Algorithms Assignment Report

Minimum Spanning Tree using Prim and Kruskal Algorithm

Due Date: 27/04/2018

Date Completed: 17/04/2018

Completed by:

William Carey

C16315253

DT228/2

Contents

Introduction:

We will be looking at Graphs and how to work with them. A Graph has multiple edges and vertices, where the numbers of edges are usually greater than the number of vertices. Each vertex is connected to another by the edges in between each one. A combination of any number of each

would result in a graph. There are different types of graphs and different ways to approach them. There are incomplete graphs and complete graphs. The difference between the two is a complete graph has all the vertices connected to each other one way or another, which leaves no vertex hanging on its own. An incomplete graph is different in the sense that there is at least one vertex that is only connected to one other vertex itself.

We will create a subset of one graph in this report. This is known as a tree. Specifically, we will be looking at the smallest possible tree to be made from the graph. This is known as the minimum spanning tree and takes into the account making a tree that connects all the vertices with the smallest weight possible. To achieve this, we will implement and document through two algorithms. One is known as the Kruskal algorithm while the other is the Prim Algorithm. These approaches do similar approach as they both create a minimum spanning tree. However, the approaches are different.

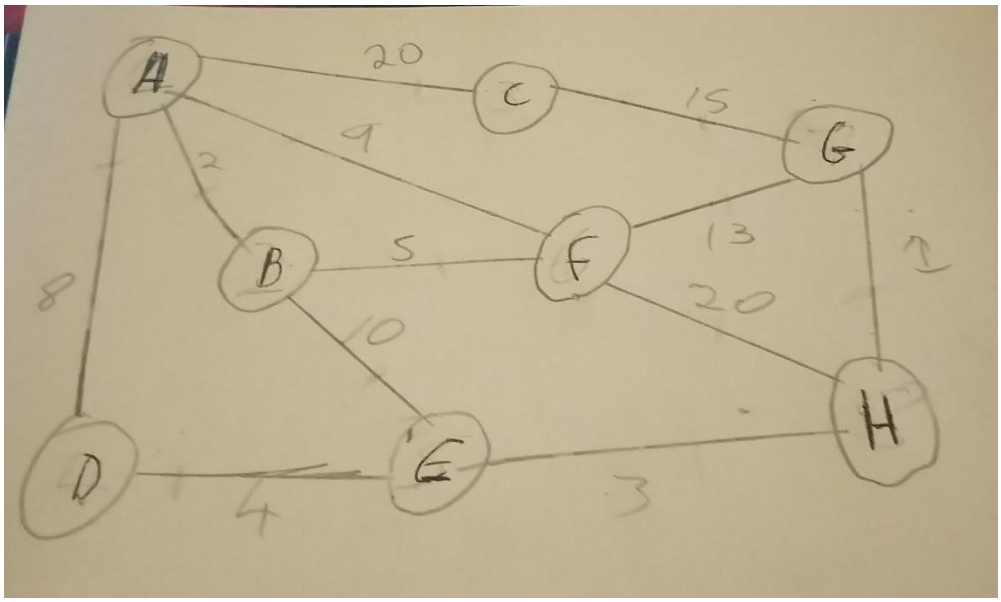
Prim takes in a starting point and works its way around the tree by connecting all the vertices. It checks all the possible routes and takes the shortest route. Lets say we have vertex A. A is connected to B, C and D. The distance between A and these vertices as follows are: to B is 5, C is 3 and D is 7. What Prim will do is go to C as that is the shortest path. They continue to connect until all the vertices are connected creating the minimum spanning tree.

Kruskal is different. Instead of reading vertices, Kruskal reads the edges in between each vertex and connects them based off the smallest edges. Taking the above example and extending the information so B connects to C with weight of 7, to D with a weight of 2 and C connects to D with a weight of 10. Kruskal connects B to D as it is the smallest, then A to C as it is the second smallest. If both was the same size it would not matter which one was picked first as them both will be connected in the minimum spanning tree.

Both will be assisted by sorting methods to assist in where they are positioned and the combining of them. For example, Kruskal will be assisted by Union by rank, Set Representation and a heap in this report, where Prim will be assisted by the heap, distance and keeping track of its parent as well as its heap position. These mechanisms will make each running of the algorithm within a java programme more efficient and smooth out the layout. Although both heaps are different in their content, they are both priority heaps in their layout.

myGraph.txt:

Diagram:



Contents:

myGraph.txt

```
8 12
1 2 2
1 3 20
1 4 8
1 6 9
2 5 10
2 6 5
3 7 15
4 5 4
5 8 3
6 7 13
6 8 20
7 8 1
```

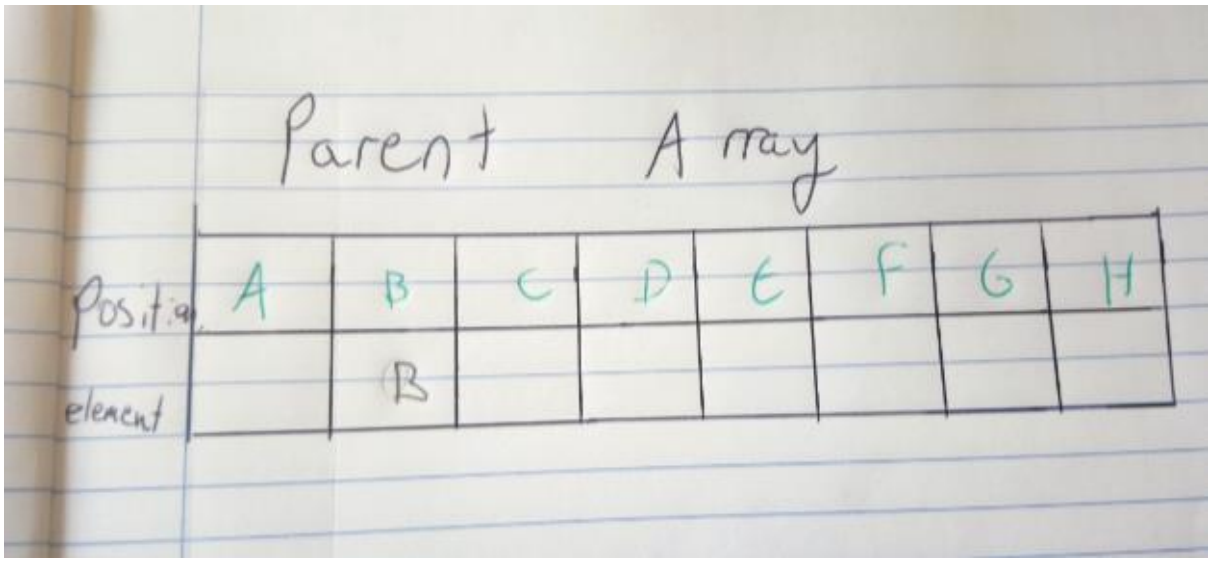
Adjacency List Representation:

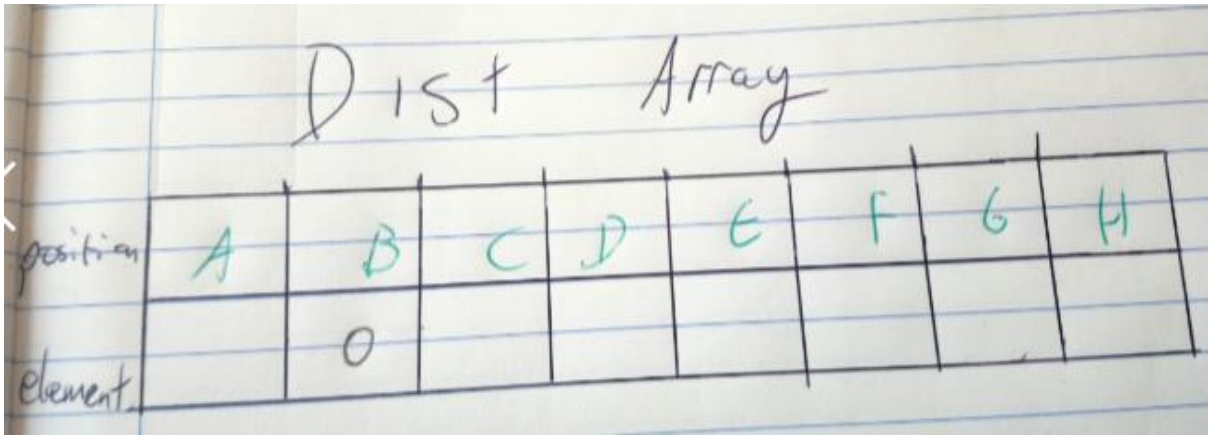
```
adj[A] -> |F | 9| -> |D | 8| -> |C | 20| -> |B | 2| -> NULL
adj[B] -> |F | 5| -> |E | 10| -> |A | 2| -> NULL
adj[C] -> |G | 15| -> |A | 20| -> NULL
adj[D] -> |E | 4| -> |A | 8| -> NULL
adj[E] -> |H | 3| -> |D | 4| -> |B | 10| -> NULL
adj[F] -> |H | 20| -> |G | 13| -> |B | 5| -> |A | 9| -> NULL
adj[G] -> |H | 1| -> |F | 13| -> |C | 15| -> NULL
adj[H] -> |G | 1| -> |F | 20| -> |E | 3| -> NULL
```

