Andy Guo
172004093

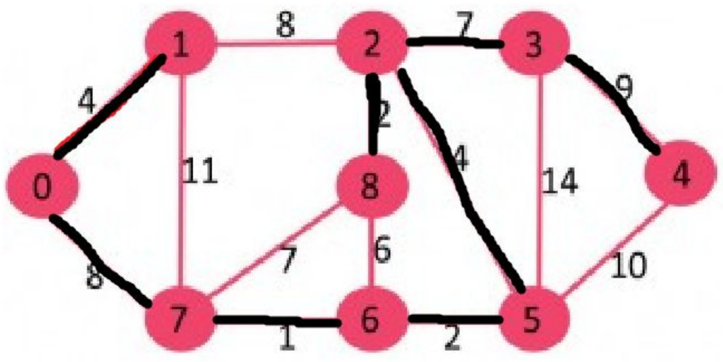
Assignment 5

1) For the graph pictured below, complete two things

a. Kruskal's MST

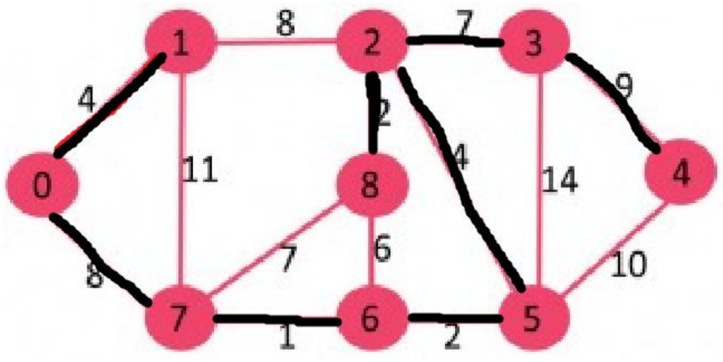
MST Edges	6-7	2-8	5-6	2-5	1-0	6-8	7-8	3-2	0-7	1-2	3-4	5-4	1-7	3-5
Edge Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14

RED means it is a traversable path



b. Prim's MST

MST Edges	**2-8**	**2-5**	**5-6**	**6-7**	**7-0**	**0-1**	**2-3**	**3-4**
Edge Weight	**2**	**4**	**2**	**1**	**8**	**4**	**7**	**9**



2) Answer

a. How does Kruskal's Algorithm implement cycle detection?

Kruskal's Algorithm implements the cycle detection with a concept like that of an adjacency matrix. You create a matrix length of v-1 and index each vertex from 0 to v-1. When we add an edge, you will update the matrix to edge V and W have the same values or both edges were visited. This implementation is also known as the Union Find algorithm. In this case we will do union(v,w) every time an edge v-w is added. And we can find (v2,w2) any time a new edge v2-w2 is added.

b. How does Prim's Algorithm implement cycle detection?

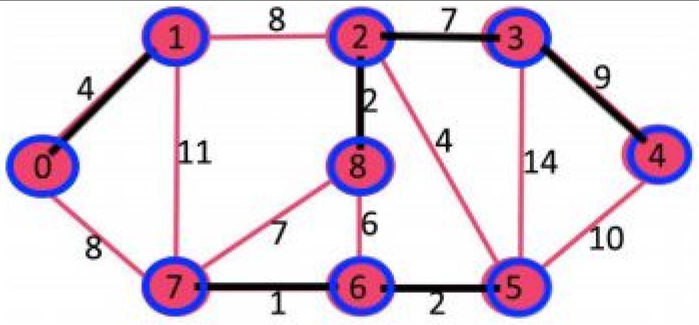
Prim's Algorithm implements cycle detection by using the priority queue. First it will sort the edges in descending order according to the edge weights. From there it will delete the edge in each iteration, essentially it will be "marked" in the system. This way we can check to see that it will not be a cycle by checking if the graph is still connected.

c. What would happen if you tried to use Prim's Algorithm cycle detection in Kruskal's Algorithm?

In Prim's algorithm there's a priority queue. Each time an iteration happens it will check to see if its vertices is marked or unmarked.

