**DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING**

**COLLEGE OF E&ME, NUST, RAWALPINDI**

# AI & Decision Support Systems

## Lab Report #5

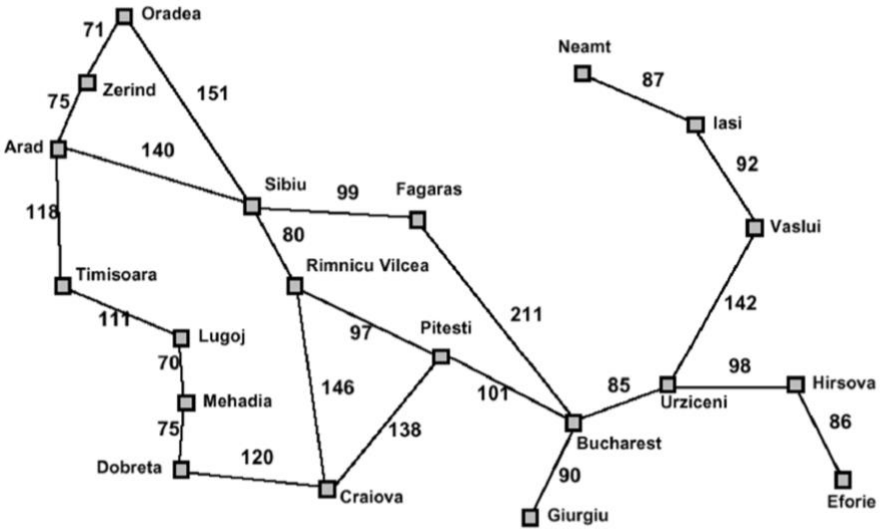
**Student Name: Nawab Aarij Imam**

**Degree/ Syndicate: 43 CE - A**

# Task1:

1. **Implement Priority Queue and implement A* algorithm in python for following graph:**

***Start Node: Arad, Goal: Bucharest***




| Straight−line distance to Bucharest | |
|---|---|
| Arad | 366 |
| Bucharest | 0 |
| Craiova | 160 |
| Dobreta | 242 |
| Eforie | 161 |
| Fagaras | 178 |
| Giurgiu | 77 |
| Hirsova | 151 |
| Iasi | 226 |
| Lugoj | 244 |
| Mehadia | 241 |
| Neamt | 234 |
| Oradea | 380 |
| Pitesti | 98 |
| Rimnicu Vilcea | 193 |
| Sibiu | 253 |
| Timisoara | 329 |
| Urziceni | 80 |
| Vaslui | 199 |
| Zerind | 374 |

