Good morning !!!

SOLVING PROBLEMS BY SEARCHING

Chapter 3

Outline

- Problem-solving agents
- Problem formulation
- Example problems
- Uninformed search algorithms

Problem solving **agent**

- Particular type of **goal-based agent**
  - Goal-based agents act to achieve their goals

- **Problem-solving agents** use **atomic representation**
  - states of the world have no internal structure visible to the problem-solving algorithm

- **Solution:** fixed sequence of actions (in this chapter)

Problem solving **agent**

- **Intelligent agents** are supposed to **maximize their performance measure**

    Achieving this is sometimes **simplified**

    if the agent can **select a goal** and aim at **satisfying it**

Example: Romania

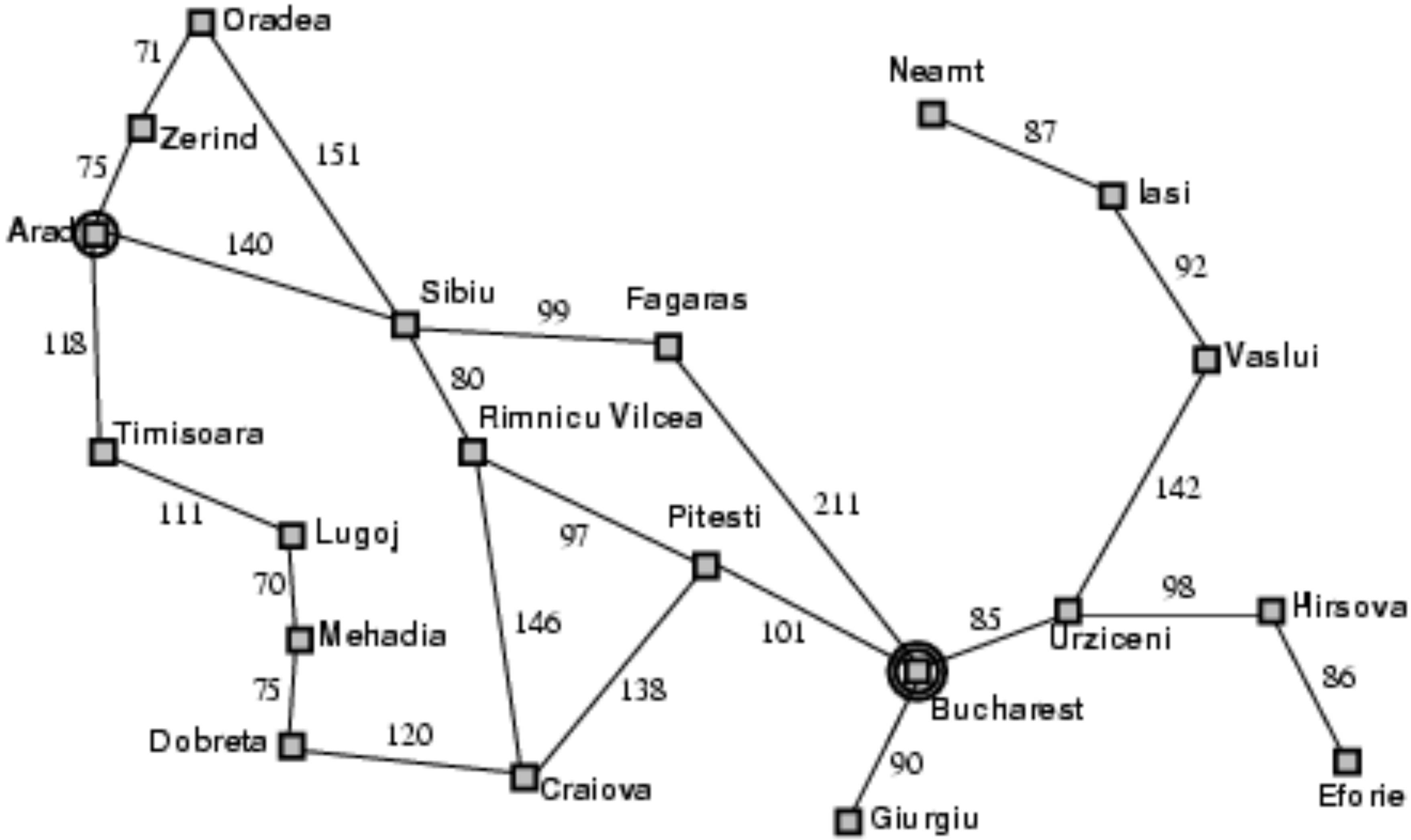
- On road **trip** in Romania. Currently **in Arad**
- **Flight back** leaves tomorrow **from Bucharest**

- 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



Formulate, search and execute

- If the **environment** is:
  - observable (the agent always knows the current state)
  - known (the agent knows which states are reached by each action)
  - deterministic (each action has exactly one outcome)

  → **a solution** to a problem is a fixed sequence of actions
- **Search:** process of looking for such a sequence
- **Search algorithm:**
  - **input**: problem
  - **output**: an action sequence
- The sequence in output can then be **executed**

