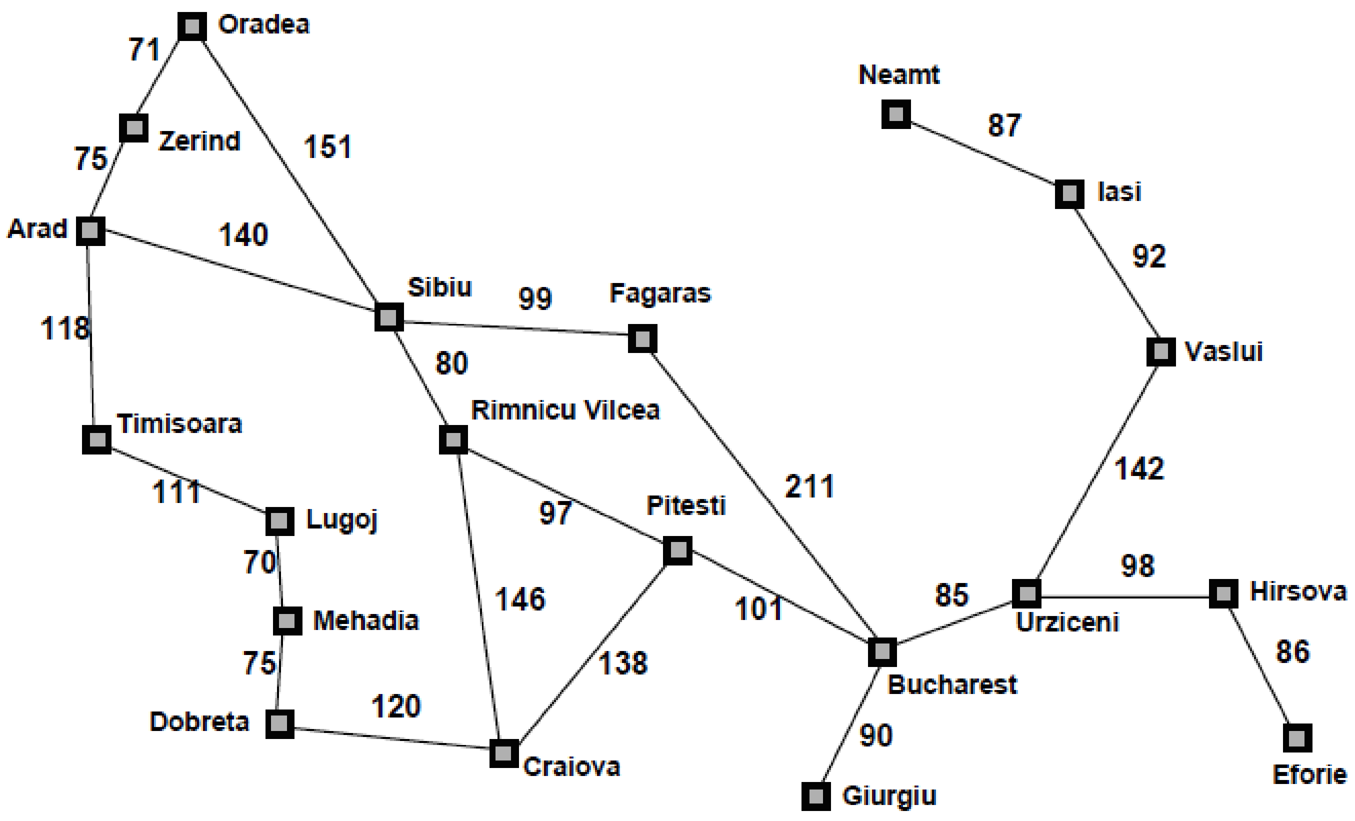
Lab 08 - Task ¶

Travelling on vacations to Romania

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



The map of Romania is represented as a graph, with cities as **nodes** and roads as **edges**. You are given the starting city and the goal city. You need to write two functions, one using **BFS** and one using **DFS**, to find a path between the starting city and the goal city.

Your functions should take the following inputs:

- G: A NetworkX graph object representing the map of Romania.
- start: The starting city.
- goal: The goal city.

Your functions should return a path (a list of cities) from the starting city to the goal city, or None if no such path exists.