*Everything in red is traversed

MST Edges	6-7	2-8	5-6	2-5	1-0	6-8	7-8	3-2	0-7	1-2	3-4	5-4	1-7	3-5
Edge Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
Marked	**6,7**	**2,8**	**5**		**0,1**			**3**			**4**			

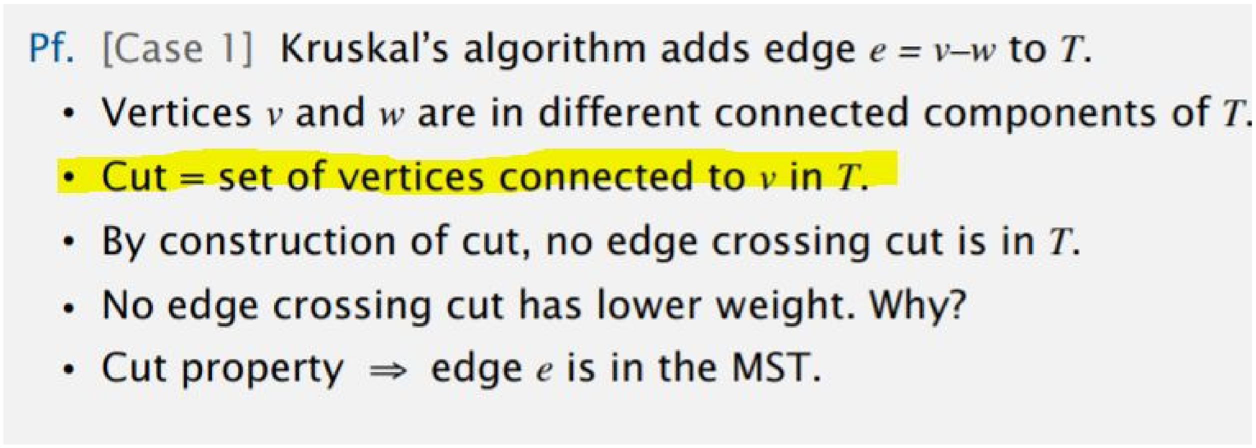


The black line is the paths and the blue circles is the marked paths

If we used cycle detection for Kruskal's we can see that this does not reproduce the correct graph. This is because if we were to go through this, everything will happen normally. But at 5-2, we can see that at vertices 2 and 5 we already have those two linked therefore if we were to use cycle detection it will not link the edge. This problem is shown for 0-7 as well. The cycle detection only makes sure that the two vertices that its trying to connect are both unmarked. If they are both marked then the edge is ignored. This causes the wrong output graph since some of the edges will not be considered.

3) What happens if you apply Kruskal and Prim's algorithm to a directed graph?

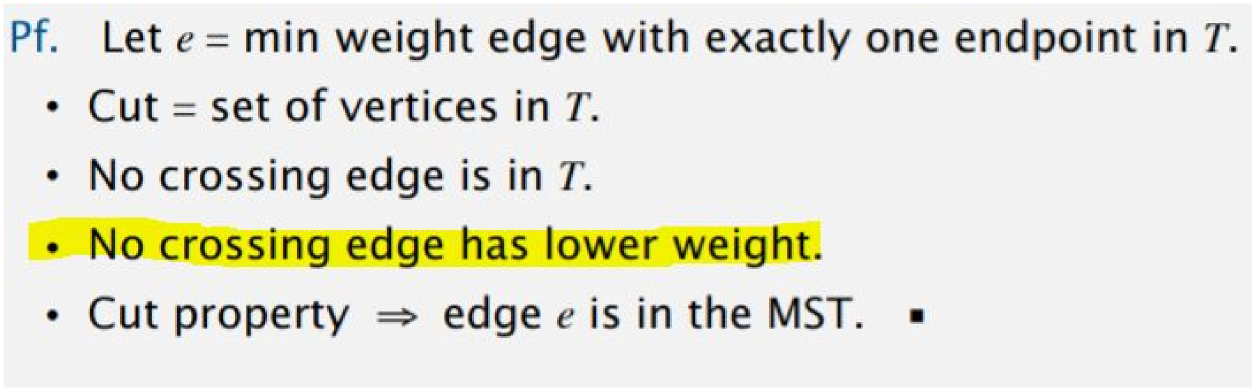
Using Kruskal's Algorithm to a directed graph will not work properly. Because these cases are mainly used for undirected graph. This means it assumes that every node is reachable from every node. Using this assumption it will go from edge weights that is the least in its connecting edges. Then that edge that has the lowest weight will it's path. However it does not take account that that edge might be the wrong direction.



Pf. [Case 1] Kruskal's algorithm adds edge $e = v–w$ to T.

- Vertices v and w are in different connected components of T.
- Cut = set of vertices connected to v in T.
- By construction of cut, no edge crossing cut is in T.
- No edge crossing cut has lower weight. Why?
- Cut property $\Rightarrow$ edge e is in the MST.

