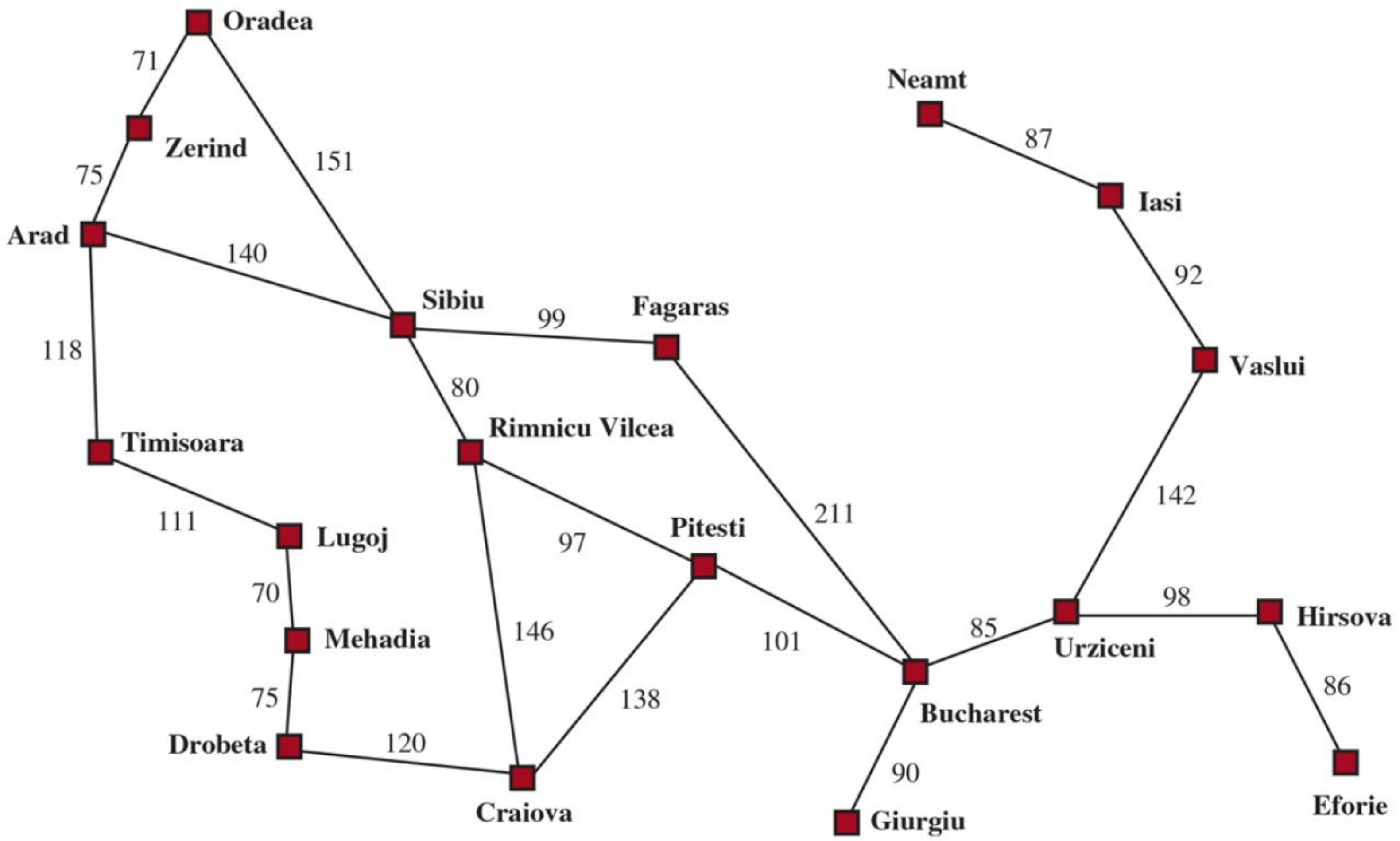
# Solving Problems by Searching

Thitipong Tanprasert

# Problem-solving Agent

- When the correct action to take is not immediately obvious,
  - an agent may need to to **plan ahead**:
  - to consider a sequence of actions that form a path to a *goal state*
  - The computational process it undertakes is called ***search***.

# A Touring Vacation in Romania




A simplified road map of part of Romania, with road distances in miles.

