IQRA University

CSC 471 Artificial Intelligence

Lab# 04 Informed Search Algorithms in Artificial Intelligence

Objective:

This experiment introduces the students to informed searches. The experiment also implements an optimal search (A*) while establishing the understanding of Admissible Heuristics.

Name of Student: **FARAZ ALAM**

Roll No: **13948** **Sec.** **Saturday 12:00-14:00 pm**

Date of Experiment: _____

Lab 04: Optimal and Adversarial Search

Informed Search

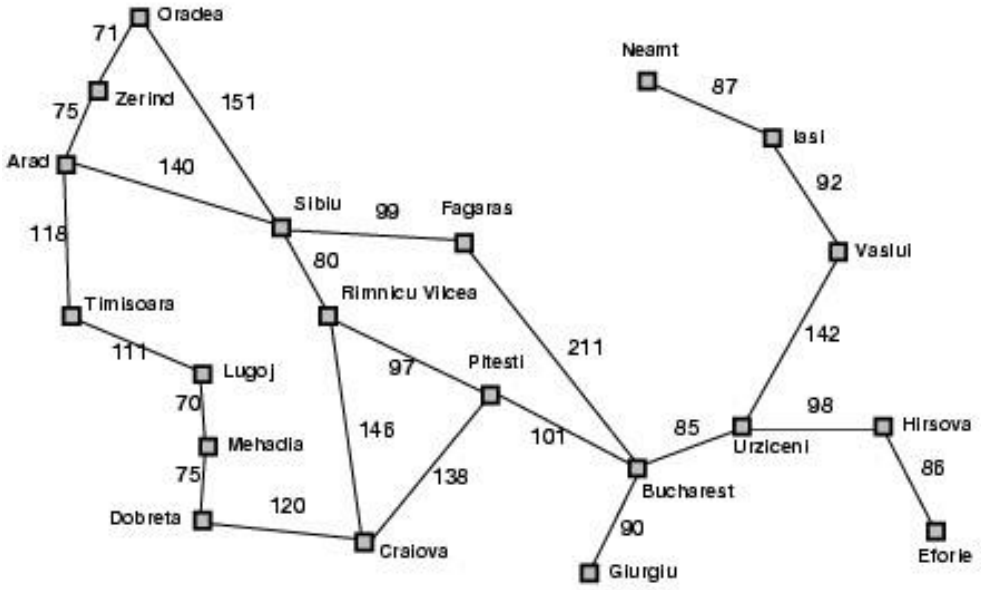
A search strategy which searches the most promising branches of the state-space first can:

- find a solution more quickly,
- find solutions even when there is limited time available,
- often find a better solution, since more profitable parts of the state-space can be examined, while ignoring the unprofitable parts.

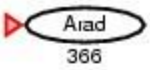
A search strategy which is better than another at identifying the most promising branches of a search-space is said to be more informed.

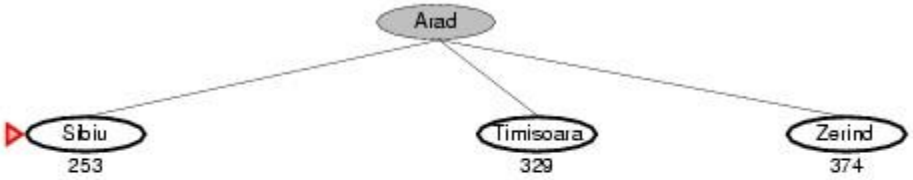
Greedy Best-First Search:

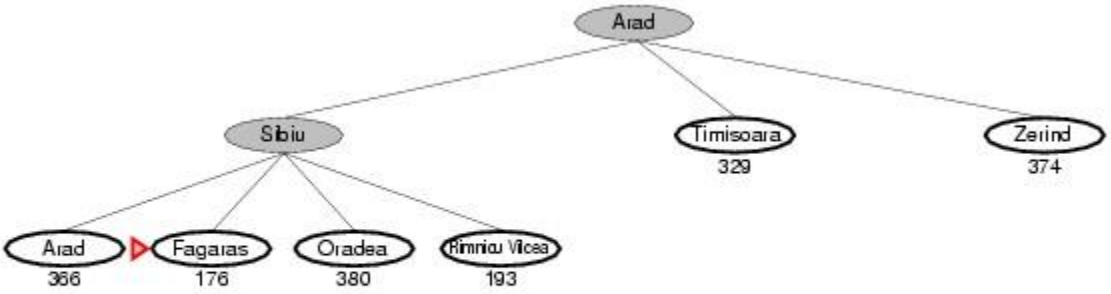
Greedy (Best-First) search is similar in spirit to Depth-First Search. It keeps exploring until it has to back up due to a dead end. Greedy search is not complete and not optimal, but is often fast and efficient, depending on the heuristic function h find a solution more quickly.

