DSA Homework 4

Name: Shounak Rangwala
Netid: snr85

Q1)
The cycle detection works well for the directed graphs and determining if the topological order of a given digraph can be obtained.
However, since the undirected graph is also a special directed graph, the only condition that we have to change here (since this will be applicable to each of the edge) is:

When running DFS on the graph; if the any vertex in the given node’s adjacency list has been visited and is not the node that came before this node in the DFS traversal, then there is a cycle in the undirected graph.

We are assuming that the 2-way direction of each edge will not be counted as a cycle in of itself.

Q2)
Kruskal MST: this implementation was done using the union find data structure and the quickFind method for finding if adding the next edge was creating a cycle.
The complexity of this implementation is O(E log V) average case because we are using the union find array and iterating through it E (edge) number of times.

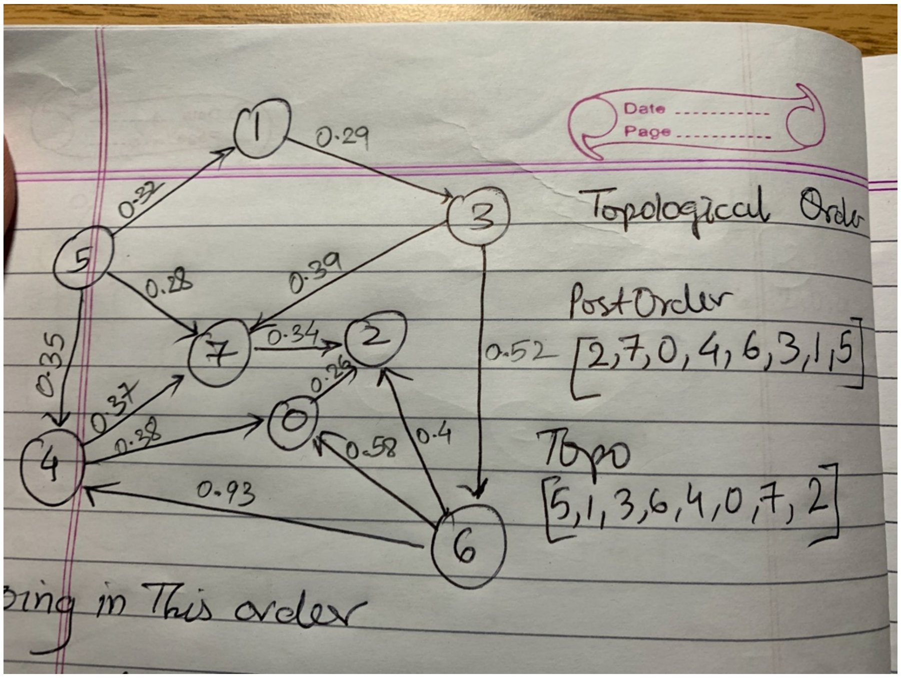
Prims MST: this implementation was done using a normal array with the nodes associated to the tree and the nodes in the queue (so to speak) which will we tested for having the minimum weight in them and then that node will be added to the former array. This is the lazy version of the implementation and the priority queue was made using simple lists in python.
The complexity for this implementation is O(E LogE) where we maintain an additional array of E length and decide which is the minimum distance to be added from it. This operation takes E time.

As you can see, the E is more than V in the dataset and so Kruskal should theoretically take lesser time than Prims, which in the implementation , it indeed does.

Running the implementation 5 times for the dataset :

	Kruskal MST	Prims MST
1	0.01413726806640625	0.014069795608520508
2	0.011714935302734375	0.01599907875061035
3	0.012466096878051758	0.011793375015258789
4	0.011471986770629883	0.01747298240661621
5	0.011118888854980469	0.013916730880737305
AVG	**0.012181835174561**	**0.014650392532349**

3)



Topological Orde

PostOrder [2,7,0,4,6,3,1,5]

Topo [5,1,3,6,4,0,7,2]

oing in This order

Algorithm for shortest path:

- Go in the topological order -> consider vertices in this order
- For each vertex calculate the edges and the distances to neighbors. Add this distance to the distTo array after relaxing edge and update edgeTo entry if there is an edge relaxation.

