GABRIEL GRIMBERG

C15478448

Dublin Institute of Technology · Institiúid Teicneolaíochta Átha Cliath ·

D·I·T

ALGORITHM & DATA STRUCTURES ASSIGNMENT

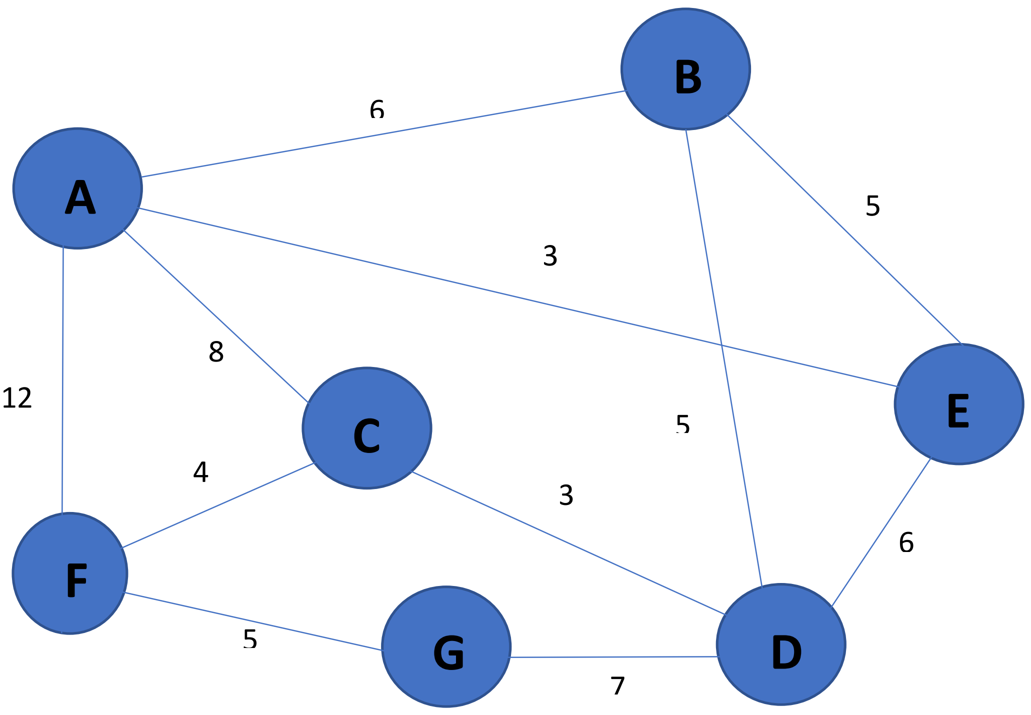
COURSE: DT228/2

Report for Prim's Algorithm and Kruskal's Algorithm

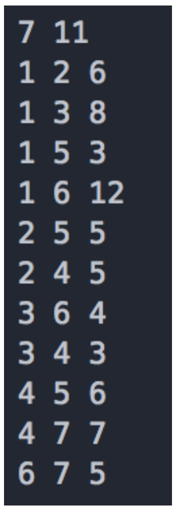
Table of Contents

Graph Made

The graph below is a graph I have made as requested and named it **myGraph.txt**.



I have then turned this graph into a text file, as this is how the program will read the graph. The first line has two numbers **7** and **11**. 7 is the amount of Vertices and 11 is the amount of the Edges. The rest of the lines are basically the weight each node is connected to its neighbour, so for example on the second line you **1** which is A is connected to **2** which is B with the weight of 6 and so on.

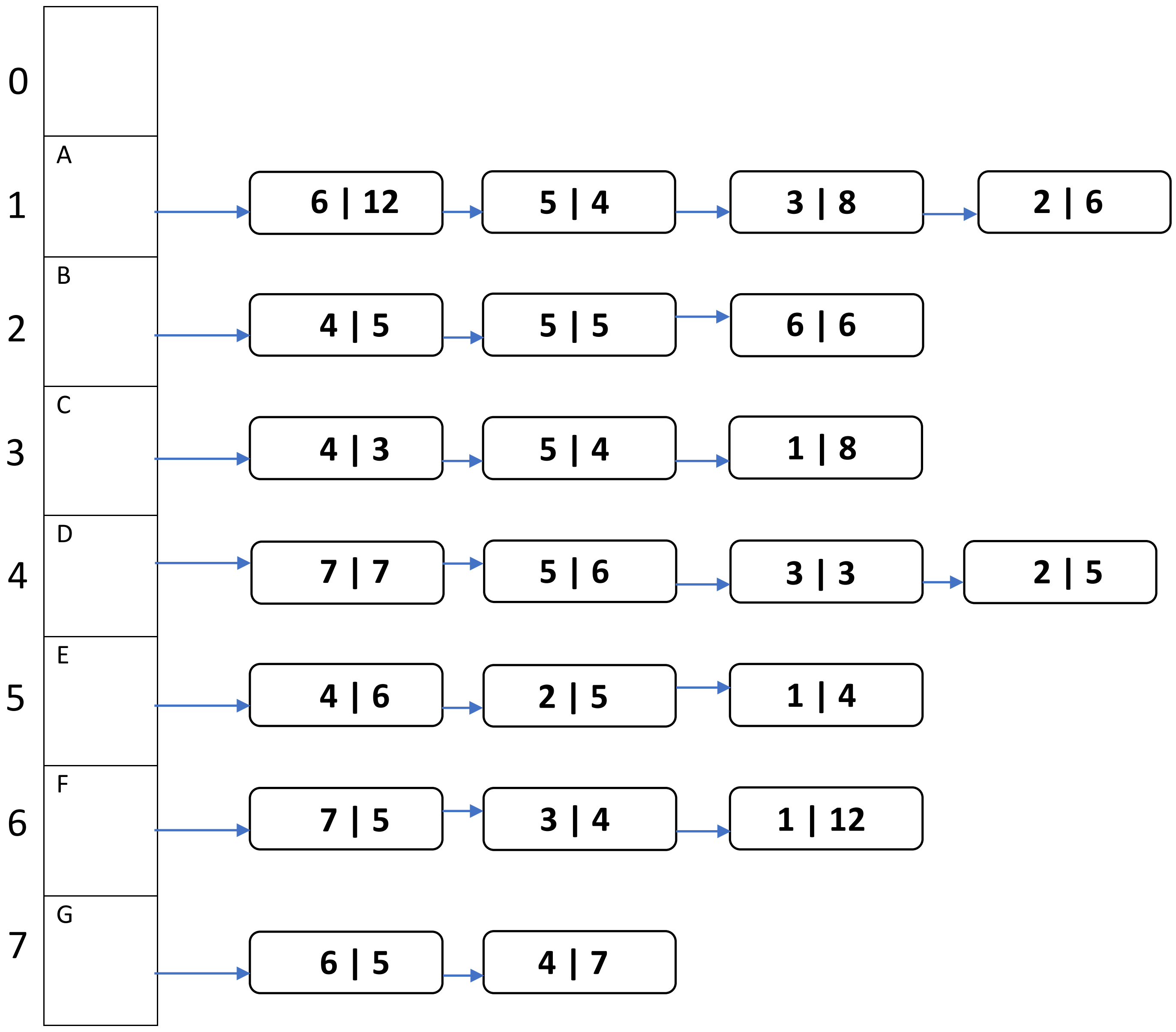


```
7 11
1 2 6
1 3 8
1 5 3
1 6 12
2 5 5
2 4 5
3 6 4
3 4 3
4 5 6
4 7 7
6 7 5
```

Graph Representation

In this section I will draw out the linked list representation of the graph I have created. Every Vertex has a linked list associated with it which only records those vertices connected to it with an edge the weights are also recorded. So for example **A** has four Vertices connected to it so it will only show those four Vertices with the weight.

After Reading All Edges



This is the representation of my sample graph using the linked list diagram version, I presume this is the only one that's needed instead of iterating it one by one.

