CS 4341
Introduction to Artificial Intelligence

Lecture 4 and 5: Solving Problems by Searching

By

Ben C.K. Ngan

Worcester Polytechnic Institute

Problem-Solving Agents

- **Problem-solving agents** and **Planning agents** are Goal-based agents

- A **problem-solving agent** performs a sequence of actions that **form a path** to a goal state.

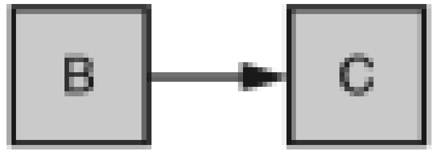
  - ❑ The agent uses **atomic** representations for states, e.g., solutions, locations, status, etc.
  - ❑ The computational process that the agent undertakes is called **search**.
  - ❑ Formalized as a search through possible solutions.

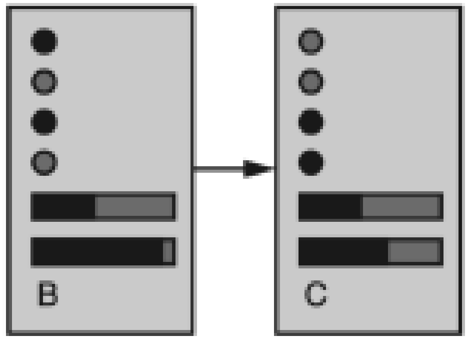
- The agents that use **factored** or **structured** representations of states are called **planning agents** that we may not cover.

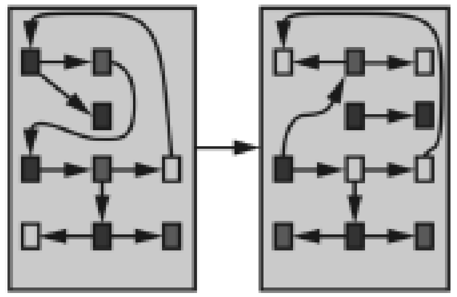
- Search Algorithms of **Problem-solving Agents** can be:

  - ❑ **Uninformed Algorithms**: The agent is not provided with any prior information about the problems rather than its definition to look for solutions. The agent checks all the plausible paths and pick the best one. **No such goal estimate is available**.

  - ❑ **Informed (Heuristic) Algorithms**: The agent is provided with some guidance and prior information to look for solutions and **estimate how far it is from the goal.**







Problem-Solving Agents

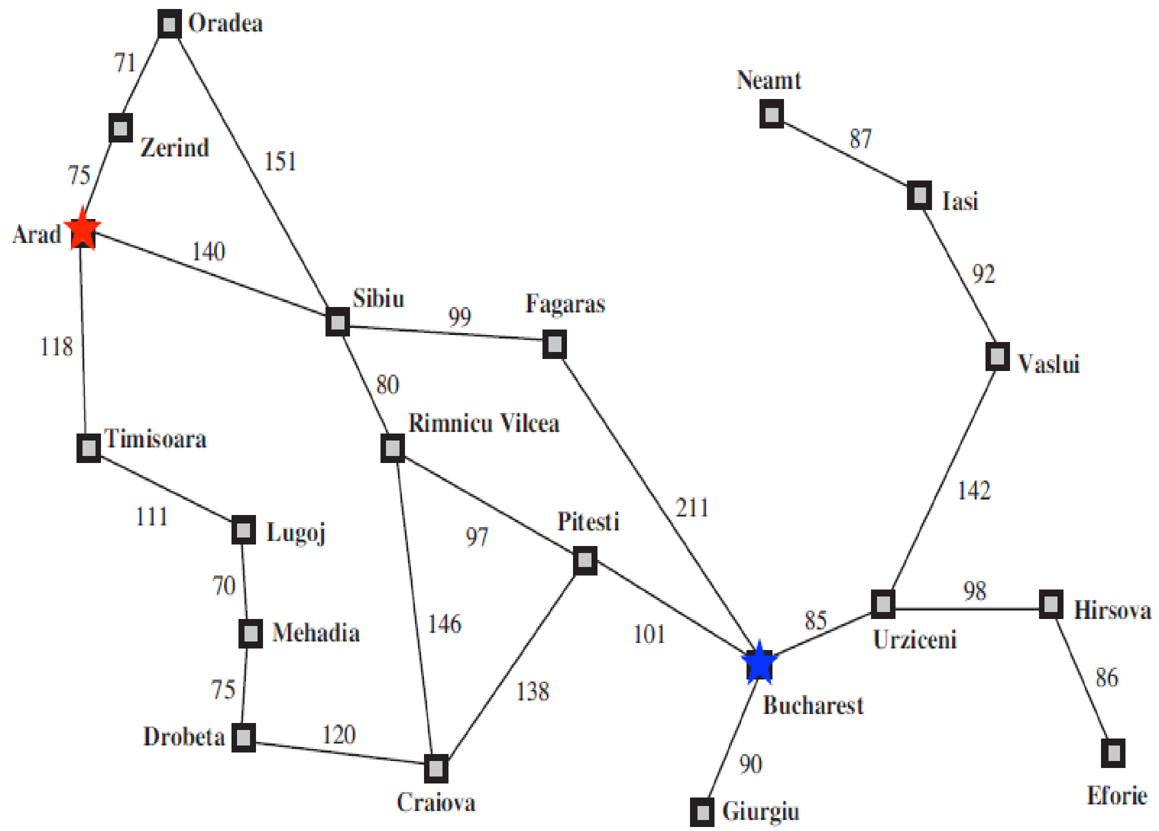
- With the known environment information with access to information about the world, such as the map, **the agent can follow this four-phase problem-solving process**:
  - ❑ Goal Formulation: Adopt the objective
  - ❑ * Problem Formulation: Devise a description of the states and actions necessary to reach the goal
  - ❑ Search Solution: Simulate all the possible sequences of actions in its model and search them until a sequence of actions, i.e., a solution, that reaches the goal is found.
  - ❑ Solution Execution: Execute the actions in the solution, one at a time.
- **Example: Travelling in Romania**
  - ❑ A touring vacation in Romania
  - ❑ Currently in the city of **Arad (A)**, Romania
  - ❑ Non-refundable ticket to fly out from Bucharest (B) tomorrow
  - ❑ Must reach **Bucharest (B)** on time

Romania
UKRAINE
HUNGARY
Maramureş
Bucovina
Moldavia
Crişana
Transylvania
Banat
Muntenia
Oltenia
Dobrogea
MOLDOVA
SERRBIA
BULGARIA
Black Sea
Oradea
Cluj-Napoca
Timişoara
Sibiu
Braşov
Iaşi
Bacau
Craiova
Piteşti
Ploieşti
Bucharest
Constanta
Galati
Tulcea
Belgrade
Szeged
100 km
100 miles

Problem-Solving Agents

- **Example: Travelling in Romania**
  - ❑ A touring vacation in Romania
  - ❑ Currently in the city of **Arad (A)**, Romania
  - ❑ Non-refundable ticket to fly out from **Bucharest (B)** tomorrow
  - ❑ Must reach Bucharest on time

- **Four-phase Problem-solving Process**
  - ❑ **Goal Formulation**: To be in Bucharest (B) on time
  - ❑ * **Problem Formulation (see next two slides)**:
    - ○ **States:** Romanian Cities
    - ○ **Actions:** Drive on the roads among the cities
  - ❑ **Search Solution**: Find the driving **route** among the cities, e.g., Arad → Sibiu →Rimnicu Vilcea → Pitesti → Bucharest, to reach Bucharest on time.
  - ❑ **Solution Execution**: Execute this driving route



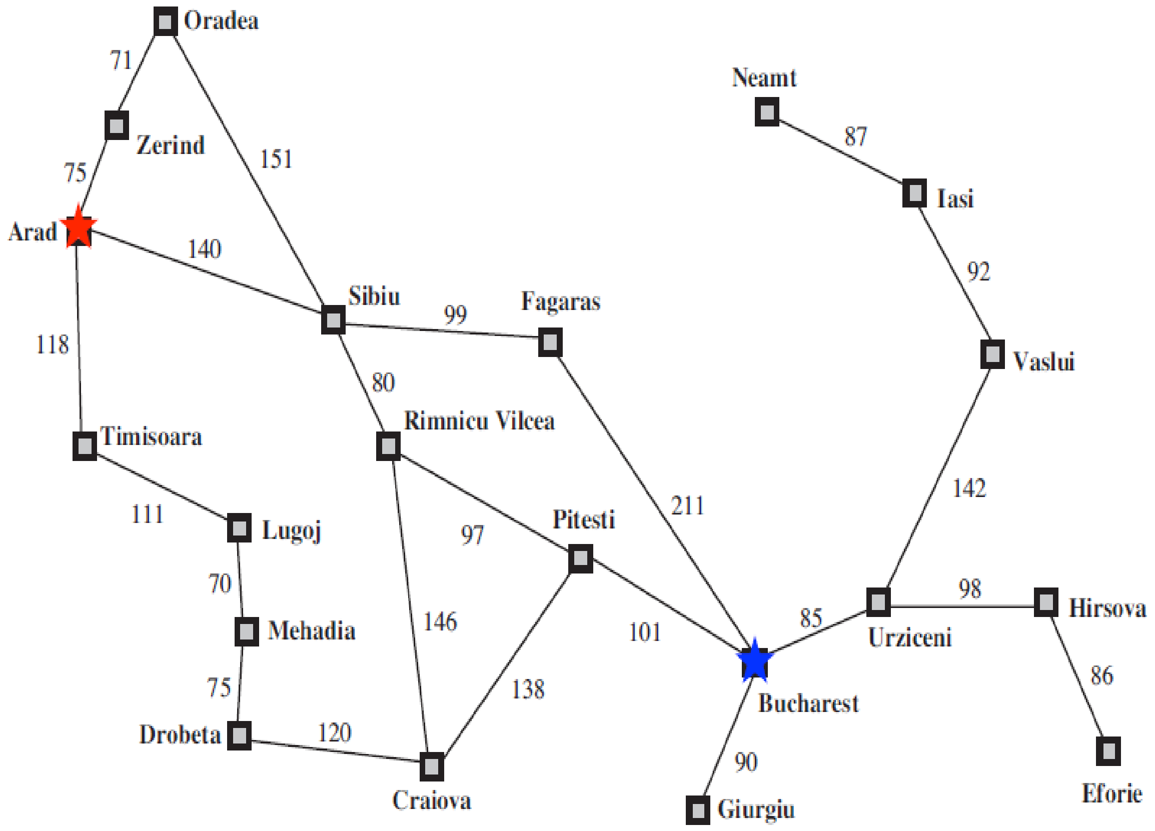
Problem-Solving Agents

A **Search Problem Formulation** can be defined formally by ***six*** components:

1. A **set of possible states** of an agent that the environment can be in, i.e., ***the state space represented as a graph***, e.g., undirected graph
   - Any city that the agent will be in is one of the possible states.
   - Solving a problem amounts to searching through different states, called state-space ***G = (V, E)***, where V is a set of vertices, and E is a set of edges between them.

2. The **initial state** that the agent starts in, for example, ***Arad (A)***

3. A **set of one or more goal states**. Sometimes,
   - There is one goal state, e.g., ***Bucharest (B)***, in our example.
   - There is a small set of alternative goal states.



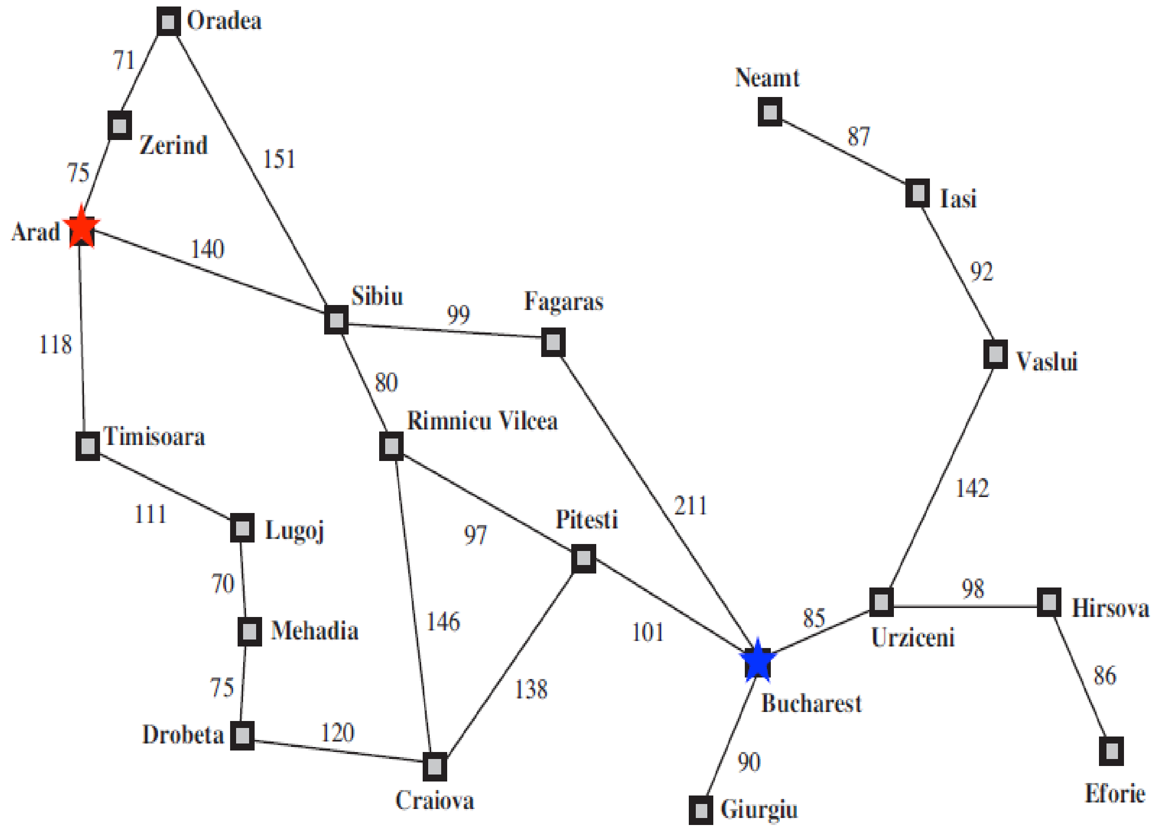
Problem-Solving Agents

A **Search Problem Formulation** can be defined formally by six components:

4. The **possible actions** at ***A*** state, i.e., the current state, available to the agent.
   - Given a state ***s***, ***Actions(s)*** returns a finite set of actions executable from ***s***.
   - *s* = *Arad*, Actions(*Arad*) = {*ToSibiu*, *ToTimisoara*, *ToZerind*}

5. A **transition model** describes what each action does to go to the next state from a state.
   - Given a state ***s*** and action ***a***, **Results(*s*, *a*)** returns the next state after doing the action ***a*** in state ***s***.
   - Results(*Arad*, *ToZerind*) = *Zerind*

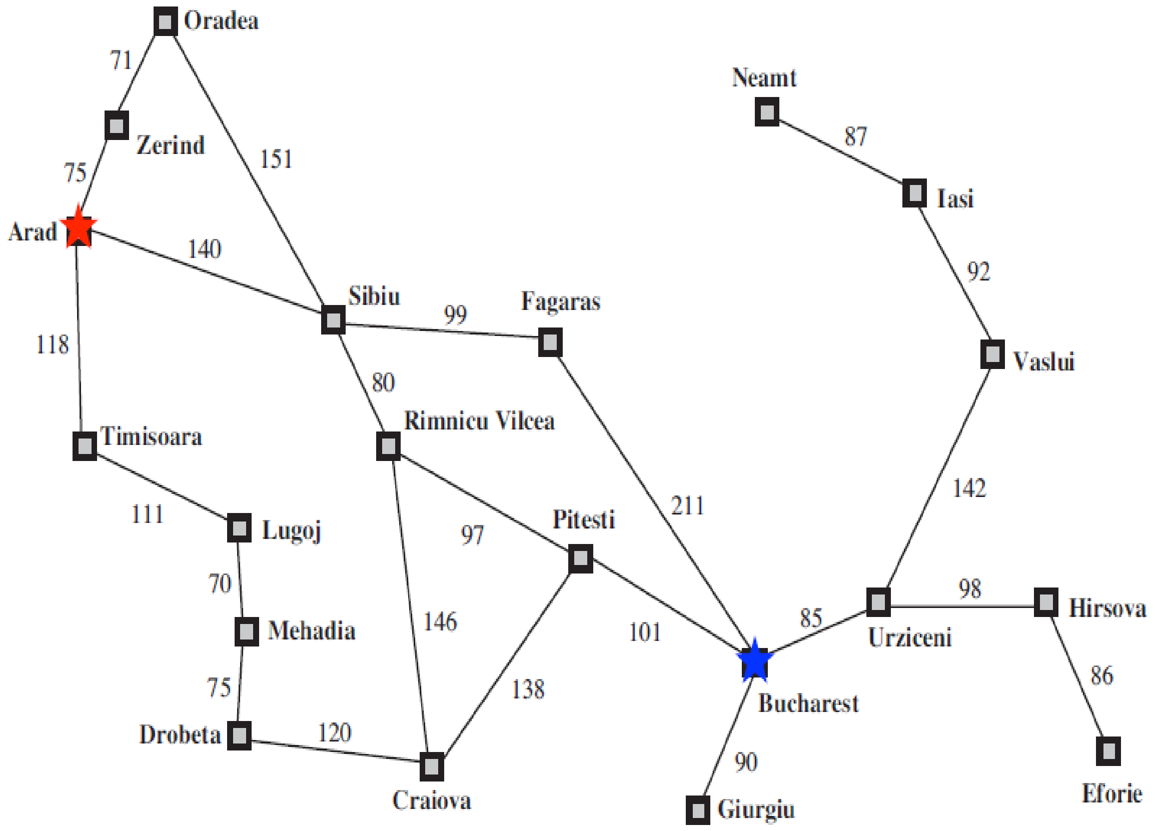
6. A **action cost function**, i.e., ***ACTION-COST(s, a, s')***, gives the numeric cost of applying action *a* in state ***s*** to reach ***s'***.
   - *ACTION-COST*(*Arad*, *ToSibiu*, *Sibiu*) = 140



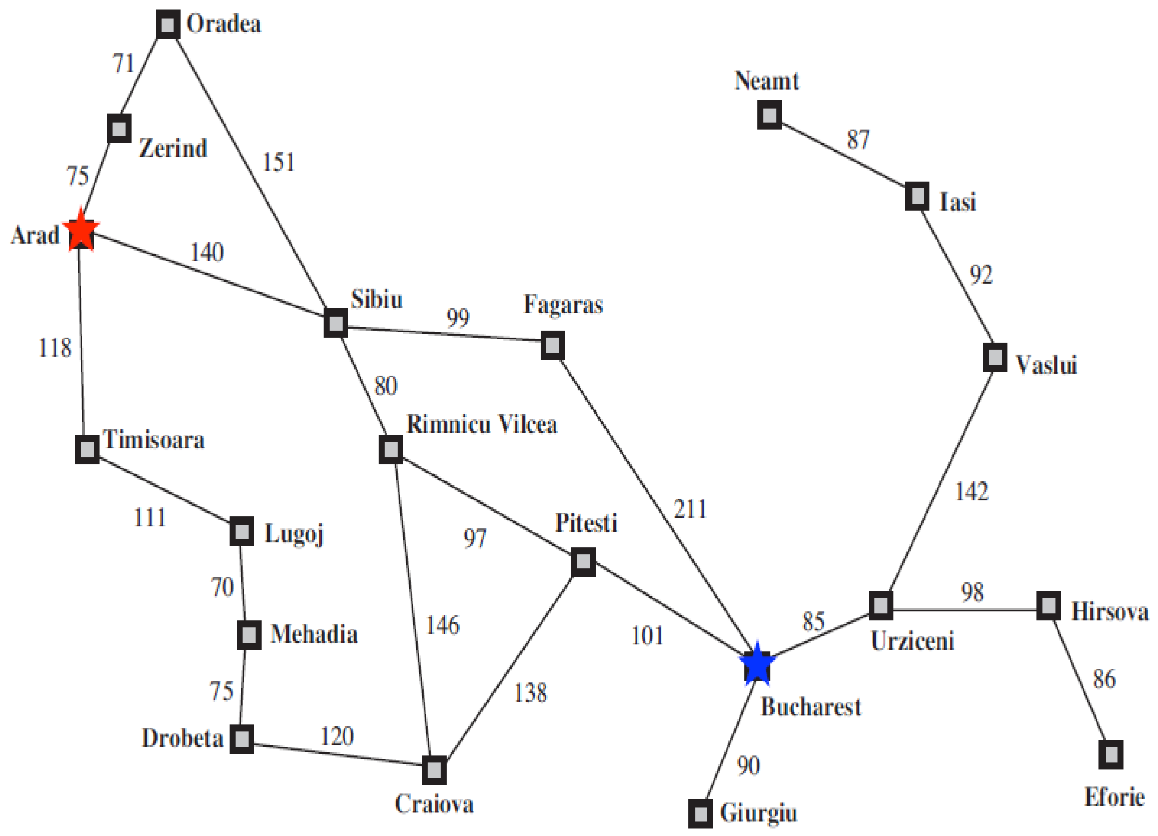
Problem-Solving Agents

The Summary of a Search Problem Formulation:

- The **state space** can be represented as a **graph** in which the vertices are states and the directed edges between them are actions.
- A sequence of actions **forms a path**.
- A **solution** is a path from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.
- The formulation of a problem is a **mathematical model**.



Problem-Solving Agents



```
romania_map = UndirectedGraph(dict(
    Arad=dict(Zerind=75, Sibiu=140, Timisoara=118),
    Bucharest=dict(Urziceni=85, Pitesti=101, Giurgiu=90, Fagaras=211),
    Craiova=dict(Drobeta=120, Rimnicu=146, Pitesti=138),
    Drobeta=dict(Mehadia=75),
    Eforie=dict(Hirsova=86),
    Fagaras=dict(Sibiu=99),
    Hirsova=dict(Urziceni=98),
    Iasi=dict(Vaslui=92, Neamt=87),
    Lugoj=dict(Timisoara=111, Mehadia=70),
    Oradea=dict(Zerind=71, Sibiu=151),
    Pitesti=dict(Rimnicu=97),
    Rimnicu=dict(Sibiu=80),
    Urziceni=dict(Vaslui=142)))

romania_map.locations = dict(
    Arad=(91, 492), Bucharest=(400, 327), Craiova=(253, 288),
    Drobeta=(165, 299), Eforie=(562, 293), Fagaras=(305, 449),
    Giurgiu=(375, 270), Hirsova=(534, 350), Iasi=(473, 506),
    Lugoj=(165, 379), Mehadia=(168, 339), Neamt=(406, 537),
    Oradea=(131, 571), Pitesti=(320, 368), Rimnicu=(233, 410),
    Sibiu=(207, 457), Timisoara=(94, 410), Urziceni=(456, 350),
    Vaslui=(509, 444), Zerind=(108, 531))
```

https://github.com/aimacode/aima-python/blob/master/search.py

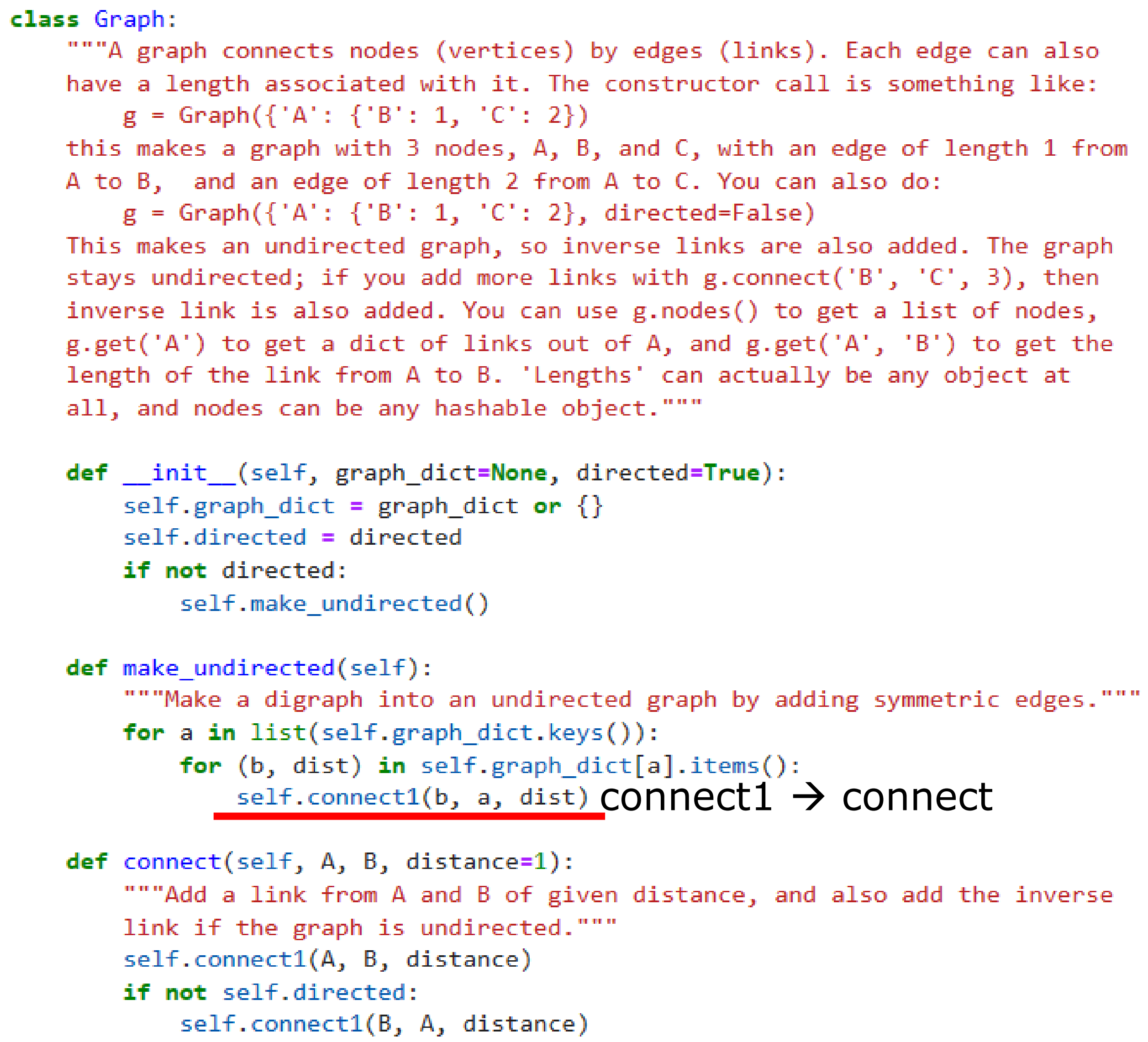
```
def UndirectedGraph(graph_dict=None):
    """Build a Graph where every edge (including future ones) goes both ways."""
    return Graph(graph_dict=graph_dict, directed=False)
```

Class Exercise 1

Review the **Graph** class in search.py, https://github.com/aimacode/aima-python/blob/master/search.py, and do the following with your teammates/classmates

- Which function in the class is used to create the graph of Romania map?
- There are two functions, connect() and connect1(), defined in the Graph class. What is the difference between them in terms of their purposes?
- Is there any issue in the Graph class that we use to address this Romania Traveling problem? If yes, what is it? How can we fix it?

