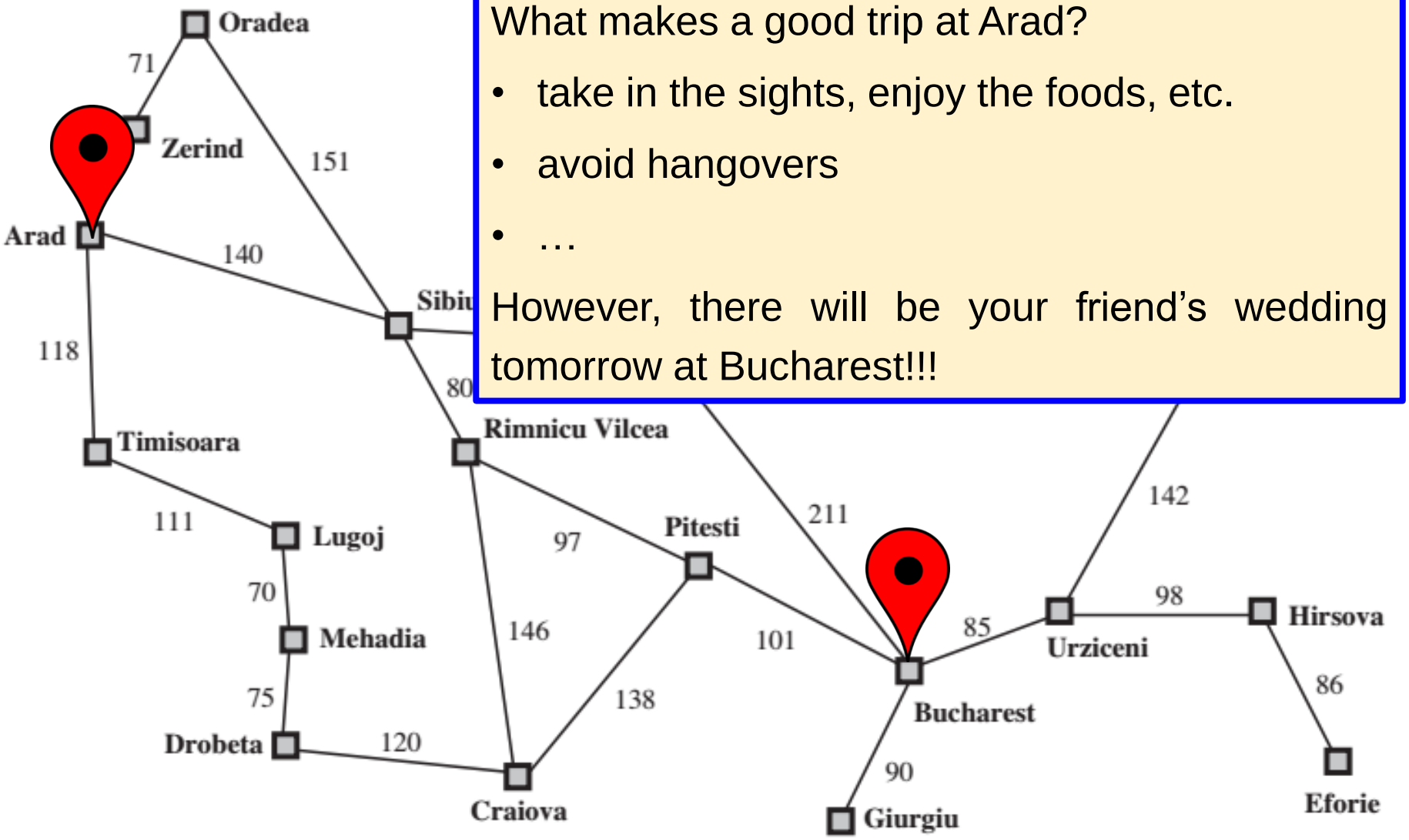
Outline

- Problem-Solving Agents
- Example Problems
- Searching for Solutions

Problem-Solving Agents

- *Well-defined Problems and Solutions*
- *Formulating Problems*

A touring holiday in Romania



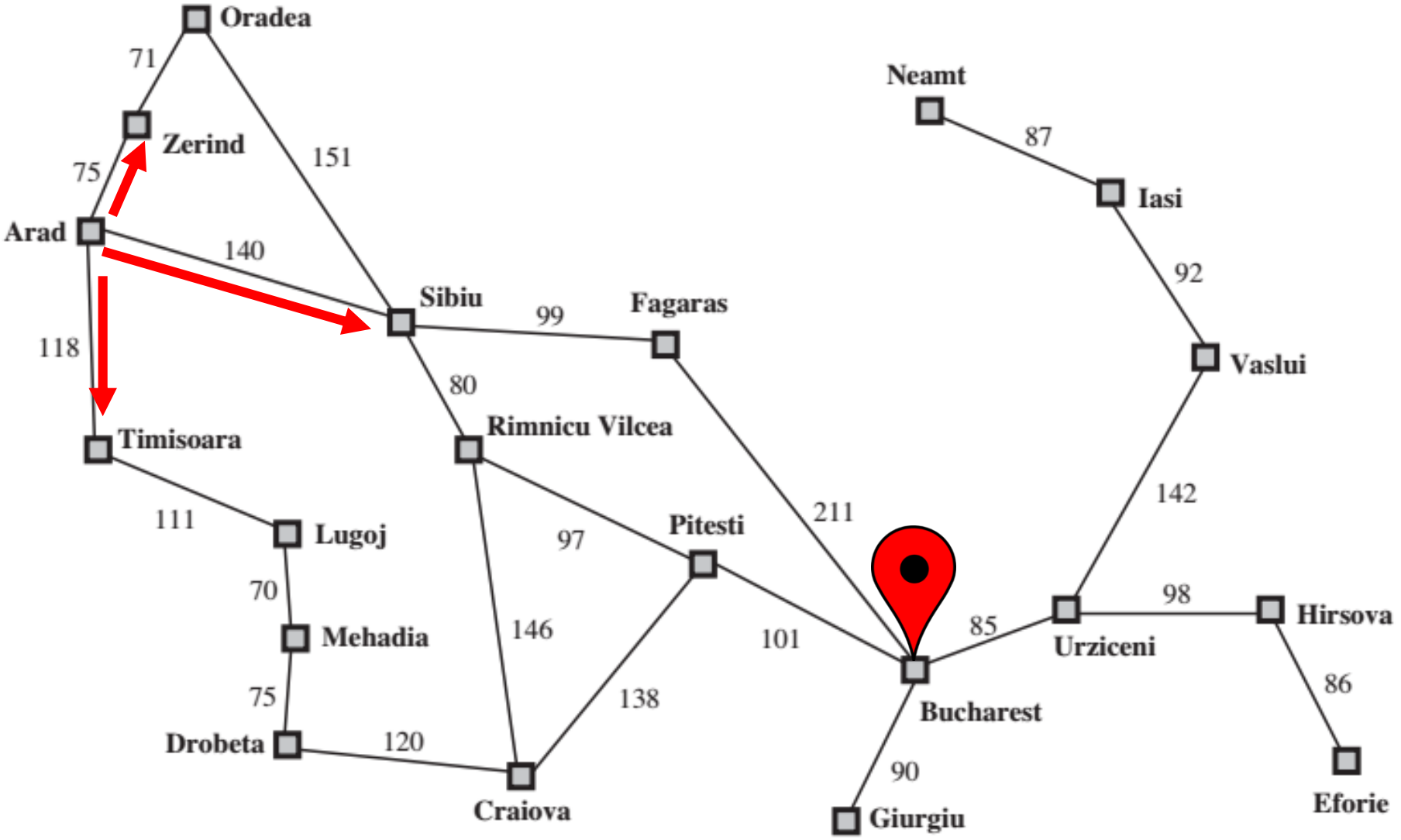
Goal-based agents

- Intelligent agents are supposed to maximize their performance measure.
  - Different factors for performance measure $\rightarrow$ 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.
- The agent finds out how to act, now and in the future, so that it reaches a **goal state**.
- **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 $\rightarrow$ 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.

An agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.

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.
  - While executing the solution, the agent ignores its percepts when choosing an action → **open-loop** system

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., $Result\big(In(Arad), Go(Zerind)\big) = In(Zerind)$