Trace:

Starting from 5, so

distTo = [inf, inf, inf, inf, inf, 0, inf, inf]

edgeTo = {0:, 1:, 2:, 3:, 4:, 5:0, 6:, 7:}

5: add(1,4,7)

distTo = [inf, 0.32, inf, inf, 0.35, 0, inf, 0.28]

edgeTo = {0:, 1:(5,1), 2:, 3:, 4:(5,4), 5:0, 6:, 7:(5,7)}

1: add(3)

distTo = [inf, 0.32, inf, 0.61, 0.35, 0, inf, 0.28]

edgeTo = {0:, 1:(5,1), 2:, 3:(1,3), 4:(5,4), 5:0, 6:, 7:(5,7)}

3: add(6,7)

distTo = [inf, 0.32, inf, 0.61, 0.35, 0, 1.13, 0.28] # *distTo[7] = 1 > existing , no relaxation*

edgeTo = {0:, 1:(5,1), 2:, 3:(1,3), 4:(5,4), 5:0, 6:(3,6), 7:(5,7)}

6: add(0,2,4)

distTo = [1.61, 0.32, 1.53, 0.61, 0.35, 0, 1.13, 0.28] # *distTo[4]=2.06 > existing ,no relaxation*

edgeTo = {0:(6,0), 1:(5,1), 2:(6,2), 3:(1,3), 4:(5,4), 5:0, 6:(3,6), 7:(5,7)}

4: add(0,7)
distTo = [0.73, 0.32, 1.53, 0.61, 0.35, 0, 1.13, 0.28] # *distTo[0]=0.73 < existing so relaxation*
edgeTo = {0:(4,0), 1:(5,1), 2:(6,2), 3:(1,3), 4:(5,4), 5:0, 6:(3,6), 7:(5,7)}

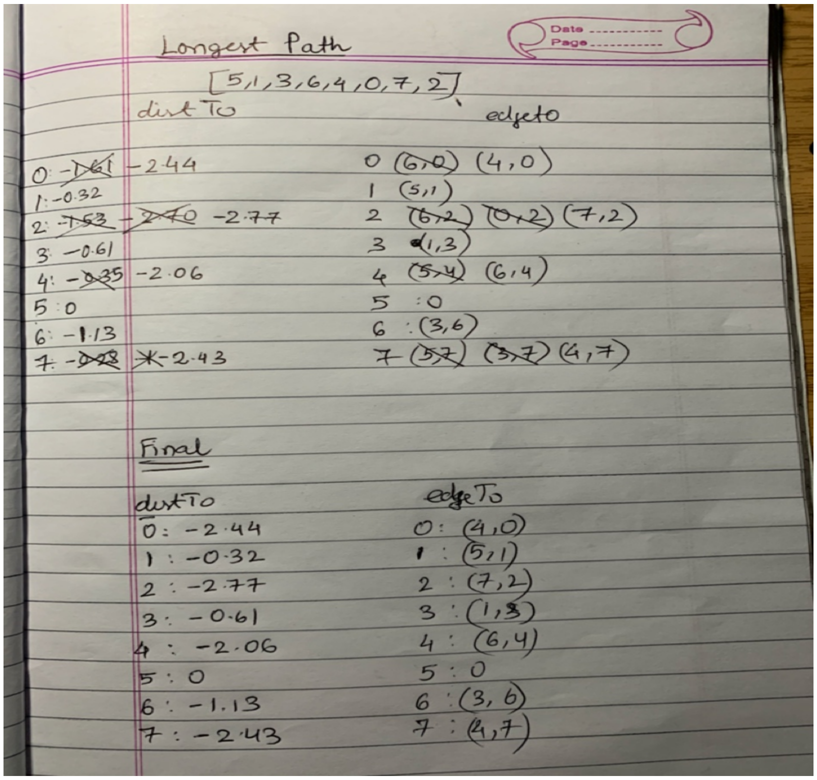
0: add(2)
distTo = [0.73, 0.32, 0.99, 0.61, 0.35, 0, 1.13, 0.28] # *distTo[2]=0.99 < existing so relaxation*
edgeTo = {0:(4,0), 1:(5,1), 2:(0,2), 3:(1,3), 4:(5,4), 5:0, 6:(3,6), 7:(5,7)}

