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	s in python.			
Tools Used:				
Spyder IDLE				
	Submission Date:			
Evaluation:	Signatures of Lab Engineer:			
	~- g			

Task # 1:

Change the function find path to return shortest path.

```
Program:
                                                class DGraph(Graph):
                                                    def add_edge(self, u, v):
class Graph:
                                                        self.adj[u] = self.adj.get(u,
    def __init__(self, nodes=None,
                                                []) + [v]
edges=None):
                                                class WGraph(Graph):
        self.nodes, self.adj = [], {}
                                                    def __init__(self, nodes=None,
        if nodes != None:
                                                edges=None):
            self.add nodes from(nodes)
                                                        self.nodes, self.adj,
        if edges != None:
                                                self.weight = [], {}, {}
            self.add_edges_from(edges)
                                                        if nodes != None:
    def length(self):
                                                            self.add_nodes_from(nodes)
        return len(self.nodes)
                                                        if edges != None:
                                                            self.add_edges_from(edges)
    def traverse(self):
        return 'V: %s\nE: %s' %
                                                    def add_edge(self, u, v, w):
(self.nodes, self.adj)
                                                        self.adj[u] = self.adj.get(u,
    def __str__(self):
                                                []) + [v]
        string = ""
                                                        self.adj[v] = self.adj.get(v,
        for node in self.nodes:
                                                []) + [u]
            string += f"{node} ->
                                                        self.weight[(u, v)] = w
{self.adj[node]}\n"
                                                        self.weight[(v, u)] = w
        return string
    def add node(self, n):
                                                    def get_weight(self, u, v):
        if n not in self.nodes:
                                                        return self.weight[(u, v)]
            self.nodes.append(n)
                                                class DWGraph(WGraph):
            self.adj[n] = []
                                                    def add_edge(self, u, v, w):
    def add_edge(self, u, v): #
                                                        self.adj[u] = self.adj.get(u,
undirected unweighted graph
                                                []) + [v]
        self.adj[u] = self.adj.get(u,
                                                        self.weight[(u, v)] = w
[1) + [v]
                                                    def find_path(self, start, end,
        self.adj[v] = self.adj.get(v,
                                                path=[]):
[]) + [u]
                                                        path = path + [start]
                                                        if start == end:
    def number_of_nodes(self):
                                                            return path
        return len(self.nodes)
                                                        if start not in self.adj:
                                                            return None
    def number_of_edges(self):
                                                        for node in self.adj[start]:
        return sum(len(1) for , 1 in
                                                            if node not in path:
self.adj.items())
                                                                newpath =
                                                self.find_path(node, end, path)
```

```
directedWeightedGraph.add_edge('A', 'B',
                if newpath:
                    return newpath
        return None
                                                directedWeightedGraph.add_edge('A', 'C',
    def find shortest path(self, start,
end, path=[]):
                                                directedWeightedGraph.add_edge('B', 'C',
        path = path + [start]
                                                2)
        if start == end:
                                                directedWeightedGraph.add edge('B', 'D',
            return path
        if start not in self.adj:
                                                directedWeightedGraph.add edge('C', 'D',
            return None
        Shortest = None
                                                directedWeightedGraph.add edge('C', 'F',
        for node in self.adj[start]:
                                                3)
            if node not in path:
                                                directedWeightedGraph.add edge('D', 'C',
                newpath =
self.find_shortest_path(node, end, path)
                                                directedWeightedGraph.add_edge('D', 'E',
                if newpath:
                    if not Shortest or
                                                directedWeightedGraph.add_edge('E', 'F',
len(newpath) < len(Shortest):</pre>
                                                3)
                        Shortest =
                                                directedWeightedGraph.add edge('F', 'C',
newpath
        return Shortest
                                                directedWeightedGraph.add edge('F', 'E',
directedWeightedGraph = DWGraph()
directedWeightedGraph.add node('A')
                                                print(directedWeightedGraph)
directedWeightedGraph.add node('B')
                                                print("\nPath is")
                                                print(directedWeightedGraph.find_path('A
directedWeightedGraph.add node('C')
                                                ', 'D'))
directedWeightedGraph.add node('D')
                                                print("\nShortest Path is")
directedWeightedGraph.add node('E')
directedWeightedGraph.add node('F')
                                                print(directedWeightedGraph.find_shortes
                                                t path('A', 'D'))
```

