***Artificial Intelligence***

***CSL 411***

***Lab Journal***

****

**Saifullah Bhatti**

**01-134182-047**

**6-A**

**Department of Computer Science**

**BAHRIA UNIVERSITY, ISLAMABAD**

**Lab # : 4**

**Objectives:**

**Tools Used:**

**Submission Date:**

**Evaluation: Signatures of Lab Engineer:**

**Task # 1**

Change the function find path to return shortest path.

**Procedure/Program:**

class Graph:

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

"""Initialize a graph object.

Args:

nodes: Iterator of nodes. Each node is an object.

edges: Iterator of edges. Each edge is a tuple of 2 nodes.

"""

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

if nodes != None:

self.add\_nodes\_from(nodes)

if edges != None:

self.add\_edges\_from(edges)

def length(self):

"""Returns the number of nodes in the graph.

>>> g = Graph(nodes=[x for x in range(7)])

>>> len(g)

7

"""

return len(self.nodes)

def traverse(self):

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

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()) // 2

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):

"""Initialize a graph object.

Args:

nodes: Iterator of nodes. Each node is an object.

edges: Iterator of edges. Each edge is a tuple of 2 nodes and a weight.

"""

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 get\_edges(self, u):

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\_pathC(self, start, end, path=[], cost=0):

path = path + [start]

if start == end:

return path, cost

if start not in self.adj:

return None, cost

for node in self.adj[start]:

if node not in path:

min=self.get\_weight(start,node)

print(min)

newpath, cost = self.find\_pathC(node, end, path, cost + self.get\_weight(start, node))

if newpath:

return newpath, cost

return None, cost

Dw=DWGraph()

Dw.add\_node('A')

Dw.add\_node('B')

Dw.add\_node('C')

Dw.add\_node('D')

Dw.add\_node('E')

Dw.add\_node('F')

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

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

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

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

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

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

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

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

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

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

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

print(Dw.traverse())

print(Dw.find\_pathC("A", "F"))

**Result/Output:**

****

**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\_\_.

**Procedure/Program:**

class Graph:

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

"""Initialize a graph object.

Args:

nodes: Iterator of nodes. Each node is an object.

edges: Iterator of edges. Each edge is a tuple of 2 nodes.

"""

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

if nodes != None:

self.add\_nodes\_from(nodes)

if edges != None:

self.add\_edges\_from(edges)

def length(self):

"""Returns the number of nodes in the graph.

>>> g = Graph(nodes=[x for x in range(7)])

>>> len(g)

7

"""

return len(self.nodes)

def traverse(self):

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

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()) // 2

def get\_vertix(self):

return self.nodes

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):

"""Initialize a graph object.

Args:

nodes: Iterator of nodes. Each node is an object.

edges: Iterator of edges. Each edge is a tuple of 2 nodes and a weight.

"""

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 get\_edges(self, u):

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\_pathC(self, start, end, path=[], cost=0):

path = path + [start]

if start == end:

return path, cost

if start not in self.adj:

return None, cost

for node in self.adj[start]:

if node not in path:

min=self.get\_weight(start,node)

print(min)

newpath, cost = self.find\_pathC(node, end, path, cost + self.get\_weight(start, node))

if newpath:

return newpath, cost

return None, cost

Dw=DWGraph()

Dw.add\_node('A')

Dw.add\_node('B')

Dw.add\_node('C')

Dw.add\_node('D')

Dw.add\_node('E')

Dw.add\_node('F')

Dw.add\_edge('A','B',5)

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

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

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

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

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

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

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

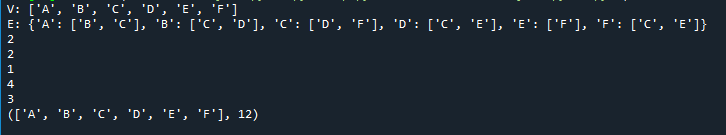
Dw.add\_edge('E','F',3)

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

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

print(Dw.find\_path("A", "E"))

**Result/Output:**

****

**Analysis/Conclusion:**