NAMAL UNIVERSITY MIANWALI

DEPARTMENT OF ELECTRICAL ENGINEERING

DATA STRUCTURE AND ALGORITHM

LAB # 12

REPORT

Title : Implementation of Shortest Path Algorithm and Minimum Spanning Tree in Python

Name	Fahim-Ur-Rehman Shah
Roll No	NIM-BSEE-2021-24
Intructor	Ms. Naureen Shaukat
Date	06-July-2023
Marks	

1. Lab Objectives

The objective of this lab is to implement shortest path algorithm (Dijkstra)

2. Lab Outcomes

- CLO 1: Recognize the usage of fundamental Data structure using Python Programming Language.
- CLO 2: Analyzing computation cost of different algorithms.
- CLO 3: Demonstrate solution to real life problems using appropriate data structures.

3. Equipment

- Software o IDLE (Python 3.11)

4. Instructions

1. This is an individual lab. You will perform the tasks individually and submit a report.
2. Some of these tasks (marked as 'Example') are for practice purposes only while others (marked as 'Task') have to be answered in the report.
3. When asked to display an output in the task, either save it as jpeg or take a screenshot, in order to insert it in the report.
4. The report should be submitted on the given template, including:
    a. Code (copy and pasted, NOT a screenshot)
    b. Output figure (as instructed in 3)
    c. Explanation where required
5. The report should be properly formatted, with easy to read code and easy to see figures.
6. Plagiarism or any hint thereof will be dealt with strictly.
7. Late submission of report is allowed within 03 days after lab with 20% deduction of marks every day.
8. You have to submit report in pdf format (Reg.X_DSA_LabReportX.pdf).

5. Background

Weighted Graph

Until now, we have worked for the unweighted graphs. In this lab you have to make the weighted graph in python. A ***weighted graph*** is a graph that has a numeric (for example, integer) label $w(e)$ associated with each edge e, called the ***weight*** of edge e. For $e = (u,v)$, we let notation $w(u,v) = w(e)$.

Use the previously implemented code for unweighted graph. Add another data item of weight in the edge class.

The implementation code for the weighted graph is provided on the QOBE.

Dijkstra Algorithm for Shortest Path

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. The algorithm is given as,

Algorithm ShortestPath(G, s):
Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.
Output: The length of a shortest path from s to v for each vertex v of G.
Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
{pull a new vertex u into the cloud}
u = value returned by Q.remove_min()
for each vertex v adjacent to u such that v is in Q **do**
{perform the *relaxation* procedure on edge (u, v)}
if $D[u] + w(u, v) < D[v]$ **then**
$D[v] = D[u] + w(u, v)$
Change to $D[v]$ the key of vertex v in Q.
return the label $D[v]$ of each vertex v

There are few concepts that are new to the students for implementing the Dijkstra algorithm.

1. Priority Queue
   There is a built-in module of queue in python. You can import the priority queue class from the module queue. A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first. However, if elements with the same priority occur, they are served according to their order in the queue. * An example of priority queue is provided on QOBE.

2. Defining infinity in python.
   You can define the infinity in python using float('inf') in python. * An example of priority queue is provided on QOBE.

Lab Task 1:

For the given graph in Fig. 1, implement the Dijkstra's algorithm in Python.

