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 04

BS Electrical Engineering

LAB # 4

Lab Activities:

Activity 1:

```
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:

```
class Node:
    def __init__(self, state, parent, actions):
        self.state = state
        self.parent = parent
        self.actions = actions

def DFS():
    initialState = 'A'
    goalState = 'F'

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

    frontier = [initialState]
    explored = []
```

```
    while len(frontier) != 0:
        currentNode = frontier.pop()  # Using LIFO for DFS
        print(currentNode)
        explored.append(currentNode)

        if currentNode == goalState:
            print(explored)
            return actionSequence(graph, initialState, goalState)

        currentChildren = 0
        for child in graph[currentNode].actions:
            if child not in frontier and child not in explored:
                graph[child].parent = currentNode
                frontier.append(child)
                currentChildren += 1

        if currentChildren == 0:
            explored.pop()  # Removing the last node if it has no unexplored children

    return None

def actionSequence(graph, initialState, goalState):
    solution = [goalState]
    currentParent = graph[goalState].parent

    while currentParent is not None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent

    solution.reverse()
    return solution

solution = DFS()
if solution:
    print("Solution:", ' -> '.join(solution))
else:
    print("No solution found.")
```

Output:

```
bash-3.2$ python3 Task-1.py
bash-3.2$ python3 Task-2.py
A
C
G
F
['A', 'C', 'F']
Solution: A -> C -> F
bash-3.2$ 
```

Activity 3:

```
from collections import deque

class Node:
    def __init__(self, state, parent, actions):
        self.state = state
        self.parent = parent
        self.actions = actions

def BFS(graph, initial_state, goal_state):
    frontier = deque([initial_state])
    explored = set()

    while frontier:
        current_node = frontier.popleft()  # Using FIFO for BFS
        explored.add(current_node)

        if current_node == goal_state:
            return action_sequence(graph, initial_state, goal_state)

        for child in graph[current_node].actions:
            if child not in frontier and child not in explored:
                graph[child].parent = current_node
                frontier.append(child)

    return None

def action_sequence(graph, initial_state, goal_state):
    solution = [goal_state]
    current_parent = graph[goal_state].parent

    while current_parent != initial_state:
        solution.append(current_parent)
        current_parent = graph[current_parent].parent

    solution.append(initial_state)
    solution.reverse()
    return solution

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

```python
initial_state = 'D'
goal_state = 'C'

solution = BFS(graph, initial_state, goal_state)
if solution:
    print("Solution Path:", ' -> '.join(solution))
    print("Explored Node Sequence:", ', '.join(solution))
else:
    print("No solution found.")
```

```python
    currentParent = graph[currentParent].parent
    solution.reverse()
    return solution
```

Solution:

```
bash-3.2$ python3 Task-3.py
Solution Path: D -> B -> A -> C
Explored Node Sequence: D, B, A, C
bash-3.2$
```

Home Task:

```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):
```

```python
    # 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:

```
bash-3.2$ python3 Task-4.py
['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
bash-3.2$ 
```

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.