Jahidul Islam - 171001155 - PM-II HW 5

Problem 1:

A)

Sorted by edge weight with consideration that we cannot have a cycle present

Kruskal's Algorithm

Edgeweight	Connection
1	7-6
2	6-5
2	8-2
4	0-1
4	2-5
6	8-6 (Creates Cycle)
7	2-3
7	7-8 (Creates Cycle)
8	1-2
8	0-7 (Creates Cycle)
9	3-4
10	5-4 (Creates Cycle)
11	1-7 (Creates Cycle)
14	3-5 (Creates Cycle)

B)
Prim's Algorithm (Highlights mark least weight or shortest path making it part of tree)

Start	Choice 1	Weight 1	Choice 2	Weight 2	Choice 3	Weight 3	Choice 4	Weight 4
8	8-7	7	8-6	6	8-2	2	-	-
2	2-8	2(Already Connected)	2-1	8	2-3	7	-	-
3	3-2	7(Already Connected)	3-4	9	3-5	14	-	-
4	4-3	9(Already Connected)	4-5	10	-	-	-	-
5	5-2	4(Creates Cycle)	5-4	10(Already Connected)	5-3	14(Creates Cycle)	5-6	2
6	6-8	6(Creates Cycle)	6-7	1	6-5	2(Already Connected)	-	-
7	7-8	7(Creates Cycle)	7-1	11	7-0	8	7-6	1(Already Connected)
0	0-7	8(Already Connected)	0-1	4	-	-	-	-
1	1-0	4(Already Connected)	1-7	11(Creates Cycle)	1-2	8(Creates Cycle)	-	-

So we stop at 1 as going further would not agree with Prim's Algorithm standards.

Problem 2:

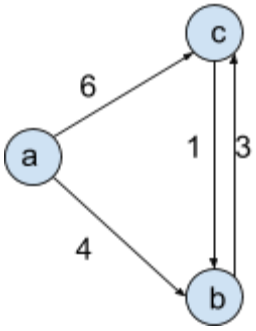
A) The way that Kruskal's algorithm works is by building up connected parts of the vertices. So we start with each vertex is its own separate part that will be a part of the tree. Kruskal's algorithm continuously checks for the lightest remaining edge and tests whether the two endpoints lie within the same connected component. If this is found to be true, that edge is discarded since adding it would mean a cycle was present in the tree. If those endpoint however, are in different components, the edge is added in and the components are merged.

B) Prim's algorithm clearly creates a spanning tree. This is done due to the fact that no cycle can be introduced by adding edges between tree and non-tree vertices. Unlike Kruskal's which deals with edges, with Prim's we add vertex to the growing spanning tree.

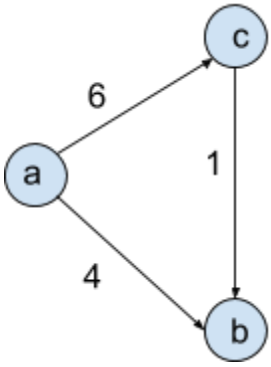
C) Connecting the red would mean those vertices connected would create a cycle and with Prim's detection of connected vertices, that path would be discarded during the process of creating the tree.

Edgeweight	Connection
1	7-6
2	6-5
2	8-2
4	0-1
4	2-5
6	8-6 (Creates Cycle)
7	2-3
7	7-8 (Creates Cycle)
8	1-2
8	0-7 (Creates Cycle)
9	3-4
10	5-4 (Creates Cycle)
11	1-7 (Creates Cycle)
14	3-5 (Creates Cycle)

Problem 3:



For Kruskal's on a directed graph we arise into issues. The following diagram displays a possible directed graph. The route c-b will be taken here due to it having the weight 1 and then stop as the directed graph indicates. What the actual desired path however, could have been a-b with weight 4 and then b-c with weight 3.



For Prim's algorithm we can view the shown directed graph for possible problems. If we started at a, Prim's algorithm would dictate we go to b due to the fact that it has distance 4. However, what we desired could have been a-c which has a higher distance of 6. Thus Prim's would not allow such a path proving to be problematic.

Problem 4:

For the eager implementation we don't need to keep track of all the edges from w to the vertices tree on the priority queue unlike the lazy implementation. For eager, we only keep track of the minimum-weight edge and then we can check if the addition of v to the tree gives any reason to update that minimum, if the edge v-w has lower weight. This can be done by now processing each edge in s, adjacency list. Thus, we have a priority queue with just one edge for each non-tree vertex, which is the shortest edge that connects it to the tree. In short, the only change in the eager version is that we do not enqueue extra values onto the binary heap when building the minimum spanning tree. Instead, we change values of vertices on the binary heap whenever something smaller is found and then the heap is restructured. With the lazy solution we consider minimum weight edges while with the eager solution we consider vertices because we do not need to have all the edges enqueued as previously explained. The lazy version of Prim's algorithm uses space proportional to E and time proportional to E log E to compute the MST of a connected edge-weighted graph with (E) edges and (V) vertices while the eager version uses space proportional to V and time proportional to E log V. This is all in the worst case consideration.