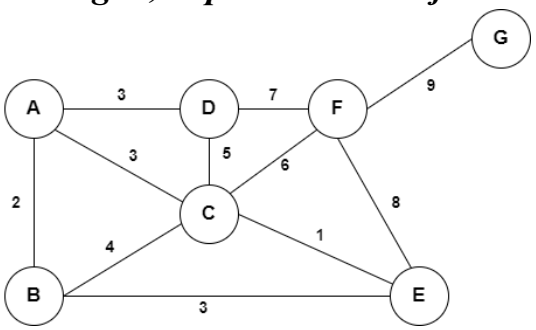

Figure 1 Graph

Code :

```
import sys

class Vertex:
    def __init__(self, name):
        self.name = name
        self.neighbors = []
        self.distance = sys.maxsize
```

```
        self.visited = False
        self.previous = None

    def add_neighbor(self, neighbor, weight):
        self.neighbors.append((neighbor, weight))

    def __lt__(self, other):
        return self.distance < other.distance

class Graph:
    def __init__(self):
        self.vertices = {}

    def add_vertex(self, name):
        vertex = Vertex(name)
        self.vertices[name] = vertex
        return vertex

    def get_vertex(self, name):
        if name in self.vertices:
            return self.vertices[name]
        else:
            return None

    def add_edge(self, start, weight, end):
        self.vertices[start.name].add_neighbor(end.name, weight)

    def print_graph(self):
        for vertex_name, vertex in self.vertices.items():
            neighbors = ", ".join([f"{neighbor} ({weight})" for neighbor, weight in
vertex.neighbors])
            print(f"{vertex_name}: {neighbors}")

    def dijkstra(self, start):
        start.distance = 0
        unvisited_vertices = list(self.vertices.values())

        while unvisited_vertices:
            current_vertex = min(unvisited_vertices)
            unvisited_vertices.remove(current_vertex)

            for neighbor_name, weight in current_vertex.neighbors:
                neighbor = self.get_vertex(neighbor_name)
                if neighbor.visited:
                    continue

                new_distance = current_vertex.distance + weight
                if new_distance < neighbor.distance:
                    neighbor.distance = new_distance
```

```
                    neighbor.previous = current_vertex

                current_vertex.visited = True

        def get_shortest_path(self, end):
            path = []
            current_vertex = self.get_vertex(end)
            while current_vertex:
                path.insert(0, current_vertex.name)
                current_vertex = current_vertex.previous
            return path


    # Create a graph
    graph = Graph()

    # Add vertices
    vertex_a = graph.add_vertex("A")
    vertex_b = graph.add_vertex("B")
    vertex_c = graph.add_vertex("C")
    vertex_d = graph.add_vertex("D")
    vertex_e = graph.add_vertex("E")
    vertex_f = graph.add_vertex("F")
    vertex_g = graph.add_vertex("G")

    # Add edges with weights
    graph.add_edge(vertex_a, 2, vertex_b)
    graph.add_edge(vertex_a, 3, vertex_d)
    graph.add_edge(vertex_a, 3, vertex_c)
    graph.add_edge(vertex_b, 4, vertex_c)
    graph.add_edge(vertex_b, 3, vertex_e)
    graph.add_edge(vertex_d, 7, vertex_f)
    graph.add_edge(vertex_d, 5, vertex_c)
    graph.add_edge(vertex_c, 6, vertex_f)
    graph.add_edge(vertex_c, 1, vertex_e)
    graph.add_edge(vertex_f, 8, vertex_e)
    graph.add_edge(vertex_f, 9, vertex_g)

    # Run Dijkstra's algorithm
    graph.dijkstra(vertex_a)

    # Get the shortest path to vertex G
    shortest_path = graph.get_shortest_path(vertex_g.name)
    print("Shortest Path: A to G :: ", shortest_path)
    print()
    shortest_path = graph.get_shortest_path(vertex_e.name)
    print("Shortest Path: A to E :: ", shortest_path)
```

Output :

PS E:\Semester 4\Data Structure and Algorithm\Lab\Lab 12\New folder> python -u "e:\Semester 4\Data Structure and Algorithm\Lab\Lab 12\New folder\DJK_algo.py"

Shortest Path: A to G :: ['A', 'C', 'F', 'G']

Shortest Path: A to E :: ['A', 'C', 'E']

PS E:\Semester 4\Data Structure and Algorithm\Lab\Lab 12\New folder>

Prims Algorithm for Minimum Spanning Tree

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which form a tree that includes every vertex has the minimum sum of weights among all the trees that can be formed from the graph.

Algorithm for Prims based MST is given as,

Algorithm PrimJarnik(G):
Input: An undirected, weighted, connected graph G with n vertices and m edges
Output: A minimum spanning tree T for G
Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
$D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
$(u, e) =$ value returned by Q.remove_min()
Connect vertex u to T using edge e.
for each edge $e' = (u, v)$ such that v is in Q **do**
{check if edge (u, v) better connects v to T}
if $w(u, v) < D[v]$ **then**
$D[v] = w(u, v)$
Change the key of vertex v in Q to $D[v]$.
Change the value of vertex v in Q to (v, e').
return the tree T

Lab Task 2: **Make a minimum spanning tree from the graph provided in Fig.1 using prims algorithm.**