Problem-Solving Agents

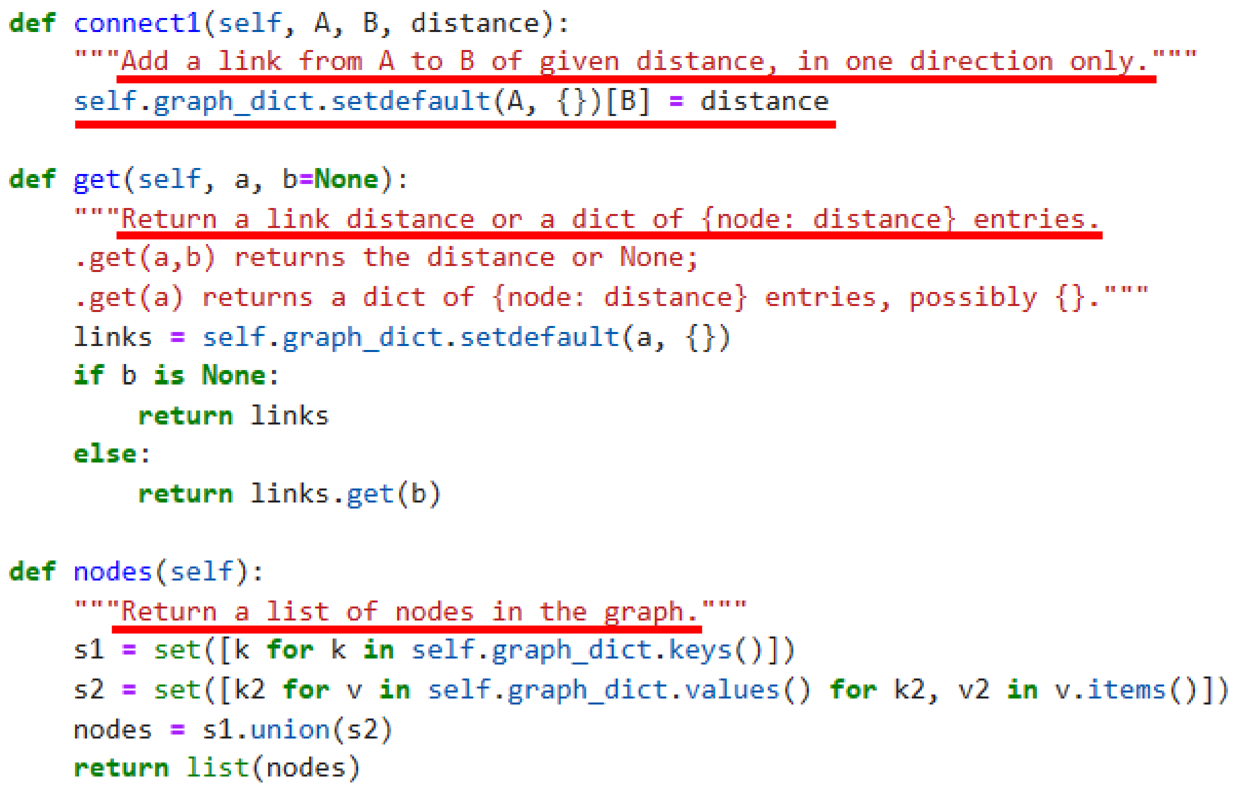


```python
class Graph:
    """A graph connects nodes (vertices) by edges (links). Each edge can also
    have a length associated with it. The constructor call is something like:
        g = Graph({'A': {'B': 1, 'C': 2})
    this makes a graph with 3 nodes, A, B, and C, with an edge of length 1 from
    A to B,  and an edge of length 2 from A to C. You can also do:
        g = Graph({'A': {'B': 1, 'C': 2}, directed=False)
    This makes an undirected graph, so inverse links are also added. The graph
    stays undirected; if you add more links with g.connect('B', 'C', 3), then
    inverse link is also added. You can use g.nodes() to get a list of nodes,
    g.get('A') to get a dict of links out of A, and g.get('A', 'B') to get the
    length of the link from A to B. 'Lengths' can actually be any object at
    all, and nodes can be any hashable object."""

    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()

    def make_undirected(self):
        """Make a digraph into an undirected graph by adding symmetric edges."""
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.connect1(b, a, dist)

    def connect(self, A, B, distance=1):
        """Add a link from A and B of given distance, and also add the inverse
        link if the graph is undirected."""
        self.connect1(A, B, distance)
        if not self.directed:
            self.connect1(B, A, distance)
```

connect1 → connect

https://github.com/aimacode/aima-python/blob/master/search.py

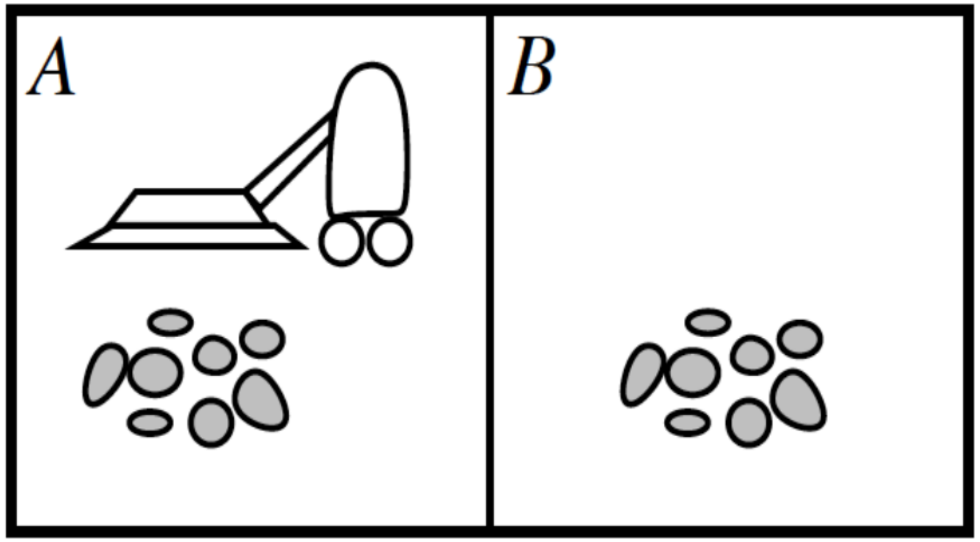
This method returns the key value available in the dictionary and if given key is not available, i.e., A, then it will return the provided default value, i.e., {}, the empty set.



```python
    def connect1(self, A, B, distance):
        """Add a link from A to B of given distance, in one direction only."""
        self.graph_dict.setdefault(A, {})[B] = distance

    def get(self, a, b=None):
        """Return a link distance or a dict of {node: distance} entries.
        .get(a,b) returns the distance or None;
        .get(a) returns a dict of {node: distance} entries, possibly {}."""
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)

    def nodes(self):
        """Return a list of nodes in the graph."""
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items()])
        nodes = s1.union(s2)
        return list(nodes)
```

A Search Problem Example: Vacuum World

- A grid world problem is **a two-dimensional rectangular array** of square cells in which agents can move from cell to cell.
  - ❑ The agent can move to any obstacle-free adjacent cell – horizontally or vertically or diagonally.
  - ❑ A cell can contain objects.
    - The agent can pick up, push, or act upon
    - If there is a wall or any impassible obstacle in a cell, it prevents an agent from moving into that cell.
  - ❑ The **VACCUM WORLD** is a grid world problem.



Percepts: location and contents, e.g., $[A, Dirty]$

Actions: $Left$, $Right$, $Suck$, $NoOp$

A Search Problem Example: Vacuum World

- **Possible States**:
  - ❑ A Vacuum Agent (VA) is either in one of the two cells
  - ❑ Each Cell (C) either contains DIRT or CLEAN
  - ❑ **1 Cell** → 2 possibilities = 1 * 2^1 = 2 states
    - o VA with D
    - o VA without D
  - ❑ **2 Cells** → 8 possibilities = 2 * 2^2 = 8 states
    - o VA with D
    - o VA without D
    - o D only
    - o None of Both
  - ❑ **n Cells** = n * 2^n states

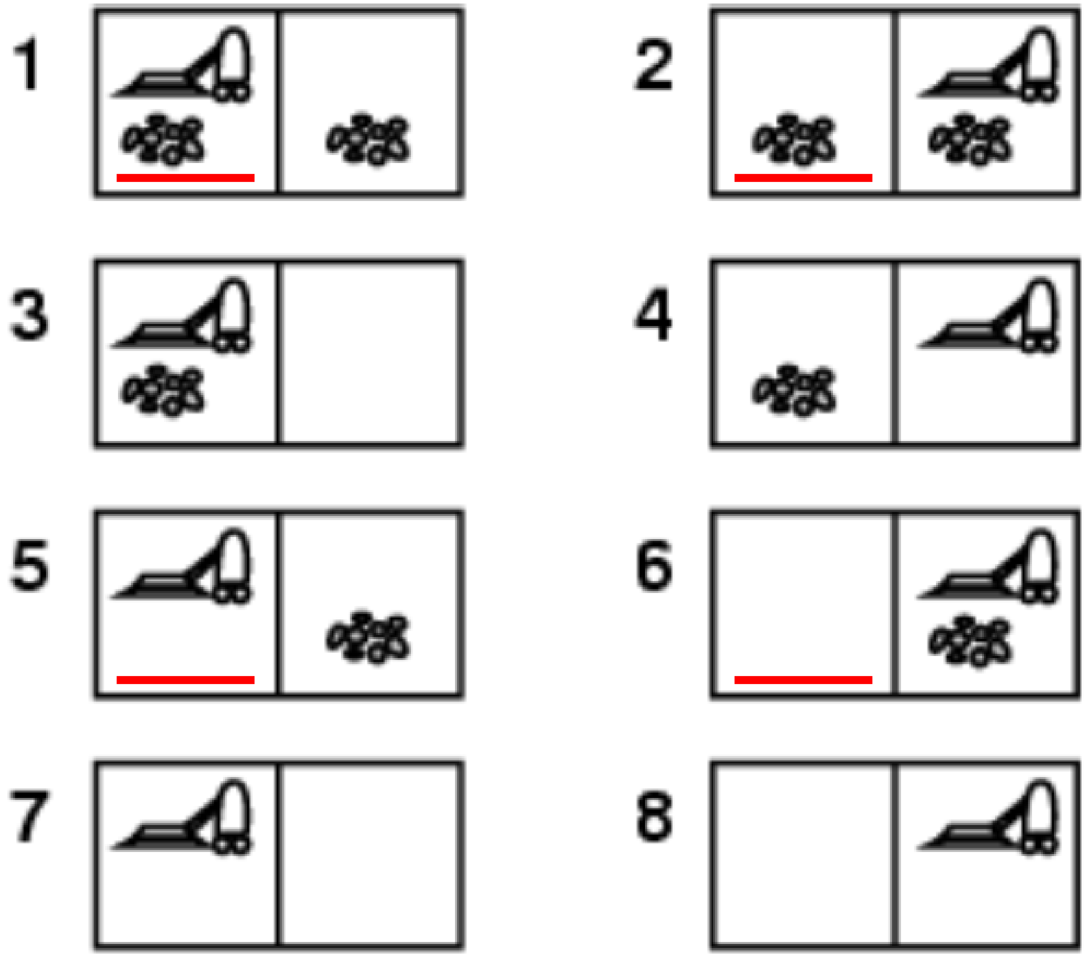
- **Initial State**: Any possible state can be the initial state.

- **Goal States**: The states in which every cell is clean, i.e., S_7 or S_8.

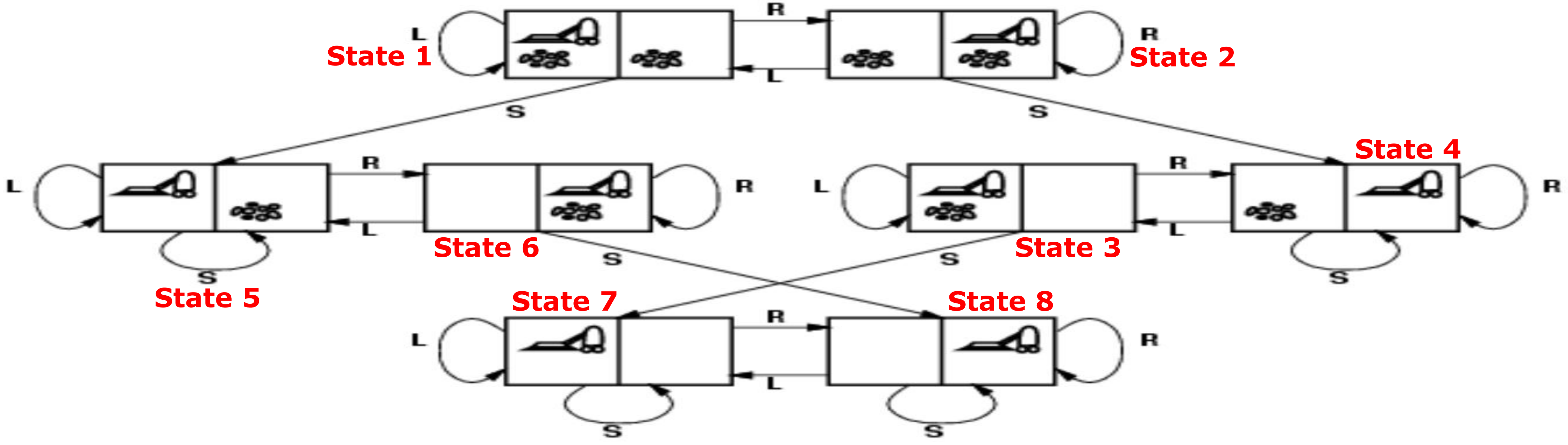
- **Possible Actions**: {Left, Right, Suck, NoOp}

- **Transition Models**: Results(S_1, *Suck*) = S_5 & Results(S_5, *Right*) = S_6 & Results(S_6, *Suck*) = S_8 & Results(S_8, *NoOp*) = $\boldsymbol{S_8}$ ***or*** Results(S_8, *NoOp*) = $\boldsymbol{S_7}$

- **Action Cost**: Each action costs 1, for example, from one state to the next state



A Search Problem Example: Vacuum World



Eight possible states of the vacuum world. <u>Each state</u> is represented as:

	"State of the left room"	**"State of the right room"**	**"Room in which the agent is present"**
1 - DDL	Dirty	Dirty	Left
2 - DDR	Dirty	Dirty	Right
3 - DCL	Dirty	Clean	Left
4 - DCR	Dirty	Clean	Right
5 - CDL	Clean	Dirty	Left
6 - CDR	Clean	Dirty	Right
7 - CCL	Clean	Clean	Left
8 - CCR	Clean	Clean	Right

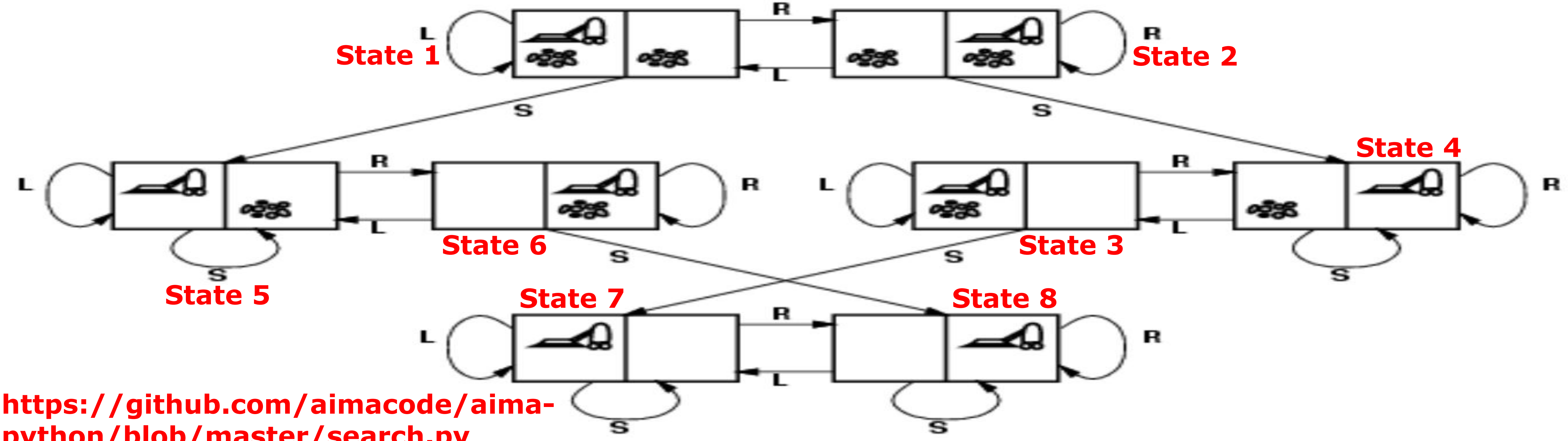
Class Exercise 2

- Review the vacuum world graph (see below) in search.py, https://github.com/aimacode/aima-python/blob/master/search.py, and do the following with your teammates/classmates

- Is there any error in the vacuum world graph below? If yes, what is it? How can we fix it?

```
vacuum_world = Graph(dict(
    State_1=dict(Suck=['State_7', 'State_5'], Right=['State_2']),
    State_2=dict(Suck=['State_8', 'State_4'], Left=['State_2']),
    State_3=dict(Suck=['State_7'], Right=['State_4']),
    State_4=dict(Suck=['State_4', 'State_2'], Left=['State_3']),
    State_5=dict(Suck=['State_5', 'State_1'], Right=['State_6']),
    State_6=dict(Suck=['State_8'], Left=['State_5']),
    State_7=dict(Suck=['State_7', 'State_3'], Right=['State_8']),
    State_8=dict(Suck=['State_8', 'State_6'], Left=['State_7'])
))
```

A Search Problem Example: Vacuum World



https://github.com/aimacode/aima-python/blob/master/search.py

```
vacuum_world = Graph(dict(
    State_1=dict(Suck=[[illegible]ate_7[illegible]'State_5'], Right=['State_2']),
    State_2=dict(Suck=[illegible]e[illegible]tate_4'], Left=['State_2']),
    State_3=dict(Suck=['[illegible]Right=['State_4']),
    State_4=dict(Suck=['St[illegible], 'State_2'], Left=['State_3']),
    State_5=dict(Suck=['[illegible]'State_1'], Right=['State_6']),
    State_6=dict(Suck=[illegible]e[illegible]eft=['State_5']),
    State_7=dict(Suck=[[illegible]ate_7[illegible]State_3'], Right=['State_8']),
    State_8=dict(Suck=['State_8', 'State_6'], Left=['State_7'])
))
```

```
vacuum_world = Graph(dict(
    State_1=dict(Suck=['State_5'], Left=['State_1'], Right=['State_2']),
    State_2=dict(Suck=['State_4'], Left=['State_1'], Right=['State_2']),
    State_3=dict(Suck=['State_7'], Left=['State_3'], Right=['State_4']),
    State_4=dict(Suck=['State_4'], Left=['State_3'], Right=['State_4']),
    State_5=dict(Suck=['State_5'], Left=['State_5'], Right=['State_6']),
    State_6=dict(Suck=['State_8'], Left=['State_5'], Right=['State_6']),
    State_7=dict(Suck=['State_7'], Left=['State_7'], Right=['State_8']),
    State_8=dict(Suck=['State_8'], Left=['State_7'], Right=['State_8'])
))
```

titute

A Search Problem Example: 8-Puzzle Game

- **Possible States**:
  - ❑ 9! is the total number of possible states, i.e., 9 x 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1
  - ❑ 9!/2 is the total number of solvable possible states.

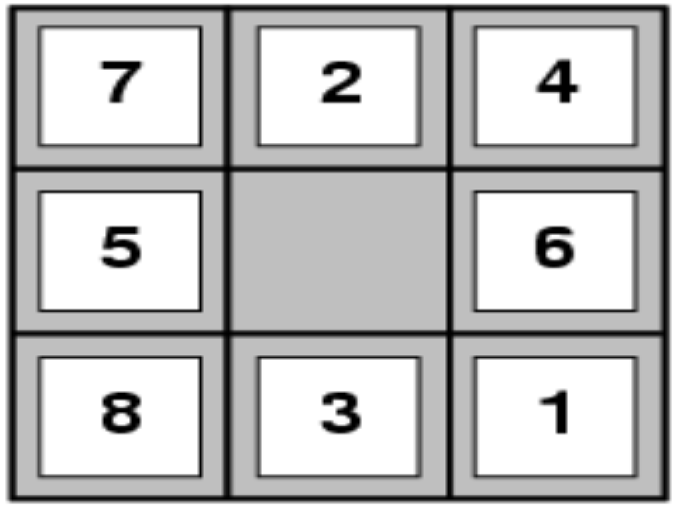
- **Initial State**: Any one of the possible 9! states can be the initial state.

- **Goal States**: A state with the numbers in order that can be reached from 9!/2 solvable possible states.

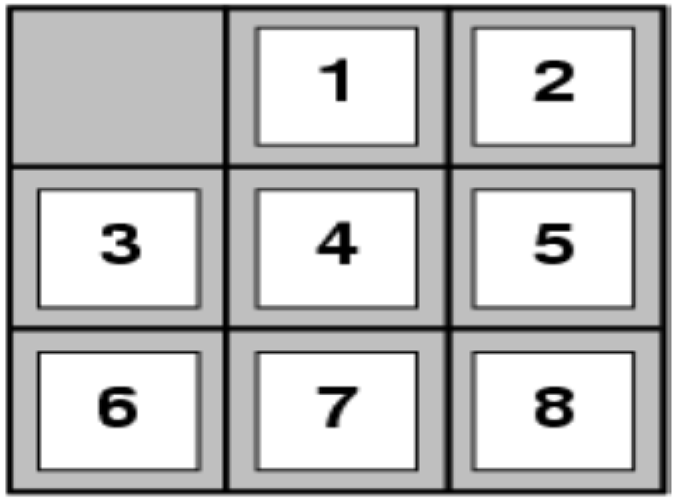
- **Possible Actions**: {Left, Right, Up, Down}

- **Transition Model**: Results(S_1, *Left*) = S_2, where S_2 is the next state with the 5 and **the blank switched**.

- **Action Cost**: Each action costs 1, for example, from one state to the next state



Start State



Goal State

A Search Problem Example: 8-Puzzle Game

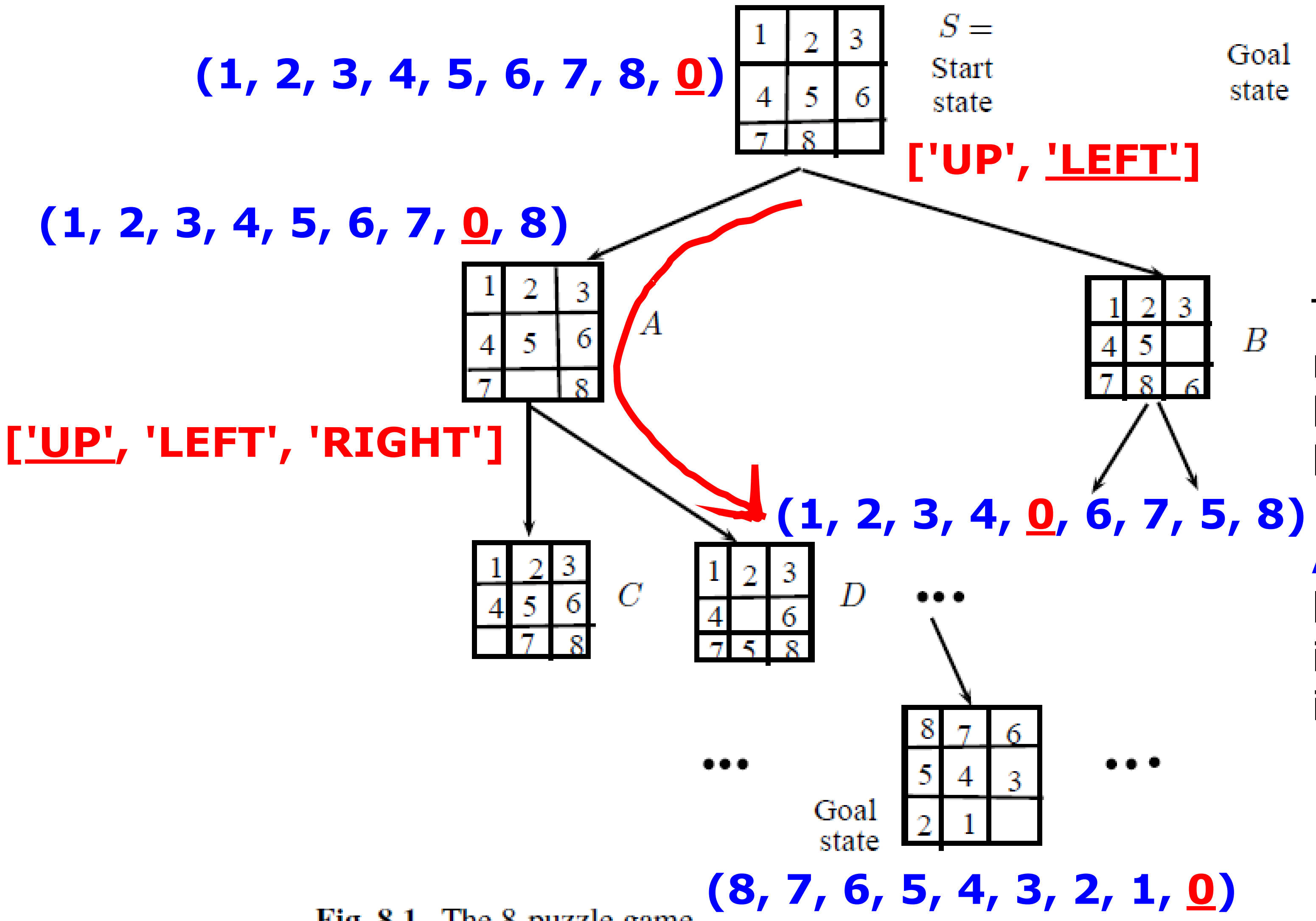
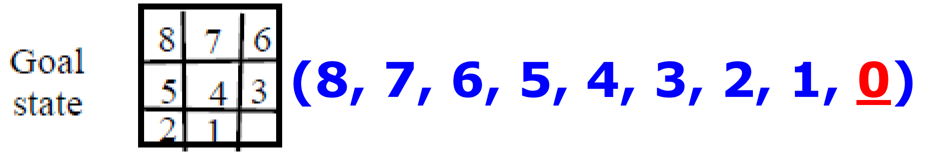


Fig. 8.1 The 8-puzzle game



```
class EightPuzzle(Problem)
```

The problem of sliding tiles numbered from 1 to 8 on a 3x3 board, where one of the squares is a blank.

