Thomas Killeen

C16394453

DT 228 2

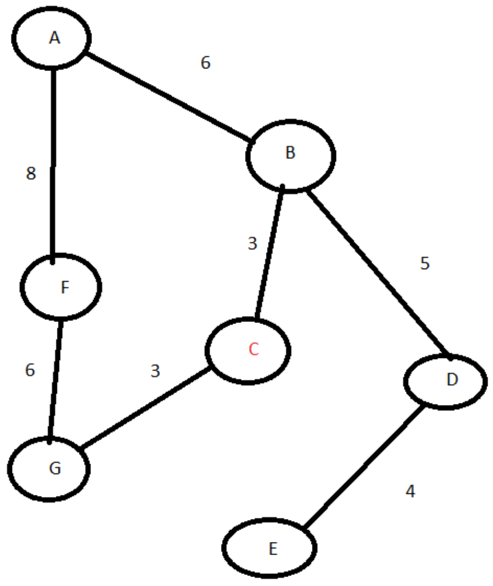
Algorithms and Data Structures

Introduction

This is my report for the assignment on Prims List and Kruskals Trees. I worked on this a while and the idea is that for Prims list you use it to find the minimum spanning tree for a connected weighted undirected graph. You have to find the subsets of the edges that include every vertex that forms a tree, where the weight of the edges in the tree is minimized. The algorithm increases the size of the tree, one edge at a time, starting with a single vertex in a tree until it spans all vertices.

For Kruskals algorithm is the ability to find the minimum spanning tree for a connected weighted graph. Like before you find all subsets of the edges that include every vertex. If it’s not connected it finds the minimum spanning forest.

Graph



<u>Minimum spanning tree</u>

Adjacency List

Start at C

C	3	3
G	6	3
F	8	6
A	6	8
B	5	6

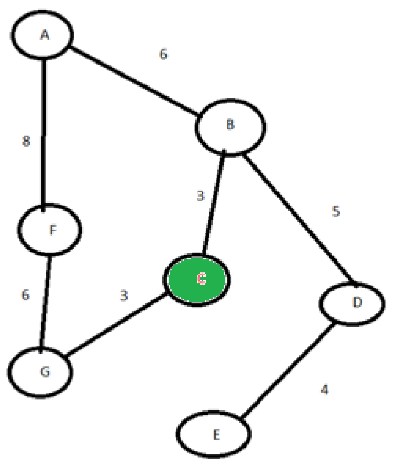
Started off by creating the graph. From creating the graph I could see that positon C would be the start of the spanning tree. Start at position C and move to G, then move to F, then A, then B, then D, than finally position E.

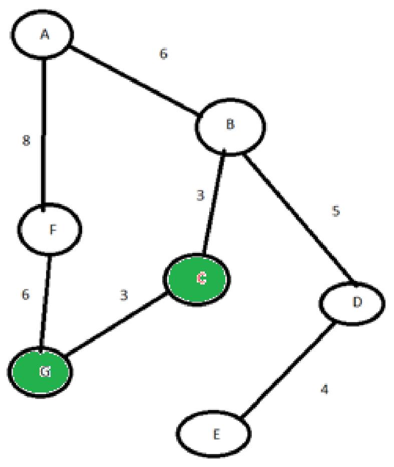
In the code this is represented by:

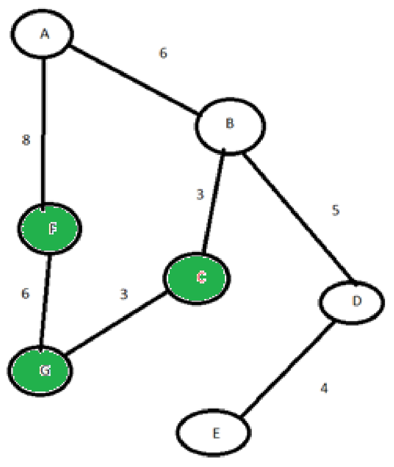
Weight	Source	Destination
3	2(C)	3(G)
6	3(G)	4(F)
8	4(F)	5(A)
6	5(A)	1(B)
5	1(B)	6(D)
4	6(D)	7(E)