Code :

```
import heapq
class Vertex:
    def __init__(self, key):
        self.key = key
        self.neighbors = []

    def add_neighbor(self, neighbor, weight):
        self.neighbors.append((weight, neighbor))

    def get_neighbors(self):
```

```
        return self.neighbors

    def __lt__(self, other):
        return False

    def __le__(self, other):
        return False

    def __gt__(self, other):
        return False

    def __ge__(self, other):
        return False

class Graph:
    def __init__(self):
        self.vertices = []

    def add_vertex(self, key):
        vertex = Vertex(key)
        self.vertices.append(vertex)
        return vertex

    def add_edge(self, vertex1, weight, vertex2):
        vertex1.add_neighbor(vertex2, weight)
        vertex2.add_neighbor(vertex1, weight)

    def print_graph(self):
        for vertex in self.vertices:
            print("Vertex", vertex.key)
            for weight, neighbor in vertex.get_neighbors():
                print(" -> Neighbor:", neighbor.key, "Weight:", weight)

    def minimum_spanning_tree(self, start_vertex):
        mst = []
        heap = []
        visited = set()


        visited.add(start_vertex)

        for weight, neighbor in start_vertex.get_neighbors():
            heapq.heappush(heap, (weight, start_vertex, neighbor))

        while heap:

            weight, source, destination = heapq.heappop(heap)

            if destination not in visited:
```

```
                mst.append((source.key, destination.key, weight))


                visited.add(destination)

                for weight, neighbor in destination.get_neighbors():
                    heapq.heappush(heap, (weight, destination, neighbor))

        return mst

    # Create a graph
    graph = Graph()

    # Add vertices
    vertex_a = graph.add_vertex("A")
    vertex_b = graph.add_vertex("B")
    vertex_c = graph.add_vertex("C")
    vertex_d = graph.add_vertex("D")
    vertex_e = graph.add_vertex("E")
    vertex_f = graph.add_vertex("F")
    vertex_g = graph.add_vertex("G")

    # Add edges with weights
    e1 = graph.add_edge(vertex_a, 2, vertex_b)
    e2 = graph.add_edge(vertex_a, 3, vertex_d)
    e3 = graph.add_edge(vertex_a, 3, vertex_c)
    e4 = graph.add_edge(vertex_b, 4, vertex_c)
    e5 = graph.add_edge(vertex_b, 3, vertex_e)
    e6 = graph.add_edge(vertex_d, 7, vertex_f)
    e7 = graph.add_edge(vertex_d, 5, vertex_c)
    e8 = graph.add_edge(vertex_c, 6, vertex_f)
    e9 = graph.add_edge(vertex_c, 1, vertex_e)
    e10 = graph.add_edge(vertex_f, 8, vertex_e)
    e11 = graph.add_edge(vertex_f, 9, vertex_g)

    # Print the graph
    #graph.print_graph()

    # Find the minimum spanning tree starting from vertex A
    minimum_spanning_tree = graph.minimum_spanning_tree(vertex_a)
    print("\nMinimum Spanning Tree:")
    for edge in minimum_spanning_tree:
        print(edge)
```

Output :

```
PS E:\Semester 4\Data Structure and Algorithm\Lab\Lab 12\New folder> python -u "e:\Semester 4\Data Structure and Algorithm\Lab\Lab 12\New folder\Prims_algo.py"

    Minimum Spanning Tree:
    ('A', 'B', 2)
    ('A', 'D', 3)
    ('B', 'E', 3)
    ('E', 'C', 1)
    ('C', 'F', 6)
    ('F', 'G', 9)

PS E:\Semester 4\Data Structure and Algorithm\Lab\Lab 12\New folder>
```

Krushkal Algorithm for Minimum Spanning Tree

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which form a tree that includes every vertex has the minimum sum of weights among all the trees that can be formed from the graph.

Algorithm Kruskal(G):
Input: A simple connected weighted graph G with n vertices and m edges
Output: A minimum spanning tree T for G
for each vertex v in G **do**
Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ {T will ultimately contain the edges of the MST}
while T has fewer than $n - 1$ edges **do**
(u, v) = value returned by Q.remove_min()
Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
if $C(u) \neq C(v)$ **then**
Add edge (u, v) to T.
Merge $C(u)$ and $C(v)$ into one cluster.
return tree T

Lab Task 3: ***Make a minimum spanning tree from the graph provided in Fig.1 using krushkal algorithm.***