A state is represented as **a tuple** of length 9, where an element at index i represents the tile number at index i (**0** if it's an empty square)

A Search Problem Example: 8-Puzzle Game

https://github.com/aimacode/aima-python/blob/master/search.py

```python
class Problem:
    """The abstract class for a formal problem. You should subclass
    this and implement the methods actions and result, and possibly
    __init__, goal_test, and path_cost. Then you will create instances
    of your subclass and solve them with the various search functions."""

    def __init__(self, initial, goal=None):
        """The constructor specifies the initial state, and possibly a goal
        state, if there is a unique goal. Your subclass's constructor can add
        other arguments."""
        self.initial = initial
        self.goal = goal

    def actions(self, state):
        """Return the actions that can be executed in the given
        state. The result would typically be a list, but if there are
        many actions, consider yielding them one at a time in an
        iterator, rather than building them all at once."""
        raise NotImplementedError

    def result(self, state, action):
        """Return the state that results from executing the given
        action in the given state. The action must be one of
        self.actions(state)."""
        raise NotImplementedError
```

```python
    def goal_test(self, state):
        """Return True if the state is a goal. The default method compares the
        state to self.goal or checks for state in self.goal if it is a
        list, as specified in the constructor. Override this method if
        checking against a single self.goal is not enough."""
        if isinstance(self.goal, list):
            return is_in(state, self.goal)
        else:
            return state == self.goal

    def path_cost(self, c, state1, action, state2):
        """Return the cost of a solution path that arrives at state2 from
        state1 via action, assuming cost c to get up to state1. If the problem
        is such that the path doesn't matter, this function will only look at
        state2. If the path does matter, it will consider c and maybe state1
        and action. The default method costs 1 for every step in the path."""
        return c + 1

    def value(self, state):
        """For optimization problems, each state has a value. Hill Climbing
        and related algorithms try to maximize this value."""
        raise NotImplementedError
```

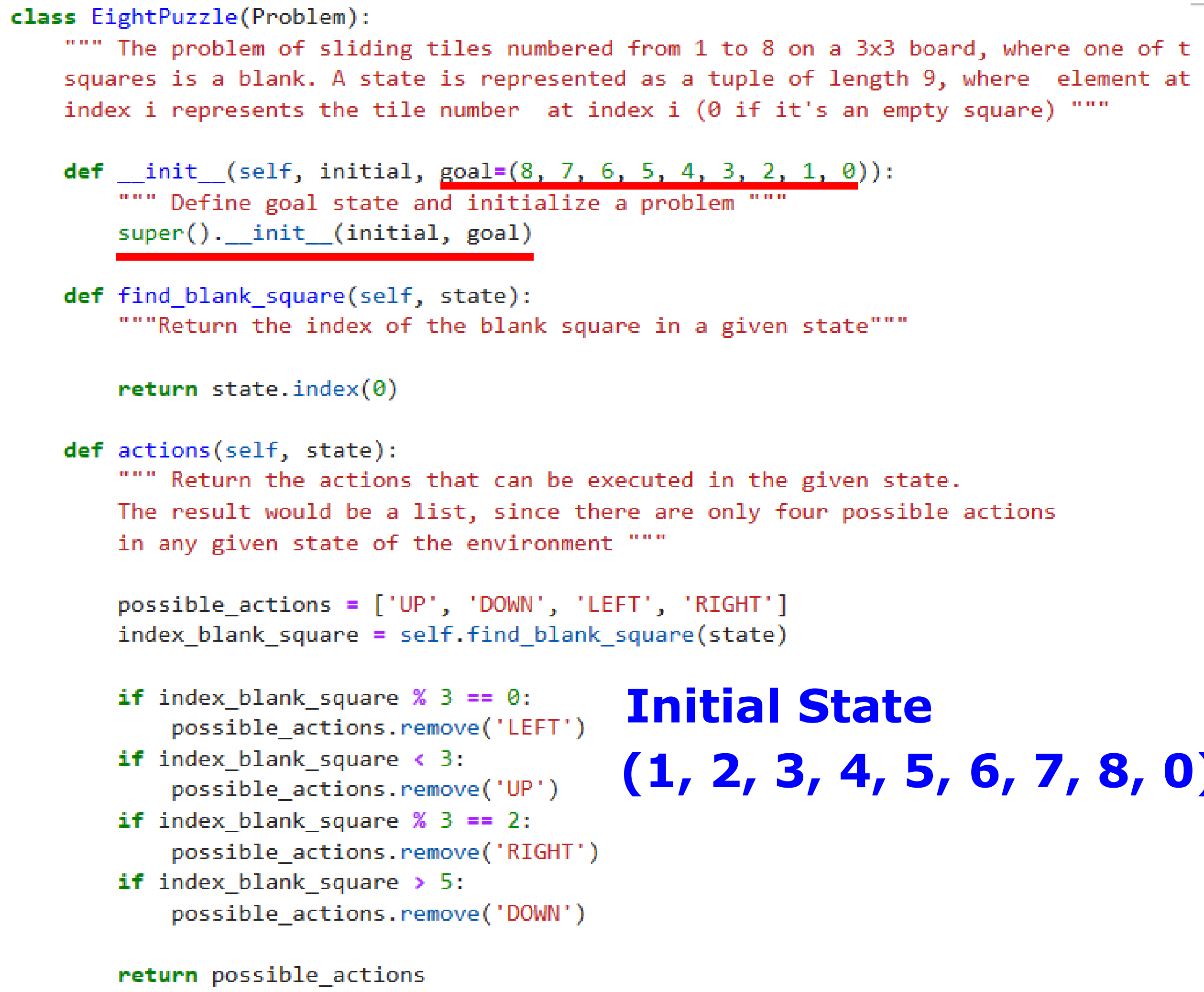
Class Exercise 3

Review the EightPuzzle(Problem) class in search.py, https://github.com/aimacode/aima-python/blob/master/search.py, and do the following with your teammates/classmates

- How to represent a state for this problem in the Python code?
- Which is the purpose of the actions(self, state) function? That is, what are those “if” statements doing?
- What is the purpose of the result(self, state, action) function? What is delta = {'UP': -3, 'DOWN': 3, 'LEFT': -1, 'RIGHT': 1}?
- In the the EightPuzzle(Problem) class, it does not use the path_cost(self, c, state1, action, state2)? Why?

A Search Problem Example: 8-Puzzle Game

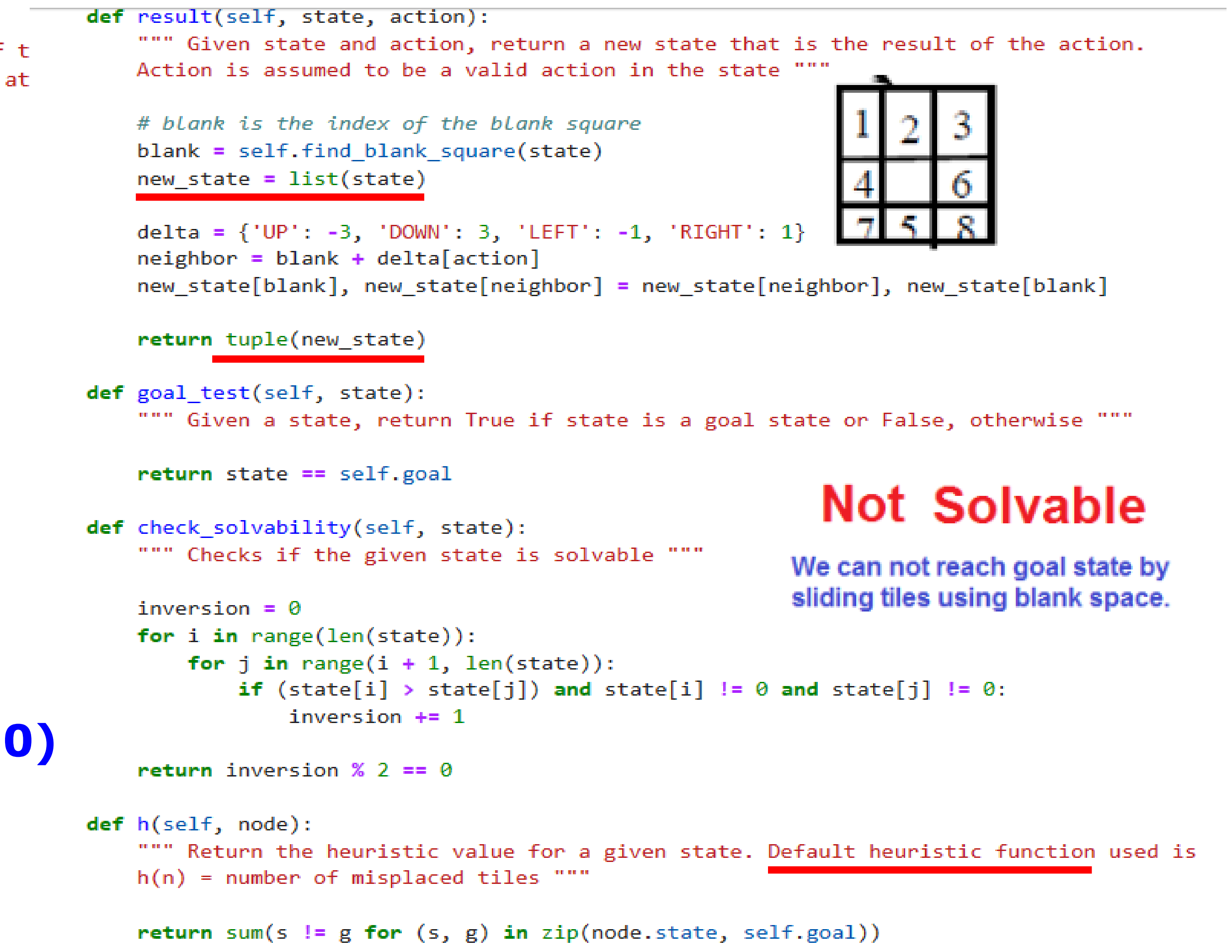
https://github.com/aimacode/aima-python/blob/master/search.py



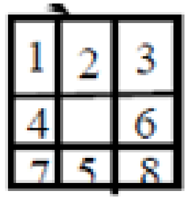
```python
class EightPuzzle(Problem):
    """ The problem of sliding tiles numbered from 1 to 8 on a 3x3 board, where one of t
    squares is a blank. A state is represented as a tuple of length 9, where  element at
    index i represents the tile number  at index i (0 if it's an empty square) """

    def __init__(self, initial, goal=(8, 7, 6, 5, 4, 3, 2, 1, 0)):
        """ Define goal state and initialize a problem """
        super().__init__(initial, goal)

    def find_blank_square(self, state):
        """Return the index of the blank square in a given state"""

        return state.index(0)

    def actions(self, state):
        """ Return the actions that can be executed in the given state.
        The result would be a list, since there are only four possible actions
        in any given state of the environment """

        possible_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
        index_blank_square = self.find_blank_square(state)

        if index_blank_square % 3 == 0:
            possible_actions.remove('LEFT')
        if index_blank_square < 3:
            possible_actions.remove('UP')
        if index_blank_square % 3 == 2:
            possible_actions.remove('RIGHT')
        if index_blank_square > 5:
            possible_actions.remove('DOWN')

        return possible_actions
```

Initial State

(1, 2, 3, 4, 5, 6, 7, 8, 0)



```python
    def result(self, state, action):
        """ Given state and action, return a new state that is the result of the action.
        Action is assumed to be a valid action in the state """

        # blank is the index of the blank square
        blank = self.find_blank_square(state)
        new_state = list(state)

        delta = {'UP': -3, 'DOWN': 3, 'LEFT': -1, 'RIGHT': 1}
        neighbor = blank + delta[action]
        new_state[blank], new_state[neighbor] = new_state[neighbor], new_state[blank]

        return tuple(new_state)

    def goal_test(self, state):
        """ Given a state, return True if state is a goal state or False, otherwise """

        return state == self.goal

    def check_solvability(self, state):
        """ Checks if the given state is solvable """

        inversion = 0
        for i in range(len(state)):
            for j in range(i + 1, len(state)):
                if (state[i] > state[j]) and state[i] != 0 and state[j] != 0:
                    inversion += 1

        return inversion % 2 == 0

    def h(self, node):
        """ Return the heuristic value for a given state. Default heuristic function used is
        h(n) = number of misplaced tiles """

        return sum(s != g for (s, g) in zip(node.state, self.goal))
```

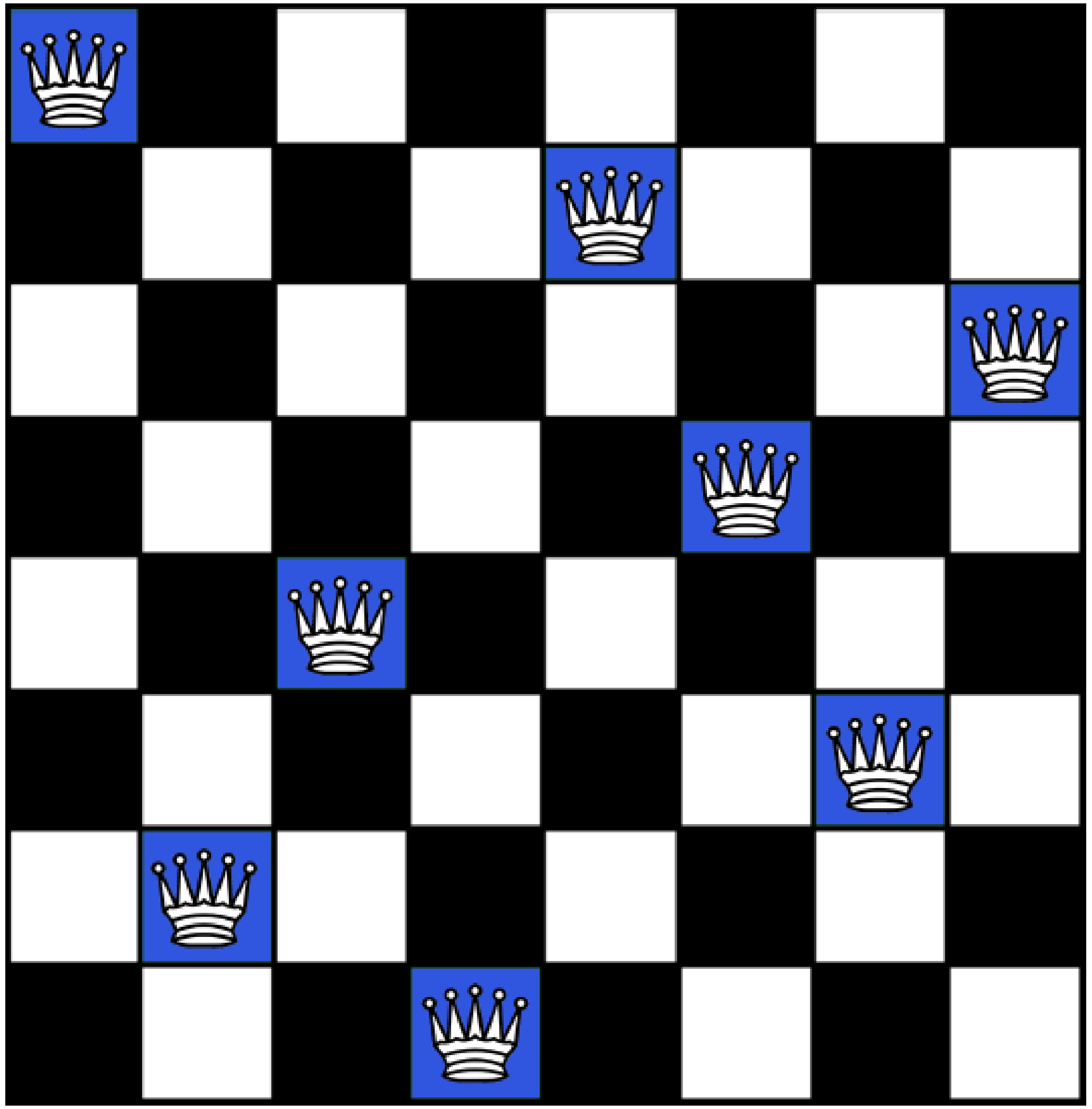


Not Solvable

We can not reach goal state by sliding tiles using blank space.

A Search Problem Example: 8-Queen Problem

- **Possible States**: 64 * 63 * 62 * … * 57 = 1.8 x 10^{14}
- **Initial State**: No queen on the board
- **Goal States**: 8 queens on the board with none attacked
- **Possible Actions**: ***ADD*** a queen to any empty square in each column from left to right
- **Transition Model**: Update the board
- **Action Cost**: The total number of attacks on the board from one state to the next state.



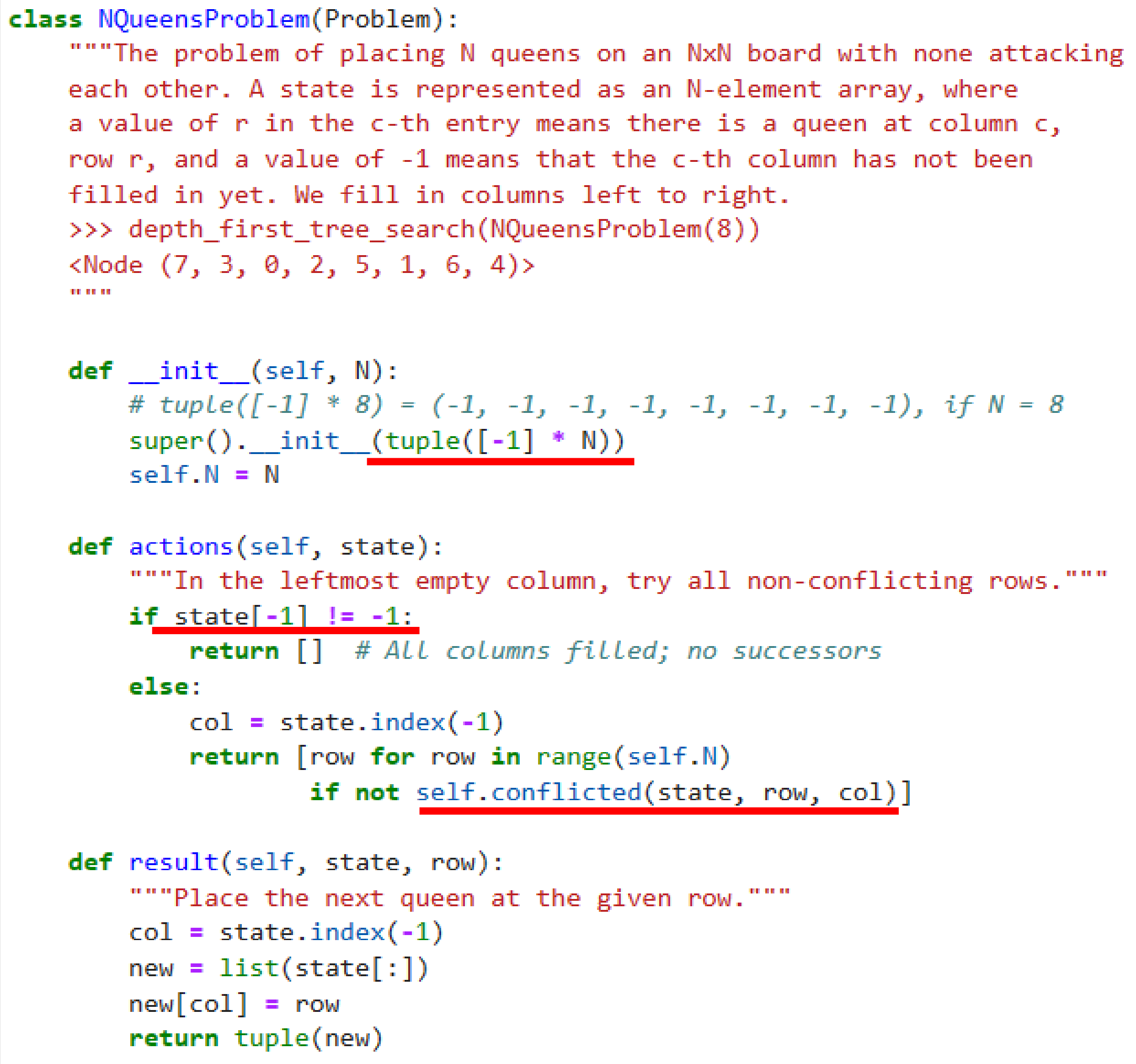
Class Exercise 4

Review the NQueensProblem(Problem) class in search.py, https://github.com/aimacode/aima-python/blob/master/search.py, and do the following with your teammates/classmates

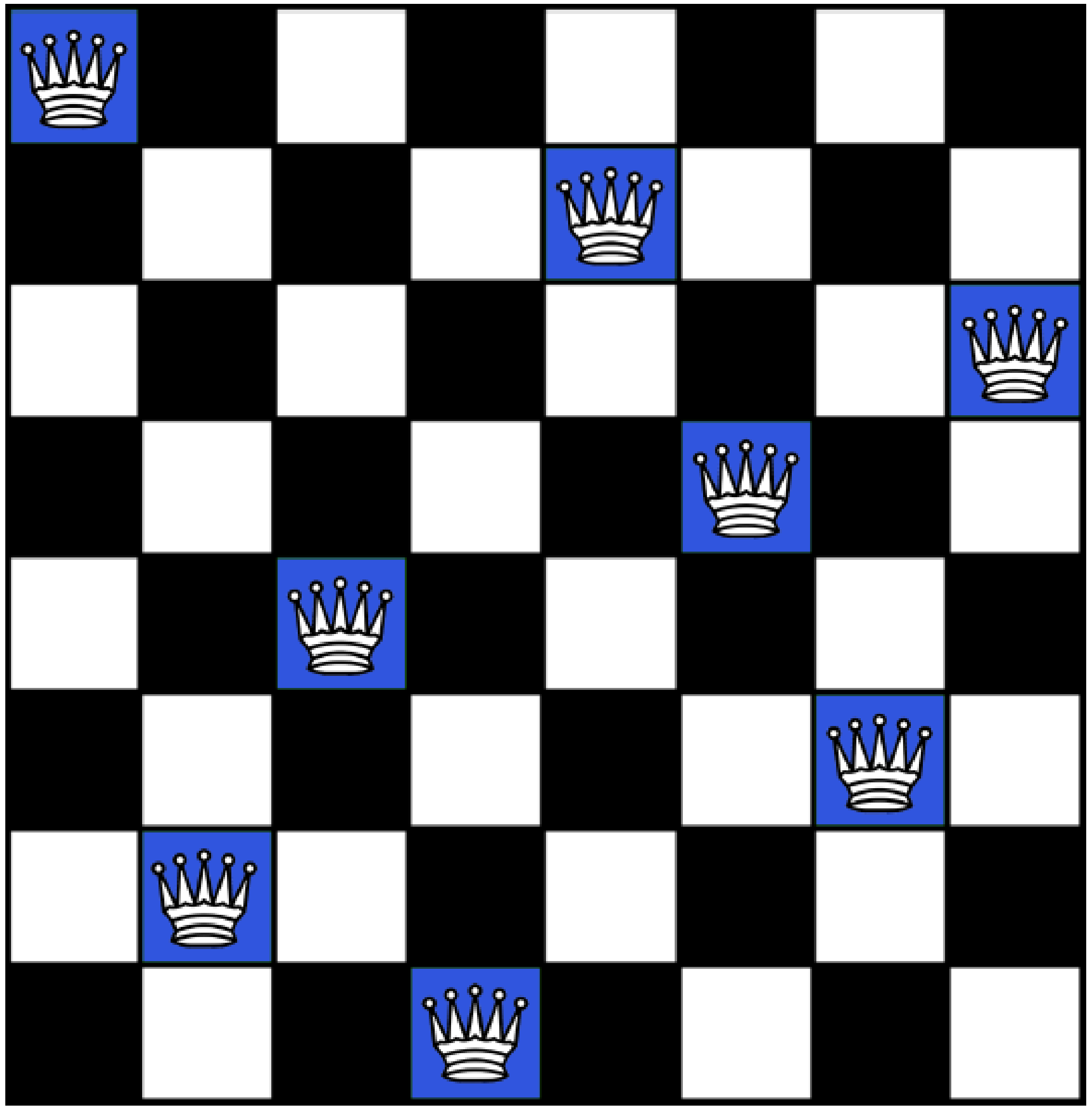
- How to represent a state for this problem in the Python code?
- What is the goal state of this problem? Do we need to define the goal state?
- What is the purpose of conflicted(self, state, row, col) and conflict(self, row1, col1, row2, col2)
- In this 8-Queen Problem class, it does not use the path_cost(self, c, state1, action, state2) as well? Why?
- How to count the total number of queen attacks at a state?

https://github.com/aimacode/aima-python/blob/master/search.py

A Search Problem Example: 8-Queen Problem



```python
class NQueensProblem(Problem):
    """The problem of placing N queens on an NxN board with none attacking
    each other. A state is represented as an N-element array, where
    a value of r in the c-th entry means there is a queen at column c,
    row r, and a value of -1 means that the c-th column has not been
    filled in yet. We fill in columns left to right.
    >>> depth_first_tree_search(NQueensProblem(8))
    <Node (7, 3, 0, 2, 5, 1, 6, 4)>
    """

    def __init__(self, N):
        # tuple([-1] * 8) = (-1, -1, -1, -1, -1, -1, -1, -1), if N = 8
        super().__init__(tuple([-1] * N))
        self.N = N

    def actions(self, state):
        """In the leftmost empty column, try all non-conflicting rows."""
        if state[-1] != -1:
            return []  # All columns filled; no successors
        else:
            col = state.index(-1)
            return [row for row in range(self.N)
                    if not self.conflicted(state, row, col)]

    def result(self, state, row):
        """Place the next queen at the given row."""
        col = state.index(-1)
        new = list(state[:])
        new[col] = row
        return tuple(new)
```

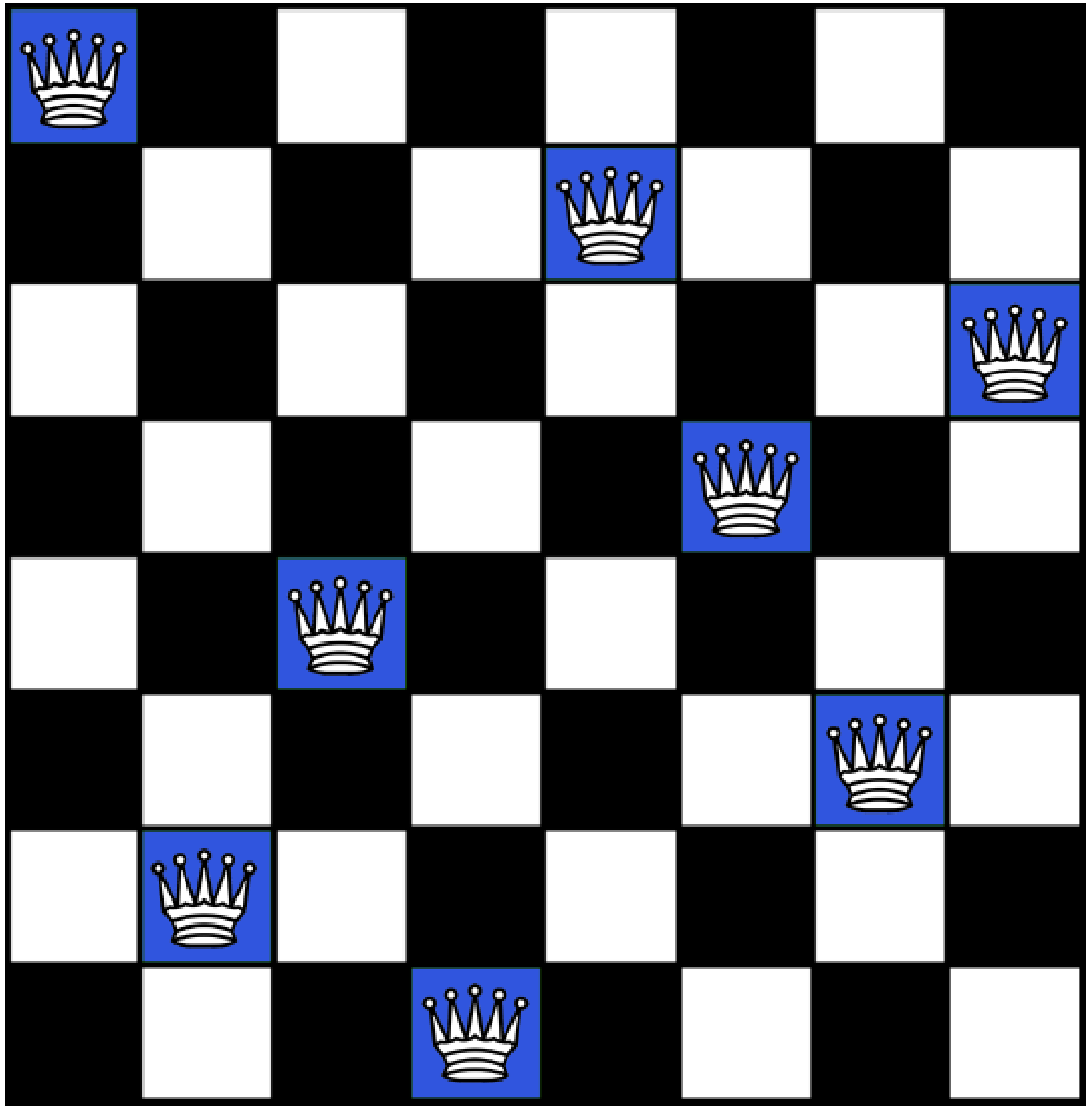


https://github.com/aimacode/aima-python/blob/master/search.py

A Search Problem Example: 8-Queen Problem

```python
def conflicted(self, state, row, col):
    """Would placing a queen at (row, col) conflict with anything?"""
    return any(self.conflict(row, col, state[c], c)
               for c in range(col))

def conflict(self, row1, col1, row2, col2):
    """Would putting two queens in (row1, col1) and (row2, col2) conflict?"""
    return (row1 == row2 or  # same row
            col1 == col2 or  # same column
            row1 - col1 == row2 - col2 or  # same \ diagonal
            row1 + col1 == row2 + col2)  # same / diagonal

def goal_test(self, state):
    """Check if all columns filled, no conflicts."""
    if state[-1] == -1:
        return False
    return not any(self.conflicted(state, state[col], col)
                   for col in range(len(state)))

def h(self, node):
    """Return number of conflicting queens for a given node"""
    num_conflicts = 0
    for (r1, c1) in enumerate(node.state):
        for (r2, c2) in enumerate(node.state):
            if (r1, c1) != (r2, c2):
                num_conflicts += self.conflict(r1, c1, r2, c2)

    return num_conflicts
```

Total number of queen attacks at a node, i.e., a state.