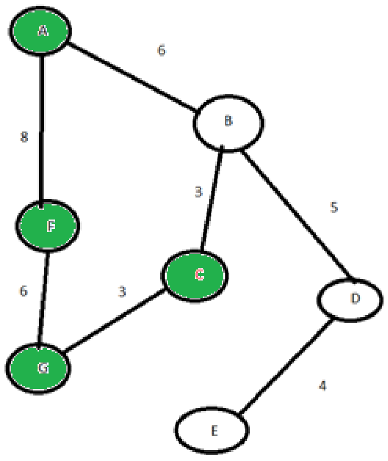
For Prim's Algorithm with my MST starts by reading the file, then it initialises the first vertex for the starting Node in this case C. It repeats the process finding the initial weight that would be most efficient and moves, so it would move to position G. So in this case it choses an edge of low weight and where the new vertex doesn't equal the old one, but there could be multiple weights of the same but it will choose any of them. The vertex is then added to the new vertex and is repeated. Then it keeps repeating until new vertex and the new edge describe the MST.

For Kruskal's, in the case of the Union Find Set keeps track of the elements in a disjointed subset, which will find the subsets, so the elements positions in the graph and the join the two subsets into a single subset. Kruskal's in this case finds the subsets of the edges and forms a tree that includes the vertexs. In this case that they aren't connected it will create an MSforest. In the case of my graph it will create a set of trees for the forest, where the vertexs are the separate trees. It then creates a set which contains all the edges in the graph. While the set isn't empty the tree isn't spanning so it removes an edge with the lowest weight from the set. If the edge connects to two different trees then its added to the forest combining them into a single tree, else it discards the edge.