Problem-solving agent

Goal formulation based on the current situation and the agent's performance measure

```
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 do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Problem formulation: what **states and actions** to consider given the goal

Search: Decide **what to do next** by exploring consequences of actions in the future and if they lead to the goal

Problem-solving agents

This simple problem-solving agent

1. **Formulates** a **goal** and a **problem**
2. **Searches** for a *sequence of actions* that would solve the problem
3. **Executes the actions** one at a time

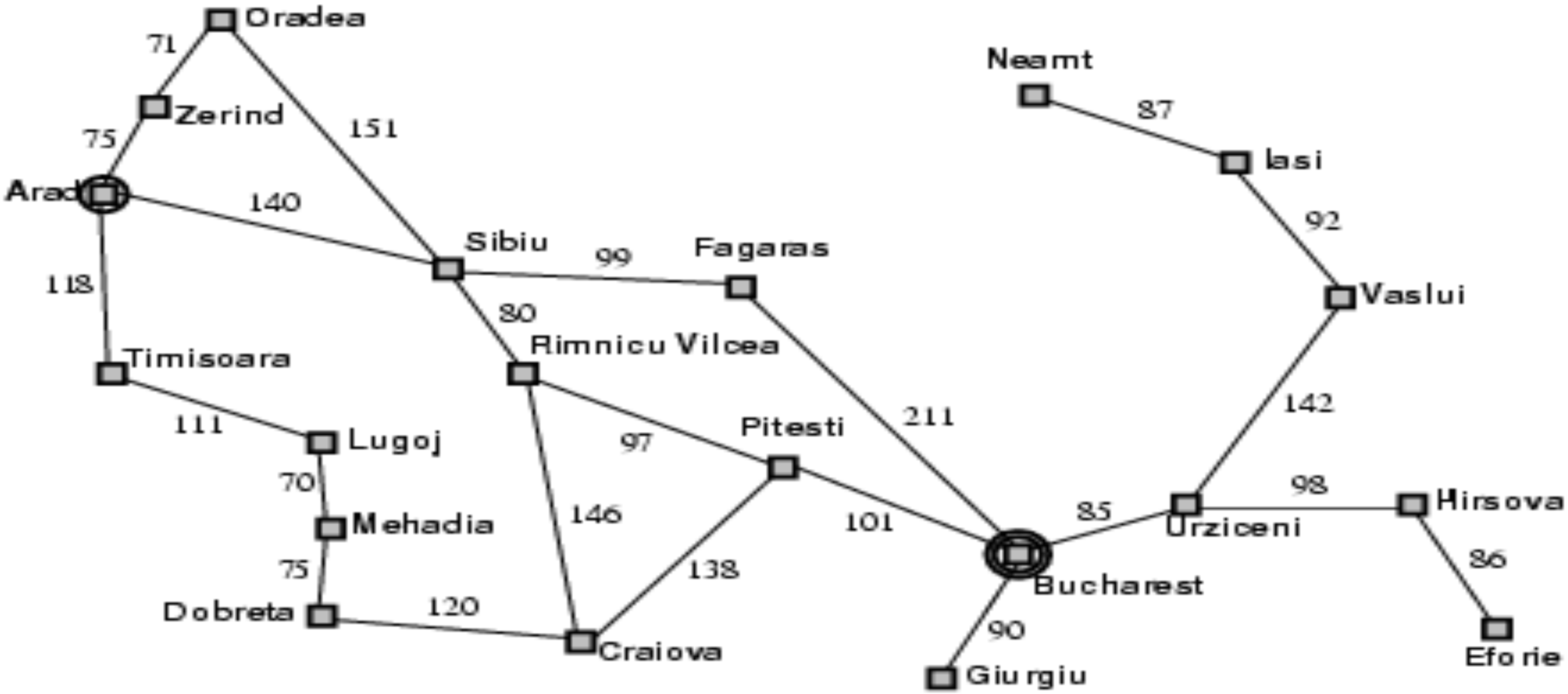
When this is complete,
it formulates **another goal** and **starts over**

Outline

- Problem-solving agents
- Problem formulation
- Example problems
- Uninformed search algorithms

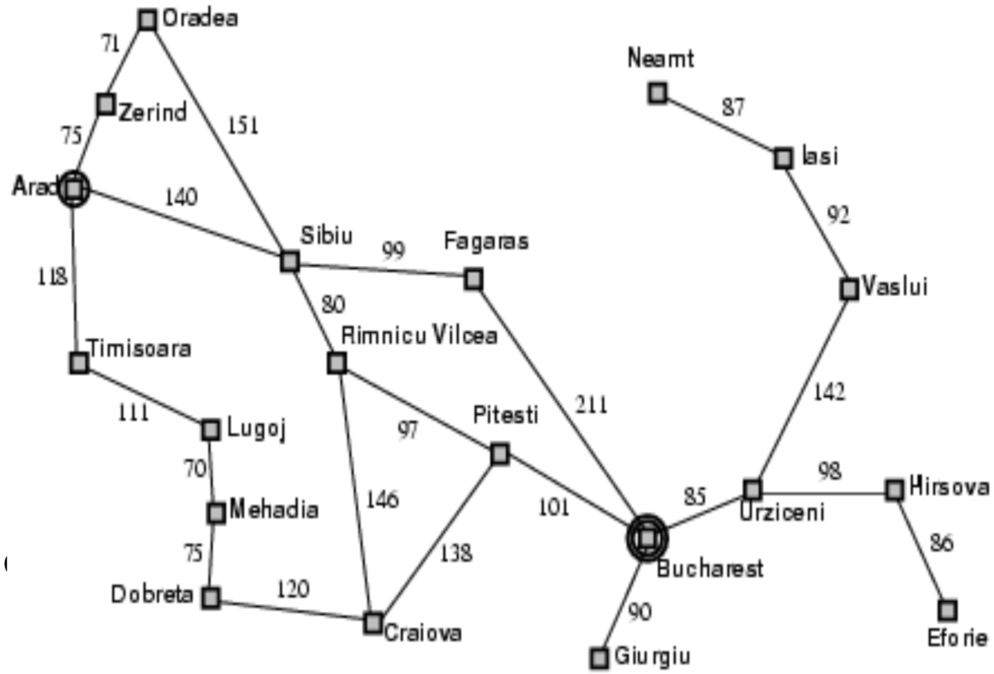
Example: Romania

- On road **trip** in Romania.
- Currently **in Arad**
- **Flight back** leaves tomorrow **from Bucharest**



Example: Romania

- Currently **in Arad**
- **Flight back** leaves tomorrow **from Bucharest**



- Formulate goal
  - Be in Bucharest
- Formulate problem
  - states: various cities
  - actions: drive between citi
- Find solution
  - sequence of cities (e.g., Arad, Sibiu, Fagaras, Bucharest)

Problem formulation

A **problem** can be defined formally by:

- **Initial state**
- **Actions**
- **Transition model**
- **Goal test**
- **Path cost**

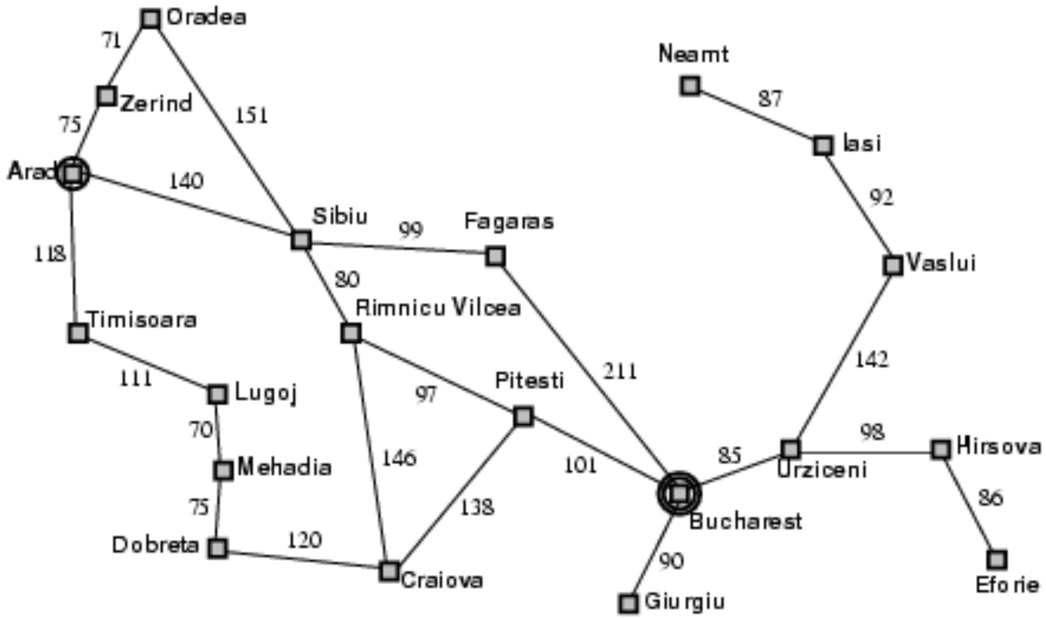
Solution: **sequence of actions** (path) leading from the initial state to a goal state

