***Artificial Intelligence***

***CSL 411***

***Lab Journal 4***

****

**Ahmad Hassan**

**01-134191-002**

**BSCS-6B**

**Department of Computer Science**

**BAHRIA UNIVERSITY, ISLAMABAD**

**Lab # 4: Graphs in Python**

**Objectives:**

To implement the concepts of graphs in python.

**Tools Used:**

Spyder IDLE

**Submission Date:**

**Evaluation: Signatures of Lab Engineer:**

**Task # 1:**

Change the function find path to return shortest path.

**Program:**

class Graph:

    def \_\_init\_\_(self, nodes=None, edges=None):

        self.nodes, self.adj = [], {}

        if nodes != None:

            self.add\_nodes\_from(nodes)

        if edges != None:

            self.add\_edges\_from(edges)

    def length(self):

        return len(self.nodes)

    def traverse(self):

        return 'V: %s\nE: %s' % (self.nodes, self.adj)

    def \_\_str\_\_(self):

        string = ""

        for node in self.nodes:

            string += f"{node} -> {self.adj[node]}\n"

        return string

    def add\_node(self, n):

        if n not in self.nodes:

            self.nodes.append(n)

            self.adj[n] = []

    def add\_edge(self, u, v):  # undirected unweighted graph

        self.adj[u] = self.adj.get(u, []) + [v]

        self.adj[v] = self.adj.get(v, []) + [u]

    def number\_of\_nodes(self):

        return len(self.nodes)

    def number\_of\_edges(self):

        return sum(len(l) for \_, l in self.adj.items())

class DGraph(Graph):

    def add\_edge(self, u, v):

        self.adj[u] = self.adj.get(u, []) + [v]

class WGraph(Graph):

    def \_\_init\_\_(self, nodes=None, edges=None):

        self.nodes, self.adj, self.weight = [], {}, {}

        if nodes != None:

            self.add\_nodes\_from(nodes)

        if edges != None:

            self.add\_edges\_from(edges)

    def add\_edge(self, u, v, w):

        self.adj[u] = self.adj.get(u, []) + [v]

        self.adj[v] = self.adj.get(v, []) + [u]

        self.weight[(u, v)] = w

        self.weight[(v, u)] = w

    def get\_weight(self, u, v):

        return self.weight[(u, v)]

class DWGraph(WGraph):

    def add\_edge(self, u, v, w):

        self.adj[u] = self.adj.get(u, []) + [v]

        self.weight[(u, v)] = w

    def find\_path(self, start, end, path=[]):

        path = path + [start]

        if start == end:

            return path

        if start not in self.adj:

            return None

        for node in self.adj[start]:

            if node not in path:

                newpath = self.find\_path(node, end, path)

                if newpath:

                    return newpath

        return None

    def find\_shortest\_path(self, start, end, path=[]):

        path = path + [start]

        if start == end:

            return path

        if start not in self.adj:

            return None

        Shortest = None

        for node in self.adj[start]:

            if node not in path:

                newpath = self.find\_shortest\_path(node, end, path)

                if newpath:

                    if not Shortest or len(newpath) < len(Shortest):

                        Shortest = newpath

        return Shortest

directedWeightedGraph = DWGraph()

directedWeightedGraph.add\_node('A')

directedWeightedGraph.add\_node('B')

directedWeightedGraph.add\_node('C')

directedWeightedGraph.add\_node('D')

directedWeightedGraph.add\_node('E')

directedWeightedGraph.add\_node('F')

directedWeightedGraph.add\_edge('A', 'B', 2)

directedWeightedGraph.add\_edge('A', 'C', 1)

directedWeightedGraph.add\_edge('B', 'C', 2)

directedWeightedGraph.add\_edge('B', 'D', 5)

directedWeightedGraph.add\_edge('C', 'D', 1)

directedWeightedGraph.add\_edge('C', 'F', 3)

directedWeightedGraph.add\_edge('D', 'C', 1)

directedWeightedGraph.add\_edge('D', 'E', 4)

directedWeightedGraph.add\_edge('E', 'F', 3)

directedWeightedGraph.add\_edge('F', 'C', 1)

directedWeightedGraph.add\_edge('F', 'E', 2)

print(directedWeightedGraph)

print("\nPath is")

print(directedWeightedGraph.find\_path('A', 'D'))

print("\nShortest Path is")

print(directedWeightedGraph.find\_shortest\_path('A', 'D'))

**Result/Output:**

Text

Description automatically generated**Analysis/Conclusion:**

**Task # 2:**

Consider a simple (directed) graph (digraph) having six nodes (A-F) and the following arcs (directed edges) with respective cost of edge given in parentheses:

A -> B (2)

A -> C (1)

B -> C (2)

B -> D (5)

C -> D (1)

C -> F (3)

D -> C (1)

D -> E (4)

E -> F (3)

F -> C (1)

F -> E (2)

Using the code for a directed weighted graph in Example 2, instantiate an object of DWGraph in \_\_main\_\_, add the nodes and edges of the graph using the relevant functions, and implement a function find\_path() that takes starting and ending nodes as arguments and returns at least one path (if one exists) between those two nodes. The function should also keep track of the cost of the path and return the total cost as well as the path. Print the path and its cost in \_\_main\_\_.

