B.Sc. in Computer Science and Engineering Thesis

Study on Minimum Spanning Tree Algorithms

Submitted by

Afsana Khan
201414066

Afrida Anzum Aesha
201414067

Juthi Sarker Aka
201414080

Supervised by

Dr. M. Kaykobad

Professor

Department Of Computer Science & Engineering

Bangladesh University of Engineering and Technology

Department of Computer Science and Engineering
Military Institute of Science and Technology

December 2017

CERTIFICATION

This thesis paper titled **"Study on Minimum Spanning Tree Algorithms "**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in December 2017.

Group Members:

1. Afsana Khan
2. Afrida Anzum Aesha
3. Juthi Sarker Aka

Supervisor:

Dr. M. Kaykobad
Professor, CSE Department
Bangladesh University of Engineering and Technology

CANDIDATES' DECLARATION

This is to certify that the work presented in this thesis paper, titled, **"Study on Minimum Spanning Tree Algorithms"**, is the outcome of the investigation and research carried out by the following students under the supervision of Dr. M. Kaykobad, Professor, CSE Department, Bangladesh University of Engineering & Technology.

It is also declared that neither this thesis paper nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Afsana Khan
201414066

Afrida Anzum Aesha
201414067

Juthi Sarker Aka
201414080

ACKNOWLEDGEMENT

We are thankful to Almighty Allah for his blessings for the successful completion of our thesis. Our heartiest gratitude, profound indebtedness and deep respect go to our supervisor, **Dr. M. Kaykobad**, Professor, CSE Department, Bangladesh University of Engineering & Technology, for his constant supervision, affectionate guidance and great encouragement and motivation. His keen interest on the topic and valuable advices throughout the study was of great help in completing thesis.

We are especially grateful to the Department of Computer Science and Engineering (CSE) of Military Institute of Science and Technology (MIST) for providing their all out support during the thesis work.

Finally, we would like to thank our families and our course mates for their appreciable assistance, patience and suggestions during the course of our thesis.

Dhaka
December 2017

1. Afsana Khan
2. Afrida Anzum Aesha
3. Juthi Sarker Aka

ABSTRACT

A minimum spanning tree is a special kind of tree that minimizes the lengths (or weights) of the edges of the tree. Minimum spanning trees (MSTs) have long been used in data mining, pattern recognition and machine learning. Finding an efficient MST in various types of networks is a well-studied problem in theory and practical applications. A number of efficient algorithms have been already developed for this problem. In this thesis paper, we have proposed a new approach for finding MST, which combines the idea of eliminating edges before sorting. By this time, some of the edges are omitted before we run sorting algorithm on all edges. After eliminating edges, a forest is created. It can reduce the runtime for sorting edges. This algorithm gives better performance when the graph is not too dense.

Contents

List of Figures

List of Tables

LIST OF ABBREVIATION

MST : Minimum Spanning Tree

CHAPTER 1
INTRODUCTION

1.1 Problem Definition

The minimum-weight spanning tree problem is one of the most well-known problems of combinatorial optimisation. The problem of MST is to find a spanning tree such that the sum of weights is minimized. It is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. If there are n vertices in the graph, then each spanning tree has $n - 1$ edges. There may be several minimum spanning trees of the same weight. Any edge-weighted undirected graph has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components. The importance of this problem derives from its direct applications in the design of computer, communication, transportation, power and piping networks; from its appearance as part of solution methods to other problems to which it applies less directly such as network reliability, clustering and classification problems and from its occurrence as a sub-problem in the solution of other problems like the travelling salesman problem, the multi-terminal flow problem, the matching problem and the capacitated MST problem.

1.2 Literature Review

Efficient solution techniques had been trying to invented for many years. In the last two decades asymptotically faster algorithms have been invented. The first fully realized algorithm was devised by Boruvka[1].It was a classical MST algorithm. Its purpose was an efficient electrical coverage of Moravia. The algorithm proceeds in a sequence of stages. In each stage, called Boruvka step,. Each Boruvka step takes linear time. Since the number of vertices is reduced by at least half in each step, Boruvka's algorithm takes $O(m \log n)$ time. Prim's algorithm [2], which was invented by Jarnik in 1930 and rediscovered by Prim in 1957 .Basically, it grows the MST(T) one edge at a time. Initially, T contains an arbitrary vertex. In each step, T is augmented with a least-weight edge (x, y) such that x is in T and y is not yet in T. By the Cut property, all edges added to T are in the MST. Its run-time is either $O(m \log n)$ or $O(m + n \log n)$, depending on the data-

