Artificial Intelligence

CSL 411

Lab Journal 4

Tauheed Butt
01-134191-068
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:

GRAPH.py

```python
class Graph:
    def __init__(self, nodes, is_directed=False):
        self.nodes = nodes
        self.adj_list = {}
        self.is_directed = is_directed
        for node in self.nodes:
            self.adj_list[node] = []

    def __str__(self):
        string = ""
        for node in self.nodes:
            string += f"{node} -> {self.adj_list[node]}\n"
        return string

    def create_directed_edge(self, src, dst, cost):
        if dst in self.nodes:
            self.adj_list[src].append((dst, cost))


    def create_undirected_edge(self, src, dst):
        if (dst not in self.nodes) or (src not in self.nodes) or (len(dst) > 1) or
(len(src) > 1):
            return
        self.adj_list[src].append(dst, cost)
        self.adj_list[dst].append(src, cost)

    def add_edge(self, src, dst, cost):
        if self.is_directed:
            self.create_directed_edge(src, dst, cost)
        else:
            self.create_undirected_edge(src, dst, cost)
```

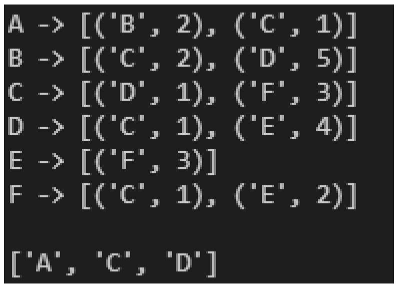
PriorityQueue.py

```
# Priority Queue class taken from https://www.geeksforgeeks.org/priority-queue-in-
python/
# to make the assignment faster

class PriorityQueue:
    def __init__(self, items):
        self.queue = []
        for item in items:
            self.push(item)

    def __str__(self):
        return ' '.join([str(i) for i in self.queue])

    # for checking if the queue is empty
    def is_empty(self):
        return len(self.queue) == 0

    # for inserting an element in the queue
    def push(self, data):
        self.queue.append(data)

    # for popping an element based on Priority
    def pop(self):
        try:
            max = 0
            for i in range(len(self.queue)):
                if self.queue[i][1] == self.queue[max][1]:
                    if self.queue[i][0] > self.queue[max][0] and
len(self.queue[i][0])==1 and len(self.queue[max][0])==1:
                        max = i
                if self.queue[i][1] < self.queue[max][1]:
                    max = i
            item = self.queue[max]
            del self.queue[max]
            return item
        except IndexError:
            print()
            exit()
```

task_1.py

```python
from Graph import *
from PriorityQueue import *


def shortest_path(g, start, end, path=[]):
    path.append(start)
    if start == end:
        return path
    if start not in g.nodes:
        return None
    q = PriorityQueue(g.adj_list[start])
    while not q.is_empty():
        node = q.pop()
        newpath = shortest_path(g, node[0], end, path)
        if newpath:
            return newpath
    return None



nodes = ['A', 'B', 'C', 'D', 'E', 'F']
edges = [
    ('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)
]

g = Graph(nodes, True)

for src,dst,cost in edges:
    g.add_edge(src, dst, cost)

print(g)
print(shortest_path(g, 'A', 'D'))
```

Result/Output:



```
A -> [('B', 2), ('C', 1)]
B -> [('C', 2), ('D', 5)]
C -> [('D', 1), ('F', 3)]
D -> [('C', 1), ('E', 4)]
E -> [('F', 3)]
F -> [('C', 1), ('E', 2)]

['A', 'C', 'D']
```

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__.

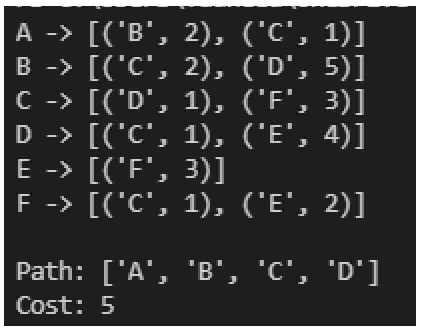
Program:

Same Files used as before, except changings in __main__ file:

Task_2.py

```python
from Graph import *
from PriorityQueue import *


def find_path(g, start, end, cost=0, path=[]):
    path.append(start)
    if start == end:
        return path,cost
    if start not in g.nodes:
        return None
    for node in g.adj_list[start]:
        cost += node[1]
        newpath,cost = find_path(g, node[0], end, cost, path)
        if newpath:
            return newpath,cost
    return None



nodes = ['A', 'B', 'C', 'D', 'E', 'F']
edges = [
    ('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)
]

g = Graph(nodes, True)

for src,dst,cost in edges:
    g.add_edge(src, dst, cost)

print(g)
path,cost = find_path(g, 'A', 'D')
print(f'Path: {path}\nCost: {cost}')
```

Result/Output:



```
A -> [('B', 2), ('C', 1)]
B -> [('C', 2), ('D', 5)]
C -> [('D', 1), ('F', 3)]
D -> [('C', 1), ('E', 4)]
E -> [('F', 3)]
F -> [('C', 1), ('E', 2)]

Path: ['A', 'B', 'C', 'D']
Cost: 5
```