Desmond Johnson – drj64 (NetID)

1a.

Edge	Edge's Weight
7-6	1
6-5	2
2-8	2
0-1	4
2-5	4
2-3	7
1-2	8
3-4	9

This table and the one below (1b.) indicate the order they were added to the MST.

Edges that were ignored because they would create a cycle in the order, they were observed are 6-8, 7-8, 0-7, and 5-4.

1b.

Edge	Edge's Weight
2-8	2
2-5	4
6-5	2
7-6	1
2-3	7
1-2	8
0-1	4
3-4	9

Edges that were ignored because they would create a cycle in the order, they were observed are 6-8, 7-8, and 0-7.

2a.

Kruskal's Algorithm implements cycle detection by using the union find data structure. When the algorithm is first called it creates V subsets where V is the total number of vertices. Each subset is independent of one another. As each edge is checked in ascending order the algorithm checks to see if the two vertices belonging to the corresponding edge are part of the same subset. This is the find portion of the union find data structure. If they are part of the same sub-set then adding this edge to MST would create a cycle. That indicates that this edge should be skipped. If they are not part of the same sub-set the edges are unionized which means each vertex's subset is combined. After this point both vertices are part of the same subset. The algorithm ends when there are V-1 edges in the MST

2b.

Prim's Algorithm implements cycle detection by marking each vertex connected to edge as visited when it is added to the MST. When the algorithm first runs every vertex is marked as not visited. The code then looks for the edge with the smallest edge weight that is adjacent to the starting vertex. The code checks to see if both vertices associated with the corresponding edge have been visited. If both vertices have been visited this edge is not added because it would create a cycle. If at least one of the two vertices have not been visited the edge is added to the MST and both vertices are marked as visited. This process continues until there are V-1 edges. The algorithm will always look for the edge with the smallest weight that is adjacent to all of the visited vertices.

2c.

Since Kruskal's algorithm does not start at one vertex and greedily expand out it will incorrectly skip edges that should be part of the MST. This means that the resulting output would not be an MST since edges would not be connected. The table below outlines when the code breaks.

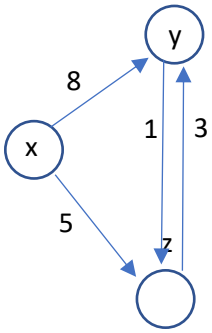
Edge	Edge's Weight	Have both vertices been visited?
7-6	1	No (add to MST)
6-5	2	No (add to MST)
2-8	2	No (add to MST)
0-1	4	No (add to MST)
2-5	4	No (add to MST)
6-8	6	Yes (skip edge)
2-3	7	No (add to MST)
1-2	8	Yes (skip edge)
0-7	8	Yes (skip edge)
3-4	9	No (add to MST)

As shown in the table above, the issue that edge 1-2 was not added to the MST because of Prim's cycle detection. This has caused the output to be disconnected which means that the output is not an MST.

3.

For a directed graph to be an MST one vertex (the root) has to be able to reach every other vertex in the graph. The other rules for an undirected MST also apply to a directed MST

Kruskal's Algorithm



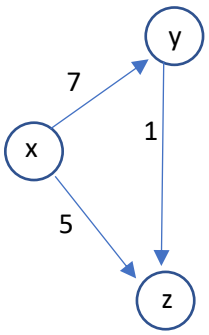
Edge	Edge's Weight	Are both vertices in the same set?
y->z	1	No (add to MST)
z->y	3	Yes (skip edge)

x->z	5	No (add to MST)
x->y	8	Yes (skip edge)

The following graph showcase why Kruskal's Algorithm does not work on directed graphs. The table below shows the steps of the algorithm on the graph.

The table above shows that the result of the algorithm is not a spanning tree. This is because there is not a vertex that can reach every other vertex for the graph. If the root of the graph was x the MST for this graph should include edges x->z, and z->y.

Prim's Algorithm



The following graph showcase why Prim's Algorithm does not work on directed graphs. The table below shows the steps of the algorithm on the graph assuming that x is the starting vertex and the root of the MST.