structures used. One of classic algorithm is Kruskal's algorithm [3], 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.which also takes$o(m \log n)$. Another commonly used algorithm is reverse-delete algorithm, which is reverse of Kruskal's algorithm. Its runtime is $O(m \log n(\log \log n)^3)$. All these four are greedy algorithm. After that, an survey paper by Graham and Hell [4] describes the history of the problem up to 1985. In the last two decades, faster algorithms have been found. An algorithm of Gabow [5] used Fibonacci heap to find MST with a running time of $O(mlog\beta(m,n))$ on a graph of n vertices and m edges. Here,$\beta\big(m,n\big) = min\big\{i|\log^i n \leq m/n\big\}$. This and earlier algorithms used as a computational model the sequential unit-cost random-access machine with the restriction that the only operations allowed on the edge weights are binary comparisons. Fredman and Willard [6] considered a more powerful model that allows bit manipulation of the binary representations of the edge weights. In this model, they were able to devise a linear-time algorithm. A problem related to finding minimum spanning trees is that of verifying that a given spanning tree is minimum. Tarjan [7] gave a verification algorithm running in $O\big(m\alpha\big(m,n\big)\big)$ time, where a is a functional inverse of Ackerman's function.Chazelle [1997] presented an MST algorithm based on the Soft Heap[14] having complexity $O\big(m\alpha\big(m,n\big)\log\alpha\big(m,n\big)\big)$.Later, Komlos [1985] showed that a minimum spanning tree can be verified in $O(m)$ binary comparisons of edge weights, but with nonlinear overhead to decide which comparisons to make. Dixon [8] [1992]combined these algorithms with a table lookup technique to obtain an $O(m)$ time verification algorithm. King [1993] recently obtained a simpler $O(m)$ time verification algorithm that combines ideas of Borka, Kom16s, Dixon, Rauch, and Tarjan[9]. The representation of candidate solutions and the variation operators are fundamental design choices in an evolutionary algorithm (EA).R. Raidl[10] used special initialization, crossover, and mutation operators are used to generate new, always feasible candidate solutions. Within the EA, a candidate spanning tree is simply represented by its set of edges.

1.3 Objectives and Scopes of the Thesis

The scope of thesis is the areas covered in the research. This part of research paper is telling exactly what was done and where the information that was used-specifically came from. Several researchers have tried to find more computationally-efficient minimum spanning tree algorithms.To solve minimum spanning tree problem, Prim's algorithm is widely used where the run-time is $O(E \log E)$. It is significantly faster in the limit when dense graph with many more edges than vertices is used. Kruskal's algorithm, which takes $O(E \log E)$ time. It performs better in typical situations (sparse graphs) because it uses simpler data structures.

Also, at any instant, Prim's algorithm gives connected component as well as it works only on

connected graph in its basic form but Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components without any modification. Prim's can also be easily modified to work for disconnected graph, but Kruskal works without modifications.
In this thesis paper, we introduce a new algorithm to find minimum spanning tree which reduces the complexity of sorting edges in Kruskal's algorithm. In Kruskal's algorithm sorting edges takes $O(E \log E)$. In our algorithm sorting edges takes $O(m \log m)$ as edges are decreased here from 1st iteration. It can function on disconnected graphs too.
The aim of this thesis paper is to find out the conditions under which the existing minimum spanning tree's run time will be optimized with minimum cost.

1.4 Thesis Organization

In Chapter 2, existing minimum spanning trees, their algorithms, simulation and complexity is shown.

In chapter 3, our proposed minimum spanning tree has been discussion with algorithm. Also the steps of simulation has been described elaborately.

In chapter 4, we have experimented some cases. Our proposed minimum spanning tree and existing minimum spanning tree's run time has been calculated to show the complexity.

At last in the conclusion, we concluded our thesis work.

CHAPTER 2
MINIMUM SPANNING TREE ALGORITHMS

A minimum spanning tree (MST) or minimum weight spanning tree 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. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. There are many algorithms to find MST.

2.1 Boruvka's Algorithm

Borůvka's algorithm is used to find the minimum spanning tree in the given graph. The algorithm was developed in 1926 by Czech mathematician Otakar Borůvka, when he was trying to find an optimal routing for the electrical grid in Moravia. Because the algorithm was later several times reinvented (among others by M. Sollin), the procedure is also sometimes called Sollin's algorithm.