Optimal solution: a solution with minimal path-cost

Problem formulation

A **problem** can be defined formally by:

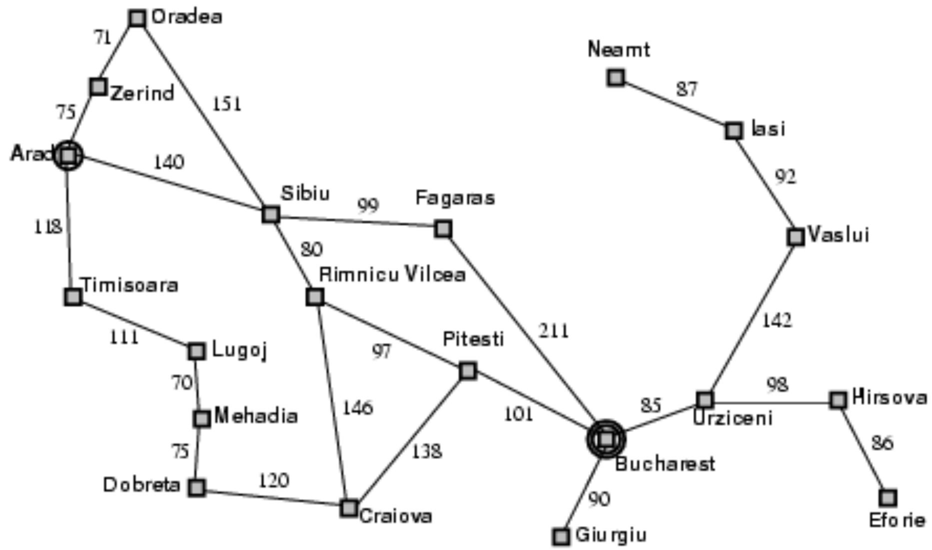
- **Initial state** that the agent starts in
  - **Example:** In(Arad)



Problem formulation

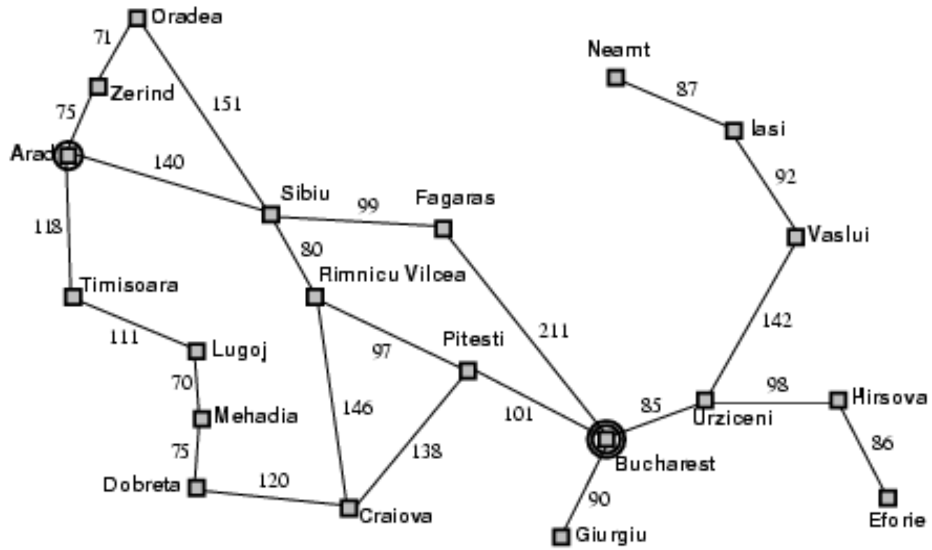
- **Actions** available to the agent
  - Given a state **s**, **ACTIONS(s)** returns the set of actions that can be executed in s
  - We say that each of these actions is **applicable** in **s**

    - **Example:** from the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.



Problem formulation

- **Transition model:**
  - **RESULT(s,a)** returns the **state** obtained from doing action **a** in state **s**
  - **Successor:** any state reachable from a given state by a single action
    - Example: **RESULT** (In(Arad), Go(Zerind)) = In(Zerind)



Problem formulation

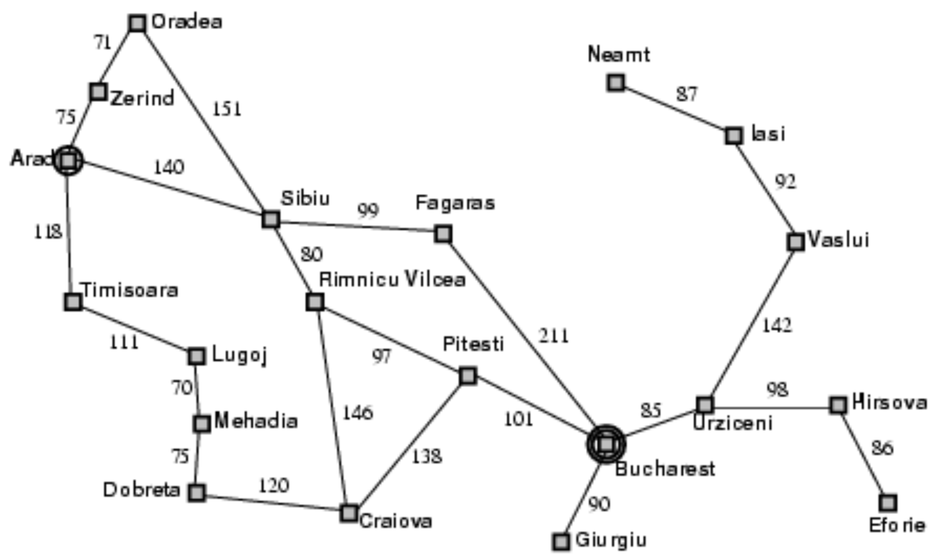
- Problem **state space** = **(initial state, actions, transition model)**

- **State space** = the set of all states reachable from the initial state by any sequence of actions

- The **state space** can be depicted as a **directed graph:**
  nodes → states
  edges → actions
  path → sequence of states *connected* by a sequence of actions

Problem formulation

- **Goal test:** allows to check if a state is a **goal**
  - **Example:** The agent's goal in Romania is the singleton set { In(Bucharest) }



Problem formulation

The **state space** can be depicted as a **directed graph**:

nodes → states

edges → actions

