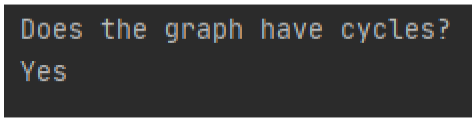
DATA STRUCTURES AND ALGORITHMS

HOMEWORK – 4

1. **Write a program that answers the following for an undirected graph: Is a graph acyclic? Run your program on graph (linked after Q2)**

   In order to detect a cycle in the given undirected graph, I used the depth first search (DFS) algorithm, which has a time complexity of O(V+E). According to this algorithm, for any unvisited vertex v, the adjacent vertices are checked recursively for their adjacent vertices. Once the adjacent vertices are visited, the vertices are pushed into a stack.

   The idea behind detecting cycles using this algorithm is that for every visited vertex ‘v’, if adjacent vertex ‘u’ is visited which is not the parent of ‘v’, then there is a cycle in the graph. This algorithm has a runtime of O(V+E), where V=number of vertices and E=number of edges. Using this idea, a cycle was detected in the graph, as shown below:



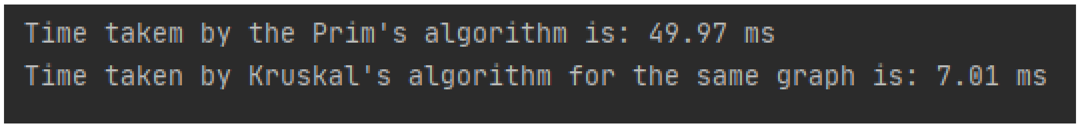
```
Does the graph have cycles?
Yes
```

2. **Implement and execute Prim's and Kruskal's algorithms on the graph linked below (the third field is the weight of an edge). Which performs better? Explain your answer.**

   Prim’s algorithm is a Greedy algorithm which is used to obtain the minimum spanning tree of the graph. It maintains 2 data structures – one which has the vertices already included in the MST, and the other has vertices that are not included yet. In each iteration, all edges that connect the 2 sets are considered and the minimum weight edge is selected. Once this edge is selected, the other endpoint of the edge is moved to the set containing the vertices of the MST.

   Kruskal’s algorithm is also a Greedy algorithm used to obtain the MST of the graph. The basic idea behind this algorithm is to sort all the edges in ascending order and select the smallest edge, provided it doesn’t form a cycle. If it does form a cycle, the edge is discarded. This step is repeated until the number of vertices in the MST is (V-1), V being the number of vertices in the graph.

The following runtimes were obtained by running both of these algorithms:



```
Time takem by the Prim's algorithm is: 49.97 ms
Time taken by Kruskal's algorithm for the same graph is: 7.01 ms
```

It can be seen that Kruskal's algorithm takes lesser time than Prim's algorithm to obtain the MST. This is supported through the big O notation – Kruskal's algorithm takes O(logV) and Prim's algorithm takes a runtime of O(V^2).

3. **For the edge-weighted directed acyclic graph given below, compute (i.e., manually trace) both the longest path and the shortest path.**

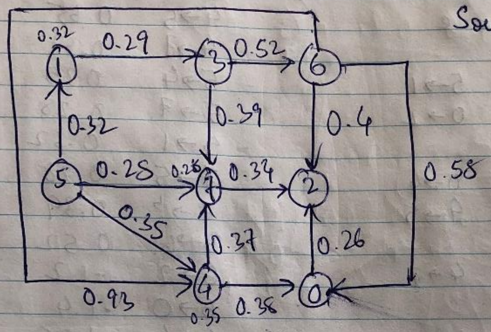
8
13
5 4 0.35
4 7 0.37
5 7 0.28
5 1 0.32
4 0 0.38
0 2 0.26
3 7 0.39
1 3 0.29
7 2 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93

The shortest path for the given graph was computed using Dijkstra's algorithm. Initially, the distances from the source (in this case, vertex 5), was set to infinity. The distance to 5 was set to 0, and the distances to the adjacent vertices were checked and updated accordingly, if the distance obtained was less than the initial value. Using this, the shortest path for each vertex from the source was obtained.

For the case of the longest path, I set the values of the edges to be negative and checked for the shortest path again, since the graph is directed acyclic (DAG). Once the shortest path was obtained, I took the absolute value of the distances and thus, obtained the longest path from the source to each vertex.

