|  |
| --- |
|  |
| Algorithms and Data Structures Assignment |
| Kruskal’s Algorithm |
|  |
| **Student No: C11733511** |
| **Name: Anthony Hayden** |

|  |
| --- |
|  |

# Introduction

The following report will show the implementation of Kruskal’s Algorithm for finding the minimum spanning tree in a weighted connecting graph in C#.

Kruskal’s algorithm is a greedy algorithm, meaning it finds the subset of the edges that forms a tree to include all vertices within a graph, having a total minimum weight.

It’s complexity is O(E log V) and this report does not implement path compression ( where each *findSet()* would redirect traversed nodes to point to a single root ) .

## How it works

Edges, Vertices and weights are all read from a text file.

Each edge is stored in an array. An array of indices is also created to act as pointers to these edges.

Using a heap, the indices are sorted by their related edges weight, so that the lowest weight is at the top.

In the class *UnionFindSets* an array of integers called *treeParent* is created to store the parent of each tree. Each tree represents individual vertices, with its index representing the vertices’ and the data representing its parent. Initially, each tree is a parent/root unto itself, creating a number of disjoint sets equal to the number of vertices.

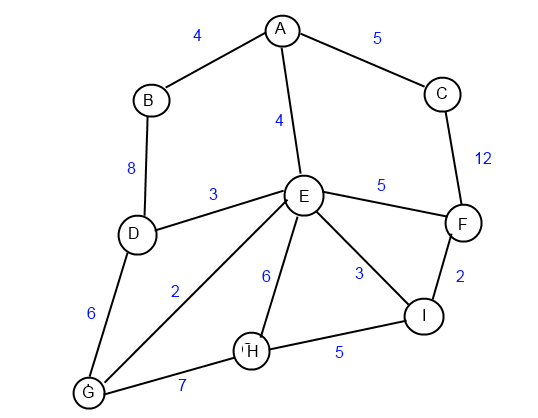
To construct the MST, the first edge with the lowest weight is taken. The root of both vertices are compared (*findSet(vertex)*) to see if they are equal. If not, then they are both in different sets and can be joined (*union(set1, set2*)). Within this method, we assign the root of one set to the root of the other. This edge is then removed from the heap and stored in an array of edges which will form the MST.

If they both are in the same set then we remove this edge’s index from the heap without adding it to our MST as this would create a cycle within the graph.

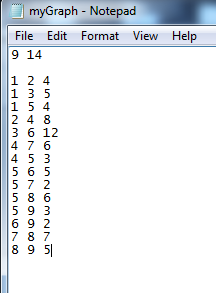
The heap is then readjusted and the process is repeated until the number of edges in the MST is one less than the number of vertices.

This leaves the array of edges, which make up our MST, containing the minimum total weight.

## Sample Graph



Text File Version:



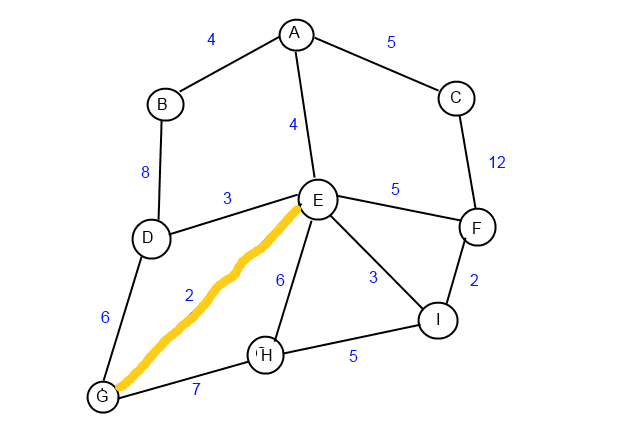
### Step 1

Initially, Sets are : {A}, {B}, {C}, {D}, {E}, {F}, {G}, {H}, {I}.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertice | A | B | C | D | E | F | G | H | I |
| Parent | A | B | C | D | E | F | G | H | I |

treeParent:

### Step 2

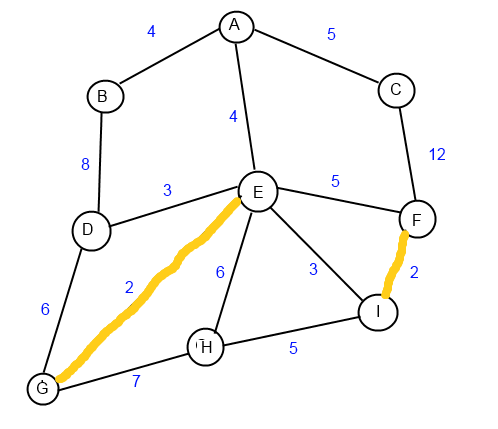


Sets: {A}, {B}, {C}, {D}, {E, G}, {F}, {H}, {I}.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertice | A | B | C | D | E | F | G | H | I |
| Parent | A | B | C | D | E | F | E | H | I |

treeParent:

### Step 3

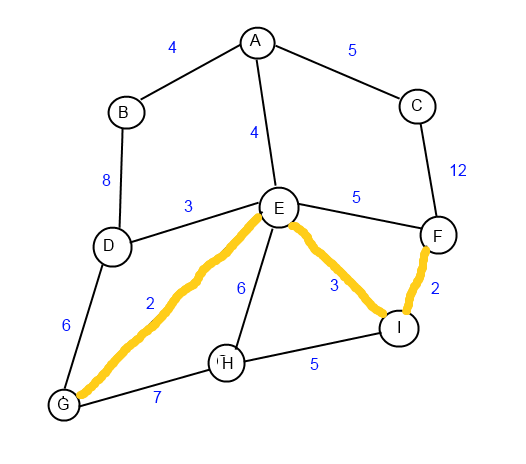


Sets: {A}, {B}, {C}, {D}, {E, G}, {F, I}, {H}.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertice | A | B | C | D | E | F | G | H | I |
| Parent | A | B | C | D | E | F | E | H | F |

treeParent:

### Step 4



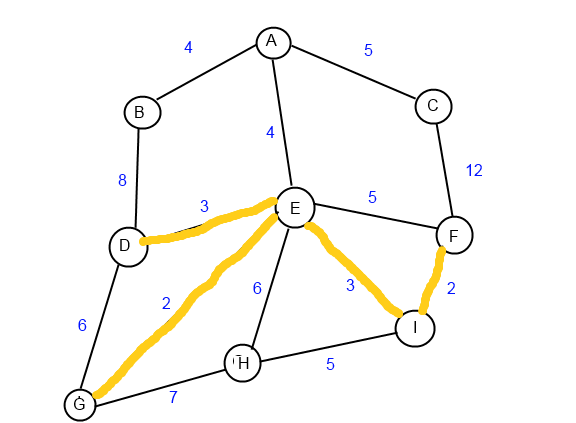
Sets: {A}, {B}, {C}, {D}, {E, F, G, I}, {H}.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertice | A | B | C | D | E | F | G | H | I |
| Parent | A | B | C | D | E | E | E | H | F |

treeParent:

*E is now the parent of F, as F was previously the parent in set {F, I} and E was the parent of set {E, G}*

### Step 5

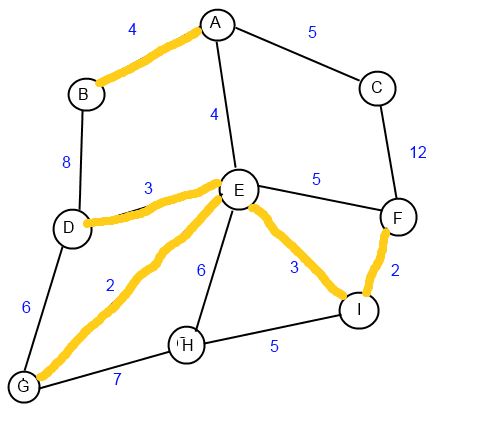


Sets: {A}, {B}, {C}, {D, E, F, G, I}, {H}.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertice | A | B | C | D | E | F | G | H | I |
| Parent | A | B | C | D | D | E | E | H | F |

treeParent:

### Step 6

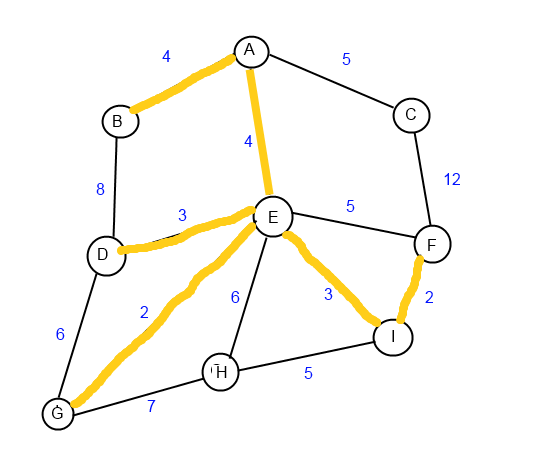


Sets: {A, B}, {C}, {D, E, F, G, I}, {H}.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertice | A | B | C | D | E | F | G | H | I |
| Parent | A | A | C | D | D | E | E | H | F |

treeParent:

### Step 7

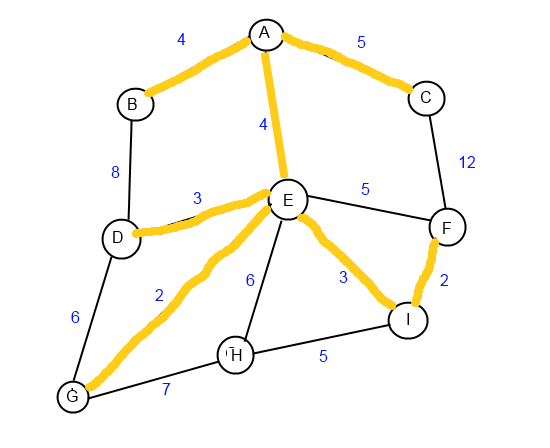


Sets: {A, B, D, E, F, G, I}, {C}, {H}.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertice | A | B | C | D | E | F | G | H | I |
| Parent | A | A | C | A | D | E | E | H | F |

treeParent:

### Step 8

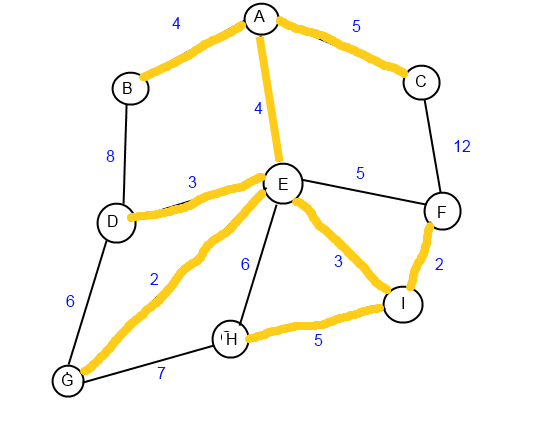


Sets: {A, B, C, D, E, F, G, I}, {H}.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertice | A | B | C | D | E | F | G | H | I |
| Parent | A | A | A | A | D | E | E | H | F |

treeParent:

### Step 9



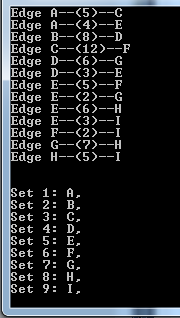
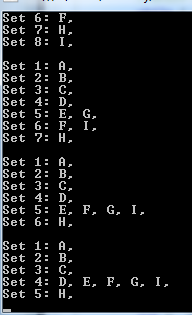
Sets: {A, B, C, D, E, F, G, H, I}.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertice | A | B | C | D | E | F | G | H | I |
| Parent | A | A | A | A | D | E | E | I | F |

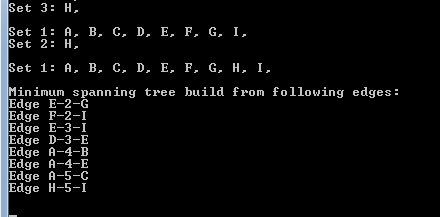
treeParent:

# Screen Captures of Program Running

Initial values read from file Union of Sets

Final Output



# Conclusion/Reflection

While a simple concept to grasp, the coding of this algorithm was a little tricky. A particular stumbling block was the trying to figure out to union the trees which were formed. As it turned out, simply replacing the root of one with the parent of another worked fine and the size of trees did not matter as I had initially thought.

A useful algorithm for perhaps solving network problems and similar in function to Primms, however it would appear that Primm’s would be more efficient the higher the Vertice to Edge ratio is.