path → sequence of states connected by a sequence of actions

Path cost: numeric value associated to each path reflecting the desired performance measure

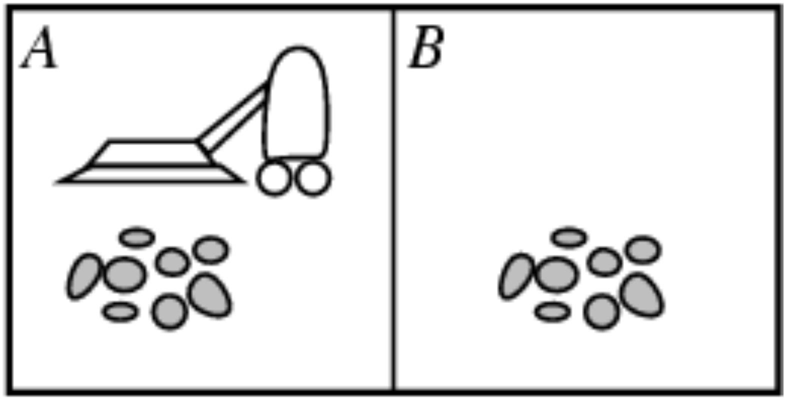
- e.g., sum of distances, number of actions executed, etc.
- ***c(x,a,y)*** is the step cost, for going from state x to state y by performing action a
- assumed to be ≥ **0**
- We assume path costs to be additive: **sum of step costs**

Problem formulation

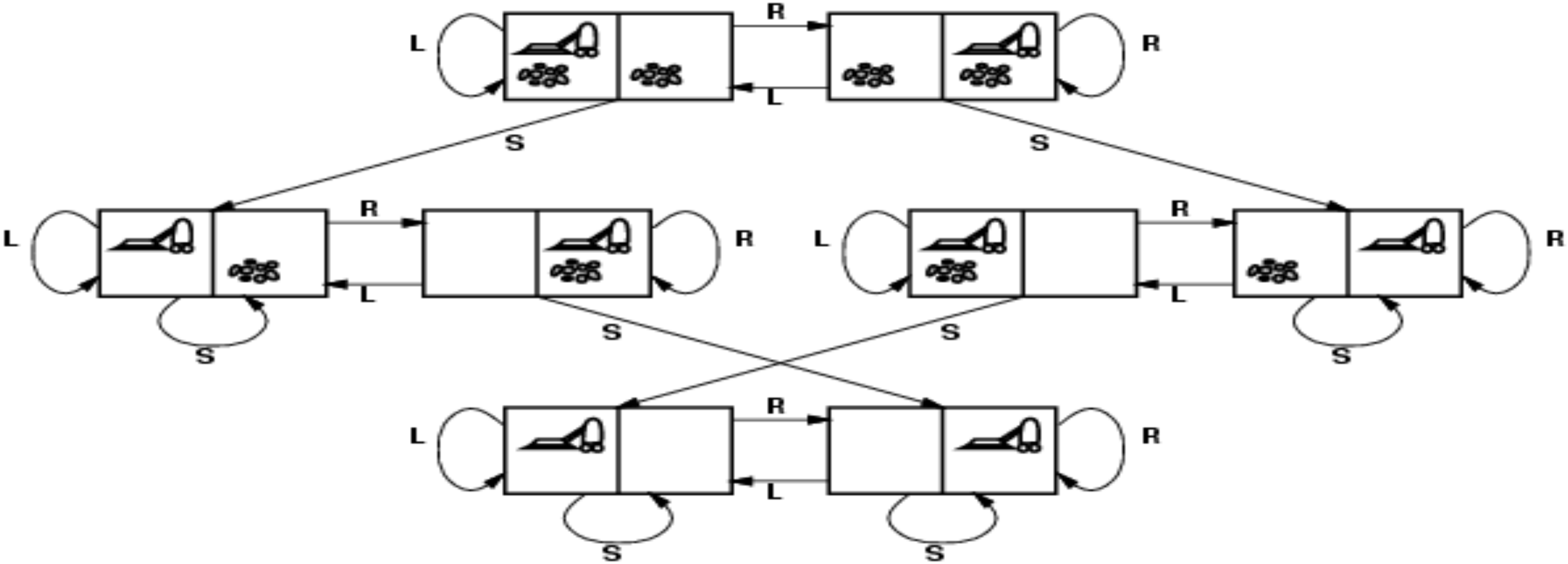
- **Solution:** a sequence of actions (path) leading from the initial state to a goal state

- **Optimal solution:** a solution with minimal path-cost

Example: Vacuum-cleaner world

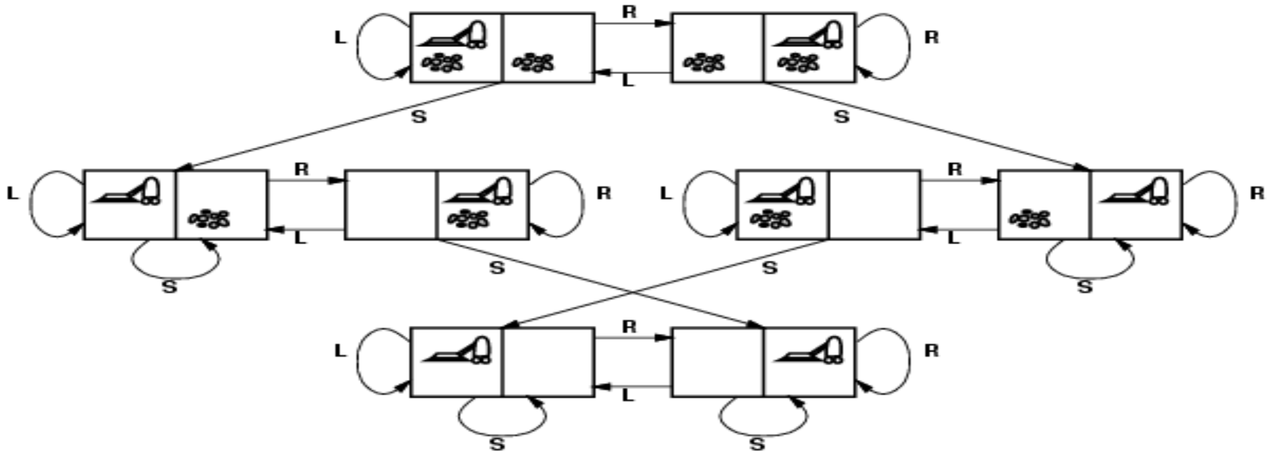


Example: Vacuum world



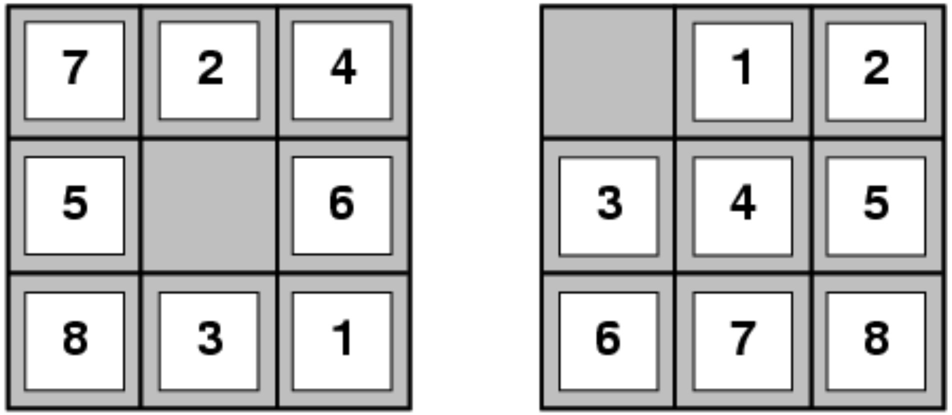
- states?
- initial state?
- actions?
- transition model?
- goal test?
- path cost?

Example: Vacuum world



- states? dirt locations and robot location
- initial state? any state
- actions? *Left*, *Right*, *Suck*
- *Trasition model? The actions have the 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? no dirt at all locations
- path cost? Each step costs 1 → path cost is the number of steps in the path

Example: The 8-puzzle

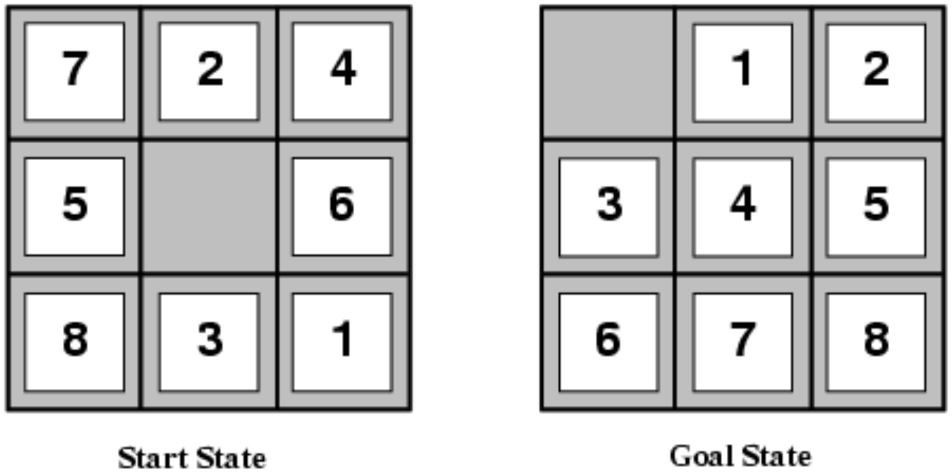


Start State　　Goal State

The 8-puzzle

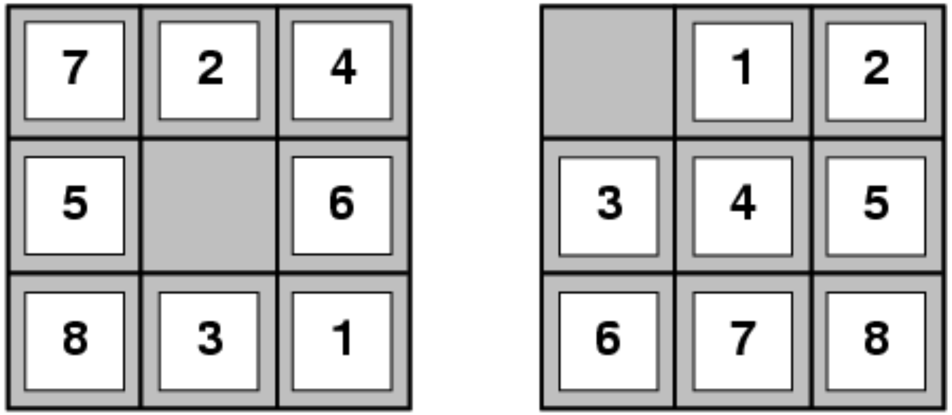
- A 3×3 board with **eight numbered tiles** and a **blank space**
- A **tile adjacent** to the blank space **can slide** into that space
- Object: to reach a specified goal state such as the one shown on the right of the figure