3. 8 vertices, 13 edges

Source s = 5



Shortest Path → initially, set all distances to ∞.

v	0	1	2	3	4	5	6	7
d	0.73	0.32	0.62	0.61	0.35	0	1.13	0.28

Longest Path

v	0	1	2	3	4	5	6	7
d	[illegible]	0.32	1.53	0.61	2.06	0	1.13	[illegible]

⇓

v	0	1	2	3	4	5	6	7
d	2.44	0.32	2.70	0.61	2.06	0	1.13	2.43

4. a) For the digraph with negative weights, compute (i.e. manually trace) the progress of the Bellman-Ford Algorithm.

8
15
4 5 0.35
5 4 0.35
4 7 0.37
5 7 0.28
7 5 0.28
5 1 0.32

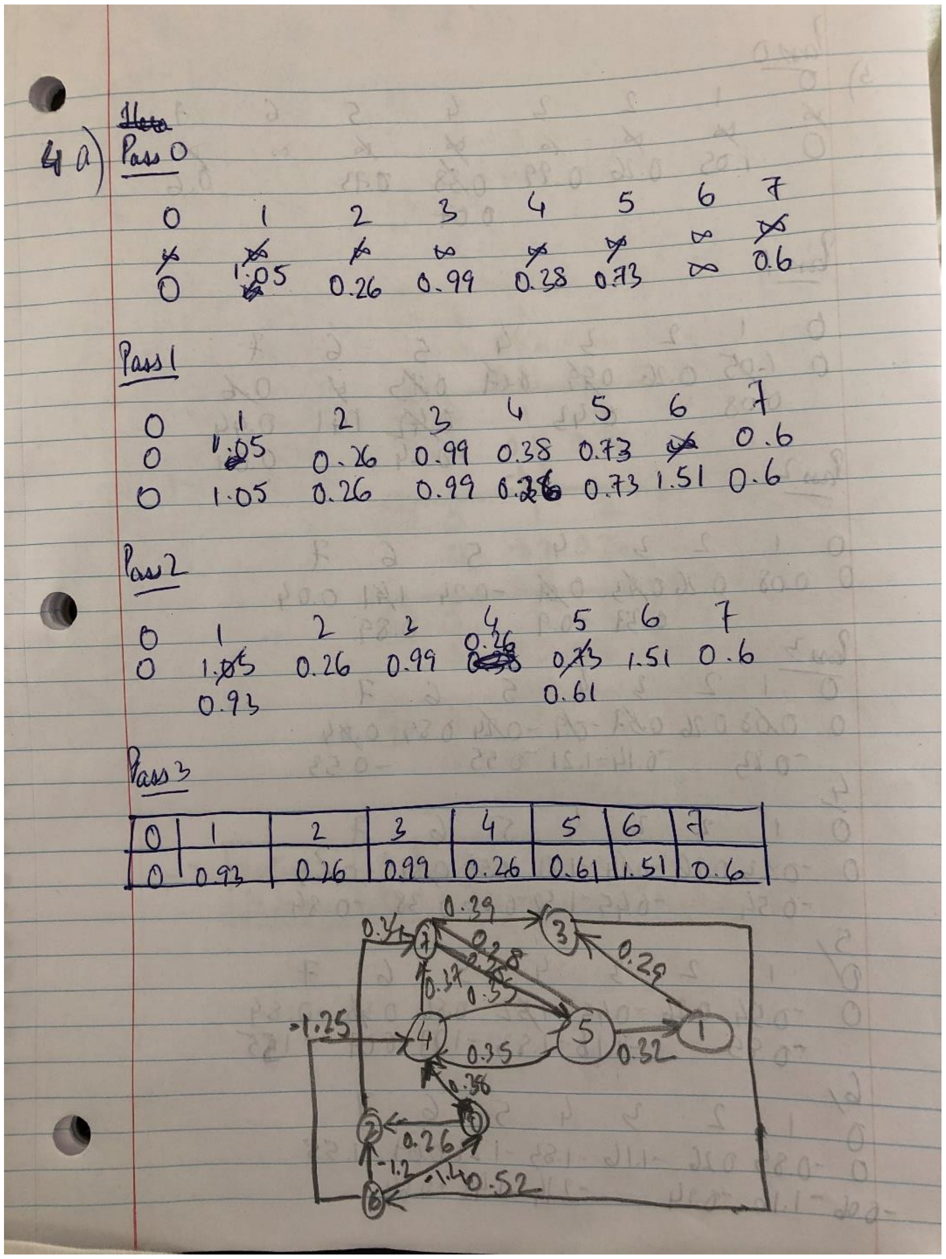
```
0 4  0.38
0 2  0.26
7 3  0.39
1 3  0.29
2 7  0.34
6 2 -1.20
3 6  0.52
6 0 -1.40
6 4 -1.25
```

b) For the digraph with a negative cycle, compute (i.e. manually trace) the progress of the Bellman-Ford Algorithm.

```
8
15
4 5  0.35
5 4 -0.66
4 7  0.37
5 7  0.28
7 5  0.28
5 1  0.32
0 4  0.38
0 2  0.26
7 3  0.39
1 3  0.29
2 7  0.34
6 2  0.40
3 6  0.52
6 0  0.58
6 4  0.93
```

