DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF E&ME, NUST, RAWALPINDI

AI & Decision Support Systems

Lab Report #2

Student Name: **Nawab Aarij Imam**

Degree/ Syndicate: **43 CE - A**

Task1:

Q1: Perform the following tasks

a. Implement the undirected Graph 1 in Python. Show the connectivity as well as the degree of each node within these graphs.

```
{1: [5, 2], 2: [5, 3, 1], 3: [4, 2], 4: [6, 3, 5], 5: [4, 2, 1], 6: [4]}
1 : 2
2 : 3
3 : 2
4 : 3
5 : 3
6 : 1
```

b. Implement the directed Graph 2 in Python. Show the connectivity, indegree and outdegree of each node within these graphs.

```
Graph structure: {'A': ['B'], 'B': ['C', 'D', 'E'], 'C': ['E'], 'D': ['E'], 'E': ['F'], 'F': [], 'G': ['D']}
A:OutDegree: 1, InDegree: 0
B:OutDegree: 3, InDegree: 1
C:OutDegree: 1, InDegree: 1
D:OutDegree: 1, InDegree: 2
E:OutDegree: 1, InDegree: 3
F:OutDegree: 0, InDegree: 1
G:OutDegree: 1, InDegree: 0
```

c. Write a method to find any path between node 6 to node 1 in Graph 1.

```
path: [6, 4, 5, 1]
```

d. Write a method to find any path between node A to node F in Graph 2.

```
path: ['A', 'B', 'E', 'F']
```

e. Modify Task c to show all possible paths between node 6 to node 1 in Graph 1.

```
all path: [[6, 4, 3, 2, 5, 1], [6, 4, 3, 2, 1], [6, 4, 5, 2, 1], [6, 4, 5, 1]]
```

f. Modify Task f to show all possible paths between node A to node F in Graph 2.

```
all paths: [['A', 'B', 'E', 'F'], ['A', 'B', 'D', 'E', 'F']
```

g. Represent Graph 1 and Graph 2 by adjacency list.

```
{1: [5, 2], 2: [5, 3, 1], 3: [4, 2], 4: [6, 3, 5], 5: [4, 2, 1], 6: [4]}
Graph structure: {'A': ['B'], 'B': ['C', 'D', 'E'], 'C': ['E'], 'D': ['E'], 'E': ['F'], 'F': [], 'G': ['D']}
```

Code:

Undirected Graph:

```
class UndirectedGraph:
  graph = {}

  def __init__(self):
    self.graph = {}
```

```python
def add_vertex(self, vertex):
    if vertex not in self.graph:
        self.graph[vertex] = []

def add_edge(self,vertex1, vertex2):
    self.graph[vertex1].append(vertex2)
    self.graph[vertex2].append(vertex1)

def get_degree(self, vertex):
    return len(self.graph[vertex])

def print_graph(self):
    for vertex in self.graph:
        print(f"{vertex} : {self.graph[vertex]}")

def find_all_paths(self, start, end, path=None):
    if path is None:
        path = []

    path = path + [start]

    if start == end:
        return [path]

    if start not in self.graph:
        return []

    paths = []
    for neighbor in self.graph[start]:
        if neighbor not in path:
            new_paths = self.find_all_paths(neighbor, end, path)
            for p in new_paths:
                paths.append(p)

    return paths
```

```python
def dijkstra(self, start_node):
    distances = {node: float('inf') for node in self.graph}
    distances[start_node] = 0
    visited = set()
    predecessors = {node: None for node in self.graph}

    while len(visited) < len(self.graph):
        current_node = None
        current_min_distance = float('inf')

        for node in distances:
            if node not in visited and distances[node] < current_min_distance:
                current_min_distance = distances[node]
                current_node = node

        if current_node is None:
            break

        visited.add(current_node)

        for neighbor in self.graph[current_node]:
            if neighbor not in visited:
                weight = abs(current_node - neighbor)
                new_distance = distances[current_node] + weight

                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance
                    predecessors[neighbor] = current_node

    return distances, predecessors

def shortest_path(self, start_node, end_node):
    distances, predecessors = self.dijkstra(start_node)

    path = []
```

```
    current_node = end_node
    while current_node is not None:
        path.insert(0, current_node)
        current_node = predecessors[current_node]

    if distances[end_node] == float('inf'):
        return None
    else:
        return path, distances[end_node]

def find_path(self, start, end):
    queue = [[start]]
    visited = set()

    if start == end:
        return [start]

    while queue:
        path = queue.pop(0)
        node = path[-1]

        if node not in visited:
            neighbors = self.graph[node]

            for neighbor in neighbors:
                new_path = path + [neighbor]

                if neighbor == end:
                    return new_path

                queue.append(new_path)

            visited.add(node)

    return None
```

```
    def create_graph_from_table(self,table):
        graph = UndirectedGraph()
        rows = len(table)
        cols = len(table[0])

        for row in table:
            for value in row:
                graph.add_vertex(value)

        for i in range(rows):
            for j in range(cols):
                current_value = table[i][j]

                if j + 1 < cols:
                    graph.add_edge(current_value, table[i][j + 1])

                if i + 1 < rows:
                    graph.add_edge(current_value, table[i + 1][j])

        self.graph = graph.graph



if __name__ == "__main__":
    graph = UndirectedGraph()
    for i in range(1,7):
        graph.add_vertex(i)
    edges = [
        (6,4),(4,3),(4,5),(5,2),(3,2),(5,1),(1,2)
    ]

    for edge in edges:
        graph.add_edge(edge[0], edge[1])
```

```python
print(graph.graph)
for x in graph.graph:
    print(f"{x} : {graph.get_degree(x)}")

start_node = 6
end_node = 1
path = graph.find_path(start_node, end_node)
print(f"path: {path}")



all_paths = graph.find_all_paths(start_node, end_node)
print(f"all path: {all_paths}")
```