Example: The 8-puzzle



- states?
- initial state?
- actions?
- transition model?
- goal test?
- path cost?

Example: The 8-puzzle



Start State

Goal State

- states? locations of eight tiles and the blank in one of the nine squares
- initial state? any
- actions? move blank left, right, up, down
- transition model? Given a state and an action, it returns the resulting state (that is, the puzzle configuration after moving the blank)
- goal test? goal state (given)
- path cost? Each step cost 1 per move, path cost is the number of steps in the path

Outline

- Problem-solving agents
- Problem formulation
- Example problems
- Uninformed search algorithms

Review: Problem-solving agent

This simple problem-solving agent

1. **Formulates** a **goal** and a **problem**
2. **Searches** for a **sequence of actions** that would solve the problem
3. **Executes** the **actions** one at a time

When this is complete,
it formulates **another goal** and **starts over**

Review: Problem formulation

A **problem** can be defined formally by:

- **Initial state** that the agent starts in
- **Actions** available to the agent
- **Transition model:** A description of what each action does
- **Goal test:** Determines whether a given state is a **goal state**
- **Path cost:** **Numeric value** associated to each path reflecting the desired performance measure

The Search Tree

Solution: a **sequence of actions** (path) leading from the initial state to a goal state

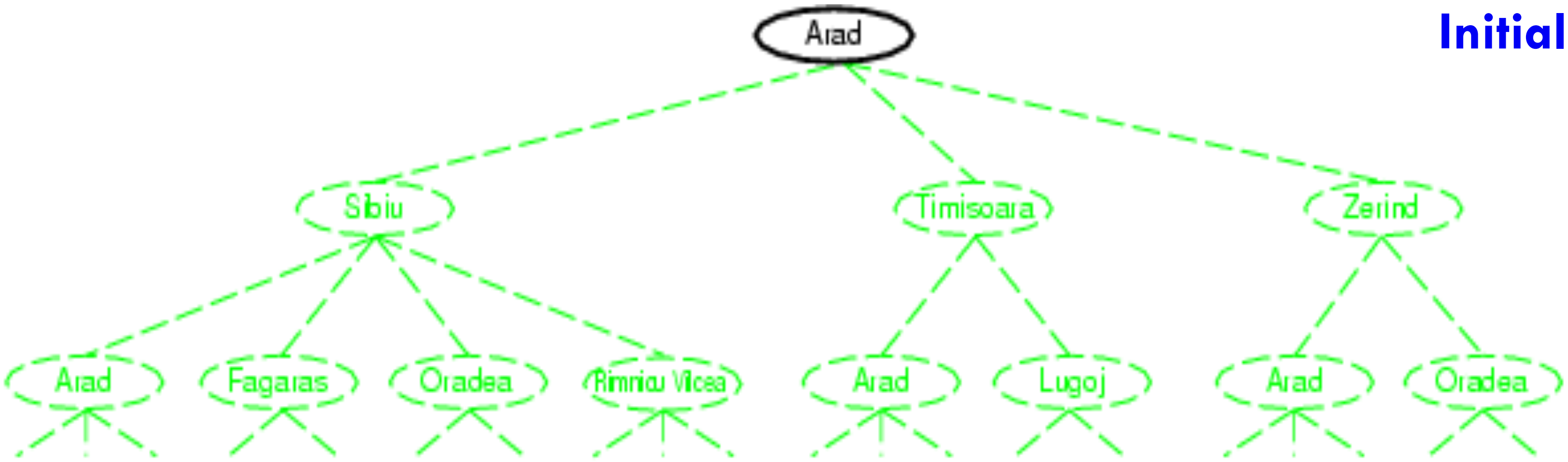
- **Search algorithms** consider possible **sequences of actions**