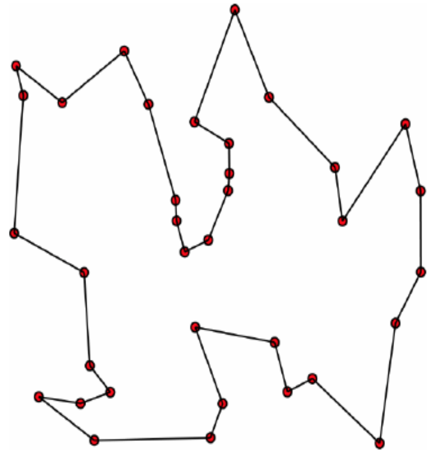


Real-world Examples

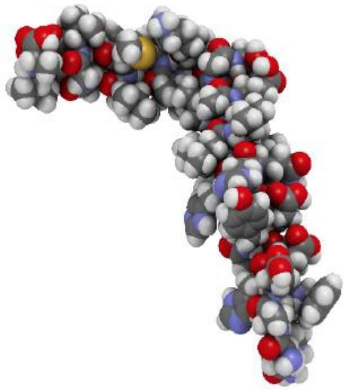
Route finding problem: typically our example of map search, where we need to go from location to location using links or transitions. Example of applications include tools for driving directions in websites, in-car systems, etc.

Automatic assembly sequencing: find an order in which to assemble parts of an object which is in general a difficult and expensive geometric search.

Traveling salesperson problem: Find the shortest tour to visit each city exactly once.

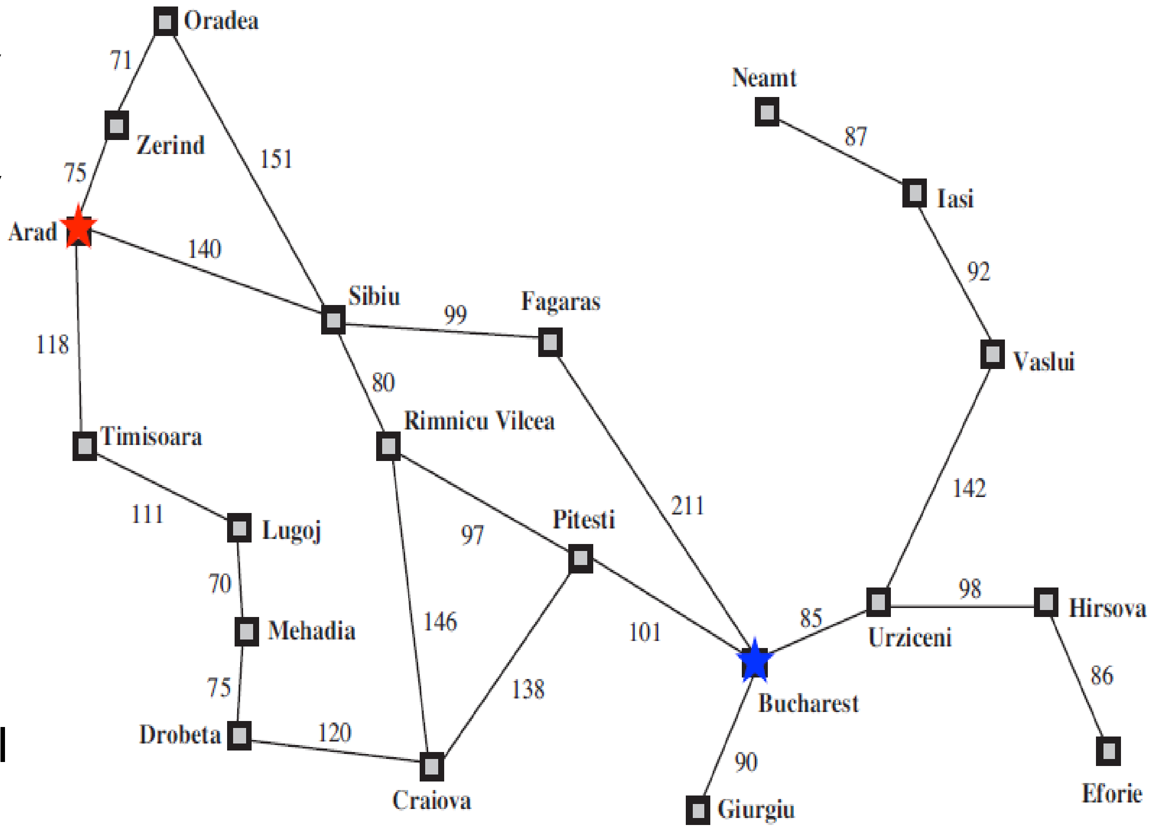


Protein design: find a sequence of amino acids that will fold into a 3D protein with the right properties to cure some disease.

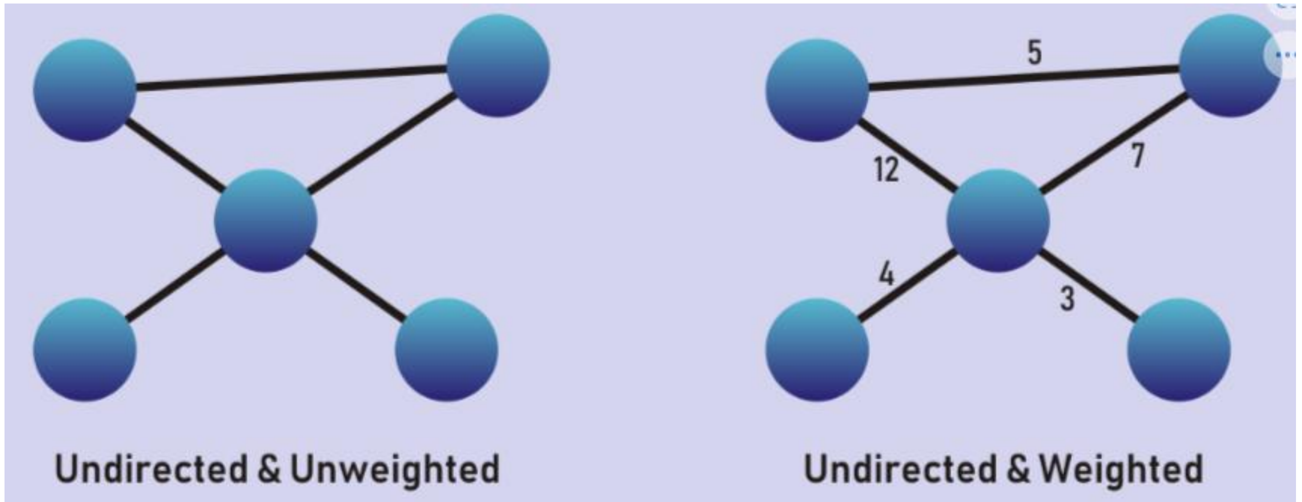


A Search Problem Summary

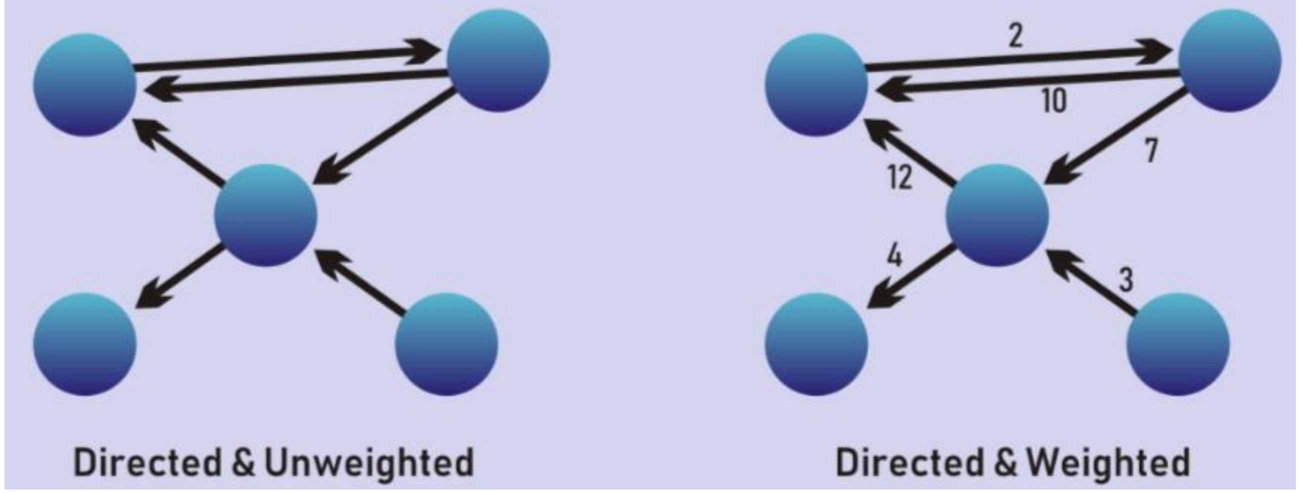
- A search problem formation consists of:
  - A state space S
    - In a state space graph, each state occurs only once!
    - The size of the state-space graph = |V| + |E|, where |V| is the number of vertices (state nodes) of the graph and |E| is the number of edges.
  - An initial state s_0
  - Actions $A(s)$ in each state
  - Transition model $Result(s,a)$
  - A goal state $G(s)$
  - Action cost $c(s,a,s')$
- A solution is an action sequence that reaches a goal state.
- An optimal solution has the min cost or max profit (depend on the problem) among all solutions.

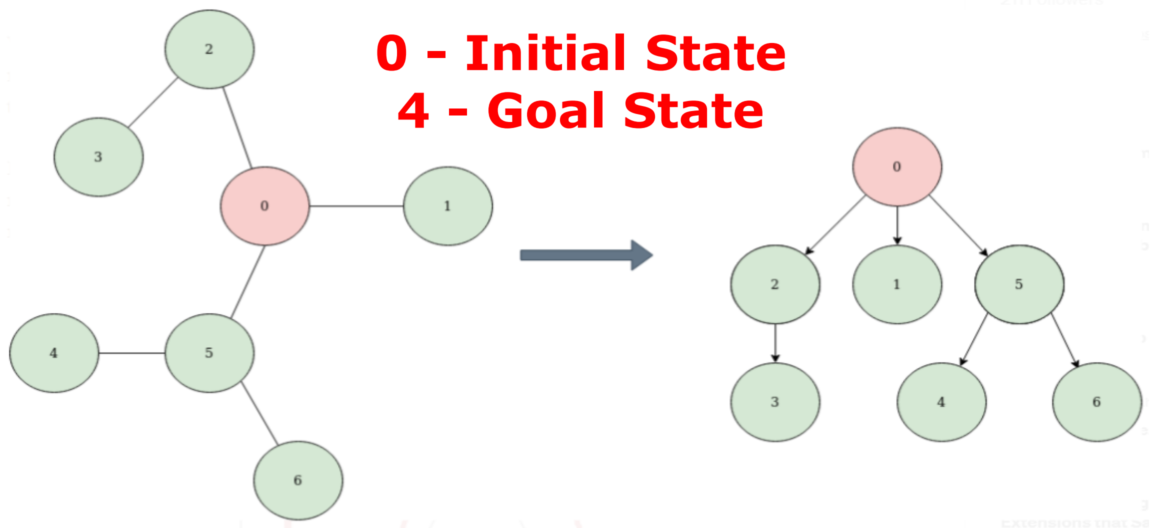


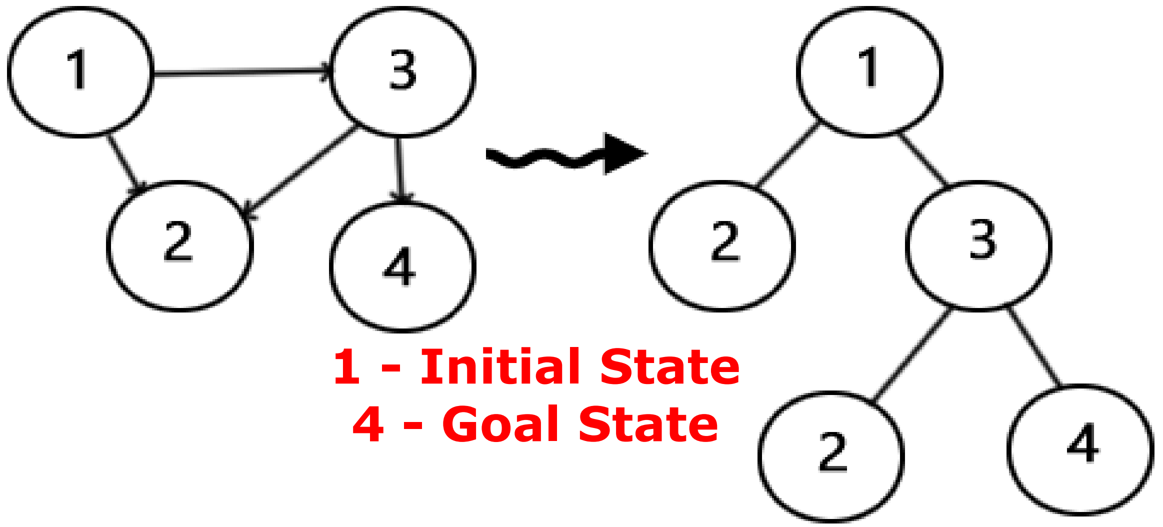
Graphs Representing Search Problems and Solutions



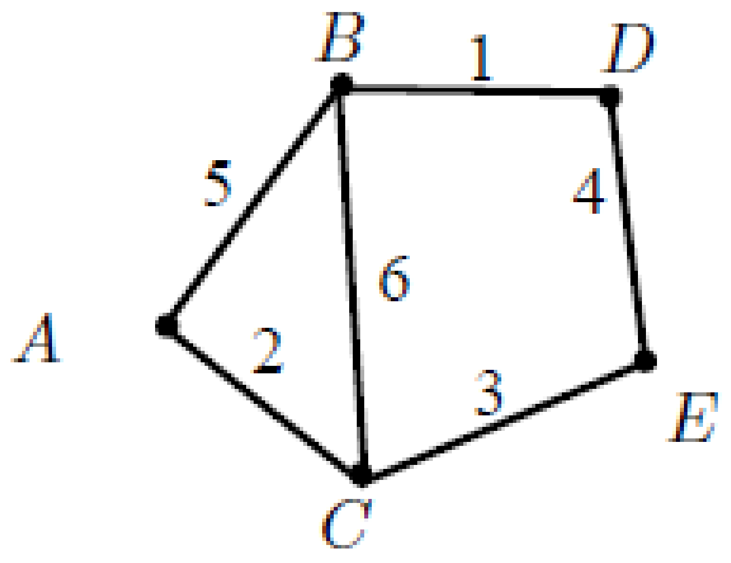
A graph contains several states (nodes or vertices) with connections among them (edges).





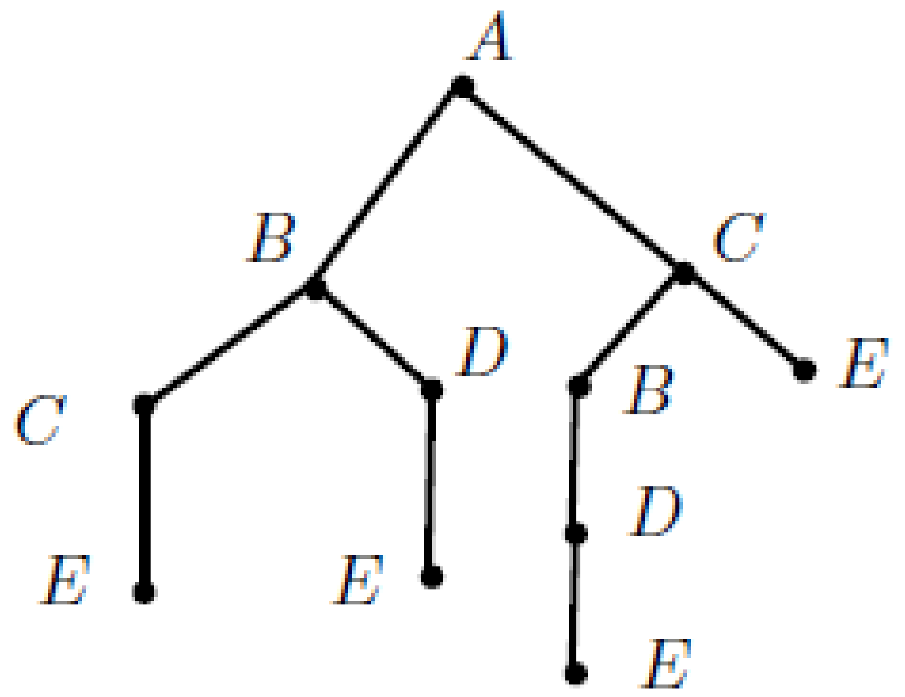


Graphs Representing Search Problems and Solutions



Start node = A
Goal node = E

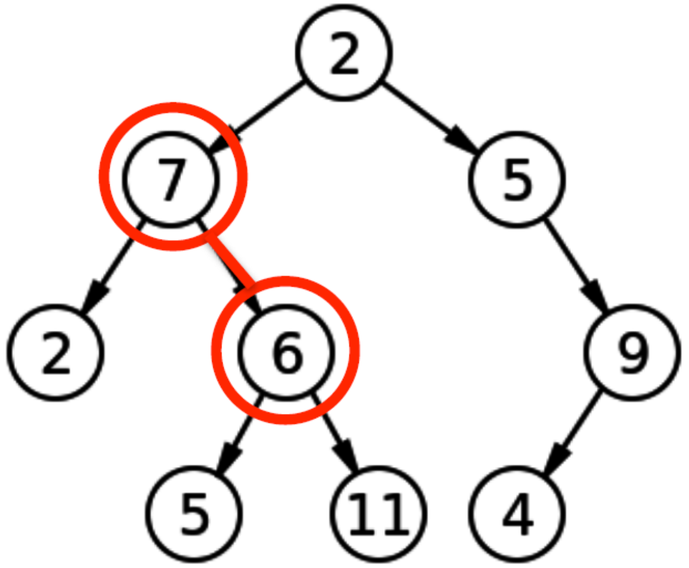
(a) Graph with vertices A-E.

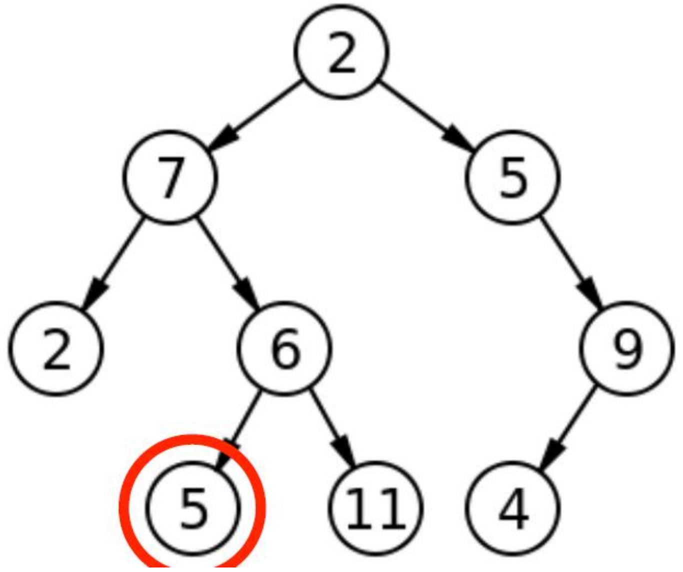


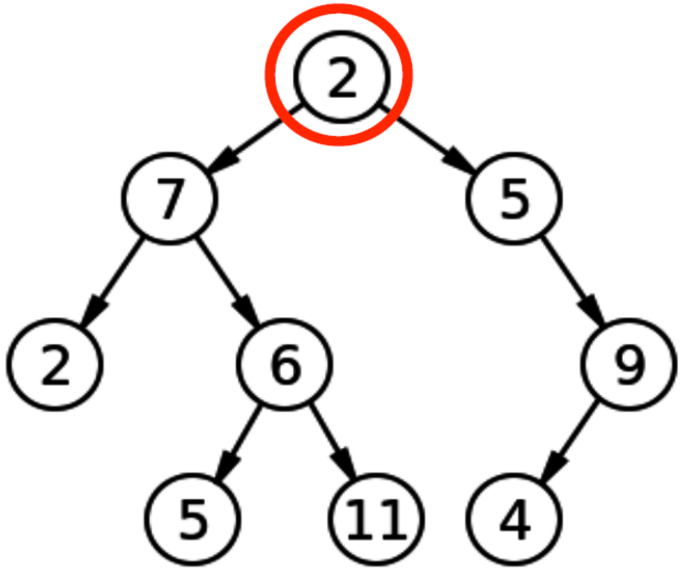
(b) Graph search is a tree.