The same idea is applied to Prim's algorithm since it assumes the same thing. It will order the edge weights and go from there. However it doesn't look at the direction that the arrow is point. Thus it will create the same error as Kruskal's method.



Pf. Let e = min weight edge with exactly one endpoint in T.

- Cut = set of vertices in T.
- No crossing edge is in T.
- No crossing edge has lower weight.
- Cut property $\Rightarrow$ edge e is in the MST. ▪

4) Prim’s Algorithm and Eager Prim’s Algorithm.Explain the differences in these two implementations by highlighting the changed parts(from naïve to eager) and explain how this affects space and time complexities. Explain all of the change, including the switch from vertices to edges.

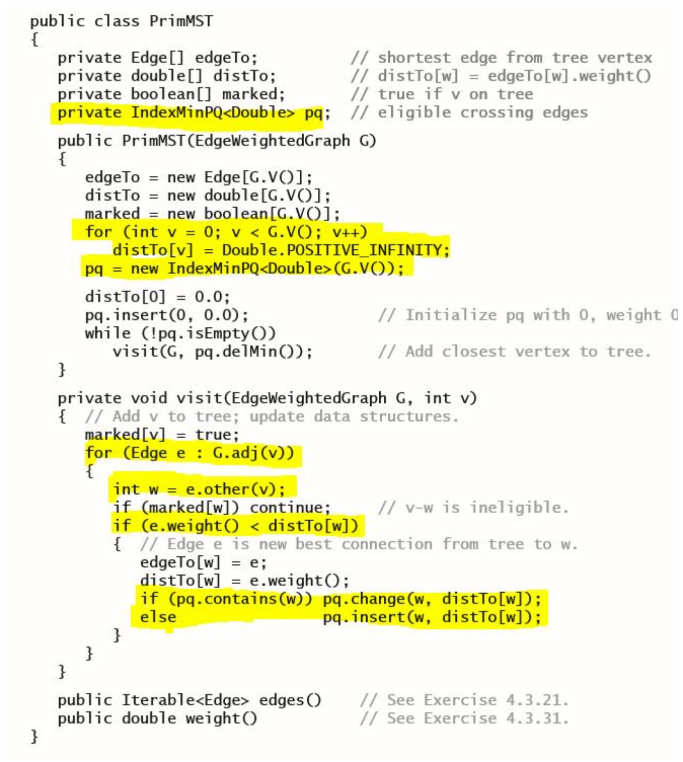
The difference between eager and lazy is that lazy runs the algorithm through priority queue and a MST using only the edges and its edge weights. To do this it will list all the adjacent vertices of a vertex and storing the edges and edge weight in the priority queue. The minimum edges are found and then deleted from the queue and inserting that into the MST. This process is repeated over again and again by traversing through the graph while finding its minimum edge. Throughout this process it will also eliminate all the redundant edges. This will make sure that there isn’t an edge that will connect two edges that are already marked, this is done so there is no cycles.

Eager Prim’s algorithm does a similar process, it also uses a priority queue that is sorted by increasing edge weights and an MST. But this time it will store only the relevant connections to a given vertex, instead of only constantly finding the shortest edge to traverse through. Using the same technique as lazy prim essentially. But unlike lazy prim it does not enqueue any extra values into the queue. This means it will take less space. While doing this it will also update the shortest path found at any given vertex.

Eager’s time complexity is O(E log(V)) while the space is O(V) as for lazy implementation it is O(E log(E)) with a space of O(E).

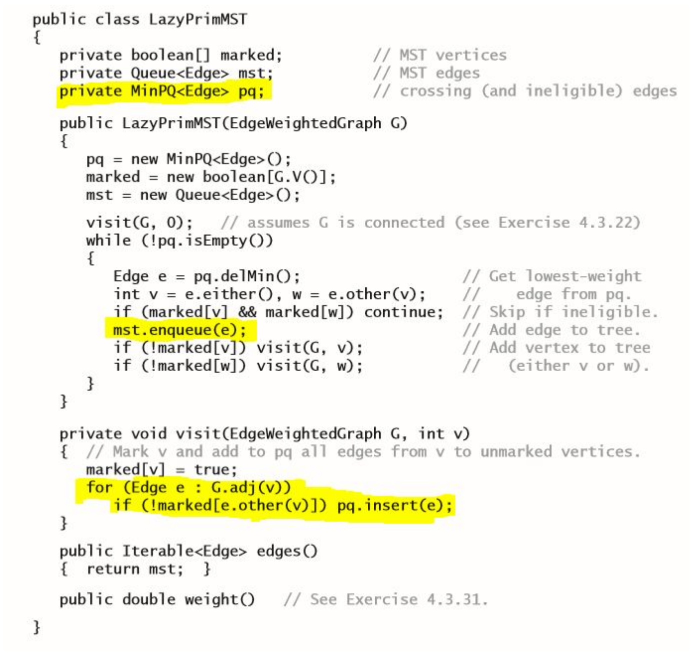
The difference between the two code and be found below (in yellow highlight):