7: add(2)
distTo = [0.73, 0.32, 0.62, 0.61, 0.35, 0, 1.13, 0.28] # *distTo[2]=0.62 < existing so relaxation*
edgeTo = {0:(4,0), 1:(5,1), 2:(7,2), 3:(1,3), 4:(5,4), 5:0, 6:(3,6), 7:(5,7)}

Final :
distTo = [0.73, 0.32, 0.62, 0.61, 0.35, 0, 1.13, 0.28]
edgeTo = {0:(4,0), 1:(5,1), 2:(7,2), 3:(1,3), 4:(5,4), 5:0, 6:(3,6), 7:(5,7)}

Algorithm for longest path:

- Negate all weights
- Apply algorithm for shortest path
- Reverse the weight in the final answer



Date ....
Page ....

Longest Path

[5,1,3,6,4,0,7,2]

dist To	edgeto
0: −~~1.61~~ −2.44	0 ~~(6,0)~~ (4,0)
1: −0.32	1 (5,1)
2: −~~1.53~~ −~~2.70~~ −2.77	2 ~~(6,2)~~ ~~(0,2)~~ (7,2)
3: −0.61	3 (1,3)
4: −~~0.35~~ −2.06	4 ~~(5,4)~~ (6,4)
5 : 0	5 : 0
6: −1.13	6 : (3,6)
7: −~~0.28~~ *−2.43	7 ~~(5,7)~~ ~~(3,7)~~ (4,7)

