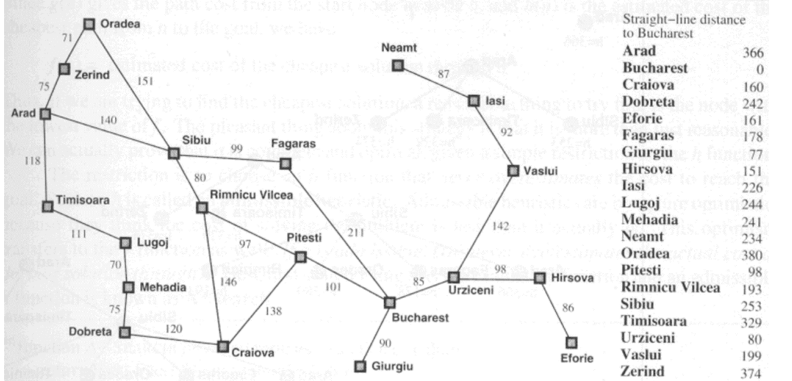
**Task 1: Implement A\* Search**

**Objectives:**

The program implements the A\* search algorithm to find the optimal path between Arad and Bucharest in the given map of Romania.

**Task Name:**

Implement A\* search to find an optimal path in between Arad and Bucharest in the given figure.



**Code:**

import heapq

# *Define the graph as a dictionary where each node has a dictionary of connected nodes and their distances*

graph = {

    'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},

    'Zerind': {'Arad': 75, 'Oradea': 71},

    'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},

    'Zerind': {'Arad': 75, 'Oradea': 71},

    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},

    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},

    'Drobeta': {'Mehadia': 75, 'Craiova': 120},

    'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},

    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},

    'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},

    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},

    'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},

    'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},

    'Giurgiu': {'Bucharest': 90},

    'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},

    'Hirsova': {'Urziceni': 98, 'Eforie': 86},

    'Eforie': {'Hirsova': 86},

    'Vaslui': {'Urziceni': 142, 'Iasi': 92},

    'Iasi': {'Vaslui': 92, 'Neamt': 87},

    'Neamt': {'Iasi': 87}

}

# *Define a heuristic function. Here, straight-line distances to Bucharest are used as heuristics.*

# *The values are estimated and may not be accurate.*

heuristic = {

    'Arad': 366, 'Zerind': 374, 'Oradea': 380, 'Timisoara': 329,

    'Lugoj': 244, 'Mehadia': 241, 'Drobeta': 242, 'Craiova': 160,

    'Sibiu': 253, 'Rimnicu Vilcea': 193, 'Fagaras': 178, 'Pitesti': 98,

    'Bucharest': 0, 'Giurgiu': 77, 'Urziceni': 80, 'Hirsova': 151,

'Eforie': 161, 'Vaslui': 199, 'Iasi': 226, 'Neamt': 234

}

def a\_star\_search(graph, start, goal):

    # *Priority queue to hold the nodes to explore*

    open\_list = []

    heapq.heappush(open\_list, (0 + heuristic[start], 0, start, []))

    # *Set to hold explored nodes*

    closed\_list = set()

    while open\_list:

        # *Get the node with the lowest f(n) = g(n) + h(n)*

        \_, cost, current\_node, path = heapq.heappop(open\_list)

        if current\_node in closed\_list:

            continue

        # *Add current node to the closed list*

        closed\_list.add(current\_node)

        # *Path to current node*

        path = path + [current\_node]

        # *Check if we have reached the goal*

        if current\_node == goal:

            return path, cost

        # *Explore neighbors*

        for neighbor, distance in graph[current\_node].items():

            if neighbor not in closed\_list:

                total\_cost = cost + distance

  heapq.heappush(open\_list, (total\_cost + heuristic[neighbor], total\_cost, neighbor, path))

