DSA Homework 4 Report

Submitted By: Diksha Prakash (dp978)

Question 1:

The graph is cyclic, which implies that it contains at least one circle in it. The idea implemented here for the testing of whether a graph is acyclic or not is quite simple. First, a 'Depth First search' (DFS) is performed on all the vertices of the graph. If the next vertex to be visited has already been visited i.e. 'marked', while this marked vertex is also not the parent, we can conclude that there exists a cycle in the graph.

While the DFS traversal happens, we maintain two data structures so that we can maintain a track of both the 'marked' vertices as well as keep a track of the occurrence of a cycle. If a vertex 'u' from G.adj(v) has been visited, and it is not equal to 'w' (i.e. where we started from), the graph is cyclic.

Question 2:

Upon running the program five times, a table like this is generated:

Running Time (ns) per Iteration	Kruskal's	Lazy Prim's	Eager Prim's
Iteration #1	18549082	7307623	4284228
Iteration #2	14262912	4726915	2006753
Iteration #3	13692611	4292006	2356476
Iteration #4	14142170	4470252	2139114
Iteration #5	13580289	4682520	2252010

It is worth noting that that the weight of the MST generated very time is **10.46351**

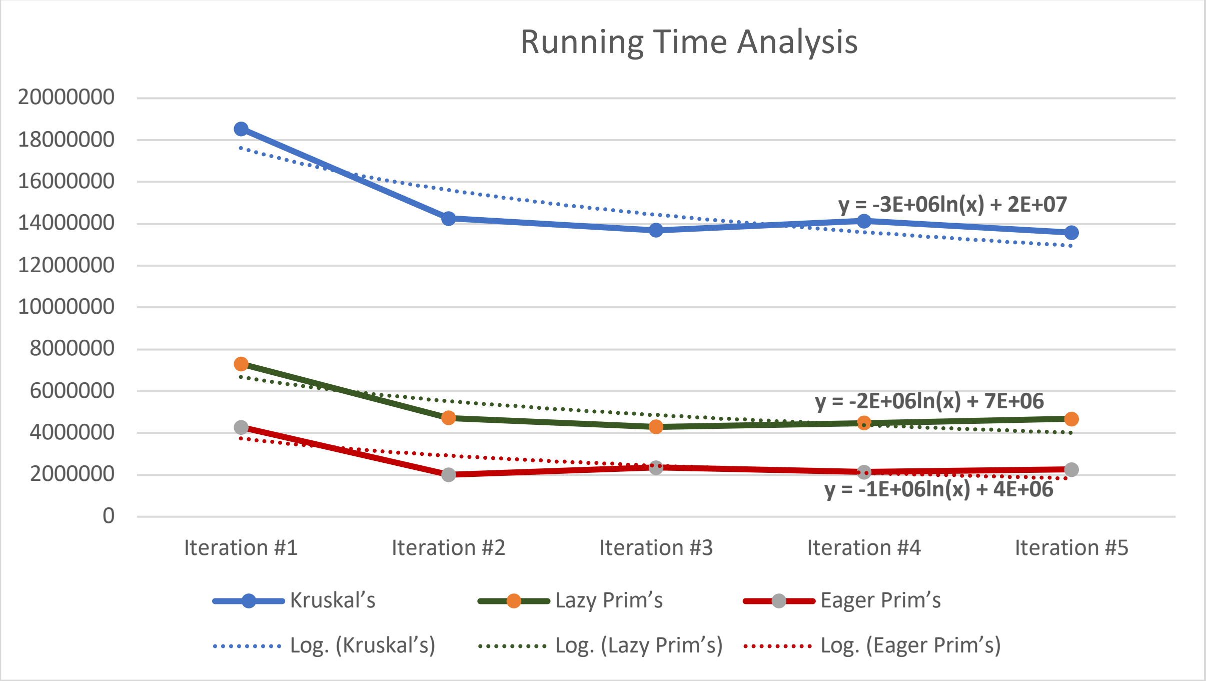
The time complexity of Kruskal is **$E + E_0 logE$**, where E_0 represents the number of edges whose weight is less than the weight of the MST edge which has the highest weight. It can be ibserved that Kruskal's algorithm is slower than the Prim's algorithm as for each edge, apart from the priority-queue operations that both algorithms do for each edge process, it calls '*connected()*' method for every edge.

As the graph involved in this problem is a dense one (as V < E), the eager Prim's performs better than other two algorithms.

