Purna Haque
PM Homework #5

1a.

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	2	5
4	0	1
7	2	3
8	0	7
9	3	4

1b.

Weight	Src	Dest
2	8	2
4	2	5
1	6	7
7	2	3
8	7	0
4	0	1
9	3	4

2a. Kruskal's algorithm uses the union-find data structure which maintains a set for each connected component in the tree. If the nodes of the edge we are looking at is in the same set, then adding that edge by merging the sets would create a cycle.

2b. Prim's algorithm uses eager implementation where a PQ of vertices connected by an edge to the tree where the priority vertex v = weight of the lightest edge connecting v to the tree. This deletes the minimum vertex, v, and adds its associated edge e = v-w to the tree. This also updates PQ considering al edges e=v-x incident to v but ignore if x is already in the tree, add x if it is not in the tree, and decrease priority of x if v-x becomes the lightest edge connecting x to the tree.

2c.

Weight	Src	Dest
1	7	6
2	6	5
4	5	2
2	2	8
7	2	3
8	7	0
4	0	1

3. Kruskal's algorithm would not work if the parent node is adjacent to and from another node while the next node pointing to the current one. Prim's algorithm would not work in a digraph whenever the parent node goes to a child node for weight less than the other child. However, now to reach the third node, you must go through that node making it inefficient.

4.
Prim's Algorithm:

```
public class LazyPrimMST
{
   private boolean[] marked;   // MST vertices
   private Queue<Edge> mst;    // MST edges
   private MinPQ<Edge> pq;     // PQ of edges

    public LazyPrimMST(WeightedGraph G)
    {
        pq = new MinPQ<Edge>();
        mst = new Queue<Edge>();
        marked = new boolean[G.V()];
        visit(G, 0);

        while (!pq.isEmpty() && mst.size() < G.V() - 1)
        {
           Edge e = pq.delMin();
           int v = e.either(), w = e.other(v);
           if (marked[v] && marked[w]) continue;
           mst.enqueue(e);
           if (!marked[v]) visit(G, v);
           if (!marked[w]) visit(G, w);
        }

   }
}
```

```
   private void visit(WeightedGraph G, int v)
   {
     marked[v] = true;
     for (Edge e : G.adj(v))
        if (!marked[e.other(v)])
           pq.insert(e);
   }

   public Iterable<Edge> mst()
   {  return mst;  }
```

