Lab Task: 08

————————————786————————————

Name: Muhammad Sherjeel Akhtar

Roll No: 20p-0101

Subject: Artificial Intelligence Lab

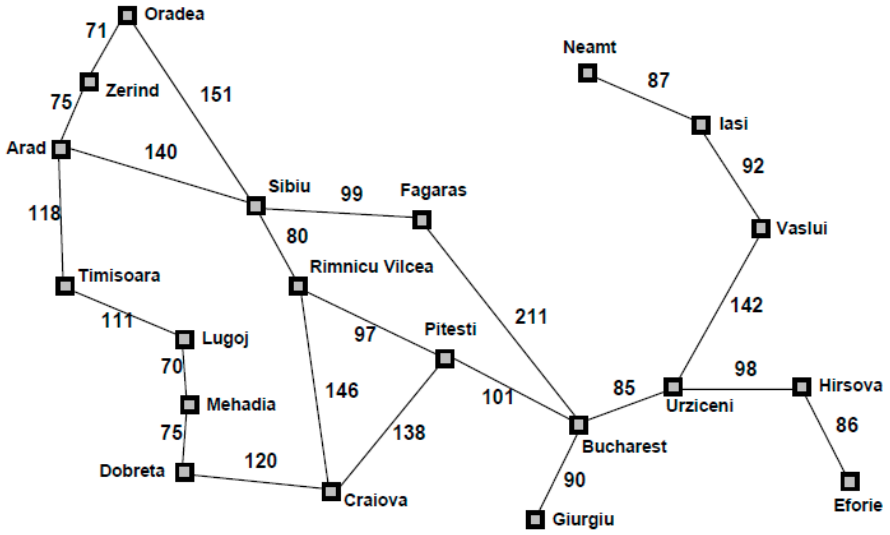
Submitted To Respected
Ma'am: Hurmat Hidayat

Section: BCS-6C

Header: Travelling on vacations to Romania

Answer:

Suppose that you plan to spend your summer vacations in Romania. Following is the map of Romania.



Libraries Importing:

Import All the required libraries here

In [26]:
```
import networkx as nx
import matplotlib.pyplot as plt
```

Generating Graph:

Generate graph for Map of Romania

Note: Complete the missing part of the code

In [27]:
```
# Define the graph
romania = nx.Graph()
# Add the cities as nodes
romania.add_nodes_from([
    'Oradea', 'Zerind', 'Arad', 'Timisoara', 'Lugoj',
    'Mehadia', 'Drobeta', 'Sibiu', 'Rimnicu', 'Craiova',
    'Fagaras', 'Pitesti', 'Bucharest', 'Giurgiu', 'Urziceni',
    'Hirsova', 'Eforie', 'Vaslui', 'Iasi', 'Neamt'
])
```

