SUBMITTED TO:
DR. DILSHAD SABIR

SUBMITTED BY:
DANISH ZULFIQAR

REGISTRATION NO.:
FA21-BEE-042

SEMESTER-SECTION:
5-B

COURSE:
Artificial Intelligence
EEE 462

Lab Report 03

BS Electrical Engineering

LAB # 3

Lab Activities:

Activity 1:

Consider a toy problem that can be represented as a following graph. How would you represent this graph in python?

```
class Node:
   def __init__(self, state, parent, actions, totalCost):
      self.state = state
      self.parent = parent
      self.actions = actions
      self.totalCost = totalCost
graph = {'A': Node('A', None, ['B', 'C', 'E'], None),
       'B': Node('B', None, ['A', 'D', 'E'], None),
       'C': Node('C', None, ['A', 'F', 'G'], None),
       'D': Node('D', None, ['B', 'E'], None),
       'E': Node('E', None, ['A', 'B', 'D'], None),
       'F': Node('F', None, ['C'], None),
       'G': Node('G', None, ['C'], None)}
```

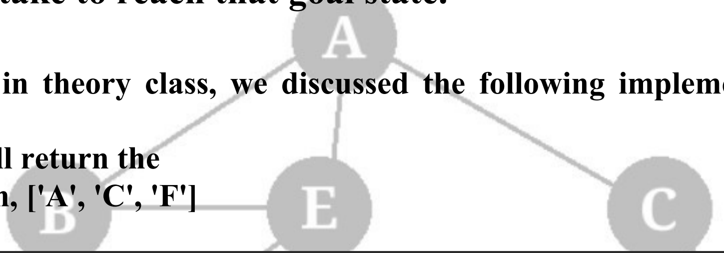
Activity 2:

For the graph in previous activity, imagine node A as starting node and your goal is to reach F. Keeping breadth first search in mind, describe a sequence of actions that you must take to reach that goal state.

Solution:

Remember that in theory class, we discussed the following implementation of breadth first search.

Calling BFS() will return the following solution, ['A', 'C', 'F']




```
class Node:
   def __init__(self, state, parent, actions, totalCost):
      self.state = state
      self.parent = parent
      self.actions = actions
      self.totalCost = totalCost
graph = {'A': Node('A', None, ['B', 'C', 'E'], None),
       'B': Node('B', None, ['A', 'D', 'E'], None),
       'C': Node('C', None, ['A', 'F', 'G'], None),
       'D': Node('D', None, ['B', 'E'], None),
       'E': Node('E', None, ['A', 'B', 'D'], None),
       'F': Node('F', None, ['C'], None),
       'G': Node('G', None, ['C'], None)}

def BFS():
```

```
    initialState = 'A'
    goalState = 'F'
    graph = {'A': Node('A', None, ['B', 'C', 'E'], None),
             'B': Node('B', None, ['A', 'D', 'E'], None),
             'C': Node('C', None, ['A', 'F', 'G'], None),
             'D': Node('D', None, ['B', 'E'], None),
             'E': Node('E', None, ['A', 'B', 'D'], None),
             'F': Node('F', None, ['C'], None),
             'G': Node('G', None, ['C'], None)}
    frontier = [initialState]
    explored = []
    while len(frontier) != 0:
        currentNode = frontier.pop(0)
        explored.append(currentNode)
        for child in graph[currentNode].actions:
            if child not in frontier and child not in explored:
                graph[child].parent = currentNodE
                if graph[child].state == goalState:
                    return actionSequence(graph, initialState, goalState)
                frontier.append(child)
    solution = BFS()
    print(solution)
def actionSequence(graph, initialState, goalState):
    solution = [goalState]
    currentParent = graph[goalState].parent
    while currentParent != None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent
    solution.reverse()
    return solution
```

Output:

```
C:\Users\CL-3\PycharmProjects\pythonP
['A','C','F']

Process finished with exit code 0
```

Activity 3:

Change initial state to D and set goal state as C. What will be resulting path of BFS search?

```
class Node:
    def __init__(self, state, parent, actions, totalCost):
        self.state = state
        self.parent = parent
        self.actions = actions
        self.totalCost = totalCost
graph = {'A': Node('A', None, ['B', 'C', 'E'], None),
```

```
       'B': Node('B', None, ['A', 'D', 'E'], None),
       'C': Node('C', None, ['A', 'F', 'G'], None),
       'D': Node('D', None, ['B', 'E'], None),
       'E': Node('E', None, ['A', 'B', 'D'], None),
       'F': Node('F', None, ['C'], None),
       'G': Node('G', None, ['C'], None)}


def BFS():
   initialState = 'D'
   goalState = 'C'
   graph = {'A': Node('A', None, ['B', 'C', 'E'], None),
          'B': Node('B', None, ['A', 'D', 'E'], None),
          'C': Node('C', None, ['A', 'F', 'G'], None),
          'D': Node('D', None, ['B', 'E'], None),
          'E': Node('E', None, ['A', 'B', 'D'], None),
          'F': Node('F', None, ['C'], None),
          'G': Node('G', None, ['C'], None)}
   frontier = [initialState]
   explored = []
   while len(frontier) != 0:
      currentNode = frontier.pop(0)
   explored.append(currentNode)
   for child in graph[currentNode].actions:
      if child not in frontier and child not in explored:
         graph[child].parent = currentNode
   if graph[child].state == goalState:
      return actionSequence(graph, initialState, goalState)
   frontier.append(child)


solution = BFS()
print(solution)


def actionSequence(graph, initialState, goalState):
   solution = [goalState]
   currentParent = graph[goalState].parent
   while currentParent != None:
      solution.append(currentParent)
   currentParent = graph[currentParent].parent
   solution.reverse()
   return solution
```