Import All the required libraries here

In [1]:

```
import networkx as nx
import matplotlib.pyplot as plt
```

Generate graph for Map of Romania

Note: Complete the missing part of the code

In [2]:

```
G = nx.Graph()

G.add_nodes_from(["Arad", "Bucharest","Oradea","Zerind","Sibiu","Timisoara","Lugoj","Mah
```

In [3]:

```
edges = [("Arad", "Zerind", 75),("Arad", "Sibiu", 140),("Arad", "Timisoara", 118),("Buch
("Hirsova", "Urziceni", 98),("Iasi", "Neamt", 87),("Iasi", "Vaslui", 92),("Lugoj", "Meha

for edge in edges:
    G.add_edge(edge[0], edge[1], weight=edge[2])
```

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.

In [4]:

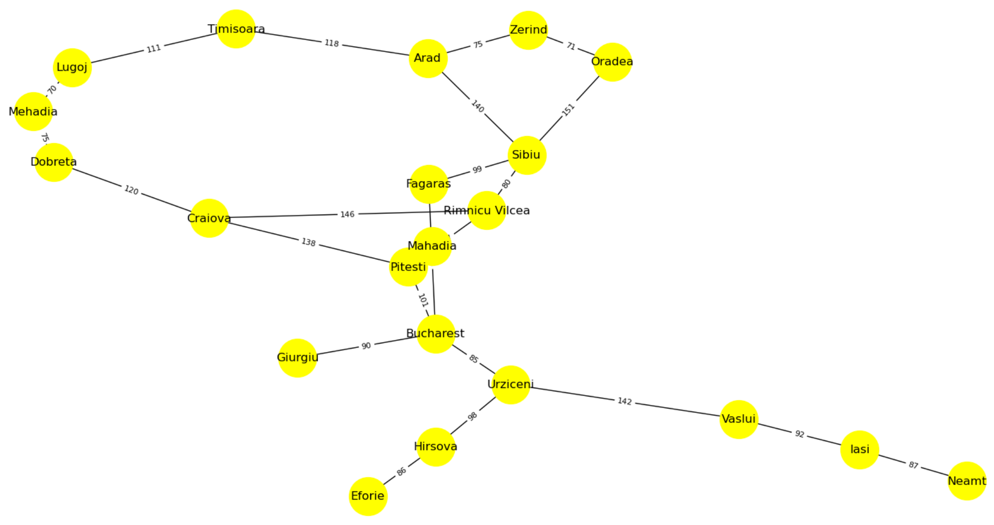
```
# Set node positions using Kamada-Kawai layout
pos = nx.kamada_kawai_layout(G)
```

In [5]:

```
# Draw graph with labels and edge weights
plt.figure(figsize=(16, 8))
nx.draw(G, pos, with_labels=True, font_size=12, node_size= 1500, node_color ="yellow")

edge_labels = nx.get_edge_attributes(G, "weight")
nx.draw_networkx_edge_labels(G, 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.

In [6]:

```
class Node():
    def __init__(self, state, parent, action):
        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 [7]:

```
class StackFrontier():
    def __init__(self):
        self.frontier = []

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

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

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return node
```

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.