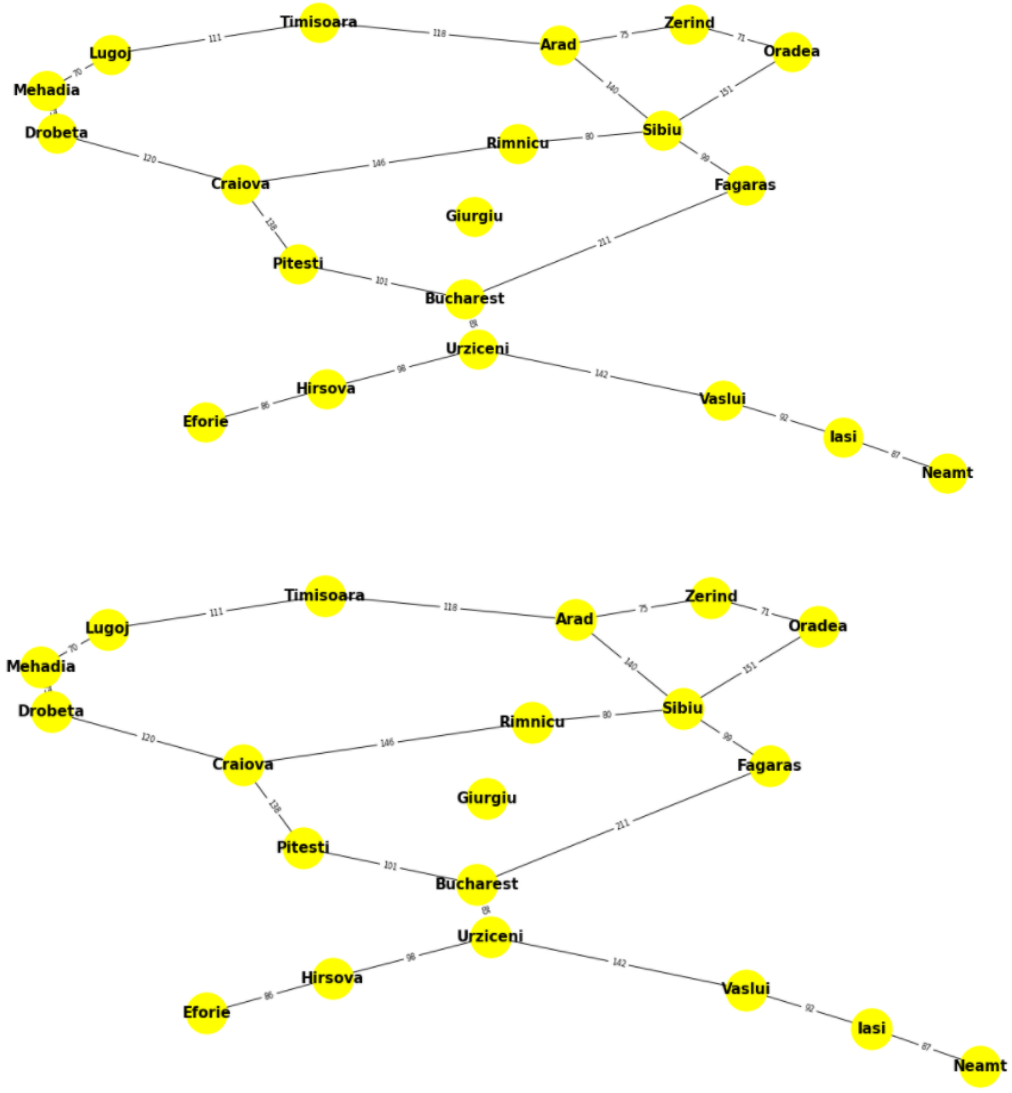
In [28]:
```
edges = [("Oradea", "Zerind", 71), ("Oradea", "Sibiu", 151),("Zerind", "Arad", 75),
         ("Arad", "Sibiu", 140),("Arad", "Timisoara", 118), ("Timisoara", "Lugoj", 111),
         ("Lugoj", "Mehadia", 70), ("Mehadia", "Drobeta", 75),("Drobeta", "Craiova", 120),
         ("Sibiu", "Rimnicu", 80),("Rimnicu", "Craiova", 146), ("Sibiu", "Fagaras", 99),
         ("Fagaras", "Bucharest", 211), ("Pitesti", "Bucharest", 101),("Craiova", "Pitesti", 138),
         ("Urziceni", "Bucharest", 85),("Urziceni", "Hirsova", 98), ("Eforie", "Hirsova", 86),
         ("Urziceni", "Vaslui", 142), ("Vaslui", "Iasi", 92),("Iasi", "Neamt", 87)]

romania.add_weighted_edges_from(edges)
```

The Kamada-Kawai layout is a method for graph layout that aims to produce an aesthetically pleasing layout by minimizing the total energy of the graph. The layout is computed by treating the edges of the graph as springs and the nodes as charged particles, and then using an iterative algorithm to find the optimal positions of the nodes that balance the forces between them.

```
# Draw graph with labels and edge weights
plt.figure(figsize=(16, 8))
nx.draw(romania, pos, with_labels=True, font_size=15,font_weight = "bold", font_family = "sans-serif",node_size= 1700, node_color ="yellow"

edge_labels = nx.get_edge_attributes(romania, "weight")
nx.draw_networkx_edge_labels(romania, pos, edge_labels=edge_labels, font_size=8)

plt.show()
```

Graph Visually:

In [38]:
```
# Draw graph with labels and edge weights
plt.figure(figsize=(16, 8))
nx.draw(romania, pos, with_labels=True, font_size=15,font_weight = "bold", font_family = "sans-serif",node_size= 17(

edge_labels = nx.get_edge_attributes(romania, "weight")
nx.draw_networkx_edge_labels(romania, pos, edge_labels=edge_labels, font_size=8)

plt.show()
```



The **init** method is the constructor for the Node class and takes three arguments: state, parent, and action.

- **state** represents the state of the node, which is usually a position or a configuration in a search problem.
- **parent** is a reference to the parent node in the search tree.
- **action** is the action that was taken to get to the current node from its parent.

Node Class And Stack Frontier Class:

In [31]:
```python
# Define the Node class for the search algorithms
class Node:
    def __init__(self, state, parent=None, action=None):
        self.state = state
        self.parent = parent
        self.action = action
```

This code defines a class StackFrontier that represents a stack data structure for storing nodes in a search algorithm. It has the following methods:

- **__init__** Initializes an empty list to store nodes.
- **add(node):** Adds a node to the top of the stack.
- **contains_state(state):** Checks if the given state is present in any of the nodes in the stack.
- **empty():** Returns True if the stack is empty, False otherwise.
- **remove():** Removes and returns the top node from the stack. If the stack is empty, it raises an exception.