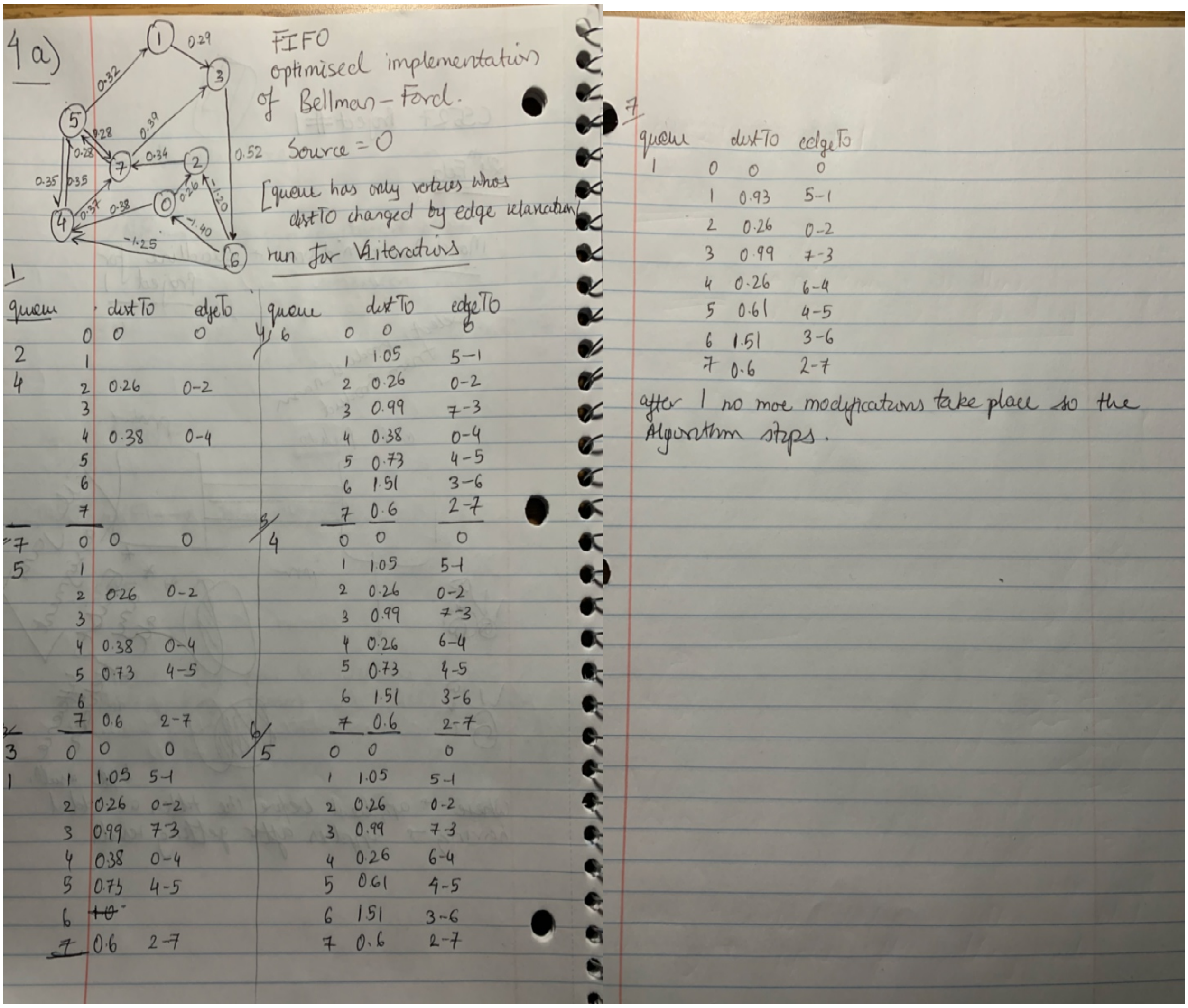
Final

distTo	edgeTo
0 : −2.44	0 : (4,0)
1 : −0.32	1 : (5,1)
2 : −2.77	2 : (7,2)
3 : −0.61	3 : (1,3)
4 : −2.06	4 : (6,4)
5 : 0	5 : 0
6 : −1.13	6 : (3, 6)
7 : −2.43	7 : (4,7)

Negate the final distTo array values to get the longest path distances.

4)

Part a:


1
3
5
7
2
0
4
6
0.32
0.29
0.28
0.28
0.39
0.34
0.52
0.35
0.35
0.37
0.38
0.26
1.20
-1.40
-1.25

4 a)

FIFO optimised implementation of Bellman-Ford.

Source = 0

[queue has only vertices whos distTo changed by edge relaxation]

run for V iterations

1

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

7	0	0	0
5	1		
	2	0.26	0-2
	3		
	4	0.38	0-4
	5	0.73	4-5
	6		
	7	0.6	2-7

3	0	0	0
1	1	1.05	5-1
	2	0.26	0-2
	3	0.99	7-3
	4	0.38	0-4
	5	0.73	4-5
	6	~~1.0~~	
	7	0.6	2-7

queue		distTo	edgeTo
~~4~~, 6	0	0	0
	1	1.05	5-1
	2	0.26	0-2
	3	0.99	7-3
	4	0.38	0-4
	5	0.73	4-5
	6	1.51	3-6
	7	0.6	2-7

~~3~~ 4	0	0	0
	1	1.05	5-1
	2	0.26	0-2
	3	0.99	7-3
	4	0.26	6-4
	5	0.73	4-5
	6	1.51	3-6
	7	0.6	2-7