  - **Successor:** a state reachable from a given state by a single action

The state space

- The set of all states reachable from the initial state by any sequence
  - 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 assigns a numeric cost to each path
  - Nonnegative, reflecting the agent's performance measure
  - E.g., $c\big(In(Arad), Go(Zerind), In(Zerind)\big) = 75$
- An **optimal solution** has the lowest path cost.

Formulating problems by abstraction

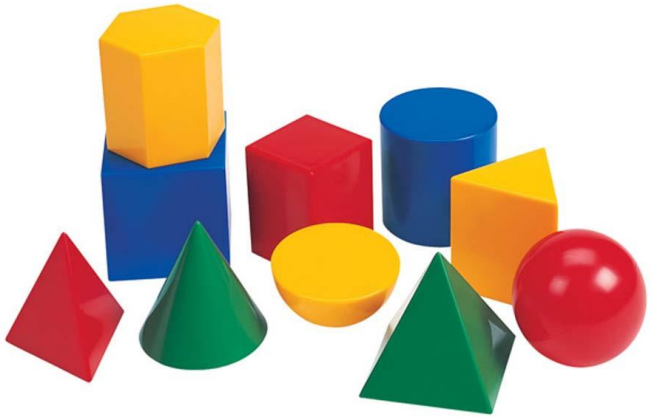
- The process of removing detail from a representation
- **Navigation example:** how do we define states and actions?
  - First step is to abstract “the big picture”, i.e., solve a map problem
    - Nodes = cities, links = roads connecting cities (a high-level description)
  - Later worry about details – traveling companions, road condition, weather, etc.
  - Similarly for driving action – time, fuel consumption, pollution, etc. – are removed