In [32]:
```python
class StackFrontier(QueueFrontier):
    def remove(self):
        if self.frontier:
            return self.frontier.pop()
        else:
            raise Exception("Frontier is empty")
```

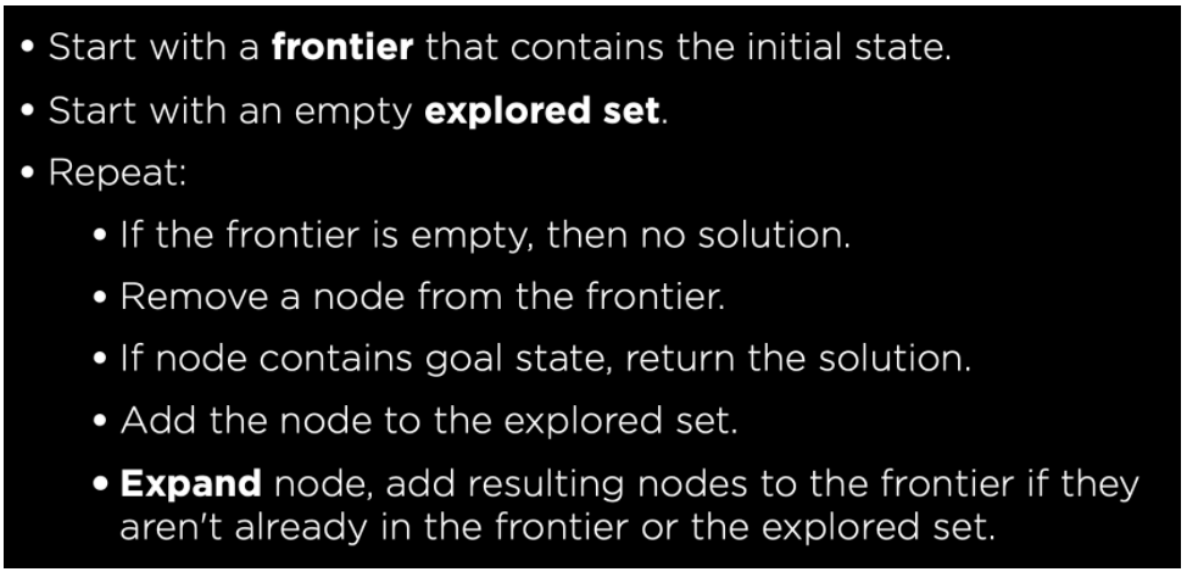
QueueFrontier is a class that is derived from StackFrontier (using inheritance). It inherits all the attributes and methods of StackFrontier and can have additional attributes and methods or override existing ones.

The remove method in QueueFrontier is an override of the remove method in StackFrontier. Instead of removing the last node added to the frontier (like StackFrontier does), it removes the first node added to the frontier, which makes it operate like a queue (FIFO). If the frontier is empty, it raises an exception.

Queue Frontier:

In [33]:
```python
# Define the frontier classes for BFS and DFS
class QueueFrontier:
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)

    def remove(self):
        if self.frontier:
            return self.frontier.pop(0)
        else:
            raise Exception("Frontier is empty")

    def empty(self):
        return not self.frontier

    def contains_state(self, state):
        return any(node.state == state for node in self.frontier)
```

Method:



Breadth First Search (BFS):

In [40]:
```python
# Define the search algorithms
def bfs_search(graph, start, goal):
    frontier = QueueFrontier()
    frontier.add(Node(state=start, parent=None, action=None))
    explored = set()
    while not frontier.empty():
        node = frontier.remove()
        explored.add(node.state)
        for neighbor, weight in graph[node.state].items():
            if neighbor not in explored and not frontier.contains_state(neighbor):
                child = Node(state=neighbor, parent=node, action=weight)
                if child.state == goal:
                    path = []
                    while child.parent is not None:
                        path.append({'state': child.state, 'weight': child.action})
                        child = child.parent
                    path.reverse()
                    return path
                frontier.add(child)
    return None
```

```
# Define the search algorithms
def bfs_search(graph, start, goal):
    frontier = QueueFrontier()
    frontier.add(Node(state=start, parent=None, action=None))
    explored = set()
    while not frontier.empty():
        node = frontier.remove()
        explored.add(node.state)
        for neighbor, weight in graph[node.state].items():
            if neighbor not in explored and not frontier.contains_state(neighbor):
                child = Node(state=neighbor, parent=node, action=weight)
                if child.state == goal:
                    path = []
                    while child.parent is not None:
                        path.append({'state': child.state, 'weight': child.action})
                        child = child.parent
                    path.reverse()
                    return path
                frontier.add(child)
    return None
```

Depth First Search (DFS):

