Artificial Intelligence

CSL 411

Lab Journal 4

Student Name:
Enrolment No.:
Class and Section:

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 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("\nSimple Path is")
print(directedWeightedGraph.find_path('A', 'E') )
print("\nShortest Path is")
print(directedWeightedGraph.find_shortest_path('A','E'))
```

Result/Output:

```
In [120]: runfile('C:/Users/HP/.spyder-py3/temp.py',
wdir='C:/Users/HP/.spyder-py3')

Simple Path is
['A', 'B', 'C', 'D', 'E']

Shortest Path is
['A', 'B', 'D', 'E']
```

Analysis/Conclusion:

Find_Path function recursively finds the path between given two nodes and return the path.
Find_Shortest_Path function recursively finds the shortest path between given two nodes and return the shortest path.

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 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:
                print(node,end)
                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("\nSimple Path is")
print(directedWeightedGraph.find_path('A', 'E') )
print("\nShortest Path with cost is")
print(directedWeightedGraph.find_shortest_path('A','E'))
```

Result/Output:

```
In [136]: runfile('C:/Users/HP/.spyder-py3/temp.py',
wdir='C:/Users/HP/.spyder-py3')

Simple Path is
['A', 'B', 'C', 'D', 'E']

Shortest Path with cost is
B E
C E
D E
E E
(['A', 'B', 'C', 'D', 'E'], 9)
```

Analysis/Conclusion:

Find_Shortest_Path function recursively finds the shortest path between given two nodes and return the path with total cost of path traversal.