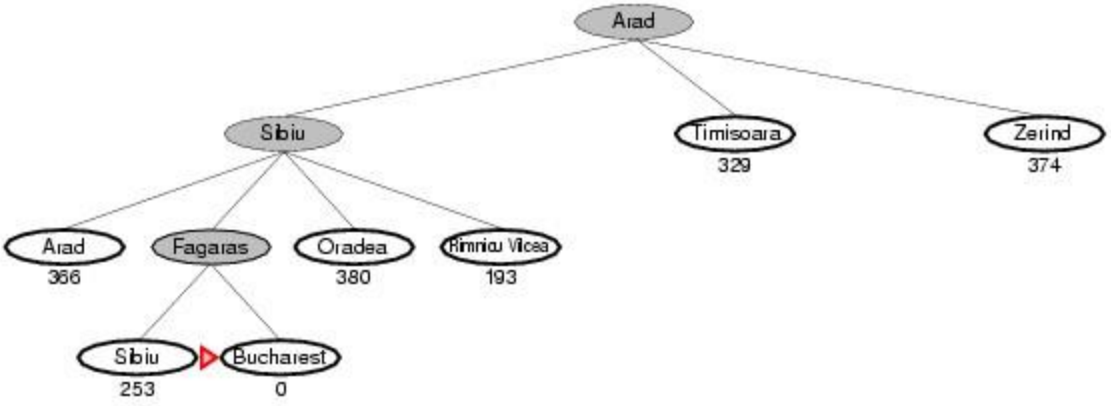


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374


Arad
366


Arad
Sibiu
253
Timisoara
329
Zerind
374


Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vilcea
193


Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
Oradea
380
Rimnicu Vilcea
193
Sibiu
253
Bucharest
0

A* Search

A* is an informed search algorithm, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes $f(n) = g(n) + h(n)$, where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal.

A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.

Admissibility Measures:

Using the evaluation function $f(n) = g(n) + h(n)$ that was introduced in the last section, we may characterize a class of admissible heuristic search strategies. If n is a node in the state space graph, $g(n)$ measures the depth at which that state has been found in the graph, and $h(n)$ is the heuristic estimate of the distance from n to a goal. In this sense $f(n)$ estimates the total cost of the path from the start state through n to the goal state. In determining the properties of admissible heuristics, we define an evaluation function f*: $f^*(n) = g^*(n) + h^*(n)$

where $g^*(n)$ is the cost of the shortest path from the start to node n and h* returns the actual cost of the shortest path from n to the goal. It follows that $f^*(n)$ is the actual cost of the optimal path from a start node to a goal node that passes through node n. The resulting search strategy is admissible. Although oracles such as f* do not exist for most real problems, we would like the evaluation function f to be a close estimate of f*. In algorithm A, $g(n)$, the cost of the current path to state n, is a reasonable estimate of g*, but they may not be equal: $g(n) \geq g^*(n)$. These are equal only if the graph search has discovered the optimal path to state n.