Eager Prim's Algorithm code:



```
public class PrimMST
{
   private Edge[] edgeTo;          // shortest edge from tree vertex
   private double[] distTo;        // distTo[w] = edgeTo[w].weight()
   private boolean[] marked;       // true if v on tree
   private IndexMinPQ<Double> pq;  // eligible crossing edges

   public PrimMST(EdgeWeightedGraph G)
   {
      edgeTo = new Edge[G.V()];
      distTo = new double[G.V()];
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      pq = new IndexMinPQ<Double>(G.V());

      distTo[0] = 0.0;
      pq.insert(0, 0.0);              // Initialize pq with 0, weight 0
      while (!pq.isEmpty())
         visit(G, pq.delMin());       // Add closest vertex to tree.
   }

   private void visit(EdgeWeightedGraph G, int v)
   {  // Add v to tree; update data structures.
      marked[v] = true;
      for (Edge e : G.adj(v))
      {
         int w = e.other(v);
         if (marked[w]) continue;     // v-w is ineligible.
         if (e.weight() < distTo[w])
         {  // Edge e is new best connection from tree to w.
            edgeTo[w] = e;
            distTo[w] = e.weight();
            if (pq.contains(w)) pq.change(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
         }
      }
   }

   public Iterable<Edge> edges()    // See Exercise 4.3.21.
   public double weight()           // See Exercise 4.3.31.
}
```

Lazy Prim's Algorithm code:



```
public class LazyPrimMST
{
   private boolean[] marked;          // MST vertices
   private Queue<Edge> mst;           // MST edges
   private MinPQ<Edge> pq;            // crossing (and ineligible) edges

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

      visit(G, 0);   // assumes G is connected (see Exercise 4.3.22)
      while (!pq.isEmpty())
      {
         Edge e = pq.delMin();                    // Get lowest-weight
         int v = e.either(), w = e.other(v);      //    edge from pq.
         if (marked[v] && marked[w]) continue;    // Skip if ineligible.
         mst.enqueue(e);                          // Add edge to tree.
         if (!marked[v]) visit(G, v);             // Add vertex to tree
         if (!marked[w]) visit(G, w);             //   (either v or w).
      }
   }

   private void visit(EdgeWeightedGraph G, int v)
   {  // Mark v and add to pq all edges from v to unmarked vertices.
      marked[v] = true;
      for (Edge e : G.adj(v))
         if (!marked[e.other(v)]) pq.insert(e);
   }

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

   public double weight()   // See Exercise 4.3.31.
}
```

5) Imagine a client presents to you a graph, G, and a MST, already generated, as a variable.

a. To add an edge to MST we need to find a way to optimize the existing MST. First we will need to replace the existing minimum in the MST with the potential new edge. If the edge that is larger than the existing maximum weight in the MST, then the MST will stay the same.

```
i. Pseudocode:
    1. For a given MST, T, check the maximum edge weight in the tree
        a. Do this by running the maximum function max(T), this will find the
           maximum edge weight
    2. Find the new edge's edge weight
    3. Add edge to the priority queue
        a. If new edge has weight less than max(T)
            i. Add the new edge into the MST
        b. Else
            i. Do not add it to the MST
```

```
b. 1. Run BFS on MST to find the maximum edge weight
   2. Compare the edge weight with the edge weight in the graph on the optimized tree
   3. If edge tree is smaller than edge weight in the graph
       a. Tree is optimized
   4. Else
       a. Replace edge in tree with edge in graph to create new MST
```

```
c. 1. BFS on graph to find the minimum edge weight
   2. Compare edge weight with edge weight of the increased tree
   3. If edge of the tree is smaller than the edge weight of graph
       a. The tree is optimized
   4. Else
       a. Replace edge in tree with edge from the graph
```

*all of these will have a runtime of O(V)