**Graph 1**

# Code:

```
import heapq
from typing import Dict, List, Tuple

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def isEmpty(self) -> bool:
        return len(self.elements) == 0

    def putItem(self, item, priority: float):
        heapq.heappush(self.elements, (priority, item))

    def getItem(self):
        return heapq.heappop(self.elements)[1]

class Graph:
```

```
    def __init__(self):
        self.edges: Dict[str, List[Tuple[str, int]]] = {}
        self.heuristics: Dict[str, int] = {}

    def addEdge(self, fromNode: str, toNode: str, cost: int):
        if fromNode not in self.edges:
            self.edges[fromNode] = []
        self.edges[fromNode].append((toNode, cost))

    def addHeuristic(self, node: str, value: int):
        self.heuristics[node] = value

def reconstructPath(cameFrom: Dict[str, str], start: str, goal: str) -> List[str]:
    current = goal
    path = []
    while current != start:
        path.append(current)
        current = cameFrom[current]
    path.append(start)
    path.reverse()
    return path

def aStarSearch(graph: Graph, start: str, goal: str) -> Tuple[List[str], int]:
    frontier = PriorityQueue()
    frontier.putItem(start, 0)
    cameFrom: Dict[str, str] = {}
    costSoFar: Dict[str, int] = {}
    cameFrom[start] = None
    costSoFar[start] = 0

    while not frontier.isEmpty():
        current = frontier.getItem()

        if current == goal:
            break
```

```
        for nextNode, cost in graph.edges.get(current, []):
            newCost = costSoFar[current] + cost
            if nextNode not in costSoFar or newCost < costSoFar[nextNode]:
                costSoFar[nextNode] = newCost
                priority = newCost + graph.heuristics.get(nextNode, 0)
                frontier.putItem(nextNode, priority)
                cameFrom[nextNode] = current

    path = reconstructPath(cameFrom, start, goal)
    return path, costSoFar[goal]


graph = Graph()


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

for edge in edges:
    graph.addEdge(edge[0], edge[1], edge[2])
    graph.addEdge(edge[1], edge[0], edge[2])


heuristics = {
    "Arad": 366, "Bucharest": 0, "Craiova": 160, "Dobreta": 242,
```

```
    "Eforie": 161, "Fagaras": 178, "Giurgiu": 77, "Hirsova": 151,
    "Iasi": 226, "Lugoj": 244, "Mehadia": 241, "Neamt": 234,
    "Oradea": 380, "Pitesti": 98, "Rimnicu Vilcea": 193, "Sibiu": 253,
    "Timisoara": 329, "Urziceni": 80, "Vaslui": 199, "Zerind": 374
}

for node, value in heuristics.items():
    graph.addHeuristic(node, value)


start = "Arad"
goal = "Bucharest"
path, cost = aStarSearch(graph, start, goal)
print(f"Path from {start} to {goal}: {' -> '.join(path)}")
print(f"Total cost: {cost}")
```

## Output:

```
❯ python task1.py
Path from Arad to Bucharest: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
Total cost: 418
```

## Task2:

**2. Design a program that uses the A* algorithm to find the optimal path through a maze. The agent starts at a designated position and must find its way to the exit. Here 0 is a walkable path and 1 is a blocked path.**
**Hint: (*Manhattan distance will be used to calculate heuristic.*)**
**The maze is given below:**

**0 0 0 1 0**

**1 1 0 1 0**

**0 0 0 1 0**

**0 1 1 0 0**

**S 0 0 0 E**

## Code:

```python
import heapq
from typing import Dict, List, Tuple
import task1 as t1

def manhattanDistance(a: Tuple[int, int], b: Tuple[int, int]) -> int:
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def createGraphFromMaze(maze: List[str]) -> Tuple[t1.Graph, Tuple[int, int], Tuple[int, int]]:
    graph = t1.Graph()
    rows, cols = len(maze), len(maze[0])
    start, end = None, None

    for i in range(rows):
        for j in range(cols):
            if maze[i][j] in 'OSE':
                node = f"{i},{j}"
                if maze[i][j] == 'S':
                    start = (i, j)
                elif maze[i][j] == 'E':
                    end = (i, j)
                for di, dj in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
                    ni, nj = i + di, j + dj
                    if 0 <= ni < rows and 0 <= nj < cols and maze[ni][nj] in 'OSE':
                        neighbor = f"{ni},{nj}"
                        graph.addEdge(node, neighbor, 1)

    return graph, start, end

def solveMaze(maze: List[str]) -> Tuple[List[Tuple[int, int]], int]:
    graph, start, end = createGraphFromMaze(maze)
    rows, cols = len(maze), len(maze[0])

    for i in range(rows):
        for j in range(cols):
            if maze[i][j] in 'OSE':
                node = f"{i},{j}"
```

```
            graph.addHeuristic(node, manhattanDistance((i, j), end))

    start_node = f"{start[0]},{start[1]}"
    end_node = f"{end[0]},{end[1]}"

    path, cost = t1.aStarSearch(graph, start_node, end_node)

    coord_path = [tuple(map(int, node.split(','))) for node in path]

    return coord_path, cost

maze = [
    "00010",
    "11010",
    "00010",
    "01100",
    "S000E"
]

optimal_path, total_cost = solveMaze(maze)
print(f"Optimal path: {optimal_path}")
print(f"Total cost: {total_cost}")
```

## Output:

```
Optimal path: [(4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]
Total cost: 4
```