



# Artificial Intelligence

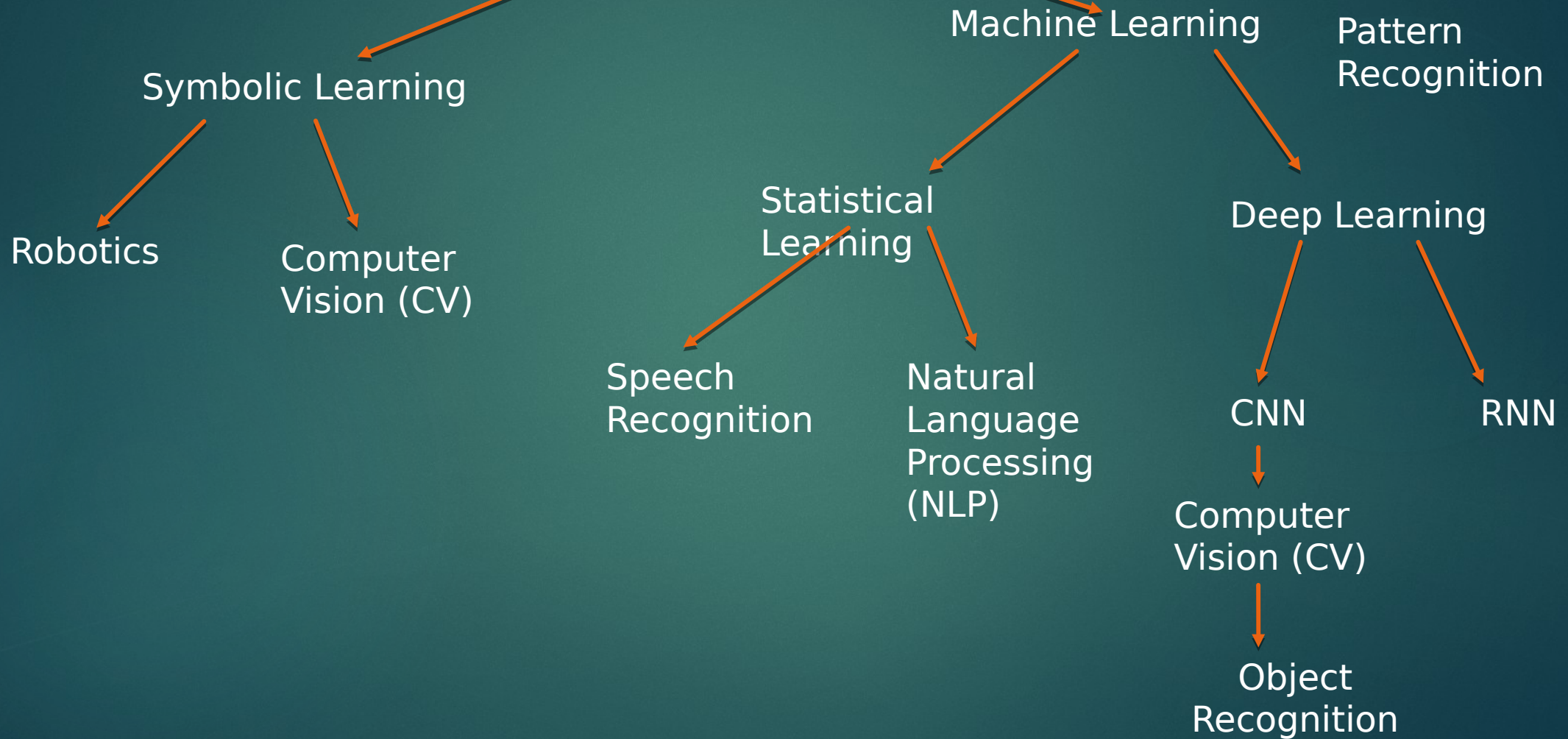
**INTERIM SEMESTER 2021-22**  
**BPL**  
**CSE3007-LT-AB306**  
**FACULTY: SIMI V.R.**

# Computer Science

## Artificial Intelligence (AI)

Symbol  
Based

Data Based

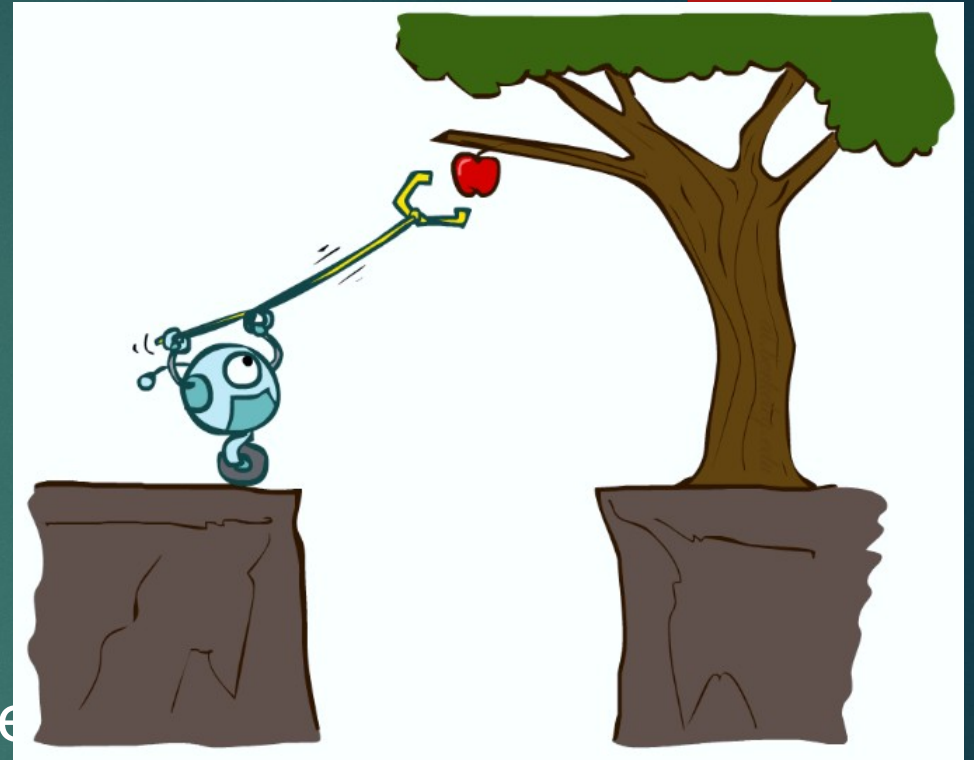
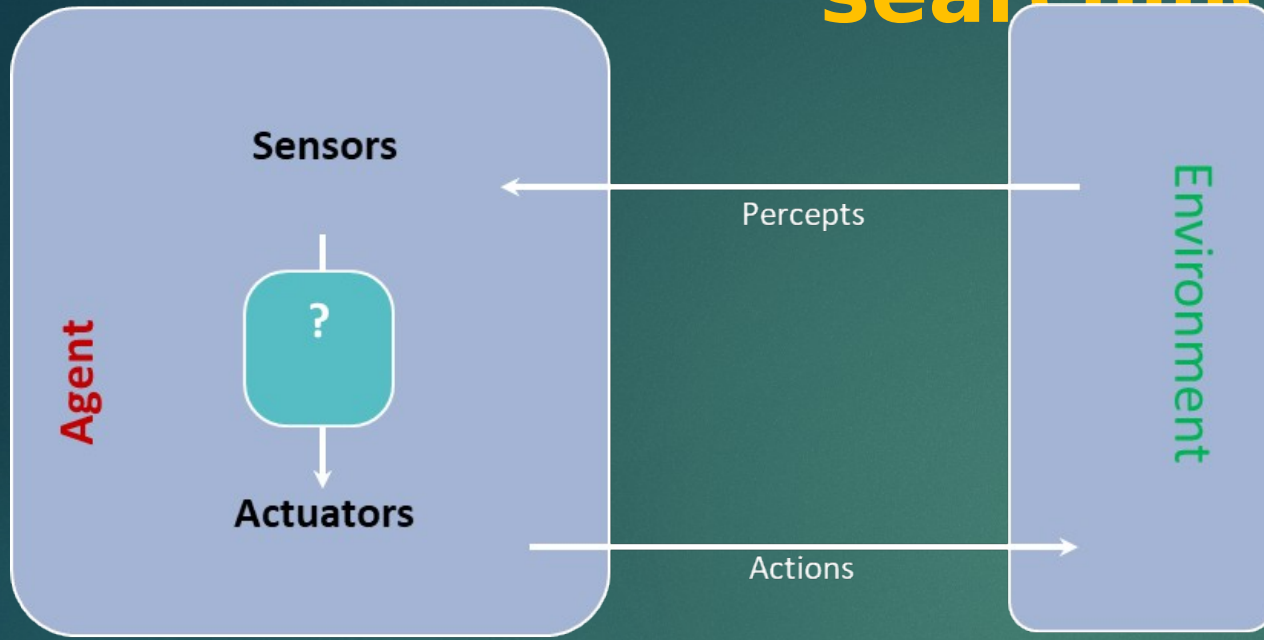


# What can AI do today?

- ▶ **Robotic vehicles**
- ▶ **Speech recognition**
- ▶ **Autonomous planning and scheduling**
- ▶ **Game playing**
- ▶ **Spam fighting**
- ▶ **Logistics planning**
- ▶ **Robotics**
- ▶ **Machine Translation**