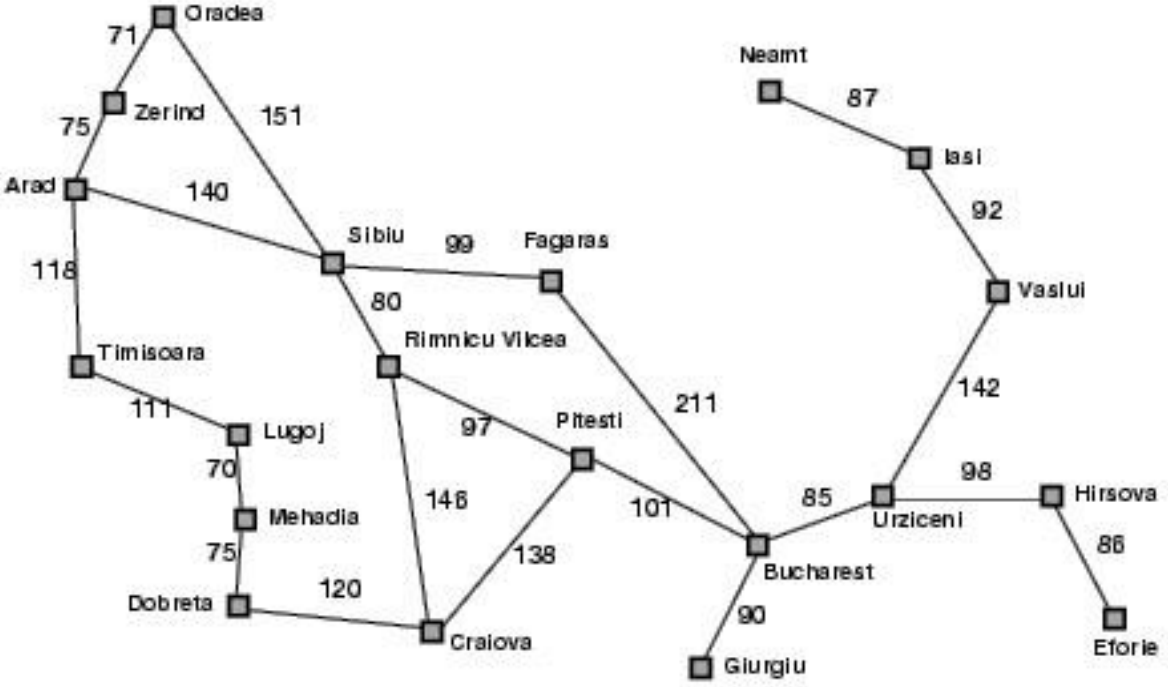
Similarly, we replace $h^*(n)$ with $h(n)$, a heuristic estimate of the minimal cost to a goal state. Although we usually may not compute h*, it is often possible to determine whether or not the heuristic estimate, $h(n)$, is

bounded from above by h*(n), i.e., is always less than or equal to the actual cost of a minimal path. If algorithm A uses an evaluation function f in which h(n) h*(n), it is called algorithm A*.

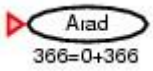
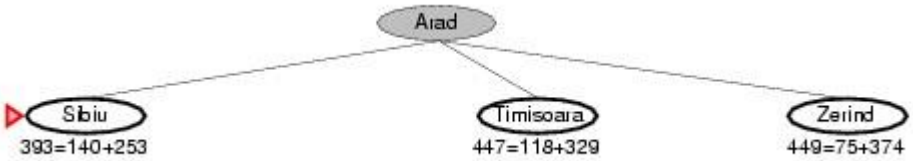
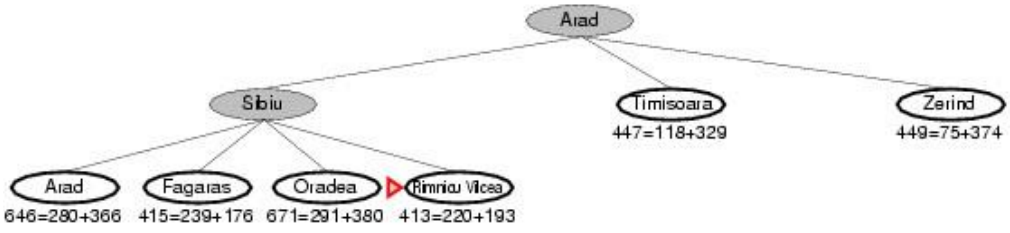
Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty).