**Program:**

class Graph:

    def \_\_init\_\_(self, nodes=None, edges=None):

        self.nodes, self.adj = [], {}

        if nodes != None:

            self.add\_nodes\_from(nodes)

        if edges != None:

            self.add\_edges\_from(edges)

    def length(self):

        return len(self.nodes)

    def traverse(self):

        return 'V: %s\nE: %s' % (self.nodes, self.adj)

    def \_\_str\_\_(self):

        string = ""

        for node in self.nodes:

            string += f"{node} -> {self.adj[node]}\n"

        return string

    def add\_node(self, n):

        if n not in self.nodes:

            self.nodes.append(n)

            self.adj[n] = []

    def add\_edge(self, u, v):  # undirected unweighted graph

        self.adj[u] = self.adj.get(u, []) + [v]

        self.adj[v] = self.adj.get(v, []) + [u]

    def number\_of\_nodes(self):

        return len(self.nodes)

    def number\_of\_edges(self):

        return sum(len(l) for \_, l in self.adj.items())

class DGraph(Graph):

    def add\_edge(self, u, v):

        self.adj[u] = self.adj.get(u, []) + [v]

class WGraph(Graph):

    def \_\_init\_\_(self, nodes=None, edges=None):

        self.nodes, self.adj, self.weight = [], {}, {}

        if nodes != None:

            self.add\_nodes\_from(nodes)

        if edges != None:

            self.add\_edges\_from(edges)

    def add\_edge(self, u, v, w):

        self.adj[u] = self.adj.get(u, []) + [v]

        self.adj[v] = self.adj.get(v, []) + [u]

        self.weight[(u, v)] = w

        self.weight[(v, u)] = w

    def get\_weight(self, u, v):

        return self.weight[(u, v)]

class DWGraph(WGraph):

    def add\_edge(self, u, v, w):

        self.adj[u] = self.adj.get(u, []) + [v]

        self.weight[(u, v)] = w

        self.pathCost=0

    def find\_path(self, start, end, path=[]):

        path = path + [start]

        if start == end:

            return path

        if start not in self.adj:

            return None

        for node in self.adj[start]:

            if node not in path:

                newpath = self.find\_path(node, end, path)

                if newpath:

                    return newpath

        return None

    def find\_shortest\_path(self, start, end, path=[],cost=0):

        path = path + [start]

        if start == end:

            self.pathCost=cost

            return path,cost

        if start not in self.adj:

            return None,cost

        for node in self.adj[start]:

            if node not in path:

                cost=cost+self.get\_weight(start,node)

                newpath = self.find\_shortest\_path(node, end, path,cost)

                if newpath:

                    return newpath

        return None,cost

directedWeightedGraph = DWGraph()

directedWeightedGraph.add\_node('A')

directedWeightedGraph.add\_node('B')

directedWeightedGraph.add\_node('C')

directedWeightedGraph.add\_node('D')

directedWeightedGraph.add\_node('E')

directedWeightedGraph.add\_node('F')

directedWeightedGraph.add\_edge('A', 'B', 2)

directedWeightedGraph.add\_edge('A', 'C', 1)

directedWeightedGraph.add\_edge('B', 'C', 2)

directedWeightedGraph.add\_edge('B', 'D', 5)

directedWeightedGraph.add\_edge('C', 'D', 1)

directedWeightedGraph.add\_edge('C', 'F', 3)

directedWeightedGraph.add\_edge('D', 'C', 1)

directedWeightedGraph.add\_edge('D', 'E', 4)

directedWeightedGraph.add\_edge('E', 'F', 3)

directedWeightedGraph.add\_edge('F', 'C', 1)

directedWeightedGraph.add\_edge('F', 'E', 2)

print(directedWeightedGraph)

print("\nPath is")

print(directedWeightedGraph.find\_path('A', 'F'))

print("\nShortest Path is")

print(directedWeightedGraph.find\_shortest\_path('A', 'F'))

**Result/Output:**

Text

Description automatically generated

**Analysis/Conclusion:**