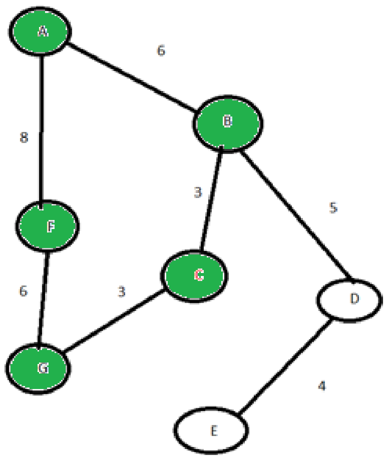

A
6
8
B
3
5
F
C
6
3
D
G
4
E


A
6
8
B
3
5
F
C
6
3
D
G
4
E

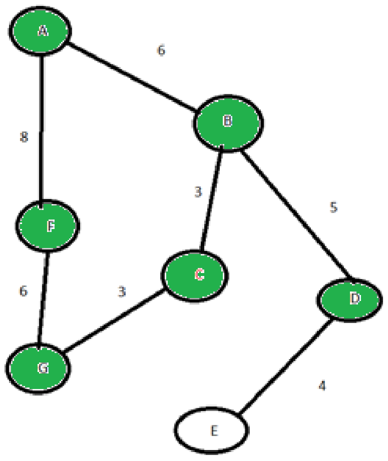

A
6
8
B
3
5
F
C
6
3
D
G
4
E



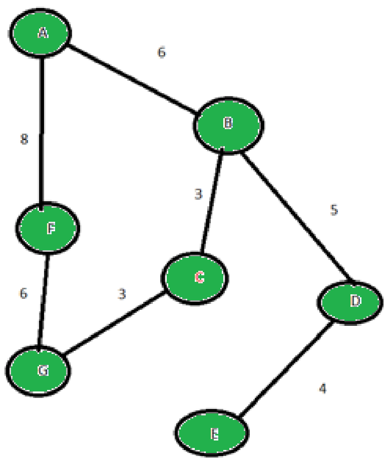
A
B
C
D
E
F
G
6
8
3
5
6
3
4



A
B
C
D
E
F
G
6
8
3
5
6
3
4



A
B
C
D
E
F
G
6
8
3
5
6
3
4


A
6
B
8
3
5
F
C
6
3
D
G
4
E

Code

Primm's List

```
// Simple weighted graph representation

// Uses an Adjacency Linked Lists, suitable for sparse graphs


import java.io.*;


class Heap

{

   private int[] h;     // heap array

   private int[] hPos;            // hPos[h[k]] == k

   private int[] dist;    // dist[v] = priority of v


   private int N;        // heap size


   // The heap constructor gets passed from the Graph:

   //    1. maximum heap size

   //    2. reference to the dist[] array

   //    3. reference to the hPos[] array

   public Heap(int maxSize, int[] _dist, int[] _hPos)

   {

      N = 0;

      h = new int[maxSize + 1];

      dist = _dist;

      hPos = _hPos;

   }



   public boolean isEmpty()
```

```
{
  return N == 0;
}



public void siftUp( int k)
{
  int v = h[k];
              h[0] = 0;
  h[0] = Integer.MAX_VALUE;

              //The vertex is using the dist to move up the heap
  while(dist[v] > h[k/2])
  {
    h[k] = h[k/2];//Parent vertex is given the position of the child vertex
                      hPos[h[k]] = k;//The addition of the hPos
    k=k/2;//The childs index is the last parents position
  }
  h[k] = v;//The pos of the vertex is added to the heap
              hPos[v] = k;//Index of pos updated in hPos

  // code yourself--
  // must use hPos[] and dist[] arrays
}

      //Removing the vertex at the top of the heap
      //resizes heap
public void siftDown( int k)
{
    int v, j;


    v = h[k];
    while(k <= N/2)
    {
        j = 2k;
        if(j>N && dist[h[j]] > dist[h[j+1]])//If the node is > left child
        {
            ++j;//Increment child
        }
        if(dist[v] <= dist[h[j]])//If the parent is greater than the child
        {
            break;//Stop the system
        }
        h[k] = h[j];//If the parent is greater than the child, child is given parent pos
                                    hPos[h[k]] = k;//Update pos of last child
        k=j;//Assign a new vertex pos
    }
    h[k] = v; //assign vertex for heap
    hPos[v] = k;//Update the vertex in hPos


    // code yourself --
    // must use hPos[] and dist[] arrays
}


        //Assigns a new vertex to the end of the heap and passes it to siftUp
public void insert( int x)
{
    h[++N] = x;
```

```
        siftUp( N);
    }


            //Last node is moved to the top and passed into siftDown
    public int remove()
    {
        int v = h[1];
        hPos[v] = 0; // v is no longer in heap
        h[N+1] = 0;  // put null node into empty spot


        h[1] = h[N--];
        siftDown(1);


        return v;
    }


}


class Graph {
    class Node {
        public int vert;
        public int wgt;
        public Node next;
    }


    // V = number of vertices
    // E = number of edges
    // adj[] is the adjacency lists array
    private int V, E;
private Node[] adj;

private Node z;

private int[] mst;


// used for traversing graph

private int[] visited;

private int id;




// default constructor

public Graph(String graphFile)  throws IOException

{

   int u, v;

   int e, wgt;

   Node t;


                    //Majority of this is used to read a file through input and handles issues like white spaces

   FileReader fr = new FileReader(graphFile);

                    BufferedReader reader = new BufferedReader(fr);


   String splits = " +";  // multiple whitespace as delimiter

                    String line = reader.readLine();

   String[] parts = line.split(splits);