The runtime complexity of each of the three algorithms can be estimated as under:

1. Kruskal's Algorithm: ElogE
2. Prim's Lazy Implementation: ElogE
3. Prim's Eager Implementation: ElogV

The obtained graph for Table 1 is as under:



It can be clearly seen from the graph that the ‘Eager Prim’s Implementation’ takes the least amount of time, followed by ‘Lazy Prim’s Implementation’ and finally the ‘Kruskal’s Algorithm’.

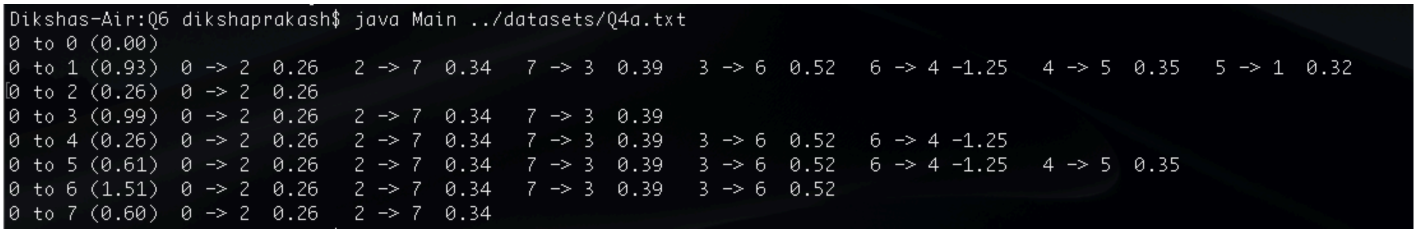
Question 5:

BFS will run properly on the dataset. However, if the simple recursive version of DFS is used, stack overflow will occur. To solve this, we must maintain a stack by ourselves.
First, we push the first vertex into the stack and mark it as visited.
Second, when the stack is not empty, the third step and the fourth step are repeated.
Third, the first vertex in the stack is popped.
Finally, we visit all the adjacent vertices of the vertex and then push the unvisited vertices into the stack, later on marking them as visited.

Both DFS and BFS will print out the number of vertices they visit. The value should be 264346.

Question 6:

The output of the Q4(a) is shown as below. It is the same as the result in Q4(a).



```
Dikshas-Air:Q6 dikshaprakash$ java Main ../datasets/Q4a.txt
0 to 0 (0.00)
0 to 1 (0.93)  0 -> 2  0.26   2 -> 7  0.34   7 -> 3  0.39   3 -> 6  0.52   6 -> 4 -1.25   4 -> 5  0.35   5 -> 1  0.32
0 to 2 (0.26)  0 -> 2  0.26
0 to 3 (0.99)  0 -> 2  0.26   2 -> 7  0.34   7 -> 3  0.39
0 to 4 (0.26)  0 -> 2  0.26   2 -> 7  0.34   7 -> 3  0.39   3 -> 6  0.52   6 -> 4 -1.25
0 to 5 (0.61)  0 -> 2  0.26   2 -> 7  0.34   7 -> 3  0.39   3 -> 6  0.52   6 -> 4 -1.25   4 -> 5  0.35
0 to 6 (1.51)  0 -> 2  0.26   2 -> 7  0.34   7 -> 3  0.39   3 -> 6  0.52
0 to 7 (0.60)  0 -> 2  0.26   2 -> 7  0.34
```

The program will get stuck into an endless loop when the data comes from Q4(b).

The digraph in Q4(a) has a negative edge but no negative cycle. Every edge vertex1->vertex2 will be relaxed only once in the case of Dijkstra.
However, once a vertex has been relaxed, we assume that it has the minimum distTo[], and it won't be updated later.

Considering the situation where Dijkstra selects vertex 3 immediately after 0. But the shortest path from 0 to 3 is 0→1→2→3. So, when the negative-weighted edges exist, Dijkstra cannot guarantee the right result, but it is possible to find the correct result if we visit the path that has negative-weight edge first. The digraph in Q4(b) has a negative cycle, so the Dijkstra algorithm will keep relaxing the edges in the cycle and never get out.

Question 3 : given, DAG ∴ Topological Sort is our best choice.

