Homework 4: Pratik Mistry – DSA Spring 2020 (pdm79)

Solution 1:

- The graph for the dataset given is not acyclic i.e. there will be an instance of one cycle in the graph
- Detecting whether a graph is acyclic or not is based purely on performing DFS on all the vertices
- Also, while performing DFS on each vertex we send parent vertex as well (where Parent is nothing but a vertex from where the current vertex is called)
- Thus to detect a cycle we check two conditions:
    - Vertex in adjacency list is already visited
    - Parent of the vertex is not same as vertex which is visited in adjacency list
- For e.g. let U be the parent vertex that performs DFS one of the adjacent vertex V. Thus if any vertex w that is *adj(V)* is *visited* and is not same as parent U , then cycle is detected and hence graph is acyclic.
- The Time complexity of this algorithm is same as DFS i.e. O (V+E).

Solution 2:

- I have implemented Kruskal's algorithm and Prim's algorithm (Lazy approach) to find the Minimum Spanning Tree and its edges for the given graph
- The weight of Minimum Spanning Tree for both the algorithm is: **10.463510000000001**
- Below is the result of the running time for Kruskal and Prim algorithms to find MST and its edges:

Algorithm	Time Complexity (in seconds)
Kruskal Algorithm	0.18191 seconds
Prim Algorithm (Lazy Approach)	0.03850 seconds

Table 1. Performance Comparison between Kruskal and Prim Algorithm

- As we can see from the Table 1. Prim's algorithm performs better than Kruskal's Algorithm to find the Minimum Spanning Tree.
- The worst case time complexity for Kruskal's algorithm is **ELog(E**) and for Prim's Algorithm (lazy approach) is **ELog(E).**
- ELogE in Kruskal is because of Priority Queue implementation and ELogE in Prim is because of Binary Heap implementation.
- The reason why Kruskal's implementation performs bad compared to Prim's is because in Kruskal's algorithm we perform *Union ()* between two vertices V times, which **adds** VLog(V) time in the performance.
  Note: VLog(V) is ignored because we have defined *upper bound* for Kruskal's algorithm in above point.
- More efficient algorithm than above two algorithms is Prim's Eager approach.

Solution 3:

- For directed edge weighted digraph with no directed cycles, it is easier to find the shortest path than in general digraph using Topological Sort
- Also, topological sort can be used to find the longest path as well
- For the given question and dataset, Figure 1 represents the graph in pictorial format

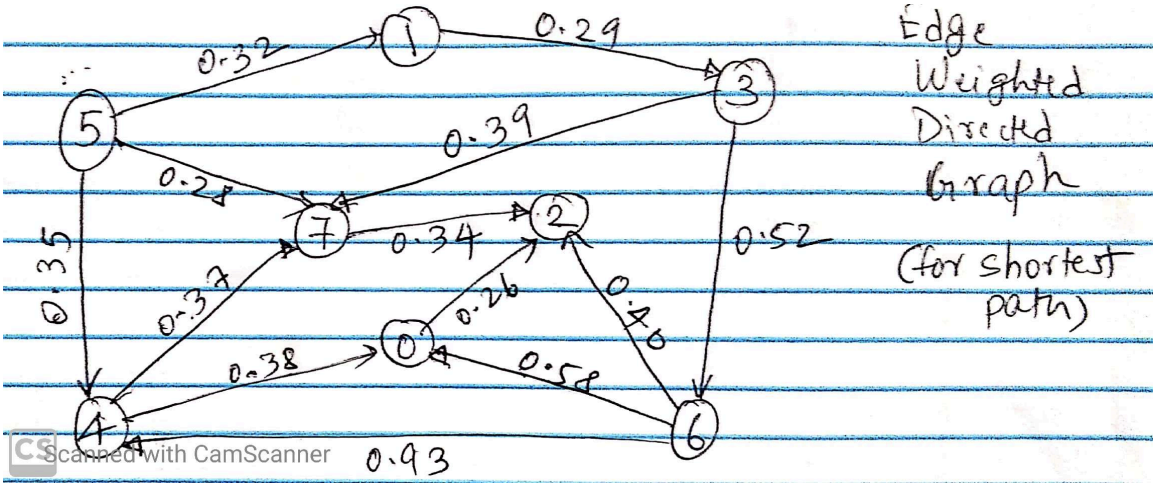

Figure 1: Graph Representation for given dataset

- Now, we will find the topological sort of the given graph. Recall topological sort. Below is the pseudo code:

TopologicalSort (Graph)
Create stack
Initialize array of size V to store whether vertex is visited or not
For each vertex in graph:
If vertex not visited: Perform DFS from that vertex

Return stack

DFS (Vertex: v):
Mark v as visited
For each adjacent vertex:
If adjacent vertex not visited:
Recursively Perform DFS from current vertex
After all the adjacent vertices are visited, push vertex in stack

- The stack returned from topological sort is reverse DFS post order i.e. topological sort of given graph.
- For our case, topological sort is would be:

5 1 3 6 4 7 0 2

- We will find shorted and longest path by relaxing edges adjacent to each of the vertex of topological sort

- Below is the pseudo code to find the shortest paths in DAGs:

SP_DAG (Graph, Source):
Create edgeTo and distTo array of size V
Insert INFINITY as distance in distTo for all vertices
Insert distTo first vertex of topological sort to 0
For each vertex in topological sort order:
For each adjacent edges:
Relax the edge

Return distTo and edgeTo

Relax (Edge: e):
Get u and v from edge
If distTo v is greater than distTo u + edge weight:
Add the distTo v as distTo u + weight
Add the edge for the visited vertex v in edgeTo

Shortest Path:

- For calculating shortest path, we will consider the graph weights as original
- Edge relaxation try to find and update the shortest distance to visiting node based on distance to parent node and edge weight
- We will find the shortest path in DAGs based on algorithm described in above pseudo code with vertices in Topological Sort order

1. Consider 5 and relax all the edges adjacent to 5 i.e. 5->1, 5->7 and 5->4

Vertex	0	1	2	3	4	5	6	7
distTo		0.32			0.35	0		0.28
edgeTo		5->1			5->4	-		5->7

2. Consider 1 and relax adjacent edge to 1 i.e. 1->3

Vertex	0	1	2	3	4	5	6	7
distTo		0.32		0.61	0.35	0		0.28
edgeTo		5->1		1->3	5->4	-		5->7

3. Consider 3 and relax edges adjacent to 3 i.e. 3->6, 3->7
   Distance to vertex 7 won't be updated

Vertex	0	1	2	3	4	5	6	7
distTo		0.32		0.61	0.35	0	1.13	0.28
edgeTo		5->1		1->3	5->4	-	3->6	5->7

4. Consider 6 and relax adjacent edge to 6 i.e. 6->2, 6->0, 6->4.
   Distance to vertex 4 won't change because it shorter distance of traversal

Vertex	0	1	2	3	4	5	6	7
distTo	1.71	0.32	1.53	0.61	0.35	0	1.13	0.28
edgeTo	6->0	5->1	6->2	1->3	5->4	-	3->6	5->7

5. Consider 4 and relax edges adjacent to 4 i.e. 4->7, 4->0
   Distance to vertex 7 won't change because it has shorter distance of traversal
   Distance to vertex 0 will be updated

Vertex	0	1	2	3	4	5	6	7
distTo	0.73	0.32	1.53	0.61	0.35	0	1.13	0.28
edgeTo	4->0	5->1	6->2	1->3	5->4	-	3->6	5->7

6. Consider 7 and relax edges adjacent to 7 i.e. 7->2

Vertex	0	1	2	3	4	5	6	7
distTo	0.73	0.32	0.62	0.61	0.35	0	1.13	0.28
edgeTo	4->0	5->1	7->2	1->3	5->4	-	3->6	5->7

7. Consider 0 and relax edges adjacent to 0 i.e. 0->2.
   Distance to 2 won't be updated because it has shorter distance of traversal. Hence Table remains unchanged
8. Consider 2 and since it don't have adjacent edges. Table remains unchanged

- Thus, since adjacent edges of all the vertices of topological sort is considered. Below is the final table having shortest paths and parent edges for all the vertices of graph

Vertex	0	1	2	3	4	5	6	7
distTo	0.73	0.32	0.62	0.61	0.35	0	1.13	0.28
edgeTo	4->0	5->1	7->2	1->3	5->4	-	3->6	5->7

Longest Path:

- To calculate the Longest Path, we will
  - Negate all the weights of the graph,
  - Perform edges relaxation for each vertex in topological sort order
  - Negate all the result values i.e. distTo array values to get the longest path to each vertex
- Figure 2 below shows the graph with negated weights