Step by Step Construction of the MST on myGraph.txt:

Including parent[] and Dist[] with Prim Algorithm:

```
Current Vertex is B with  itself as the Orgin point
Current Vertex is F with parent value: B  and the distance between the two vertices is 5
Current Vertex is E with parent value: B  and the distance between the two vertices is 10
Current Vertex is A with parent value: B  and the distance between the two vertices is 2
Current Vertex is D with parent value: A  and the distance between the two vertices is 8
Current Vertex is C with parent value: A  and the distance between the two vertices is 20
Current Vertex is E with parent value: D  and the distance between the two vertices is 4
Current Vertex is H with parent value: E  and the distance between the two vertices is 3
Current Vertex is G with parent value: H  and the distance between the two vertices is 1
Current Vertex is C with parent value: G  and the distance between the two vertices is 15
```





Parent Array

position	A	B	C	D	E	F	G	H
element	B	B						

Dist Array

position	A	B	C	D	E	F	G	H
element	2	0						

Parent Array

position	A	B	C	D	E	F	G	H
element	B	B				B		

Dist Array

position	A	B	C	D	E	F	G	H
element	2	0				5		

Parent Array

Position	A	B	C	D	E	F	G	H
element	B	B		A		B		

Dist Array

position	A	B	C	D	E	F	G	H
element	2	0		8		5		

Parent Array

Position	A	B	C	D	E	F	G	H
element	B	B		A	D	B		

Dist Array

position	A	B	C	D	E	F	G	H
element	2	0		8	4	5		

Parent Array

Position	A	B	C	D	E	F	G	H
element	B	B		A	D	B		C

Dist Array

position	A	B	C	D	E	F	G	H
element	2	0		8	4	5		3

Parent Array

Position	A	B	C	D	E	F	G	H
element	B	B		A	D	B	H	C

Dist Array

position	A	B	C	D	E	F	G	H
element	2	0		8	4	5	1	3

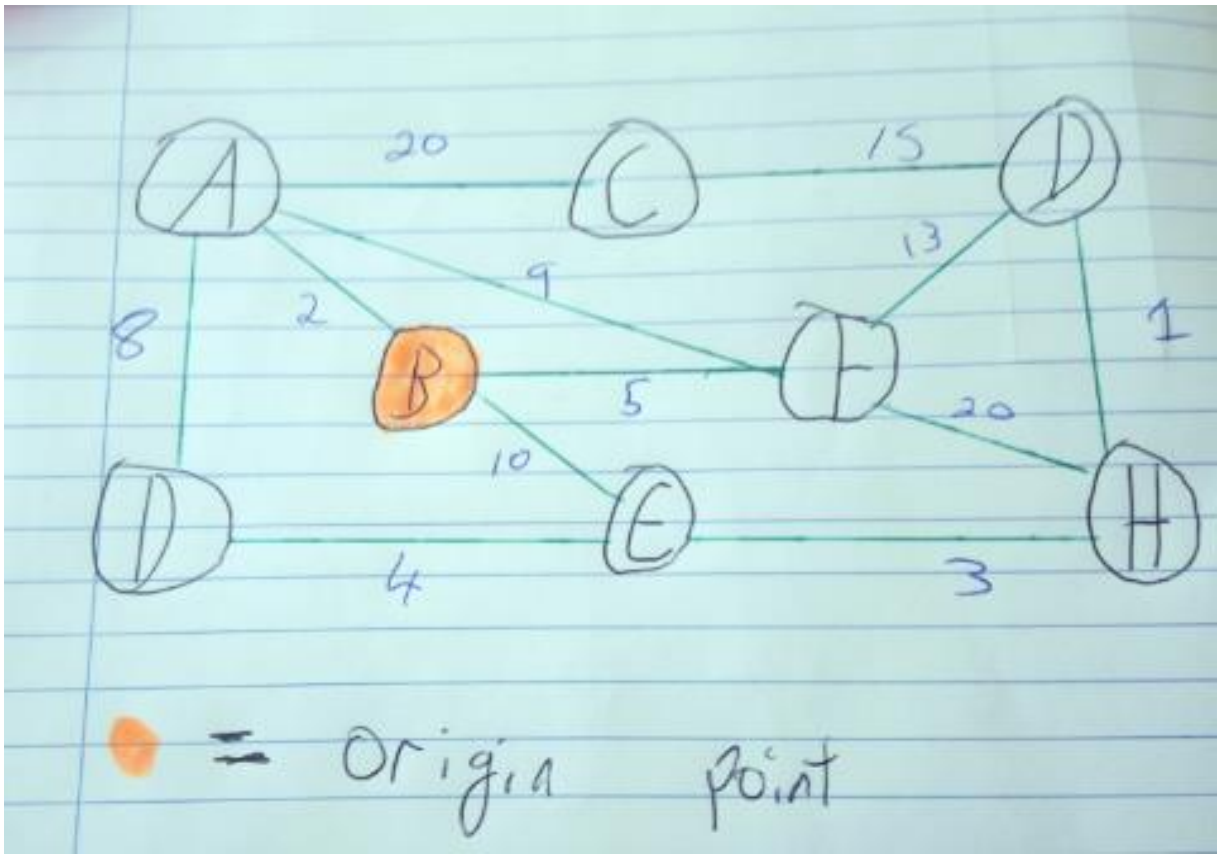
Parent Array

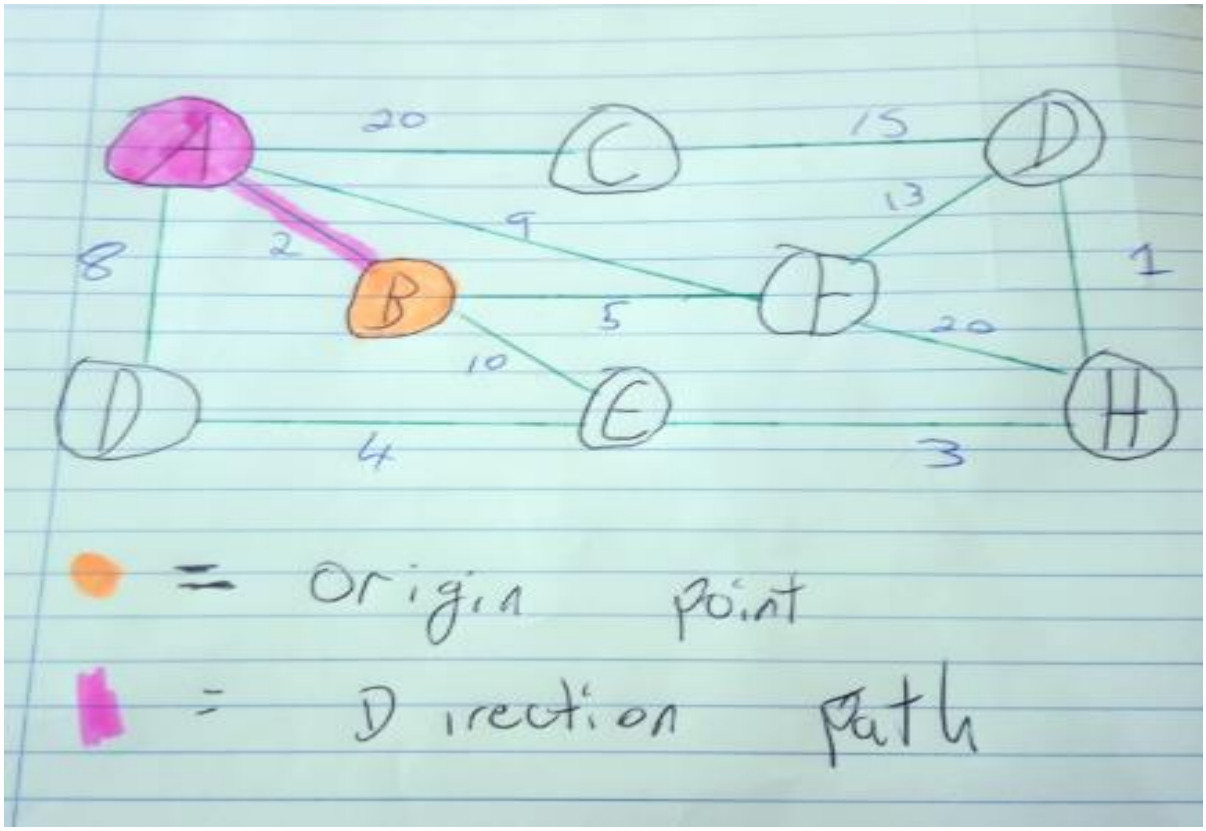
position	A	B	C	D	E	F	G	H
element	B	B	G	A	D	B	H	E