Prim’s Algorithm

Introduction

This is a program called Prim’s Algorithm that is written in Java to find the shortest path and to return the weight of that shortest path. There are many ways to write Prim’s Algorithm although for this assignment it uses heap and linked list to create the graph from the text file given.

Once the file has been read it will create a linked list and display the vertices connected to other vertices and what their weight is.

The program will then cycle through the heap to find the Minimum Spanning Tree and calculate the weight of that Minimum Spanning Tree.

Explanation

Prim’s Algorithm is a greedy algorithm that finds a Minimum Spanning Tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every Vertex, where the total weight of all the edges in the tree is minimized.

Prim’s Algorithm is Vertex based when compared to Kruskal’s Algorithm which is Edge based. In Prim’s Algorithm we build the spanning tree from a given Vertex, adding the smallest edge to the Minimum Spanning Tree.

This program uses the Eager Implementation of Prim’s Algorithm as we keep updating the heap if the distance from a Vertex to the Minimum Spanning Tree has changed.

Time Complexity for Adjacency List: O(E + V Log V)

Step by Step MST Construction

In this section it will be showing what's in the Parent[] and Dist[] arrays after each traverse of Prim's Algorithm. This is the traverse of when we start from the Vertex D also known as 4.

1st Traverse

Parent[]	**Dist[]**
A -> **@**	**1** or **A** -> **Max Value**
B -> **D**	**2** or **B** -> **5**
C -> **D**	**3** or **C** -> **3**
D -> **@**	**4** or **D** -> **0**
E -> **D**	**5** or **E** -> **6**
F -> **@**	**6** or **F** -> **Max Value**
G -> **D**	**7** or **G** -> **7**

2nd Traverse

Parent[]	**Dist[]**
A -> **C**	**1** or **A** -> **8**
B -> **D**	**2** or **B** -> **5**
C -> **D**	**3** or **C** -> **-3**
D -> **@**	**4** or **D** -> **0**
E -> **D**	**5** or **E** -> **6**
F -> **C**	**6** or **F** -> **4**
G -> **D**	**7** or **G** -> **7**

3rd Traverse

Parent[]	**Dist[]**
A -> **C**	**1** or **A** -> **8**
B -> **D**	**2** or **B** -> **5**
C -> **D**	**3** or **C** -> **-3**
D -> **@**	**4** or **D** -> **0**
E -> **D**	**5** or **E** -> **6**
F -> **C**	**6** or **F** -> **-4**
G -> **F**	**7** or **G** -> **5**

4th Traverse

Parent[]	**Dist[]**
A -> **B**	**1** or **A** -> **6**
B -> **D**	**2** or **B** -> **-5**
C -> **D**	**3** or **C** -> **-3**
D -> **@**	**4** or **D** -> **0**
E -> **B**	**5** or **E** -> **5**
F -> **C**	**6** or **F** -> **-4**
G -> **F**	**7** or **G** -> **5**

5th Traverse

Parent[]	**Dist[]**
A -> **B**	**1** or **A** -> **6**
B -> **D**	**2** or **B** -> **-5**
C -> **D**	**3** or **C** -> **-3**
D -> **@**	**4** or **D** -> **0**
E -> **B**	**5** or **E** -> **5**
F -> **C**	**6** or **F** -> **-4**
G -> **D**	**7** or **G** -> **-5**

6th Traverse

Parent[]	**Dist[]**
A -> **E**	**1** or **A** -> **4**
B -> **D**	**2** or **B** -> **-5**
C -> **D**	**3** or **C** -> **-3**
D -> **@**	**4** or **D** -> **0**
E -> **B**	**5** or **E** -> **-5**
F -> **C**	**6** or **F** -> **-4**
G -> **F**	**7** or **G** -> **-5**

7th Traverse

Parent[]	**Dist[]**
A -> **E**	**1** or **A** -> **-4**
B -> **D**	**2** or **B** -> **-5**
C -> **D**	**3** or **C** -> **-3**
D -> **@**	**4** or **D** -> **0**
E -> **B**	**5** or **E** -> **-5**
F -> **C**	**6** or **F** -> **-4**
G -> **F**	**7** or **G** -> **-5**

- The heap **pq** is a heap to find the next Vertex that is nearest to the Minimum Spanning Tree so it can be added to it.

- The Dist[] array is an integer array that holds the current distance of a Vertex from the Minimum Spanning Tree.

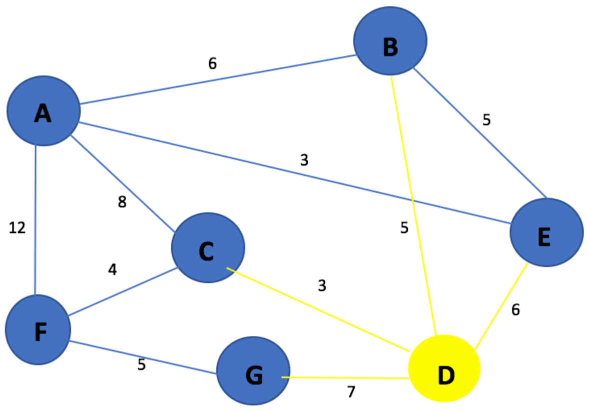
- The Parent[] array is an integer array that stores the Minimum Spanning Tree.

- The hPos[] array is an integer array that holds the position of any Vertex with the heap array h[].

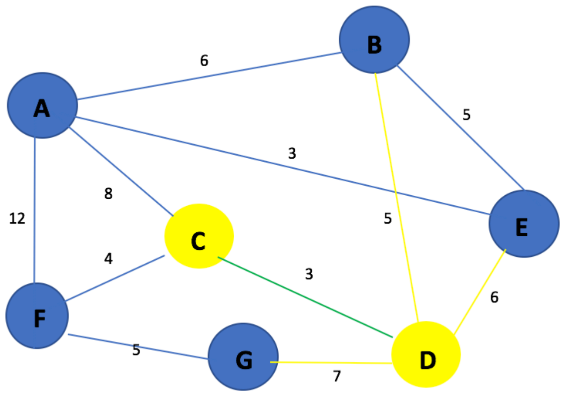
Prim's Algorithm On Graph Diagram

Here is the graph, D will be the starting Vertex as for Prim's Algorithm you need to pick a starting Vertex.

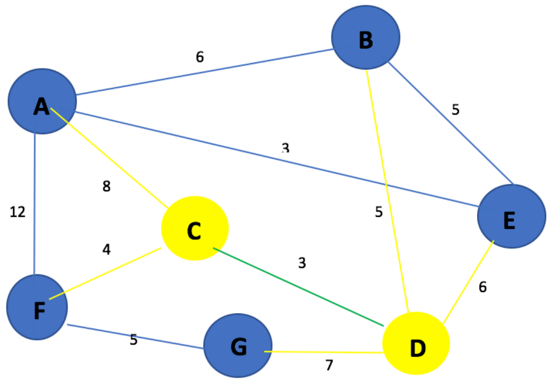
So we have four options to go to G, B, C or E. We need to look at which Vertex has the lowest edge so that would be C because the edge between D and C is 3 so we go to C. Heap Content: DG – 7, **DC – 3**, DB – 5, DE – 6.



The connection between D and C is made as the edge between them was the lowest. Heap Content: DG – 7, DB – 5, DE – 6.