# PROBLEM SOLVING-Solving problems by searching



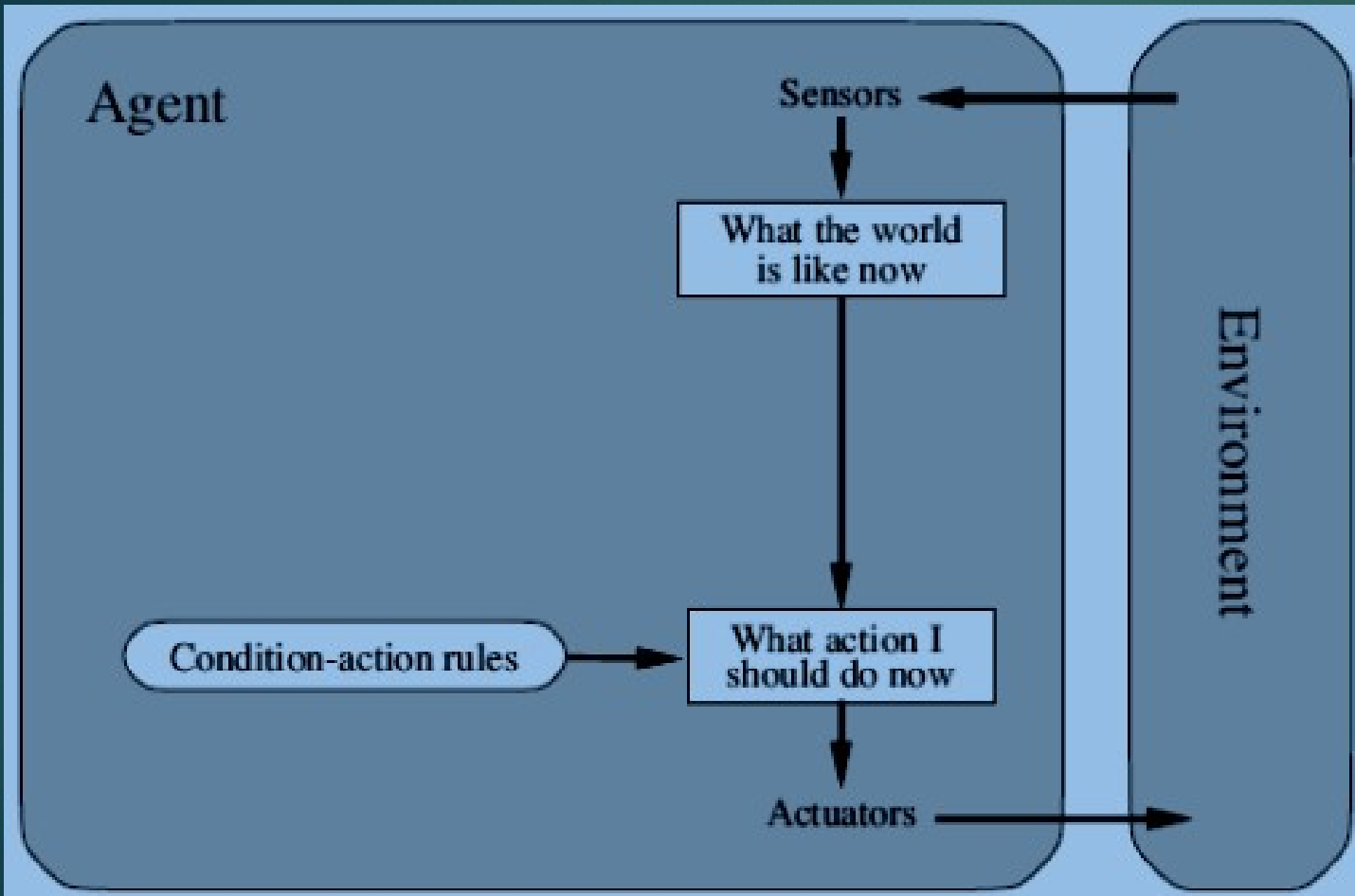
**Problem-solving agent** - Goal-based agent

**Goal formulation** - is the first step in problem solving.

**Problem formulation** - is the process of deciding what actions and states to consider

- ▶ *an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually*

# Problem-solving agent



```
function SIMPLE-REFLEX-AGENT(percept) returns an action  
persistent: rules, a set of condition–action rules  
  
state ← INTERPRET-INPUT(percept)  
rule ← RULE-MATCH(state, rules)  
action ← rule.ACTION  
return action
```

- A simple reflex agent.
- It acts according to a rule whose condition matches the current state, as defined by the percept.

Schematic diagram of a simple reflex agent.

# Examples of agent types and their PEAS descriptions

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry



# Problem solving agents

- ▶ **The agent function** for an agent specifies the action taken by the agent in response to any percept sequence.
- ▶ **The performance measure** evaluates the behaviour of the agent in an environment.
- ▶ **An agent** is something that perceives and acts in an environment.
- ▶ **A rational agent** is one that does the right thing-conceptually speaking, every entry in the table for the agent function is filled out correctly. It acts so as to maximize the expected value of the performance measure, given the percept sequence it has seen so far.
- ▶ **A task environment** specification includes the performance measure, the external environment, the actuators, and the sensors.

In designing an agent, the first step must always be to specify the task environment as fully as possible. They can be fully or partially observable, single-agent or multiagent, deterministic or stochastic, episodic or sequential, static or dynamic, discrete or continuous, and known or unknown.

- ▶ **The agent program** implements the agent function. There exists a variety of basic agent-program designs reflecting the kind of information made explicit and used in the decision process. The designs vary in efficiency, compactness, and flexibility. The appropriate design of the agent program depends on the nature of the environment.
- ▶ **Simple reflex agents** respond directly to percepts, whereas model-based reflex agents maintain internal state to track aspects of the world that are not evident in the current percept.
- ▶ **Goal-based agents** act to achieve their goals
- ▶ **Utility-based agents** try to maximize their own expected “happiness.”
- ▶ All agents can improve their performance through **learning**.

**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*  $\leftarrow$  UPDATE-STATE(*state*, *percept*)

**if** *seq* is empty **then**

*goal*  $\leftarrow$  FORMULATE-GOAL(*state*)

*problem*  $\leftarrow$  FORMULATE-PROBLEM(*state*, *goal*)

*seq*  $\leftarrow$  SEARCH(*problem*)

**if** *seq* = *failure* **then return** a null action

*action*  $\leftarrow$  FIRST(*seq*)

*seq*  $\leftarrow$  REST(*seq*)

**return** *action*

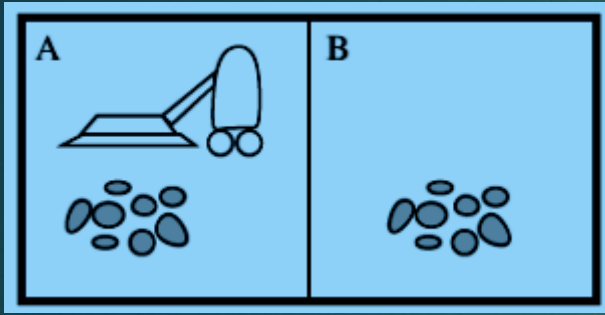
A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.



# Toy problems - Vacuum world

- **States:** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are  $2 \times 2^2 = 8$  possible world states. A larger environment with  $n$  locations has  $n \cdot 2^n$  states.
- **Initial state:** Any state can be designated as the initial state.
- **Actions:** In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
- **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:** This checks whether all the squares are clean.
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

# Intelligent Agents



**A vacuum-cleaner world with just two locations.**

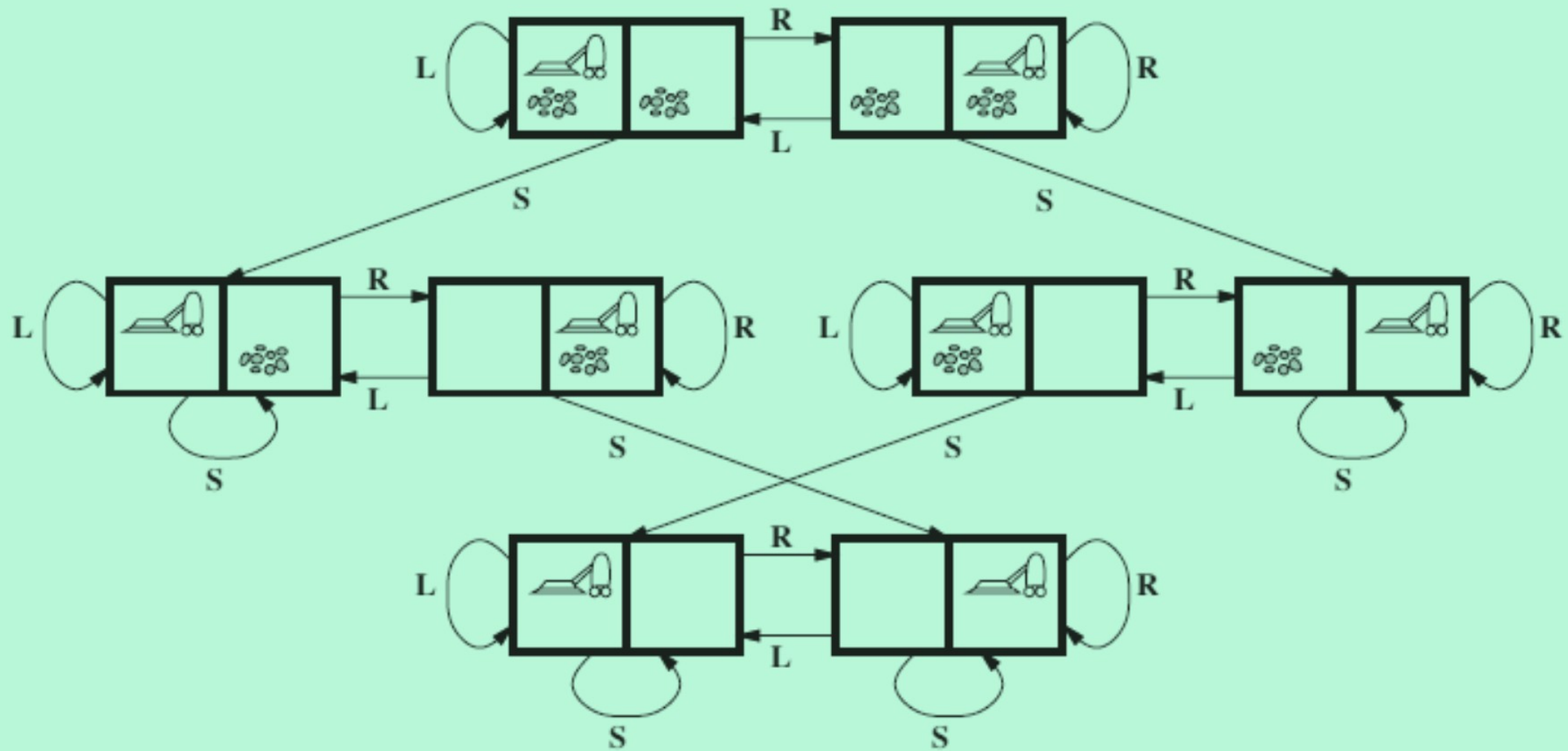
Percept sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
⋮	⋮
[A, Clean], [A, Clean], [A, Clean]	Right
[A, Clean], [A, Clean], [A, Dirty]	Suck
⋮	⋮