   System.out.println("Parts[] = " + parts[0] + " " + parts[1]);


   V = Integer.parseInt(parts[0]);

   E = Integer.parseInt(parts[1]);


   // create sentinel node
```

```
z = new Node();

z.next = z;


// create adjacency lists, initialised to sentinel node z

adj = new Node[V+1];

for(v = 1; v <= V; ++v)

    adj[v] = z;


// read the edges

System.out.println("Reading edges from text file");

for(e = 1; e <= E; ++e)

{

    line = reader.readLine();

    parts = line.split(splits);

    u = Integer.parseInt(parts[0]);

    v = Integer.parseInt(parts[1]);

    wgt = Integer.parseInt(parts[2]);


    System.out.println("Edge " + toChar(u) + "--(" + wgt + ")--" + toChar(v));


    // write code to put edge into adjacency matrix

    t = new Node();

                                    t.vert = v;

                                    t.wgt = wgt;

                                    t.next = adj[u];

                                    adj[u] = t;


                                    t = new Node();

                                    t.vert = u;
```

```
                    t.wgt = wgt;

                    t.next = adj[v];

                    adj[v] = t;

  }

}


// convert vertex into char for pretty printing

private char toChar(int u)

{

  return (char)(u + 64);

}


// method to display the graph representation

public void display() {

  int v;

  Node n;


  for(v=1; v<=V; ++v){

    System.out.print("\nadj[" + toChar(v) + "] ->" );

    for(n = adj[v]; n != z; n = n.next)

      System.out.print(" |" + toChar(n.vert) + " | " + n.wgt + "| ->");

  }

  System.out.println("");

}




      public void MST_Prim(int s)

      {
```

```
int v, u;

int wgt, wgt_sum = 0;

int[]  dist, parent, hPos;

Node t;


//code here--


dist = new int[v+1];

            parent = new int[v+1];

            hPos = new int[v+1];


            for(v = 0; v <= v; ++v)

            {

                    dist[v] = Integer.MAX_VALUE;

                    parent[v] = 0;

                    hPos[v] = 0;

            }


            dist[s] = 0;




Heap pq =  new Heap(V, dist, hPos);

pq.insert(s);


while (pq.isEmpty != 0)

{

  // most of alg here

  v = pq.remove();

                    wgt_sum += dist[v];//Add the dist and wgt of vert to the spanning tree
                    dist[v] = -dist[v];//Its finished so make it minus


                    for( t = adj[v]; t != z; t = t.next)

                    {

                            u = t.vert;

                            wgt = t.wgt;

                            if(wgt < dist[u])//If the weight is less than the value u

                            {

                                    dist[u] = wgt;

                                    parent[u] = v;

                                    if(hPos[u] == 0)//If its not in insert

                                    {

                                            pq.insert(u);

                                    }

                                    else//If in heap modify

                                    {

                                            pq.siftUp(hPos[u]);

                                    }

                            }

                    }

  }

  System.out.print("\n\nWeight of MST = " + wgt_sum + "\n");


  mst = parent;

      }


public void showMST()

{

    System.out.print("\n\nMinimum Spanning tree parent array is:\n");
```

```
            for(int v = 1; v <= V; ++v)

                System.out.println(toChar(v) + " -> " + toChar(mst[v]));

            System.out.println("");

    }


}


public class PrimLists {

    public static void main(String[] args) throws IOException

    {

        int s = 2;

        String fname = "wGraph3.txt";


        Graph g = new Graph(fname);


        g.display();




    }




}
```