Formulating problems by abstraction

- Abstraction is critical for automated problem solving
  - Real-world is too detailed to model exactly
  - Create an approximate, simplified, model of the world for the computer to deal with
- The choice of a good abstraction thus involves
  - Removing as much detail as possible while
  - Retaining validity and ensuring that the abstract actions are easy to carry out

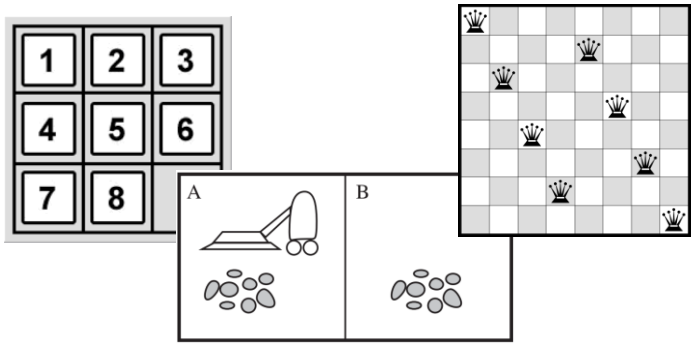
Example Problems

- *Toy Problems*
- *Real-world Problems*



Toy problems vs. Real-world problems

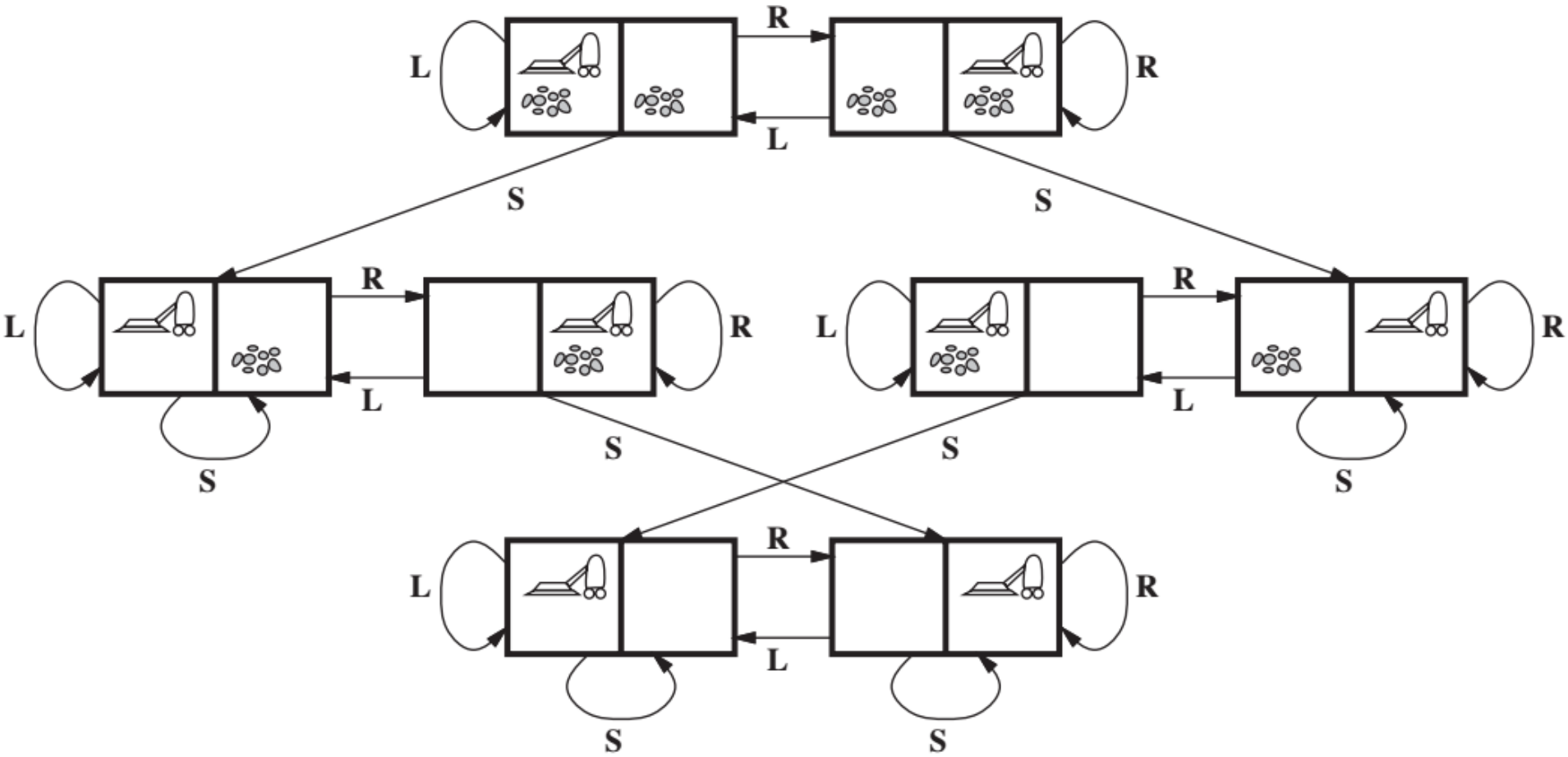
Toy problems	Real-world problems
Illustrate or exercise various problem-solving methods	More difficult
Concise, exact description	No single, agreed-upon description
Can be used to compare performance	
E.g., 8-puzzle, 8-queens problem, Cryptarithmetic, Vacuum world, Missionaries and cannibals, simple route finding	Examples: Route finding, Touring and traveling salesperson problems, VLSI layout, Robot navigation, Assembly sequencing