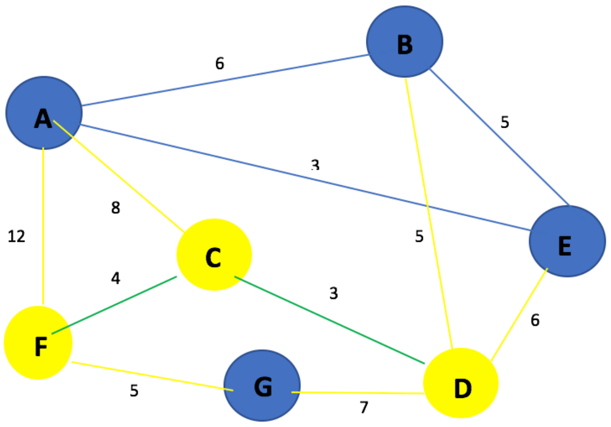


Now we must choose where we want to go from C, we have the Vertices A and F as options. Vertex F will be picked as the edge between C and F is lower than the edge between C and A. So we now go to Vertex F. Heap Content: DG – 7, DB – 5, DE – 6, CA – 8, **CF – 4.**

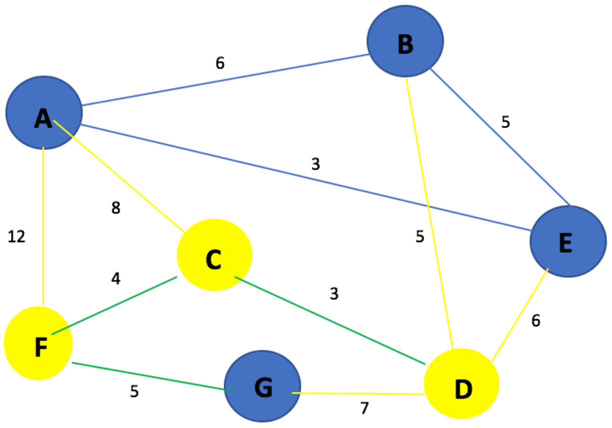


Now we have to make a decision on where to go from F. Remember we consider edges to vertexes that we have not picked yet and we pick the one with the lowest cost.

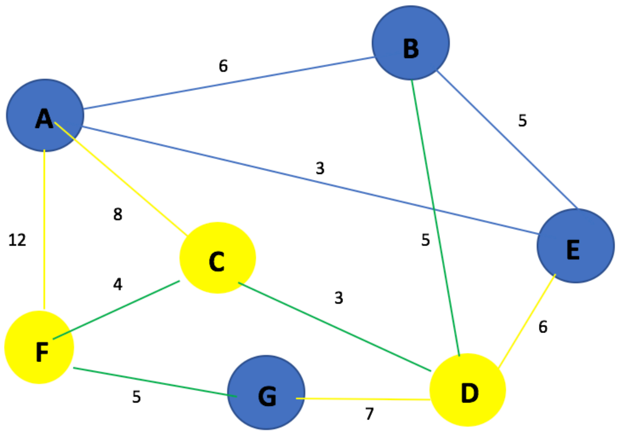
In this case F to G is better than F to A as the cost is lower so we go to Vertex G. Heap Content: DG – 7, DB – 5, DE – 6, CA – 8, FA – 12, **FG – 5**.



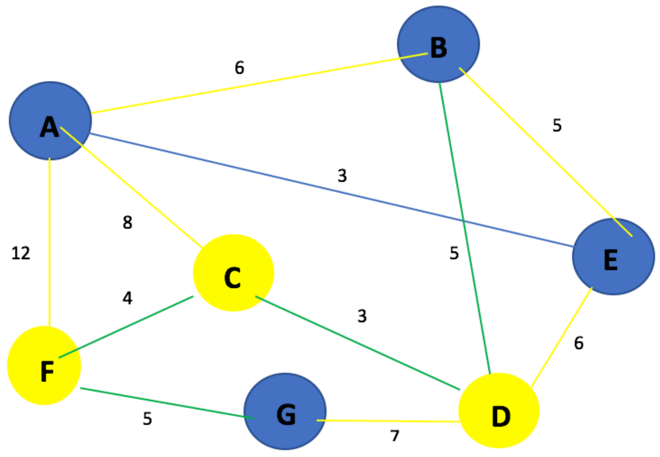
Going from G to D won't be possible as D has already been visited so we go back to D and consider where we go to, D to B or D to E. Heap Content: DG – 7, DB – 5, DE – 6, CA – 8, FA – 12.



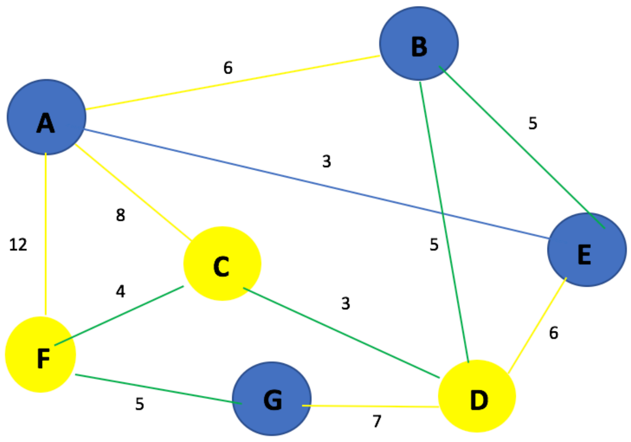
Going to B from D will be picked as the weight between DB is 5 and the weight between DE is 6. Heap Content: DG – 7, **DB – 5**, DE – 6, CA – 8, FA – 12.



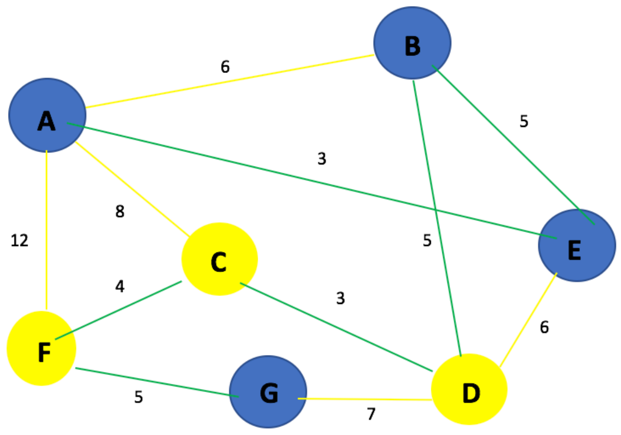
Now that we are on Vertex B we have two option either BA or BE, We will go with BE as the weight is 5 compared to BA which is 6. Heap Content: DG – 7, DE – 6, CA – 8, FA – 12, BA – 6, **BE – 5**.



We are now on Vertex E. We look at the option we have, we can't go back to D as that Vertex has already been visited, we can't go back to B as that Vertex has already been visited too. So our only option is going to A. Heap Content: DG – 7, DE – 6, CA – 8, FA – 12, BA – 6, **EA – 3**.

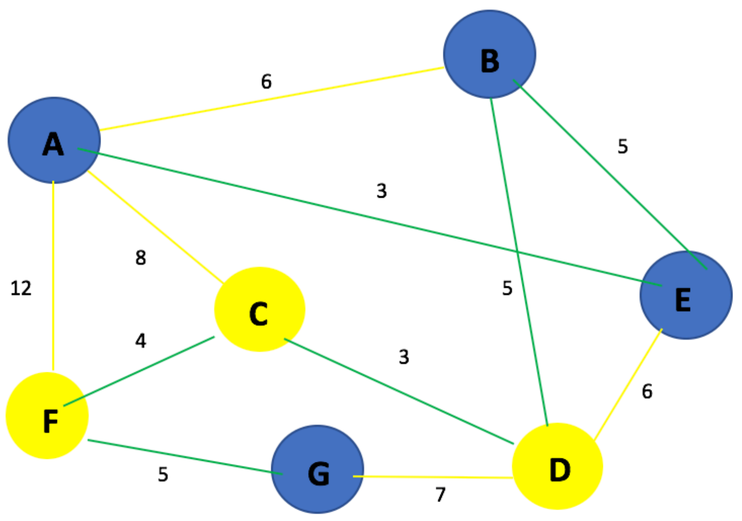


As you can see we now have a Minimum Spanning Tree as all of the Vertices have been visited. Now we can calculate the total cost of the Minimum Spanning Tree which is: 3 + 4 + 5 + 5 + 5 + 3 = **25**