Edge	Edge's Weight	Have both verifies been visited?
x->z	5	No (add to MST)
y->z (not looked at because it is not on the adjacency list of x or z)		
x->y	7	Yes (skip edge)

The table above shows that the result of the algorithm is a spanning tree, but not a MST. If edges x->y were used the cost of the tree would be 8 instead of the tree that Prim's algorithm found which is 12.

4.

Prim's Lazy Code

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

    private double weight;       // total weight of MST
    private Queue<Edge> mst;     // edges in the MST
    private boolean[] marked;    // marked[v] = true iff v on tree
    private MinPQ<Edge> pq;      // edges with one endpoint in tree

    /**
     * Compute a minimum spanning tree (or forest) of an edge-weighted graph.
     * @param G the edge-weighted graph
     */
    public LazyPrimMST(EdgeWeightedGraph G) {
        mst = new Queue<Edge>();
        pq = new MinPQ<Edge>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)      // run Prim from all vertices to
            if (!marked[v]) prim(G, v);      // get a minimum spanning forest

        // check optimality conditions
```

```
        assert check(G);
    }

    // run Prim's algorithm
    private void prim(EdgeWeightedGraph G, int s) {
        scan(G, s);
        while (!pq.isEmpty()) {                      // better to stop when
mst has V-1 edges
            Edge e = pq.delMin();                    // smallest edge on pq
            int v = e.either(), w = e.other(v);      // two endpoints
            assert marked[v] || marked[w];
            if (marked[v] && marked[w]) continue;    // lazy, both v and w
already scanned
            mst.enqueue(e);                          // add e to MST
            weight += e.weight();
            if (!marked[v]) scan(G, v);              // v becomes part of
tree
            if (!marked[w]) scan(G, w);              // w becomes part of
tree
        }
    }

    // add all edges e incident to v onto pq if the other endpoint has not
yet been scanned
    private void scan(EdgeWeightedGraph G, int v) {
        assert !marked[v];
        marked[v] = true;
        for (Edge e : G.adj(v))
            if (!marked[e.other(v)]) pq.insert(e);
    }
```

Prim's Eager Code