Yellow Highlight: Consideration in edges vs vertices
Blue Highlight: Change from considering all edges to keeping track of only minimum weight edge

Lazy solution. Maintain a PQ of edges with (at least) one endpoint in T.
Key = edge; priority = weight of edge.
DELETE-MIN to determine next edge e = v–w to add to T.
If both endpoints v and w are marked (both in T), disregard.
Otherwise, let w be the unmarked vertex (not in T):
- add e to T and mark w
- add to PQ any edge incident to w (assuming other endpoint not in T)

Eager solution. Maintain a PQ of vertices connected by an edge to T, where priority of vertex v = weight of lightest edge connecting v to T.
Delete min vertex v; add its associated edge e = v–w to T.
Update PQ by considering all edges e = v–x incident to v
- ignore if x is already in T
- add x to PQ if not already on it
- decrease priority of x if v–x becomes lightest edge connecting x to T

Problem 5:

A)

Check whether the new edge in the cycle formed by it and the MST has the new edge as the heaviest edge

If so then MST stays the same

Otherwise, add this edge to MST and remove the heaviest edge

B)

Check whether the new edge is the heaviest in the cycle formed by it and the MST

If so MST remains the same

Otherwise, replace heaviest weight edge and add this edge to the MST

C)

Check the cycles which this edge was a part of

If in at least one cycle this edge is the heaviest

Delete this edge and add minimum weight edge to the cycle for the MST

Otherwise the MST does not change

Problem 6:

A)(The red highlights highlight the shortest path from A-B)

v	distTo[ ]	edgeTo[ ]
A	0.0	-
1	6.0	A-1
2	5.0	A-2
3	7.0	2-3
4	13.0	1-4
5	11.0	2-5
6	13.0	3-6
7	19.0	6-7
8	21.0	4-8
B	22.0	7-B

B)

v	distTo[ ]	edgeTo[ ]	Relaxed Edge
A	0.0	-	
1	6.0	A-1	
2	5.0	A-2	A-1 [Total Count:1]
3	7.0	2-3	2-5 [Total Count 2]

4	13.0	1-4	
5	11.0	2-5	
6	13.0	3-6	[Total Count 2]
7	19.0	6-7	6-8,4-6, [Total Count 4]
8	21.0	4-8	
B	22.0	7-B	[Total Relaxed 4]

C)

v	distTo[ ]	edgeTo[ ]	Relaxed Edge
A	0.0	-	
1	6.0	A-1	
2	5.0	A-2	A-1 [Total Count:1]
3	7.0	2-3	2-5 [Total Count 2]
4	13.0	1-4	
5	11.0	2-5	
6	13.0	3-6	[Total Count 2]
7	19.0	6-7	6-8,4-6, [Total Count 4]
8	21.0	4-8	
B	22.0	7-B	[Total Relaxed 4]

Problem 7:

With a cyclic graph, you may have a cycle of 1-2-3...1-2-3… If you arrive at 1 before 2 or 3, then the sort would not be satisfied as 2 or 3 have yet to be visited. If you get to 2 before 1 or 3 then again the sorting is not satisfied. With this directed cycle, a topological sort can never be satisfied. If you need 1 and 2 for 3, and 2 and 3 for 1, then you start at 1 but need 3. They if you try to start at 3 you need 1 meaning topological sorting cannot be done without DAGs. The highlighted portion shows the point in the code that topological order is considered. If the graph is not acyclic then the directed edges and which edge to relax would not properly work to sort the graph.

public class AcyclicSP

```
{
        private DirectedEdge[] edgeTo;
        private double[] distTo;

        public AcyclicSP(EdgeWeightedDigraph G, int s)
        {
                edgeTo = new DirectedEdge[G.V()];
                distTo = new double[G.V()];

                for (int v = 0; v < G.V(); v++)
                        distTo[v] = Double.POSITIVE_INFINITY;
                distTo[s] = 0.0;

                Topological topological = new Topological(G);
                for (int v : topological.order())
                        for (DirectedEdge e : G.adj(v))
                                relax(e);
        }
}
```

Problem 8:

With Dijkstra's Algorithm, we keep two sets of information. One set contains vertices included in the shortest path tree we are creating while the other set includes vertices that are not yet part of the shortest path tree. At each step we find a vertex which is in the set that is not yet included in the SPT which also has the minimum distance from the source. Due to having these two sets a cycle detection implementation is not needed.

Problem 9:

Dijkstra's algorithm relies on the fact and only works when weights are non-negative due to its use of adding edges to determine the shortest path. Having no negatives ensures in always works optimally. An long path may be hidden by negative weights and thus an incorrect path would get selected. The highlighted parts of the code display that there would be issues when deciding which edge to relax as the calculations for weights would be changed if negative values were introduced.

```
public class DijkstraSP
{
        private DirectedEdge[] edgeTo;
        private double[] distTo;
        private IndexMinPQ pq;
```

```
	public DijkstraSP(EdgeWeightedDigraph G, int s)
	{
		edgeTo = new DirectedEdge[G.V()];
		distTo = new double[G.V()];
		pq = new IndexMinPQ(G.V());
		for (int v = 0; v < G.V(); v++)
			distTo[v] = Double.POSITIVE_INFINITY;
		distTo[s] = 0.0;
		pq.insert(s, 0.0);
		while (!pq.isEmpty())
		{
			int v = pq.delMin();
			for (DirectedEdge e : G.adj(v))
				relax(e);
		}
	}
}
private void relax(DirectedEdge e)
{
	int v = e.from(), w = e.to();
	if (distTo[w] > distTo[v] + e.weight())
	{
		distTo[w] = distTo[v] + e.weight();
		edgeTo[w] = e;
		if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
		else pq.insert (w, distTo[w]);
	}
}
```

Problem 10:

A)

S	1	4	t	
-	10/11	10/18	10/10	0+10=10

S	3	6	t	
-	10/10	10/16	10/16	10+10=20

S	2	5	t	
-	16/22	16/17	16/16	20+16=36

S	2	3	6	t	
-	20/22	4/4	14/16	14/16	36+4=40

S	2	5	6	t	
-	21/22	17/17	1/5	15/16	40+1=41

B)Mincut has no full forward facing edges and no empty backward facing edges.

Excluded From Mincut	Mincut
s-3	s-1
2-3	s-2
2-5	1-4
2-6	3-6
1-2	5-6
1-5	6-t
4-5	
5-t	
4-t	

C)Capacity of the mincut is 41