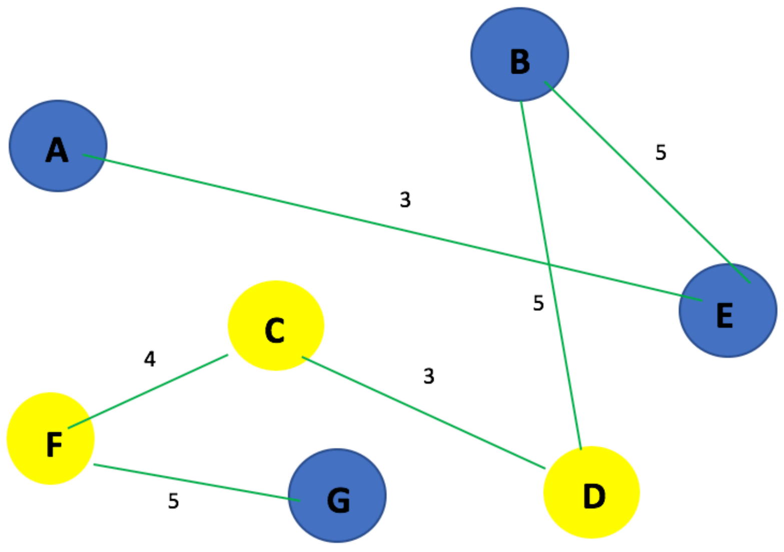


Complete Minimum Spanning Tree

Below is a complete Minimum Spanning Tree containing all the edges.



Below is a complete Minimum Spanning Tree without the edges that are not used.



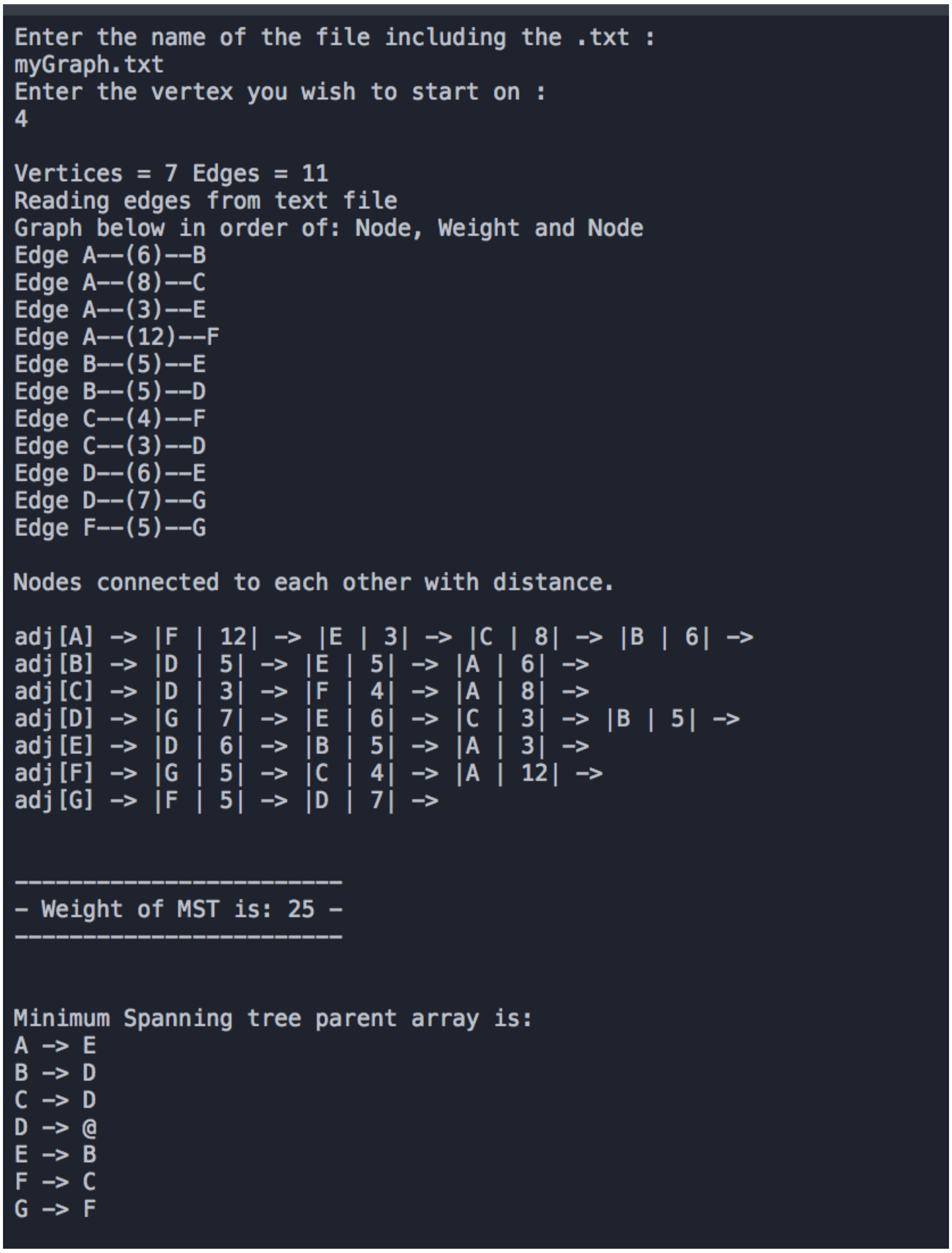
Prim’s Algorithm Code

```
/*
    Name: Gabriel Grimberg.
    Course: DT228/2.
    Module: Algorithms & Data Structures.
    Type: Final Year Assignment.
    Code: Prim's Algorithm using a Priority Queue(Heap).
*/

import java.io.*;
import java.util.Scanner; //For keyboard input.

class Heap
{
    private int[] h;     // Heap array
    private int[] hPos;         // hPos[h[k]] == k
    private int[] dist;    // dist[v] = priority of v

    //Size of heap.
    private int N;

    // 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;                      //Assume size of heap is 0.
        h = new int[maxSize + 1];
        dist = _dist;               //Given as parameters and you intiliasie them.
        hPos = _hPos;               //Given as parameters and you intiliasie them.
    }

    //Method that checks if the heap is empty.
    public boolean isEmpty()
    {
        return N == 0;
    }


    public void siftUp( int k)
    {
        //Current position of the vertex.
        int v = h[k];

        //Making a dummy node to place at the top of the heap.
        //h[0] = 0;

        //Smallest value into distance 0 so it can be compared.
        dist[0] = 0;

        //While distance value at the current element.
        //Is less than the distance value at k / 2.
        //Keep dividing going up the list to insert the element
        //At the correct place.
        while(dist[v] < dist[ h[k / 2] ])
        {
            //If not found, then siftUp the value to next value.
            h[k] = h[k / 2];
            hPos[ h[k] ] = k;

            //Then divide the element by 2 to go up.
            k = k / 2;
        }

        //Insert Vertex into the correct place on to the heap array.
        h[k] = v;
        hPos[v] = k;
    }


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

        v = h[k];

        j = 2 * k;

        while(j <= N)
        {
            if( (j + 1 <= N) && dist[ h[j] ] > dist[ h[j + 1] ] )
            {
                j++; //Next element.
            }

            //If the distance of vertex element we are currently on is
            //Greater than the vertex we are sifting down then abort.
            if( dist[ h[j] ] >= dist[v] )
            {
                break;
            }
            //If it's not then continue down the list.
            else
            {
                h[k] = h[j];
                k = j;
                j = k * 2;
            }
        }

        //When the right position is found put it into the array.
        h[k] = v;
        hPos[v] = k;
    }


    public void insert( int x)
    {
        //Adding a new vertex onto the end of the heap array.
        h[++N] = x;

        //Sifting up the vertex to find the correct location.
        siftUp( N);
    }

    public int remove()
    {
        //Give the element a 0 value then sift it down.
        int v = h[1];

        //V is no longer in heap.
        hPos[v] = 0;

        //Put null node into empty spot
        h[N+1] = 0;

        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;

    FileReader fr = new FileReader(graphFile);
          BufferedReader reader = new BufferedReader(fr);

    //Multiple whitespace as delimiter
    String splits = " +";
          String line = reader.readLine();
    String[] parts = line.split(splits);
    System.out.println("\nVertices = " + parts[0] + " Edges = " + 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");
    System.out.println("Graph below in order of: Node, Weight and Node");

    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));

        //Putting edge into adjacency matrix into the linked list.
        t = new Node();
        t.wgt = wgt;
        t.vert = v;
```

```
            t.next = adj[u];
            adj[u] = t;

            t = new Node();
            t.wgt = wgt;
            t.vert = u;
            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;

        System.out.println("\nNodes connected to each other with distance.");
        for(v = 1; v <= V; ++v)
        {
            System.out.print("\nadj[" + toChar(v) + "] ->" );

            //Go through the vertices, for each one start at the beginning of the Linked
               List.
            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;

        dist = new int[V + 1];      //Distance from node to node.
        parent = new int[V + 1];    //Parent node.
        hPos = new int[V + 1];      //Current heap position.


        for(v = 0; v <= V; v++)
        {
            dist[v] = Integer.MAX_VALUE; //Set to infinity.

            parent[v] = 0; //Treat 0 as a special null vertex.

            hPos[v] = 0; //Indicates that it's not on the heap.
        }

        //Creating a new empty priority heap.
        //V is the max size of the heap array.
        Heap pq =  new Heap(V, dist, hPos);

        //Insert first element into the heap, s is used a the root of the MST.
        pq.insert(s);

        //Set the distance to 0.
        dist[s] = 0;
        while (!pq.isEmpty())
        {
            v = pq.remove(); //Adding V to the MST.
            dist[v] = -dist[v]; //Marking V as in the MST.

            Node n;
            int w;

            //Examine each neighbour u of v.
            for (n = adj[v]; n != z; n = n.next)
            {
                u = n.vert;
                w = n.wgt;

                if (w < dist[u])
                {
                    if (dist[u] != Integer.MAX_VALUE)
                    {
                        wgt_sum -= dist[u];
                    }

                    dist[u] = w;
                    parent[u] = v;
                    wgt_sum += w;

                    if (hPos[u] == 0)
                    {
                        pq.insert(u);
                    }
                    else
                    {
                        pq.siftUp(hPos[u]);
                    }
                }
            }

            /*
            //Contents of Dist[] and Parent[] Array Step by Step. *For Report Purposes*
            System.out.println("\n");

            for(int i = 1; i <= V; ++i)
            {
                System.out.println("Parent Array");
                System.out.println(toChar(i) + " -> " + toChar(parent[i]));
            }
            System.out.println("\n");

            for(int x = 1; x <= V; ++x)
            {
                System.out.println("Dist[] Array");
                System.out.println(x + " OR " + toChar(x) + " -> " + dist[x] );
            }
            */
        }
        //Displaying the weight of the graph.
        System.out.println("\n");
        System.out.println("-------------------------");
        System.out.println("- Weight of MST is: " + wgt_sum + " -");
        System.out.println("-------------------------");

        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
    {
        Scanner keyInput = new Scanner(System.in);

        //Entering the file name.
        System.out.println("Enter the name of the file including the .txt : ");
        String fname = keyInput.nextLine();

        //Entering the starting vertex.
        System.out.println("Enter the vertex you wish to start on : ");
        int startVertex = keyInput.nextInt();

        Graph g = new Graph(fname);

        g.display();     //Display the graph.
        g.MST_Prim(startVertex);   //Performe Algorithm and display the weight of it.
        g.showMST();     //Show the MST.

    }
}
```