## Partial tabulation of a simple agent function for the vacuum-cleaner world

**Example—the vacuum-cleaner world:** This world is so simple that we can describe everything that happens. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing.

**One very simple agent function is the following:** if the current square is dirty, then suck; otherwise, move to the other square.





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

# 8-puzzle (sliding-block puzzles)

7	2	4
5		6
8	3	1

Start State




	1	2
3	4	5
6	7	8

Goal State

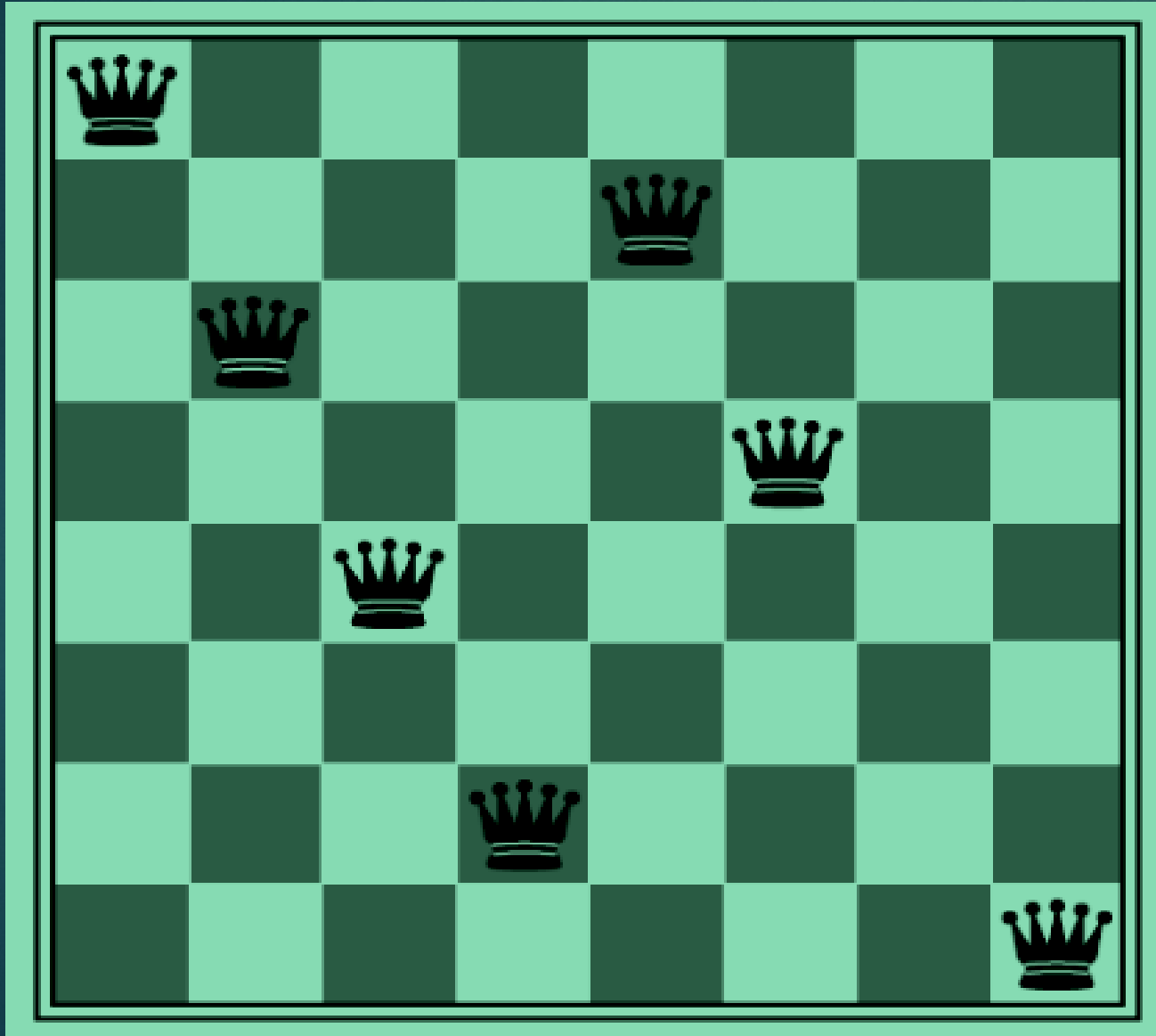
- ▶ Which are often used as test problems for new search algorithms in AI.
- ▶ An instance is shown in Figure.
- ▶ Consists of a 3×3 board with eight numbered tiles and a blank space.
- ▶ A tile adjacent to the blank space can slide into the space.
- ▶ The object is to reach a specified goal state, such as the one shown on the right of the figure.
- ▶ Sliding-block puzzles - NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described here.



# 8-puzzle: The standard formulation

- **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state:** Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states 
- **Actions:** The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure  the resulting state has the 5 and the blank switched.
- **Goal test:** This checks whether the state matches the goal configuration shown in Figure 
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

# 8-queens problem



Goal: is to place eight queens on a chessboard such that no queen attacks any other.

(A queen attacks any piece in the same row, column or diagonal.)

An attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.



# 8-queens problem

## 1. Incremental formulation

- ▶ Involves operators that *augment* the state description, starting with an empty state.
- ▶ Each action adds a queen to the state.

## 2. A complete-state formulation

- ▶ Starts with all 8 queens on the board and moves them around.
- ▶ In either case, the path cost is of no interest because only the final state counts.

## The incremental formulation

**States:** Any arrangement of 0 to 8 queens on the board is a state.

**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.

# Real-world problems

## Route-finding problem

- ▶ Web sites
- ▶ In-car systems that provide driving directions
- ▶ Routing video streams in computer networks
- ▶ Military operations planning
- ▶ Airline travel-planning systems

## Touring problems

### Traveling salesperson problem (TSP)

### VLSI layout problem

### Robot navigation

### Automatic assembly sequencing



# Airline travel problems solved by a travel-planning web site

**States:** Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these “historical” aspects.

**Initial state:** This is 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.

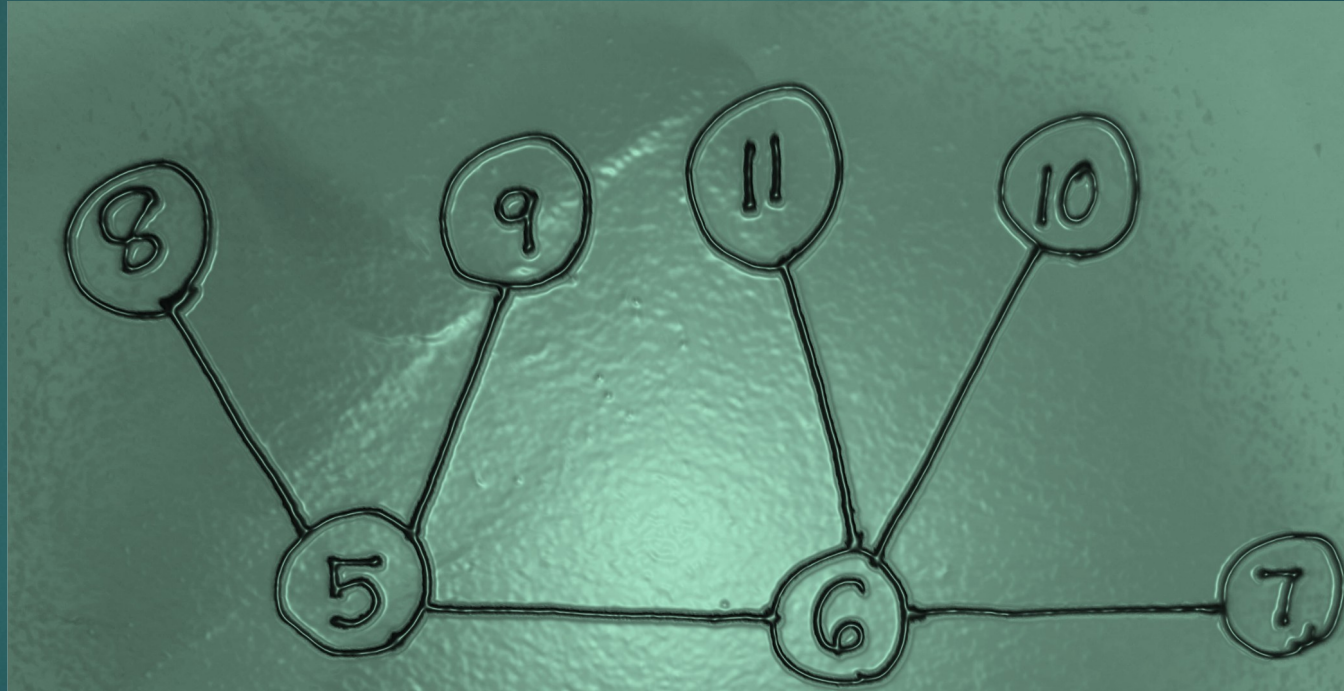
**Transition model:** The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.