The idea behind the Bellman Ford algorithm is the initial distance of all vertices are set to infinity and the source distance is set to 0. The algorithm is run V times (V being the number of vertices) and the distances are updated accordingly.

a) Since there were no updates after the 4th iteration, I stopped the running of the algorithm. The following results were obtained for the given graph:



~~Here~~

4 a) Pass 0

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

Pass 1

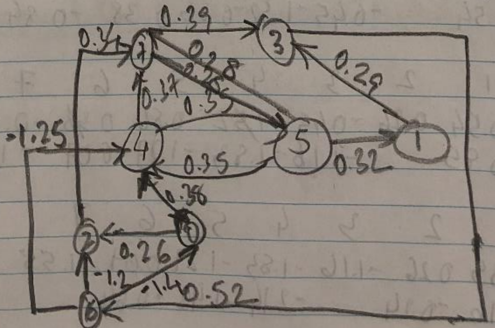
0	1	2	3	4	5	6	7
0	1.05	0.26	0.99	0.38	0.73	~~∞~~	0.6
0	1.05	0.26	0.99	0.26	0.73	1.51	0.6

Pass 2

0	1	2	2	4	5	6	7
0	~~1.05~~ 0.93	0.26	0.99	~~0.38~~ 0.26	~~0.73~~ 0.61	1.51	0.6

Pass 3

0	1	2	3	4	5	6	7
0	0.93	0.26	0.99	0.26	0.61	1.51	0.6



b) Since the given graph has a negative cycle, the weights of each edge will be updated in each iteration. Because of this, I manually computed the distances of each edge from the source 8 times, since there are 8 vertices in the graph. The following results were obtained for the given graph:

4. b)

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

0	0	1	2	3	4	5	6	7
	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	∞	~~∞~~
	0	1.05	0.26	0.99	0.38	0.73		0.6
					0.06			

1	0	1	2	3	4	5	6	7
	0	~~1.05~~	0.26	~~0.99~~	~~0.06~~	~~0.73~~	~~∞~~	~~0.6~~
		0.73		0.82	-0.25	0.41	1.51	0.43

2	0	1	2	3	4	5	6	7
	0	~~0.73~~	0.26	~~0.82~~	~~-0.25~~	~~0.41~~	~~1.51~~	~~0.43~~
		0.42		0.51	-0.56	0.1	1.34	0.12

3	0	1	2	3	4	5	6	7
	0	~~0.42~~	0.26	~~0.51~~	~~-0.56~~	~~0.1~~	~~1.34~~	~~0.12~~
		0.11		0.2	-0.87	-0.21	1.03	-0.19

4	0	1	2	3	4	5	6	7
	0	~~0.11~~	0.26	~~0.2~~	~~-0.87~~	~~-0.21~~	~~1.03~~	~~-0.19~~
		-0.2		-0.11	-1.18	-0.52	0.72	-0.5

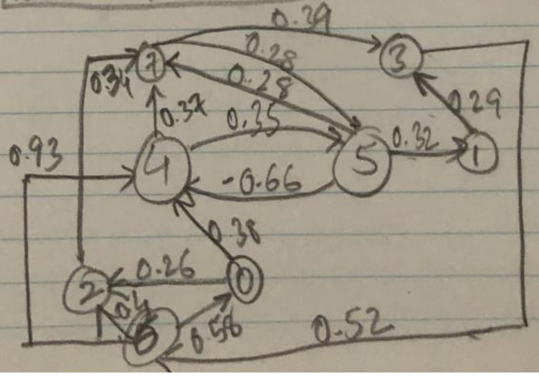
5	0	1	2	3	4	5	6	7
	0	~~-0.2~~	0.26	~~-0.11~~	~~-1.18~~	~~-0.52~~	~~0.72~~	~~-0.5~~
		-0.51		-0.42	-1.49	-0.83	0.41	-0.81

6	0	1	2	3	4	5	6	7
	0	~~-0.51~~	0.26	~~-0.42~~	~~-1.49~~	~~-0.83~~	~~0.41~~	~~-0.81~~
		-0.82		-0.73	-1.8	-1.14	0.1	-1.12

