Kruskal's Minimum Spanning Tree

CSE 4614 Technical Report Writing

Shakleen Ishfar
ID: 160041029

Department of Computer Science and Engineering
Islamic University of Technology
Bangladesh
04 August, 2019

Contents

List of Figures

List of Tables

1 Introduction

Kruskal's algorithm is a *minimum-spanning-tree (MST)* algorithm which finds an edge of the least possible weight that connects any two trees in the forest. The algorithm was written by *Joseph Kruskal* and published in 1956 at the *Proceedings of American Mathematical Society.*

2 Problem Formulation

Before getting to the problem statement there are some terminologies which need to be addressed.
Spanning tree: A *spanning tree* of a connected and undirected graph is a sub-graph which is a tree connecting all the vertices together. A single graph can have many different spanning trees. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.
Minimum Spanning Tree: A *minimum spanning tree (MST)*, for a weighted, connected and undirected graph, is a spanning tree with weight less than or equal

to the weight of every other spanning tree.
Problem Statement: Given, a weidghted, connected and undirected graph having V vertices. Kruskal's algorithm will return a graph which will be tree having $V-1$ edges. The total weight of the MST, W is supposed to be equal or less than all other possible spanning trees. Here W is the following:

$$W = \sum_{i=1}^{V-1} W_i \tag{1}$$

where W_i is the weight of the i-th edge of the MST.

3 General Solution

Before jumping into the steps of Kruskal's Algorithm some basic knowledge is required to understand the procedure of the algorithm. These are discussed here.

3.1 Union-Find by Rank

The union by rank algorithm is used in Kruskal's MST algorithm for cycle detection. A short explanation of Union by Rank is given here for convenience.
Rank: A rank is basically the maximum depth value of a tree.
Union-Find Algorithm: In Union-Find algorithm, a tree having a lower rank is attached to a tree having a higher rank. In other words, the shorter tree is attached to the larger tree. What this algorithm does is that all the vertices in a tree is thought of as being under the same set. Let's say we have a tree with vertices 1, 2, 3 and 4. Then all the vertices of this tree will be seen as being under the same set. Again, let there be two trees respectively having vertices 1, 2, 3 and 4, 5. Then the vertices of each tree will belong to a separate set. Each set is given a value, typically one of the values of the vertices. So a set having 1, 2, 3 might be assigned the value of either 1 or 2 or 3 and the set with members 4, 5 might be given the value of 4 or 5. Now there are essentially 3 operations in this algorithm.

1. **Union(u, v):** Unites the sets u and v. So merging sets 1, 2, 3 and 4, 5 will result in 1, 2, 3, 4, 5.
2. **FindParent(u):** FInds the parent node of the set u. Meaning, the assigned value for that set.
3. **MakeSet(u):** Creates a set with one member u and assigns the set to have u as it's assigned value.

3.2 Union by Rank in Cycle Detection

Let's say that initially a vertice will belong to a set that is assigned a value of the vertice value. A group of connected vertices will be under the same set,

represented by a value of one of the member vertices (Following the rule of Union-Find algorithm). We shall call this specific vertice as the parent vertice of this group of vertices. Now, when we add a vertice to this graph of connected vertices through an edge we are essentially adding the new vertice set to the group vertice set. Simply, we are merging their set. This is done using Union(u, v) operation. Now, a cycle will form if, we try to add a vertice through an edge that is already in the set of the group of vertice set. So, checking for the set assigned value using FindParent(u) is enough to detect a cycle in the graph. This is how we use the Union-Find by Rank algorithm in Kruskal to detect cycles. There is a much more efficient implementation of Union-Find algorithm that implements path compression to reduce time complexity even more. However, discussion of this implementation is outside the scope of this tutorial.

3.3 Procedure

This is a Greedy Algorithm. Here the Greedy Choice is to pick the smallest weight bearing edge that does not form a cycle in the MST constructed so far. The steps to finding the MST of a graph is written below:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it. Cycle detection is done using *Union by Rank and Path Compression* algorithm.
3. Repeat step 2 until there are $V-1$ edges in the spanning tree.