In [41]:
```python
def dfs_search(graph, start, goal):
    frontier = StackFrontier()
    frontier.add(Node(state=start, parent=None, action=None))
    explored = set()
    while not frontier.empty():
        node = frontier.remove()
        explored.add(node.state)
        for neighbor, weight in graph[node.state].items():
            if neighbor not in explored and not frontier.contains_state(neighbor):
                child = Node(state=neighbor, parent=node, action=weight)
                if child.state == goal:
                    path = []
                    while child.parent is not None:
                        path.append({'state': child.state, 'weight': child.action})
                        child = child.parent
                    path.reverse()
                    return path
                frontier.add(child)
    return None
```

```
def dfs_search(graph, start, goal):
    frontier = StackFrontier()
    frontier.add(Node(state=start, parent=None, action=None))
    explored = set()
    while not frontier.empty():
        node = frontier.remove()
        explored.add(node.state)
        for neighbor, weight in graph[node.state].items():
            if neighbor not in explored and not frontier.contains_state(neighbor):
                child = Node(state=neighbor, parent=node, action=weight)
                if child.state == goal:
                    path = []
                    while child.parent is not None:
                        path.append({'state': child.state, 'weight': child.action})
                        child = child.parent
                    path.reverse()
                    return path
                frontier.add(child)
    return None
```

Applying BFS:

In [36]:
```python
start = 'Arad'
goal = 'Bucharest'

# BFS
bfs_path = bfs_search(romania, start, goal)
if bfs_path is None:
    print(f"No path found from {start} to {goal} using BFS")
else:
    bfs_cities = [start] + [step['state'] for step in bfs_path]
    bfs_weights = [step['weight'] for step in bfs_path]
    print(f"BFS path from {start} to {goal}:")
    print(list(zip(bfs_cities, bfs_weights)))
```

```
BFS path from Arad to Bucharest:
[('Arad', {'weight': 140}), ('Sibiu', {'weight': 99}), ('Fagaras', {'weight': 211})]
```

```
start = 'Arad'
goal = 'Bucharest'

# BFS
bfs_path = bfs_search(romania, start, goal)
if bfs_path is None:
    print(f"No path found from {start} to {goal} using BFS")
else:
    bfs_cities = [start] + [step['state'] for step in bfs_path]
    bfs_weights = [step['weight'] for step in bfs_path]
    print(f"BFS path from {start} to {goal}:")
    print(list(zip(bfs_cities, bfs_weights)))
```

Applying DFS:

In [24]:
```
# DFS
dfs_path = dfs_search(romania, start, goal)
if dfs_path is None:
    print(f"No path found from {start} to {goal} using BFS")
else:
    dfs_cities = [start] + [step['state'] for step in dfs_path]
    dfs_weights = [step['weight'] for step in dfs_path]
    print(f"DFS path from {start} to {goal}:")
    print(list(zip(dfs_cities, dfs_weights)))
```

```
DFS path from Arad to Bucharest:
[('Arad', {'weight': 118}), ('Timisoara', {'weight': 111}), ('Lugoj', {'weight': 70}), ('Mehadia', {'weight': 7
5}), ('Drobeta', {'weight': 120}), ('Craiova', {'weight': 138}), ('Pitesti', {'weight': 101})]
```

```
# DFS
dfs_path = dfs_search(romania, start, goal)
if dfs_path is None:
    print(f"No path found from {start} to {goal} using BFS")
else:
    dfs_cities = [start] + [step['state'] for step in dfs_path]
    dfs_weights = [step['weight'] for step in dfs_path]
    print(f"DFS path from {start} to {goal}:")
    print(list(zip(dfs_cities, dfs_weights)))
```

In Comparison With Sample Output:

In [24]:
```
# DFS
dfs_path = dfs_search(romania, start, goal)
if dfs_path is None:
    print(f"No path found from {start} to {goal} using BFS")
else:
    dfs_cities = [start] + [step['state'] for step in dfs_path]
    dfs_weights = [step['weight'] for step in dfs_path]
    print(f"DFS path from {start} to {goal}:")
    print(list(zip(dfs_cities, dfs_weights)))
```

```
DFS path from Arad to Bucharest:
[('Arad', {'weight': 118}), ('Timisoara', {'weight': 111}), ('Lugoj', {'weight': 70}), ('Mehadia', {'weight': 7
5}), ('Drobeta', {'weight': 120}), ('Craiova', {'weight': 138}), ('Pitesti', {'weight': 101})]
```

Sample output

BFS path from Arad to Bucharest:

[{'weight': 140}, {'weight': 99}, {'weight': 211}]

['Sibiu', 'Fagaras', 'Bucharest']

DFS path from Arad to Bucharest:

[{'weight': 118}, {'weight': 111}, {'weight': 70}, {'weight': 75}, {'weight': 120}, {'weight': 138}, {'weight': 101}]

['Timisoara', 'Lugoj', 'Mehadia', 'Dobreta', 'Craiova', 'Pitesti', 'Bucharest']

FIN.