Screen Capture of Output



```
Enter the name of the file including the .txt :
myGraph.txt
Enter the vertex you wish to start on :
4

Vertices = 7 Edges = 11
Reading edges from text file
Graph below in order of: Node, Weight and Node
Edge A--(6)--B
Edge A--(8)--C
Edge A--(3)--E
Edge A--(12)--F
Edge B--(5)--E
Edge B--(5)--D
Edge C--(4)--F
Edge C--(3)--D
Edge D--(6)--E
Edge D--(7)--G
Edge F--(5)--G

Nodes connected to each other with distance.

adj[A] -> |F | 12| -> |E | 3| -> |C | 8| -> |B | 6| ->
adj[B] -> |D | 5| -> |E | 5| -> |A | 6| ->
adj[C] -> |D | 3| -> |F | 4| -> |A | 8| ->
adj[D] -> |G | 7| -> |E | 6| -> |C | 3| -> |B | 5| ->
adj[E] -> |D | 6| -> |B | 5| -> |A | 3| ->
adj[F] -> |G | 5| -> |C | 4| -> |A | 12| ->
adj[G] -> |F | 5| -> |D | 7| ->


------------------------
- Weight of MST is: 25 -
------------------------


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

Kruskal’s Algorithm

Introduction

This is a program called Kruskal’s Algorithm that is written in Java to find the shortest path and to return the weight of that shortest path. There are many ways to write Kruskal’s Algorithm although for this assignment it uses the Union Find Data Structure known as Disjoint Set.

Once the file has been read it will create a linked list and display the vertices connected to other vertices and what their weight is.

The program will then cycle through the heap to find the Minimum Spanning Tree and calculate the weight of that Minimum Spanning Tree.

Explanation

Kruskal’s Algorithm sort’s the edges according to their edge weights, it can be done with Merge-Sort or Quick-Sort although for this assignment the Union Find Data Structure is used.

Kruskal’s algorithm can be shown to run in O(E Log V) time.

The implementation for this algorithm is that we build the spanning tree separately adding the smallest edge to the spanning tree if there is no cycle. As mentioned before Prim’s Algorithm is Vertex based and Kruskal’s algorithm is Edge based.