**Goal test:** Are we at the final destination specified by the user?

**Path cost:** This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

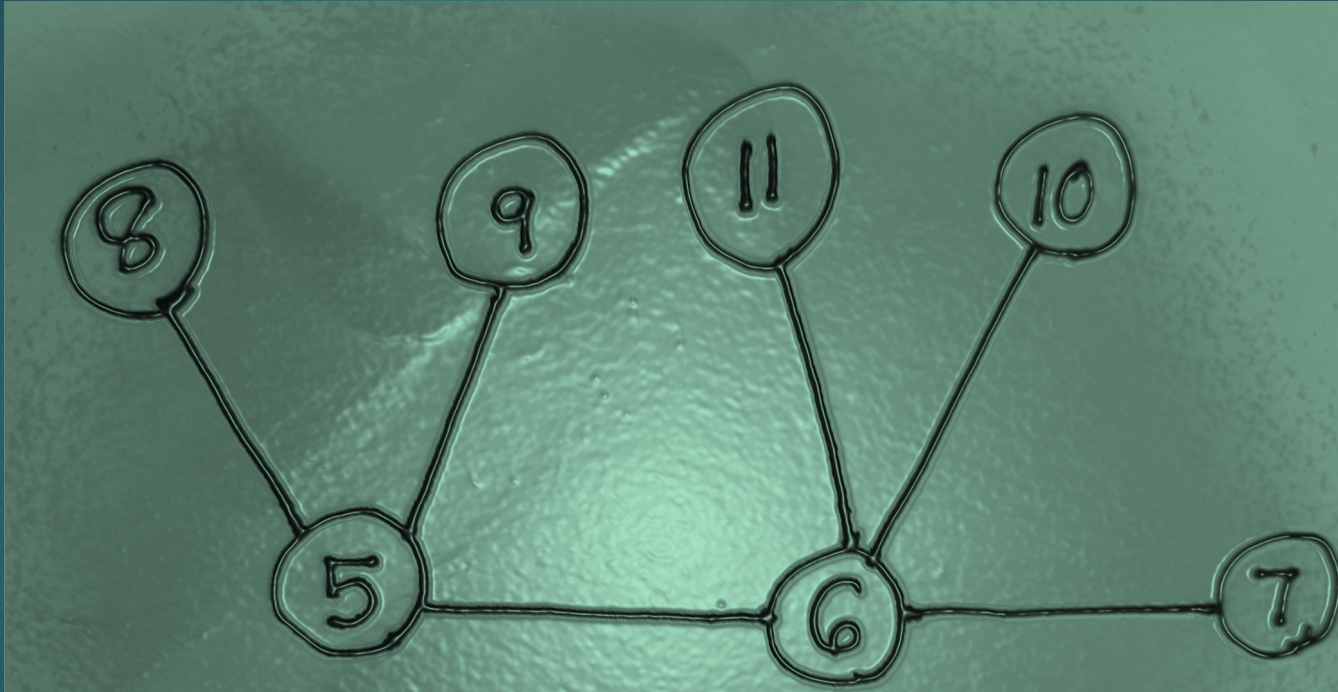
# BREADTH-FIRST-SEARCH (BFS)