```
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]);
            }
        }
    }
```

The code shown for each implementation shows that the major difference between the two implementation is that the lazy implementation stores a priority queue of adjacent edges to marked vertices with their corresponding weight value. The eager implementation creates a priority queue of adjacent vertices to marked vertices that contain information about the corresponding edge associated with that vertex and what the weight of it is (distTo). The distTo of the starting vertex is set to 0 and then placed in the PQ. If the eager implementation finds a new edge that is associated with one of the vertices in the PQ and the vertex has not been visited the weight (distTo) and the corresponding edge (edgeTo) for that vertex is updated in the PQ. The eager implementation also initially sets the distTo of every vertex to infinity to ensure that distTo can be updated properly in the situation outlined in the previous sentence. Since the lazy implementation stores edges which can not be replaced the PQ will have leftover edges after the algorithm finishes. The eager implementation will not have any extra vertices in the PQ after the algorithm finishes.

The PQ of the lazy implementation stores edges, so the space complexity is equal to O(E) where E is the total number of edges in a graph. Since the PQ of the eager implementation is stores vertices. The space complexity is O(V) where V is the total number of vertices in the graph. Since a graph almost always has less edges than vertices the eager implementation has the better space complexity. The time to delete, insert, and decrease a key for the eager implementation is O(log(V)). The lazy implementation doesn't have to decrease keys so the only task necessary are insert and delete which both take O(log(E)). Since the frequency of the delete, insert, and decrease (for eager) is O(E) the total time complexity of the eager implementation is O(Elog(V)) and the total time complexity of the lazy implementation is O(Elog(E)).

5a.

To simplify the explanation, I am assuming that the edge added to the graph connects vertex u to v

1. Perform a recursive tree traversal of the MST (inorder, postorder, preorder) from vertex u to v
2. Each time you step out of a recursion level keep track of the which edge is the max
3. If the maximum edge discovered is larger than the one added
    a. Delete the maximum edge from the MST and then add the new edge to the MST

The time complexity of the recursive tree traversal is proportional to the number of nodes or in this case vertices of the tree which mean that the time complexity for the traversal is O(V). The time complexity to add and remove an edge from a tree is O(1) which means the total time complexity of the algorithm is O(V).

5b.

To simplify the explanation, I am assuming that the edge changed connects vertex u to v

1. Add the edge to the MST (creates cycle)
2. Locate the cycle in the graph by performing BFS traversal on MST with the edge added starting at vertex u
3. Remove the maximum weight located on the cycle

The time complexity to perform BFS traversal and remove the maximum weight is O(V) since the total number of edges in a MST is V-1. The total time to add the edge to the MST is O(1) which means the total run time of this algorithm is O(V).

5c.

To simplify the explanation, I am assuming that the edge changed connects vertex u to v

1. Remove the edge from the MST
2. Perform BFS traversal on the two sub trees starting at vertex u and v
3. While performing the traversals mark each vertex that belongs to subtree u and subtree v
4. Examine each edge of the graph (including the one that was increased) and find the smallest one that connects the two subsets together.
5. Add the corresponding edge to the MST

Removing and adding a edge to MST takes O(1). The time complexity to perform BFS traversal for each sub tree is O(V) since the total number of edges in the MST is V-1. The time to examine each edge is O(E) since every edge has to be viewed. This means the total time complexity is O(V)+ O(E) = O(V+E).

6a.

Pass 1

Vertex	A	1	2	3	4	5	6	7	8	B
distTo[]	0	6	5	7	13	11	13	18	21	21
edgeTo[]	-	A->1	A->2	2->3	1->4	2->5	3->6	5->7	4->8	7->B

Pass 2

Vertex	A	1	2	3	4	5	6	7	8	B
distTo[]	0	6	5	7	13	11	13	18	21	21
edgeTo[]	-	A->1	A->2	2->3	1->4	2->5	3->6	5->7	4->8	7->B

Since the distTo did not change for any vertex during pass 2 it will not change for the remaining passes. This means the shortes path to B is A->2->5->7->B with B having a distTo of 21.

6b.

Vertex	A	1	2	3	4	5	6	7	8	B
distTo[]	0	6	5	7	13	11	13	18	21	21
edgeTo[]	-	A->1	A->2	2->3	1->4	2->5	3->6	5->7	4->8	7->B

The shortest path and the corresponding distTo is the same as problem 6a.

(Vertical Table was used here and in 6c. since a horizontal table would not format properly with the content in the table).

Edge	Relax Check Count	u->v caused distTo[v] /edgeto[v] to be updated when checked
A->1	1	Y
A->2	1	Y
1->3	1	N
1->4	1	Y
2->3	1	Y
2->5	1	Y
3->6	1	Y
4->6	1	N
4->8	1	Y
5->6	1	N
5->7	1	Y
5->B	1	Y
6->7	1	N
6->8	1	N
7->8	1	N
7->B	1	Y
8->B	1	N

6c.

Topological order chosen = A,2,5,1,4,3,6,7,8,B

Vertex	A	1	2	3	4	5	6	7	8	B
distTo[]	0	6	5	7	13	11	13	18	21	21
edgeTo[]	-	A->1	A->2	2->3	1->4	2->5	3->6	5->7	4->8	7->B

The shortest path and the corresponding distTo is the same as problem 6a.

Edge	Relax Check Count	u->v caused distTo[v] /edgeto[v] to be updated when checked
A->1	1	Yes
A->2	1	Yes
1->3	1	No
1->4	1	Yes
2->3	1	Yes
2->5	1	Yes
3->6	1	Yes
4->6	1	Yes
4->8	1	Yes
5->6	1	Yes
5->7	1	Yes
5->B	1	Yes
6->7	1	No
6->8	1	No
7->8	1	No
7->B	1	Yes
8->B	1	No

7.

The topological sort algorithm only works on acyclic graphs because if there was a cycle in the graph the ordering that the topological sort would produce would not be valid and contradict the definition of topological ordering. “Topological ordering is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering” (Wikipedia). If the vertex u had a directed edge going to v and vertex v had a directed edge going from v to u there would be a cycle and the generated ordering from a topological sort would violate the definition. For example if the order was u,v the edge going from v to u would contradict the definition of topological ordering. This means that the order obtained by running a topological sort would not be valid and produce different results depending on the order of the adjacency list of each vertex. The below code shows where the topological sort is called and used. Since the order of vertices obtain from the sort is not valid and can vary significantly depending on the adjacency list. The algorithm cannot consistently find the shortest path.