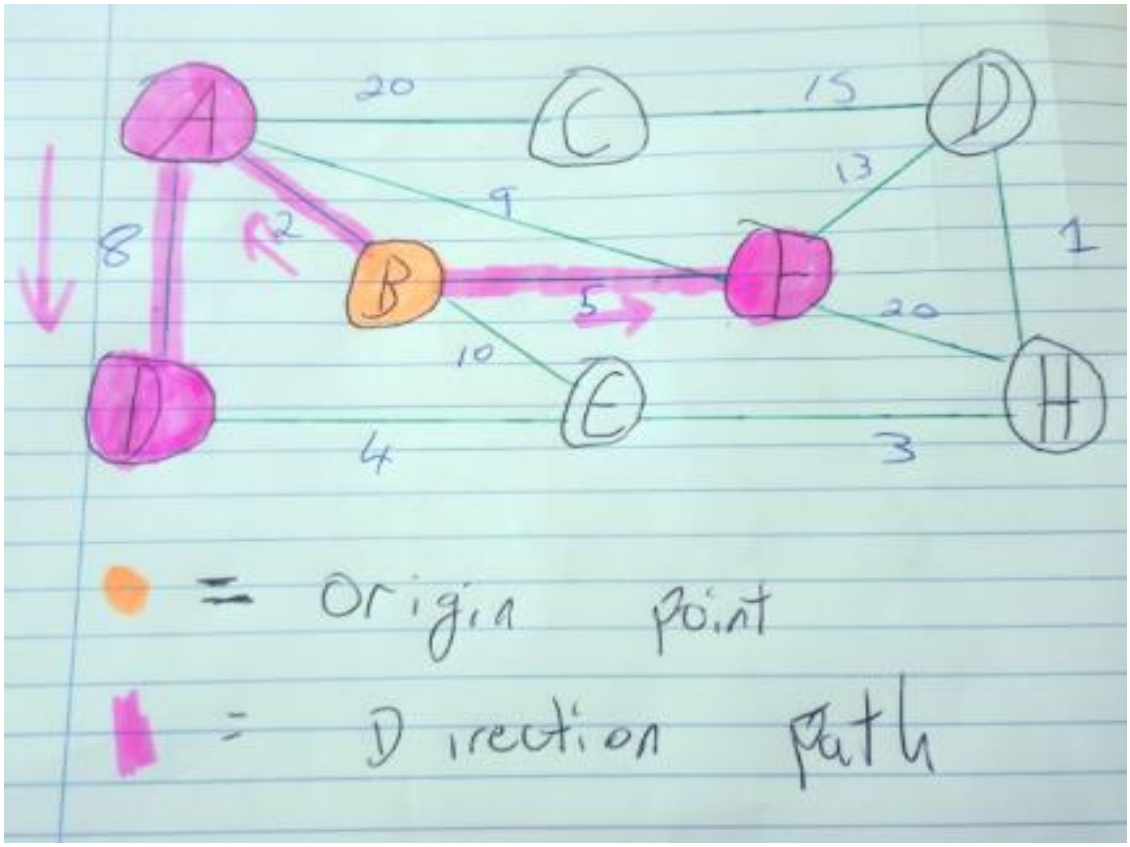
Dist Array

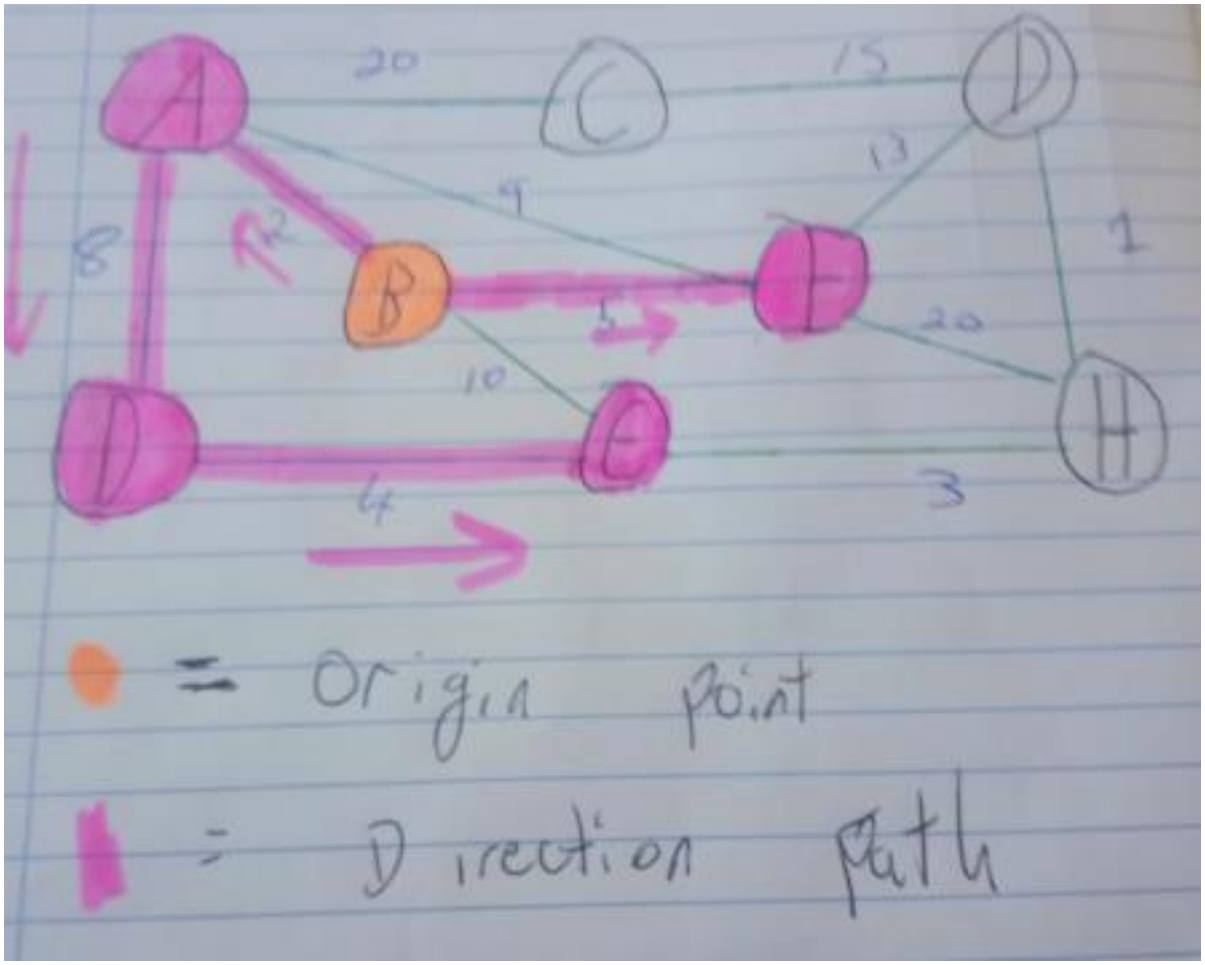
position	A	B	C	D	E	F	G	H
element	2	0	15	8	4	5	1	3

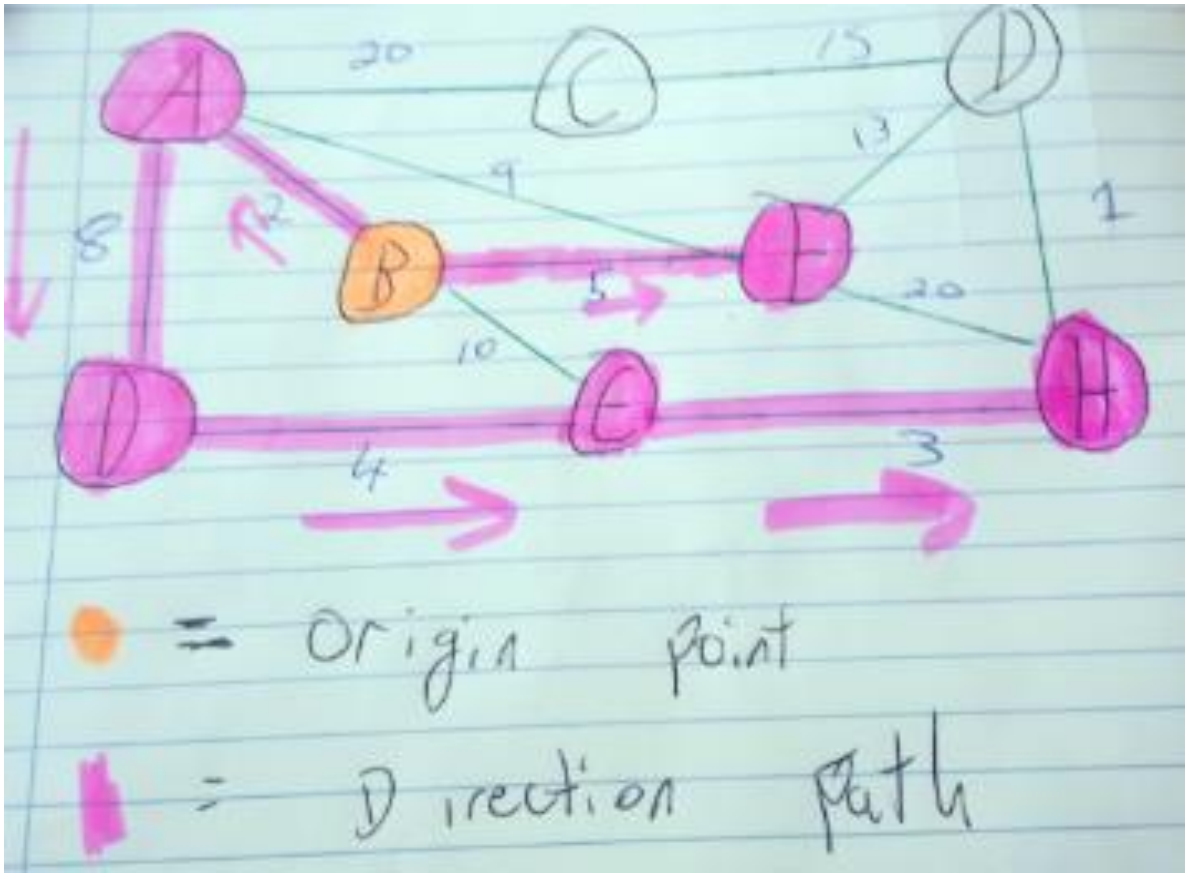
Diagram Representation:

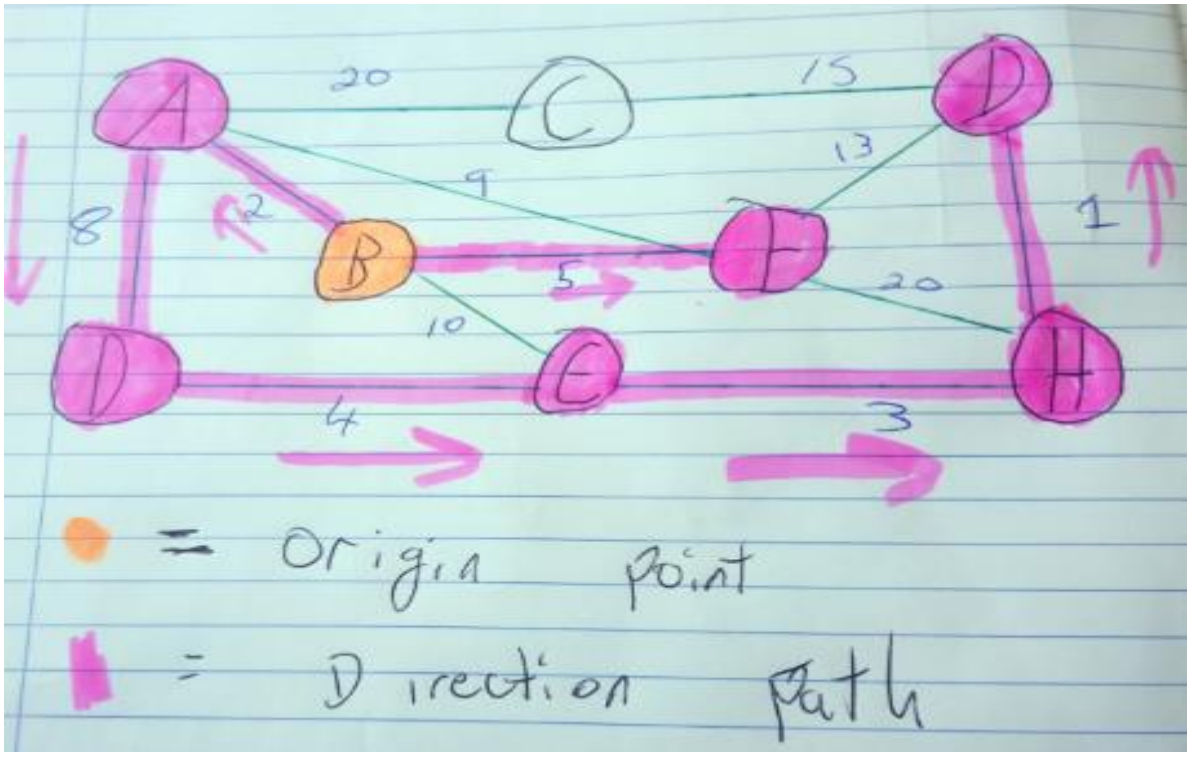


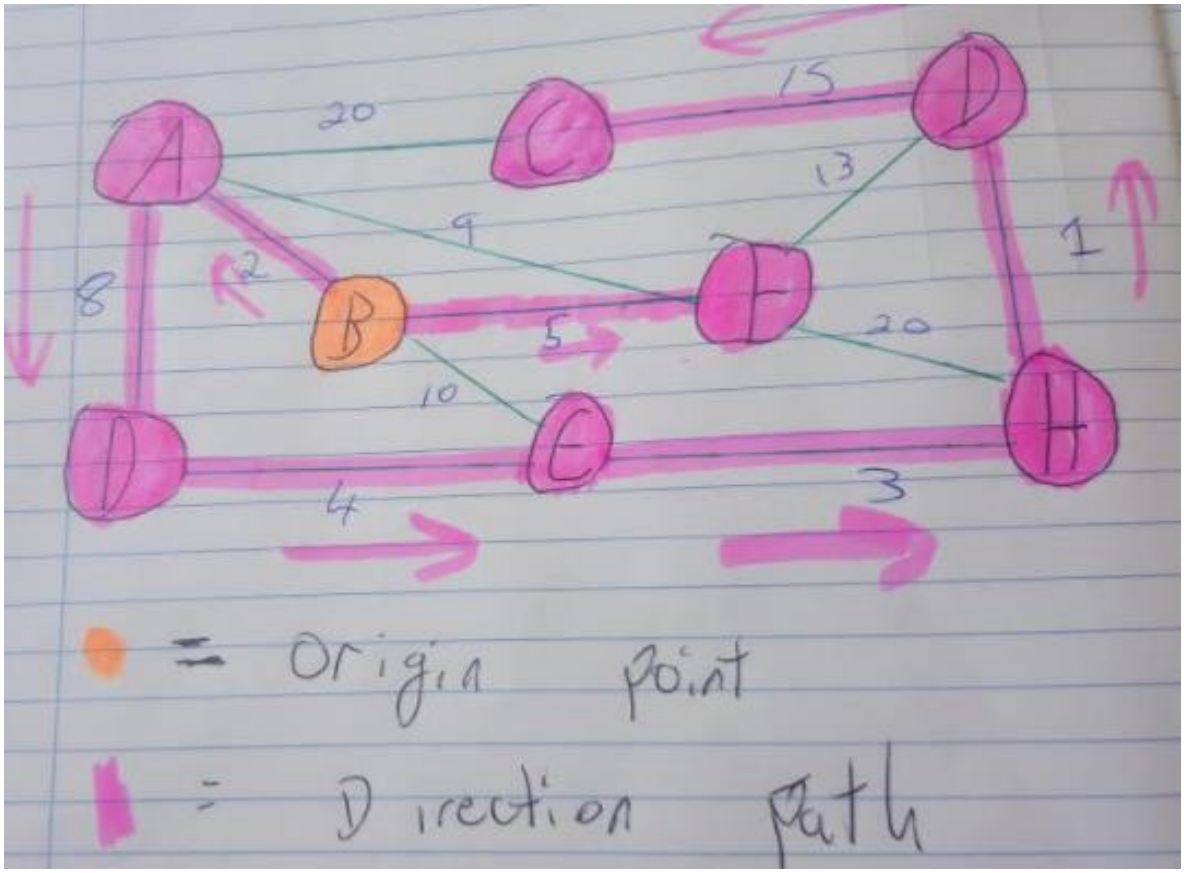



A
C
D
B
F
D
E
H
20
15
13
9
8
12
1
5
20
10
4
3
= Origin point
= Direction path

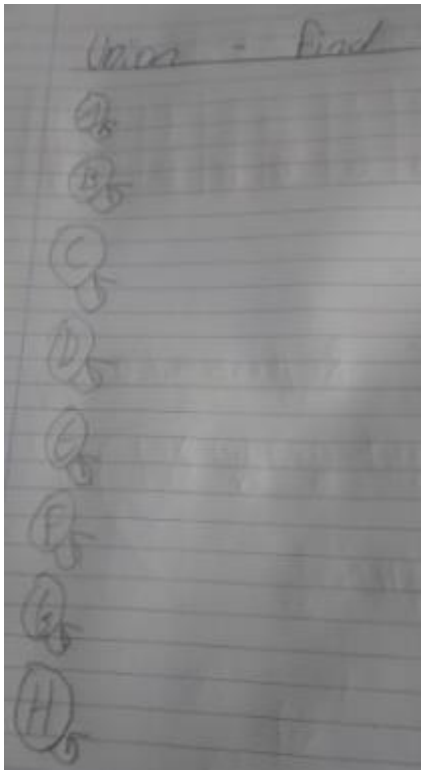

A
C
D
B
F
D
E
H
20
15
13
9
8
12
1
5
20
10
4
3
= Origin point
= Direction path

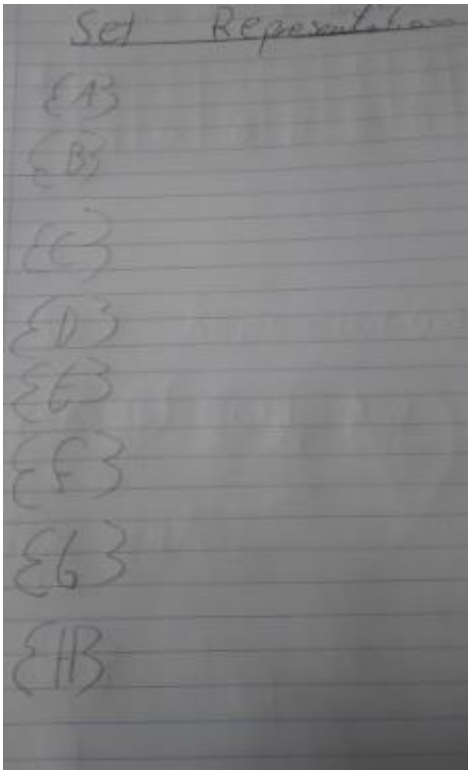

A
C
D
B
F
D
E
H
20
15
13
9
12
8
1
5
20
10
4
3
= Origin point
= Direction path

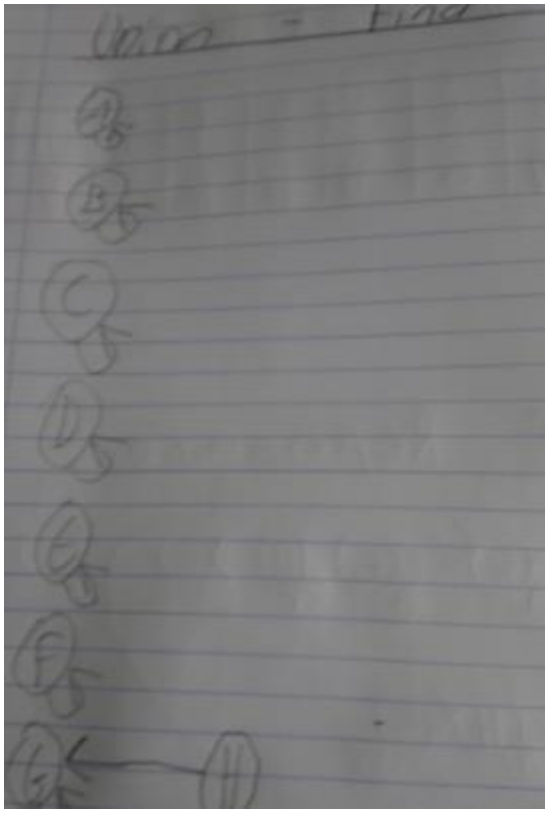

A
C
D
B
F
D
E
H
20
15
13
9
12
8
1
5
20
10
4
3
= Origin point
= Direction path

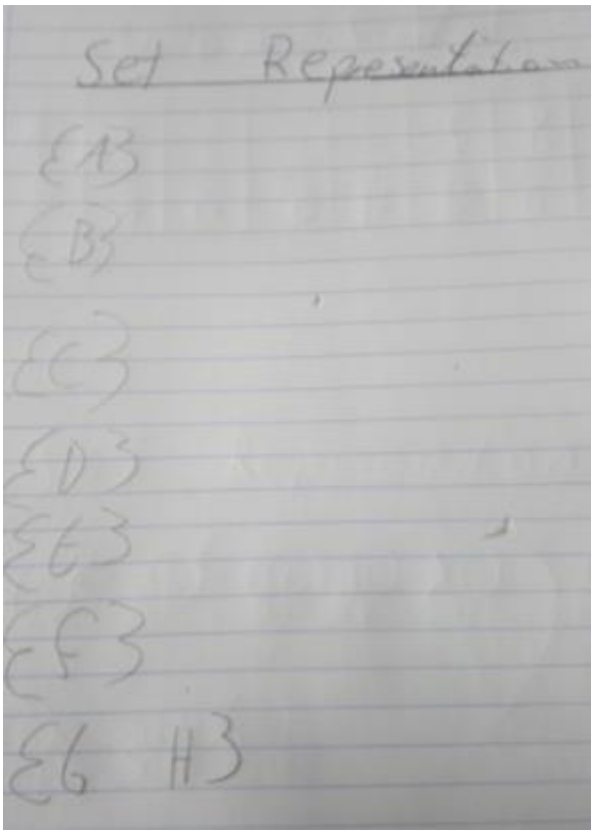

A
B
C
D
E
F
G
H
20
15
13
9
12
8
1
5
20
10
4
3
= Origin point
= Direction path

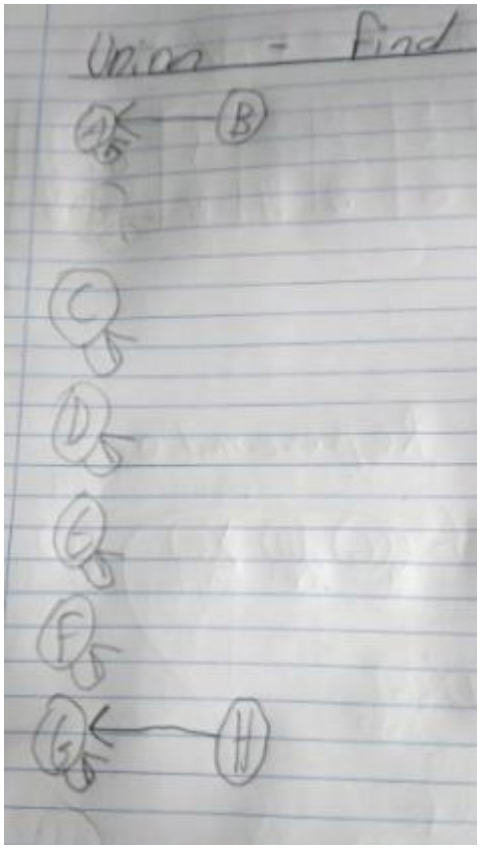
Including Union-Find Partition and Set Representation with Kruskal Algorithm:

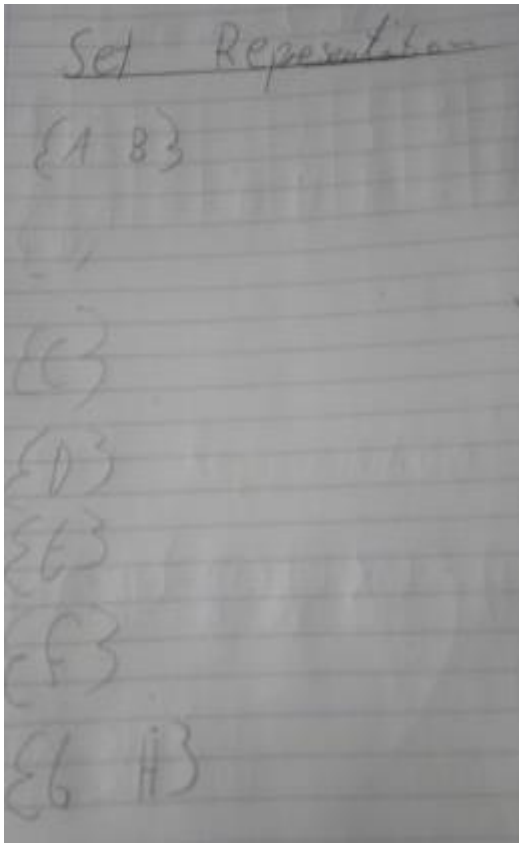


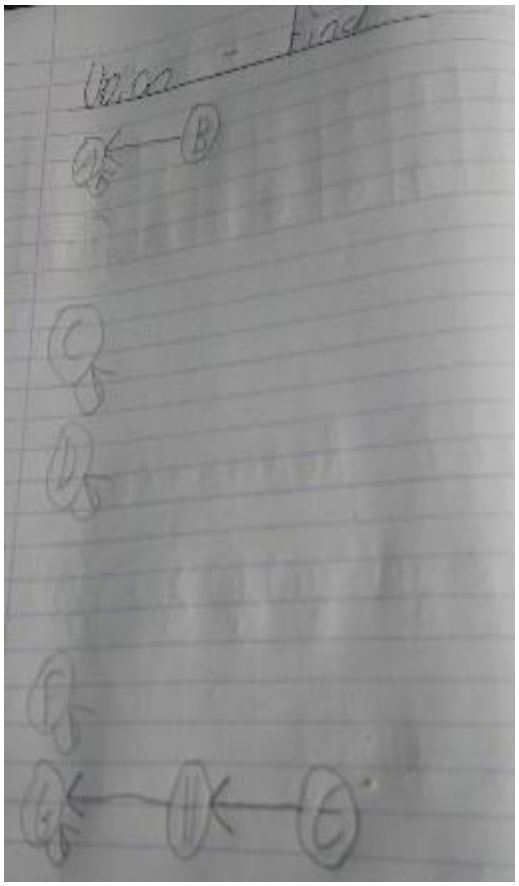


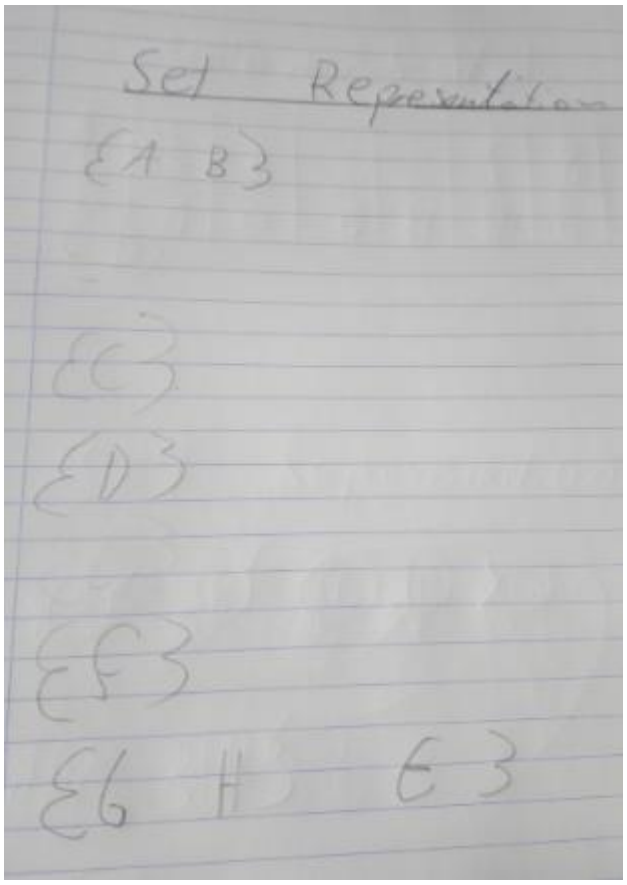





Union - Find
A
B
C
D
E
F
G
H

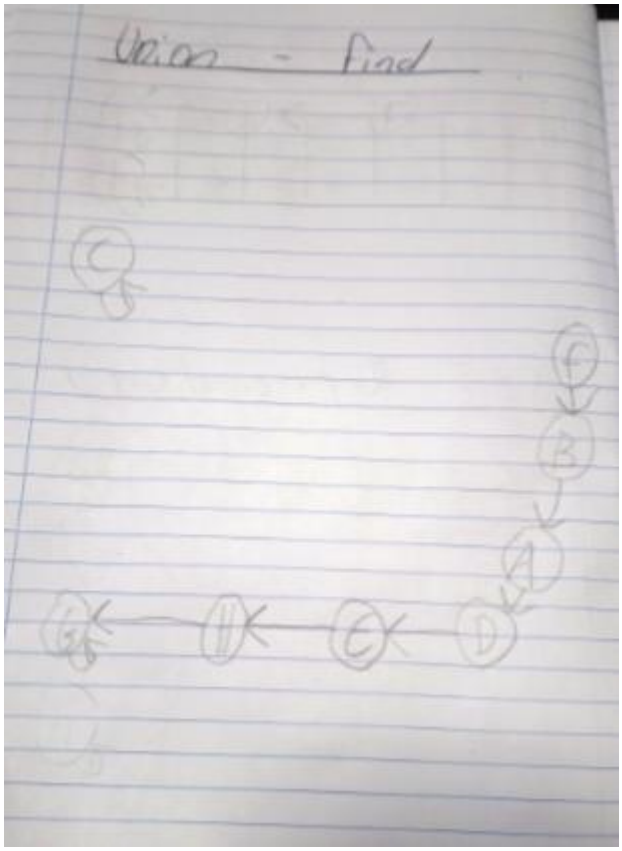

Set Representation
{A B}
{C}
{D}
{E}
{F}
{G H}


Union - Find
A
B
C
D
F
G
H
E