The Vacuum-cleaner world

- **States:** determined by both 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, and Suck
  - 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

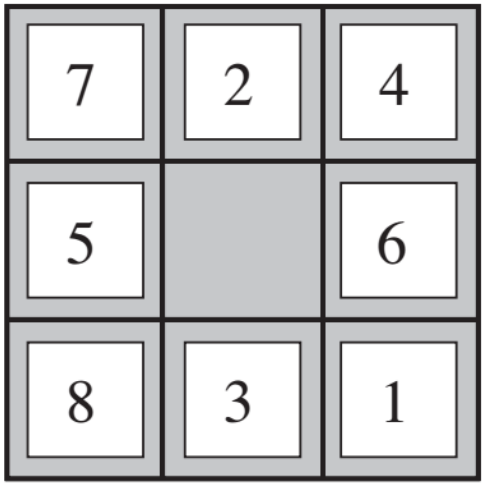


The state space for the vacuum world
Links denote actions: L = Left, R = Right, S = Suck.

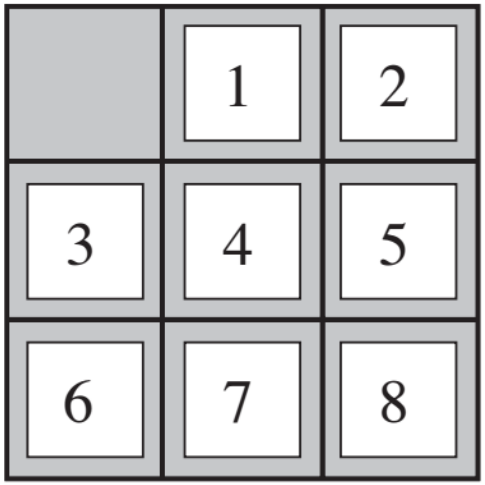
The 8-puzzle

- **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:** whether the state matches the goal configuration
- **Path cost:** each step costs 1

The 8-puzzle



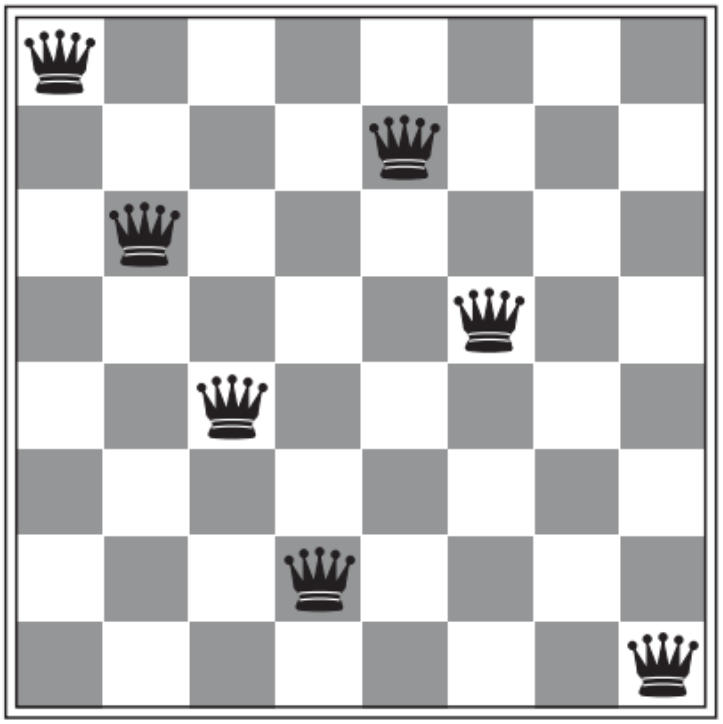
Start State



Goal State

- A member of the family of sliding-block puzzles, NP-complete
- 8-puzzle: 9!/2 = 181,440 reachable states $\rightarrow$ easily solved.
- 15-puzzle: 1.3 trillion (10^{12}) states $\rightarrow$ optimally solved in a few millisecs
- 24-puzzle: around 10^{25} states $\rightarrow$ 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 moves them around
- The path cost is of no interest 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
- 8-queens: reduce the spate space from 1.8×10^{14} to just 2,057
- 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.