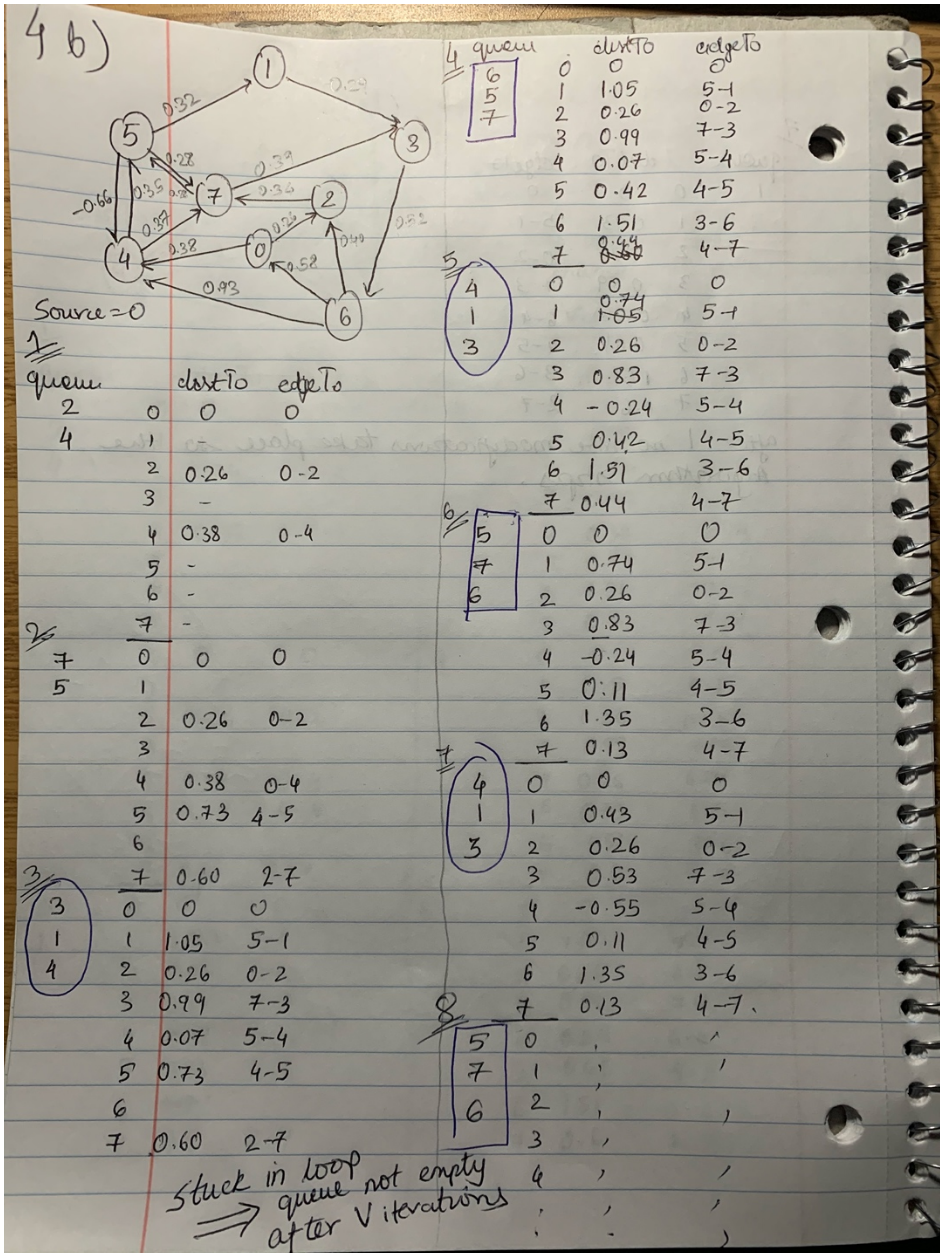
~~6~~ 5	0	0	0
	1	1.05	5-1
	2	0.26	0-2
	3	0.99	7-3
	4	0.26	6-4
	5	0.61	4-5
	6	1.51	3-6
	7	0.6	2-7

7

queue		distTo	edgeTo
1	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

after 1 no more modifications take place so the Algorithm stops.