Visiting Vertex  
Exploration of Vertex



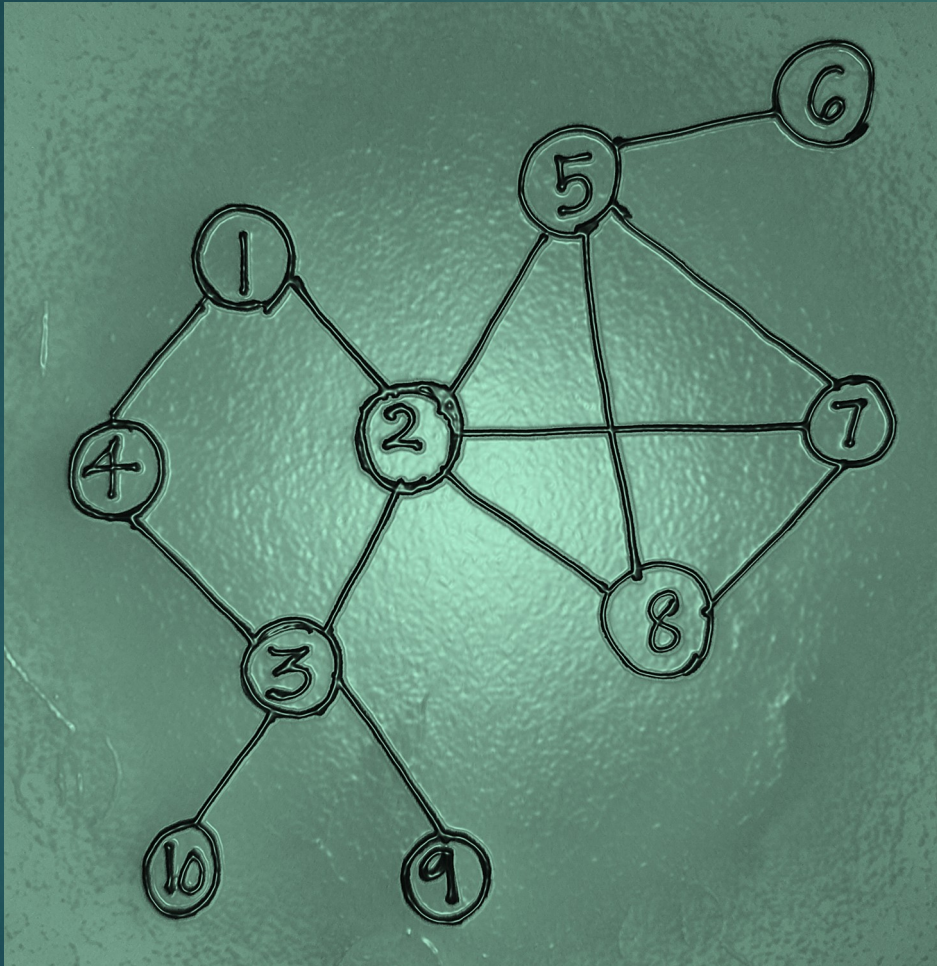


# DEPTH-FIRST-SEARCH (DFS)

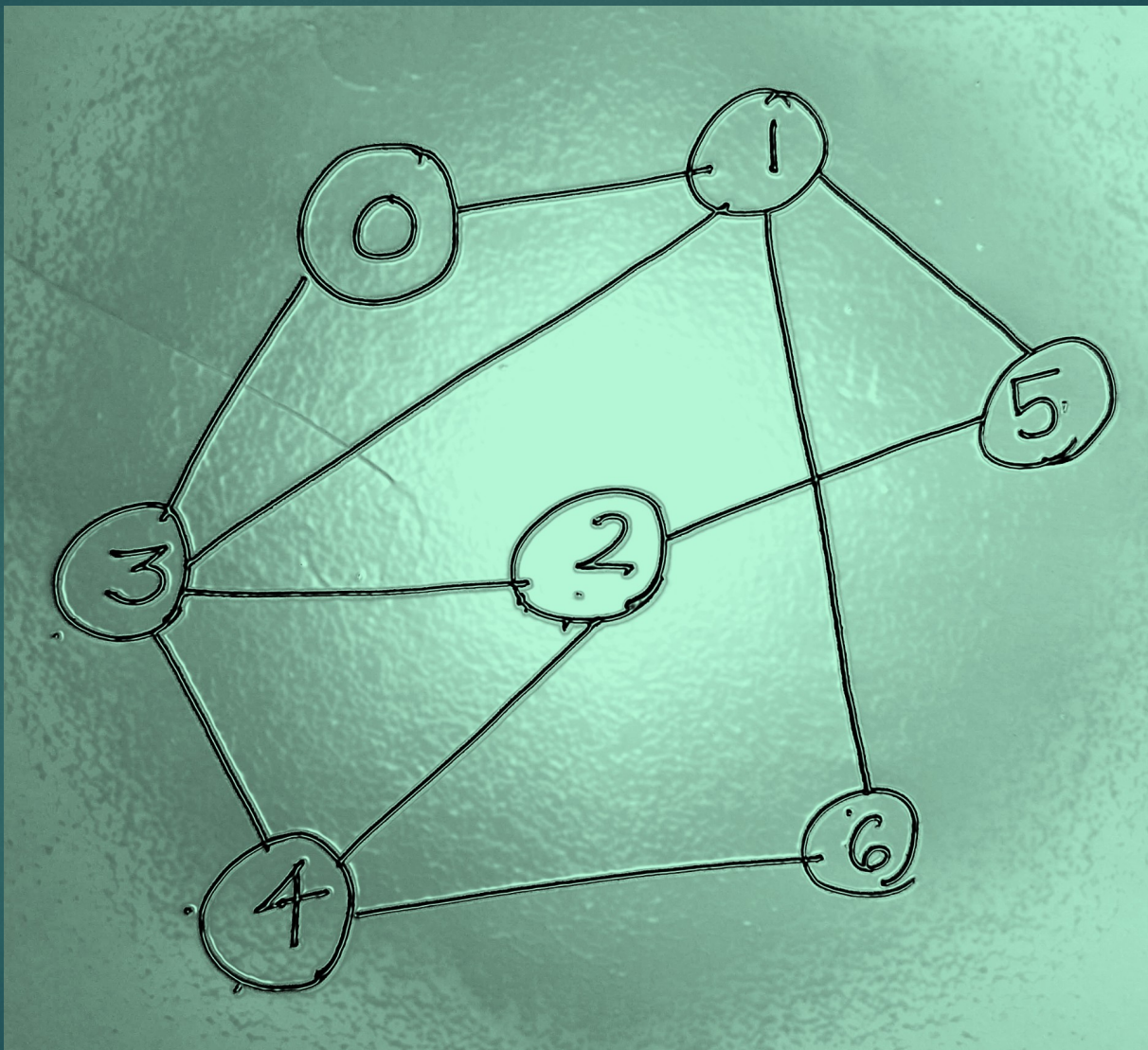




# Perform BFS and DFS for the graph given below

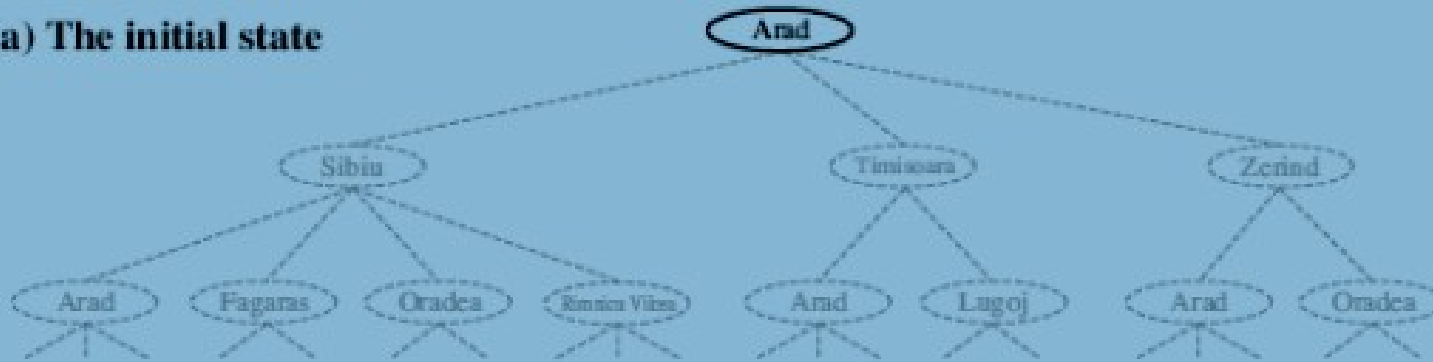




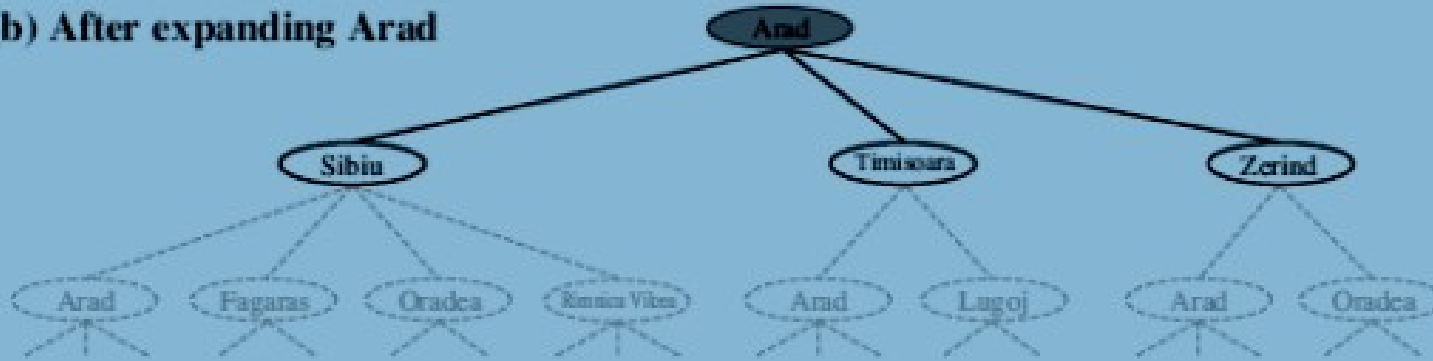


# Search Tree for finding Route

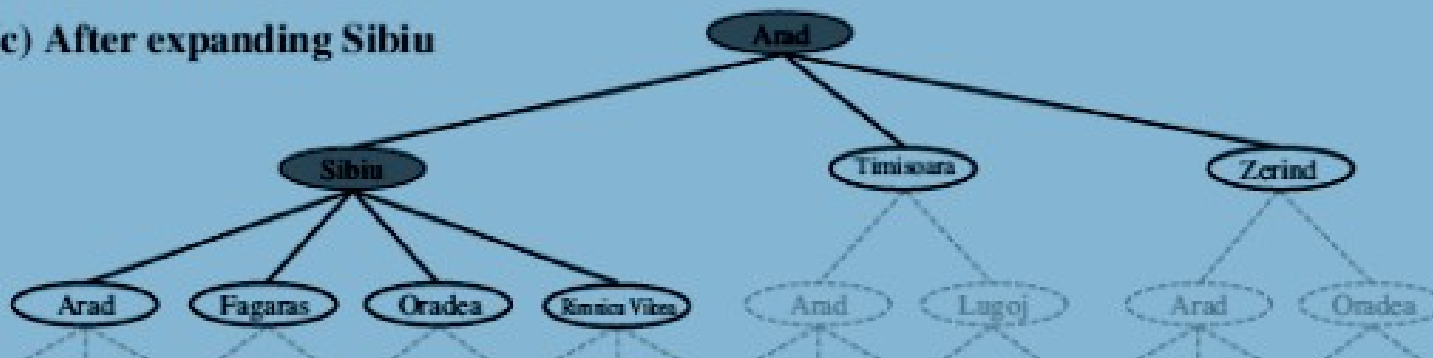
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



- Partial search trees for finding a route from Arad to Bucharest.
- Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.



# An informal description of the general tree-search and graph-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
```

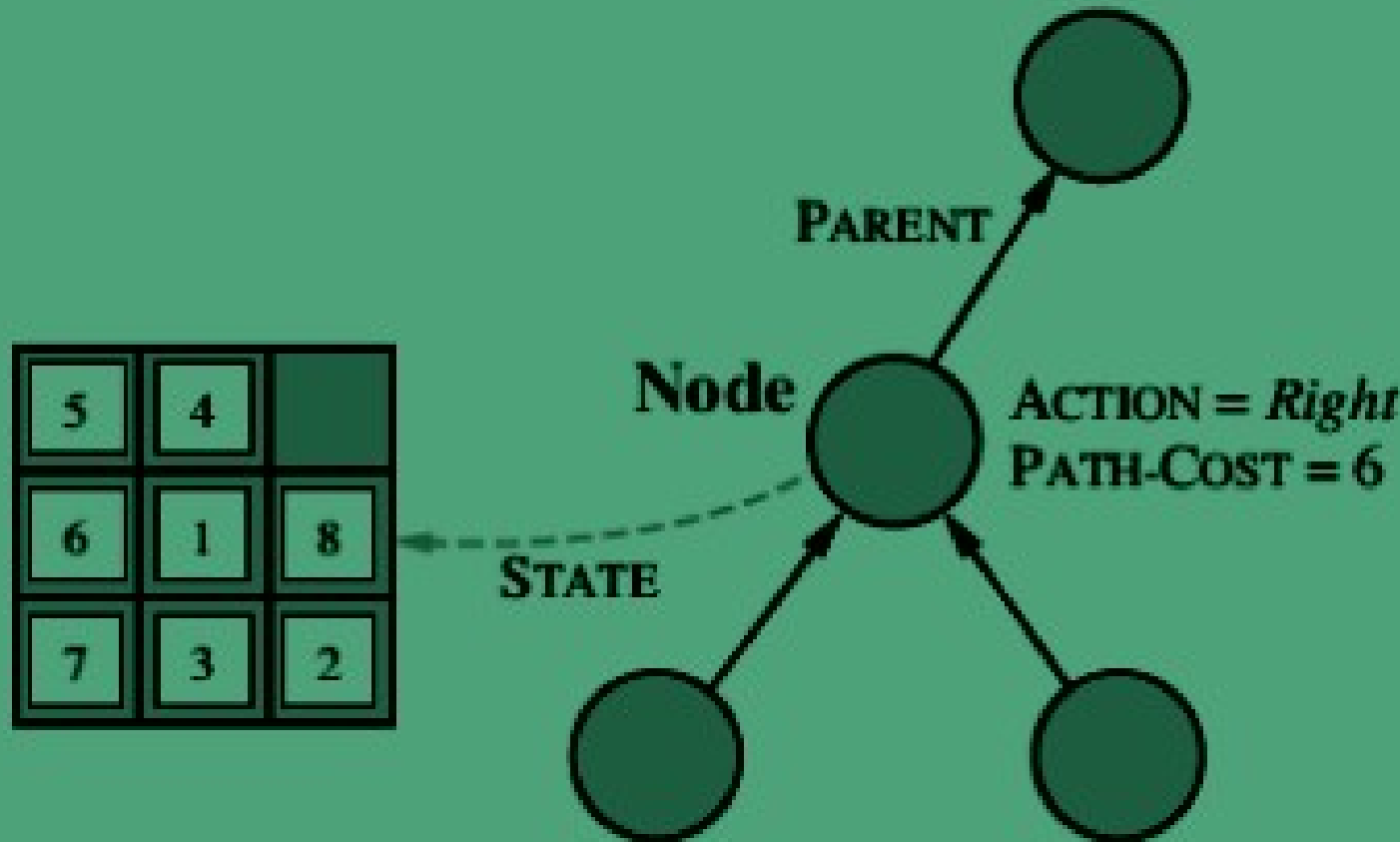
---