2.1.1 Algorithm

The Boruvka's algorithm is based on merging of disjoint components. At the beginning of the procedure each vertex is considered as a separate component. In each step the algorithm connects (merges) every component with some other using strictly the cheapest outgoing edge of the given component (every edge must have a unique weight). This way it is guaranteed that no cycle may occur in the spanning tree.

Algorithm 1 Boruvka's Algorithm

1: $T \leftarrow \emptyset$
2: $Comp = v1, v2, ..., vn \ \ (where$ n $is\ the\ size\ of$ V and vi∈V)
3: **while** size of Comp 6 = 1 **do**
4: **for** each set $c \in Comp$ **do**
5: find the lowest weight edge e from c to Comp c
6: add edge e to T
7: **end for**
8: construct any sets that connect via edges in T
9: **end while**
10: **return** T

2.1.2 Simulation

Let us understand the algorithm with below example.

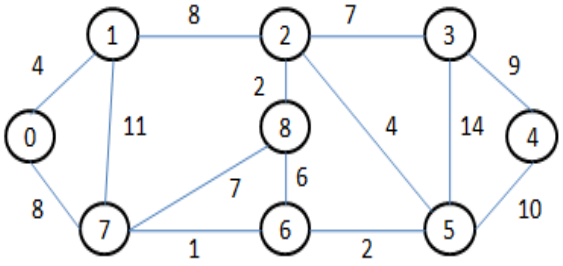

Figure 2.1: Boruvka's Graph(a)

Initially MST is empty. Every vertex is singe component in the diagram. For every component, cheapest edge will be found that connects it to some other component. The cheapest edges are highlighted with black color. Now MST becomes 0-1, 2-8, 2-3, 3-4, 5-6, 6-7.

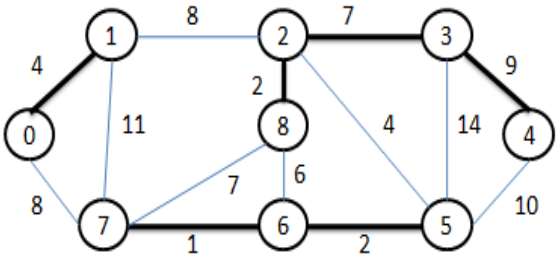

Figure 2.2: Boruvka's Graph(b)

After above step, components are [0,1, 2,3,4,8, 5,6,7]. The components are encircled with blue color.

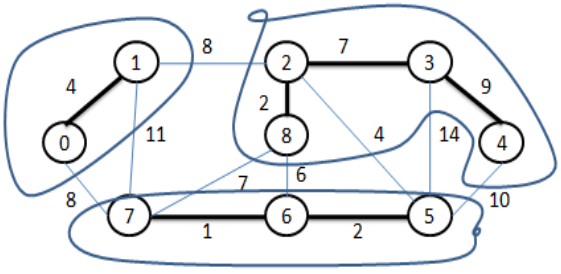

Figure 2.3: Boruvka's Graph(c)

We again repeat the step, for every component, find the cheapest edge. The cheapest edges are highlighted with black color. Now MST becomes 0-1, 2-8, 2-3, 3-4, 5-6, 6-7, 1-2, 2-5

At final stage, there is only one component 0, 1, 2, 3, 4, 5, 6, 7, 8 which has all edges. Since there is only one component left, we stop and return MST.

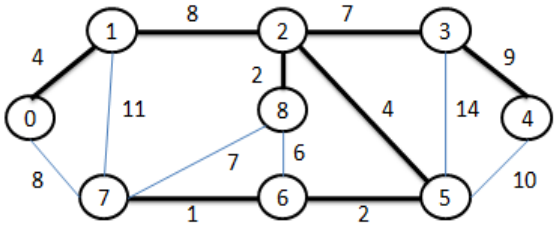

Figure 2.4: Boruvka's Graph(d)

2.1.3 Complexity of Boruvka's Algorithm

Since every step connects at least half of the currently unconnected components, it is clear that the algorithm terminates in $O(\log_2(|V|))$ steps. Because each step takes up to $O(|E|)$ operations to find the cheapest outgoing edge for all the components, the asymptotic complexity of Borůvka's algorithm is $O(|E|.\log_2(|V|))$

2.2 Prim's Algorithm

Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST. A group of edges that connects two set of vertices in a graph is called cut in graph theory. So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

2.2.1 Algorithm

The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Algorithm 2 Prim's Algorithm (G, w, r)