Prim Eager's Algorithm:

```
public class PrimMST {
    private static final double FLOATING_POINT_EPSILON = 1E-12;
```

```
    private Edge[] edgeTo;        // edgeTo[v] = shortest edge from tree
vertex to non-tree vertex
    private double[] distTo;      // distTo[v] = weight of shortest such edge
    private boolean[] marked;     // marked[v] = true if v on tree, false
otherwise
    private IndexMinPQ<Double> pq;

    /**
     * Compute a minimum spanning tree (or forest) of an edge-weighted graph.
     * @param G the edge-weighted graph
     */
    public PrimMST(EdgeWeightedGraph G) {
        edgeTo = new Edge[G.V()];
        distTo = new double[G.V()];
        marked = new boolean[G.V()];
        pq = new IndexMinPQ<Double>(G.V());
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;

        for (int v = 0; v < G.V(); v++)      // run from each vertex to find
            if (!marked[v]) prim(G, v);      // minimum spanning forest

        // check optimality conditions
        assert check(G);
    }

    // run Prim's algorithm in graph G, starting from vertex s
    private void prim(EdgeWeightedGraph G, int s) {
        distTo[s] = 0.0;
        pq.insert(s, distTo[s]);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            scan(G, v);
        }
    }

    // scan vertex v
    private void scan(EdgeWeightedGraph G, int v) {
        marked[v] = true;
        for (Edge e : G.adj(v)) {
            int w = e.other(v);
            if (marked[w]) continue;         // v-w is obsolete edge
            if (e.weight() < distTo[w]) {
                distTo[w] = e.weight();
                edgeTo[w] = e;
                if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
                else                pq.insert(w, distTo[w]);
            }
        }
    }

    /**
     * Returns the edges in a minimum spanning tree (or forest).
     * @return the edges in a minimum spanning tree (or forest) as
     *    an iterable of edges
     */
```

```
    public Iterable<Edge> edges() {
        Queue<Edge> mst = new Queue<Edge>();
        for (int v = 0; v < edgeTo.length; v++) {
            Edge e = edgeTo[v];
            if (e != null) {
                mst.enqueue(e);
            }
        }
        return mst;
    }

    /**
     * Returns the sum of the edge weights in a minimum spanning tree (or
forest).
     * @return the sum of the edge weights in a minimum spanning tree (or
forest)
     */
    public double weight() {
        double weight = 0.0;
        for (Edge e : edges())
            weight += e.weight();
        return weight;
    }


    // check optimality conditions (takes time proportional to E V lg* V)
    private boolean check(EdgeWeightedGraph G) {

        // check weight
        double totalWeight = 0.0;
        for (Edge e : edges()) {
            totalWeight += e.weight();
        }
        if (Math.abs(totalWeight - weight()) > FLOATING_POINT_EPSILON) {
            System.err.printf("Weight of edges does not equal weight(): %f
vs. %f\n", totalWeight, weight());
            return false;
        }

        // check that it is acyclic
        UF uf = new UF(G.V());
        for (Edge e : edges()) {
            int v = e.either(), w = e.other(v);
            if (uf.connected(v, w)) {
                System.err.println("Not a forest");
                return false;
            }
            uf.union(v, w);
        }

        // check that it is a spanning forest
        for (Edge e : G.edges()) {
            int v = e.either(), w = e.other(v);
            if (!uf.connected(v, w)) {
                System.err.println("Not a spanning forest");
                return false;
            }
```

```
        }

        // check that it is a minimal spanning forest (cut optimality
conditions)
        for (Edge e : edges()) {

            // all edges in MST except e
            uf = new UF(G.V());
            for (Edge f : edges()) {
                int x = f.either(), y = f.other(x);
                if (f != e) uf.union(x, y);
            }

            // check that e is min weight edge in crossing cut
            for (Edge f : G.edges()) {
                int x = f.either(), y = f.other(x);
                if (!uf.connected(x, y)) {
                    if (f.weight() < e.weight()) {
                        System.err.println("Edge " + f + " violates cut
optimality conditions");
                        return false;
                    }
                }
            }

        }

        return true;
    }
```

Prim's algorithm maintains a PQ of edges with at least one endpoint in a tree where the key is the edge and priority is weight of the edge. You delete the minimum to determine the next edge to add to the tree. If both endpoints are already in the tree, the disregard the edge, otherwise let one vector be unmarked vertex and add the other to the tree while marking the unmarked vertex. While, Eager Prim's algorithm does the same except decreased the priority of a vertex if that edge becomes the lightest edge to the tree

5a. When a new edge is added into the graph, it will affect the MST if the cycle is formed by taking edges of MSTs along with the new edge. Create a code that says if the new edge has maximum weight in the cycle, then the MST will remain the same, else add this edge to MST and remove the edge with maximum weight in the cycle from tree. The overall time complexity will be O(V).

5b. Create a code that says if the reduced edge has maximum edge in the cycle then MST remains the same, else replace maximum weight edge in this cycle from MST and add the reduced edge in MST. The overall time complexity is also O(V).

5c. Create a code that says if this edge is maximum weighted edge in at least one cycle, then MST will delete this edge and add the minimum edge in that cycle, else if this edge is not maximum weighted edge among any cycle, then MST will not change. The overall time complexity will be O(V+E).

6a.

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

6b.

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

6c.

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

7.

```
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);

   }
}
```

The Topological Sort Algorithm requires a DAG because it is a linear ordering of vertices such that for every directed edge, one vertex comes before the other in the ordering. If it is not a DAG an error will occur because it will not be in order so therefore the error will occur within the for loop where v goes through the graph in order.

8. With Dijkstra's algorithm, we keep two sets of information; one set that contains vertices included in the shortest path tree and another set that 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 s 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.

9.

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

   public DijkstraSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(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]);
   }
}
```

Within the for loop we see that the distTo[] for any of the vertices is always positive numbers up to infinity which fails to acknowledge negative edge weights because the Dijkstra's algorithm does not go back to vertices already included in the shortest path tree.

10a.

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

10b.

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	

10c. Capacity of the mincut is 41.