- Possible **sequences of actions** from initial state **form** a **search tree**
  - **Root** → initial state
  - **Nodes** → states
  - **Branches** → actions
  - The **same state** can appear **multiple time**
  - ***Outgoing* edges** from a node →
    all possible actions available in the state represented by the node

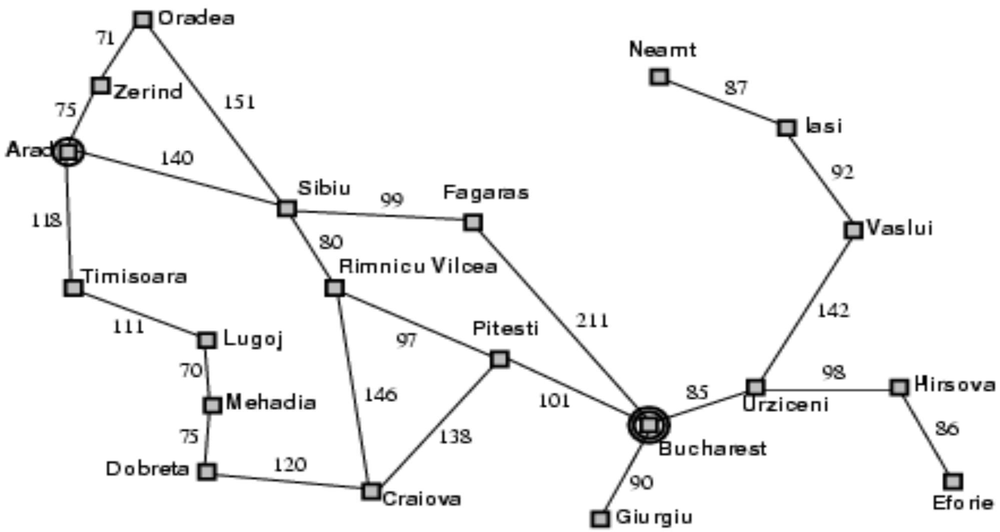
Partial search trees for finding a route **from Arad to Bucharest**

Tree search example

Initial state

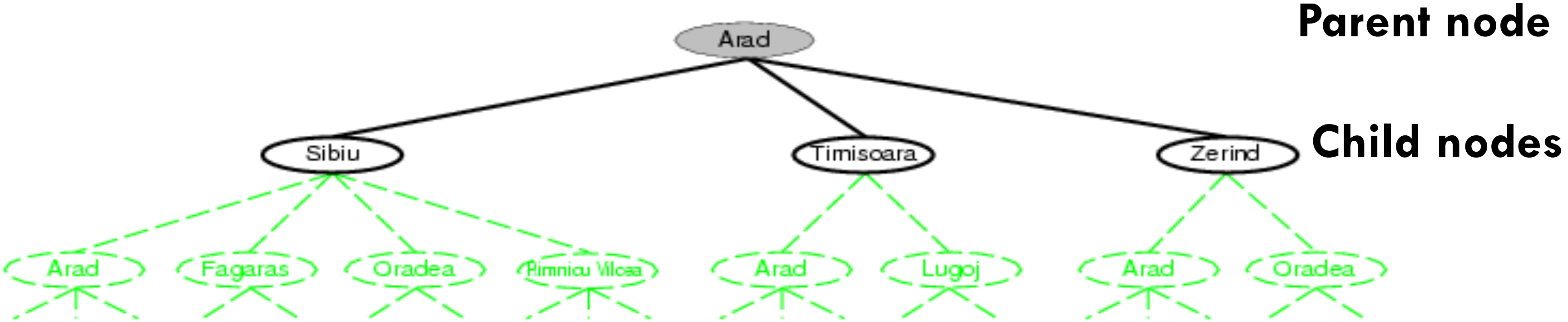
In(Arad)

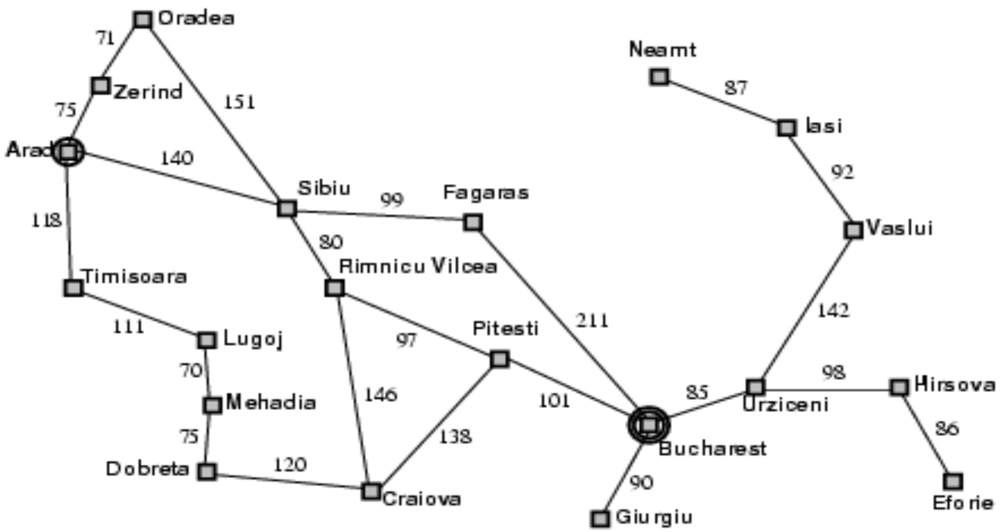




Tree search example

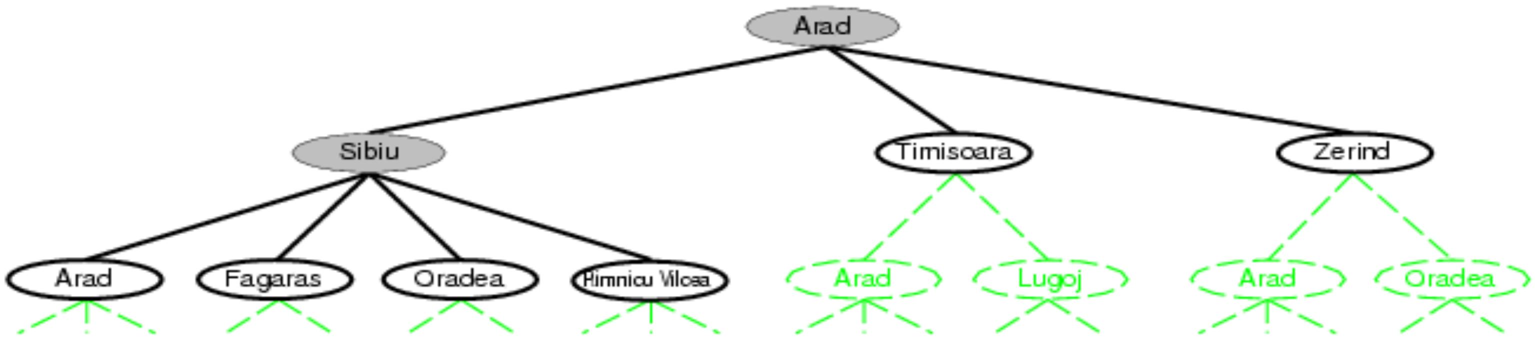
After expanding Arad





Tree search example

After expanding Sibiu

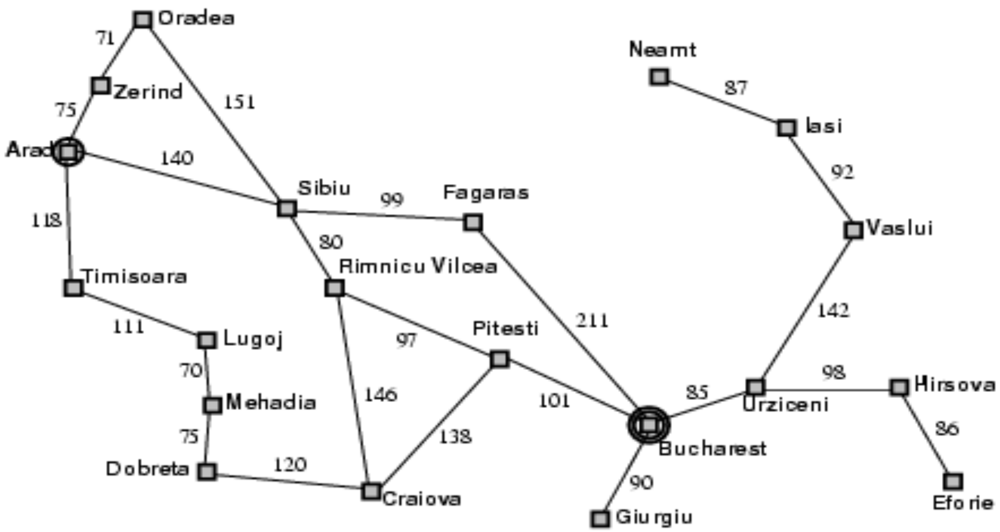


Leaf node:

a node with no children in the tree

Frontier:

The set of **all leaf nodes** available **for expansion** at any given point



Tree search algorithm

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

initialize the frontier using the initial state of problem

loop do

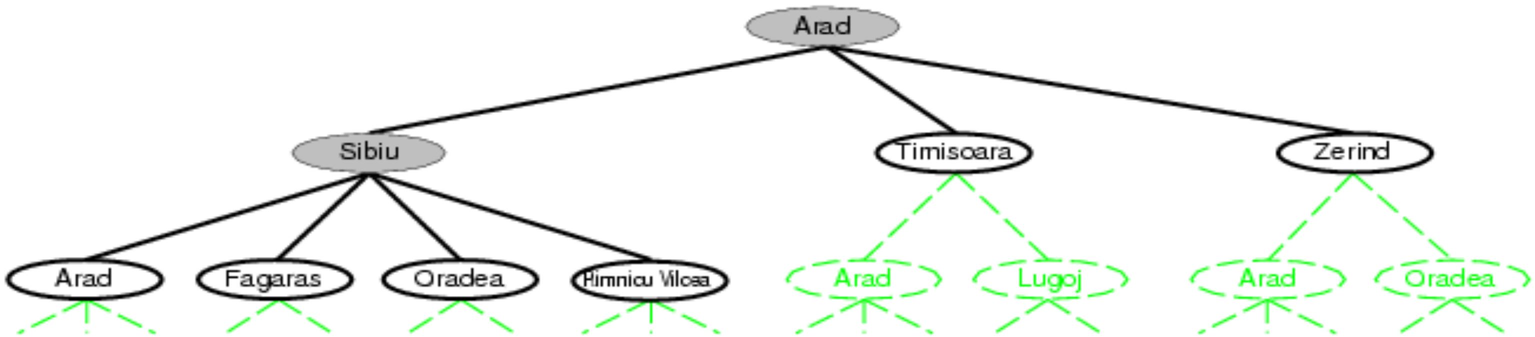
if the frontier is empty **then return failure**

choose a leaf node and remove it from the frontier

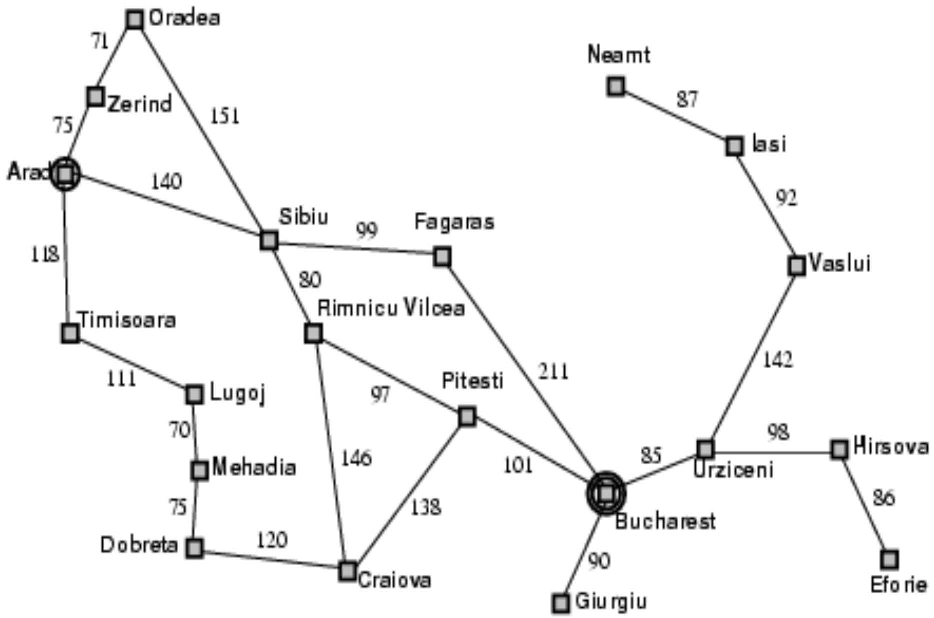
if the node contains a goal state **then return** the corresponding **solution**

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

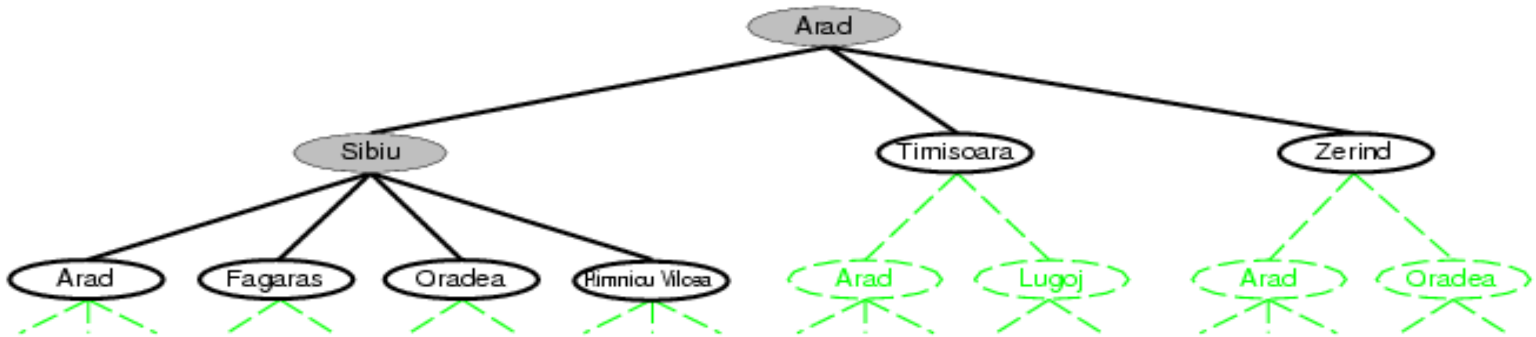
Tree search example



- The search tree includes the path **from Arad to Sibiu** and back **to Arad again**!
- The state **In(Arad)** is a **repeated state** generated by a **loopy path**