The obtained topological order : 5 1 3 6 4 0 7 2
of the diagraph

To build the shortest path tree from the vertex 5, the steps are as under :

i) Add 5 and all edges leaving it to the tree

ii) Add 1 and 1→3 to the tree

iii) Add 3 and 3→6 to the tree

iv) Add 6 and 6→2 and 6→0 to the tree

v) Add 4 and 4→0 to the tree

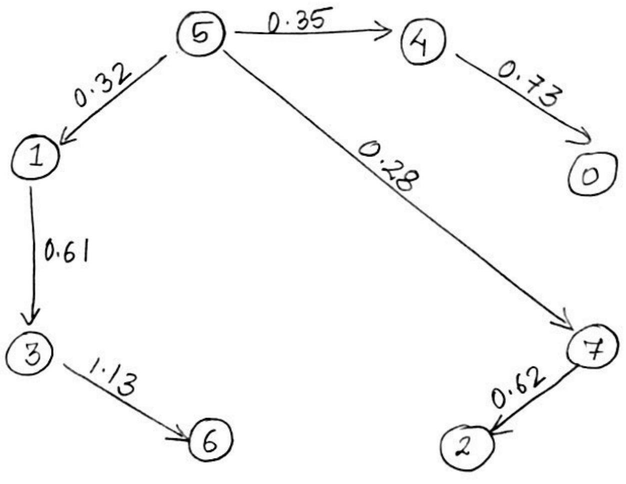
vi) Add 0 to the tree

vii) Add 7 and 7→2 to the tree

viii) Add 2 to the tree.

Thus,

Vertex	distTo [ ]	edgeTo [ ]
0	0.73	4→0
1	0.32	5→1
2	0.62	7→2
3	0.61	1→3
4	0.35	5→4
5	Θ	-
6	1.13	3→6
7	0.28	5→7



To build the longest path tree from vertex 5, first we negate all the weights of the edges & then do the following steps :

i> Add 5 and all the edges leaving it to the tree

ii> Add 1 and 1→3 to the tree

iii> Add 3 and 3→6 , 3→7 to the tree

iv> Add 6 and 6→2 , 6→0 and 6→4 to the tree

v> Add 4 and 4→7 , 4→0 to the tree

vi> Add 0 to the tree

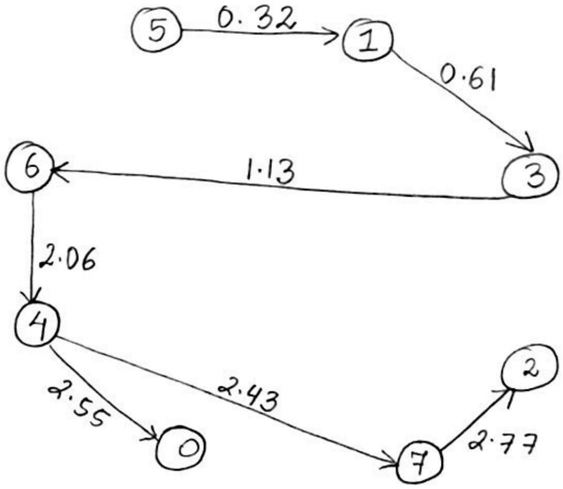
vii> Add 7 and 7→2 to the tree

viii> Add 2 to the tree.

Finally, we negate the total weight to get the answer.

Thus,

Vertex	distTo[ ]	edgeTo[ ]
0	2.55	4→0
1	0.32	5→1
2	2.77	7→2
3	0.61	1→3
4	2.06	6→4
5	0	–
6	1.13	3→6
7	2.43	4→7



Question 4 Let the source be '0'

PART A

Iteration #	Queue	Vertex	distTo [ ]	edgeTo [ ]
I	2	0		
	4	1		
		2	0.26	0→2
		3		
		4	0.38	0→4
		5		
		6		
		7		
II	7	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
III	3	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		
		7	0.60	2→7
IV		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	0.60	2→7

Iteration #	Queue	Vertex	DistTo [ ]	Edge To [ ]
⑤		0		
	4	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	0.60	2→7
⑥	5	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	0.60	2→7
⑦	1	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	0.60	2→7

Question 4 > PART B >

... (same till 2nd iteration)

Iteration #	Queue	Vertex	DistTo [ ]	Edge To [ ]
③	7	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

Iteration#	Queue	Vertex	DistTo[ ]	EdgeTo[ ]
4	3	0		
	1	1	1.05	5 → 1
	4	2	0.26	0 → 2
		3	2.99	7 → 3
		4	0.07	5 → 4
		5	0.73	4 → 5
		6		
		7	0.60	2 → 7
5	6	0		
	7	1	1.05	5 → 1
	5	2	0.26	0 → 2
		3	0.99	7 → 3
		4	0.07	0 → 4
		5	0.73	4 → 5
		6	1.51	3 → 6
		7	0.44	2 → 7
6	3	0		
	1	1	0.74	5 → 1
	4	2	0.26	0 → 2
		3	0.83	7 → 3
		4	-0.59	5 → 4
		5	0.73	4 → 5
		6	1.51	3 → 6
		7	0.60	2 → 7

existence of a negative cycle.