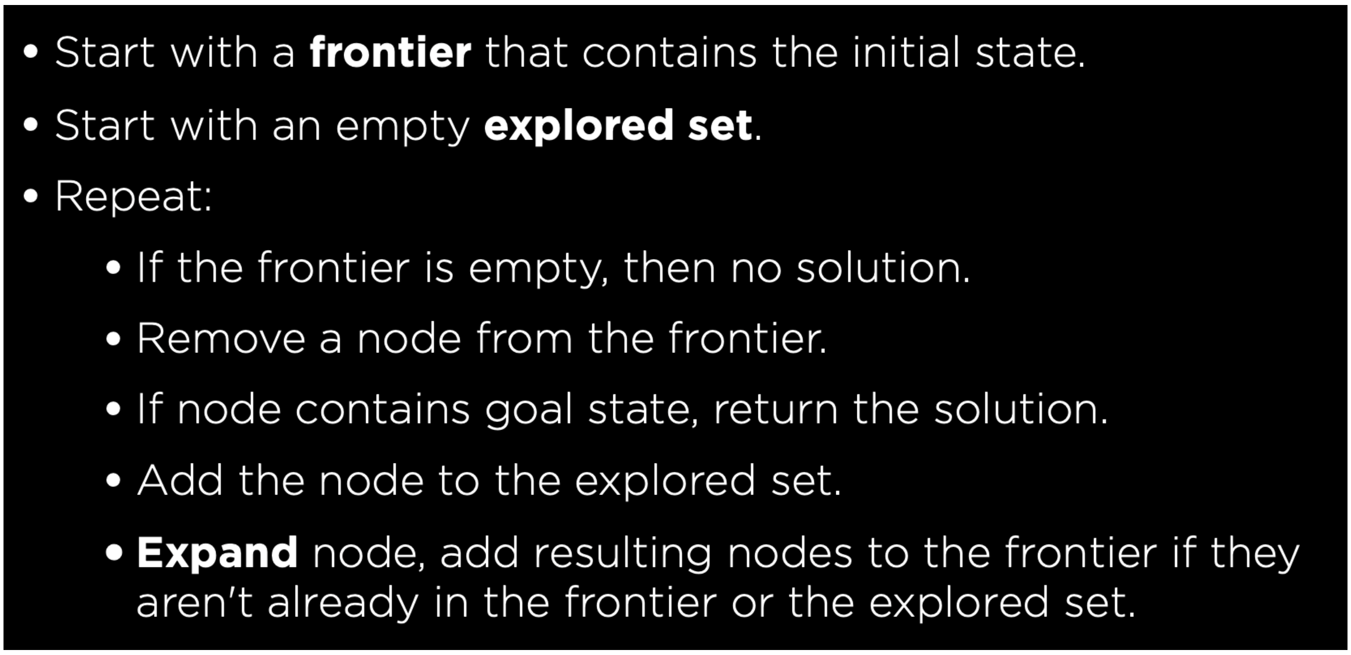
The remove method in QueueFrontier is an override of the remove method in StackFrontier. Instead of removing the last node added to the frontier (like StackFrontier does), it removes the first node added to the frontier, which makes it operate like a queue (FIFO). If the frontier is empty, it raises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

In [8]:

```
class QueueFrontier(StackFrontier):

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node
```

Breadth First Search



- Start with a **frontier** that contains the initial state.
- Start with an empty **explored set**.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove a node from the frontier.
  - If node contains goal state, return the solution.
  - Add the node to the explored set.
  - **Expand** node, add resulting nodes to the frontier if they aren't already in the frontier or the explored set.

In [9]:

```python
def bfs_search(graph, start, goal):
    frontier = QueueFrontier()
    start_node = Node(start, None, None)
    frontier.add(start_node)
    explored = set()

    while not frontier.empty():
        node = frontier.remove()

        if node.state == goal:
            actions = []
            nodes = []
            while node.parent is not None:
                actions.append({"weight": G[node.state][node.parent.state]["weight"]})
                nodes.append(node.state)
                node = node.parent
            actions.reverse()
            nodes.reverse()
            return actions, nodes

        explored.add(node.state)

        for neighbor in G.neighbors(node.state):
            if neighbor not in explored and not frontier.contains_state(neighbor):
                child = Node(neighbor, node, None)
                frontier.add(child)

    return None


    #your code
```

In [10]:

```
def dfs_search(graph, start, goal):
    frontier = StackFrontier()
    start_node = Node(start, None, None)
    frontier.add(start_node)
    explored = set()

    while not frontier.empty():
        node = frontier.remove()

        if node.state == goal:
            actions = []
            nodes = []
            while node.parent is not None:
                actions.append({"weight": G[node.state][node.parent.state]["weight"]})
                nodes.append(node.state)
                node = node.parent
            actions.reverse()
            nodes.reverse()
            return actions, nodes

        explored.add(node.state)

        for neighbor in G.neighbors(node.state):
            if neighbor not in explored and not frontier.contains_state(neighbor):
                child = Node(neighbor, node, None)
                frontier.add(child)

    return None

    #your code
```

In [11]:

```
# Perform a BFS search from Arad to Bucharest
actions, nodes = bfs_search(G, 'Arad', 'Bucharest')
print('BFS path from Arad to Bucharest:')
print(actions)
print(nodes)

# Perform a DFS search from Arad to Bucharest
actions, nodes = dfs_search(G, 'Arad', 'Bucharest')
print('DFS path from Arad to Bucharest:')
print(actions)
print(nodes)
```

```
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}, {'weigh
t': 120}, {'weight': 138}, {'weight': 101}]
['Timisoara', 'Lugoj', 'Mehadia', 'Dobreta', 'Craiova', 'Pitesti', 'Buchar
est']
```

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']