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

- **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

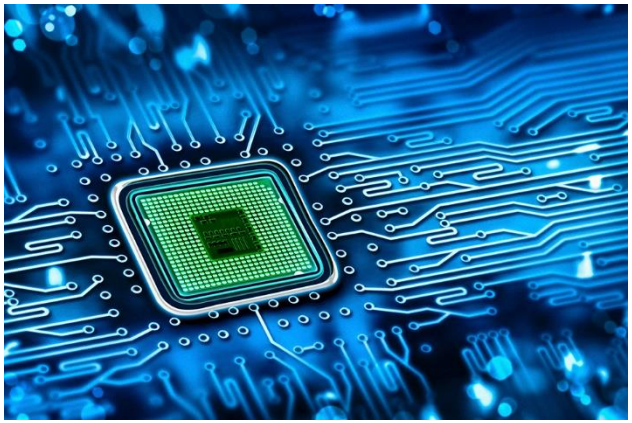
The route-finding problem

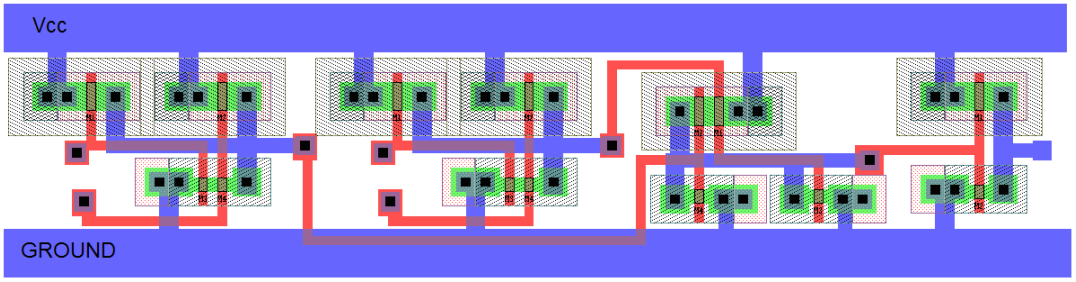
- Consider the airline travel problems solved by a travel-planning Web site.
- **States:** a location (e.g., an airport) and the current time
  - Extra information about "historical" aspects, e.g., previous segments, fare bases, statuses as domestic or international, are needed.
- **Initial state:** 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 already at the final destination specified by the user.
- **Path cost:** depends on different factors of the per performance measure

The touring problems

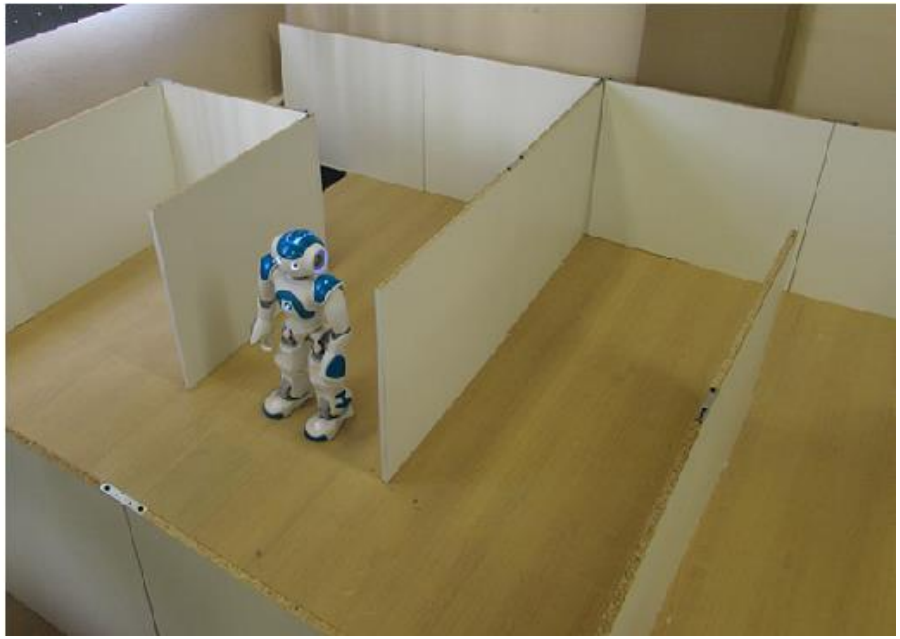
- **Actions:** correspond to trips between adjacent cities
- Each **state** must include not just the current location but also the set of cities the agent has visited.
- For example, the touring holiday in Romania
  - In(Bucharest), Visited({Bucharest}): initial state
  - In(Vaslui), Visited({Bucharest, 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.
  - Planning movements of automatic circuit-board drills and of stocking machines on shop floors, etc.