Solution:

['D', 'B', 'A', 'C']

```
C:\Users\Cl-3\PycharmProjects\pythonPr
['D','B','A','C']

Process finished with exit code 0
```

Home Task:

Code:

```python
from queue import Queue
romaniaMap = {
    'Arad': ['Sibiu', 'Zerind', 'Timisoara'],
    'Zerind': ['Arad', 'Oradea'],
    'Oradea': ['Zerind', 'Sibiu'],
    'Sibiu': ['Arad', 'Oradea', 'Fagaras', 'Rimnicu'],
    'Timisoara': ['Arad', 'Lugoj'],
    'Lugoj': ['Timisoara', 'Mehadia'],
    'Mehadia': ['Lugoj', 'Drobeta'],
    'Drobeta': ['Mehadia', 'Craiova'],
    'Craiova': ['Drobeta', 'Rimnicu', 'Pitesti'],
    'Rimnicu': ['Sibiu', 'Craiova', 'Pitesti'],
    'Fagaras': ['Sibiu', 'Bucharest'],
    'Pitesti': ['Rimnicu', 'Craiova', 'Bucharest'],
    'Bucharest': ['Fagaras', 'Pitesti', 'Giurgiu', 'Urziceni'],
    'Giurgiu': ['Bucharest'],
    'Urziceni': ['Bucharest', 'Vaslui', 'Hirsova'],
    'Hirsova': ['Urziceni', 'Eforie'],
    'Eforie': ['Hirsova'],
    'Vaslui': ['Iasi', 'Urziceni'],
    'Iasi': ['Vaslui', 'Neamt'],
    'Neamt': ['Iasi']
}


def bfs(startingNode, destinationNode):
    # For keeping track of what we have visited
    visited = {}
    # keep track of distance
    distance = {}
    # parent node of specific graph
    parent = {}

    bfs_traversal_output = []
    # BFS is queue based so using 'Queue' from python built-in
```

```
    queue = Queue()

    # travelling the cities in map
    for city in romaniaMap.keys():
        # since intially no city is visited so there will be nothing in visited list
        visited[city] = False
        parent[city] = None
        distance[city] = -1

    # starting from 'Arad'
    startingCity = startingNode
    visited[startingCity] = True
    distance[startingCity] = 0
    queue.put(startingCity)

    while not queue.empty():
        u = queue.get()     # first element of the queue, here it will be 'arad'
        bfs_traversal_output.append(u)

        # explore the adjust cities adj to 'arad'
        for v in romaniaMap[u]:
            if not visited[v]:
                visited[v] = True
                parent[v] = u
                distance[v] = distance[u] + 1
                queue.put(v)

        # reaching our destination city i.e 'bucharest'
    g = destinationNode
    path = []
    while g is not None:
        path.append(g)
        g = parent[g]

    path.reverse()
    # printing the path to our destination city
    print(path)

# Starting City & Destination City
bfs('Arad', 'Bucharest')
```

Output:

```
"C:\Users\FaT\Desktop\New folder (2)\AI\pythonProje
['Arad', 'Sibiu', 'Fagaras', 'Bucharest']

Process finished with exit code 0
```

Critical Analysis and Conclusion:

In this lab, I learned about Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms in Python for graph and tree traversal. BFS and DFS are fundamental concepts for Python programmers. I explored BFS, its algorithm, Python code implementation, and its real-world applications, using the example of solving a Rubik's Cube as a graph problem.

BFS involves traversing a graph by dividing its nodes into "Visited" and "Not Visited" categories to avoid cycles. The algorithm starts from a node and explores nodes at increasing distances from the starting point, using a queue to keep track of nodes to be visited. The BFS algorithm can be summarized as follows:

- Begin by placing one of the graph's vertices at the back of the queue.
- Take the front item from the queue and mark it as visited.
- Create a list of adjacent nodes to the current vertex and add unvisited ones to the rear of the queue.
- Repeat steps 2 and 3 until the queue is empty.
- In cases of disconnected graphs, run the BFS algorithm from every unvisited node to ensure all vertices are visited.