**DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING**

**COLLEGE OF E&ME, NUST, RAWALPINDI**

AI & Decision Support Systems

Lab Report #2

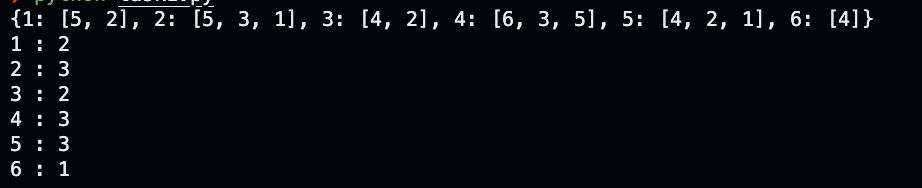
**Student Name: Nawab Aarij Imam**

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

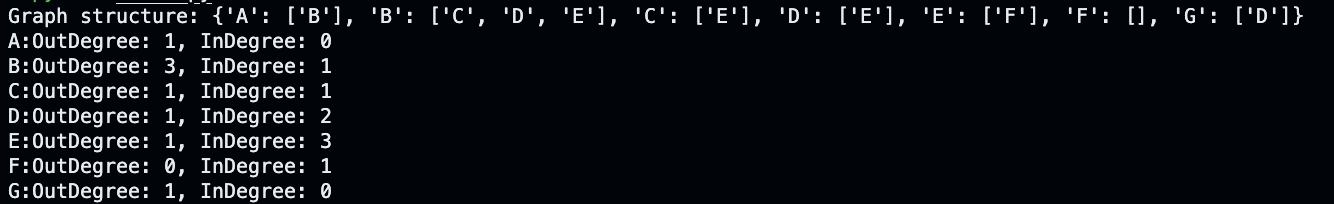
**Task1:**

**Q1: Perform the following tasks**

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



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



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



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



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



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



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





**Code:**

**Undirected Graph:**

class UndirectedGraph:

graph = {}

def \_\_init\_\_(self):

self.graph = {}

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

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])

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:**

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)

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

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**

1. **Decompose it into an undirected graph.**
2. **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}")

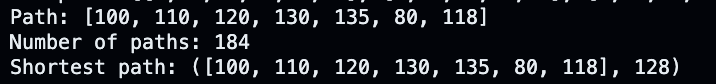
all\_paths = graph.find\_all\_paths(100, 118)

print(f"Number of paths: {len(all\_paths)}")

shortest\_path = graph.shortest\_path(100, 118)

print(f"Shortest path: {shortest\_path}")

**Output:**

****

**Task3:**

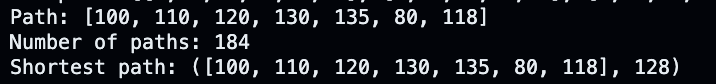
Decompose the above image into an undirected graph where each pixel represents a node and the edge cost between adjacent nodes is computed by taking the absolute difference. Now segment the object between node 100 to node 118 by computing the shortest path.

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

**Code:**

all\_paths = graph.find\_all\_paths(100, 118)

**Output:**

****