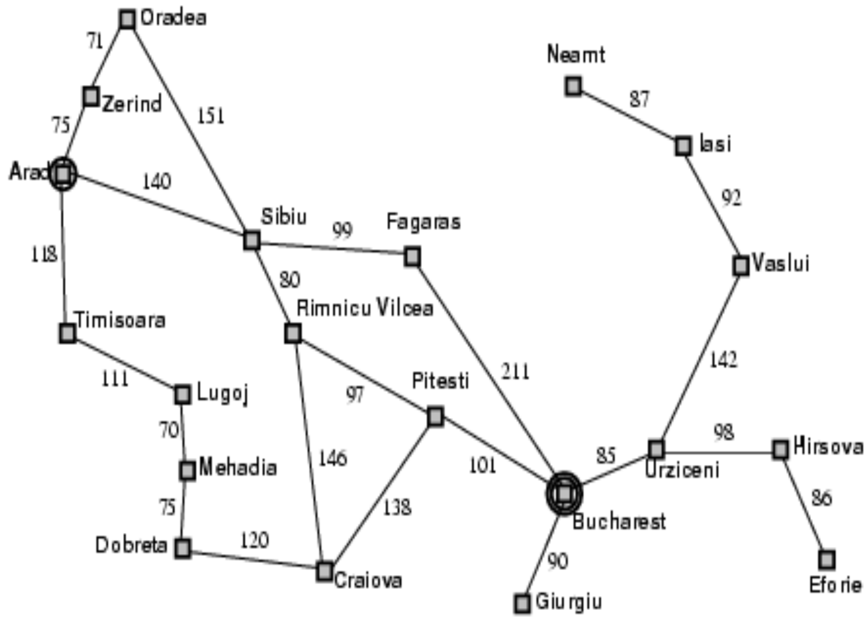


Tree search example

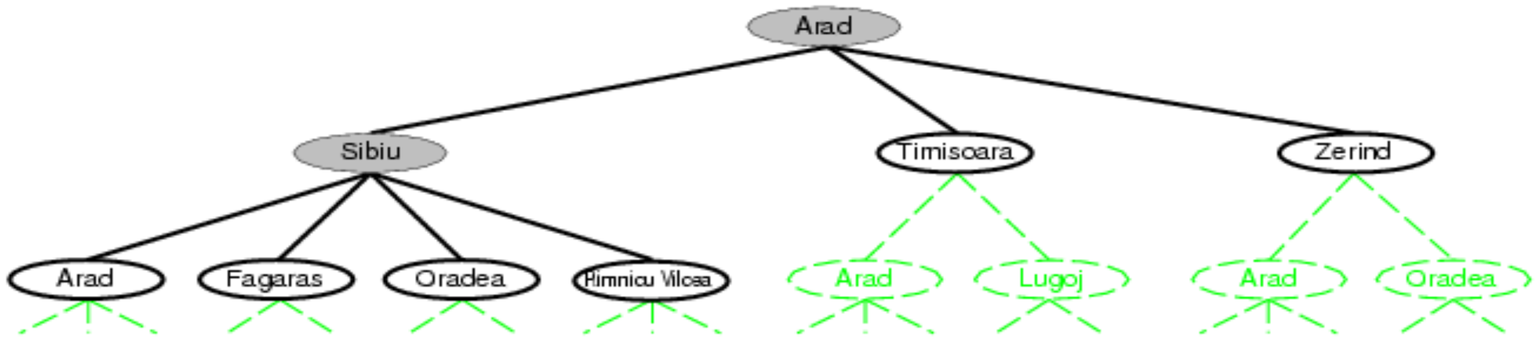


- **Loopy paths:** special case of **redundant paths**

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

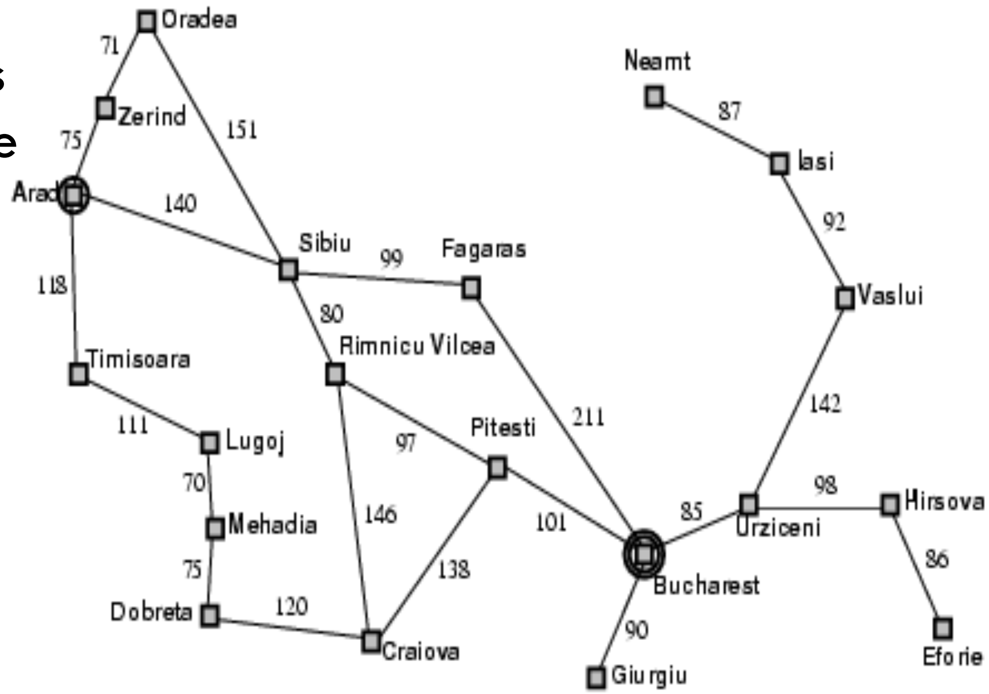


Tree search example

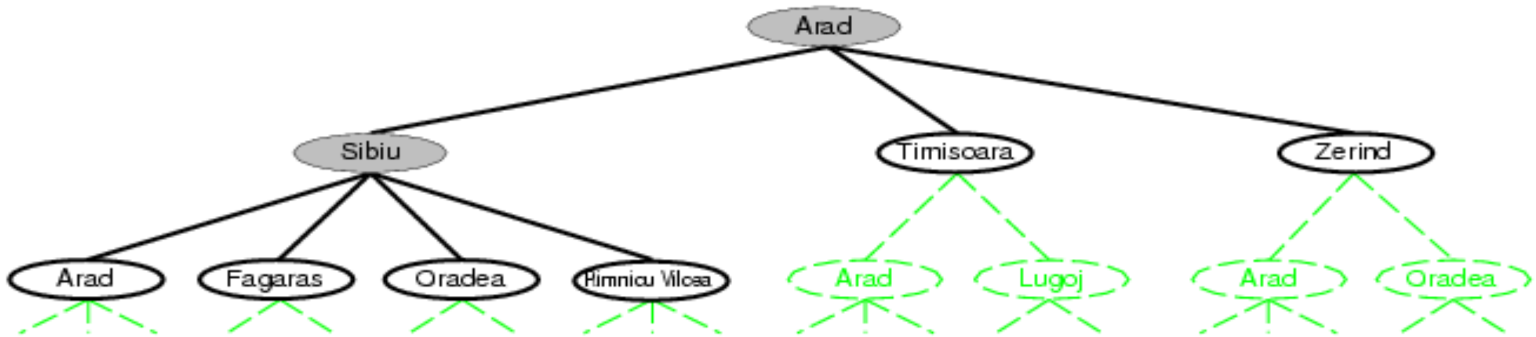


- **Redundant paths:** exist whenever there is **more than one way** to get from one state to another

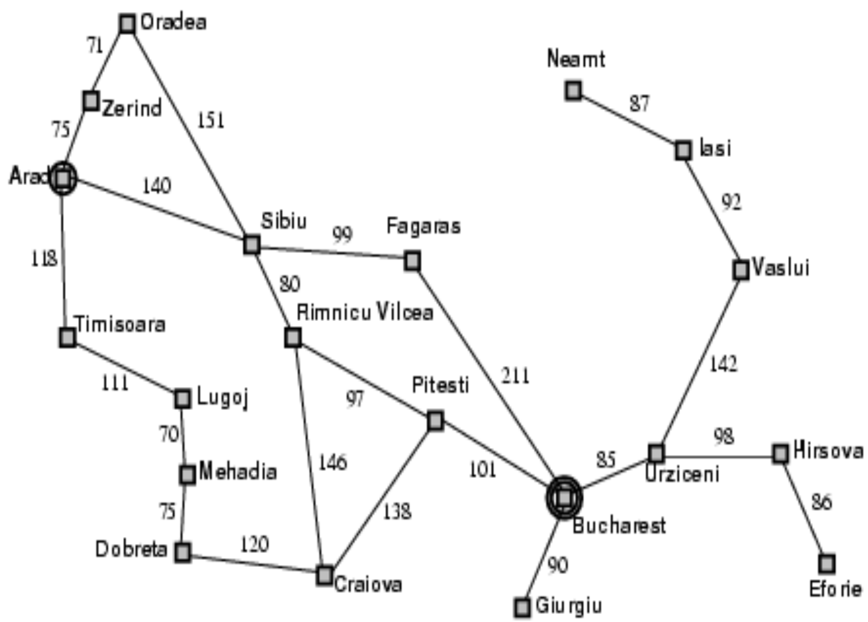
- **Example:** Consider the path
  - **Arad–Sibiu (140)**
  - **Arad–Zerind–Oradea–Sibiu (297)**
  - the **second** path is **redundant**—it's just a <u>worse</u> way to get to the same state



Tree search example



To avoid exploring **redundant paths** **TREE-SEARCH** algorithm is **augmented** with **explored set** that remembers **every expanded node**



Graph search algorithm

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

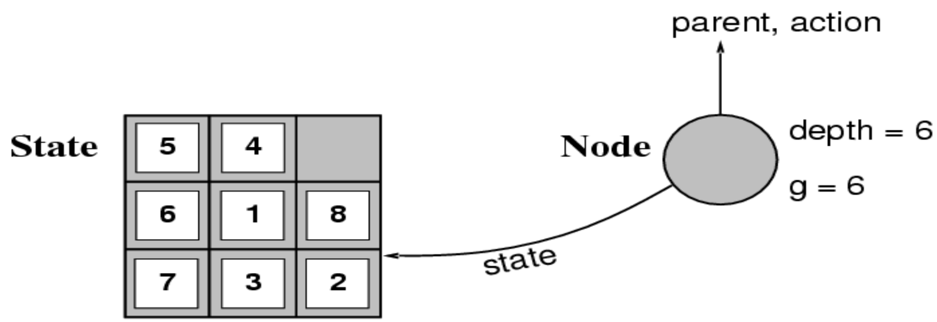
expand the chosen node, adding the resulting nodes to the frontier

only if not in the frontier or explored set

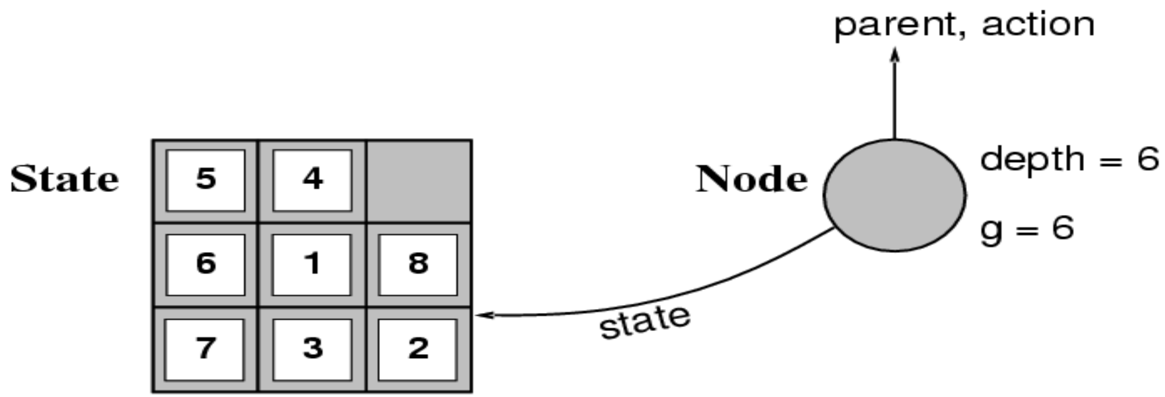
Infrastructure for search algorithms

- **Search algorithms** require a **data structure** to keep track of the **search tree**

- **For each node n** of the tree, we have a structure with:
  - 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 <u>initial state</u> to <u>the node</u>,**
    as indicated by the parent pointers



Infrastructure for search algorithms



- A state corresponds to a configuration of the world
- A node is a **data structure** used to represent the **search tree**

Frontier: The set of **all leaf nodes** available **for expansion** at any given point

Infrastructure for search algorithms

- **Frontier** needs to be stored in such a way that the **search algorithm** can easily choose the **next node** to expand
  - The appropriate data structure for this is a **queue**

Operations on a queue:

- 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

Infrastructure for search algorithms

- Queues are characterized by the *order* in which they **store** the inserted nodes

  Three common variants:
  - **FIFO queue**
    - which pops the *oldest* element of the queue
  - **LIFO queue**
    - which pops the *newest* element of the queue
  - **Priority queue**
    - which pops the element of the queue with the *highest priority* according to some ordering function

Search strategies

- A search 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**
  - space complexity: **maximum number of nodes in memory**
  - optimality: does it **always find a least-cost solution**?

Search strategies

- **Time** and **space complexity** are measured in terms of
  - ***b***: **branching factor** of the search tree
    (i.e., maximum number of successors of any node)
  - ***d***: **depth** of the **least-cost solution**
  - ***m*:** the **maximum depth** of the state space

Uninformed search strategies

- Uninformed strategies use only the information available in the problem definition
  - generate successors
  - distinguish goal

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search