6) For the graph below, complete the following for a shortest path from A to B:

a. Bellman Ford Algorithm

A-1, A-2, 1-3, 1-4, 2-3, 2-5, 3-6, 4-6, 4-8, 5-6, 5-7, 5-B, 6-7, 6-8, 7-8, 7-B, 8-B

Iteration 1:

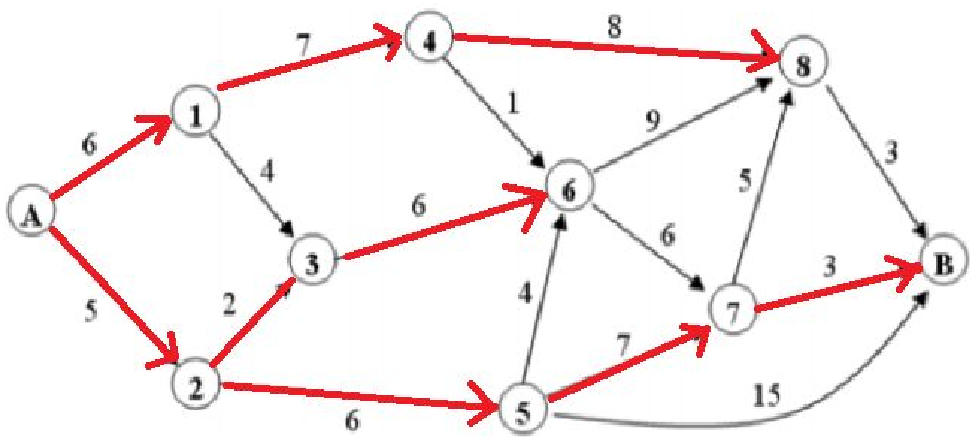
V	A	1	2	3	4	5	6	7	8	B
distTo[]	0	6	5	7	13	11	13	18	21	21
edgeTo[]	-	A-1	A-2	2-3	1-4	2-5	3-6	5-7	4-8	7-B

Iteration 2:

V	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

A-1, A-2, 1-3, 1-4, 2-3, 2-5, 3-6, 4-6, 4-8, 5-6, 5-7, 5-B, 6-7, 6-8, 7-8, 7-B, 8-B
Since iteration 2 shows no change, this is optimal. STOP



b. Dijkstra's Algorithm

V	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
relaxCount	0	1	1	1	1	1	1	1	1	2

c. Topological Sort Algorithm

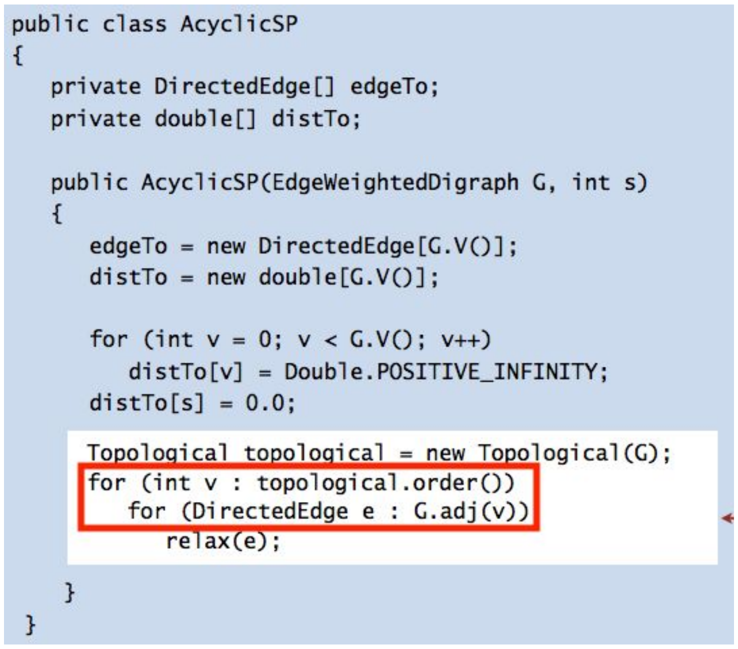
V	A	1	2	3	4	5	6	7	8	B
distTo[]	0	6	5	7	13	11	13	18	21	21
edgeTo[]	-	A-1	A-2	2-3	1-4	2-5	3-6	5-7	4-8	7-B
relaxCount	0	1	1	2	1	1	1	1	1	2

A,1,2,3,4,5,6,7,8,B

7) Explain why the Topological Sort Algorithm requires a DAG. What happens when you run the algorithm on a non-DAG?