Result/Output:

```
PS D:\STUDY\AI LAB\Code\Lab Codes> & "C:/L

"d:\STUDY\AI LAB\Code\Lab Codes\Lab 4/tas

A -> ['B', 'C']

B -> ['C', 'D']

C -> ['D', 'F']

D -> ['C', 'E']

E -> ['F']

F -> ['C', 'E']

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

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

PS D:\STUDY\AI LAB\Code\Lab Codes>
```

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)
```

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:
                                                       return 'V: %s\nE: %s' %
                                               (self.nodes, self.adj)
   def __init__(self, nodes=None,
edges=None):
                                                   def __str__(self):
        self.nodes, self.adj = [], {}
        if nodes != None:
                                                       string = ""
            self.add_nodes_from(nodes)
                                                       for node in self.nodes:
                                                           string += f"{node} ->
        if edges != None:
            self.add_edges_from(edges)
                                       {self.adj[node]}\n"
                                                       return string
   def length(self):
        return len(self.nodes)
                                                   def add_node(self, n):
                                                       if n not in self.nodes:
   def traverse(self):
                                                           self.nodes.append(n)
```

```
self.adj[n] = []
                                               class DWGraph(WGraph):
                                                    def add edge(self, u, v, w):
    def add_edge(self, u, v): #
                                                        self.adj[u] = self.adj.get(u,
undirected unweighted graph
                                                []) + [v]
        self.adj[u] = self.adj.get(u,
                                                        self.weight[(u, v)] = w
                                                        self.pathCost=0
[]) + [v]
                                                   def find path(self, start, end,
        self.adj[v] = self.adj.get(v,
[]) + [u]
                                                path=[]):
                                                        path = path + [start]
    def number_of_nodes(self):
                                                        if start == end:
        return len(self.nodes)
                                                            return path
                                                        if start not in self.adj:
    def number_of_edges(self):
                                                            return None
        return sum(len(1) for , 1 in
                                                        for node in self.adj[start]:
self.adj.items())
                                                            if node not in path:
                                                                newpath =
                                                self.find_path(node, end, path)
                                                                if newpath:
class DGraph(Graph):
                                                                    return newpath
    def add edge(self, u, v):
                                                        return None
        self.adj[u] = self.adj.get(u,
[]) + [v]
                                                    def find_shortest_path(self, start,
                                                end, path=[],cost=0):
                                                        path = path + [start]
class WGraph(Graph):
                                                        if start == end:
    def __init__(self, nodes=None,
                                                            self.pathCost=cost
edges=None):
                                                            return path, cost
        self.nodes, self.adj,
                                                        if start not in self.adj:
self.weight = [], {}, {}
                                                            return None, cost
        if nodes != None:
                                                        for node in self.adj[start]:
            self.add_nodes_from(nodes)
                                                            if node not in path:
        if edges != None:
                                                                cost=cost+self.get_weigh
            self.add_edges_from(edges)
                                               t(start, node)
                                                                newpath =
    def add_edge(self, u, v, w):
                                                self.find_shortest_path(node, end,
        self.adj[u] = self.adj.get(u,
                                                path,cost)
[]) + [v]
                                                                if newpath:
        self.adj[v] = self.adj.get(v,
                                                                    return newpath
[]) + [u]
                                                        return None, cost
        self.weight[(u, v)] = w
                                                directedWeightedGraph = DWGraph()
        self.weight[(v, u)] = w
                                                directedWeightedGraph.add node('A')
                                                directedWeightedGraph.add_node('B')
    def get_weight(self, u, v):
                                                directedWeightedGraph.add node('C')
        return self.weight[(u, v)]
                                                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('C', 'F',
3)
```

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(directedWeightedGraph.find_shortes t_path('A', 'F'))

Result/Output:

```
PS D:\STUDY\AI LAB\Code\Lab Codes> & "C:/
  "d:/STUDY/AI LAB/Code/Lab Codes/Lab 4/ta
A -> ['B', 'C']
B -> ['C', 'D']
C -> ['D', 'F']
D -> ['C', 'E']
E -> ['F']
F -> ['C', 'E']

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

Shortest Path is
(['A', 'B', 'C', 'D', 'E', 'F'], 12)
PS D:\STUDY\AI LAB\Code\Lab Codes>
```

Analysis/Conclusion:		