CSE 4705: Assignment 02 - Arad to Bucharest - BFS, DFS, UCS, GBFS, A*

Problem 1

[100] Write a routine that solves the problem of finds a travel path of cities from from Arad to Bucharest in Romania, as discussed in class. Do this using each of the following approaches (points shown in brackets):

1. [15] Breadth First Search (BFS)
2. [10] Depth First Search (DFS)
3. [25] Uniform Cost Search (UCS)
4. [25] Greedy Best First Search (GBFS)
5. [25] A*

You will use the map from Lecture 03 - Informed Search which shows the major cities in Romania and the distances between them for those cities that are directly connected. Also, you will use the straight-line-distances shown in the adjacent table for your heuristic function, $h(n)$ for GBFS and A*. A screenshot of the relevant slide is given below. Data structures that store this information, romania_map and sld_to_bucharest, have been provided so you can access and apply this data in your algorithm implementations. Details of these data structures are given below.

Output for Each Routine

Each of your routines should return an output or set of outputs that clearly indicates the following:

1. The sequence of cities from Arad to Bucharest. (Make sure the cities, Arad and Bucharest are explicitly listed as the first and last cities in your output.) One suggestion is to return this output in the form of a list.
2. Cost to travel to each city from its predecessor.
3. Total cost for the path.

In the case of A* and Uniform Cost Search, your routines should return the *cheapest path*. However, that will not necessarily be the case for BFS, DFS, or GBFS. (Why not?)

Romania Graph

You will use the data structure stored in the romania_map, assigned below to implement the search across the various cities to find a path from Arad to Bucharest.

Some details about romania_map:

- A dictionary of dictionaries

- The outer dictionary is as follows: each key is a city and the value for that city is a nested dictionary of cities to which the said city is directly connected.
- The nested dictionary contains the cities to which the parent key is directly connected (keys) and the corresponding distances from the parent city to those respective cities (values).
- For example, for the city Oradea, we have a key in the outer dictionary (Oradea), and the associated value is a dictionary containing the Zerind and Sibiu as keys, where for each of these the values are the distances from Oradea to these respective cities.

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

Heuristic Function Data - Straight-Line Distances to Bucharest

You will use the dictionary below as your resource for retrieving straight-line distance data for implementing the GBFS and A* algorithms.

```
sld_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':100,
                    'Rimnicu Vilcea':193,
                    'Sibiu':253,
                    'Timisoara':329,
                    'Urziceni':80,
                    'Vaslui':199,
                    'Zerind':374
                   }
```

1. BFS Implementation

Provide your implementation of the BFS Search below.

```
from collections import deque



# BFS
def bfs_romania(start, goal):
    # Initialize the QUEUE with the start city and the path taken to
reach it
    #First In First Out
    queue = deque([(start, [start])])
    print("queue : ", queue)

    # Set to keep track of visited cities
    visited = set([start])
    print (visited)
    while queue:
            # Dequeue the first city in the queue
            city, path = queue.popleft()

            # If we reached the goal (Bucharest), return the path
            if city == goal:
                return path

            # Visit each neighbor of the current city
            for neighbor in romania_map[city]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor, path + [neighbor]))

    # If no path is found (shouldn't happen in this case)
    return None
```

```python
# Running BFS from Arad to Bucharest
path = bfs_romania('Arad', 'Bucharest')
```

```
queue :  deque([('Arad', ['Arad'])])
{'Arad'}
```

```python
# Output the result
print("Path from Arad to Bucharest:", path)
```

```
Path from Arad to Bucharest: ['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
```

2. DFS Implementation

Provide your implementation of the DFS Search below.

```python
# DFS
def dfs_romania(start, goal):
    # Stack for DFS (LIFO)
    stack = [(start, [start])]

    # Set to keep track of visited cities
    visited = set([start])

    while stack:
        # Pop the last city added to the stack
        city, path = stack.pop()

        # If we reached the goal (Bucharest), return the path
        if city == goal:
            return path

        # Visit each neighbor of the current city
        for neighbor in romania_map[city]:
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append((neighbor, path + [neighbor]))

    # If no path is found (shouldn't happen in this case)
    return None

# Running DFS from Arad to Bucharest
path_dfs = dfs_romania('Arad', 'Bucharest')

# Output the result
print("Path from Arad to Bucharest (DFS):", path_dfs)
```

```
Path from Arad to Bucharest (DFS): ['Arad', 'Timisoara', 'Lugoj',
'Mehadia', 'Dobreta', 'Craiova', 'Pitesti', 'Bucharest']
```