Other real-world problems

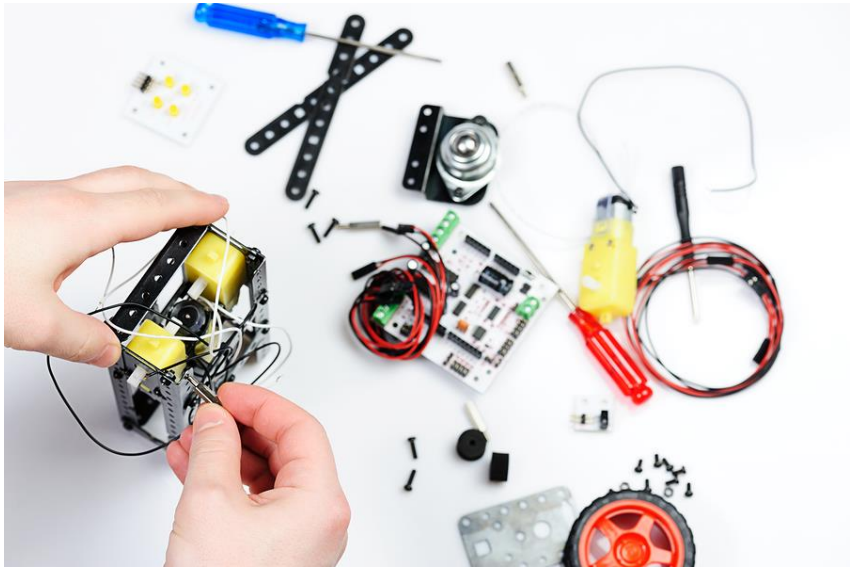




VLSI layout problem



Robot navigation



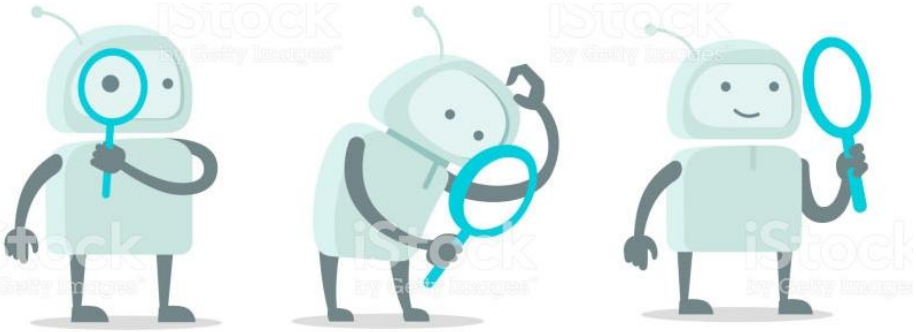
Automatic assembly sequencing of complex objects by a robot

Quiz: The Tower of Hanoi

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

Searching for Solutions

- *Infrastructure for Search Algorithms*
- *Measuring Problem-solving Performance*