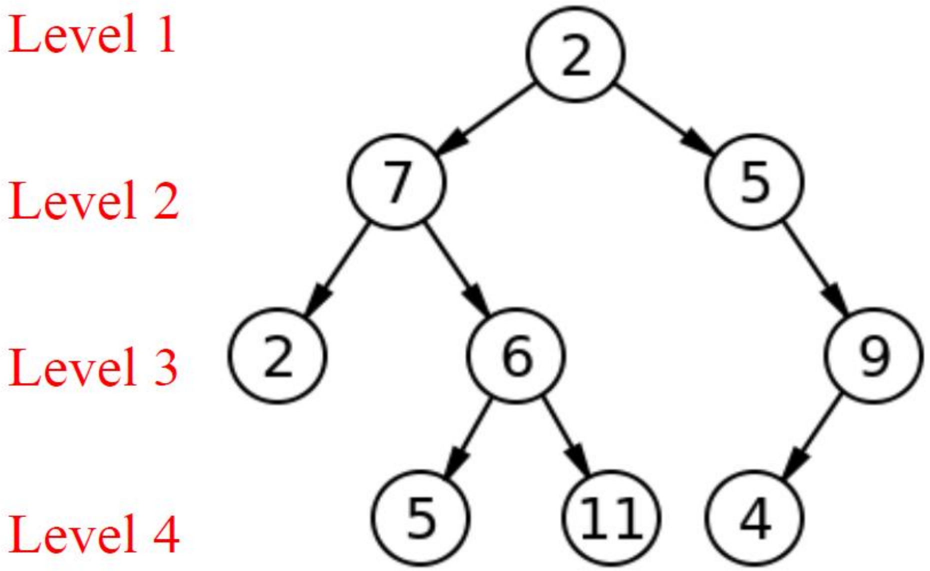
Tree Terminology

- *Root*: No parent
- *Parent/Child*: Relationship
- *Leaf*: No child
- *Level*: Depth



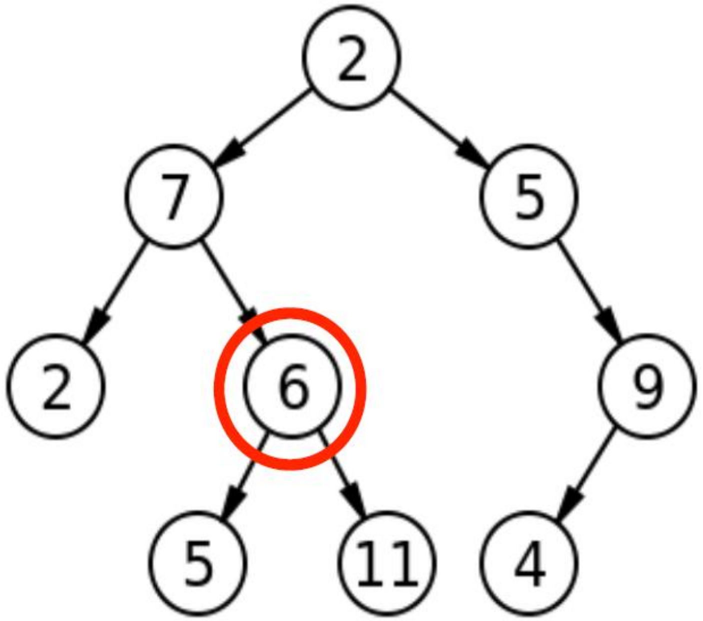


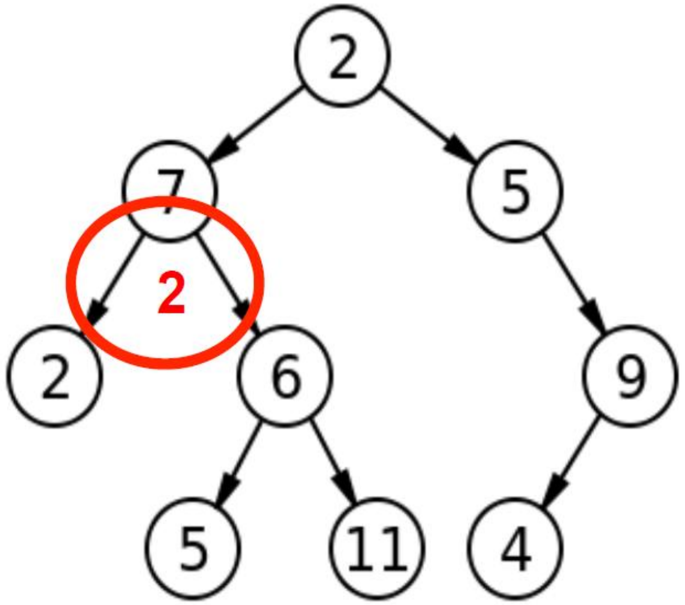


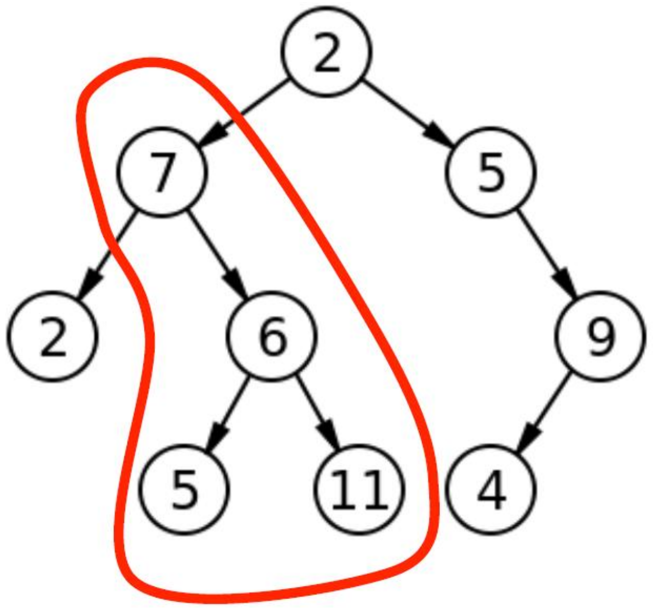


Tree Terminology

- *Internal nodes*: Parents and children
- *Branching factor (Degree)*: Number of children
- *Subtree*: A part of tree (a tree too)







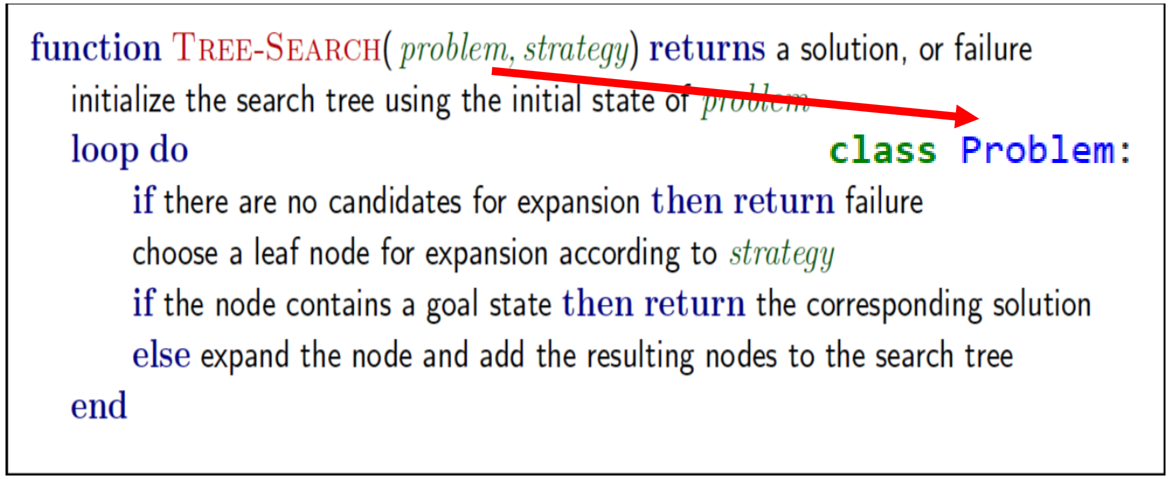
Search Algorithms

- A **search algorithm/strategy** takes a search problem as input and returns a solution or an indication of failure.

Uninformed and Informed.

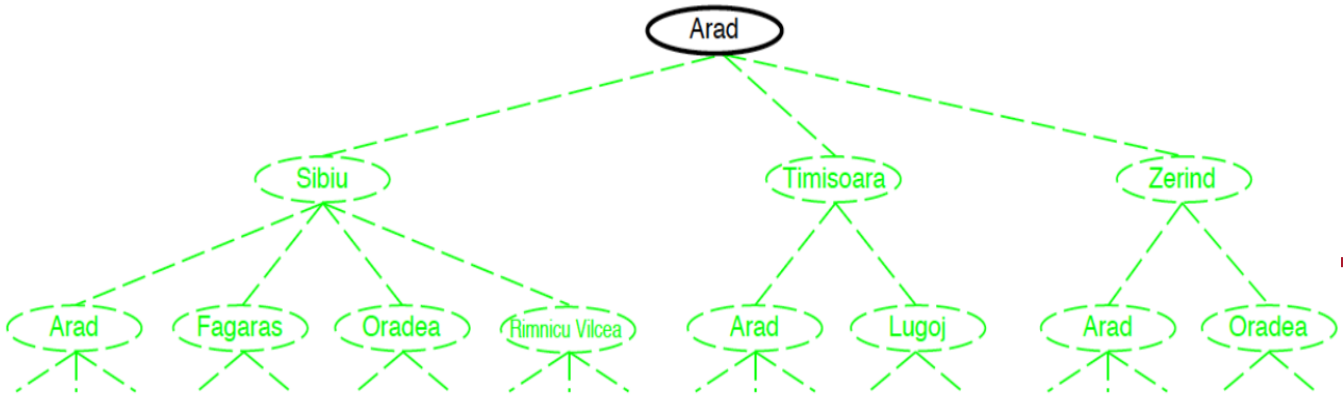
- One search approach is to build a **Search Tree** when exploring the **State Space Graph**

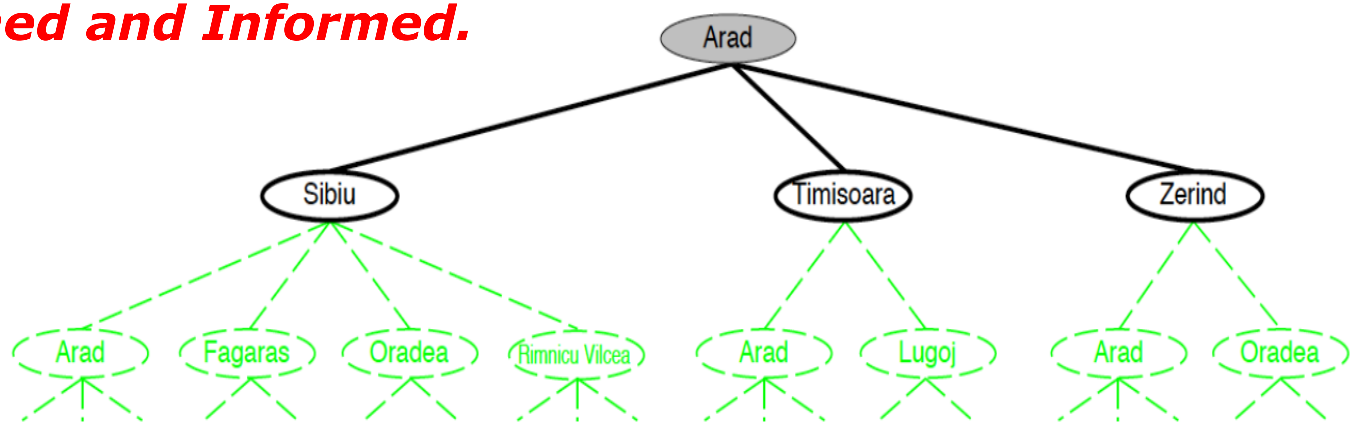
- A **search tree** forms various paths from the initial state, trying to find a path that reaches a goal state.
  - ❑ Each **node** is a **state** in the state space that describes the set of states.
  - ❑ Each **edge** is an **action** that allows transitions from one state to another.



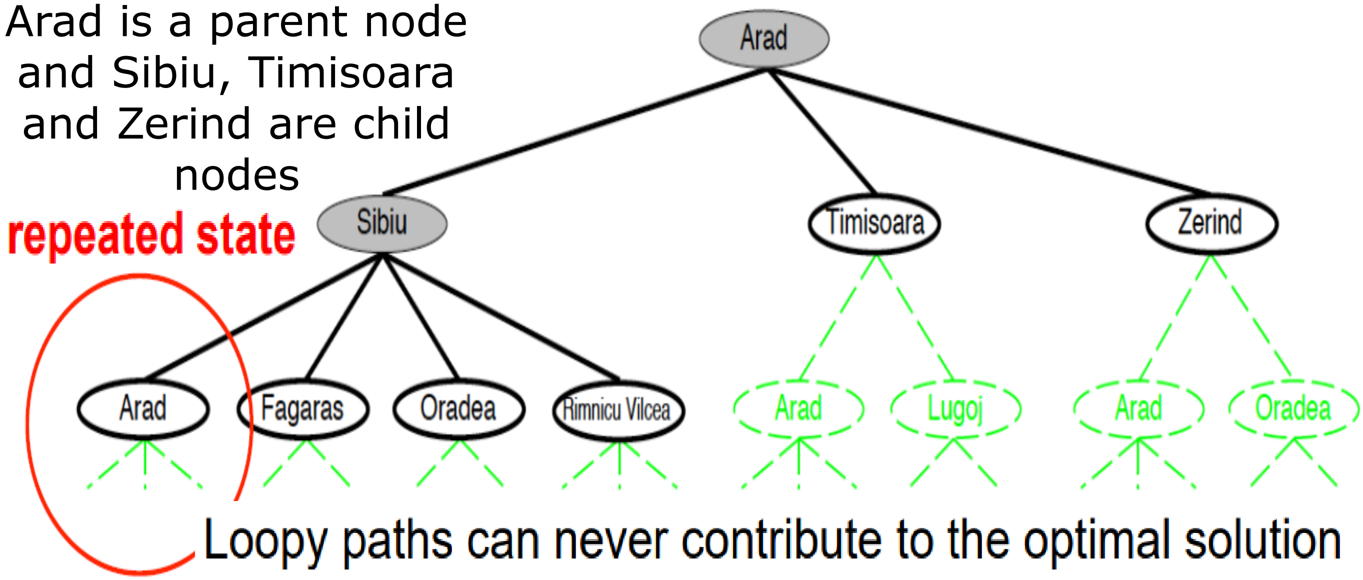
```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

```
class Problem:
```





Arad is a parent node and Sibiu, Timisoara and Zerind are child nodes



Loopy paths can never contribute to the optimal solution

Criteria for Evaluating Search Strategies (Algorithms)

Guarantees:

Completeness:	Is the strategy guaranteed to find a solution when there is one?
Optimality:	Are the returned solutions guaranteed to be optimal?

Complexity:

Time Complexity:	How long does it take to find a solution? (Measured in generated states.)
Space Complexity:	How much memory does the search require? (Measured in states.)

Typical state space features governing complexity:

Branching factor b:	How many successors does each state have?
Goal depth d:	The number of actions required to reach the shallowest goal state.

m: maximum depth of the state space ($d \leq m$)

Uninformed Search Strategies

- *An* ***uninformed (unguided, blind, or brute-force) search algorithm*** has no priori knowledge or additional information about the states and the length or cost of a path to the solution, i.e., no clue about how close a state to the goal state(s), to eliminate some paths. They check all the plausible paths and pick the best one as they can as the final solution. Some search algorithms are:

- *Breadth-first search (BFS): Expand shallowest node*

- *Depth-first search (DFS): Expand deepest node*

- *Backtracking search (BTS): DFS variant*

- *Depth-limited search (DLS): Depth first with depth limit*

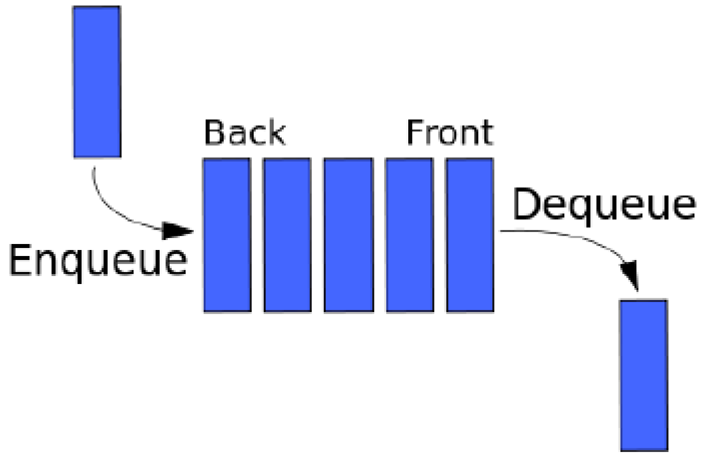
- *Iterative deepening search (IDS): DLS with increasing limit*

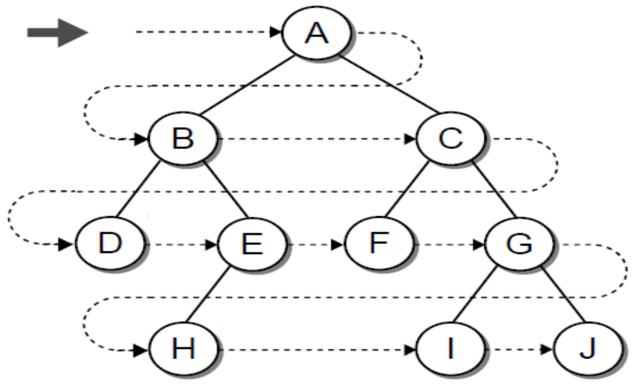
- *Uniform-cost search (UCS): Expand least cost node*

- *Bidirectional search (BDS): Simultaneously search forward from the initial state and backwards from the goal state(s)*

Breadth-first Search (BFS)

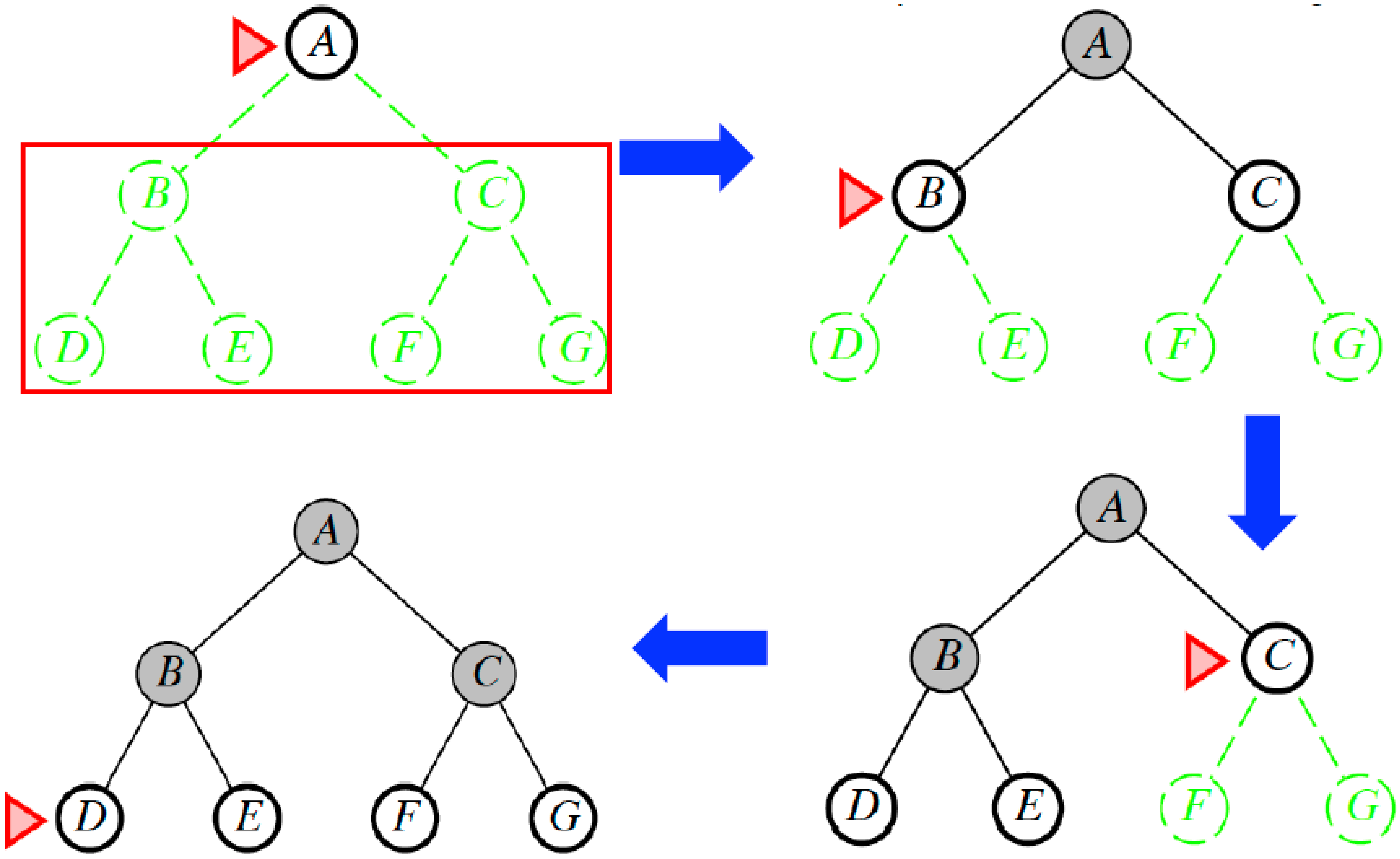
- A simple strategy in which the root node is expanded first, then **all** successors are expanded, then successors of the successors, *etc.*
- **All** the nodes are expanded at a given depth in the search tree before the any nodes at the next level are expanded.
- Implementation as a First In First Out (*FIFO*) *queue*: new successors go at end
- ***Frontier***: The set of nodes that have been reached but not yet expanded.
- ***Exterior***: The set of nodes that have not been reached and expanded.
- ***Interior***: The set of nodes that have been reached and expanded.
- The **breadth_first_tree_search(problem)** function in search.py.





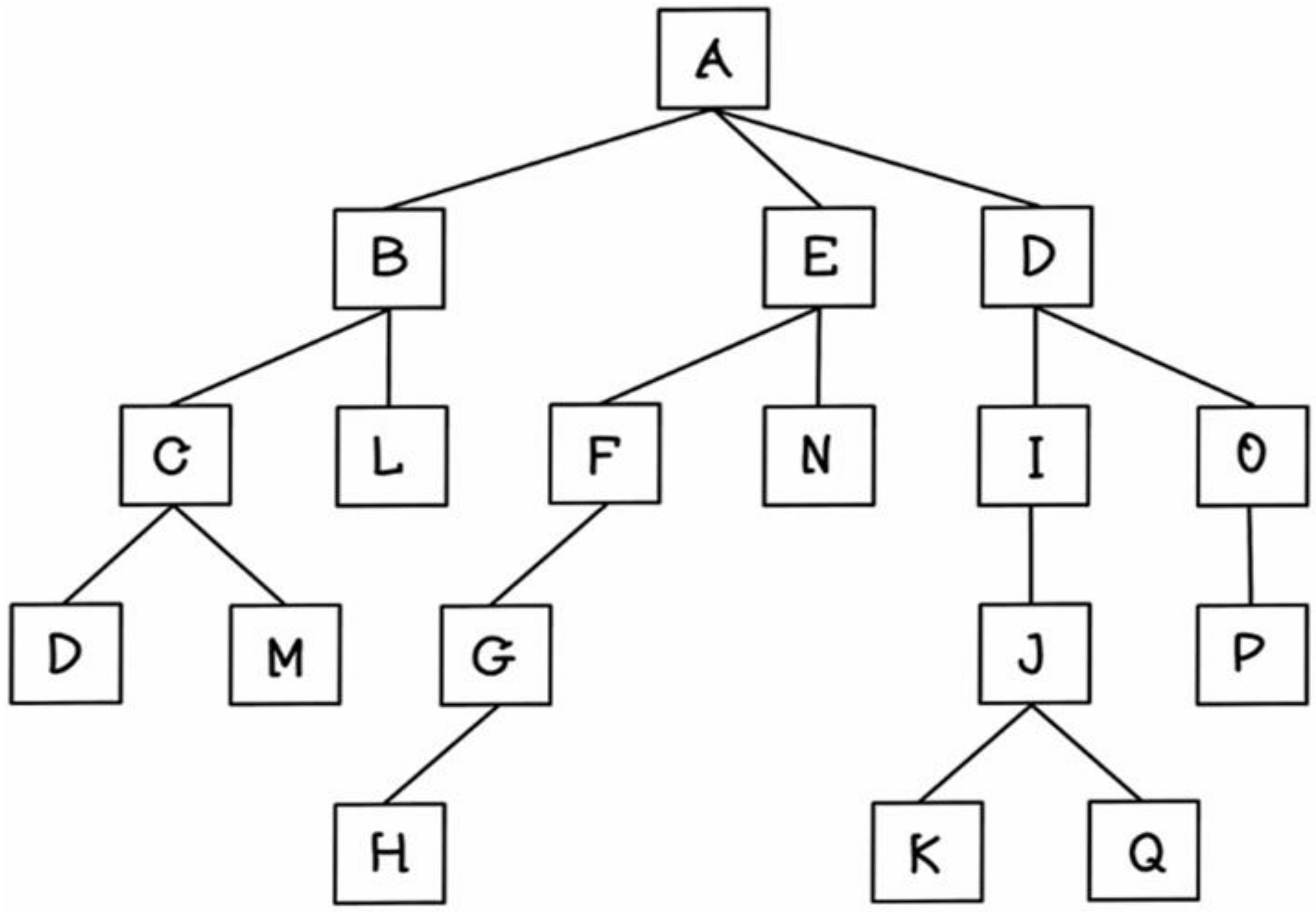
Breadth-first Search (BFS)

- ***Frontier***: The set of nodes that have been reached but not yet expanded.
- ***Exterior***: The set of nodes that have not been reached and expanded.
- ***Interior***: The set of nodes that have been reached and expanded.



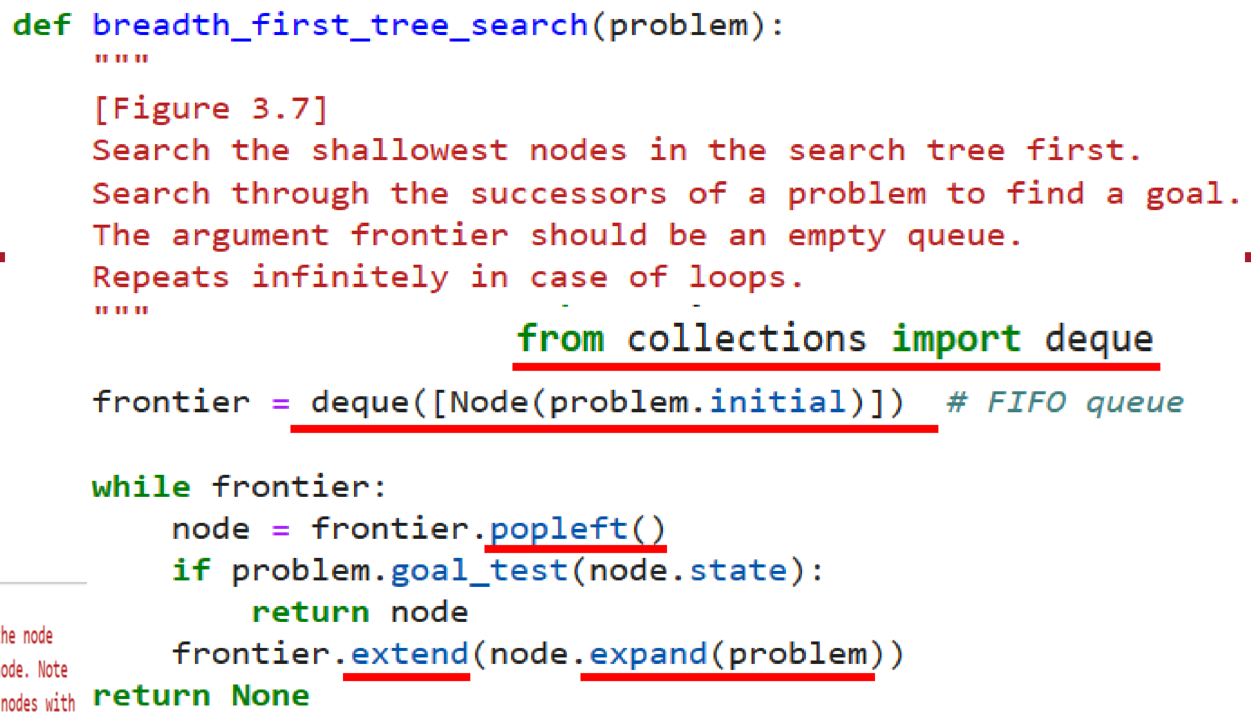
Class Exercise 5

Determine The Path To The Solution: What Would Be The Order Of Visits Using Breadth-first Search For The Following Tree?

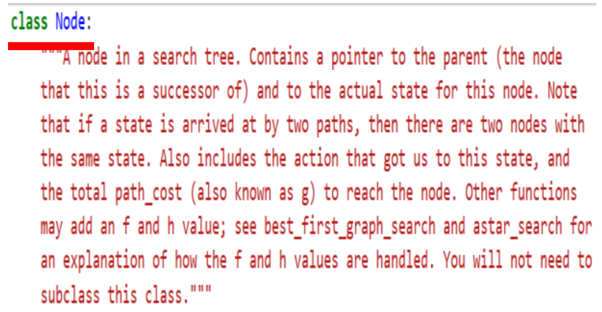


Breadth-first Search

- *Completeness*: Complete if *b is finite, and the state space either has a solution or is finite*
- *Optimality*: Optimal if action costs are all identical
- *Time complexity*: $O(b^d)$
- *Space complexity*: $O(b^d)$
- *b:* maximum branching factor of the search tree
- *d*: depth of the least-cost, optimal solution path
- *Implementation*: FIFO (Queue)

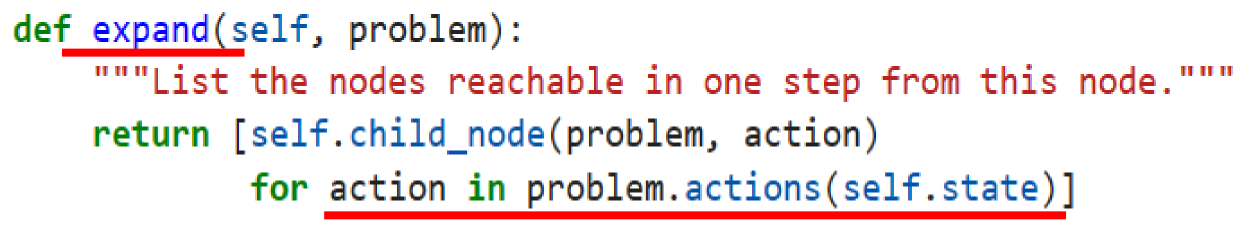


```
def breadth_first_tree_search(problem):
    """
    [Figure 3.7]
    Search the shallowest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Repeats infinitely in case of loops.
    """
                          from collections import deque

    frontier = deque([Node(problem.initial)])  # FIFO queue

    while frontier:
        node = frontier.popleft()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
```

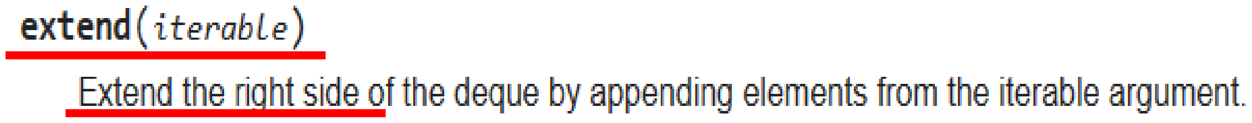


```
class Node:
    """A node in a search tree. Contains a pointer to the parent (the node
    that this is a successor of) and to the actual state for this node. Note
    that if a state is arrived at by two paths, then there are two nodes with
    the same state. Also includes the action that got us to this state, and
    the total path_cost (also known as g) to reach the node. Other functions
    may add an f and h value; see best_first_graph_search and astar_search for
    an explanation of how the f and h values are handled. You will not need to
    subclass this class."""
```

The **popleft()** method is used to remove and return left most element from the queue.