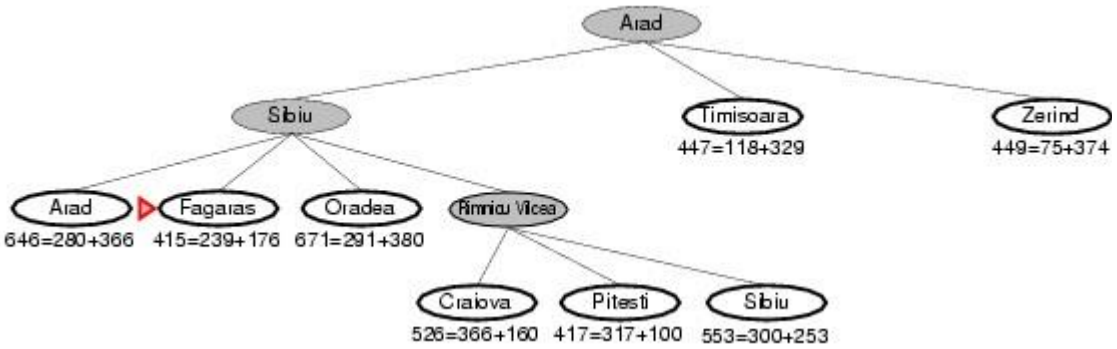
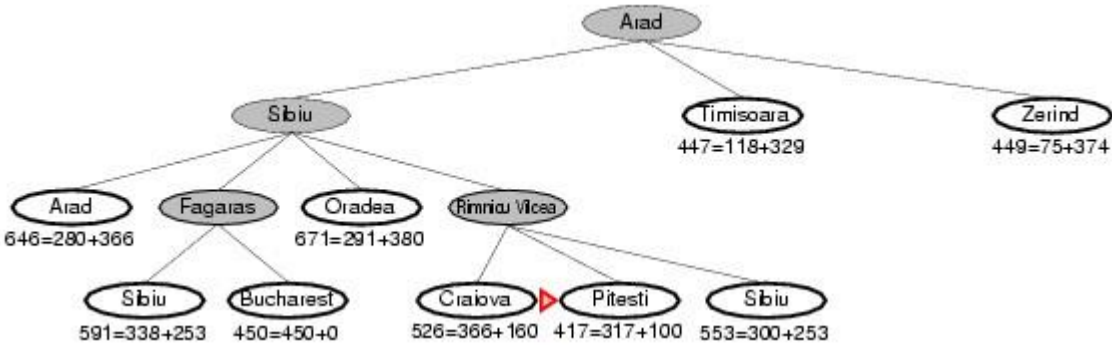
Pseudo Code for A*

```
• Input: – QUEUE: Path only containing root
• Algorithm: –  o WHILE (QUEUE not empty && first path not
    reach goal) DO
            ✦ Remove first path from QUEUE
            ✦ Create paths to all children
            ✦ Reject paths with loops
            ✦ Add paths and sort QUEUE (by f = cost + heuristic)
            ✦ IF QUEUE contains paths: P, Q
            • AND P ends in node Ni && Q contains node Ni AND
            • cost_P ≥ cost_Q
            ✦ THEN remove P  o IF goal reached THEN success ELSE
        failure
```



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Step #	Fringe	Explored
1.	**Arad**	
2.	**1.Sibiu 2.Timisoara 3.Zerind**	
3.	**1. Rimnicu Vilcea 2. Fagaras 3.Timisoara 4.Zerind 5. Arad 6. Oradea**	

4.	**1. Fagaras** **2. Pitesti** **3.Timisoara** **4.Zerind** **5. Craiova** **6. Sibiu 7. Arad** **8. Oradea**	
5.	**1. Pitesti** **2.Timisoara** **3.Zerind** **4. Bucharest** **5. Craiova** **6. Sibiu(553)** **7. Sibiu(591)** **8. Arad** **9. Oradea**	

Student Exercise

Task 1

Implement the solution for the Travel in Romania problem using the following algorithms

- Uniform Cost Search

Uniform Cost Search

In [50]:

```
travel_distances = {
    "Arad": {"Zerind": 75, "Sibiu": 140, "Timisoara": 118},
    "Zerind": {"Arad": 75, "Oradea": 71},
    "Oradea": {"Zerind": 71, "Sibiu": 151},
    "Timisoara": {"Arad": 118, "Lugoj": 111},
    "Sibiu": {"Arad": 140, "Oradea": 151, "Fagaras": 99, "Rimnicu Vilcea": 80},
    "Lugoj": {"Timisoara": 111, "Mehadia": 70},
    "Fagaras": {"Sibiu": 99, "Bucharest": 211},
    "Rimnicu Vilcea": {"Sibiu": 80, "Pitesti": 97, "Craiova": 146},
    "Mehadia": {"Lugoj": 70, "Dobreta": 75},
    "Dobreta": {"Mehadia": 75},
    "Urziceni": {"Bucharest": 85, "Hirsova": 98, "Vaslui": 142},
    "Hirsova": {"Urziceni": 98, "Eforie": 86},
    "Vaslui": {"Urziceni": 142, "Iasi": 92},
    "Iasi": {"Vaslui": 92, "Neamt": 87},
    "Neamt": {"Iasi": 87},
    "Pitesti": {"Rimnicu Vilcea": 97, "Craiova": 138, "Bucharest": 101},
}

class Node:
    def __init__(self, state, parent, move, path_cost):
        self.state = state
        self.parent = parent
        self.move = move
        self.path_cost = path_cost

queue = []
start_state = Node('Arad', None, None, 0)
goal_node = 'Bucharest'
queue.append(start_state)
visited = set()

def print_path(node):
    path = []
    while node.parent:
        path.append(node.parent.state)
        node = node.parent
    return path[::-1]