```
public class AcyclicSP

{

 private DirectedEdge[] edgeTo;

 private double[] distTo;

 public AcyclicSP(EdgeWeightedDigraph G, int s)
```

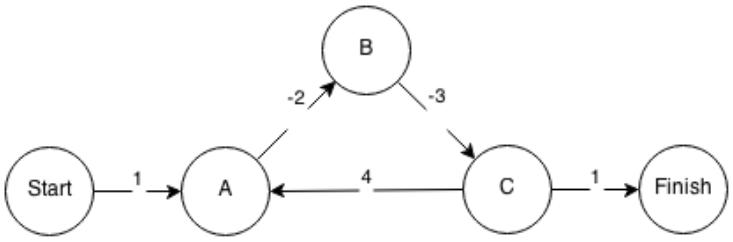
```
{
edgeTo = new DirectedEdge[G.V()];
distTo = new double[G.V()];
for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
distTo[s] = 0.0;
Topological topological = new Topological(G);
for (int v : topological.order())
        for (DirectedEdge e : G.adj(v))
                relax(e);

}
}
```

8.

The primary reason we don't implement cycle detection for Djikstra's Algorithm is because we are finding the shortest path not an MST. There is no restriction that the shortest path from one vertex to another be achieved without going through a cycle. A spanning tree can not detect a cycle which is means it needs to be checked for to ensure that the result is accurate. Additionally, Prim's algorithm chooses the next vertex by determining the closest vertex (lowest weighted edge) to the tree. The tree is all of the edges that have been visited so far and all of the edges that connect the corresponding vertices. Dijkstra's algorithm chooses the next vertex by determining the closest vertex (lowest weighted edge) to the source. This is because Dijkstra's algorithm is trying to find the shortest path. Assuming that all of the edges are positive, having a cycle in a graph would not cause any issues for Dijkstra's algorithm because following a cycle back to its starting point could never result in a shortest path. This theory does not hold if the graph has negative cycles, but Dijkstra's algorithm (traditional) can fail if there is a negative edge and the graph has no cycles.

9.

Dijkstra's algorithm does not work for negative edges because it operates under the assumption that once you go to the next vertex (shortest path from source) you no longer need to look at that vertex. This is because if the edges weren't negative it would be impossible for any other path from the source vertex to the next vertex selected to have a shorter path. The algorithm showcased below slightly modifies the algorithm to work with negative edges by reading a vertex to the PQ if the corresponding edge needs to be relaxed, but the algorithm will still fail if there is a negative cycle.



For the graph above, the algorithm will begin at the Start vertex and relax its edge. This will set the value of distTo for A to 1. Vertex A will also be added to the PQ. Then A immediately gets removed from the PQ and the edge A->B gets relaxed setting the value of distTo for B equal to -1, and B gets added to the PQ. Next B gets removed from the PQ and edge B->C gets relaxed which updates the distTo[] for C to -4. C gets added to the PQ and immediately after gets removed. After it is removed edges C->A and C->Finish causing the distTo[] of Finish to be -3 and A to be equal to 0. Since the distTo[] of A had to be lowered again it and Finish are added to the PQ. The problem occurs that an infinite loop will occur with the path A->B->C because the shortest path for A, B, and C can keep being lowered by going around the cycle repeatedly. This means the PQ will never be empty which means the code will never finish running. Aside the issue with algorithm that causes negative edges to fail, the issue that causes the infinite loop in the code is highlighted below in yellow.

```
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]);
}
}
```

10a.

Pass 1

s	1	4	t	Current total flow to t
-	10/11	10/18	10/10	10

Pass 2

s	3	6	t	Current total flow to t

-	10/10	10/16	10/16	20

Pass 3

s	2	5	t	Current total flow to t
-	16/22	16/17	16/16	36

Pass 4

s	2	3	6	t	Current total flow to t
-	20/22	4/4	14/16	14/16	40

Pass 5

s	2	5	6	t	Current total flow to t
-	21/22	17/17	1/5	15/16	41

The total flow for t is equal to 41

10b.

Set with source = {s, 1, 2, 4}

Set with sink = {3, 6, 5, t}

10c.

The capacity of the mincut is = 10 + 4 +17 +10 = 41