Step by Step MST Construction

In this section it will be showing the Union Find Partition and Set Representations every traverse for the Kruskal's Algorithm. Note that we do not pick a starting Vertex, this Algorithm compares it by the weight of the edges, starting with the lowest edge and moving up.

***

1st Traverse

Edge **C** to **D** with the weight of **3**.

Set{**A**}
Set{**B**}
Set{**CD**}
Set{**E**}
Set{**F**}
Set{**G**}

A->A B->B C->C D->C E->E F->F G->G

***

2nd Traverse

Edge **A** to **E** with the weight of **3**.

Set{**AE**}
Set{**B**}
Set{**CD**}
Set{**F**}
Set{**G**}

A->A B->B C->C D->C E->A F->F G->G

***

3rd Traverse

Edge **C** to **F** with the weight of **4**.

Set{**AE**}
Set{**B**}
Set{**CDF**}
Set{**G**}

A->A B->B C->C D->C E->A F->C G->G

***

4th Traverse

Edge **B** to **E** with the weight of **5**.

Set{**ABE**}
Set{**CDF**}
Set{**G**}

A->**B**   **B**->**B**   **C**->**C**   **D**->**C**   **E**->**A**   **F**->**C**   **G**->**G**

5th Traverse

Edge **F** to **G** with the weight of **5**.

Set{**ABE**}
Set{**CDFG**}

A->**B**   **B**->**B**   **C**->**C**   **D**->**C**   **E**->**A**   **F**->**C**   **G**->**C**

6th Traverse

Edge **B** to **D** with the weight of **5**.

Set{**ABCDEFG**}

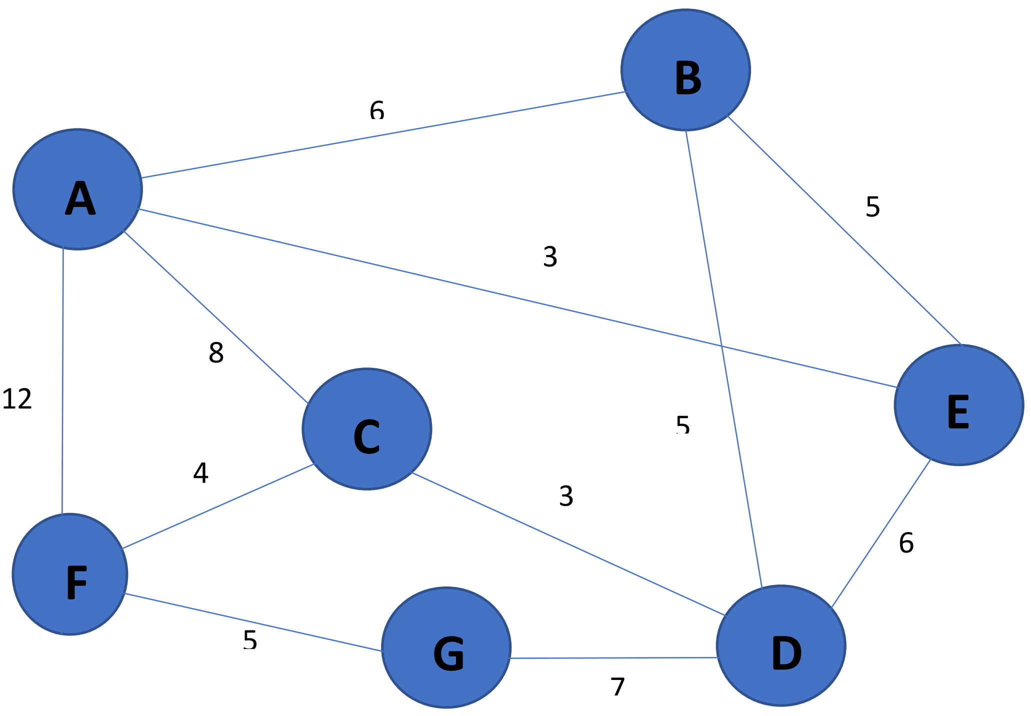
A->**B**   **B**->**B**   **C**->**B**   **D**->**C**   **E**->**A**   **F**->**C**   **G**->**C**

Kruskal’s Algorithm On Graph Diagram

Here is the graph that we will be using to find the Minimum Spanning Tree using Kruskal’s Algorithm.

We have to sort the edges 3,3,4,5,5,5,6,6,7,8,12

On every traverse we have to make sure whether by adding the new edge it will have a cycle or not.



Disjoint Sets: At the beginning we have as many sets as the number of vertices. When adding an edge, we merge two sets together. The algorithm stops when there is only a single set remains.