Set Representation
{A B}
{C}
{D}
{F}
{G H E}


Union - Find


Set Representation


Union - Find


Set Representation


Union - Find

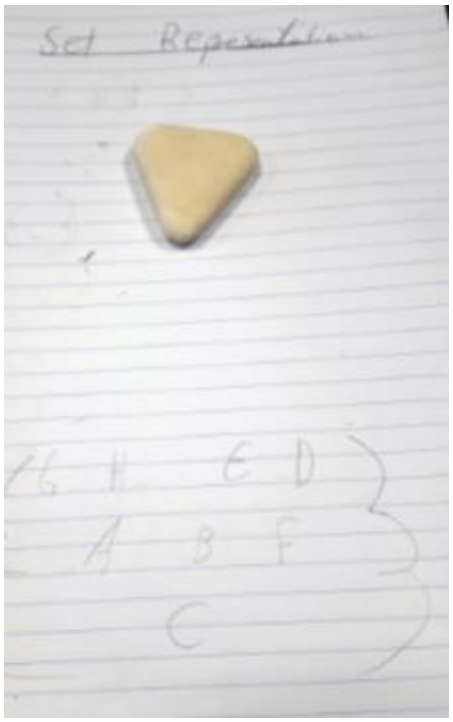
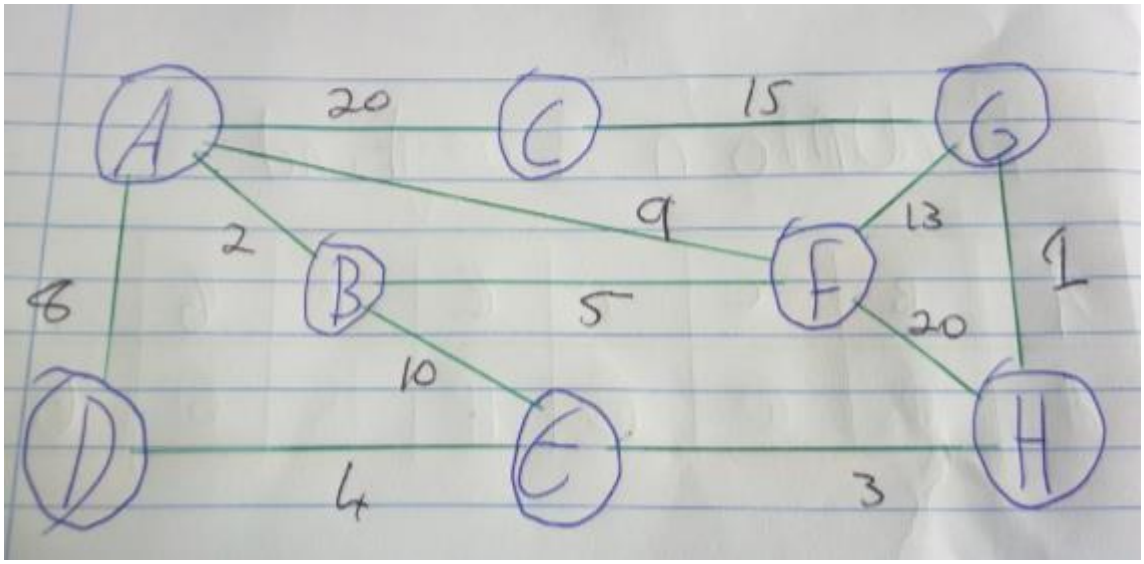
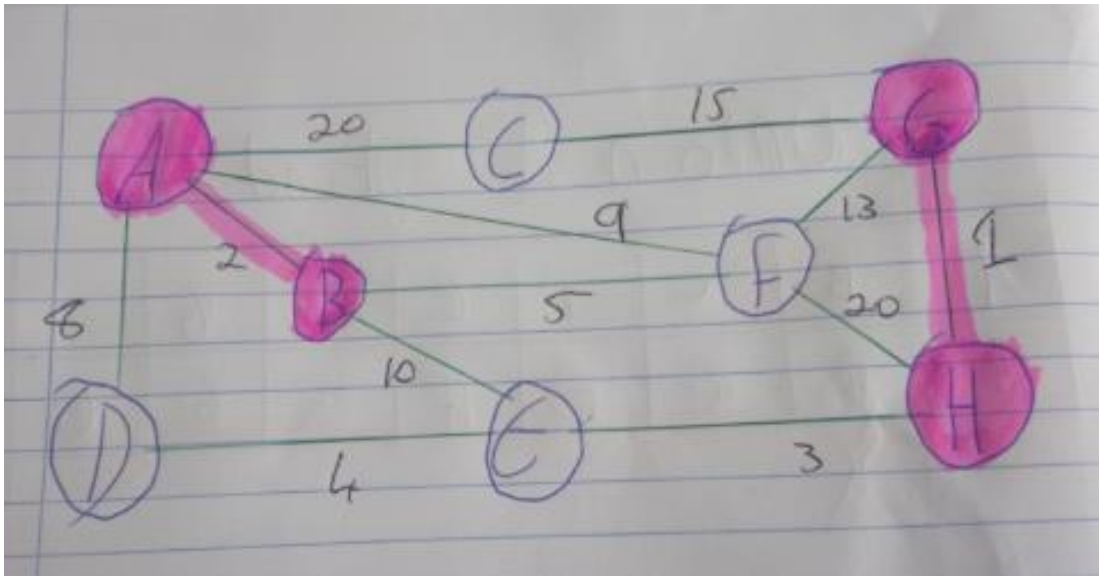
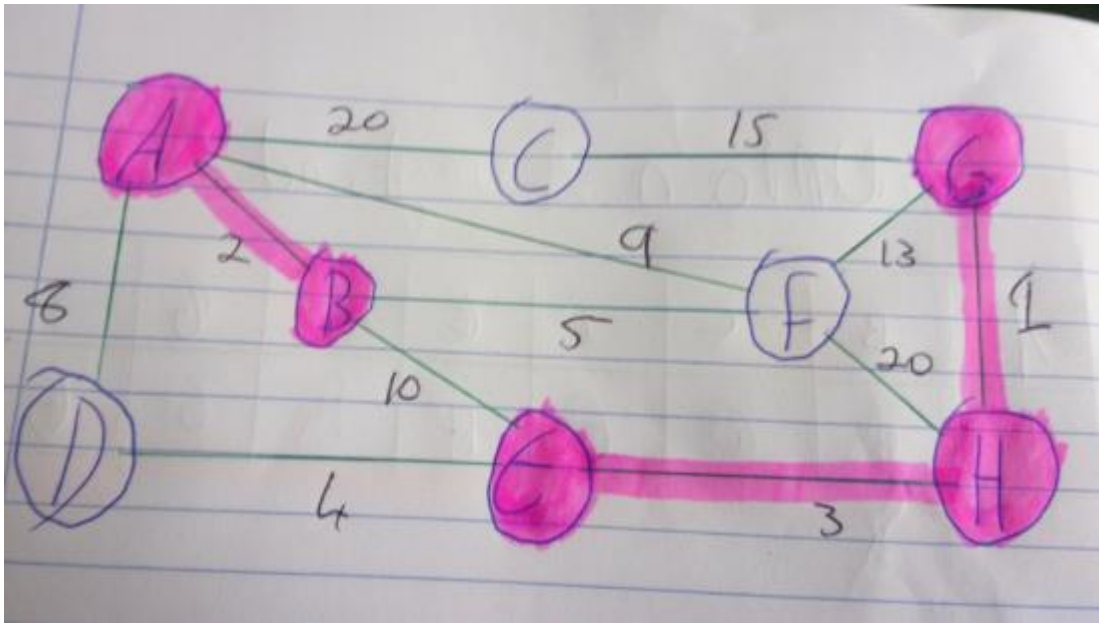

