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

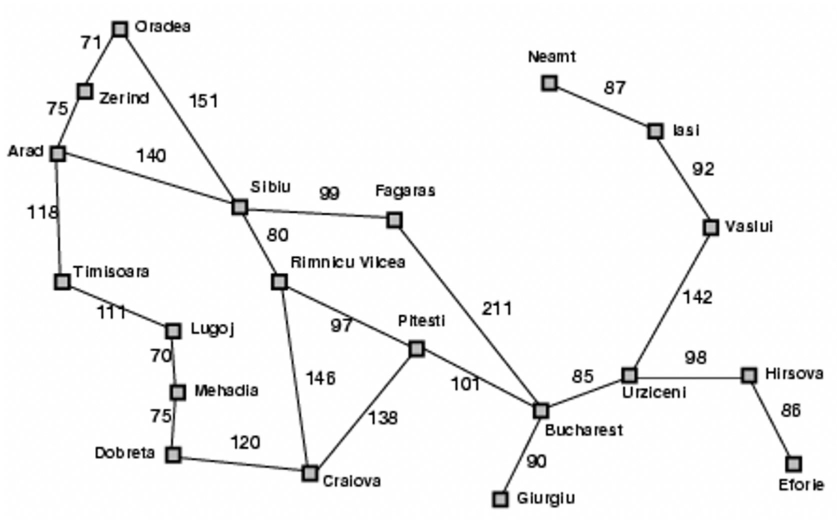
Problem 1

[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.

Example - Romania with step costs in km



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

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.

In [ ]:
```python
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.

In [ ]:

```
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,
}
```

In [ ]:

```
from collections import deque

# This function will return the pop function for the given type of structure.
def get_pop_function(givenType, struct):
    return struct.popleft \
        if givenType == deque \
        else struct.pop

'''
Parameterized function which can model either BFS/DFS depending on the type
of structure provided (i.e. deque for BFS, list (as stack) for DFS).
```

```python
'''
def graph_search(graph, struct, start, goal):
    # Setup the initiate state and visited nodes set.
    visited = set()
    initial_state = (start, [start])

    # Setup the data structure and the pop function.
    structure = struct([initial_state])
    pop = get_pop_function(struct, structure)
    extend = structure.extend

    '''
    Loop until the structure is empty:
       - Pop the first element from the structure.
        - If the element is not visited:
            - Add the element to the visited set.
            - If the element is the goal, return the path and the cost.
            - Extend the structure with the neighbors of the element.
    '''
    while struct:
        vertex, path = pop()
        if vertex not in visited:
            visited.add(vertex)
            if vertex == goal:
                return path, count_cost(graph, path)

            extend((neighbor, path + [neighbor]) for neighbor in graph[vertex] if neighbor not in visited)

    return None

# Counts the cost of the given path.
def count_cost(graph, path):
    cost = 0
    for i in range(len(path) - 1):
        cost += graph[path[i]][path[i + 1]]
    return cost

# Prints the path and cost.
def print_metrics(method, graph, path, total_cost = None):
    print(f'{method.upper()} path:')
    for i in range(len(path) - 1):
        print(f' {i + 1}. {path[i]} -> {path[i + 1]} ({graph[path[i]][path[i + 1]]})')
    if total_cost is not None:
        print(f'{method.upper()} total cost: {total_cost}')

'''
Parameterized function which can model either UCS/GBFS/A*
depending on the type of traversal function provided.
'''
def heuristic_graph_search(graph, traversal, start, goal):
    # Start node has no predecessor
    visited = {start: (None, 0)}

    # Frontier is a priority queue with the start node
    frontier = PriorityQueue()
    frontier.put((0, start))

    # Setup the explored nodes set
    explored = []

    '''
    Loop until the frontier is empty:
        - Pop the first element from the frontier.
        - Add the element to the explored set.
        - If the element is the goal, return the path and the cost.
        - For each neighbor of the element that is not in the explored set:
            - Call the traversal function to get the cost.
    '''
    while not frontier.empty():
        # Grab the current node
        current = frontier.get()[1]

        # Add it to the explored nodes set
        explored.append(current)

        # If we've reached the goal return the path + cost
        if current == goal:
            path = [current]
            total_cost = 0
            from_last = 0

            while visited[current][0] is not None:
                path.append(visited[current][0])
                total_cost += graph[current][visited[current][0]]
                current = visited[current][0]

            return path[::-1], total_cost

        '''
        Loop through the neighbors of the current node
        and call the provided traversal function.
        '''
        for neighbor in graph[current]:
            if neighbor not in explored:
                # call traversal function to get the cost
                traversal(graph, frontier, visited, current, neighbor)

    return None
```

1. BFS Implementation

Provide your implementation of the BFS Search below.

In [ ]:
```
# Call graph_search with a deque to perform BFS.
def breadth_first_search(graph, start, goal): return graph_search(graph, deque, start, goal)
```

In [ ]:
```
# Let's run BFS against `romania_map` to find a path from Arad to Bucharest.
bfs_path, bfs_cost = breadth_first_search(romania_map, 'Arad', 'Bucharest')
print_metrics('bfs', romania_map, bfs_path, bfs_cost)
```