3.4 Kruskal's Algorithm Pseudocode

4 Example

To better understand Kruskal's algorithm let us take a look at an example. The example graph is shown in figure 1 We can see that it is a bidirectional weighted graph. So Kruskal's algorithm can be applied to it to find it's MST. Let us simulate the running of the algorithm and see what happens in each step to get a good grasp of how it is working.

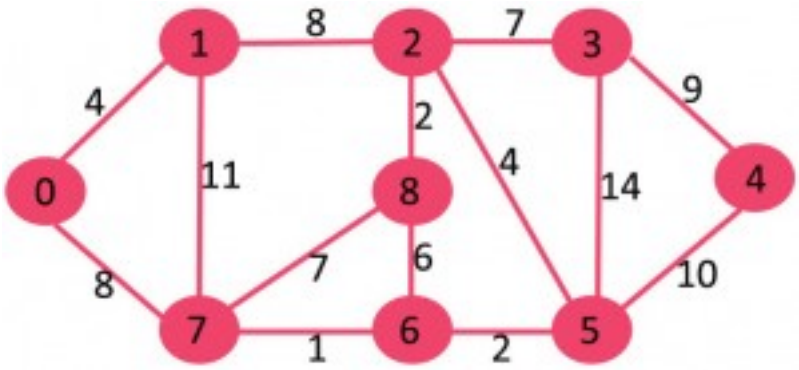
The first step is to sort all the edges in non-decreasing order of weight. Any sorting algorithm can be used for this purpose. Sorting the edges yields the following ordering of edges shown in table 1.

Now that our edges have been sorted in non-decreasing order we can begin to include the edges to form our MST. The rule to remember is that we include an edge to our MST only if it doesn't create a cycle. If the edge causes a cycle we will not include that edge. We shall do this as long as we don't have $V-1$ number of edges. Recall V is the number of vertices present in a graph. For our example graph of figure 1 $V = 9$. So we will keep including edges until there

Algorithm 1 Kruskal's algorithm

```
1:  procedure KRUSKAL(G)
2:      A = ∅                                   ▷ Will store MST to be produced
3:      for v ∈ G.V do                          ▷ Each vertice originally in it's own set
4:          MakeSet(v)                          ▷ Makes vertice v a set of v
5:      end for
6:      Sort G.E by non-decreasing weight       ▷ Sorts edges of graph G
7:      for (u, v) ∈ G.E do                     ▷ For each edge in graph G
8:          if FindSet(u) ≠ FindSet(v) then     ▷ Check cycle formulation
9:              A = A ∪ (u, v)                  ▷ Add to MST graph
10:             Union(FindSet(u), FindSet(v))   ▷ Unite u and v parent sets
11:         end if
12:     end for
13:     return A
14: end procedure
```

Figure 1: Example graph



are exactly 8 edges in our MST. So let's go through the sorted list and try to formulate our graph one edge at a time, as shown in table 2.

Table 2: Going through the graph one edge at a time

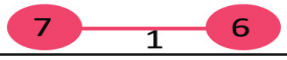
Begin of Table			
Edge	Cycle?	Verdict	Figure
7-6	No	Include	

