HOMEWORK 4

ECE573 Data Struct & Algo

Zhongze Tang (zt67)

2018-4-24

ECE 573 HW4 Report

Zhongze Tang (zt67)

Q1

The graph is cyclic, which means it contains at least one circle in it.

The idea of testing whether a graph is acyclic or not is simple. Just do the DFS for all the vertices in the graph. And if the next vertex we are going to visit has been visited, at the same time, this vertex is not where we come from (I call it "parent"), there must be a circle in the graph.

In the codes, what is different from original DFS is that, when we call the DFS function, we also add the vertex v's parent vertex w to the parameters list. If a vertex u from G.adj(v) has been visited, and it is not equal to w, there must be a circle, and the graph is cyclic.

Q2

Both the lazy and eager Prim's algorithms are faster than Kruskal's. Run the program five times and calculate the average running time, we can get a table like this:

Algorithm	Running Time (ns)	Time complexity
Kruskal	7222169	ElogE
Lazy Prim	2581351.8	ElogE
Eager Prim	1344047	ElogV

And the weight of MST is 10.46351.

The actual time complexity of Kruskal is E + E0logE, where E0 is the number of edges whose weight is less than the weight of the MST edge that has the highest weight. However, Kruskal's algorithm is still slower than the Prim's algorithm because, for each edge, it has to call *connected()* method, in addition to the priority-queue operations that both algorithms do for each edge process.

In this problem, the graph is a dense one (V < E), so the eager Prim performs better than other two algorithms.

Q3

We use the topological sort to find the shortest and longest path. The topological order of this digraph is: 5 1 3 6 4 7 0 2.

To build the shortest-paths tree from vertex 5, we do the steps as follows:

1. Add 5 and all edges leaving it to the tree.
2. Add 1 and 1->3 to the tree.
3. Add 3 and 3->6 to the tree.
4. Add 6 and 6->2 and 6->0 to the tree.
5. Add 4 and 4->0 to the tree.
6. Add 7 and 7->2 to the tree.
7. Add 0 to the tree.
8. Add 2 to the tree.

edgeTo()

0	1	2	3	4	5	6	7
4->0	5->1	7->2	1->3	5->4		3->6	5->7

distTo()

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

To build the longest-paths tree from vertex 5, we first negate all the weights of the edges, and then do the steps as follows:

1. Add 5 and all edges leaving it to the tree.
2. Add 1 and 1->3 to the tree.
3. Add 3 and 3->7, 3->6 to the tree.
4. Add 6 and 6->2, 6->0 and 6->4 to the tree.
5. Add 4 and 4->7, 4->0 to the tree.
6. Add 7 and 7->2 to the tree.
7. Add 0 to the tree.
8. Add 2 to the tree.

Then negate the total weight, we will get the final result.

edgeTo()

0	1	2	3	4	5	6	7
4->0	5->1	7->2	1->3	6->4		3->6	4->7

distTo()

0	1	2	3	4	5	6	7
2.55	0.32	2.77	0.61	2.06		1.13	2.43

Q4

4(a) source = 0.

① queue: 2, 4

	edgeTo[]	distTo[]
0		
1		
2	0→2	0.26
3		
4	0→4	0.38
5		
6		
7		

② queue: 7, 5

	edgeTo[]	distTo[]
0		
1		
2	0→2	0.26
3		
4	0→4	0.38
5	4→5	0.73
6		
7	2→7	0.60

③ queue: 3, 1

	edgeTo[]	distTo[]
0		
1	5→1	1.05
2	0→2	0.26
3	7→3	0.99
4	0→4	0.38
5	4→5	0.73
6		
7	2→7	0.60

④ queue: 6

	edgeTo[]	distTo[]
0		
1	5→1	1.05
2	0→2	0.26
3	7→3	0.99
4	0→4	0.38
5	4→5	0.73
6	3→6	1.51
7	2→7	0.60

⑤ queue: 4

	edgeTo[]	distTo[]
0		
1	5→1	1.05
2	0→2	0.26
3	7→3	0.99
4	6→4	0.26
5	4→5	0.73
6	3→6	1.51
7	2→7	0.60

⑥ queue: 5

	edgeTo[]	distTo[]
0		
1	5→1	1.05
2	0→2	0.26
3	7→3	0.99
4	6→4	0.26
5	4→5	0.61
6	3→6	1.51
7	2→7	0.60

⑦ queue: 1

	edgeTo[]	distTo[]
0		
1	5→1	0.93
2	0→2	0.26
3	7→3	0.99
4	6→4	0.26
5	4→5	0.61
6	3→6	1.51
7	2→7	0.60

4(b) source = 0

... ③ queue: 7, 5

	edgeTo[]	distTo[]
0		
1		
2	0→2	0.26
3		
4	0→4	0.38
5	4→5	0.73
6		
7	2→7	0.60

④ queue: 3, 1, 4

	edgeTo[]	distTo[]
0		
1	5→1	1.05
2	0→2	0.26
3	7→3	0.99
4	5→4	0.07
5	4→5	0.73
6		
7	2→7	0.60

⑤ queue: 6, 7

	edgeTo[]	distTo[]
0		
1	5→1	1.05
2	0→2	0.26
3	7→3	0.99
4	0→4	0.07
5	4→5	0.42
6	3→6	1.51
7	2→7	0.44

⑥ queue: 3, 1, 4

	edgeTo[]	distTo[]
0		
1	5→1	0.24
2	0→2	0.26
3	7→3	0.83
4	5→4	−0.59
5	4→5	0.73
6	3→6	1.51
7	2→7	0.60

→ a negative cycle.

Q5

BFS will run properly on the dataset. However, if we use the simple recursive version of DFS, the 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, repeat the third step and the fourth step. Third, we pop the first vertex in the stack. Last, we visit all the adjacent vertices of this vertex and push the unvisited vertices into the stack, then mark them as visited.

Both DFS and BFS will print out how may vertices they visit. The value should be 264346.

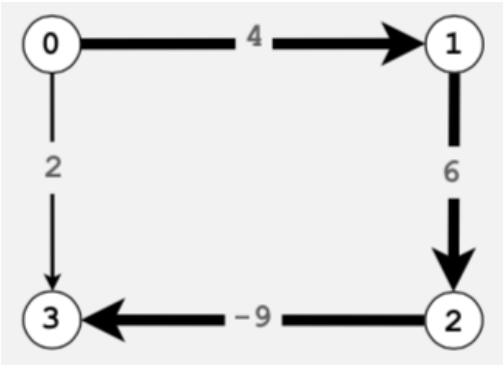
Q6

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

```
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 do not have a negative cycle. Every edge v->w will be relaxed only once in Dijkstra. However, Once a vertex has been relaxed, we consider it has the minimum distTo[], and it will never be updated later. Consider the situation below, Dijkstra selects vertex 3 immediately after 0. But 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.