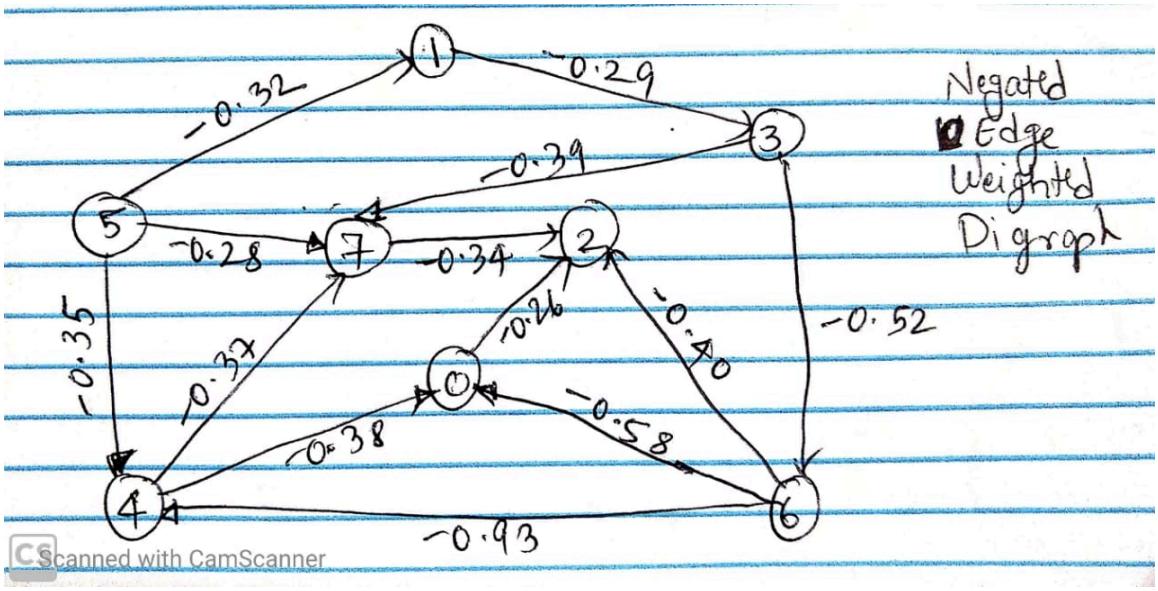

Figure 2: Graph Representation for given dataset with Negated Weights

- We will find the longest path in DAGs based on same algorithm used for shortest path with vertices in Topological Sort order

1. Consider 5 and relax all the edges adjacent to 5 i.e. 5->1, 5->7 and 5->4

Vertex	0	1	2	3	4	5	6	7
distTo		-0.32			-0.35	0		-0.28
edgeTo		5->1			5->4	-		5->7

2. Consider 1 and relax adjacent edge to 1 i.e. 1->3

Vertex	0	1	2	3	4	5	6	7
distTo		-0.32		-0.61	-0.35	0		-0.28
edgeTo		5->1		1->3	5->4	-		5->7

3. Consider 3 and relax edges adjacent to 3 i.e. 3->6, 3->7
   Distance to 7 will be updated

Vertex	0	1	2	3	4	5	6	7
distTo		-0.32		-0.61	-0.35	0	-1.13	-1
edgeTo		5->1		1->3	5->4	-	3->6	3->7

4. Consider 6 and relax adjacent edge to 6 i.e. 6->2, 6->0, 6->4.
   Distance to vertex 4 will be updated

Vertex	0	1	2	3	4	5	6	7
distTo	-1.71	-0.32	-1.53	-0.61	-2.06	0	-1.13	-1
edgeTo	6->0	5->1	6->2	1->3	6->4	-	3->6	3->7

5. Consider 4 and relax edges adjacent to 4 i.e. 4->7, 4->0
   Distance to vertex 0 and 7 will be updated

Vertex	0	1	2	3	4	5	6	7
distTo	-2.44	-0.32	-1.53	-0.61	-2.06	0	-1.13	-2.43
edgeTo	4->0	5->1	6->2	1->3	6->4	-	3->6	4->7