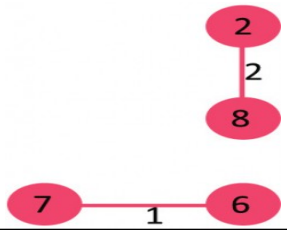
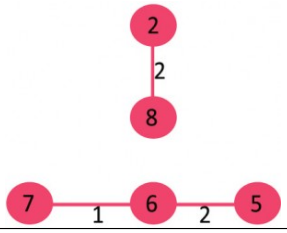
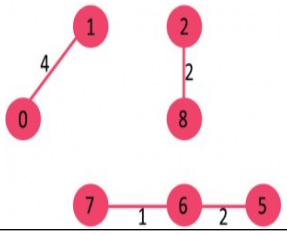
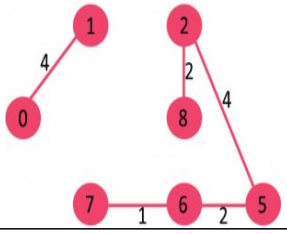
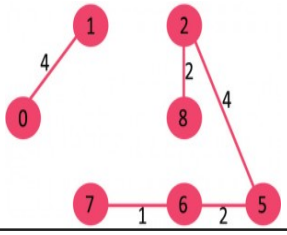
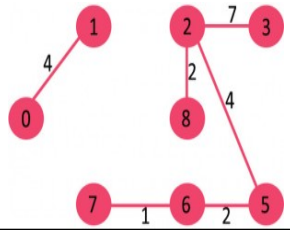
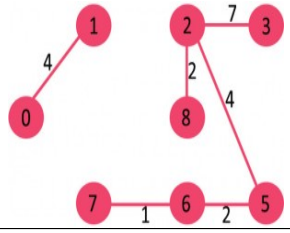
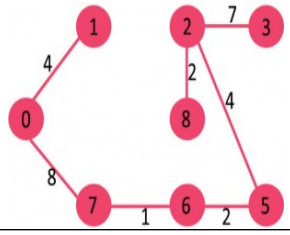
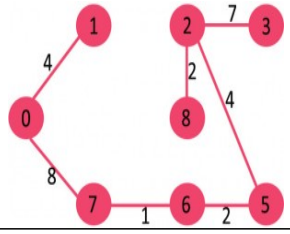
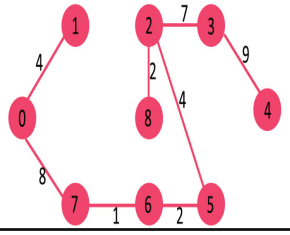
Table Continued			
Edge	Cycle?	Verdict	Figure
8-2	No	Include	
6-5	No	Include	
0-1	No	Include	
2-5	No	Include	
8-6	Yes	Discard	

Table Continued			
Edge	Cycle?	Verdict	Figure
2-3	No	Include	
7-8	Yes	Discard	
0-7	No	Include	
1-2	Yes	Discard	
3-4	No	Include	
End of Table			

The final figure in table 2 is our MST. As we'd already added $V - 1 = 8$ edges, we did not need to go through all the edges in table 1. This is how

Table 1: Result of sorting edges of example graph

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Kruskal's algorithm can be used to process a graph and build an MST.

5 Complexity Analysis

Kruskal's algorithm has a complexity of O(VlogV). We get this complexity in the following way:

- To sort the edges in non-decreasing order of weight, we can use merge or quick sort. This takes O(ElogE) time.
- We iterate through all edges E and apply find-union algorithm. The find and union operations can take at most O(logV) time.
- To perform both these operations complexity becomes O(ElogE + ElogV).
- The value of E can be at most O(V^2), so O(logV) are O(logE) same. Thus, ultimately time complexity becomes O(VlogV).