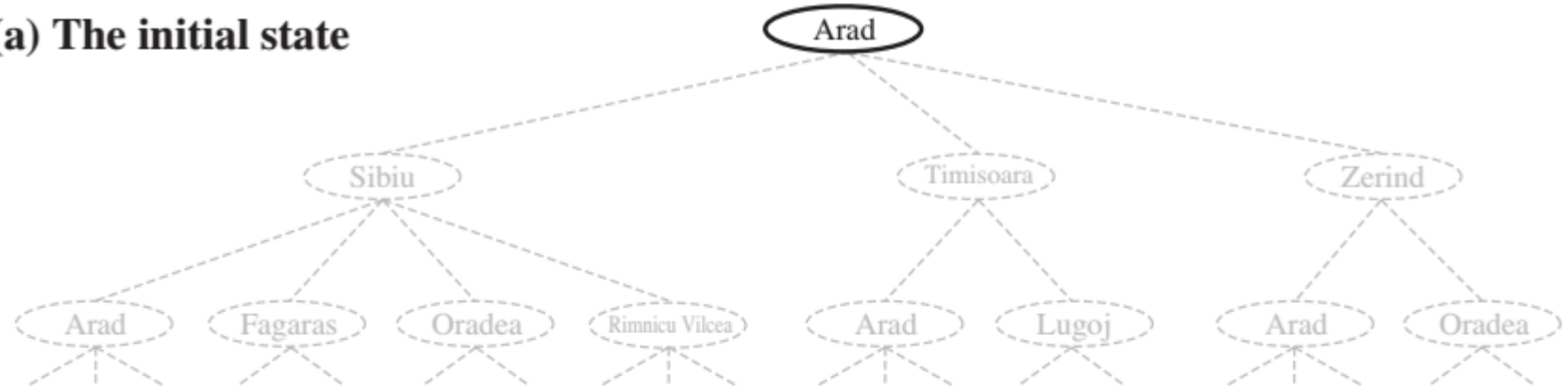


Search tree

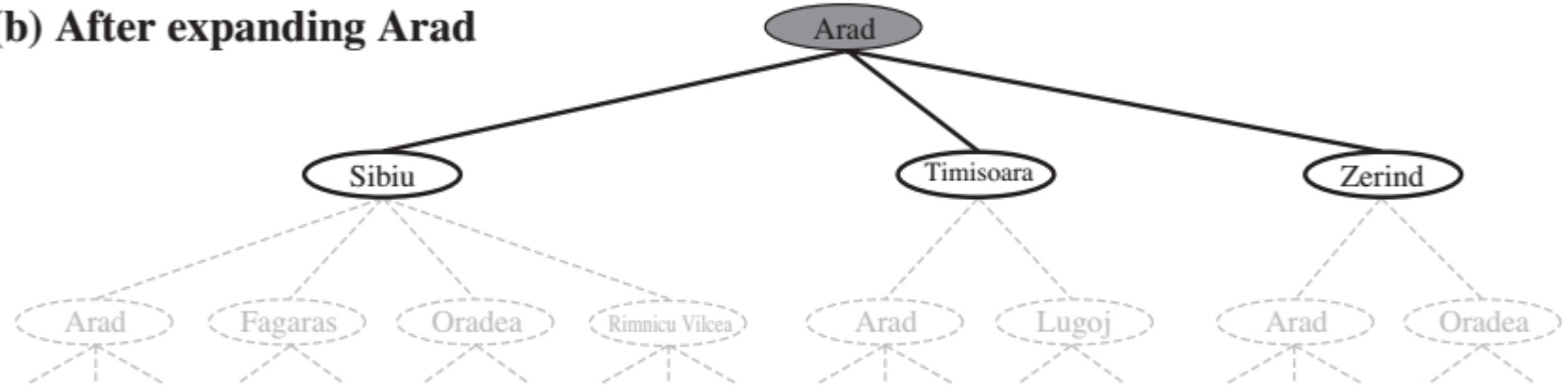
- Search algorithms consider many possible action sequences to find the solution sequence.
- **Search tree:** the possible action sequences starting at the initial state (**root**)
  - The branches are actions and the nodes correspond to states in the state space of the problem
- **Frontier:** the set 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**.*

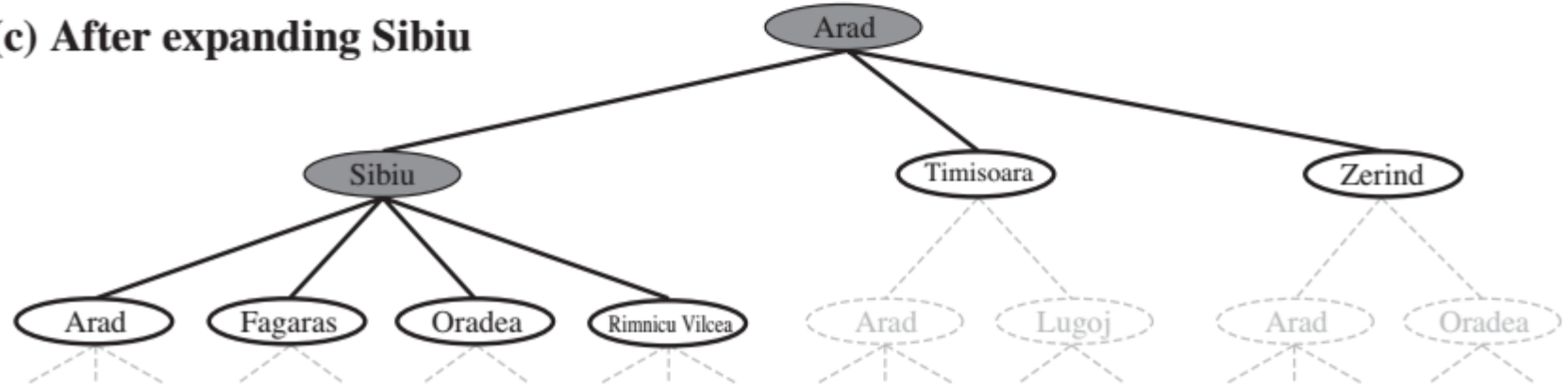
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



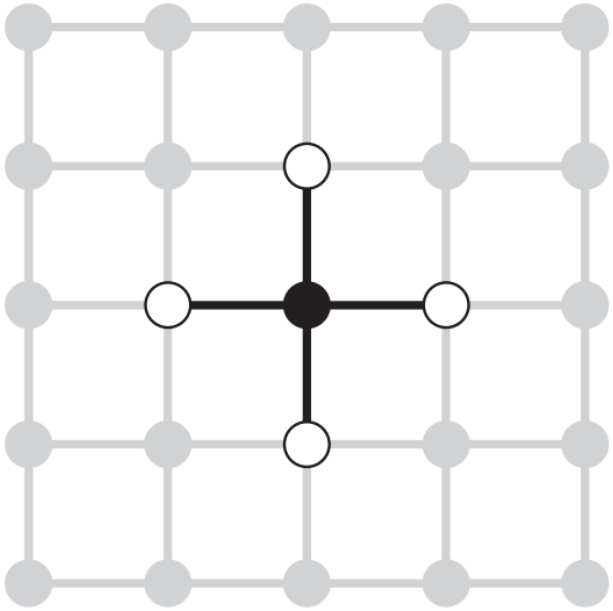
TREE-SEARCH algorithms

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 are unavoidable.
- Following redundant paths can cause a tractable problem to become **intractable**.
- This is true even for algorithms that know how to avoid infinite loops

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.