3. UCS Implementation

Provide your implementation of the UCS Search below.

```python
import heapq

# Uniform Cost Search
def ucs_romania(start, goal):
    # Priority queue for UCS
    queue = [(0, start, [])]  # (cumulative cost, current city, path
taken)

    # Dictionary to store the minimum cost to reach each city
    visited = {}

    while queue:
        # Pop the city with the lowest cumulative cost
        (cost, city, path) = heapq.heappop(queue)

        # If we reached the goal, return the path and the total cost
        if city == goal:
            return path + [city], cost

        # If the city has not been visited or a cheaper path is found
        if city not in visited or cost < visited[city]:
            visited[city] = cost
            # Visit each neighbor of the current city
            for neighbor, travel_cost in romania_map[city].items():
                heapq.heappush(queue, (cost + travel_cost, neighbor,
path + [city]))

    # If no path is found (shouldn't happen in this case)
    return None

# Running UCS from Arad to Bucharest
path_ucs, total_cost = ucs_romania('Arad', 'Bucharest')

# Output the result
print("Path from Arad to Bucharest (UCS):", path_ucs)
print("Total cost:", total_cost)
```

```
Path from Arad to Bucharest (UCS): ['Arad', 'Sibiu', 'Rimnicu Vilcea',
'Pitesti', 'Bucharest']
Total cost: 418
```

4. GBFS Implementation

Provide your implementation of the GBFS Search below.

```python
import heapq

# Greedy Best-First Search implementation
def gbfs_romania(start, goal):
    # Priority queue for GBFS (min-heap) based on the heuristic (SLD
to Bucharest)
    queue = [(sld_to_Bucharest[start], start, [])]  # (heuristic cost,
current city, path taken)

    # Set to keep track of visited cities
    visited = set()

    while queue:
        # Pop the city with the lowest heuristic cost
        (heuristic_cost, city, path) = heapq.heappop(queue)

        # If we reached the goal, return the path and the total
heuristic cost (ignores travel cost)
        if city == goal:
            return path + [city]

        # If the city has not been visited
        if city not in visited:
            visited.add(city)
            # Visit each neighbor of the current city
            for neighbor in romania_map[city]:
                if neighbor not in visited:
                    heapq.heappush(queue, (sld_to_Bucharest[neighbor],
neighbor, path + [city]))

    # If no path is found (shouldn't happen in this case)
    return None

# Running GBFS from Arad to Bucharest
path_gbfs = gbfs_romania('Arad', 'Bucharest')

# Output the result
print("Path from Arad to Bucharest (GBFS):", path_gbfs)
```

```
Path from Arad to Bucharest (GBFS): ['Arad', 'Sibiu', 'Fagaras',
'Bucharest']
```

5. A* Implementation

Provide your implementation of the A* Algorithm below.

```
import heapq

# A* Search implementation
def astar_romania(start, goal):
    # Priority queue for A* (min-heap)
    queue = [(sld_to_Bucharest[start], 0, start, [])]  # (f(n) = g(n)
+ h(n), g(n), current city, path taken)

    # Dictionary to store the minimum cost (g(n)) to reach each city
    visited = {}

    while queue:
        # Pop the city with the lowest f(n) cost
        (f_cost, g_cost, city, path) = heapq.heappop(queue)

        # If we reached the goal, return the path and the total cost
        if city == goal:
            return path + [city], g_cost

        # If the city has not been visited or a cheaper path is found
        if city not in visited or g_cost < visited[city]:
            visited[city] = g_cost
            # Visit each neighbor of the current city
            for neighbor, travel_cost in romania_map[city].items():
                new_g_cost = g_cost + travel_cost
                f_cost = new_g_cost + sld_to_Bucharest[neighbor]
                heapq.heappush(queue, (f_cost, new_g_cost, neighbor,
path + [city]))

    # If no path is found (shouldn't happen in this case)
    return None

# Running A* from Arad to Bucharest
path_astar, total_cost = astar_romania('Arad', 'Bucharest')

# Output the result
print("Path from Arad to Bucharest (A*):", path_astar)
print("Total cost:", total_cost)
```

```
Path from Arad to Bucharest (A*): ['Arad', 'Sibiu', 'Rimnicu Vilcea',
'Pitesti', 'Bucharest']
Total cost: 418
```