```
BFS path:
 1. Arad -> Sibiu (140)
 2. Sibiu -> Fagaras (99)
 3. Fagaras -> Bucharest (211)
BFS total cost: 450
```

2. DFS Implementation

Provide your implementation of the DFS Search below.

In [ ]:
```
# Call graph_search with a list (models a stack) to perform DFS.
def depth_first_search(graph, start, goal): return graph_search(graph, list, start, goal)
```

In [ ]:
```
# Let's run DFS against `romania_map` to find a path from Arad to Bucharest.
dfs_path, dfs_cost = depth_first_search(romania_map, 'Arad', 'Bucharest')
print_metrics('dfs', romania_map, dfs_path, dfs_cost)
```

```
DFS path:
 1. Arad -> Timisoara (118)
 2. Timisoara -> Lugoj (111)
 3. Lugoj -> Mehadia (70)
 4. Mehadia -> Dobreta (75)
 5. Dobreta -> Craiova (120)
 6. Craiova -> Pitesti (138)
 7. Pitesti -> Bucharest (101)
DFS total cost: 733
```

3. UCS Implementation

Provide your implementation of the UCS Search below.

In [ ]:
```
from queue import PriorityQueue

def uniform_cost_search(graph, start, goal):
    def traversal(graph, frontier, visited, current, neighbor):
        # if neighbor is not in frontier, add it to frontier
        if neighbor not in [node[1] for node in frontier.queue]:
            visited[neighbor] = (current, visited[current][1] + graph[current][neighbor])
            frontier.put((visited[neighbor][1], neighbor))
            return

        if visited[current][1] + graph[current][neighbor] < visited[neighbor][1]:
            visited[neighbor] = (current, visited[current][1] + graph[current][neighbor])
            # remove neighbor from frontier
            for node in frontier.queue:
                if node[1] == neighbor:
                    frontier.queue.remove(node)
                    break
            frontier.put((visited[neighbor][1], neighbor))

    return heuristic_graph_search(graph, traversal, start, goal)
```

In [ ]:
```
ucs_path, ucs_cost = uniform_cost_search(romania_map, 'Arad', 'Bucharest')
print_metrics('ucs', romania_map, ucs_path, ucs_cost)
```

```
UCS path:
 1. Arad -> Sibiu (140)
 2. Sibiu -> Rimnicu Vilcea (80)
 3. Rimnicu Vilcea -> Pitesti (97)
 4. Pitesti -> Bucharest (101)
UCS total cost: 418
```

4. GBFS Implementation

Provide your implementation of the GBFS Search below.

In [ ]:
```
def greedy_bfs(graph, start, goal):
    def traversal(graph, frontier, visited, current, neighbor):
        # if neighbor is not in frontier, add it to frontier
        if neighbor not in [node[1] for node in frontier.queue]:
            visited[neighbor] = (current, sld_to_Bucharest[neighbor])
            frontier.put((visited[neighbor][1], neighbor))

    return heuristic_graph_search(graph, traversal, start, goal)
```

In [ ]:
```
gbfs_path, gbfs_cost = greedy_bfs(romania_map, 'Arad', 'Bucharest')
print_metrics('gbfs', romania_map, gbfs_path, gbfs_cost)
```

```
GBFS path:
 1. Arad -> Sibiu (140)
 2. Sibiu -> Fagaras (99)
 3. Fagaras -> Bucharest (211)
GBFS total cost: 450
```

5. A* Implementation

Provide your implementation of the A* Algorithm below.

In [ ]:
```
def a_star(graph, start, goal):
    def traversal(graph, frontier, visited, current, neighbor):
        # if neighbor is not in frontier, add it to frontier
```

```python
        if neighbor not in [node[1] for node in frontier.queue]:
            visited[neighbor] = (current, visited[current][1] + graph[current][neighbor])
            frontier.put((visited[neighbor][1] + sld_to_Bucharest[neighbor], neighbor))
            return

        # if neighbor is in frontier, update its cost if necessary
        if visited[current][1] + graph[current][neighbor] < visited[neighbor][1]:
            visited[neighbor] = (current, visited[current][1] + graph[current][neighbor])
            # remove neighbor from frontier
            for node in frontier.queue:
                if node[1] == neighbor:
                    frontier.queue.remove(node)
                    break
            frontier.put((visited[neighbor][1] + sld_to_Bucharest[neighbor], neighbor))

    return heuristic_graph_search(graph, traversal, start, goal)
```

In [ ]:
```python
a_star_path, a_star_cost = a_star(romania_map, 'Arad', 'Bucharest')
print_metrics('a*', romania_map, a_star_path, a_star_cost)
```

```
A* path:
 1. Arad -> Sibiu (140)
 2. Sibiu -> Rimnicu Vilcea (80)
 3. Rimnicu Vilcea -> Pitesti (97)
 4. Pitesti -> Bucharest (101)
A* total cost: 418
```