6. Consider 7 and relax edges adjacent to 7 i.e. 7->2
   Distance to 2 will be updated

Vertex	0	1	2	3	4	5	6	7
distTo	-2.44	-0.32	-2.77	-0.61	-2.06	0	-1.13	-2.43
edgeTo	4->0	5->1	7->2	1->3	6->4	-	3->6	4->7

7. Consider 0 and relax edges adjacent to 0 i.e. 0->2.
   Distance to 2 won't be updated because it has shorter distance of traversal. Hence Table remains unchanged
8. Consider 2 and since it don't have adjacent edges. Table remains unchanged

- Thus, since adjacent edges of all the vertices of topological sort is considered. Now negate all the weights of distTo each vertices and below is the final table having longest paths and parent edges for all the vertices of graph

Vertex	0	1	2	3	4	5	6	7
distTo	2.44	0.32	2.77	0.61	2.06	0	1.13	2.43
edgeTo	4->0	5->1	7->2	1->3	6->4	-	3->6	4->7

Solution 4:

- Dijkstra's algorithm to calculate shortest path doesn't work if there is negative weight cycle in graph.
- Bellman Ford algorithm finds the shortest distance path to each vertex from source if weights are negative.
- Also, if there is negative weight cycle then it detects the negative cycle as well
- Below is the pseudo code for Bellman Ford Algorithm:

BellmanFord (Graph, Source):
Create edgeTo and distTo array of size V
Create Queue for storing vertex whose distances change in iteration
Insert INFINITY as distance in distTo for all vertices
Assign distTo source as 0
Maintain counter to count iteration 0 to V-1

```
Relax edges from source vertex
Counter + 1
While queue is not empty and counter < V:
        Pop all vertices from queue
        For each adjacent edges of vertices:
                Relax the edge
        Counter + 1

if hasNegativeCycle():          // If negative cycle exists, display the edges
        negativeCycle()
        return or exit from the program execution

Return distTo and edgeTo    // Return distTo and edgeTo if no negative cycle



// Check if there is negative cycle
hasNegativeCycle():
        For each edge (u,v) in graph:          // Check Optimality Condition
        If distTo[v] is greater than distTo[u] + edge weight
                Display Negative Cycle Exists
                Return True

// If negative cycle present – Display the edges of negative cycl;e
negativeCycle():
        While Queue not empty:
                Display the edgeTo value to that vertex
                Perform trace back using edgeTo to get the negative cycle

// Edge Relaxation
Relax (Edge: e):
Get u and v from edge
If distTo v is greater than distTo u + edge weight:
        Add the distTo v as distTo u + weight
        Add the edge for the visited vertex in edgeTo
        Add v in Queue
```

- Based on Pseudo code, we will find shortest path using Bellman Ford algorithm for Part A and Part B