```
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
```

# Infrastructure for search algorithms

- Search algorithms require a data structure to keep track of the search tree that is being constructed.
- For each node  $n$  of the tree, we have a structure that contains 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 this node
  - ❖  **$n.ACTION$ :** the action that was applied to the parent to generate the node
  - ❖  **$n.PATH-COST$ :** the cost, traditionally denoted by  $g(n)$ , of the path from the initial state to the node, as indicated by the parent pointers

# Infrastructure for search algorithms (Cont'd)



- Nodes are the data structures from which the search tree is constructed.
- Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.



**The function CHILD-NODE takes a parent node and an action and returns the resulting child node.**

```
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)
```

## QUEUE

- The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.
- The appropriate data structure for this is a queue.

The operations on a queue are as follows:

- **EMPTY?(queue)**: returns true only if there are no more elements in the queue.
- **POP(queue)** : removes the first element of the queue and returns it.
- **INSERT(element, queue)**: inserts an element and returns the resulting queue.

# Measuring problem-solving performance



**We can evaluate an algorithm's performance in four ways:**

**COMPLETENESS:** Is the algorithm guaranteed to find a solution when there is one?

**OPTIMALITY:** Does the strategy find the optimal solution?

**TIME COMPLEXITY:** How long does it take to find a solution?

**SPACE COMPLEXITY:** How much memory is needed to perform the search?



# Search Strategies

1. Uninformed search (Blind search)
2. Informed search (Heuristic search )

## 1. Uninformed search (Blind search)

- ❖ **The strategies have no additional information about states beyond that provided in the problem definition.**
- ❖ **All they can do is generate successors and distinguish a goal state from a non-goal state.**
- ❖ **All search strategies are distinguished by the *order* in which nodes are expanded.**

### **E.g. 1: Breadth-first search**

**Breadth-first search** is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.

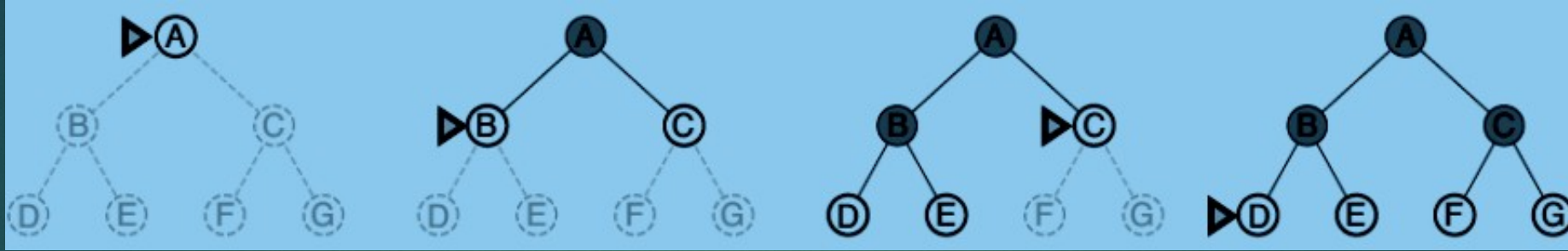
All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.



# Breadth-first search on a graph

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure  
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0  
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)  
  frontier  $\leftarrow$  a FIFO queue with node as the only element  
  explored  $\leftarrow$  an empty set  
  loop do  
    if EMPTY?(frontier) then return failure  
    node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */  
    add node.STATE to explored  
    for each action in problem.ACTIONS(node.STATE) do  
      child  $\leftarrow$  CHILD-NODE(problem, node, action)  
      if child.STATE is not in explored or frontier then  
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)  
        frontier  $\leftarrow$  INSERT(child, frontier)
```

# Breadth-first search on a simple binary tree



- At each stage, the node to be expanded next is indicated by a marker.

## Uniform-cost search

- When all step costs are equal, breadth-first search is optimal because it always expands the *shallowest* unexpanded node.
- By a simple extension, we can find an algorithm that is optimal with any step-cost function.
- Instead of expanding the shallowest node, **uniform-cost search** expands the node  $n$  with the *lowest path cost*  $g(n)$ .
- This is done by storing the frontier as a priority queue ordered by  $g$ .



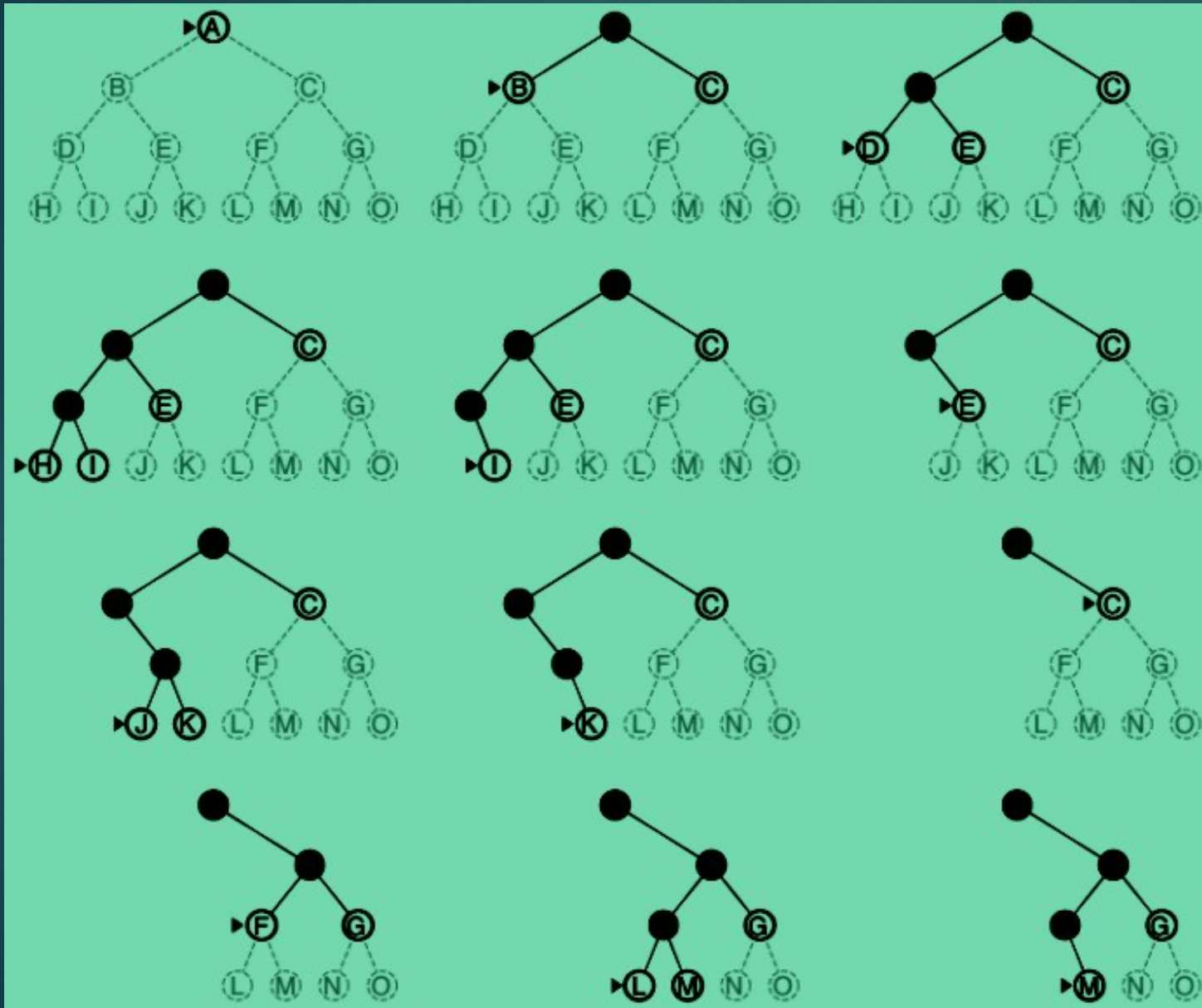
# Uniform-cost search (Cont'd)

- ▶ Uniform-cost search does not care about the *number* of steps a path has, but only about their total cost.
- ▶ Uniform-cost search is guided by path costs rather than depths

## E.g. 2: Depth-first search

- ▶ **Depth-first search** always expands the *deepest* node in the current frontier of the search tree.
- ▶ The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- ▶ As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.
- ▶ Breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.
- ▶ A LIFO queue means that the most recently generated node is chosen for expansion.
- ▶ This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

# Depth-first search on a binary tree



- The unexplored region is shown in light Gray
- Explored nodes with no descendants in the frontier are removed from memory.
- Nodes at depth 3 have no successors and M is the only goal node.