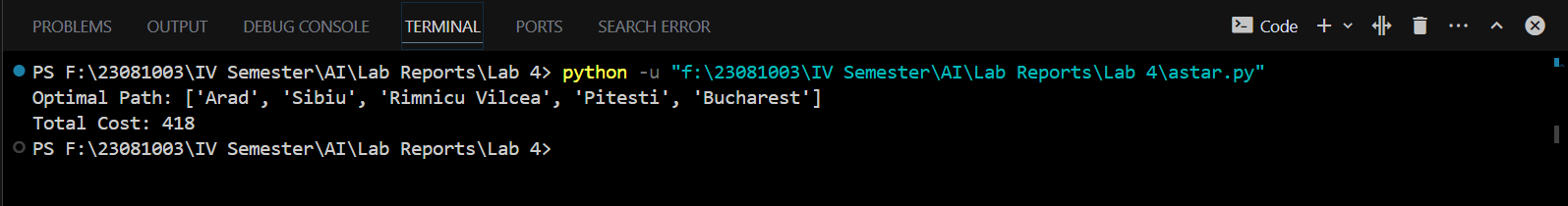
    return None, None

# *Execute A\* search from Arad to Bucharest*

path, cost = a\_star\_search(graph, 'Arad', 'Bucharest')

print("Optimal Path:", path)

print("Total Cost:", cost)

**Output:**

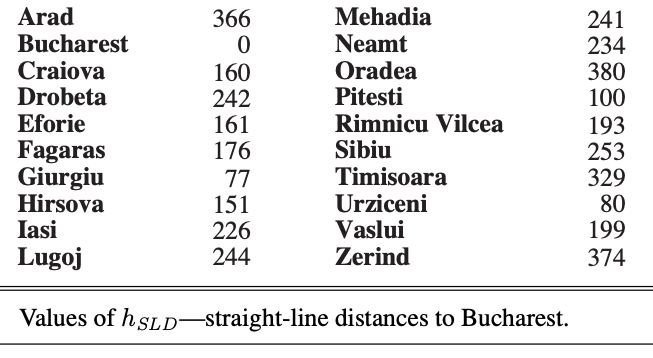
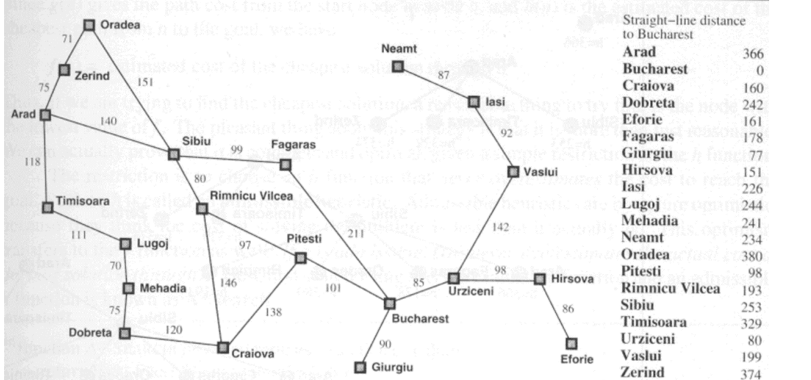
**Task 2: Implement Greedy Best First Search**

**Objectives:**

The program implements the Greedy Best-First Search algorithm to find an optimal path between Arad and Bucharest in the given map of Romania.

**Task Name:**

Write a program to implement Greedy Best First Search algorithm from figure to find an optimal path between Arad and Bucharest.

****

**Code:**

import heapq

# *Define the graph as an adjacency list*

graph = {

    'Arad': [('Zerind', 75), ('Sibiu', 140), ('Timisoara', 118)],

    'Zerind': [('Oradea', 71), ('Arad', 75)],

    'Oradea': [('Sibiu', 151), ('Zerind', 71)],

    'Sibiu': [('Oradea', 151), ('Arad', 140), ('Fagaras', 99), ('Rimnicu Vilcea', 80)],