Part A: Figure 3 Represents image of the graph for given dataset

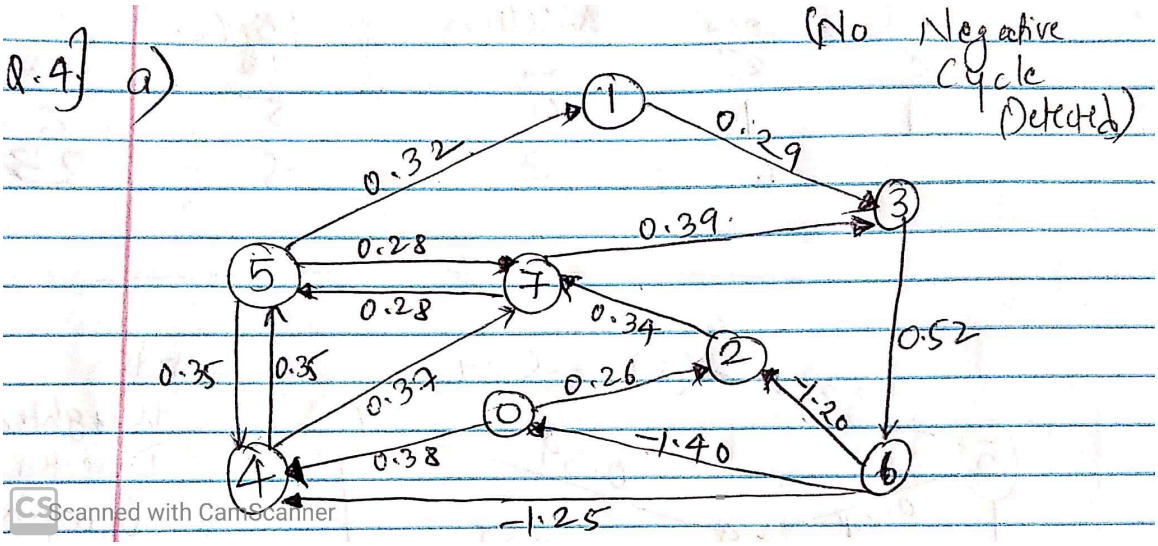

Figure 3: Graph Representation of dataset given for Question 4a

1. Start with source 0 and relax edges from 0 i.e. 0->2, 0->4
   Add 2 and 4 to Queue since their distance changes. Counter = 1

Vertex	0	1	2	3	4	5	6	7
distTo	0		0.26		0.38			
edgeTo	-		0->2		0->4			

2. Consider 2 and 4 vertices and relax edges from 2 and 4 i.e. 2->7, 4->7 and 4->5.
   Add 7 and 5 to queue since their distance changes. Counter = 2

Vertex	0	1	2	3	4	5	6	7
distTo	0		0.26		0.38	0.73		0.60
edgeTo	-		0->2		0->4	4->5		2->7

3. Consider 7 and 5 and relax edges from 7 and 5 i.e. 7->5, 7->3,5->7, 5->4, 5->1,
   Add 3 and 1 to queue since their distance is changes. Counter = 3

Vertex	0	1	2	3	4	5	6	7
distTo	0	1.05	0.26	0.99	0.38	0.73		0.60
edgeTo	-	5->1	0->2	7->3	0->4	4->5		2->7

4. Consider 3 and 1 and relax edges from 3 and 1 i.e. 3->6, 1->3.
   Add 6 to Queue as its distance is changed. Counter = 4

Vertex	0	1	2	3	4	5	6	7
distTo	0	1.05	0.26	0.99	0.38	0.73	1.51	0.60
edgeTo	-	5->1	0->2	7->3	0->4	4->5	3->6	2->7

5. Consider 6 and relax edges from 6 i.e. 6->2, 6->0 and 6->4
   Add 4 to Queue as its distance is changed. Counter = 5

Vertex	0	1	2	3	4	5	6	7
distTo	0	1.05	0.26	0.99	0.26	0.73	1.51	0.60
edgeTo	-	5->1	0->2	7->3	6->4	4->5	3->6	2->7

6. Consider 4 and relax edges from 4 i.e. 4->5 and 4->7.
   Add 5 to Queue as its distance is changed. Counter = 6

Vertex	0	1	2	3	4	5	6	7
distTo	0	1.05	0.26	0.99	0.26	0.61	1.51	0.60
edgeTo	-	5->1	0->2	7->3	6->4	4->5	3->6	2->7

7. Consider 5 and relax edges from 5 i.e. 5->4, 5->7, 5->1.
   Add 1 to Queue as its distance is changed. Counter = 7

Vertex	0	1	2	3	4	5	6	7
distTo	0	0.93	0.26	0.99	0.26	0.61	1.51	0.60
edgeTo	-	5->1	0->2	7->3	6->4	4->5	3->6	2->7

8. Consider 1 and relax edge 1->3
   Distance won't be changed and table remains unchanged.
   Queue = empty and Counter = 8

Vertex	0	1	2	3	4	5	6	7
distTo	0	0.93	0.26	0.99	0.26	0.61	1.51	0.60
edgeTo	-	5->1	0->2	7->3	6->4	4->5	3->6	2->7

- Now, since counter is 8 i.e. counter >= V and also queue is also empty. We will check whether all the edges satisfy optimality condition as defined in Pseudo Code to detect negative cycle
- Since, optimality conditions is satisfied and no distance changes, no negative cycle is detected
- Final returned distTo and edgeTo table is:

Vertex	0	1	2	3	4	5	6	7
distTo	0	0.93	0.26	0.99	0.26	0.61	1.51	0.60
edgeTo	-	5->1	0->2	7->3	6->4	4->5	3->6	2->7