In order to do topological sort we need to make sure that the graph is first acyclic. If it is not then we cannot do this. For example if we have a cyclic directed graph going from A -> B -> C -> A…. and so on. We cannot satisfy the topological sort property. In topological property we would say that A has to before B and B has to be before C. However, has a cycle we cannot satisfy this statement since This will repeat again and A will be after B and B is after C. This case will happen too if the graph was undirected. This is because now we can go backwards. Therefore it does not satisfy topological property.

Code:



```
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);

    }
 }
```

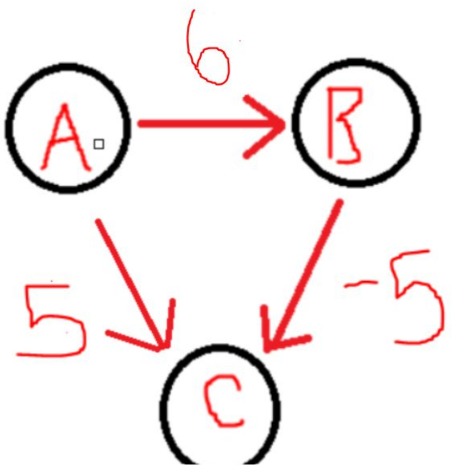
In this code the error is that for a non-DAG, the topological order is affected. When the order is created, we need to maintain that order. But when the graph is undirected this will not happen. The DirectedEdge will cause an error because the edges are no longer directed. This will mean the algorithm is allowed to go backwards and forward. This will cause an error in topological order.

8) . Why do we not implement a cycle detection for Djikstra's Algorithm, despite the fact that Prim's Algorithm for the MST -- which is nearly the same as Djikstra's – does?

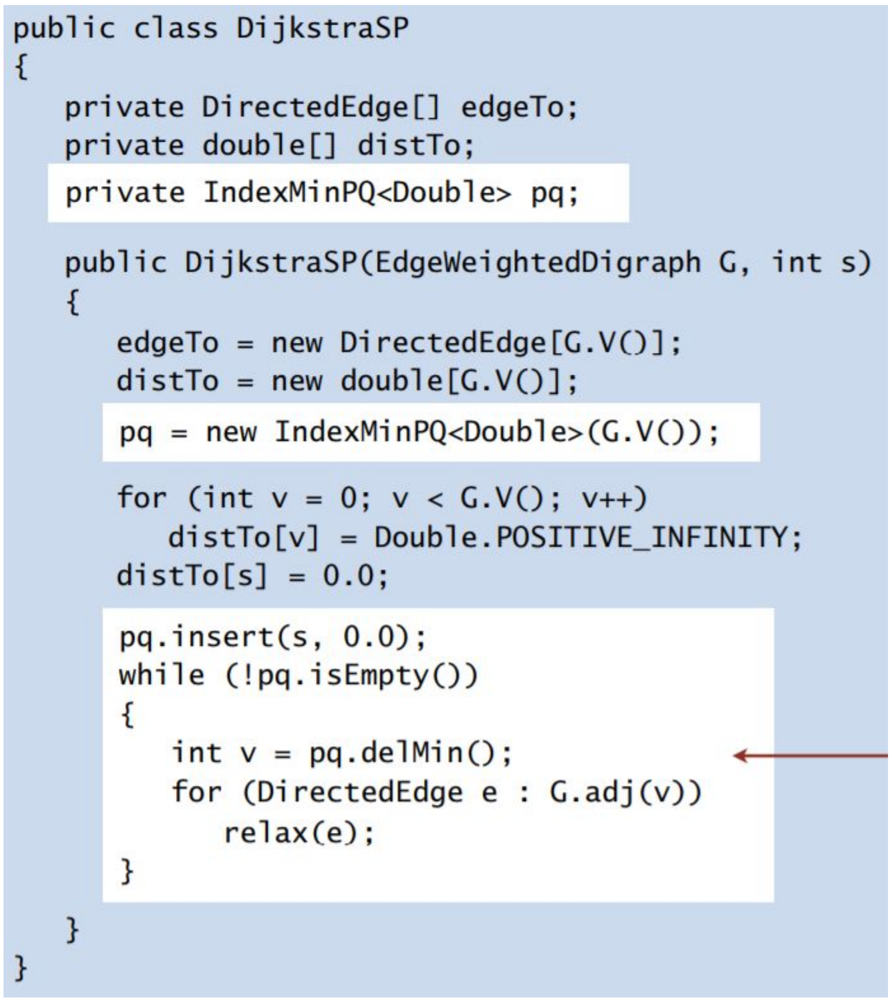
In Prim's algorithm its looking at the overall all tree. We start at one vertex and go from there. We will go in order of the edge weight of the whole tree. As Djikstra's looks only at its vertices based on the edge weights from the vertex. In other words this only looks at the adjacent vertex from the source. Because of this Prim can also looks at the tree expanding it will go backwards looking at the tree. This means we need to determine if the cycle will happen, if we do not use cycle detection then creating a cycle might happen. As Djikstra's always chooses the shorter of the possible paths from the vertex. Going backwards is never an option, therefore there cannot be a possibility of cycle detection.