Code :

```
class Vertex:
    def __init__(self, name):
        self.name = name
        self.parent = None

class Graph:
    def __init__(self):
        self.vertices = []
        self.edges = []

    def add_vertex(self, name):
        vertex = Vertex(name)
```

```
        self.vertices.append(vertex)
        return vertex

    def add_edge(self, start_vertex, weight, end_vertex):
        self.edges.append((start_vertex, weight, end_vertex))

    def find(self, vertex):
        if vertex.parent is None:
            return vertex
        return self.find(vertex.parent)

    def union(self, vertex1, vertex2):
        root1 = self.find(vertex1)
        root2 = self.find(vertex2)
        root2.parent = root1

    def kruskal(self):
        minimum_spanning_tree = []
        sorted_edges = sorted(self.edges, key=lambda x: x[1])
        for edge in sorted_edges:
            start_vertex, weight, end_vertex = edge
            if self.find(start_vertex) != self.find(end_vertex):
                minimum_spanning_tree.append(edge)
                self.union(start_vertex, end_vertex)

        return minimum_spanning_tree

    def print_graph(self):
        for vertex in self.vertices:
            print("Vertex:", vertex.name)
        for edge in self.edges:
            start_vertex, weight, end_vertex = edge
            print("Edge:", start_vertex.name, "--", weight, "--", end_vertex.name)

# Create a graph
graph = Graph()

# Add vertices
vertex_a = graph.add_vertex("A")
vertex_b = graph.add_vertex("B")
vertex_c = graph.add_vertex("C")
vertex_d = graph.add_vertex("D")
vertex_e = graph.add_vertex("E")
vertex_f = graph.add_vertex("F")
vertex_g = graph.add_vertex("G")
```

```
# Add edges with weights
graph.add_edge(vertex_a, 2, vertex_b)
graph.add_edge(vertex_a, 3, vertex_d)
graph.add_edge(vertex_a, 3, vertex_c)
graph.add_edge(vertex_b, 4, vertex_c)
graph.add_edge(vertex_b, 3, vertex_e)
graph.add_edge(vertex_d, 7, vertex_f)
graph.add_edge(vertex_d, 5, vertex_c)
graph.add_edge(vertex_c, 6, vertex_f)
graph.add_edge(vertex_c, 1, vertex_e)
graph.add_edge(vertex_f, 8, vertex_e)
graph.add_edge(vertex_f, 9, vertex_g)

# Print the graph
graph.print_graph()

# Find the minimum spanning tree
minimum_spanning_tree = graph.kruskal()

# Print the minimum spanning tree
print("Minimum Spanning Tree:")
for edge in minimum_spanning_tree:
    start_vertex, weight, end_vertex = edge
    print(start_vertex.name, "--", weight, "--", end_vertex.name)
```

Output :

```
PS E:\Semester 4\Data Structure and Algorithm\Lab\Lab 12\New folder> python -u
"e:\Semester 4\Data Structure and Algorithm\Lab\Lab 12\New folder\kruskal_algo.py"
```

Minimum Spanning Tree:
C -- 1 -- E
A -- 2 -- B
A -- 3 -- D
A -- 3 -- C
C -- 6 -- F
F -- 9 – G

```
PS E:\Semester 4\Data Structure and Algorithm\Lab\Lab 12\New folder>
```

Lab Evaluation Rubrics							
Domain	**CLOs/ Rubric**	**Performance Indicator**	**Unsatisfactory 0-2**	**Marginal 3-5**	**Satisfactory 6-8**	**Exemplary 9-10**	**Allocate d Marks**
Psychomotor	**CLO:1 R2**	Implementation with Results **(P)**	Does not try to solve problems. Many mistakes in code and difficult to comprehend for the instructor. There is not result of the problem.	Does not suggests or refine solutions but is willing to try out solutions suggested by others. Few mistakes in code, but done along with comments, and easy to comprehend for the instructor. Few mistake in result.	Refines solutions suggested by others. Complete and error-free code is done. No comments in the code, but easy to comprehend for the instructor. Results are correctly produced.	Actively looks for and suggests solution to problems. Complete and error free code is done, easy to comprehend for the instructor. Results are correctly produced. Student incorporated comments in the code.	
Affective	**CLO:3 R3**	Lab Report **(A)**	Code of the problem is not given. Outputs are not provided. Explanation of the solution is not stated.	Code of the problem is given. Output is not complete. Explanation of the solution is not satisfactory.	Code of the problem is given. Output is completely given. Explanation of the solution is not satisfactory.	Code of the problem is given. Output is completely given. Explanation of the solution is satisfactory.	
	CLO:1 R5	Discipline and Behavior **(A)**	Got and wandered around. More than two incidents of talking non-lab related stuff in lab and/or any talk with other groups, voice level exceeding the appropriate level, use of cell phones and involvement in any non-lab activity.	Got out of seat and wander around for some time. No more than two incidents of talking non-lab related stuff in lab. Voice level exceeding the appropriate level, use of cell phones and involvement in any non-lab related activity.	Stayed in seat and got up for a specific lab related reason, but took more time than required to do the job. No more than one incidents of talking non-lab related stuff in lab. Voice level exceeding the appropriate level, use of cell phones and involvement in any non-lab related activity.	Stayed in seat and got up for a specific lab related reason. Took care of lab related business and sat down right away. Voice level kept appropriate. Not used cell phones or involved in any non- lab related activity.	