```

```python
while queue:
    current_node = queue.pop(0)
    print('Expanding node : ', current_node.state)

    if current_node.state == goal_node:
        print('Solution found with path cost', current_node.path_cost)
        route = print_path(current_node)
        print(route)
        break

    visited.add(current_node.state)

    if current_node.state not in travel_distances:
        continue

    for neighbor in travel_distances[current_node.state]:
        if neighbor in visited:
            continue

        cost_to_neighbor = travel_distances[current_node.state][neighbor]
        new_path_cost = current_node.path_cost + cost_to_neighbor

        neighbor_in_queue = next((node for node in queue if node.state == neighbor), None)

        if neighbor_in_queue:
            if new_path_cost < neighbor_in_queue.path_cost:
                neighbor_in_queue.parent = current_node
                neighbor_in_queue.path_cost = new_path_cost
        else:
            state = Node(neighbor, current_node, travel_distances[current_node.state][neighbor], new_path_cost)
            queue.append(state)
    queue.sort(key=lambda x: x.path_cost)

else:
    print('Failure')
```

```
Expanding node :  Arad
Expanding node :  Zerind
Expanding node :  Timisoara
Expanding node :  Sibiu
Expanding node :  Oradea
Expanding node :  Rimnicu Vilcea
Expanding node :  Lugoj
Expanding node :  Fagaras
Expanding node :  Mehadia
Expanding node :  Pitesti
Expanding node :  Craiova
Expanding node :  Dobreta
Expanding node :  Bucharest
Solution found with path cost 418
['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti']
```

- Greedy Best First Search

Greedy Best First Search

In [60]:

```
travel_distances = {
    "Arad": {"Zerind": 75, "Sibiu": 140, "Timisoara": 118},
    "Zerind": {"Arad": 75, "Oradea": 71},
    "Oradea": {"Zerind": 71, "Sibiu": 151},
    "Timisoara": {"Arad": 118, "Lugoj": 111},
    "Sibiu": {"Arad": 140, "Oradea": 151, "Fagaras": 99, "Rimnicu Vilcea": 80},
    "Lugoj": {"Timisoara": 111, "Mehadia": 70},
    "Fagaras": {"Sibiu": 99, "Bucharest": 211},
    "Rimnicu Vilcea": {"Sibiu": 80, "Pitesti": 97, "Craiova": 146},
    "Mehadia": {"Lugoj": 70, "Dobreta": 75},
    "Dobreta": {"Mehadia": 75},
    "Urziceni": {"Bucharest": 85, "Hirsova": 98, "Vaslui": 142},
    "Hirsova": {"Urziceni": 98, "Eforie": 86},
    "Vaslui": {"Urziceni": 142, "Iasi": 92},
    "Iasi": {"Vaslui": 92, "Neamt": 87},
    "Neamt": {"Iasi": 87},
    "Pitesti": {"Rimnicu Vilcea": 97, "Craiova": 138, "Bucharest": 101},
}

heuristics = {
    "Arad": 366,
    "Bucharest": 0,
    "Craiova": 160,
    "Dobreta": 242,
    "Eforie": 161,
    "Fagaras": 176,
    "Giurgiu": 77,
    "Hirsova": 151,
    "Iasi": 226,
    "Lugoj": 244,
    "Mehadia": 241,
    "Neamt": 234,
    "Oradea": 380,
    "Pitesti": 100,
    "Rimnicu Vilcea": 193,
    "Sibiu": 253,
    "Timisoara": 329,
```

```
    "Urziceni": 80,
    "Vaslui": 199,
    "Zerind": 374
}

class Node:
    def __init__(self, state, parent, move, heuristic_value):
        self.state = state
        self.parent = parent
        self.move = move
        self.heuristic_value = heuristic_value

queue = []
start_state = Node('Arad', None, None, heuristics['Arad'])  # Use heuristic value for start state
goal_node = 'Bucharest'
queue.append(start_state)
visited = set()

def print_path(node):
    path = []
    while node.parent:
        path.append(node.state)
        node = node.parent
    return path[::-1]

while queue:
    queue.sort(key=lambda x: x.heuristic_value, reverse=True)
    current_node = queue.pop(0)
    print('Expanding node:', current_node.state)

    if current_node.state == goal_node:
        print('Solution found with heuristic value:', current_node.heuristic_value)
        route = print_path(current_node)
        print(route)
        break

    visited.add(current_node.state)

    if current_node.state not in travel_distances:
        continue

    for neighbor in travel_distances[current_node.state]:
        if neighbor in visited:
            continue
        heuristic_value = heuristics[neighbor]
        new_heuristic_value = heuristic_value

        neighbor_in_queue = any(node.state == neighbor for node in queue)

        if not neighbor_in_queue:
            state = Node(neighbor, current_node, travel_distances[current_node.state][neighbor], new_heuristic_value)
            queue.append(state)

else:
    print('Failure')
```

```
Expanding node: Arad
Expanding node: Zerind
Expanding node: Oradea
Expanding node: Timisoara
Expanding node: Sibiu
Expanding node: Lugoj
Expanding node: Mehadia
Expanding node: Dobreta
Expanding node: Rimnicu Vilcea
Expanding node: Fagaras
Expanding node: Craiova
Expanding node: Pitesti
Expanding node: Bucharest
Solution found with heuristic value: 0
['Sibiu', 'Fagaras', 'Bucharest']
```