Part B: Figure 4 Represents image of the graph for given dataset

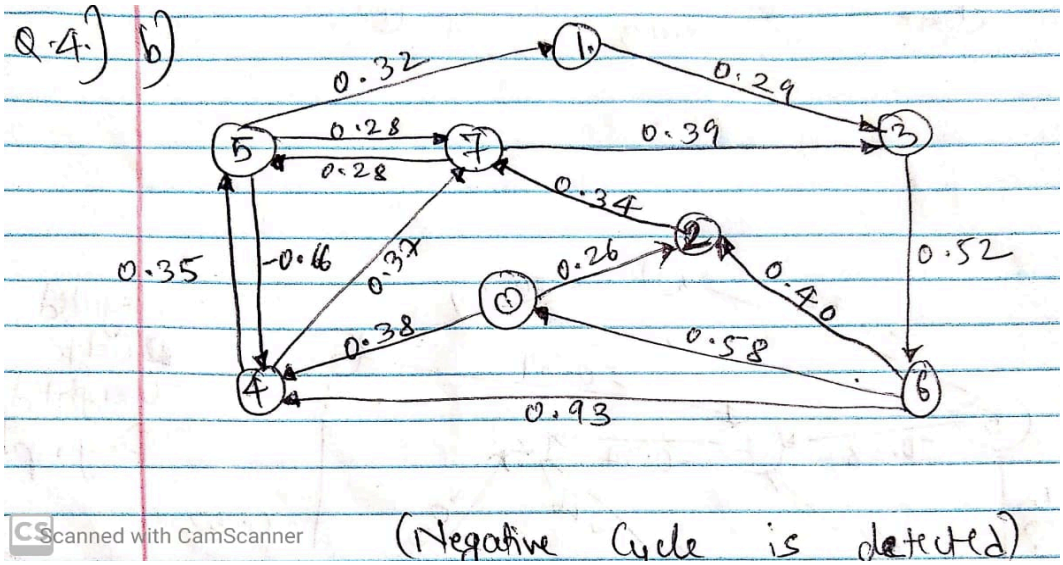

Figure 4: Graph Representation of dataset given for Question 4b

1. Start with source 0 and relax edges from 0 i.e. 0->2, 0->4
   Add 2 and 4 to Queue since their distance changes. Counter = 1

Vertex	0	1	2	3	4	5	6	7
distTo	0		0.26		0.38			
edgeTo	-		0->2		0->4			

2. Consider 2 and 4 vertices and relax edges from 2 and 4 i.e. 2->7, 4->7 and 4->5.
   Add 7 and 5 to queue since their distance changes. Counter = 2

Vertex	0	1	2	3	4	5	6	7
distTo	0		0.26		0.38	0.73		0.60
edgeTo	-		0->2		0->4	4->5		2->7

3. Consider 7 and 5 and relax edges from 7 and 5 i.e. 7->5, 7->3,5->7, 5->4, 5->1,
   Add 3,1 and 4 to queue since their distance is changes. Counter = 3

Vertex	0	1	2	3	4	5	6	7
distTo	0	1.05	0.26	0.99	0.07	0.73		0.60
edgeTo	-	5->1	0->2	7->3	5->4	4->5		2->7

4. Consider 3,1 and 4 and relax edges from 3,1 and 4 i.e. 3->6, 1->3, 4->5, 4->7
   Add 6, 7 and 5 and to Queue as its distance is changed. Counter = 4

Vertex	0	1	2	3	4	5	6	7
distTo	0	1.05	0.26	0.99	0.07	0.42	1.51	0.44
edgeTo	-	5->1	0->2	7->3	5->4	4->5	3->6	4->7

5. Consider 6, 7 and 5 and relax edges from 6, 7 and 5 i.e. 6->2, 6->0 and 6->4, 7->3 and 7->5, 5->4, 5->7 and 5->1
   Add 3,1 and 4 to Queue as its distance is changed. Counter = 5

Vertex	0	1	2	3	4	5	6	7
distTo	0	0.74	0.26	0.83	-0.24	0.42	1.51	0.44
edgeTo	-	5->1	0->2	7->3	5->4	4->5	3->6	4->7

6. Consider 3,1 and 4 and relax edges from 3,1 and 4 i.e. 3->6,1->3, 4->5 and 4->7.
   Add 6, 7 and 5 to Queue as its distance is changed. Counter = 6

Vertex	0	1	2	3	4	5	6	7
distTo	0	0.74	0.26	0.83	-0.24	0.11	1.35	0.13
edgeTo	-	5->1	0->2	7->3	5->4	4->5	3->6	4->7