1: $Q \leftarrow V[G]$
2: **for** each $u \in Q$ **do**
3: $key[u] \leftarrow \infty$
4: **end for**
5: $key[r] \leftarrow 0$
6: $p[r] \leftarrow NULL$
7: **while** Q not empty **do**
8: $u \leftarrow ExtractMin(Q)$
9: **for** each $v \in Adj[u]$ **do**
10: **if** $v \in Q$ and $w(u, v) < key[v]$) **then**
11: $p[v] \leftarrow u$
12: $key[v] \leftarrow w(u, v)$
13: **end if**
14: **end for**
15: **end while**

2.2.2 Simulation

Let us understand the algorithm with below input graph. The set mstSet is initially empty

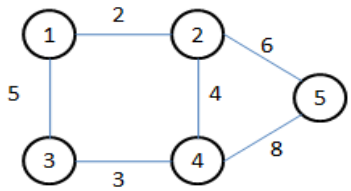

Figure 2.5: Prim's Graph (a)

and keys assigned to vertices are 0, INF, INF, INF, INF where INF indicates infinite.

Now pick the vertex with minimum key value. The vertex 1 is picked, include it in mstSet. So mstSet becomes 1.

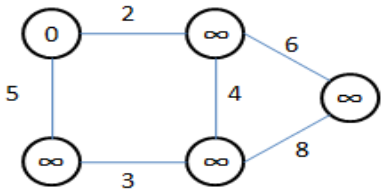

Figure 2.6: Prim's Graph (b)

Adjacent vertices of 1 are 2 and 3. The key values of 2 and 3 are updated as 2 and 5.
The minimum vertex 2 is picked and added to mstSet. So mstSet now becomes 1, 2. Update

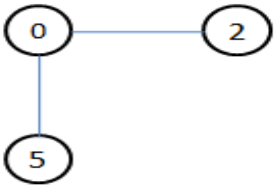

Figure 2.7: Prim's Graph (c)

the key values of adjacent vertices of 2. The key value of vertex 2 becomes 4 and 6.

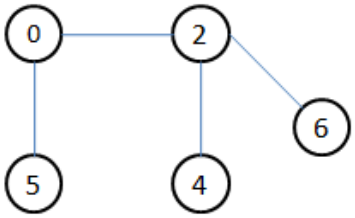

Figure 2.8: Prim's Graph (d)

Now vertex 4 is picked. So mstSet now becomes 1, 2, 4. The key value of vertex 4 becomes 3.

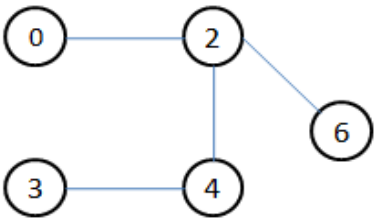

Figure 2.9: Prim's Graph (e)

We repeat the above steps until mstSet includes all vertices of given graph. Finally, we get the following graph.

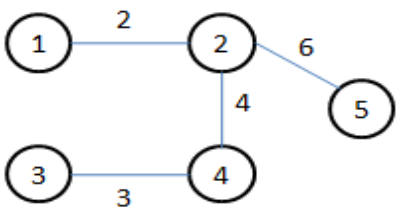

Figure 2.10: Prim's Graph (f)

2.2.3 Complexity of Prim's Algorithm

The running time depends on how we implement the queue data structure.
Binary heap: $O(E \lg V)$
Fibonacci heap: $O(V \lg V + E)$

2.3 Kruskal's Algorithm

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest.It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step.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. If the graph is not connected, then it finds a minimum spanning forest .

2.3.1 Algorithm

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

Algorithm 3 Kruskal's Algorithm

```
1: T ← ∅
2: for each v ∈ V do
3:    MakeSet(v)
4: end for
5: sort E by increasing edge weight w
6: for each (u, v) ∈ E (in sorted order) do
7:    if FindSet(u) ≠ FindSet(v) then
8:       T ← T ∪ [u, v]
9:       Union(FindSet(u), FindSet(v))
10:   end if
11: end for
```

2.3.2 Simulation

Let us understand the algorithm with below input graph. The graph contains 5 vertices and 6 edges. So, the minimum spanning tree formed will be having (5 – 1) = 4 edges
Now picking all edges one by one from sorted list of edges 1. Pick edge 1-2: No cycle is

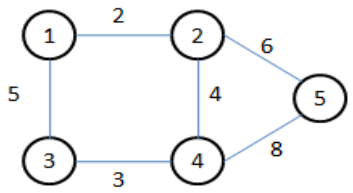

Figure 2.11: Kruskal's Graph (a)

formed, include it.