Set Representation

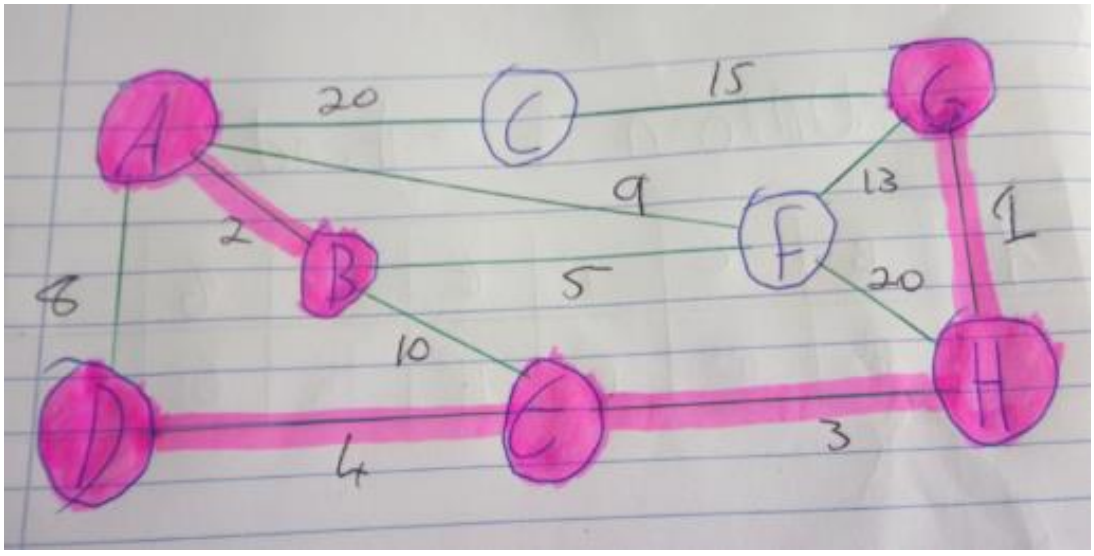
Diagram Representation:

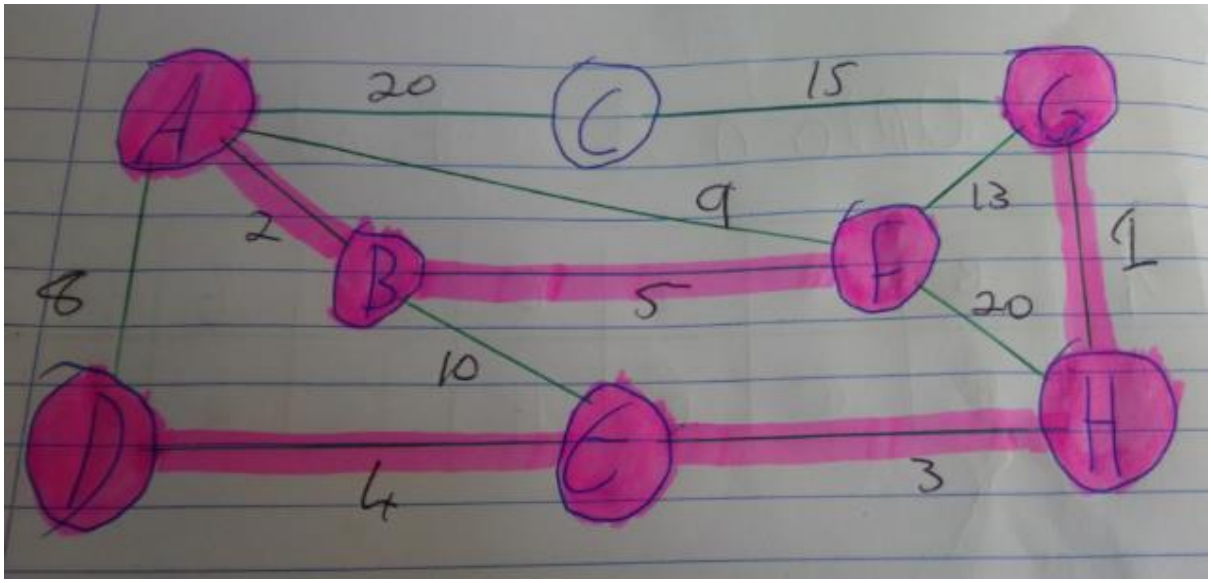
```
Reading edge 1 connecting the vertices G and H
Reading edge 2 connecting the vertices A and B
Reading edge 3 connecting the vertices E and H
Reading edge 4 connecting the vertices D and E
Reading edge 5 connecting the vertices B and F
Reading edge 8 connecting the vertices A and D
Reading edge 15 connecting the vertices C and G
```

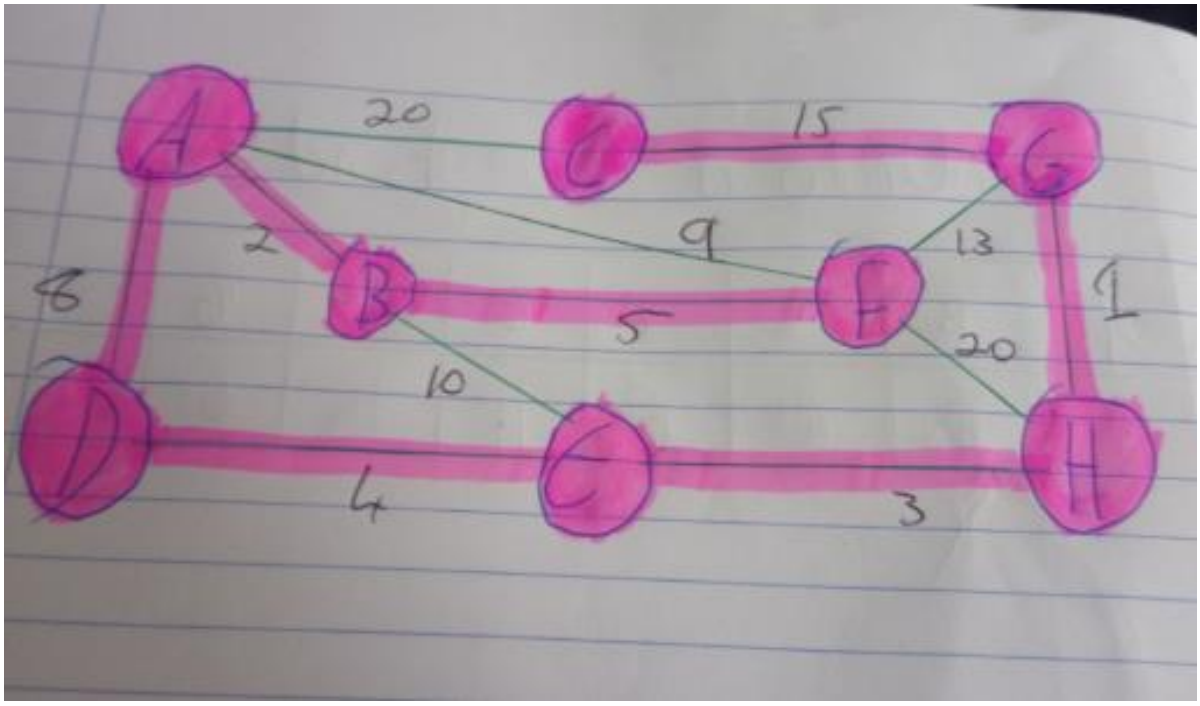





A
20
C
15
G
2
9
13
B
F
1
6
5
20
10
D
C
H
4
3


A
20
C
15
G
2
9
13
B
F
1
6
5
20
10
D
C
H
4
3


A
20
C
15
G
2
9
13
B
F
1
6
5
20
10
D
C
H
4
3


A
20
C
15
G
2
9
13
8
B
5
F
1
20
10
D
C
H
4
3

Code used in Algorithms:

Prim:

```
// Simple weighted graph representation
// Uses an Adjacency Linked Lists, suitable for sparse graphs

import java.io.*;
import java.util.*;
/************************************************************************************************
***********************************************
The class heap that gets data and sorts it out in descending order based upon their distance from each other
************************************************************************************************
***********************************************/
class Heap
{
   private int[] a;     // 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;
      a = new int[maxSize + 1];
      this.dist = dist;
      this.hPos = hPos;
      a[0] = 0;
      dist[0] = 0;
   }
```

```
public boolean isEmpty()
{
   return (N == 0 ? true : false);
}

public void insert( int x)
{
   a[++N] = x;
   siftUp( N);
}

public void siftUp( int k)
{
   int v = a[k];

   //a[0] = 0;
   //dist[0] = 0;

   while( dist[v] < dist[a[k/2]]  && k != 0) {
      a[k] = a[k/2];
      hPos[a[k]] = k;
      k = k/2;
   }
   a[k] = v;
   hPos[v] = k;

}



public void siftDown( int k)
{
   int v, j;

   v = a[k];
```

```
    while( k <= N/2) {
        j = 2 * k;
        if(j < N && dist[a[j]] < dist[a[j+1]]) ++j;
        if( dist[v] >= dist[a[j]]) break;
        a[k] = a[j];
        hPos[a[k]] = k;
        k = j;
    }
    a[k] = v;
    hPos[v] = k;
}

public int remove()
{
    int v = a[1];
    hPos[v] = 0; // v is no longer in heap
    a[N+1] = 0;  // put null node into empty spot

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

    return v;
}

// display heap values and their priorities or distances
public void display() {
    System.out.println("\n\nThe tree structure of the heaps is:");
    System.out.println( a[1] + "(" + dist[a[1]] + ")" );
    for(int i = 1; i<= N/2; i = i * 2) {
        for(int j = 2*i; j < 4*i && j <= N; ++j)
            System.out.print( a[j] + "(" + dist[a[j]] + ")  ");
        System.out.println();
    }
}
```

```
}//end class heap


/*****************************************************************************************
*****************************************
//class graph to read from the files and sort it into an adjency list
*****************************************************************************************
*****************************************/
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;



   /// default constructor
   public Graph(String graphFile)  throws IOException
   {
      int v1, v2, ed;
      int e,v;
      Node node, temp;

      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);
   v1 = Integer.parseInt(parts[0]);
   v2 = Integer.parseInt(parts[1]);
   ed = Integer.parseInt(parts[2]);

   System.out.println("Edge " + toChar(v1) + "--(" + ed + ")--" + toChar(v2));

   temp = adj[v1];
   adj[v1] = new Node();
   adj[v1].vert = v2;
```

```
        adj[v1].wgt = ed;
        adj[v1].next = temp;

        temp = adj[v2];
        adj[v2] = new Node();
        adj[v2].vert = v1;
        adj[v2].wgt = ed;
        adj[v2].next = temp;
    }
}

// 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.print(" NULL");
    }
    System.out.println("");
}


//mst based upon the prim algorithm
          public void MST_Prim(int s)
```

```
      {
int v;
int wgt_sum = 0;
int[]  dist, parent, hPos;

//code here
dist = new int[V + 1];
hPos = new int[V + 1];
parent = new int[V + 1];

for (int i = 1; i <= V; i++)
   dist[i] = Integer.MAX_VALUE;

dist[s] = 0;
parent[s] = 0;

showConstruction(s, dist, parent);

Heap pq =  new Heap(V, dist, hPos);
pq.insert(s);

// System.out.println("\nSource point is "+toChar(s));

while ( !(pq.isEmpty()) )
{
   v = pq.remove();
   dist[v] = -dist[v];
   Node u = adj[v];
   while (u != z)
   {
      if (u.wgt < dist[u.vert])
      {
         dist[u.vert] = u.wgt;
         parent[u.vert] = v;
```

```
            if (hPos[u.vert] == 0)
               pq.insert(u.vert);
            else
               pq.siftUp(hPos[u.vert]);

            showConstruction(u.vert, dist, parent);
         }

         u = u.next;
      }
   }

   for (int d: dist)
      wgt_sum += Math.abs(d);

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

   mst = parent;

   showMST();
         }

  public void showConstruction(int s, int [] dist, int [] parent)
  {
    if ( toChar(parent[s]) > 64 && toChar(parent[s]) < 91)
      System.out.println("\nCurrent Vertex is " + toChar(s) + " with parent value: " + toChar(parent[s]) + "  and the distance between the two vertices is " + dist[s]);
    else
      System.out.println("\nCurrent Vertex is " + toChar(s) + " with  itself as the Orgin point");
  }

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

```
        for(int v = 1; v <= V; ++v)
        {
           if ( toChar(mst[v]) > 64 && toChar(mst[v]) < 91)
                    System.out.println(toChar(v) + " -> " + toChar(mst[v]));
           else
                    System.out.println(toChar(v) + " -> Source Point");
        }
        System.out.println("");
   }

}//end graph algorithm



/*************************************************************************************************
*************************************
Main function
*************************************************************************************************
*************************************/
public class PrimLists {
   public static void main(String[] args) throws IOException
   {
           Scanner mstPrim = new Scanner(System.in);
           System.out.println("What is the name of the file ");
           String fname = "../" + mstPrim.nextLine();
           System.out.println("What is the starting vertice");
      int s = Integer.parseInt(mstPrim.nextLine());
      //String fname = "../wGraph3.txt";

      Graph g = new Graph(fname);

      g.MST_Prim(s);
   }



}
```

Kruskal:

```
// Kruskal's Minimum Spanning Tree Algorithm
// Union-find implemented using disjoint set trees without compression

import java.io.*;
import java.util.*;

//Class edge to store all the indiviadual information about each edge and vertices
class Edge {
   public int u, v, wgt;

   public Edge() {
      this(0,0,0);
   }

   public Edge( int x, int y, int w) {
      u = x;
      v = y;
      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);
   }
}


//Heap to sort the edges based off their weight in association with each vertices
class Heap
```

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


   // Bottom up heap construc
   public Heap(int N, Edge[] edge) {
      int i,j, temp;
      Nmax = this.N = N;
      h = new int[N+1];
      this.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(i);
      }

   }

   private void siftDown( int k) {
      int e, j;

      e = h[k];
      while( k <= N/2) {
         j = 2*k;
         if (j < N && edge[h[j]].wgt > edge[h[j+1]].wgt) j++;
```

```
            if (edge[e].wgt < edge[h[j]].wgt) break;

            h[k] = h[j];
            k = j;
        }
        h[k] = e;
    }


    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 <= V; i++)
        treeParent[i] = i;
}

public int findSet( int vertex)
{
    if (treeParent[vertex] == vertex)
        return vertex;
        else
                return findSet(treeParent[vertex]);
}

public void union( int set1, int set2)
{
    int x = findSet(set1);
    int y = findSet(set2);
    treeParent[y] = x;
}

public void showTrees()
{
    int i;
    for(i=1; i<=N; ++i)
       System.out.print(toChar(i) + "->" + toChar(treeParent[i]) + "  " );
    System.out.print("\n");
}

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");
      }

      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 to read in a file and use the kruskal to find the MST
*****************************************************************************************
*******************************************************/
class Graph
{
   private int V, E;
   private Edge[] edge;
```

```
private Edge[] mst;

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();
   String[] parts = line.split(splits);
   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 edgey = new Edge(u,v,w);
            edge[e] = edgey;
        }
    }


/**********************************************************
*
*      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);

    for(int z = 0; z < E; z++)
    {
            ei = h.remove();

            uSet = edge[ei].u;
            vSet = edge[ei].v;
```

```
            int check1 = partition.findSet(uSet);
            int check2 = partition.findSet(vSet);

            if (check1 != check2)
            {

                    mst[i++] = edge[ei];
                    //System.out.println("\n Vertice 1 = "+ mst[z].u + " Vertice2 = " + mst[z].v + " Edge =
"+mst[z].wgt);
                    partition.union(uSet,vSet);

        System.out.println("Reading edge " + edge[ei].wgt + " connecting the vertices " + toChar(uSet) + " and "
+ toChar(vSet));
            }

            if (i == V-1) break;

    }

    return mst;
}

    // convert vertex into char for pretty printing
    private char toChar(int u)
    {
       return (char)(u + 64);
    }

    public void showMST()
    {
       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


//class Kruskal tree to demonstate the kruskal algorithm in the main function using the heap class, graph class and edge class
class KruskalTrees {
    public static void main(String[] args) throws IOException
    {
        //String fname = "../wGraph3.txt";
        Scanner mstPrim = new Scanner(System.in);
        System.out.println("What is the name of the file ");
        String fname = "../" + mstPrim.nextLine();

        //System.out.print("\nInput name of file with graph definition: ");
        //fname = Console.ReadLine();

        Graph g = new Graph(fname);

        g.MST_Kruskal();

        g.showMST();

    }
}
```

Sample output of the MST implementation for the sample graph:

Prim:

```
Weight of MST = 38


Minimum Spanning tree parent array is:
A -> B
B -> Source Point
C -> G
D -> A
E -> D
F -> B
G -> H
H -> E
```

Kruskal:

```
Minimum spanning tree build from following edges:
Edge G--1--H
Edge A--2--B
Edge E--3--H
Edge D--4--E
Edge B--5--F
Edge A--8--D
Edge C--15--G
```