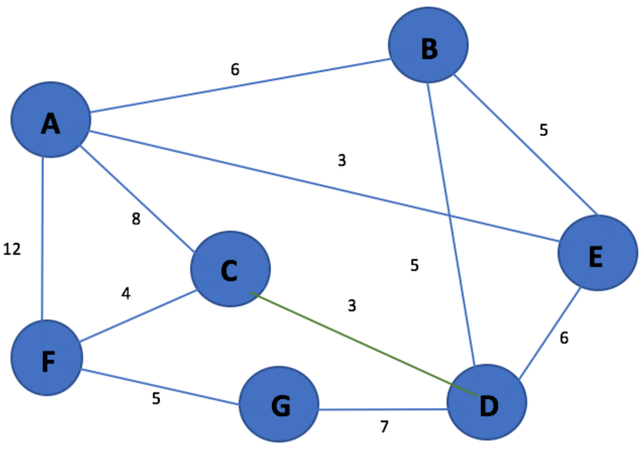
The **Disjoint Sets** in this graph are:

A B C D E F G

A B C D E F G

Since the weight to CD is 3 which is the lowest in the graph it is picked first, although there is another edge with the same weight, but in this graph presentation we pick **CD** with the weight of 3, it doesn't matter which we pick first.

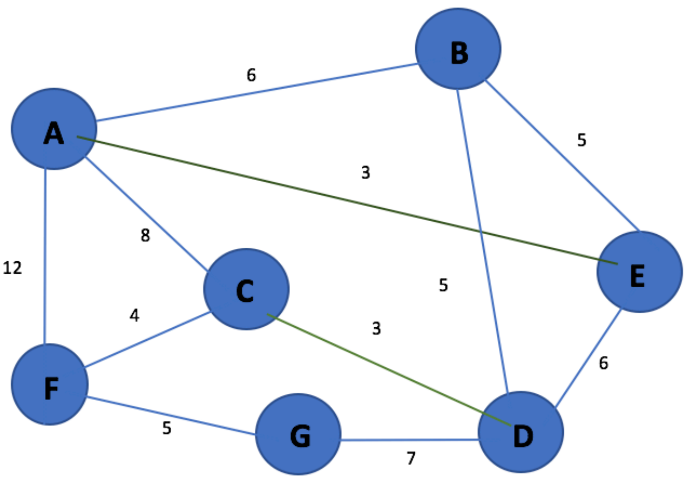
A B CD E F G



A B CD E F G

Now we pick the edge AE as it has the weight 3, this could have been picked first but it doesn't matter as long as a Vertex was never reached before.

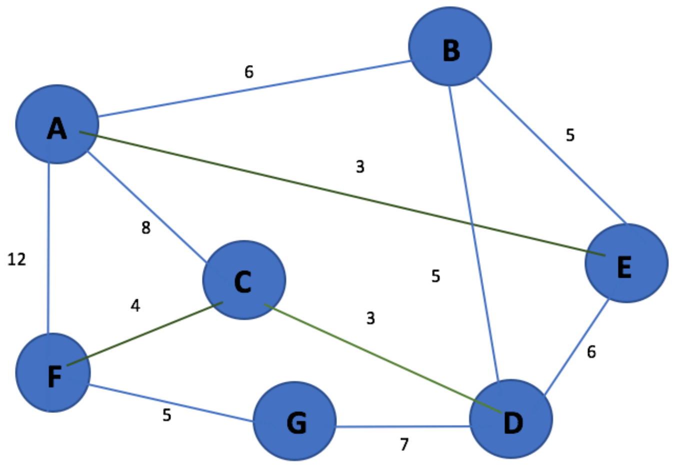
AE B CD F G



AE B CD F G

Now we pick the edge C to F as it has the next lowest edge weight and also there is no cycle as the nodes have not been visited yet.

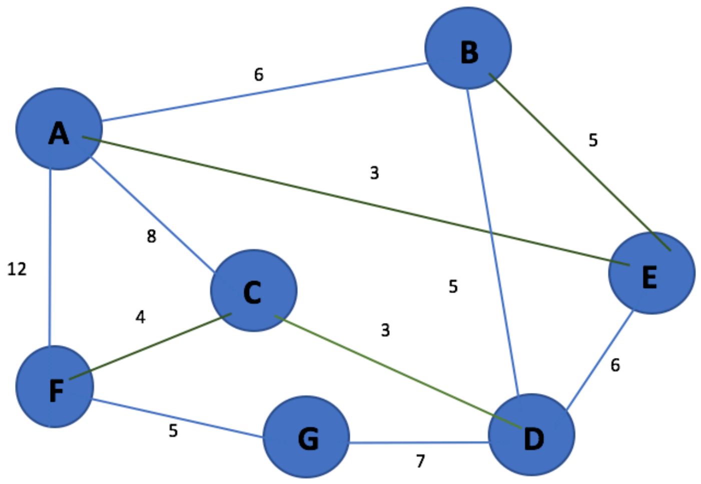
AE B CDF G



AE B CDF G

Now we pick the edge B to E as it has the next lowest edge weight and also there is no cycle as the nodes have not been visited yet.

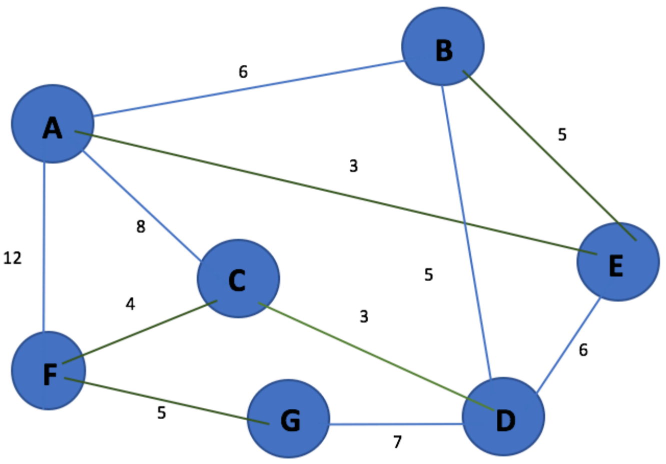
ABE CDF G



ABE CDF G

Now we pick the edge F to G as it has the next lowest edge weight and also there is no cycle as the nodes have not been visited yet.

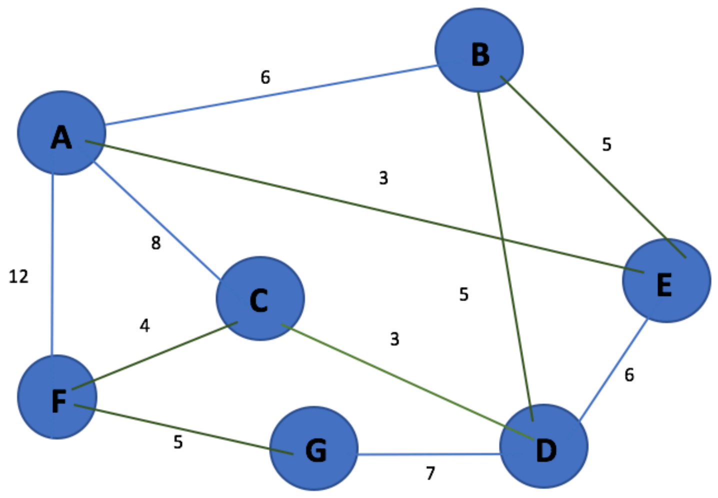
ABE CDFG



ABE CDFG

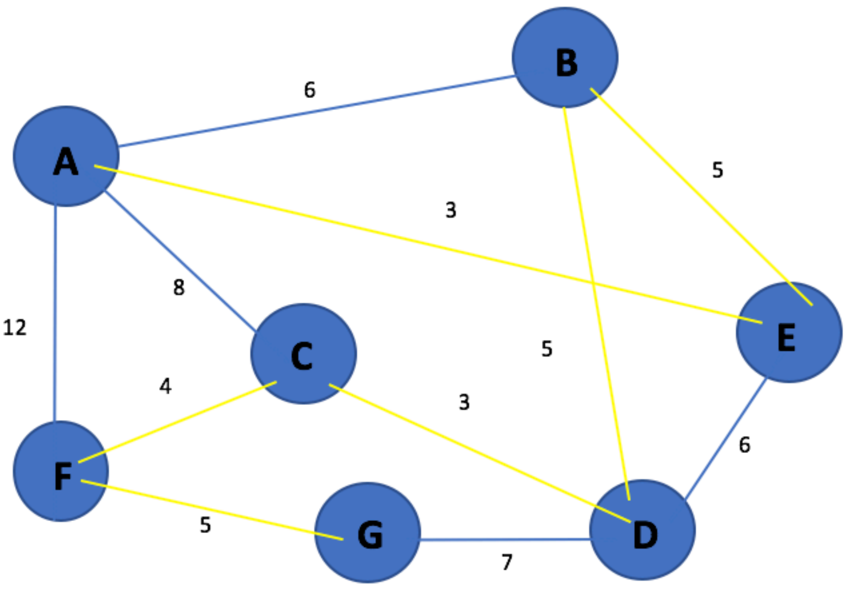
Now we pick the edge B to D as it has the next lowest edge weight and also there is no cycle as the nodes have not been visited yet. This is the last edge we pick as all the nodes have already been visited so our Minimum Spanning Tree is complete.

ABECDFG

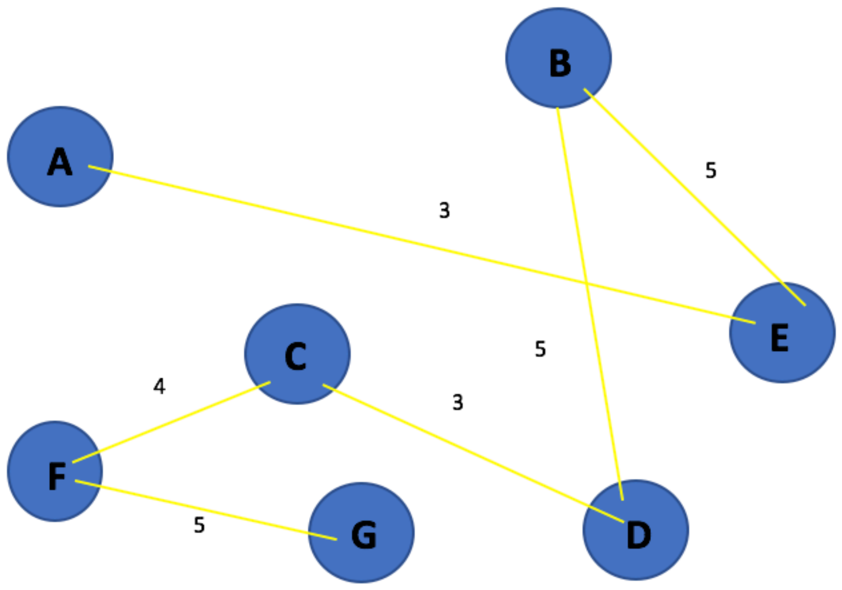


<u>Complete Minimum Spanning Tree</u>

Below is a complete Minimum Spanning Tree containing all the edges.



Below is a complete Minimum Spanning Tree without the edges that are not used.



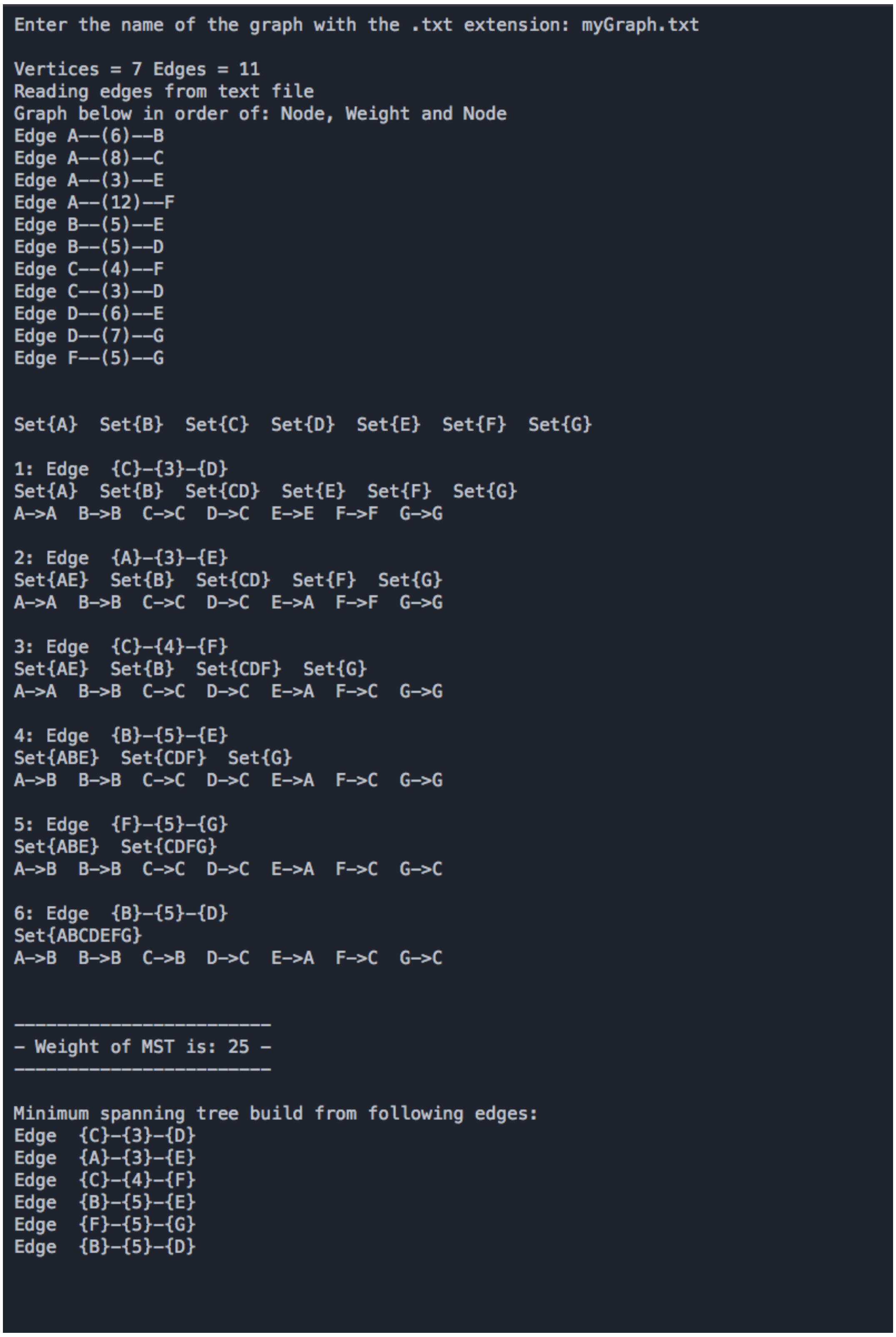
Kruskal’s Algorithm Code