6 C++ Implementation

```
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>

using namespace std;

```

```
// Edge information
struct Edge
{
    int Vertice1;   // Vertice on one side
    int Vertice2;   // Vertice on opposite side
    int Weight;     // Weight of the edge
};

// Vertice information
struct Vertice
{
    int Value;      // Value of the vertice
    int Rank;       // Rank of the vertice
    Vertice *Parent;// Parent of the vertice
};

// For sorting purpose
int Comp(const void* a, const void *b)
{
    Edge *a1 = (Edge*)a;
    Edge *b1 = (Edge*)b;
    return (a1->Weight < b1->Weight);
}

class KruskalMST
{
public:
    void MakeEdge(int Vertice1, int Vertice2, int Weight);
    // For Merging sets
    bool Union(int value1, int value2);
    // Making the minimum spanning tree. Return total weight of MST
    .
    int GenerateTree();
    vector <Edge*> EdgeStoreMST;                // Storing edges of
     MST
private:
    vector <Edge*> EdgeStore;                   // Storing edges
    map <int, Vertice*> VerticeStore;           // Store the
    vertices
    vector <Edge*>::iterator itr;               // Used for
    iteration
    map <int, Vertice*>::iterator itr1, itr2;   // Used for
    iteration
    Vertice *V1, *V2;   // Used for processing purposes
    Edge *E;            // Used for processing purposes
    int TotalWeight;    // Store the Total weight of MST
    int EdgeCount;      // No of edges added to MST
    int TotalVertices=0;// Total number of vertices created
    int TotalEdges=0;   // Total number of edges created

    // For making a set with one element
    void MakeSet(int value);
    // Find parent. Applies path compression.
    Vertice* FindParent (Vertice *V);
};

void KruskalMST::MakeEdge(int Vertice1, int Vertice2, int Weight)
```

```
{
    Edge *New = new Edge;        // Create new edge
    New->Vertice1 = Vertice1;    // Add vertice 1 value
    MakeSet(Vertice1);           // Create set for vertice 1
    New->Vertice2 = Vertice2;    // Add vertice 2 value
    MakeSet(Vertice2);           // Create set for vertice 2
    New->Weight = Weight;        // Add weight of the edge
    EdgeStore.push_back(New);    // Store the edge
    ++TotalEdges;                // Increase total edges
    return;
}

void KruskalMST::MakeSet(int value)
{
    // Check if vertice already exists or not
    if (VerticeStore[value])
        return; // It does. So do nothing.

    Vertice *New = new Vertice; // Creating new set
    New->Parent = New;          // Parent is the vertice itself
    New->Rank = 0;              // Rank is initially zero
    New->Value = value;         // Setting vertice value
    VerticeStore[value] = New;  // Storing into map for easy access
     later
    ++TotalVertices;            // Increase total number of
    vertices

    return;
}

Vertice* KruskalMST::FindParent (Vertice *V)
{
    // Getting parent of V
    Vertice *Parent = V->Parent;

    /* Check if V is parent of itself.
       If so then it's the root node or identity value holder.
       Otherwise find the parent and update through path
    compression. */
    if (Parent->Value != V->Value)
        V->Parent = Parent = FindParent(Parent);

    // Return the parent.
    return Parent;
}

bool KruskalMST::Union(int value1, int value2)
{
    // Find the nodes of value 1 and value 2
    itr1 = VerticeStore.find(value1);
    itr2 = VerticeStore.find(value2);

    // Find the parent of the nodes
    V1 = FindParent(itr1->second);
    V2 = FindParent(itr2->second);

    // Check if the vertices belong to the same set
```

```
114     if (V1->Value == V2->Value)
115         return false; // They are of the same set. So do nothing.
116 
117     // Union by rank. Higher rank becomes parent of lower rank.
118     // Rank increases if both ranks are equal
119     if (V1->Rank > V2->Rank)
120         V2->Parent = V1->Parent;
121     else
122     {
123         V1->Parent = V2->Parent;
124         V2->Rank++;
125     }
126     return true;
127 }
128 
129 int KruskalMST::GenerateTree()
130 {
131     // Initialize values
132     TotalWeight = EdgeCount = 0;
133 
134     // sort the edge list.
135     sort(EdgeStore.begin(), EdgeStore.end(), Comp);
136 
137     // Go through each Edge until n-1 edges have been merged.
138     for(itr = EdgeStore.begin(); itr != EdgeStore.end(); ++itr)
139     {
140         // n-1 edges added? If so break. MST already achieved.
141         if (EdgeCount == TotalVertices-1)
142             break;
143 
144         E = (*itr);
145 
146         // Check if union is possible. If possible then do the
    following.
147         if (Union(E->Vertice1, E->Vertice2))
148         {
149             EdgeStoreMST.push_back(E);  // Include the edge in MST
150             ++EdgeCount;                // Increase edge count
    added to MST
151             TotalWeight += E->Weight;   // Increase the total
    weight of MST
152         }
153     }
154 
155     return TotalWeight;
156 }
157 
158 int main()
159 {
160     KruskalMST KMST;
161 
162     KMST.MakeEdge(1, 2, 1);
163     KMST.MakeEdge(1, 3, 3);
164     KMST.MakeEdge(2, 3, 2);
165     KMST.MakeEdge(2, 4, 1);
166     KMST.MakeEdge(3, 4, 1);
167     KMST.MakeEdge(3, 5, 4);
```

```
168     KMST.MakeEdge(5, 6, 1);
169     KMST.MakeEdge(1, 6, 3);
170 
171     cout << KMST.GenerateTree() << endl;
172     return 0;
173 }
```