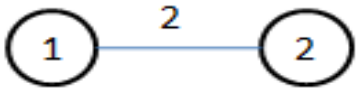

Figure 2.12: Kruskal's Graph (b)

2. Pick edge 3-4: No cycle is formed, include it.

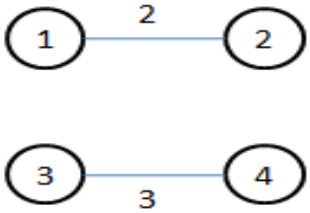

Figure 2.13: Kruskal's Graph (c)

3. Pick edge 2-4: No cycle is formed, include it.

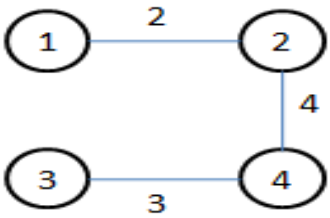

Figure 2.14: Kruskal's Graph (d)

4. Pick edge 1-3: Since including this edge results in cycle, discard it.
5. Pick edge 2-5: No cycle is formed, include it.

6. Pick edge 1-3: Since including this edge results in cycle, discard it. So, the algorithm stops here.

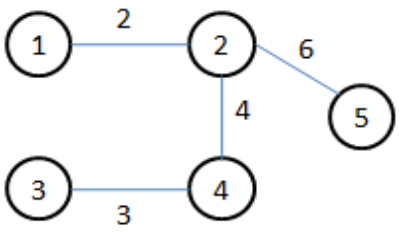

Figure 2.15: Kruskal's Graph (e)

2.3.3 Complexity of Kruskal's Algorithm

Sorting of edges takes $O(E \log E)$ time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost $O(\log V)$ time. So overall complexity is $O(E \log E + E \log V)$ time. The value of E can be atmost $O(V^2)$, so $O(\log V)$ are $O(\log E)$ same. Therefore, overall time complexity is $O(E \log E)$ or $O(E \log V)$

2.4 Randomized Algorithm

A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Karger's algorithm is a randomized algorithm to compute a minimum cut of a connected graph. It was invented by David Karger and first published in 1993.

2.4.1 Algorithm

The idea of the algorithm is based on the concept of contraction of an edge (u,v) in an undirected graph G = (V ,E). The contraction of an edge merges the nodes ***u*** and ***v*** into one, reducing the total number of nodes of the graph by one. All other edges connecting either ***u*** or ***v*** are "reattached" to the merged node, effectively producing a multigraph. Karger's basic algorithm iteratively contracts randomly chosen edges until only two nodes remain; those nodes represent a cut in the original graph. By iterating this basic algorithm a sufficient number of times, a minimum cut can be found with high probability.

Algorithm 4 Karger's Algorithm

1: Initialize contracted graph CG as copy of original graph
2: **while** there are more than 2 vertices **do**
3: Pick a random edge (u, v) in the contracted graph
4: Merge (or contract) u and v into a single vertex (update the contracted graph
5: Remove self-loops
6: **end while**
7: **return** cut represented by two vertices

2.4.2 Simulation

Let us understand Karger's algorithm through the simulation given. Given an undirected and unweighted graph, find the smallest cut. The smallest cut has 2 edges Let the first randomly

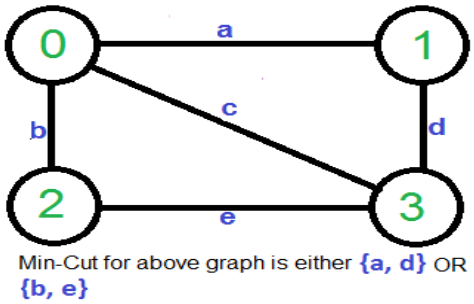

Figure 2.16: Karger's Graph (a)

picked vertex be 'a' which connects vertices 0 and 1. We remove this edge and contract the graph (combine vertices 0 and 1). We get the 2.17 graph. Let the next randomly picked

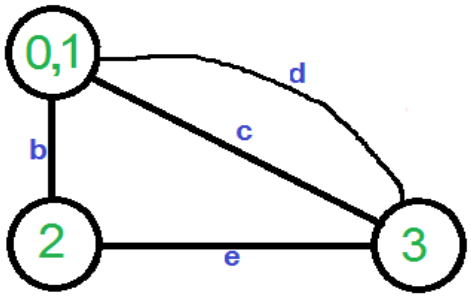

Figure 2.17: Karger's Graph (b)