```java
/*
    Name: Gabriel Grimberg.
    Course: DT228/2.
    Module: Algorithms & Data Structures.
    Type: Final Year Assignment.
    Code: Kruskal's Algorithm.
*/

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

class Edge
{
    public int u, v, wgt;

    public Edge()
    {
        u = 0;
        v = 0;
        wgt = 0;
    }

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

}//End of Edge Class.

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

    //Bottom up heap constructor
    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(i);
        }
    }

    private void siftDown(int k)
    {
        int v, j;
        v = h[k];

        while(2 * k <= N)
        {
            j = 2 * k;

            if((j < N) && (edge[h[j+1]].wgt < edge[h[j]].wgt))
            {
                ++j;
            }
            if(edge[v].wgt <= edge[h[j]].wgt)
            {
                break;
            }

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

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

}//End of Heap Class

/********************************************************************
*
                     *
*       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];

        for(int i = 1; i <= N; i++)
        {
            treeParent[i] = i;
        }
    }

    public int findSet(int vertex)
    {
        while(vertex != treeParent[vertex])
        {
            vertex = treeParent[vertex];
        }
        return vertex;
    }

    public void union(int set1, int set2)
    {
        int xRoot = findSet(set1);
        int yRoot = findSet(set2);

        treeParent[yRoot] = xRoot;
    }

    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);
    }

}//End of UnionFindSets Class.

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("\nVertices = " + parts[0] + " Edges = " + parts[1]);

    V = Integer.parseInt(parts[0]); //How many Vertices.
    E = Integer.parseInt(parts[1]); //How many Edges.

    //Create edge array
    edge = new Edge[E+1];

    //Read the edges
    System.out.println("Reading edges from text file");
    System.out.println("Graph below in order of: Node, Weight and Node");

    for(e = 1; e <= E; ++e)
    {
        line = reader.readLine();
        parts = line.split(splits);
        u = Integer.parseInt(parts[0]); //u is the First Node.
        v = Integer.parseInt(parts[1]); //v is the Node Connected to the first one.
        w = Integer.parseInt(parts[2]); //w is the weight between the 2 nodes.

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

        //Creating the edge object.
        edge[e] = new Edge(u, v, w);
    }
    System.out.println("\n");
}


/***************************************************************
*
                          *
*        Kruskal's minimum spanning tree algorithm            *
*
                          *
**************************************************************/
public Edge[] MST_Kruskal()
{
    int ei, i = 0;
    Edge e;
    int uSet, vSet;
    int totalWeight = 0;
    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;
                System.out.print("\n" + i + ": ");
                e.show();
                partition.union(uSet, vSet);
                partition.showSets();
                partition.showTrees();
                totalWeight += e.wgt;
            }
            else
            {
                System.out.print("\nIgnoring ");
                e.show();
            }

        }
        //Displaying the weight of the graph.
        System.out.println("\n");
        System.out.println("------------------------");
        System.out.println("- Weight of MST is: " + totalWeight + " -");
        System.out.println("------------------------");
        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.

public class KruskalTrees
{
    public static void main(String args[]) throws IOException
    {
        Scanner in = new Scanner(System.in);

        System.out.print("Enter the name of the graph with the .txt extension: ");

        //Entering the name of the text file with the graph in it.
        String fname = in.nextLine();

        //Creating the graph from the file.
        Graph g = new Graph(fname);

        //Getting the shortest patch using the algorithm.
        g.MST_Kruskal();

        //Displaying the shortest path made.
        g.showMST();
    }
}//End of KruskalTrees Class.
```

Screen Capture of Output



```
Enter the name of the graph with the .txt extension: myGraph.txt

Vertices = 7 Edges = 11
Reading edges from text file
Graph below in order of: Node, Weight and Node
Edge A--(6)--B
Edge A--(8)--C
Edge A--(3)--E
Edge A--(12)--F
Edge B--(5)--E
Edge B--(5)--D
Edge C--(4)--F
Edge C--(3)--D
Edge D--(6)--E
Edge D--(7)--G
Edge F--(5)--G


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

1: Edge  {C}-{3}-{D}
Set{A}  Set{B}  Set{CD}  Set{E}  Set{F}  Set{G}
A->A  B->B  C->C  D->C  E->E  F->F  G->G

2: Edge  {A}-{3}-{E}
Set{AE}  Set{B}  Set{CD}  Set{F}  Set{G}
A->A  B->B  C->C  D->C  E->A  F->F  G->G

3: Edge  {C}-{4}-{F}
Set{AE}  Set{B}  Set{CDF}  Set{G}
A->A  B->B  C->C  D->C  E->A  F->C  G->G

4: Edge  {B}-{5}-{E}
Set{ABE}  Set{CDF}  Set{G}
A->B  B->B  C->C  D->C  E->A  F->C  G->G

5: Edge  {F}-{5}-{G}
Set{ABE}  Set{CDFG}
A->B  B->B  C->C  D->C  E->A  F->C  G->C

6: Edge  {B}-{5}-{D}
Set{ABCDEFG}
A->B  B->B  C->B  D->C  E->A  F->C  G->C


------------------------
- Weight of MST is: 25 -
------------------------

Minimum spanning tree build from following edges:
Edge  {C}-{3}-{D}
Edge  {A}-{3}-{E}
Edge  {C}-{4}-{F}
Edge  {B}-{5}-{E}
Edge  {F}-{5}-{G}
Edge  {B}-{5}-{D}
```

Discussion and Learning Outcome

- I have learnt that Prim's Algorithm is significantly faster in the limit when you've got a really dense graph with more edges than vertices.
- Kruskal's Algorithm performs better in typical situations as in sparse graphs because it uses simpler data structures and we saw that on the same graph Kruskal's Algorithm found the Minimum Spanning Tree in 6 iterations compares to Prim's Algorithm which was 7 iterations, they both had the same total weight.
- Kruskal's Algorithm can have better performance if the edges can be sorted in linear time or the edges are already sorted.
- Prim's Algorithm is better if the number of edges to vertices is high and this is in dense graphs. So both of these algorithms have their own benefits.
- I have gained a massive understanding of how Heaps work and how they could be used in these Algorithms.
- Have done a lot of research on both of these Algorithms, wrote the code and performed several calculations with different types of graphs to understand what is happening.
- I have learnt how to construct graphs into a text file and how to make the program read in the graph and perform the algorithm on it so now I can make my own Boruvka's Algorithm, Dijkstra's Algorithm and so many more.
- Understood what these algorithms are used for such as a cable TV company laying cable to a new neighbourhood.