```
def expand(self, problem):
    """List the nodes reachable in one step from this node."""
    return [self.child_node(problem, action)
            for action in problem.actions(self.state)]
```

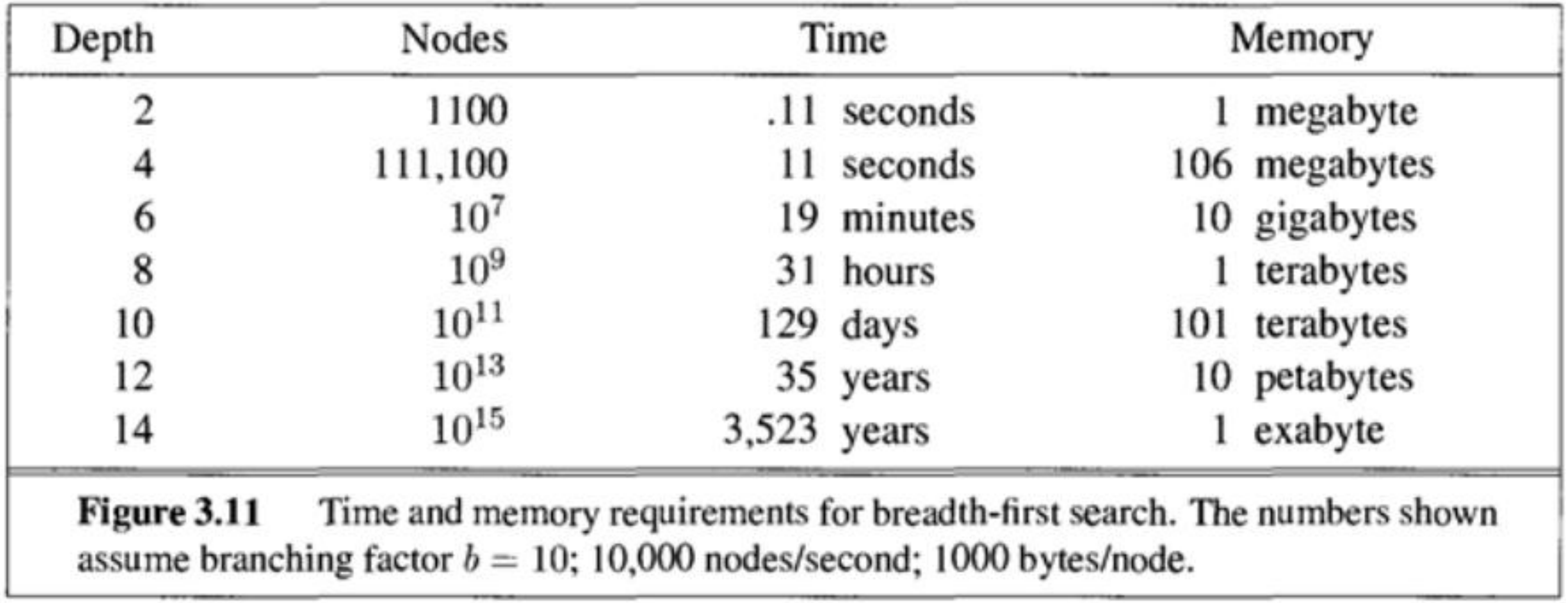


extend(*iterable*)

Extend the right side of the deque by appending elements from the iterable argument.

Breadth-first Search (BFS) : Time and Space

Time and Space Complexities are exponential

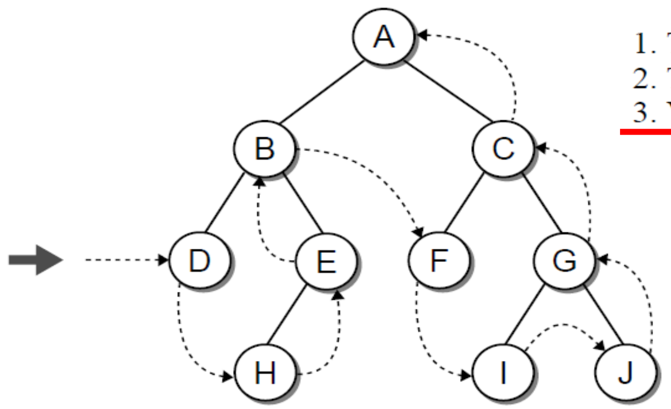


Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

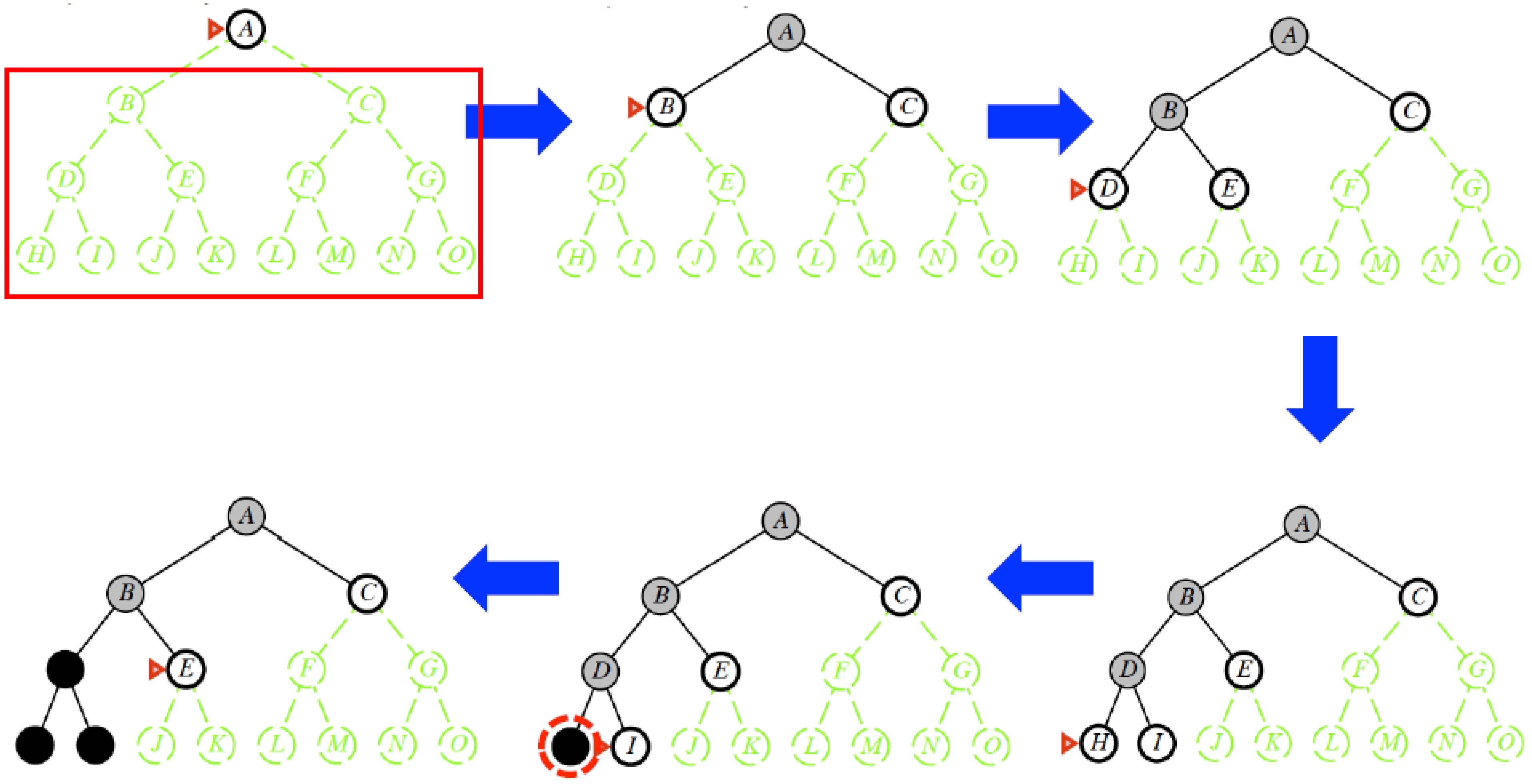
Depth-first Search (DFS)

- Always expand the deepest node in the current frontier
- As those nodes are expanded, they are dropped from the frontier, so the search backups to the next **deepest** unexplored node.
- *Implementation*: Last-in-first-out (*LIFO*) *stack*—put successors in front (most recently generated node is chosen for expansion)
- The **parent node** is processed **after** the processing of its subtrees has been completed.
- The **depth_first_tree_search(problem)** function in search.py.



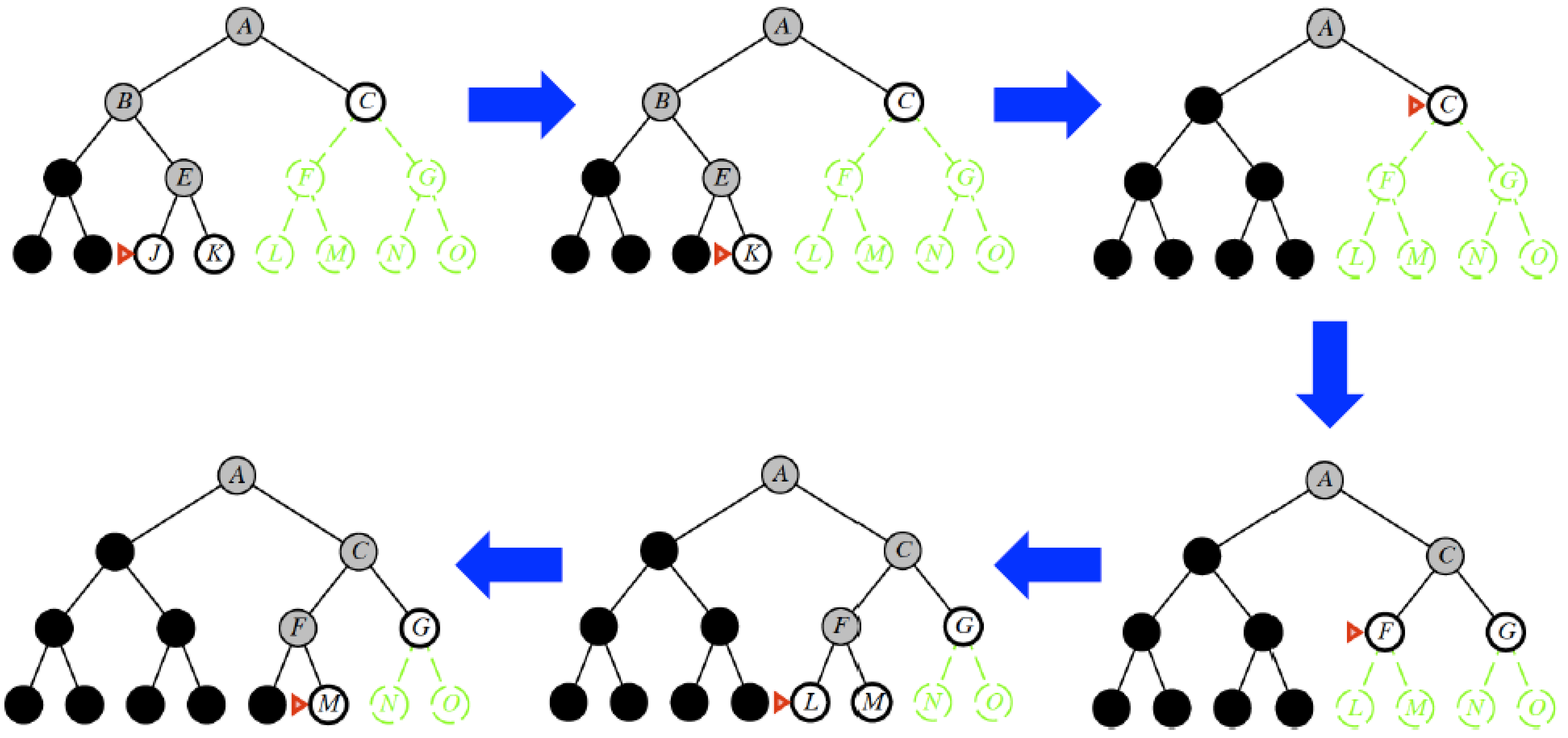
1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the node.

Depth-first Search (DFS)



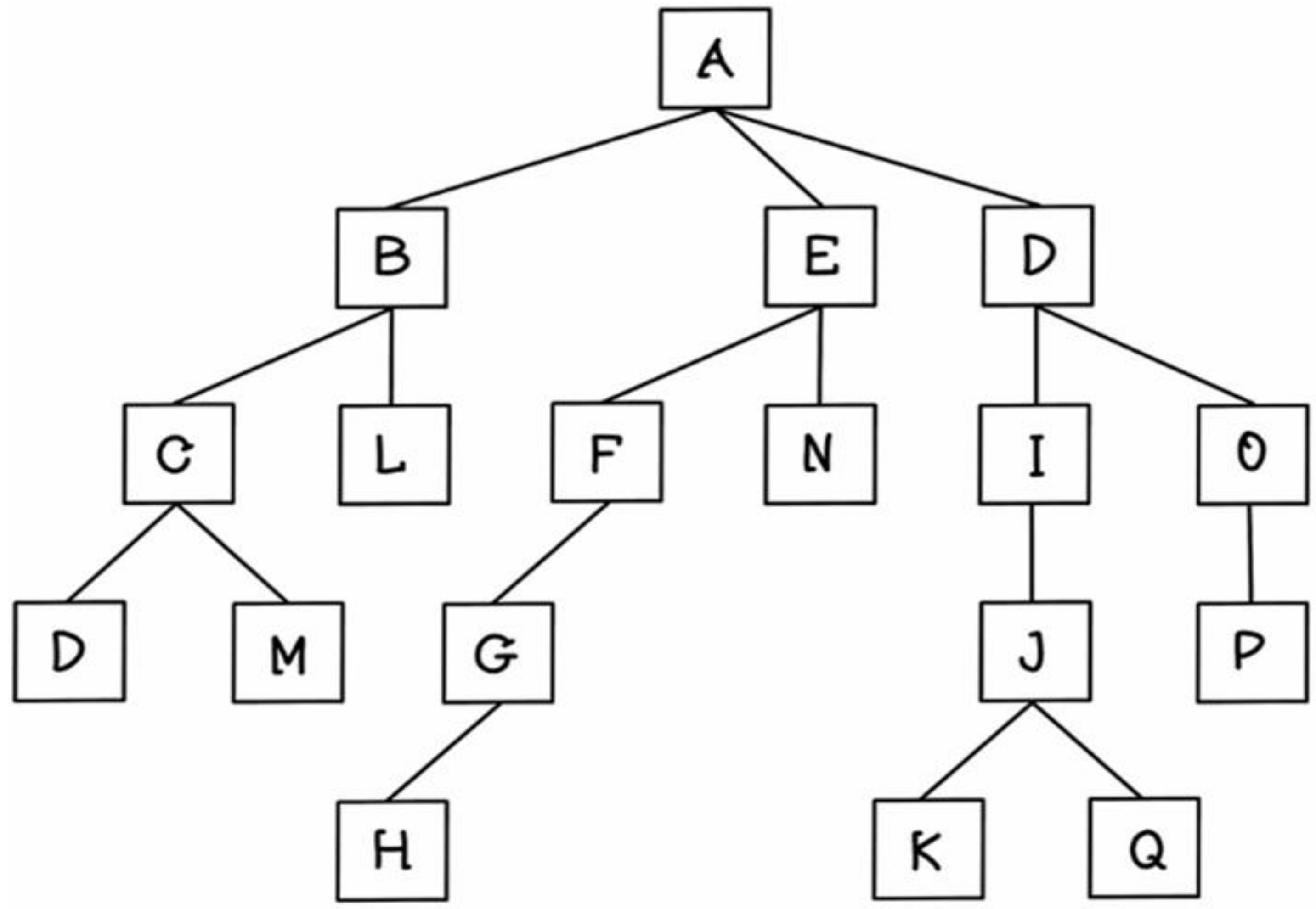
Worcester Polytechnic Institute

Depth-first Search (DFS)



Class Exercise 6

Determine The Path To The Solution: What Would Be The Order Of Visits Using depth-first Search For The Following Tree?



Depth-first Search (DFS)

- *Completeness*: No if the graph tree is **not** finite in length *m* or else it is complete
- *Optimality*: No
- *Time complexity*: *$O(b^m)$*, where m is the maximum depth of the tree.
- *Space complexity*: *$O(bm)$* — linear!!!
- *Implementation*: LIFO (Stack)

```
def depth_first_tree_search(problem):
    """
    [Figure 3.7]
    Search the deepest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Repeats infinitely in case of loops.
    """

    frontier = [Node(problem.initial)]  # Stack

    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
```

```
def expand(self, problem):
    """List the nodes reachable in one step from this node."""
    return [self.child_node(problem, action)
            for action in problem.actions(self.state)]
```

```
extend(iterable)
```

Extend the right side of the deque by appending elements from the iterable argument.

The **pop()** method is used to remove and return the right most element from the queue.

Backtracking Search (BTS)

- It is a **variant** of depth-first search.
- Only **one** successor is generated/reached and then **expanded at a time** rather than all successors
- Each **partially expanded node, i.e., the parent node,** remembers which successor to generate next
- More reading: Chapter 6 for more details
- *Completeness*: No
- *Optimality*: No
- *Time complexity*: $O(b^m)$, where m is the maximum depth of the tree.
- *Space complexity*: $O(m)$ — linear, independent of b !!!
- The **backtracking_search()** function in csp.py.

Depth-limited Search (DLS)

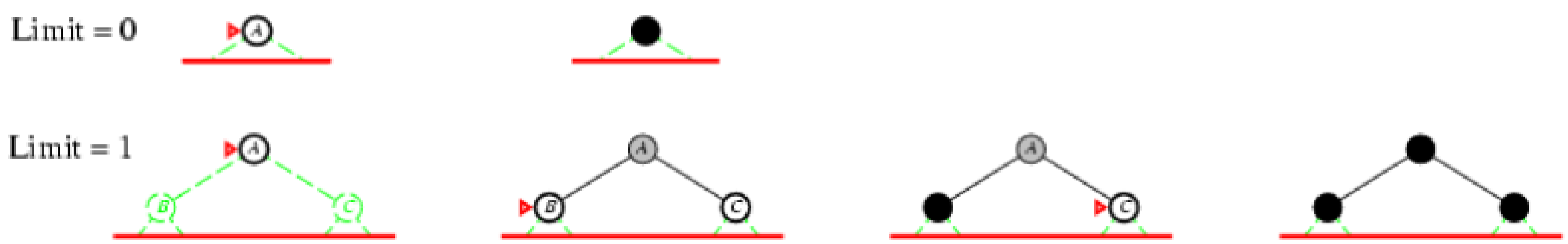
- A depth-limited search algorithm is **another variant of depth-first search** with **a predetermined limit *l***. In this algorithm, the node at the depth limit will treat as it has **no successor nodes further**.
- Allows to avoid the failure of DFS in infinite state spaces
- Sometimes the depth limit can be based on the knowledge of the problem
- *Completeness*: No
- *Optimality*: No
- *Time complexity*: $O(b^l)$ only useful if l is much smaller than m
- *Space complexity*: $O(bl)$ — linear!!!
- The **depth_limit_search(problem, limit=l)** function in search.py.

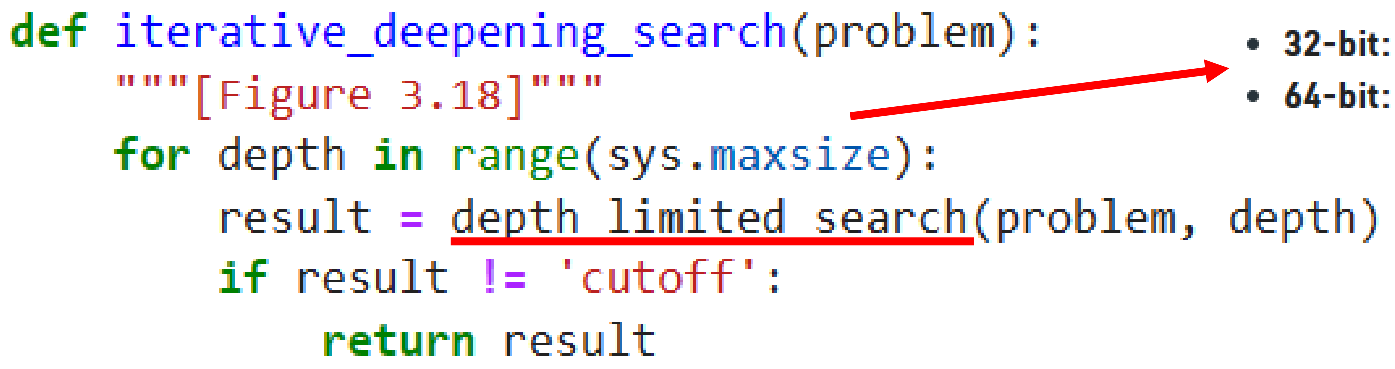
```python
def depth_limited_search(problem, limit=50):
    """[Figure 3.17]"""

    def recursive_dls(node, problem, limit):
        if problem.goal_test(node.state):
            return node
        elif limit == 0:
            return 'cutoff'
        else:
            cutoff_occurred = False
            for child in node.expand(problem):
                result = recursive_dls(child, problem, limit - 1)
                if result == 'cutoff':
                    cutoff_occurred = True
                elif result is not None:
                    return result
            return 'cutoff' if cutoff_occurred else None

    # Body of depth_limited_search:
    return recursive_dls(Node(problem.initial), problem, limit)
```

Iterative Deepening Search (IDS)

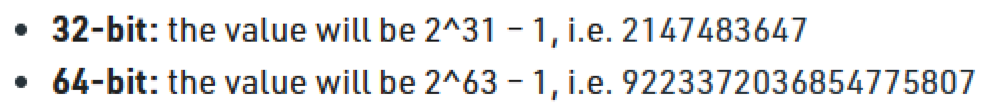
- The iterative deepening algorithm is **a combination of DFS first and then BFS algorithms** that finds out the **best depth limit (d ≤ sys.maxsize)** and does it by **gradually increasing the limit** until a goal is found.

- It gradually increases the limit from 0 to 1, to 2, to 3, until a goal is found



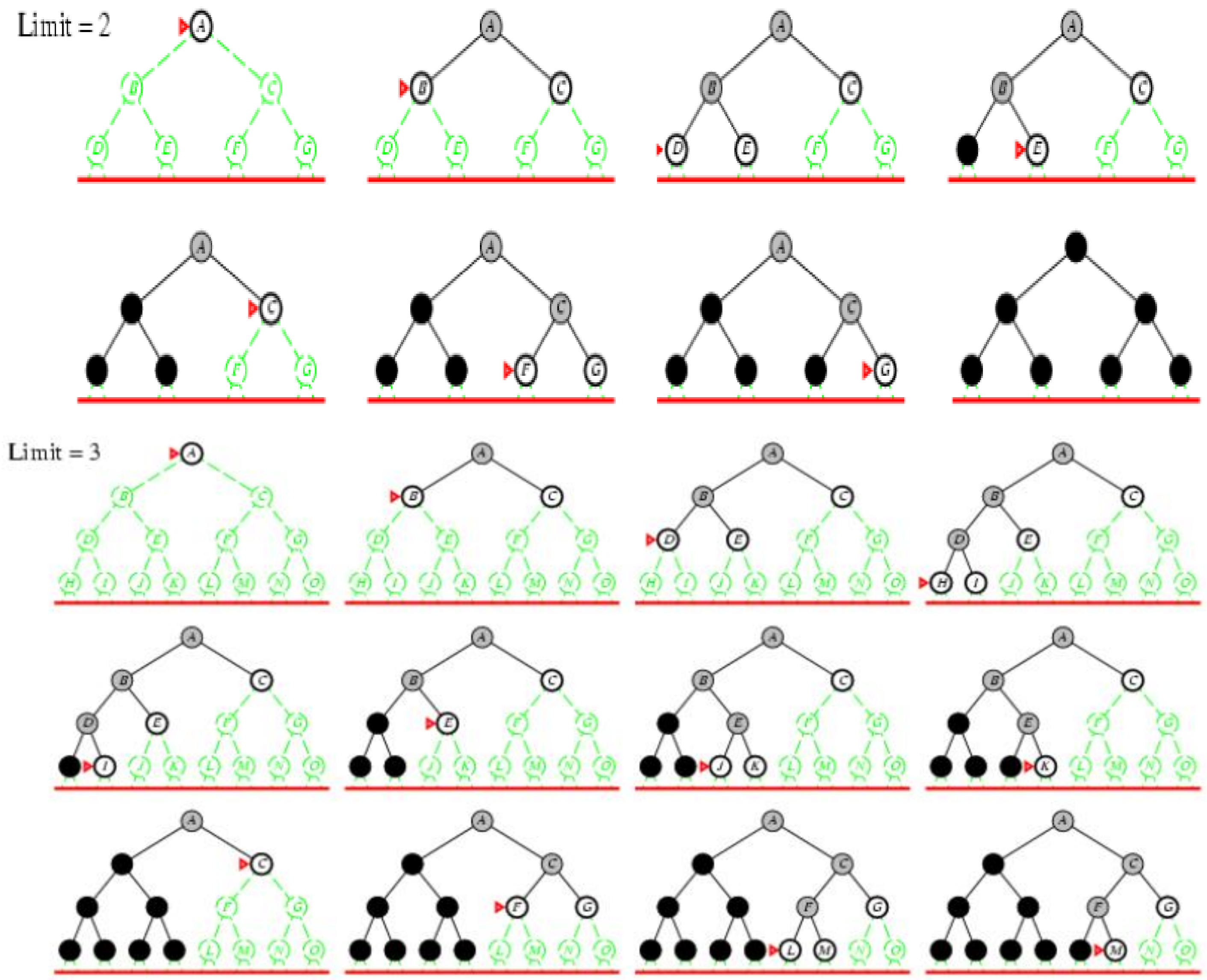