Part B:


1
2
3
4
5
6
7
0
0.32
0.29
0.28
0.28
0.39
0.34
-0.66
0.35
0.37
0.26
0.38
0.40
0.52
0.58
0.93

4 b)

Source = 0

1

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

2

7	0	0	0
5	1		
	2	0.26	0-2
	3		
	4	0.38	0-4
	5	0.73	4-5
	6		
	7	0.60	2-7

3

3	0	0	0
1	1	1.05	5-1
4	2	0.26	0-2
	3	0.99	7-3
	4	0.07	5-4
	5	0.73	4-5
	6		
	7	0.60	2-7

Stuck in loop
⇒ queue not empty after V iterations

4

queue		distTo	edgeTo
6	0	0	0
5	1	1.05	5-1
7	2	0.26	0-2
	3	0.99	7-3
	4	0.07	5-4
	5	0.42	4-5
	6	1.51	3-6
	7	~~0.60~~ 0.44	4-7

5

4	0	0	0
1	1	~~1.05~~ 0.74	5-1
3	2	0.26	0-2
	3	0.83	7-3
	4	-0.24	5-4
	5	0.42	4-5
	6	1.51	3-6
	7	0.44	4-7

6

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

7

4	0	0	0
1	1	0.43	5-1
3	2	0.26	0-2
	3	0.53	7-3
	4	-0.55	5-4
	5	0.11	4-5
	6	1.35	3-6
	7	0.13	4-7.

8

5	0	,	,
7	1	,	,
6	2	,	,
	3	,	
	4	,	,
	,	,	,
	.	-	)

5)

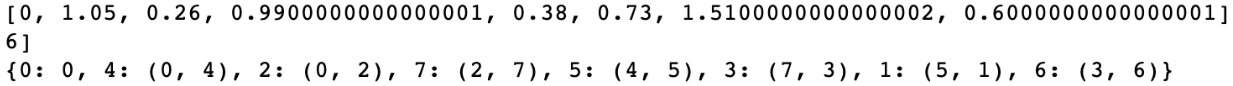
Ideally to run the DFS on an undirected graph, we would use a recursive function but since the number of nodes in the dataset is too big, I tried to increase the recursion limit, but the underlying stack limited was exceeded. I had to thus iteratively, try to run DFS on the dataset.

BFS was executed normally using a queue. The DFS implementation used a stack which would contain the nodes in the adjacency list of a particular node at any point of time. We rum the loop as long as the length of the stack does not become 0.

The output of the DFS and BFS function will give the array with the order in which all the vertices were visited.

6) In the 4b implementation, I ran my code for Djikstra and as expected, the code was caught in the negative cycle loop and executed itself infinitely.

As Djikstra is used to find the shortest path, assuming only non-negative weights. In 4A, we see that the normal execution of the algorithm till vertex 6 from vertex 0. After this, Djikstra will stop because it has calculated the positive path distances in the graph. The negative edges from vertex 6 which will cause the algorithm to execute more times (till the effect of the negative weights are accounted for) are never read and so Djikstra stops here itself.
The distTo and edgeTo arrays for 4a are shown here.



```
[0, 1.05, 0.26, 0.9900000000000001, 0.38, 0.73, 1.5100000000000002, 0.6000000000000001]
6]
{0: 0, 4: (0, 4), 2: (0, 2), 7: (2, 7), 5: (4, 5), 3: (7, 3), 1: (5, 1), 6: (3, 6)}
```

The implementation of Djikstra for the 4A dataset is done in Q6-4B.py. I have added a debug print statement to show that the loop runs infinitely. The implementation for 4A is in the Q6-4A.py file.