9) Why does Djikstra's Algorithm fail for negative edge weights? Reproduce the primary code and highlight what causes the error.

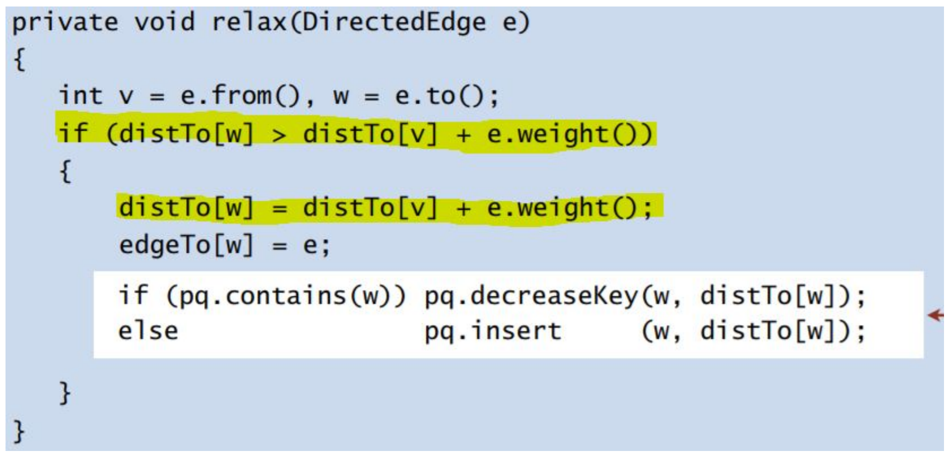
Negative edge weights fail because it needs to take the minimum edge weight first. But since there are negative edge weight it will cause errors. This is due to the fact that in Djikstra when all weights are non-negative, adding an edge can never make a path shorter. However for example:



In this example we would choose A as the source. We have two options go from A-B or A-C. This case it will go A-C since 5 is less than 6. Therefore it will determine that A-C is the shortest distance. However, if it went 6 then the C-B is negative 5 therefore the distance from A to C is 1 instead of 5. This will give us the actual shortest distance.



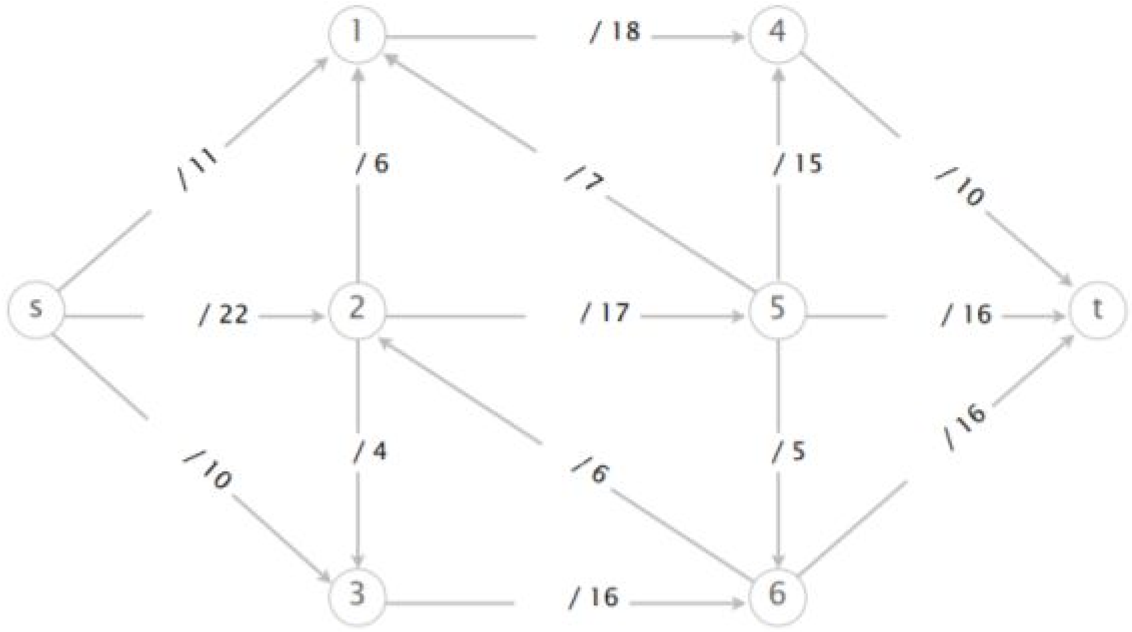
```java
public class DijkstraSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   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]);
   }
}
```