```python
def iterative_deepening_search(problem):
    """[Figure 3.18]"""
    for depth in range(sys.maxsize):
        result = depth_limited_search(problem, depth)
        if result != 'cutoff':
            return result
```



- **32-bit:** the value will be 2^31 – 1, i.e. 2147483647
- **64-bit:** the value will be 2^63 – 1, i.e. 9223372036854775807

Iterative deepening Search (IDS)

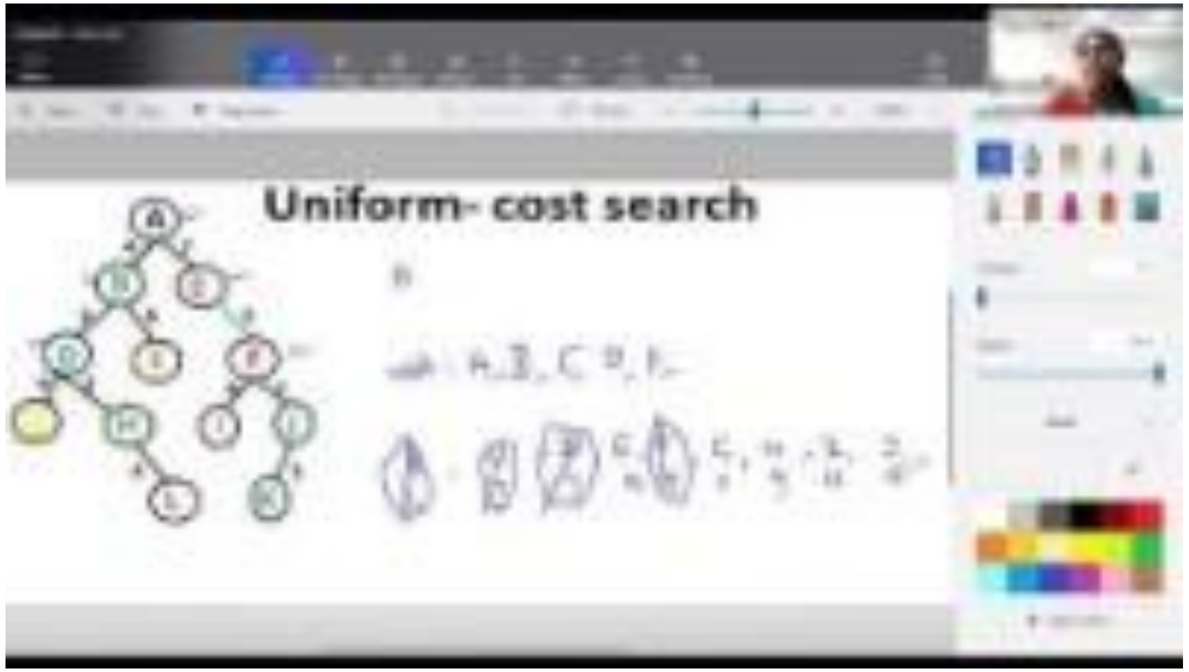


```
def iterative_deepening_search(problem):
    """[Figure 3.18]"""
    for depth in range(sys.maxsize):
        result = depth_limited_search(problem, depth)
        if result != 'cutoff':
            return result
```

- *Completeness*: Complete if *b is finite, and the state space either has a solution or is finite*
- *Optimality*: Optimal if action costs are all identical
- *Time complexity*: *O(b^d), d* is the depth of the least-cost, optimal solution path
- *Space complexity*: *O(bd)* — linear!!!
- The **iterative_deepening_search(problem)** function in search.py.

Uniform-cost Search (UCS)

- Branches of the tree (steps) have different weights (step costs)
- Expand least-cost unexpanded node.
- **g(n)** is defined as the total cost of getting to a current node n from the root node.
- **Equivalent to Breadth-first Search if step costs all equal**
- Uniform Cost Search Method Using Example Video:
- *Completeness*: Complete if ***b*** *is finite, and the state space either has a solution or is finite, and* ***all the action*** **costs are ≥ *ε***, where ***ε*** **is a lower bound on the cost of each action**
- *Optimality*: Yes
- *Time complexity*: $O(b^{1+floor(C^*/\varepsilon)})$, where C^* is the cost of the optimal solution path
- *Space complexity*: $O(b^{1+floor(C^*/\varepsilon)})$
- The **uniform_cost_search(problem)** function in search.py.



Bidirectional Search (BDS)

- Simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches, **usually the BFS**, will meet.
- Motivation: $b^{d/2} + b^{d/2} = 2\ b^{d/2} << b^d$
- The **bidirectional_search(problem)** function in search.py.

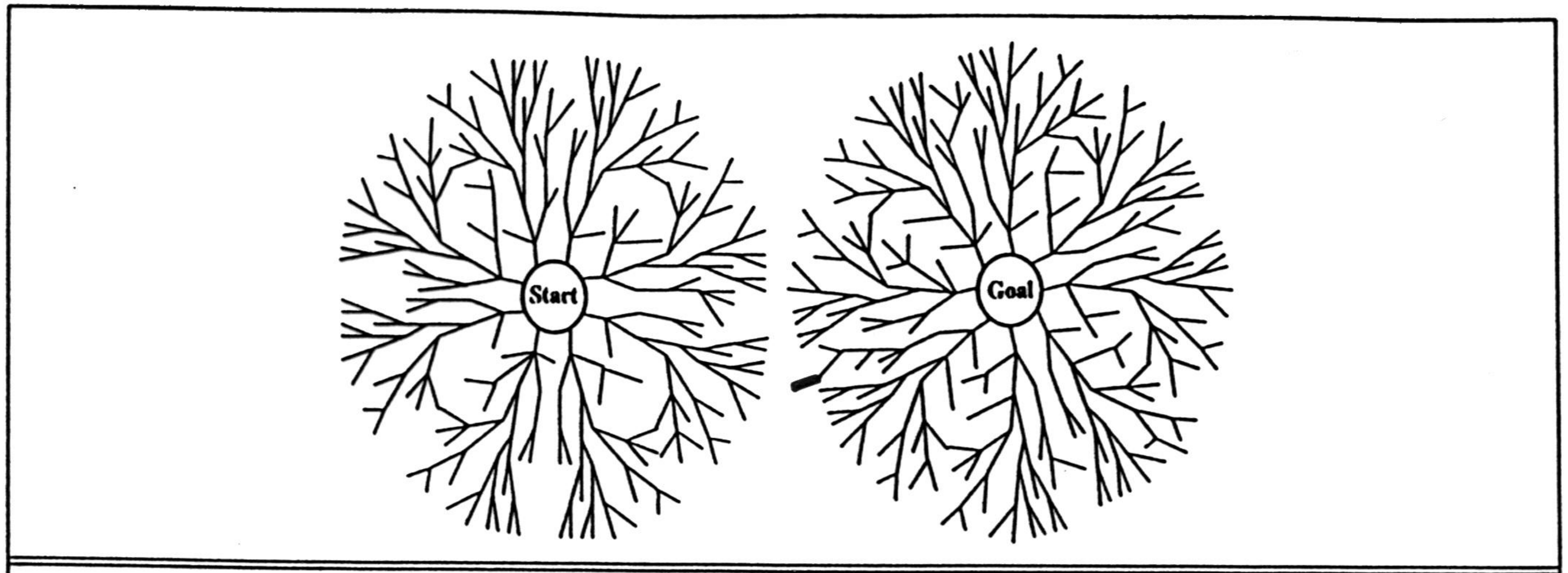
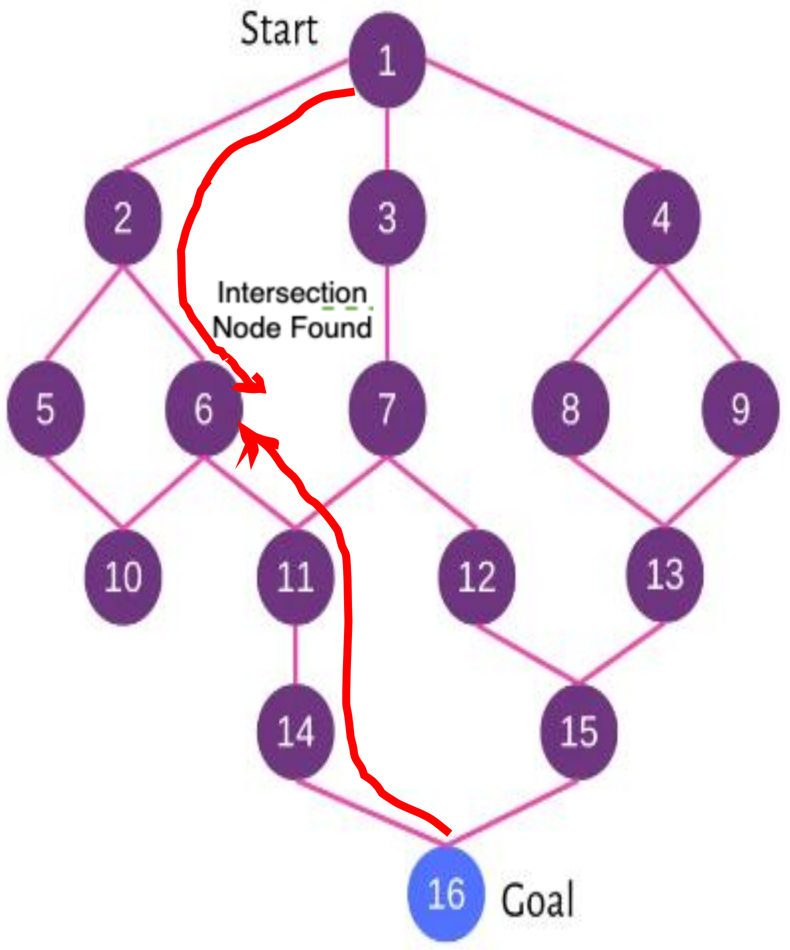


Figure 3.17 A schematic view of a bidirectional breadth-first search that is about to succeed, when a branch from the start node meets a branch from the goal node.



Comparing Uninformed Search Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bi-directional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No[e]	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$[f]	$O(bl)$[f]	$O(bd)$[f]	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Where:

b	branching factor
d	depth of solution
m	maximum depth of the search tree
l	depth limit
C^*	cost of the optimal solution
ε	minimal cost of an action

Superscripts:

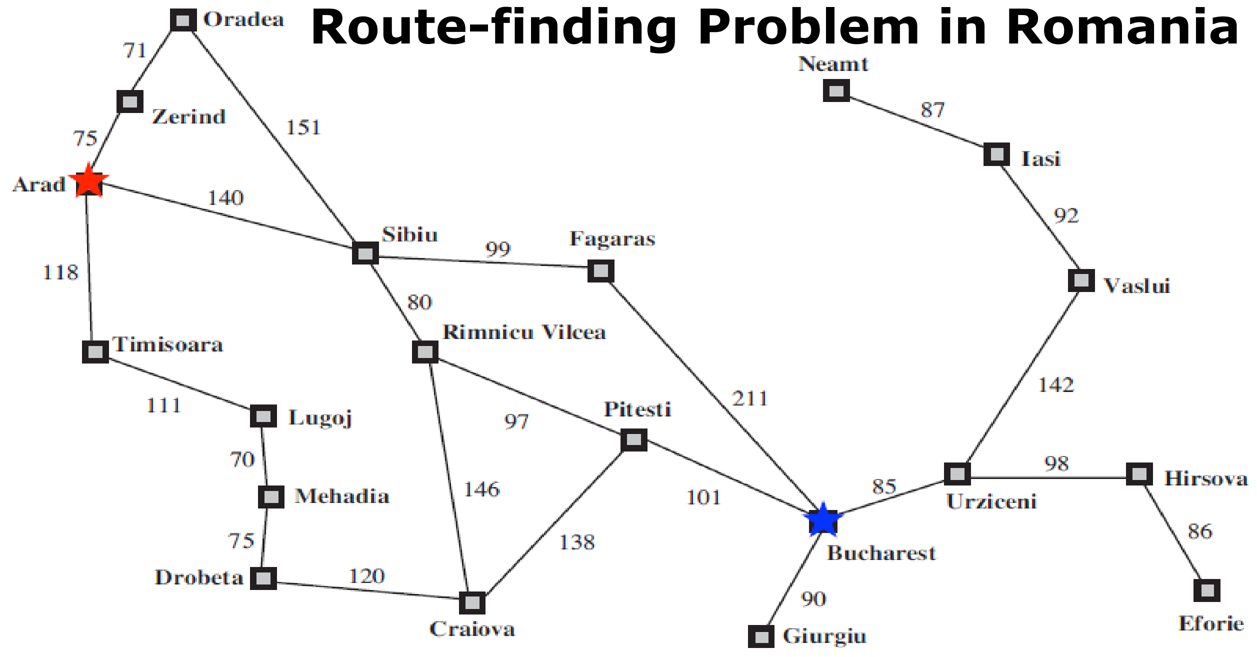
a	b is finite
b	if step costs not less than ε
c	if step costs are all identical
d	if both directions use breadth-first search
e	Yes for finite search spaces
f	$O(b)$ can be eliminated by backtracking search

Informed (Heuristic) Search Strategies

- *An informed search algorithm*
  - ❑ use domain-specific information and context about the location of goals
  - ❑ find solutions more efficiently than an uninformed search algorithm
  - ❑ Rely on a rule of thumb (i.e., prior information) that help us narrow down the search by eliminating the options and unnecessary paths that are obviously wrong. This rule of thumb is called **a heuristic**.
  - ❑ **A heuristic** is a rule or a set of rules used to evaluate a state.
- The method of using **heuristic information** to guide the search is called **a heuristic function** that **estimates** how close a state is to a goal, denoted $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.
- Some search algorithms are:
  - ❑ *Greedy best-first search*
  - ❑ *A* search*

Greedy Best-first Search

Route-finding Problem in Romania



- It calls "Greedy", as it tries to expand the node that **appears** closest to the goal first, assuming that it leads to solution quickly

- *This search makes the locally optimal choice at each step in order to find the global optimum.*

- *The greedy algorithms produce an approximate solution in a reasonable time. This approximate solution is reasonably close to the global optimal solution.*

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

Greedy Best-first Search

- Evaluation function *f(n)* = *h(n)* (heuristic) = *estimate of cost from n to goal, where n is the current node.*

- Example: $h_{SLD}(n)$ = straight-line distance from *n* to Bucharest

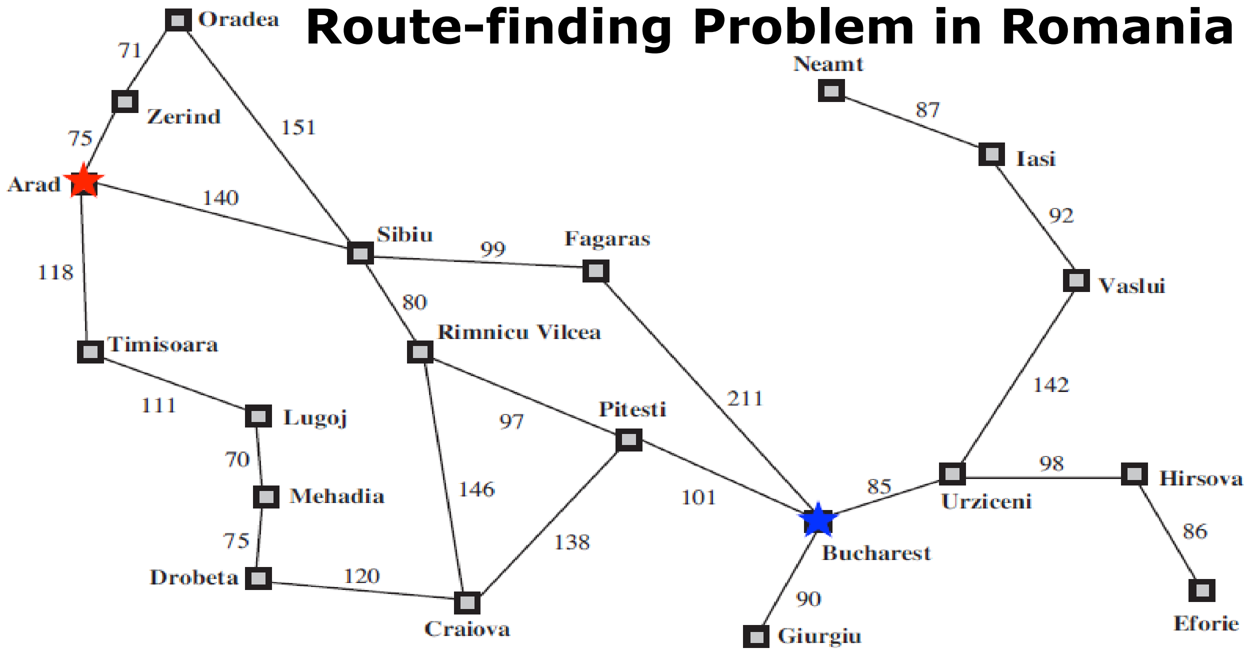
- Romania with step costs in **km**: Straight line distance to Bucharest

- Greedy best-first search using h_{SLD} finds a solution without expanding a node that is not on the solution path.

  - Take routes that may have a shorter distance but might end up taking more time.
  - Take paths that may seem faster in the shorter term but might lead to traffic jams later.
  - ONLY see the next step and not the globally-optimal final solution.

- The **best_first_graph_search(problem, f)** function in search.py, where f is a heuristic function.

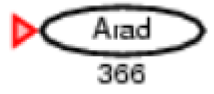
Route-finding Problem in Romania

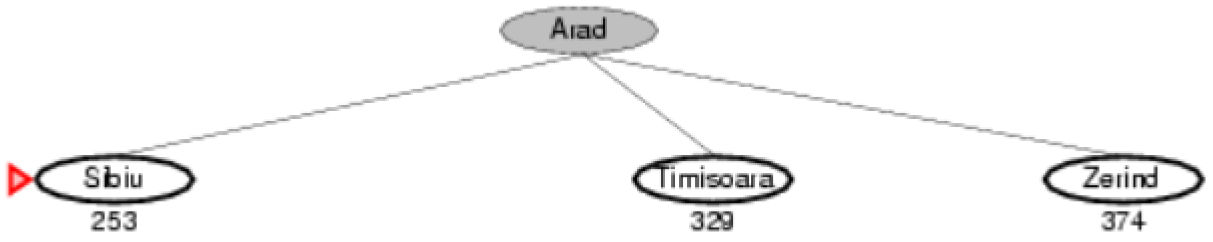


Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

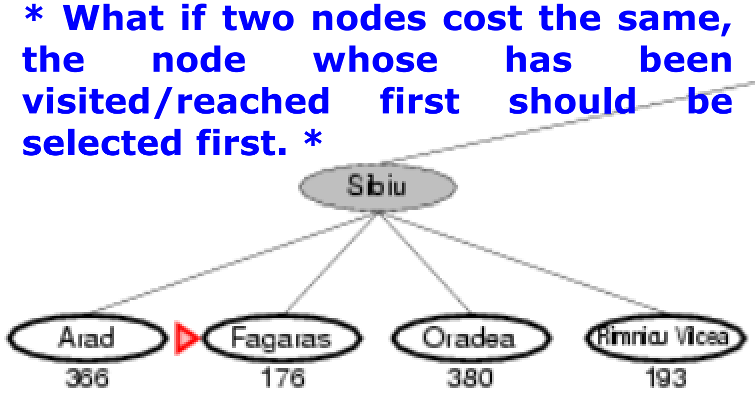
Greedy Best-first Search

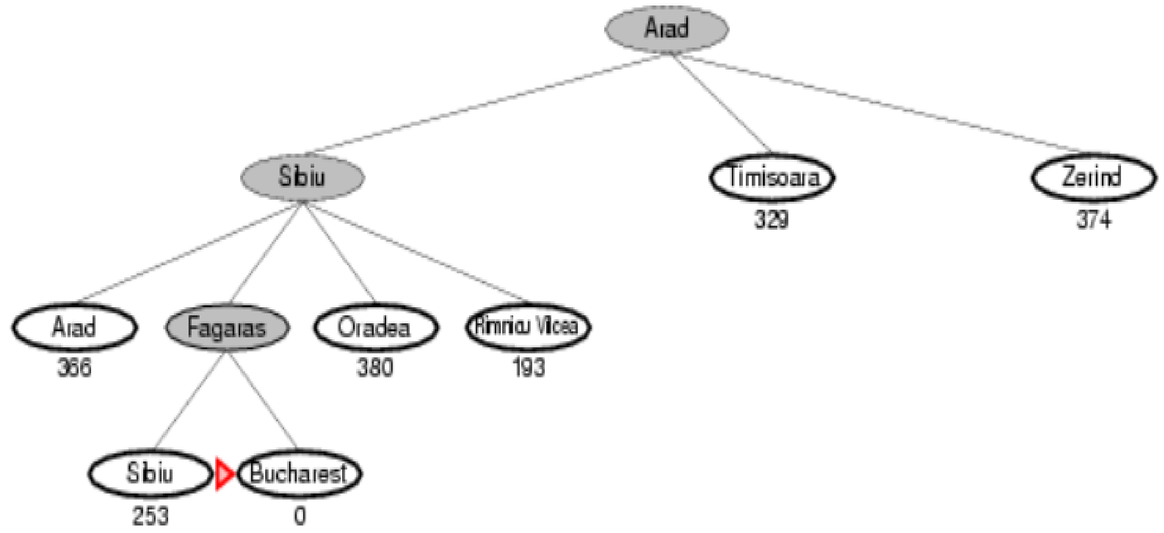
Evaluation function *f(n)* = *h(n)* (heuristic)
= estimate of cost from *n* to *goal*

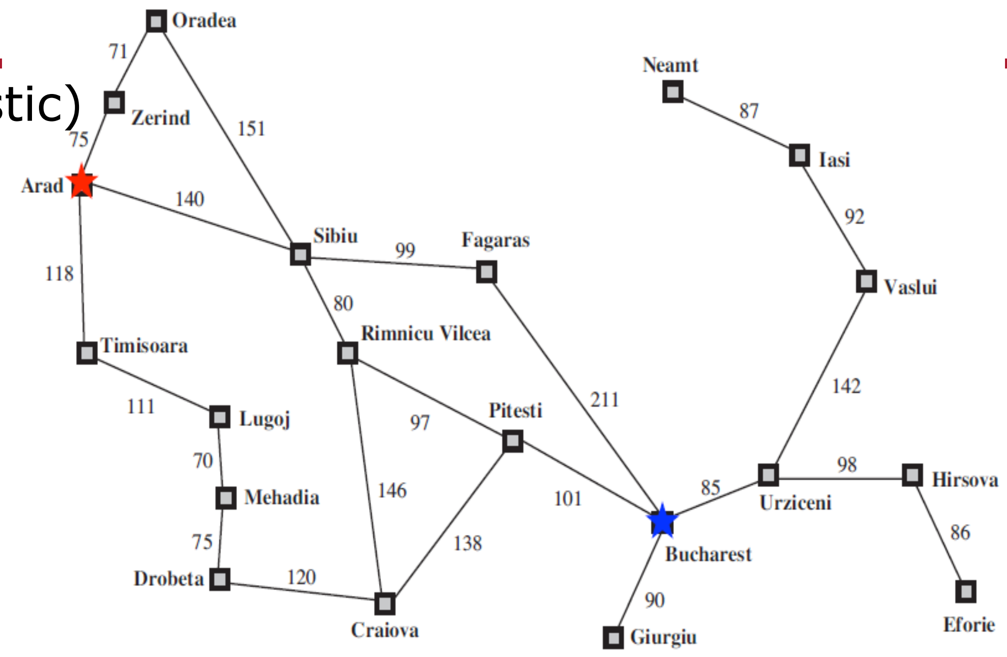




*** What if two nodes cost the same, the node whose has been visited/reached first should be selected first. ***



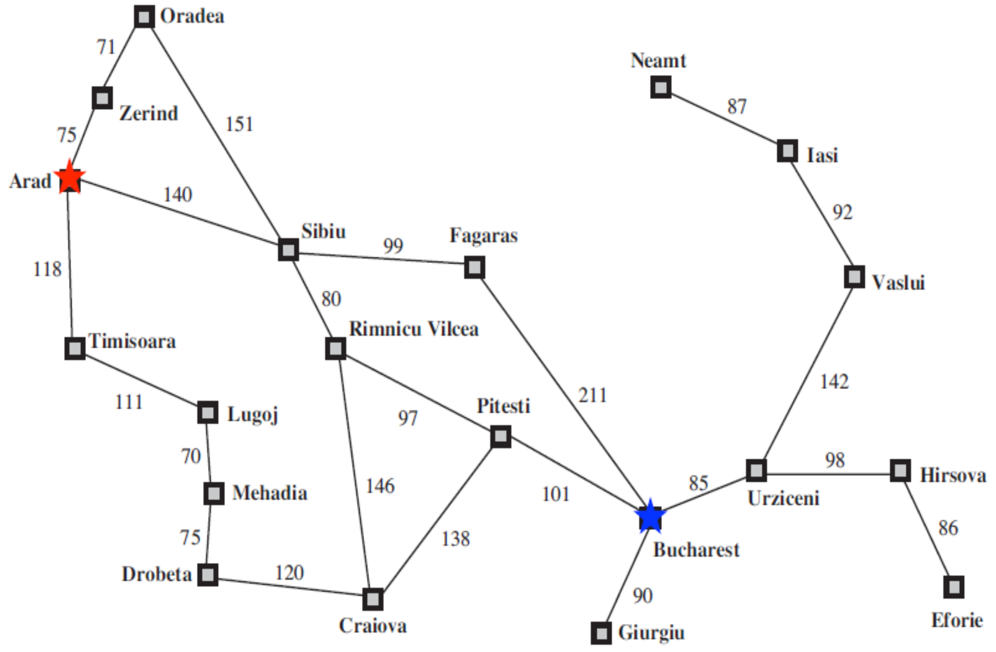




Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

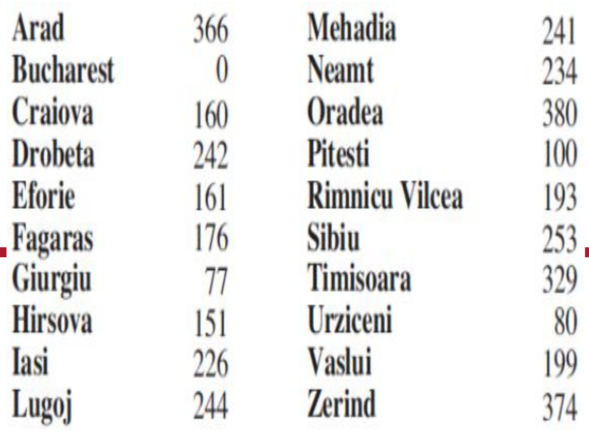
Greedy Best-first Search