# Map: The Information About The World

- With that information, the agent can follow this four-phase problem-solving process:
  - GOAL FORMULATION : e.g. Bucarest
  - PROBLEM FORMULATION :
    - considers the actions of traveling from one city to an adjacent city
    - The only fact about the state of the world (that will change due to an action) is *the current city*
  - SEARCH :
    - simulates sequences of actions in its model
    - searching until it finds a sequence of actions that reaches the goal (a ***solution***)
  - EXECUTION (the actions the solution)

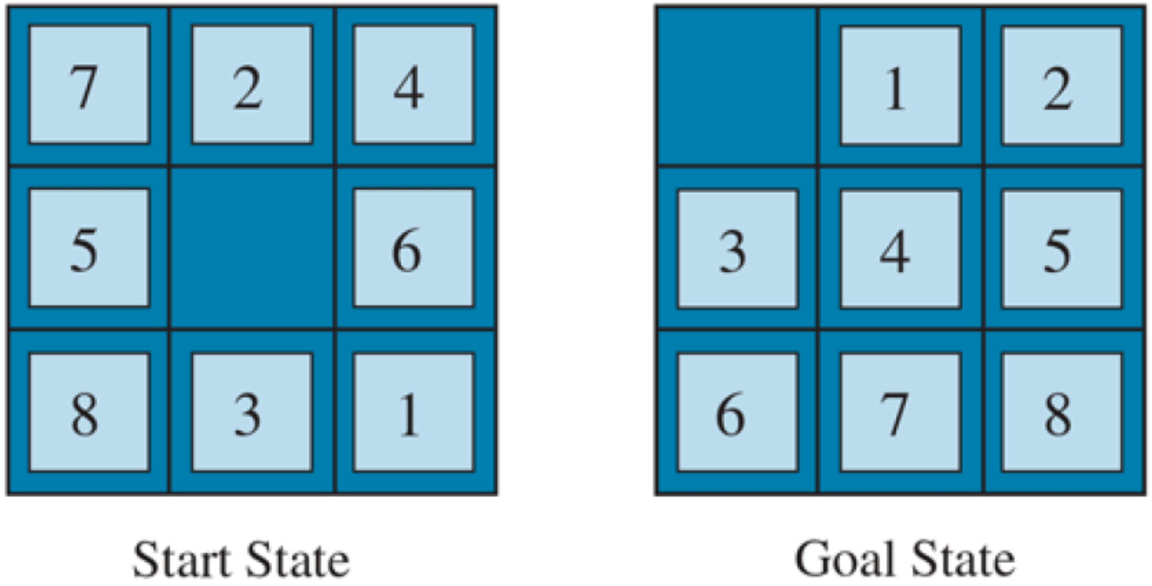
# A Search Problem

- **State space :** A set of possible states that the environment can be in
- **The initial state** that the agent starts in
- A set of one or more **goal states**
- **ACTION(s)** returns a finite set of actions that can be executed in s
- A **transition model :** RESULT(Arad, ToZerind) = Zerind
- An **action cost function**
  - reflects performance measure (e.g. distance, time)
- An **optimal solution** has the lowest cost among all solutions

# Formulating problems

- The problem of getting to Bucharest is a **model**
  - not the real thing.
  - The process of removing detail from a representation is called **abstraction**

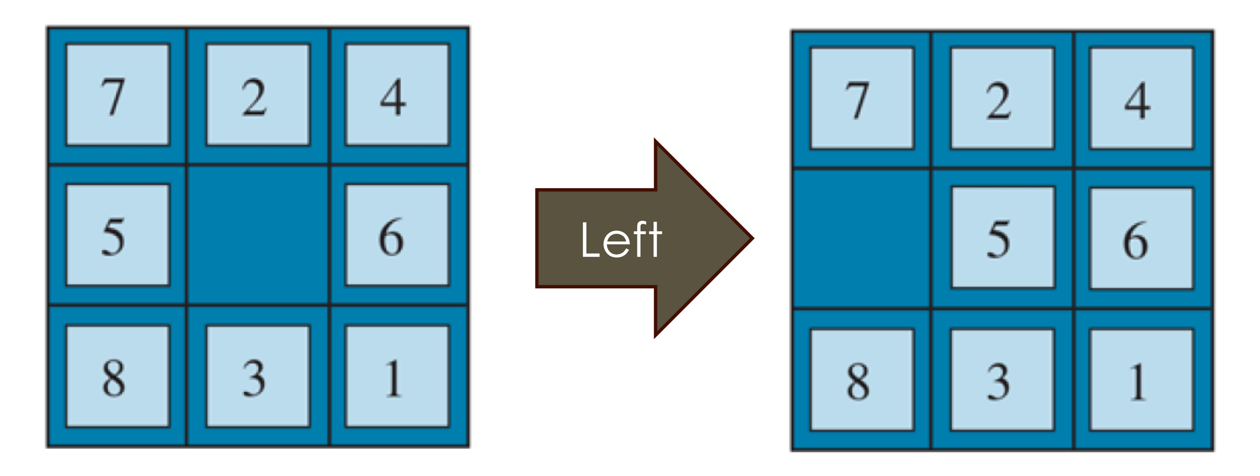
# 8-puzzle




A typical instance of the 8-puzzle.