    'Timisoara': [('Arad', 118), ('Lugoj', 111)],

    'Lugoj': [('Timisoara', 111), ('Mehadia', 70)],

    'Mehadia': [('Lugoj', 70), ('Drobeta', 75)],

    'Drobeta': [('Mehadia', 75), ('Craiova', 120)],

    'Craiova': [('Drobeta', 120), ('Rimnicu Vilcea', 146), ('Pitesti', 138)],

    'Rimnicu Vilcea': [('Sibiu', 80), ('Craiova', 146), ('Pitesti', 97)],

    'Fagaras': [('Sibiu', 99), ('Bucharest', 211)],

    'Pitesti': [('Rimnicu Vilcea', 97), ('Craiova', 138), ('Bucharest', 101)],

    'Bucharest': [('Fagaras', 211), ('Pitesti', 101), ('Giurgiu', 90), ('Urziceni', 85)],

    'Giurgiu': [('Bucharest', 90)],

    'Urziceni': [('Bucharest', 85), ('Hirsova', 98), ('Vaslui', 142)],

    'Hirsova': [('Urziceni', 98), ('Eforie', 86)],

    'Eforie': [('Hirsova', 86)],

    'Vaslui': [('Urziceni', 142), ('Iasi', 92)],

    'Iasi': [('Vaslui', 92), ('Neamt', 87)],

    'Neamt': [('Iasi', 87)],

}

# *Define the straight-line distances to Bucharest*

straight\_line\_distances = {

    'Arad': 366,

    'Bucharest': 0,

    'Craiova': 160,

    'Drobeta': 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

}

def greedy\_best\_first\_search(start, goal):

    # *Priority queue to store the nodes to be explored*

    priority\_queue = []

    heapq.heappush(priority\_queue, (straight\_line\_distances[start], start))

    # *Set to keep track of visited nodes*

    visited = set()

    # *Dictionary to store the path*

    came\_from = {}

    while priority\_queue:

        # *Get the node with the smallest heuristic value*

        \_, current = heapq.heappop(priority\_queue)

        # *If we reached the goal, reconstruct the path*

        if current == goal:

            path = []

            while current in came\_from:

                path.append(current)

                current = came\_from[current]

            path.append(start)

            path.reverse()

            return path

        # *Mark the current node as visited*

        visited.add(current)

        # *Explore the neighbors*

        for neighbor, cost in graph[current]:

            if neighbor not in visited:

heapq.heappush(priority\_queue, (straight\_line\_distances[neighbor], neighbor))

came\_from[neighbor] = current

    return None

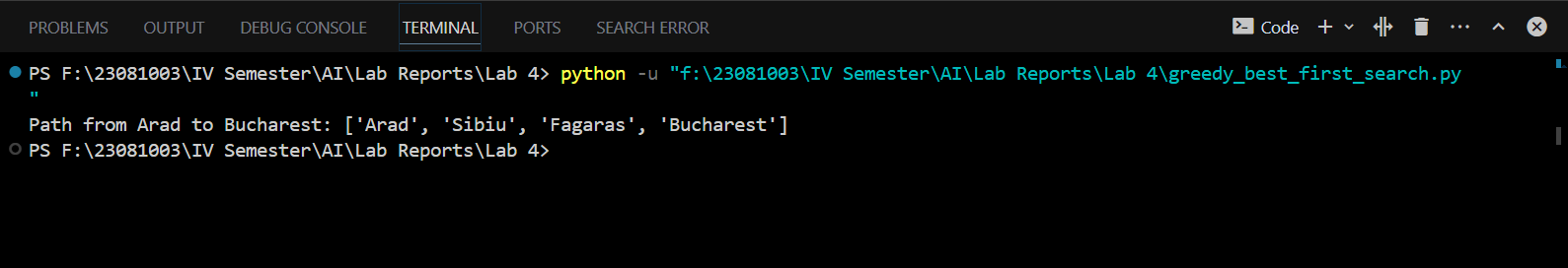
# *Test the function*

start = 'Arad'

goal = 'Bucharest'

path = greedy\_best\_first\_search(start, goal)

print(f"Path from {start} to {goal}: {path}")

**Output:**