# Depth-limited search

- ▶ The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit.
- ▶ Nodes at depth  $l$  are treated as if they have no successors.
- ▶ The depth limit solves the infinite-path problem.
- ▶ Depth-limited search will also be nonoptimal if we choose  $l > d$ .



# Informed search (Heuristic search)

- ▶ Strategies that know whether one non-goal state is “more promising” than another are called **informed search or heuristic search** strategies.
- ▶ one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy.
- ▶ Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm.

## Best-first Search

- ❖ The general approach we consider is called **best-first search**.
- ❖ Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm
- ❖ A node is selected for expansion based on an **evaluation function**,  $f(n)$ .
- ❖ The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first.
- ❖ The implementation of best-first graph search is identical to that for uniform-cost search, except for the use of  $f$  instead of  $g$  to order the priority queue.
- ▶ Most best-first algorithms include as a component of  $f$  a **heuristic function**, denoted  $h(n)$   
 $h(n)$  = estimated cost of the cheapest path from the state at node  $n$  to a goal state.



# Greedy best-first search

**Greedy best-first search** tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is,  $f(n) = h(n)$ .

## STRAIGHT- LINE DISTANCE HEURISTIC

- Route-finding problems in Romania - can use the straight line distance heuristic,  $h_{SLD}$ .
- If the goal is Bucharest, we need to know the straight-line distances to Bucharest. For example,  $h_{SLD}(\text{In}(\text{Arad})) = 366$ .

## Greedy best-first search using $h_{SLD}$ to find a path from Arad to

### Bucharest

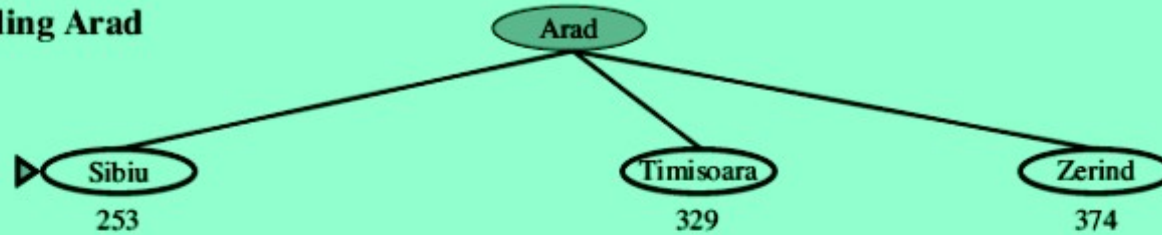
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Values of  $h_{SLD}$ —straight-line distances to  
Bucharest

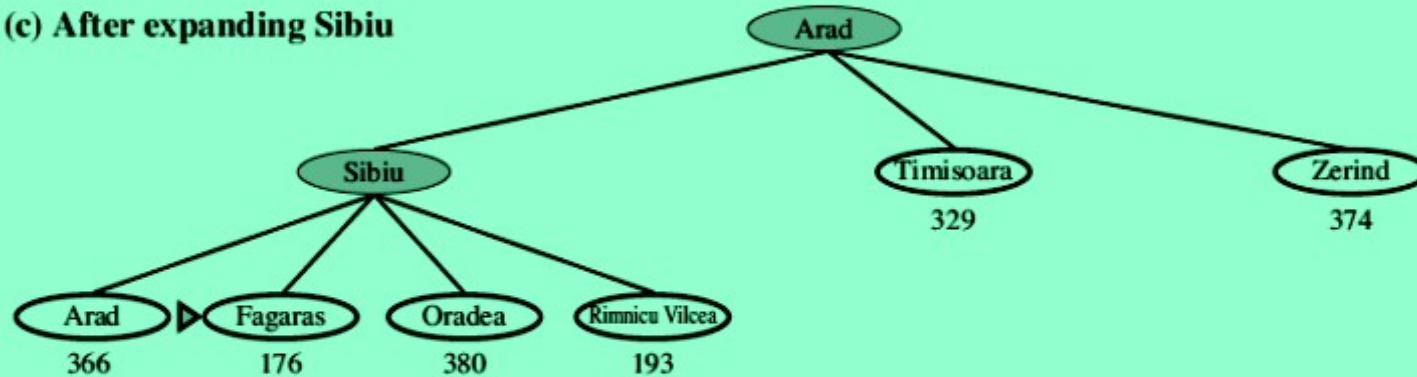
(a) The initial state



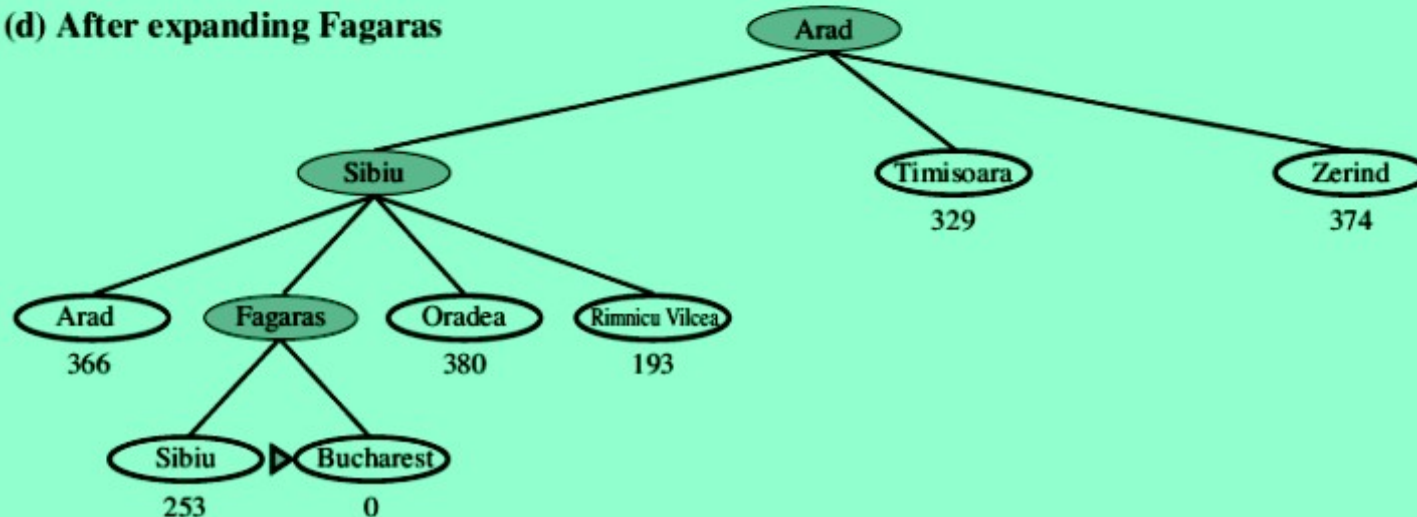
(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras



## Stages in a greedy best-first search for Bucharest with the straight-line distance heuristic $h_{SLD}$ .

- Nodes are labelled with their  $h$ -values.
- Figure shows the progress of a greedy best-first search using  $h_{SLD}$  to find a path from Arad to Bucharest.
- The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara.
- The next node to be expanded will be Fagaras because it is closest.
- Fagaras in turn generates Bucharest, which is the goal.
- For this particular problem, greedy best-first search using  $h_{SLD}$  finds a solution without ever

expanding a node that is not on



# A\* search: Minimizing the total estimated solution cost

- ▶ The most widely known form of best-first search is called A \* search.
- ▶ It evaluates nodes by combining  $g(n)$ , the cost to reach the node, and  $h(n)$ , the cost to get from the node to the goal:

$$f(n) = g(n) + h(n)$$

$g(n)$  gives the path cost from the start node to node  $n$ ,

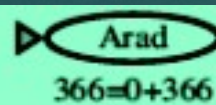
$h(n)$  is the estimated cost of the cheapest path from  $n$  to the goal,

$f(n)$  = estimated cost of the cheapest solution through  $n$

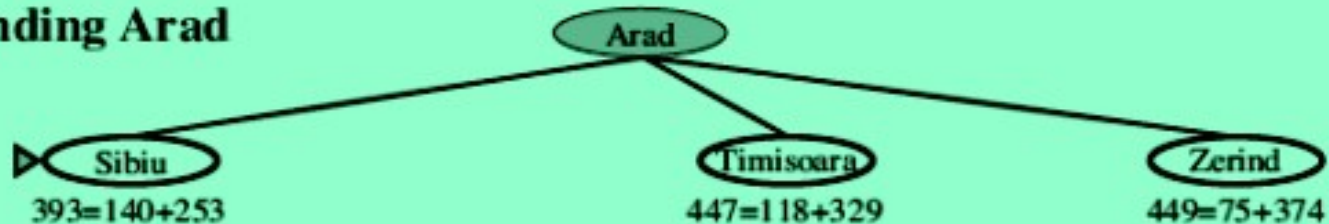
- Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of  $g(n) + h(n)$ .
- It turns out that this strategy is more than just reasonable: provided that the heuristic function  $h(n)$  satisfies certain conditions

A \* search is **both complete and optimal**.