7	0	1	2	3	4	5	6	7
	0	~~-0.82~~	~~0.26~~	~~-0.73~~	~~-1.8~~	~~-1.14~~	~~0.1~~	~~-1.12~~
		-1.13	0.19	-1.04	-2.11	-1.45	-0.21	-1.43

⇒

vertex	0	1	2	3	4	5	6	7
Distance	0	-1.13	0.19	-1.04	-2.11	-1.45	-0.21	-1.43



5. **Implement a DFS and BFS traversal for the data-set of the undirected road network of New York City. The graph contains 264346 vertices and 733846 edges. It is connected, contains parallel edges, but no self-loops. The edge weights are travel times and are strictly positive.**

   The depth first search (DFS) algorithm is used for traversal of the graph. It runs on the basic principle that if a vertex is unvisited, the vertices adjacent to that vertex are recursively visited. If cycles are present in the graph, to prevent the same node from being visited again and again, a Boolean array is maintained which keeps track of all the visited vertices. The time complexity of DFS is O(V+E). Since the maximum recursion limit could not accommodate the size of the graph, I increased the recursion limit to 1,000,000.

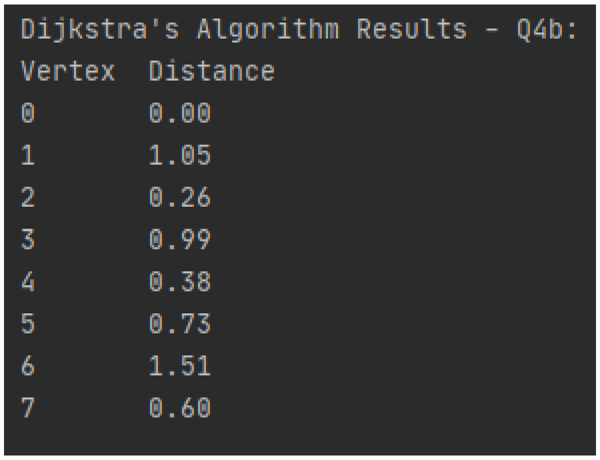
   The breadth first search (BFS) algorithm is another graph traversal algorithm which is non-recursive unlike DFS. The working principle is the same as DFS; the only difference is that DFS uses a stack to keep track of the visited vertices while BFS makes use of a queue to keep track of the visited vertices.

   Since the graph is humongous (since it covers New York City), the results take a long time to run, so I didn't include it here.

6. **Implement the shortest path using Djikstra's Algorithm for the graph in Q 4(b). Then run your implementation of Djikstra's on 4(a). What happens? Explain.**

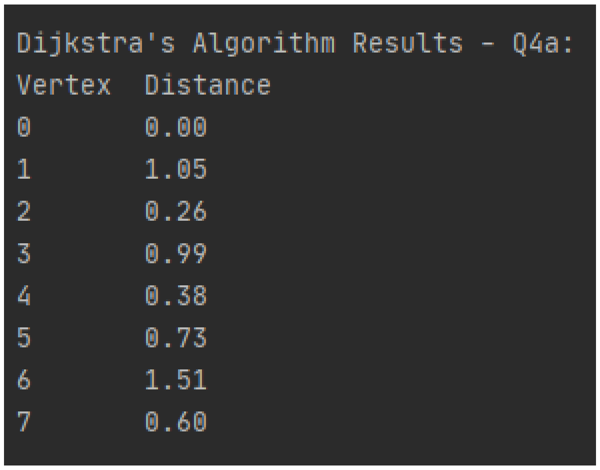
   Dijkstra's algorithm is a Greedy algorithm which obtains the shortest path from a source vertex to all the other vertices in the graph and has a time complexity of O(VlogV), using a heap as the underlying data structure. The only drawback with this algorithm is that it fails when it encounters a graph with negative edges. This is because, for Dijkstra's algorithm, once the vertex is marked as visited, it is assumed that the shortest path to that vertex has been found. This same reasoning holds true for the case of a negative cycle, where the distances to the vertices affected by the negative cycle keep getting updated. This is where the Bellman Ford algorithm comes into play – which updates the distances V times; the drawback is that it takes longer to execute, having a time complexity of O(EV).

Due to this, the following results were obtained for running Dijkstra's algorithm on Q4b:



```
Dijkstra's Algorithm Results - Q4b:
Vertex  Distance
0       0.00
1       1.05
2       0.26
3       0.99
4       0.38
5       0.73
6       1.51
7       0.60
```

For Q4a:



```
Dijkstra's Algorithm Results - Q4a:
Vertex  Distance
0       0.00
1       1.05
2       0.26
3       0.99
4       0.38
5       0.73
6       1.51
7       0.60
```