- A* Search

A* Search

In [61]:

```
travel_distances = {
    "Arad": {"Zerind": 75, "Sibiu": 140, "Timisoara": 118},
    "Zerind": {"Arad": 75, "Oradea": 71},
    "Oradea": {"Zerind": 71, "Sibiu": 151},
    "Timisoara": {"Arad": 118, "Lugoj": 111},
    "Sibiu": {"Arad": 140, "Oradea": 151, "Fagaras": 99, "Rimnicu Vilcea": 80},
    "Lugoj": {"Timisoara": 111, "Mehadia": 70},
    "Fagaras": {"Sibiu": 99, "Bucharest": 211},
    "Rimnicu Vilcea": {"Sibiu": 80, "Pitesti": 97, "Craiova": 146},
    "Mehadia": {"Lugoj": 70, "Dobreta": 75},
    "Dobreta": {"Mehadia": 75},
    "Urziceni": {"Bucharest": 85, "Hirsova": 98, "Vaslui": 142},
    "Hirsova": {"Urziceni": 98, "Eforie": 86},
    "Vaslui": {"Urziceni": 142, "Iasi": 92},
    "Iasi": {"Vaslui": 92, "Neamt": 87},
    "Neamt": {"Iasi": 87},
    "Pitesti": {"Rimnicu Vilcea": 97, "Craiova": 138, "Bucharest": 101},
}

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

class Node:
    def __init__(self, state, parent, move, path_cost, heuristic_value):
        self.state = state
        self.parent = parent
        self.move = move
        self.path_cost = path_cost
        self.heuristic_value = heuristic_value

    def total_cost(self):
        return self.path_cost + self.heuristic_value

queue = []
start_state = Node('Arad', None, None, 0, heuristics['Arad'])
goal_node = 'Bucharest'
queue.append(start_state)
visited = set()

def print_path(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

while queue:
    queue.sort(key=lambda x: x.total_cost())
    current_node = queue.pop(0)
    print('Expanding node:', current_node.state)

    if current_node.state == goal_node:
        print('Solution found with path cost:', current_node.path_cost)
        route = print_path(current_node)
        print(route)
        break

    visited.add(current_node.state)

```

```
    if current_node.state not in travel_distances:
        continue

    for neighbor in travel_distances[current_node.state]:
        if neighbor in visited:
            continue
        cost_to_neighbor = travel_distances[current_node.state][neighbor]
        new_path_cost = current_node.path_cost + cost_to_neighbor
        heuristic_value = heuristics[neighbor]
        new_heuristic_value = heuristic_value
        total_cost = new_path_cost + new_heuristic_value

        neighbor_in_queue = next((node for node in queue if node.state == neighbor), None)

        if neighbor_in_queue:
            if total_cost < neighbor_in_queue.total_cost():
                neighbor_in_queue.parent = current_node
                neighbor_in_queue.path_cost = new_path_cost
                neighbor_in_queue.heuristic_value = new_heuristic_value
        else:
            state = Node(neighbor, current_node, cost_to_neighbor, new_path_cost, new_heuristic_value)
            queue.append(state)

else:
    print('Failure')

```

```
Expanding node: Arad
Expanding node: Sibiu
Expanding node: Rimnicu Vilcea
Expanding node: Fagaras
Expanding node: Pitesti
Expanding node: Bucharest
Solution found with path cost: 418
['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest']
```