7. Consider 6, 7 and 5 and relax edges from 6, 7 and 5 i.e. 6->2, 6->0 and 6->4, 7->3 and 7->5, 5->4, 5->7 and 5->1
   Add 3,1 and 4 to Queue as its distance is changed. Counter = 7

Vertex	0	1	2	3	4	5	6	7
distTo	0	0.43	0.26	0.52	-0.55	0.11	1.35	0.13
edgeTo	-	5->1	0->2	7->3	5->4	4->5	3->6	4->7

8. Consider 3,1 and 4 and relax edges from 3,1 and 4 i.e. 3->6,1->3, 4->5 and 4->7.
   Add 6, 7 and 5 to Queue as its distance is changed. Counter = 8

Vertex	0	1	2	3	4	5	6	7
distTo	0	0.43	0.26	0.52	-0.55	-0.20	1.04	-0.18
edgeTo	-	5->1	0->2	7->3	5->4	4->5	3->6	4->7

- Now, since counter is 8 i.e. counter >= V. We will check whether all the edges satisfy optimality condition as defined in Pseudo Code to detect negative cycle
- Since, optimality conditions are not satisfied and distances change for vertices 3, 1 and 4, negative cycle is detected.
- To find edges of negative weight cycle, we use the queue that has vertices whose distTo changed in last iteration i.e. vertices 6,7 and 5.
- Thus for vertices 6,7 and 5 print the edgeTo values giving us the negative cycle edges which is: 3->6 and 4->7, 4->5. Then we perform trace back using edgeTo to find the negative cycle and in our case negative cycle exists as 5->4->7->5 as seen in Figure 4
- Thus, this graph has *negative cycle* in it and shortest distance to vertices cannot be found.

Solution 5:

- For performing DFS for the given dataset, recursive solution does not work as it gives Maximum Recursive Limit error i.e. machine's stack gets filled up entirely while program is still running thus resulting in Segmentation Fault issues
- To counter this problem I used *LIST* in Python as a Stack where operations like *list.append()* adds value in the top of stack and *list.pop()* removes/pops out least recently added value i.e. top of the stack value.
- Thus, it works as when vertex is visited it is added on stack. Then remove top vertex and add all its adjacent vertices (not visited) into the stack. Again, we remove the top vertex and further perform DFS for adjacent vertices. The idea works same as that of DFS done recursively where machine uses built-in stack memory store the nodes visited.
- For performing BFS, implementation idea is same as standard BFS where vertices visited are added to queue. And then the vertex is *dequeue* from queue and further BFS on performed for adjacent vertices (not visited)
- For BFS, I have used *LIST* in Python again as Queue to store vertex visited. To extract vertex from queue *list.pop(0)* function is used which will remove the least recently i.e. first added element from queue. To add element in Queue I have used *list.append().*
- As an output, you can uncomment the two lines of code in main function that will print out the order of edges traversed in DFS and BFS.
- I have printed output as number of nodes Traversed in DFS and BFS that is **264346**.

Solution 6:

- The dataset to be used is from Question 4a and Question 4b.
- Please refer to Figure 3 and Figure 4 for the graph representation for the data given in Question 4a and 4b.
- I have used standard implementation of Dijkstra's algorithm.
- We know that since there is no negative cycle in data for Question 4a, shortest path using Dijkstra's algorithm can be calculated.
- But since there is negative weight cycle present in data for Question 4b, shortest path using Dijkstra's algorithm cannot be calculated as the program goes in ***infinite*** loop execution.
- Below is the output of the shortest path using Dijkstra's algorithm with source vertex as 0 for Question 4a:

```
Executing Dijkstra Algorithm for graph with data of Question 4a:
Vertex   Dist To         Edge To
0        0.0             -
1        0.93            5->1
2        0.26            0->2
3        0.99            7->3
4        0.26            6->4
5        0.61            4->5
6        1.51            3->6
7        0.6             2->7
```

- Since, there is negative weight cycle in the graph of Question 4b dataset, I have used a naïve method to detect the same. Thus, below is the output when program is executed for Question 4b dataset.

```
Executing Dijkstra Algorithm for graph with data of Question 4b:
Negative cycle detected in the graph. Cannot Perform Dijkstra Algorithm on graph with negative weight cycles...!!!
```

- The naïve method to detect negative weighted cycle is if distance to any vertex while edge relaxation goes below -1000 which is practically impossible for this dataset, we print the message about the detection and exit from the code execution.