# 8-puzzle

- **STATES:** A state description specifies the location of each of the tiles.
- **INITIAL STATE:** Any state can be designated as the initial state
  - State space is partitioned into two halves.
- **ACTIONS:** blank space moving Left, Right, Up, or Down.
  - If the blank is at an edge or corner then not all actions will be applicable.
- **TRANSITION MODEL:** Maps a state and action to a resulting state




- **GOAL STATE**
- **ACTION COST:** Each action costs 1.

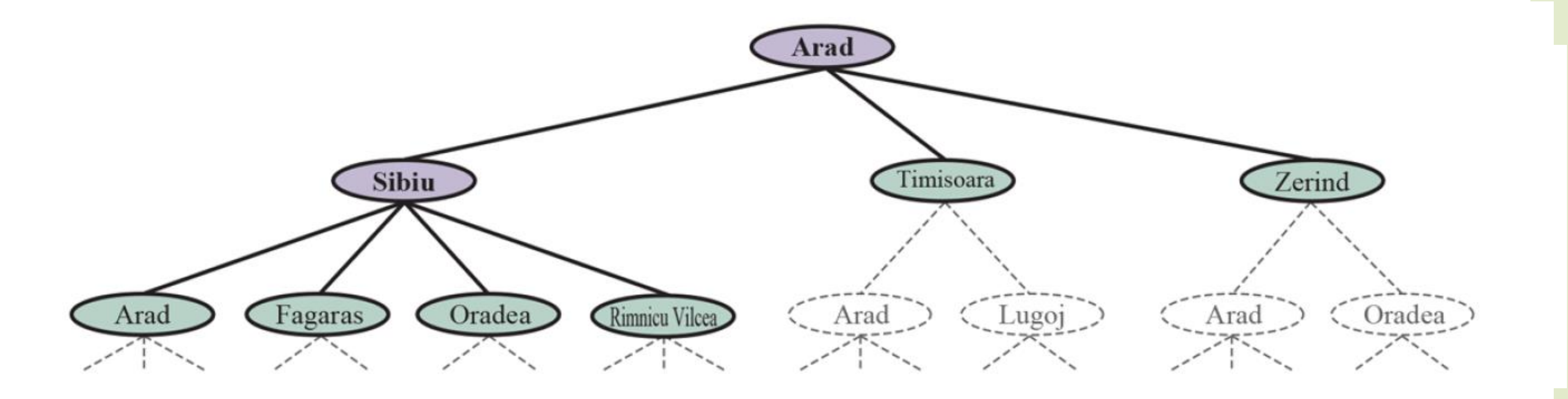
# Search Algorithms

- Algorithms that superimpose a search tree over the state-space graph
  - forming various paths from the initial state
  - trying to find a path that reaches a goal state

**The state space**
- describes the (possibly infinite) set of states in the world, and
- the actions that allow transitions from one state to another

**The search tree**
- describes paths between these states, reaching towards the goal
- may have multiple paths to any given state
- but each node in the tree has a unique path back to the root