Directed Graph:

```python
class DirectedGraph:
    graph = {}

    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        self.graph[vertex] = []

    def add_edge(self,vertex1, vertex2, isForwards = True):
        if isForwards:
            self.graph[vertex1].append(vertex2)
        else:
            self.graph[vertex2].append(vertex1)

    def get_degree(self, vertex):
        inDegree = 0
        for x in self.graph:
            if vertex in self.graph[x]:
                inDegree += 1
        return (len(self.graph[vertex]), inDegree)
```

```python
def find_path(self, start_vertex, end_vertex):
    path = []
    stack = [(start_vertex, [start_vertex])]

    while stack:
        (vertex, current_path) = stack.pop()

        if vertex == end_vertex:
            return current_path

        for neighbor in self.graph[vertex]:
            if neighbor not in current_path:
                stack.append((neighbor, current_path + [neighbor]))

    return None


def find_all_paths(self, start_vertex, end_vertex):
    all_paths = []
    stack = [(start_vertex, [start_vertex])]

    while stack:
        (vertex, current_path) = stack.pop()

        if vertex == end_vertex:
            all_paths.append(current_path)
        else:
            for neighbor in self.graph[vertex]:
                if neighbor not in current_path:
                    stack.append((neighbor, current_path + [neighbor]))

    return all_paths
```

```python
if __name__ == "__main__":
    g = DirectedGraph()
    vertices = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
    for vertex in vertices:
        g.add_vertex(vertex)

    edges = [
        ('A', 'B'),
        ('B', 'C'),
        ('B', 'D'),
        ('B', 'E'),
        ('C', 'E'),
        ('D', 'E'),
        ('E', 'F'),
        ('G', 'D')
    ]

    for edge in edges:
        g.add_edge(edge[0], edge[1])

    print("Graph structure:", g.graph)

    for x in g.graph:
        out_degree, in_degree = g.get_degree(x)
        print(f"{x}:OutDegree: {out_degree}, InDegree: {in_degree}")

    start_vertex = 'A'
    end_vertex = 'F'
    path = g.find_path(start_vertex, end_vertex)
    print(f"path: {path}")

    all_paths = g.find_all_paths(start_vertex, end_vertex)

    print(f"all paths: {all_paths}")
```

Task2:

Each pixel in an image represents a node and the nodes which are adjacent are connected with each other via 4-connectivity pattern. Suppose you have been given with a 4x4 grayscale image now perform the following tasks

a) **Decompose it into an undirected graph.**
b) **Show all the possible paths from pixel 100 and pixel 118.**

100	110	120	130
140	145	45	135
220	10	165	80
180	200	191	118

Code:

```
table = [
    [100, 110, 120, 130],
    [140, 145, 45, 135],
    [220, 10, 165, 80],
    [180, 200, 191, 118]
]

graph.create_graph_from_table(table)
path = graph.find_path(100, 118)
print(f"Path: {path}")

all_paths = graph.find_all_paths(100, 118)
print(f"Number of paths: {len(all_paths)}")
shortest_path = graph.shortest_path(100, 118)
print(f"Shortest path: {shortest_path}")
```

Output:

```
Path: [100, 110, 120, 130, 135, 80, 118]
Number of paths: 184
Shortest path: ([100, 110, 120, 130, 135, 80, 118], 128)
```

Task3:

Decompose the above image into an undirected graph where each pixel represents a node and the edge cost between adjacent nodes is computed by taking the absolute difference. Now segment the object between node 100 to node 118 by computing the shortest path.

Hint: Use nested dictionaries to represent graph with edge costs.

Code:

```
all_paths = graph.find_all_paths(100, 118)
```

Output:

```
Path: [100, 110, 120, 130, 135, 80, 118]
Number of paths: 184
Shortest path: ([100, 110, 120, 130, 135, 80, 118], 128)
```