Kruskal Trees

```
// Kruskal's Minimum Spanning Tree Algorithm

// Union-find implemented using disjoint set trees without compression


import java.io.*;


class Edge {

   public int u, v, wgt;


   public Edge() {

      u = 0;

      v = 0;

      wgt = 0;

   }


   public Edge( int x, int y, int w) {

      // missing lines

                        this.u = x;

                        this.v = y;

                        this.wgt = w;

   }


   public void show() {

      System.out.print("Edge " + toChar(u) + "--" + wgt + "--" + toChar(v) + "\n") ;

   }


   // convert vertex into char for pretty printing

   private char toChar(int u)

   {
        return (char)(u + 64);
    }
}


class Heap
{
                private int[] h;
    int N, Nmax;
    Edge[] edge;


    // Bottom up heap construc
    public Heap(int _N, Edge[] _edge) {
        int i;
        Nmax = N = _N;
        h = new int[N+1];
        edge = _edge;

        // initially just fill heap array with
        // indices of edge[] array.
        for (i=0; i <= N; ++i)
            h[i] = i;

        // Then convert h[] into a heap
        // from the bottom up.
        for(i = N/2; i > 0; --i)
           siftDown(k);// missing line;-- We need to call siftDown
    }
```

```
         //Used to remove
private void siftDown( int k) {
   int e, j;

   e = h[k];
   while( k <= N/2) {
      // missing lines--
                              while(j<=N-1) //While child is less than the Node
                              {
                                    if(j<N-1 && a[j]<a[j+1])//If child is less than the node and parent
                                    {
                                          ++j;//Increment child
                                    }
                                    if(v >= a[j])//If greater than
                                    {
                                          break;//End the program
                                    }
                                    a[k] = a[j];//If the parent is greater than the child, child get parents pos
                                    k=j;//Update the child
                                    j=2k+1;
                              }
                              a[k] = v;//Assign vertex
   }
   h[k] = e;//Update vertex
```

```
    }



    public int remove() {
        h[0] = h[1];
        h[1] = h[N--];
        siftDown(1);
        return h[0];
    }
}

/****************************************************
*
*      UnionFind partition to support union-find operations
*      Implemented simply using Discrete Set Trees
*
*****************************************************/

class UnionFindSets
{
    private int[] treeParent;
    private int N;

    public UnionFindSets( int V)
    {
        N = V;
        treeParent = new int[V+1];
        // missing lines--
                        for(int i = 1; i <= N; i++)
                        {
                                     treeParent[i] = i;
                        }
}

public int findSet( int vertex)
{
   // missing lines--
                        while(vertex != treeParent[vertex])//If the parent of the vertex is not equal to vertex
                        {
                                     vertex = treeParent[vertex];//The parents vertex equals the fn value
                        }
   return vertex;//Return the parents vertex
}

public void union( int set1, int set2)
{
   // missing--
                        //We let the the roots equal to the sets
                        int root1 = findSet(set1);
                        int root2 = findSet(set2);
                        treeParent[root2] = root1;
}

public void showTrees()
{
                        //Used to disp;ay the tree
   int i;
   for(i=1; i<=N; ++i)
```

```
        System.out.print(toChar(i) + "->" + toChar(treeParent[i]) + "  " );
    System.out.print("\n");
}

            //Used to show the roots of the sets
public void showSets()
{
    int u, root;
    int[] shown = new int[N+1];
    for (u=1; u<=N; ++u)
    {
        root = findSet(u);
        if(shown[root] != 1) {
            showSet(root);
            shown[root] = 1;
        }
    }
    System.out.print("\n");
}

            //Shows the sets
private void showSet(int root)
{
    int v;
    System.out.print("Set{");
    for(v=1; v<=N; ++v)
        if(findSet(v) == root)
            System.out.print(toChar(v) + " ");
    System.out.print("}  ");
    }


    private char toChar(int u)

    {

        return (char)(u + 64);

    }

}


class Graph

{

    private int V, E;

    private Edge[] edge;

    private Edge[] mst;


                //This sequence lets the program read in a file and create the graph

    public Graph(String graphFile) throws IOException

    {

        int u, v;

        int w, e;


        FileReader fr = new FileReader(graphFile);

                        BufferedReader reader = new BufferedReader(fr);


        String splits = " +";  // multiple whitespace as delimiter

                        String line = reader.readLine();//Reads in the lines

        String[] parts = line.split(splits);//Splits the lines

        System.out.println("Parts[] = " + parts[0] + " " + parts[1]);
        V = Integer.parseInt(parts[0]);

        E = Integer.parseInt(parts[1]);


        // create edge array

        edge = new Edge[E+1];


       // read the edges

        System.out.println("Reading edges from text file");

        for(e = 1; e <= E; ++e)

        {

            line = reader.readLine();

            parts = line.split(splits);

            u = Integer.parseInt(parts[0]);

            v = Integer.parseInt(parts[1]);

            w = Integer.parseInt(parts[2]);


            System.out.println("Edge " + toChar(u) + "--(" + w + ")--" + toChar(v));


            // create Edge object

                                    edge[e] = new Edge(u,v,w);

        }

    }



/*************************************************************

*

*      Kruskal's minimum spanning tree algorithm

*

*************************************************************/
```

```
public Edge[] MST_Kruskal()
{
   int ei, i = 0;
   Edge e;
   int uSet, vSet;
   UnionFindSets partition;

   // create edge array to store MST
   // Initially it has no edges.
   mst = new Edge[V-1];

   // priority queue for indices of array of edges
   Heap h = new Heap(E, edge);

   // create partition of singleton sets for the vertices
   partition = new UnionFindSets(V);
            partition.showSets();

            while(i <  V-1)
            {
                        ei = h.remove();
                        e = edge[ei];

                        uSet = partition.findSet(e.u);
                        vSet = partition.findSet(e.v);

                        if(uSet != vSet)
                        {
                                    mst[i] = e;
```

```
                                        ++i;


                                        e.show();


                                        partition.union(uSet, vSet);//Sends the vertexes to the union class if union set is not
equal to vertex set

                                        partition.showSets();

                                        partition.showTrees();


                                        totalWeight += e.wgt;

                                }

                }


   System.out.println("Weight " + totalWeight);




   return mst;

}




   // convert vertex into char for pretty printing

   private char toChar(int u)

   {

       return (char)(u + 64);

   }


   public void showMST()

   {

                                //Shows the minimum spanning tree
```

```
        System.out.print("\nMinimum spanning tree build from following edges:\n");
        for(int e = 0; e < V-1; ++e) {
            mst[e].show();
        }
        System.out.println();

    }

} // end of Graph class

    // test code
class KruskalTrees {
    public static void main(String[] args) throws IOException
    {
        String fname;
                              BufferedReader input1 = new BufferedReader(new inputStreamReader(system.in));
        System.out.print("\nInput name of file with graph definition: ");
        fname = input1.ReadLine();

        Graph g = new Graph(fname);

        g.MST_Kruskal();

        g.showMST();

    }
}
```