```
romania_map = UndirectedGraph(dict(
    Arad=dict(Zerind=75, Sibiu=140, Timisoara=118),
    Bucharest=dict(Urziceni=85, Pitesti=101, Giurgiu=90, Fagaras=211),
    Craiova=dict(Drobeta=120, Rimnicu=146, Pitesti=138),
    Drobeta=dict(Mehadia=75),
    Eforie=dict(Hirsova=86),
    Fagaras=dict(Sibiu=99),
    Hirsova=dict(Urziceni=98),
    Iasi=dict(Vaslui=92, Neamt=87),
    Lugoj=dict(Timisoara=111, Mehadia=70),
    Oradea=dict(Zerind=71, Sibiu=151),
    Pitesti=dict(Rimnicu=97),
    Rimnicu=dict(Sibiu=80),
    Urziceni=dict(Vaslui=142)))

romania_map.locations = dict(350
    Arad=(91, 492), Bucharest=(400, 327), Craiova=(253, 288),
    Drobeta=(165, 299), Eforie=(562, 293), Fagaras=(305, 449),
    Giurgiu=(375, 270), Hirsova=(534, 350), Iasi=(473, 506),
    Lugoj=(165, 379), Mehadia=(168, 339), Neamt=(406, 537),
    Oradea=(131, 571), Pitesti=(320, 368), Rimnicu=(233, 410),
    Sibiu=(207, 457), Timisoara=(94, 410), Urziceni=(456, 350),
    Vaslui=(509, 444), Zerind=(108, 531))
```



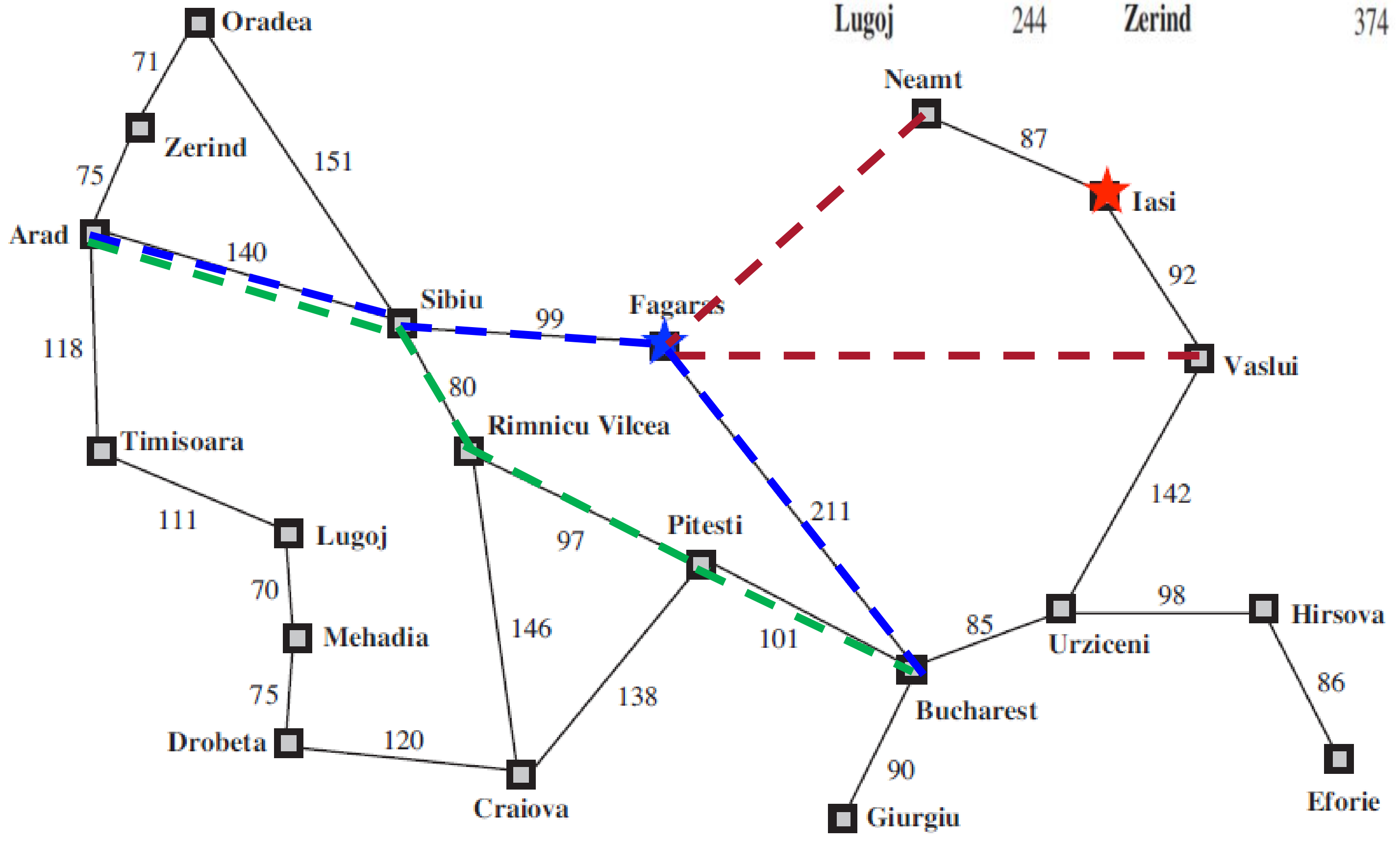
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Greedy Best-first Search



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

- *Completeness*: ***No*** – can get stuck in loops, *e.g.*, Iasi → Neamt → Iasi → Neamt →
- *Optimality*: ***No***, since there might be a shorter path, e.g.,
  - **Blue**: Arad → Sibiu → Fagaras → Bucharest (450)
  - **Green**: Arad → Sibiu → Rimnicu Vilscea → Pitesti → Bucharest (418)
  - 32 km shorter
  - *Might not find the most optimal solution, as it does not explore every single possibility due to its heuristic information, but the search is guaranteed to find a good, efficient solution in a reasonable time.*
- *Time complexity*: *O(b^m) — m is the maximum depth of the search space.*
- *Space complexity*: *O(b^m) — keeps all nodes in memory*



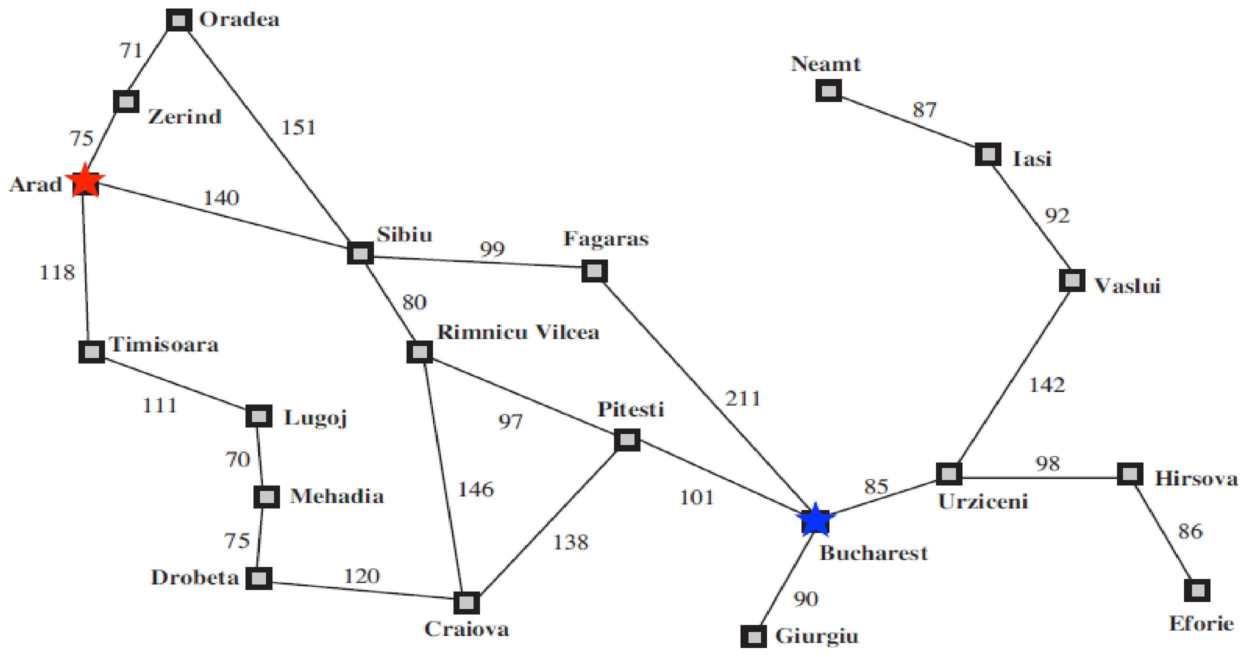
A* Search

- Most widely known form of informed search

- Evaluation function *f(n)* = *g(n)* + *h(n)* = *Uniform-Cost search g(n)* + *Greedy Best-first search h(n), where n is the current node.*

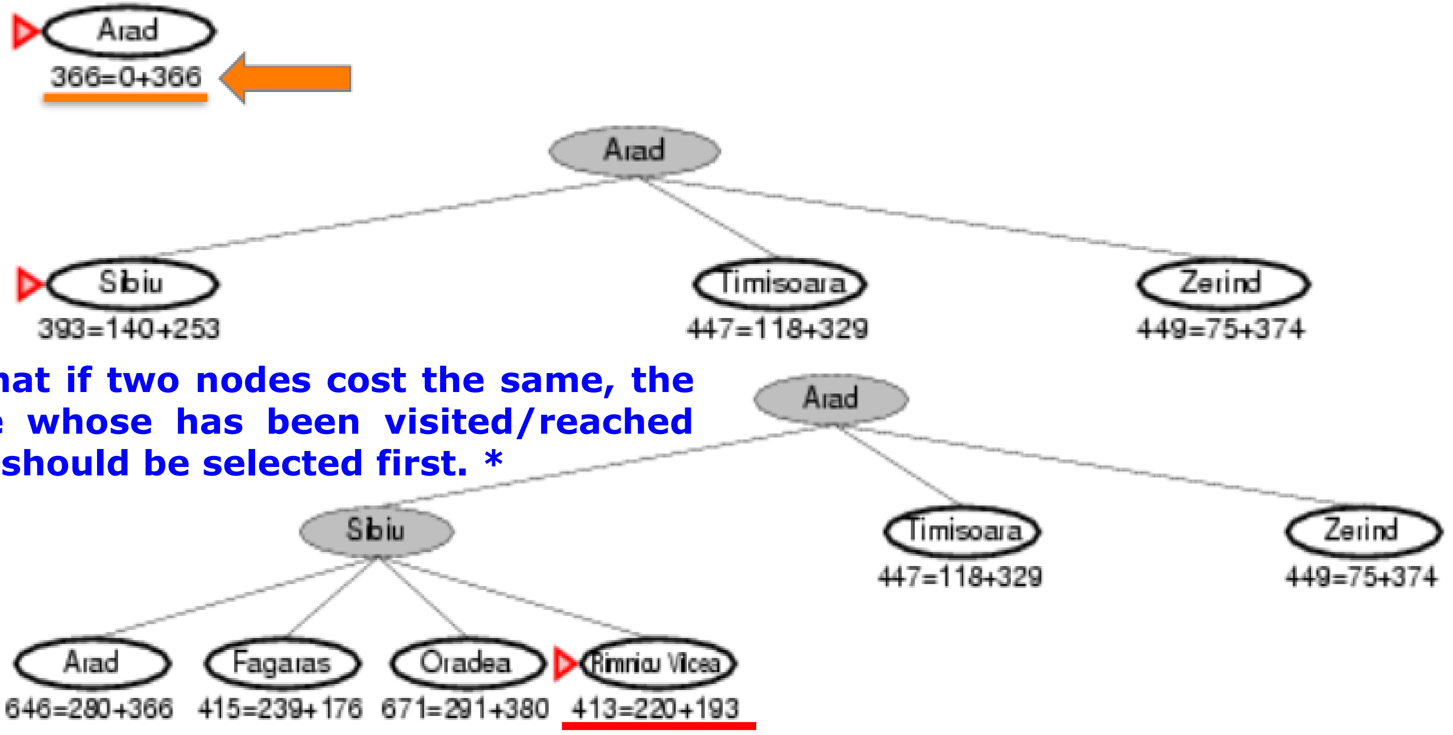
  - ❑ *g(n)* = the exact path cost from the start node to node *n, i.e.,* ***backward cost***
  - ❑ *h(n)* = the heuristic estimated cost of the cheapest path from *n* to goal, i.e., ***forward cost***
  - ❑ *f(n)* = the estimated total cost of path from the start node **through** *n* **to** goal

- The **astar_search(problem, h)** function in search.py, where h is a heuristic function.

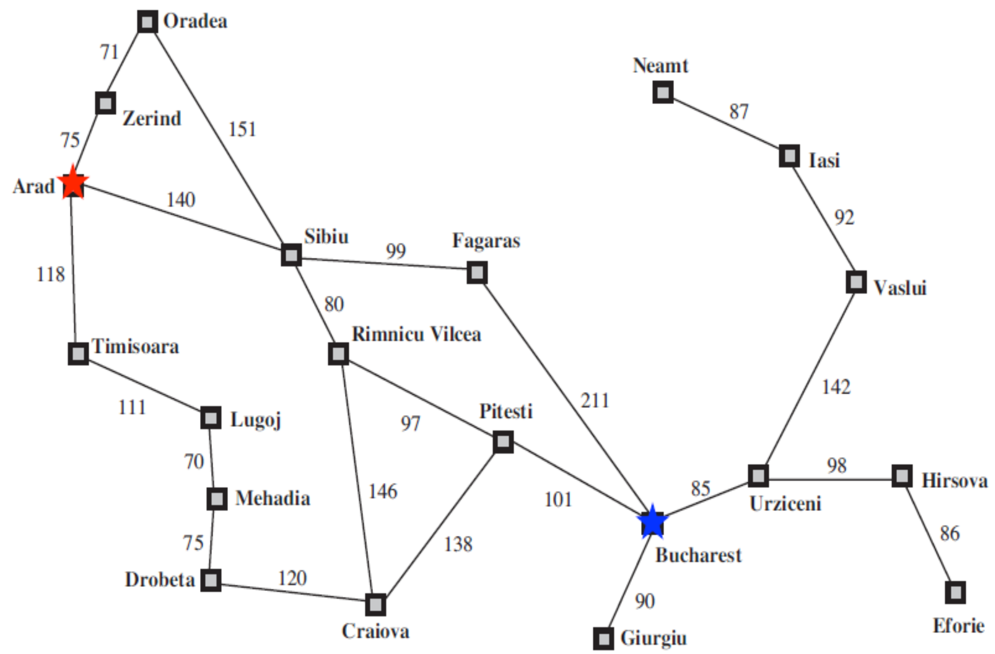


Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

A* Search



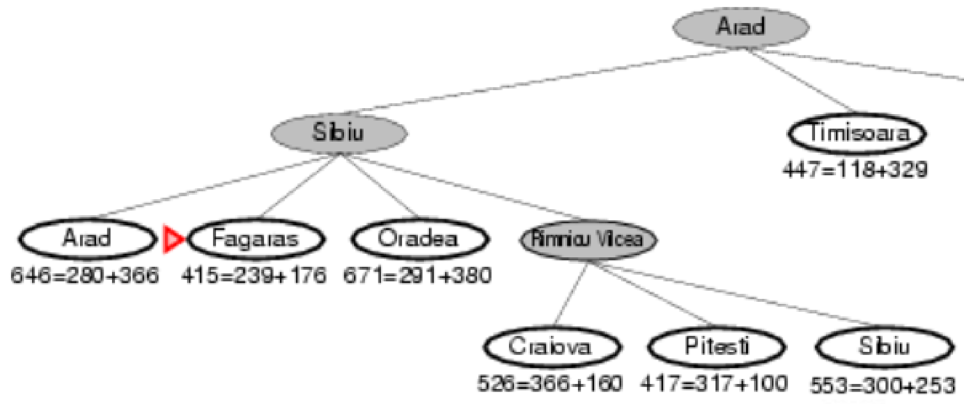
*** What if two nodes cost the same, the node whose has been visited/reached first should be selected first. ***



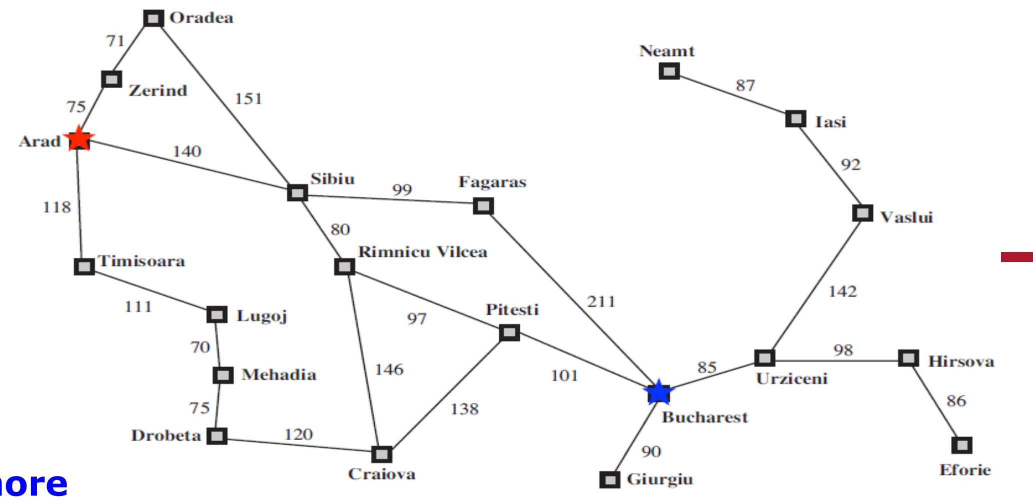
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

- Evaluation function *f(n)* = *g(n)* + *h(n)*
  - ❑ *g(n)* = the exact path cost from the start node to node *n*
  - ❑ *h(n)* = the heuristic estimated cost of the cheapest path from *n* to goal
  - ❑ *f(n)* = the estimated total cost of path from the start node through *n* to goal

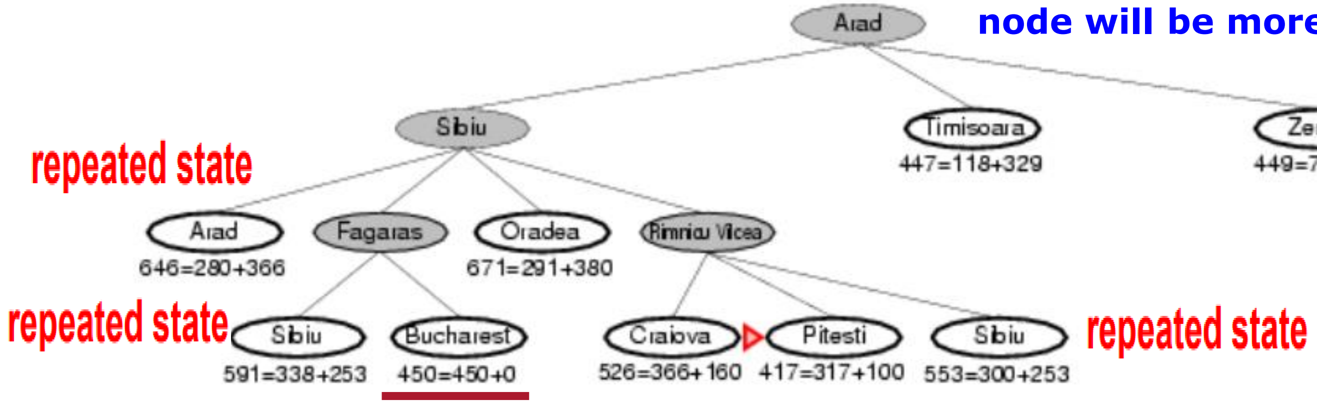
A* Search



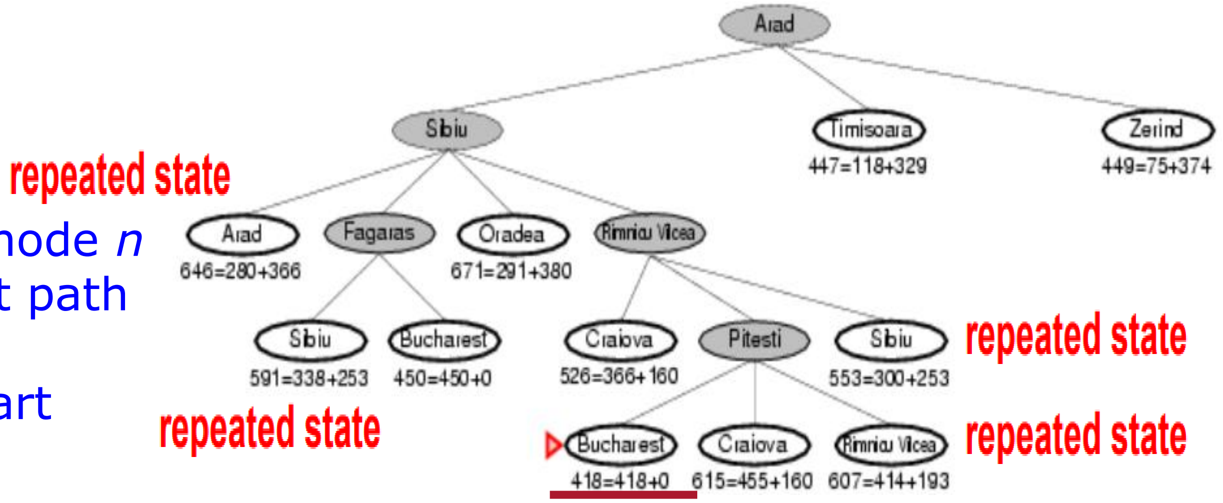
*** What if a node costs more than the current lowest-cost path to the solution, that node can be ignored because that path to a solution via that node will be more expensive. ***



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



- Evaluation function *f(n)* = *g(n)* + *h(n)*
  - ❑ *g(n)* = the exact path cost from the start node to node *n*
  - ❑ *h(n)* = the heuristic estimated cost of the cheapest path from *n* to goal
  - ❑ *f(n)* = the estimated total cost of path from the start node **through** *n* **to** goal



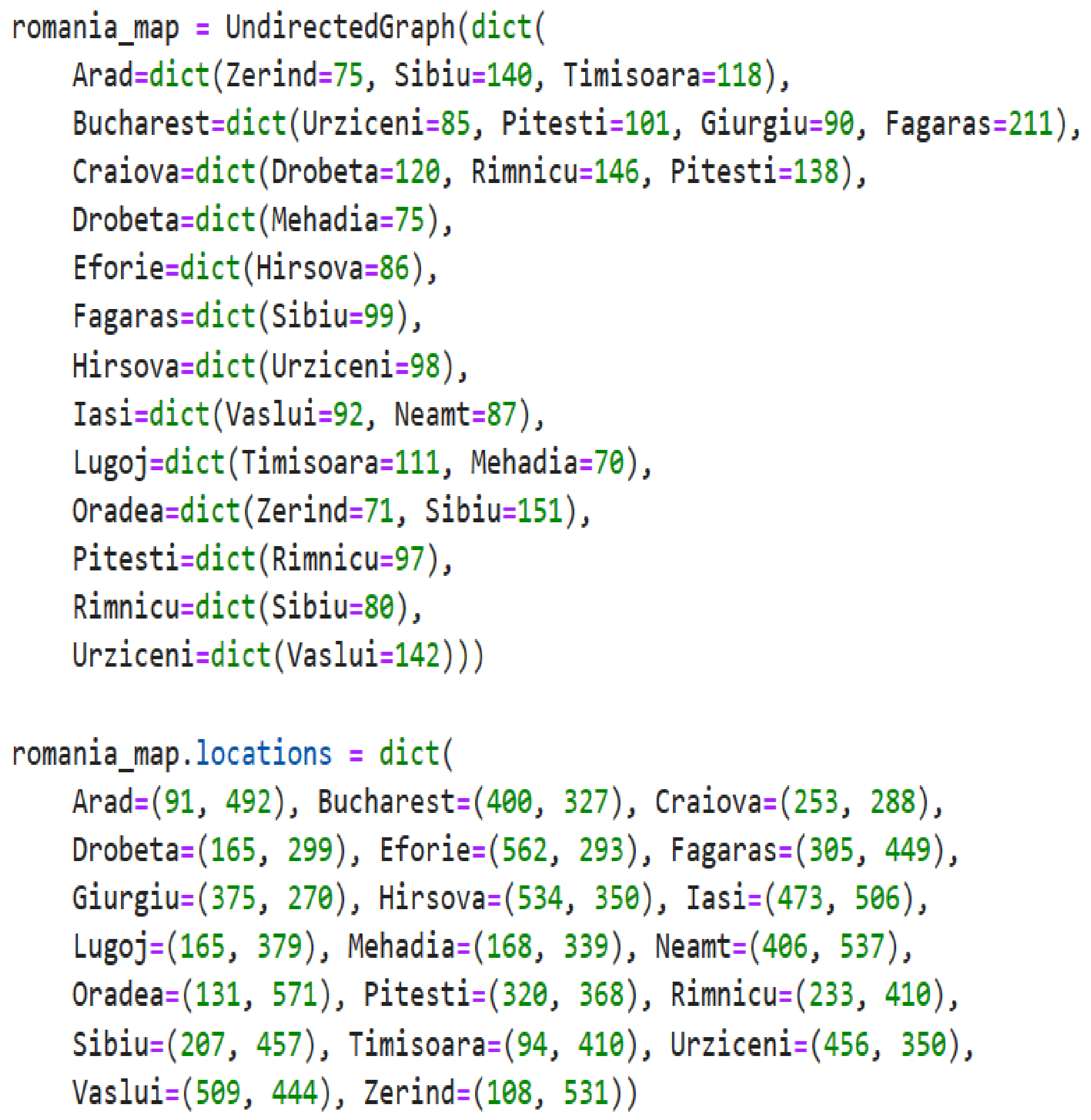
A* Search

Completeness: Yes

Optimality: Yes

Time complexity: *$O(b^d)$ — Exponential, where d is the depth of the least-cost, optimal solution path*

Space complexity: *$O(b^m)$ — keeps all nodes in memory, where m is the maximum depth of the search space*



```
romania_map = UndirectedGraph(dict(
    Arad=dict(Zerind=75, Sibiu=140, Timisoara=118),
    Bucharest=dict(Urziceni=85, Pitesti=101, Giurgiu=90, Fagaras=211),
    Craiova=dict(Drobeta=120, Rimnicu=146, Pitesti=138),
    Drobeta=dict(Mehadia=75),
    Eforie=dict(Hirsova=86),
    Fagaras=dict(Sibiu=99),
    Hirsova=dict(Urziceni=98),
    Iasi=dict(Vaslui=92, Neamt=87),
    Lugoj=dict(Timisoara=111, Mehadia=70),
    Oradea=dict(Zerind=71, Sibiu=151),
    Pitesti=dict(Rimnicu=97),
    Rimnicu=dict(Sibiu=80),
    Urziceni=dict(Vaslui=142)))

romania_map.locations = dict(
    Arad=(91, 492), Bucharest=(400, 327), Craiova=(253, 288),
    Drobeta=(165, 299), Eforie=(562, 293), Fagaras=(305, 449),
    Giurgiu=(375, 270), Hirsova=(534, 350), Iasi=(473, 506),
    Lugoj=(165, 379), Mehadia=(168, 339), Neamt=(406, 537),
    Oradea=(131, 571), Pitesti=(320, 368), Rimnicu=(233, 410),
    Sibiu=(207, 457), Timisoara=(94, 410), Urziceni=(456, 350),
    Vaslui=(509, 444), Zerind=(108, 531))
```