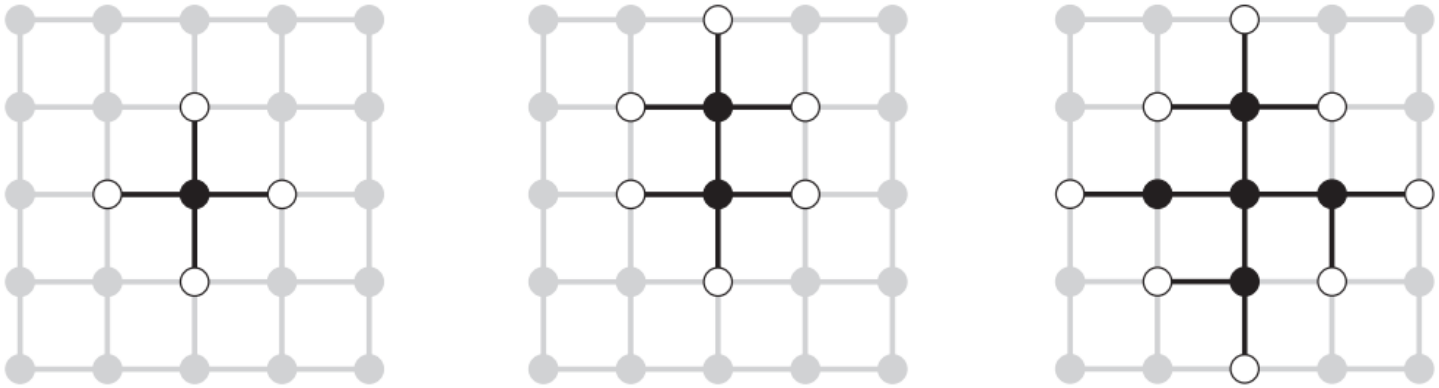


GRAPH-SEARCH algorithms

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

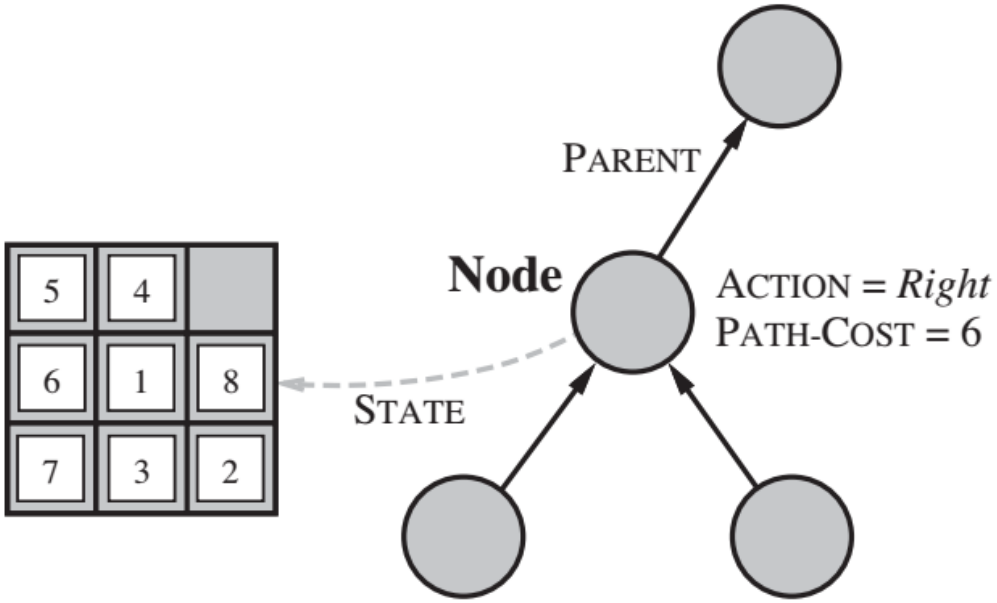
GRAPH-SEARCH examples



Infrastructure for search algorithms

- Each node n is structuralized by four components.
  - $n.\mathbf{STATE:}$ the state in the state space to which the node corresponds
  - $n.\mathbf{PARENT:}$ the node in the search tree that generated the node n
  - $n.\mathbf{ACTION:}$ the action applied to the parent to generate n
  - $n.\mathbf{PATH-COST}$ **:** the cost, denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointer
- The frontier can be implemented with a queue, stack or a priority queue.
- The explored set can be a hash table to allow efficient checking for 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 ∞)

Quiz: The Tower of Hanoi

- Draw the first two levels of the search tree for the Tower of Hanoi problem with three pegs and two disks (identical repeated states on a branch are ignored).

THE END