# A Touring Vacation in Romania



# Best-first search : A general approach

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

# Measuring problem-solving performance

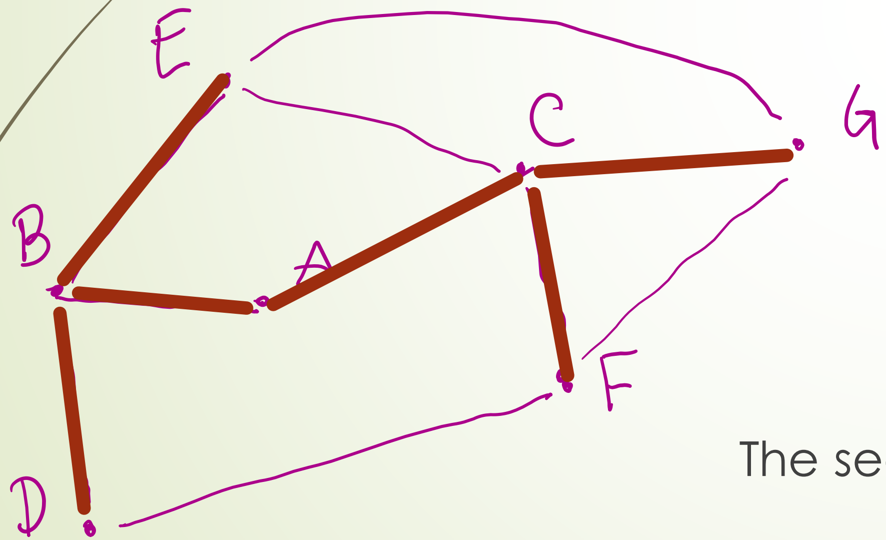
- COMPLETENESS:
  - Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
- COST OPTIMALITY:
  - Does it find a solution with the lowest path cost of all solutions?
- TIME COMPLEXITY
- SPACE COMPLEXITY
- In many AI problems, the state-space graph is represented only *implicitly* by the initial state, actions, and transition model.
- For an implicit state space, complexity can be measured in terms of
  - the **depth** or number of actions in an optimal solution; d
  - the **maximum number of actions** in any path; m
  - and **the branching factor** or number of successors of a node that need to be considered; b

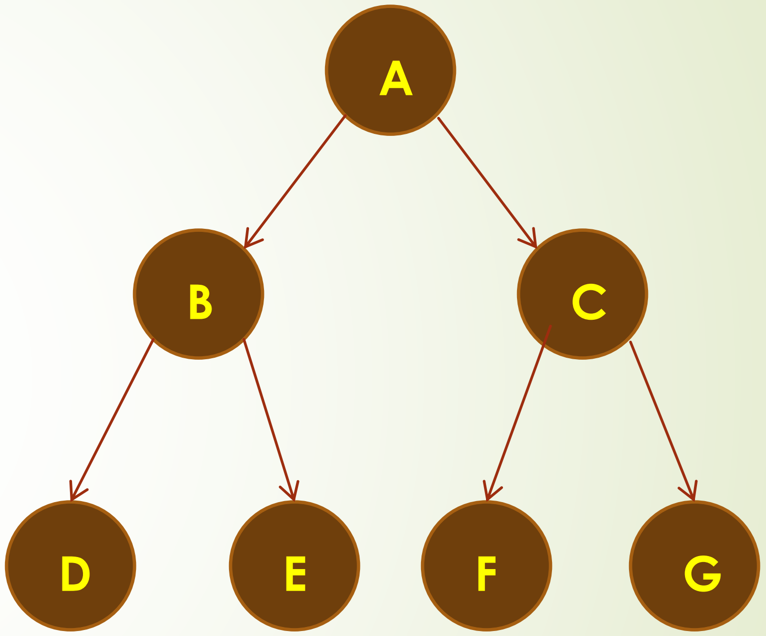
# Uninformed Search Strategies

- An **uninformed** search algorithm is given ***no clue about how close a state is to the goal(s).***
  - For example, consider our agent in ***Arad*** with the goal of reaching ***Bucharest***.
  - An **uninformed agent** with no knowledge of Romanian geography has no clue whether going to ***Zerind*** or ***Sibiu*** is a better first step.
- In contrast, an **informed agent** who knows the location of each city knows that ***Sibiu*** is much closer to ***Bucharest*** and thus more likely to be on the ***shortest path***.

# Breadth First Search

- When all actions have the same cost
- This is a systematic search strategy that is therefore complete even on infinite state spaces.




The sequence of states being searched

# Breadth-first Search

- always finds a solution with a minimal number of actions
  - When it is generating nodes at depth d, it has already generated all the nodes at depth d-1.
  - If one of them were a solution, it would have been found.
- It is complete.
- Suppose that the solution is at depth d, with branching factor = b.
  - The total number of nodes generated is $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$
  - All the nodes remain in memory
    - so both time and space complexity are $O(b^d)$

# Breadth-first Search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

avoiding repeated states

# Simplest Algorithm Structure of BFS

```
s = initial_state
while not Goal(s)
    for each successor_state x of s
        enqueue(x)
    s = dequeue()
```

# Tracing back for path

```
PRINT-PATH(G, s, v)
if v == s
    print s
elseif v.π == NIL
    print "no path from" s "to" v "exists"
else PRINT-PATH(G, s, v.π)
    print v
```

NOTE:

- G is state space graph
- s is initial state
- v is the current state
- $\pi$ is the parent state according to Breadth-first Search