The highlighted part of the code is what causes an error because this will always check for the smallest weight and adds the new weight. However the error is that if the edge weight is only looked adjacent from the current vertex. However negative numbers in the system would lead this to be false. Therefore, if we were presented negative edge weights, we can't assume that all new edges will increase the total shortest path.

10) Given the graph below, do the following:



a. Ford Fulkerson Algorithm

Iteration 1: s -> 1 -> 4 -> t

Vertex	S	1	4	t
Flow/Capacity	-	10/11	10/18	10/10

t1 = 10

Iteration 2: s -> 3 -> 6 -> t

Vertex	S	3	6	t
Flow/Capacity	-	10/10	10/16	10/16

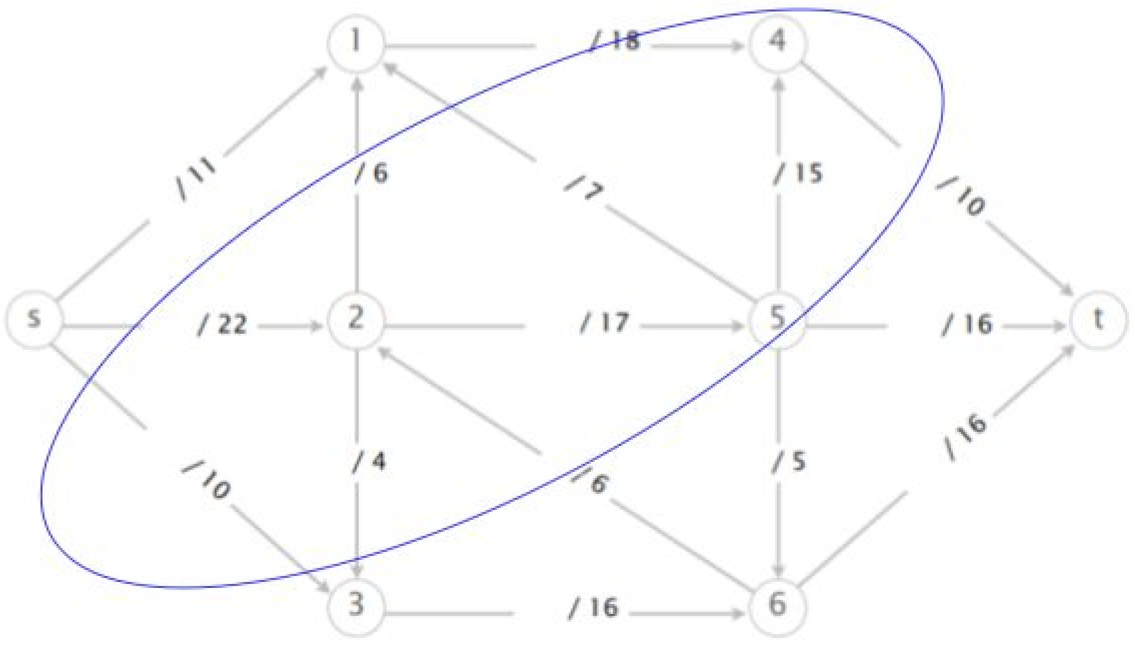
t1 = 20

Iteration 3: s -> 2 -> 3 -> 6 -> t ***2 diverges into 3 and 5**
-> 5 -> t

Vertex	S	2	3	5	6(new)	t(new)	t(from 5)
Flow/Capacity	-	21/22	4/4	17/17	15/16	14/16	16/16

t1 = 41

b. What is the mincut of the above graph?



The mincut of this graph is going to contain 5-3, 2-3, 4-t, and 2-5

c. What is the capacity of the mincut?

Max flow = mincut capacity = 41.