(a) The initial state



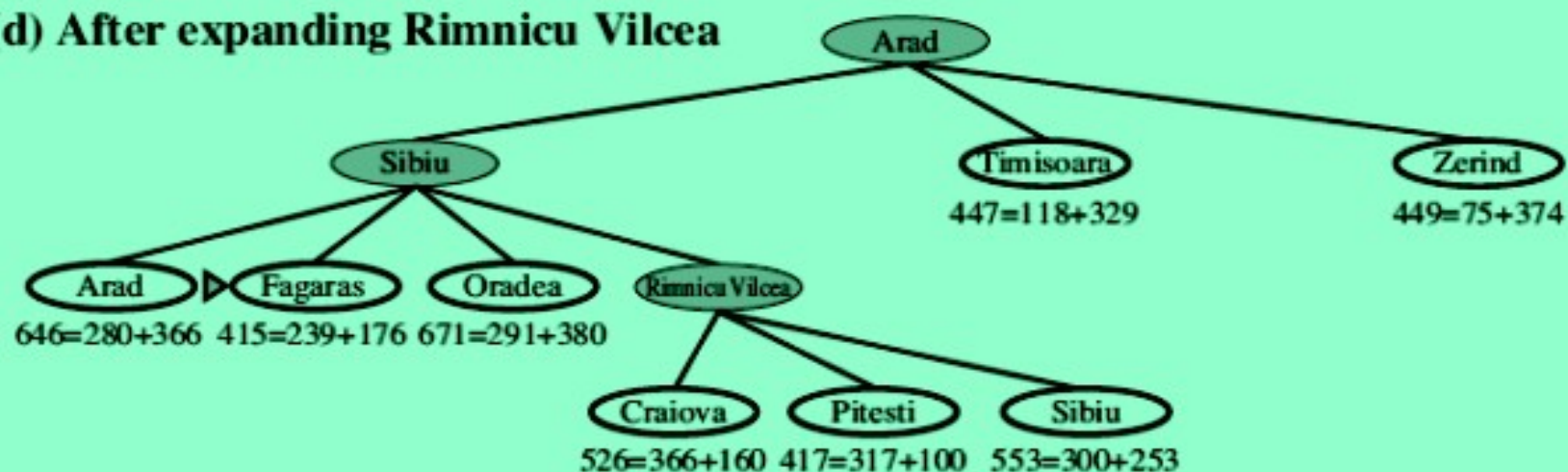
(b) After expanding Arad



(c) After expanding Sibiu



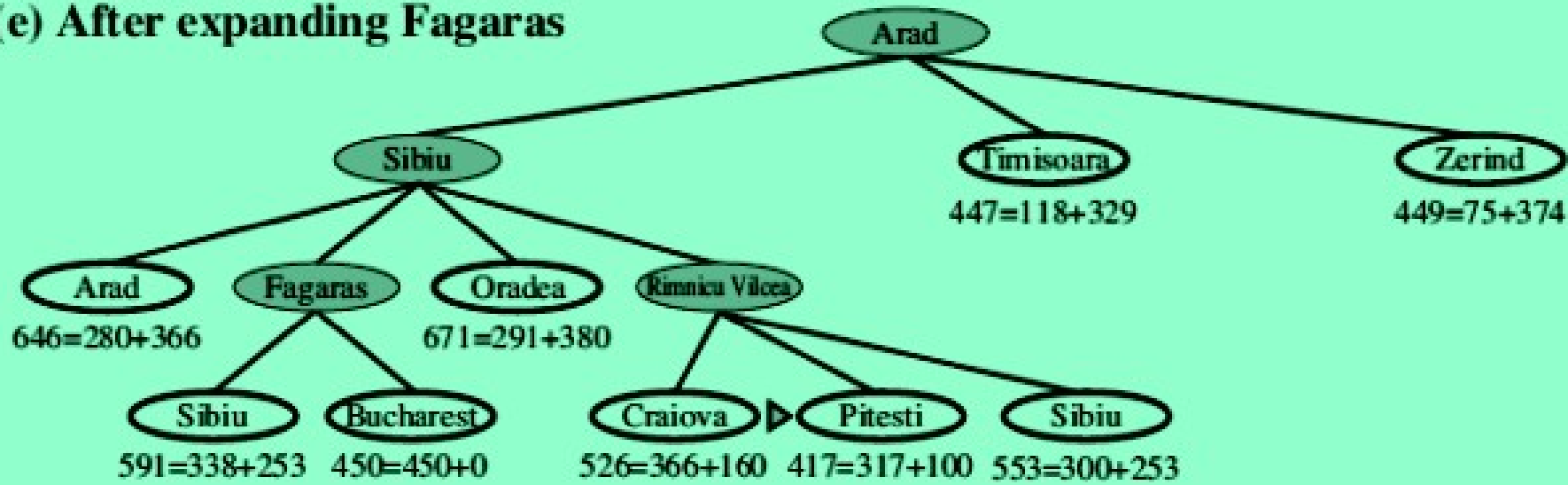
(d) After expanding Rimnicu Vilcea



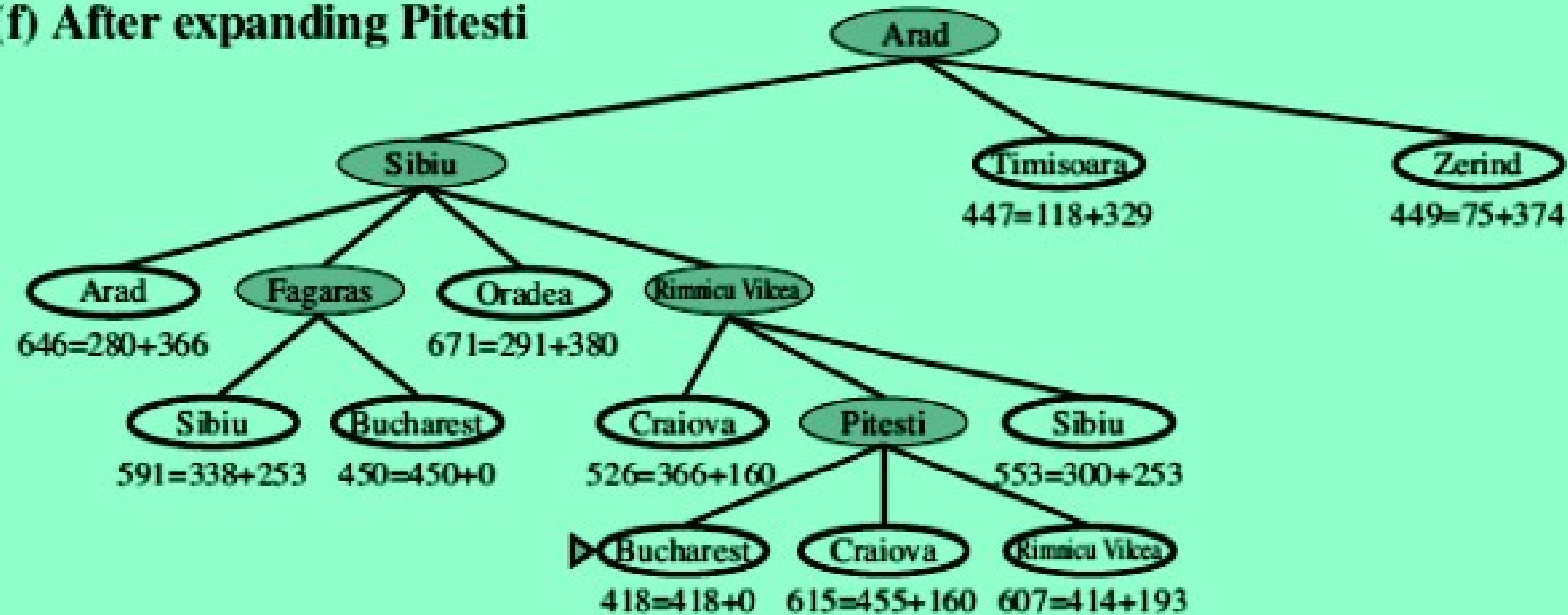
- Stages in an  $A^*$  search for Bucharest.
- Nodes are labeled with  $f = g + h$ .
- The  $h$  values are the straight-line distances to Bucharest



(e) After expanding Fagaras



(f) After expanding Pitesti



- Stages in an A\* search for Bucharest.
- Nodes are labeled with  $f = g + h$ .
- The  $h$  values are the straight-line distances to Bucharest

# Heuristic functions

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

A typical instance of the 8-puzzle. The solution is 26 steps long.

## Heuristics for the 8-puzzle

- Object of the puzzle - slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration.
- The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3.



# Heuristic functions

► Two commonly used candidates (heuristics) for the 8-puzzle

- $h_1$  = the number of misplaced tiles.

For Figure given in above all of the eight tiles are out of position, so the start state would have  $h_1 = 8$ .  $h_1$  is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.

- $h_2$  = the sum of the distances of the tiles from their goal positions.

Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the **city block distance** or **Manhattan distance**.

$h_2$  is also admissible because all any move can do is move one tile one step closer to the goal.

Tiles 1 to 8 in the start state give a Manhattan distance of

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.$$

As expected, neither of these overestimates the true solution cost, which is 26.