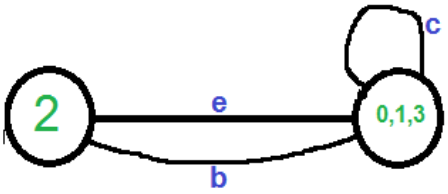

Figure 2.18: Karger's Graph (c)

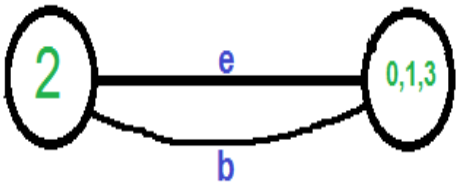

Figure 2.19: Karger's Graph (d)

edge be 'd'. We remove this edge and combine vertices (0,1) and 3. We need to remove self-loops in the graph. So we remove edge 'c'. Now graph has two vertices, so we stop. The number of edges in the resultant graph is the cut produced by Karger's algorithm.

2.4.3 Complexity of Karger's Algorithm

Randomized algorithm Karger's use Max-Flow based s-t cut algorithm to find minimum cut. Consider every pair of vertices as source 's' and sink 't', and call minimum s-t cut algorithm to find the s-t cut. Return minimum of all s-t cuts. Best possible time complexity of this algorithm is $O(V^5)$ for a graph. There are total possible V^2 pairs and s-t cut algorithm for one pair takes O(V*E) time and E = $O(V^2)$].

CHAPTER 3
PROPOSED MINIMUM SPANNING TREE ALGORITHM

3.1 Basic Idea

The node weight of each node is initialized to infinity. All the edge weights of the graph are inserted into the edge list. An edge is selected from the edge weight and its both end nodes are determined. If the node weights of the end nodes are greater than the edge weight between them then the node weights are updated with the edge weight. After traversing the entire edge list we see the node weights of each node is updated with the distance to the nearest node from that particular node. Traversing each node we select the nearest vertex from it using its node weight. We obtained subsets of the graph where each node is connected to its nearest vertex i.e connected to other node with the minimum edge weight. The edge list is now updated with the edge weights which are not selected in this process of making subsets. Now Kruskal's algorithm is applied using only the edge weights in the updated edge list. At the end each subset will be connected to only one subset with only one edge which will be existing in the edge list. This is done so as to avoid formation of cycle. The final graph after the connection of subsets gives the minimum spanning tree.

3.2 Algorithm

Proposed algorithm starts with an empty spanning tree. The idea is to maintain two sets of nodes. The first set contains the nodes already included in the MST, the other set contains the nodes not yet included in MST. This algorithm also works disjoint graph.

Algorithm 5 Proposed Algorithm

$M \leftarrow N[G]$
for each $a \in M$ **do**
  $W[a] \leftarrow \infty$
  Choose any edge randomly W[E]
  if Wn[a($1st\ \ node\ \ of\ \ that\ \ edge$)] $> We[E]$) **then**
    Wn[a($1st\ \ node\ \ of\ \ that\ \ edge$)] $\leftarrow We[E]$
  end if
  if Wn[a($2nd\ \ node\ \ of\ \ that\ \ edge$)] $> We[E]$) **then**
    Wn[a($2nd\ \ node\ \ of\ \ that\ \ edge$)] $\leftarrow We[E]$
  end if
end for
Makes subset using nearest vertex -¿ L
sort unused edges m by increasing edge weight w
for each $(u, v) \in m$ (in sorted order) **do**
  if $FindSet(u) \neq FindSet(v)$ **then**
    $L \leftarrow L \cup [u, v]$
    Union(FindSet(u), FindSet(v))
  end if
end for

3.3 Complexity of the Algorithm

In this algorithm, making subset and union take $O(V)$ runtime. Finding set takes $O(m)$ time. As instead of running on E, proposed algorithm runs in m unused edges. So, sorting edges takes $O(m \log m)$. The overall time complexity of this algorithm becomes $O(V)$ + $O(E)$ + $O(m \log m)$ = $O(E + m \log m)$.

3.4 Simulation

Let us consider the following graph. We will apply our proposed algorithm on this graph to find the MST.

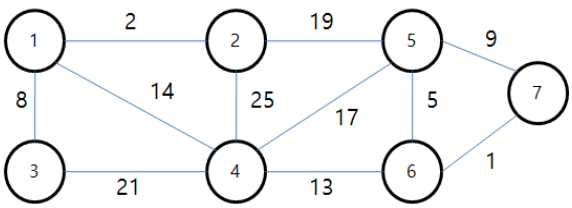

Figure 3.1: Given Graph

All the weights of the nodes are initialized to infinity.

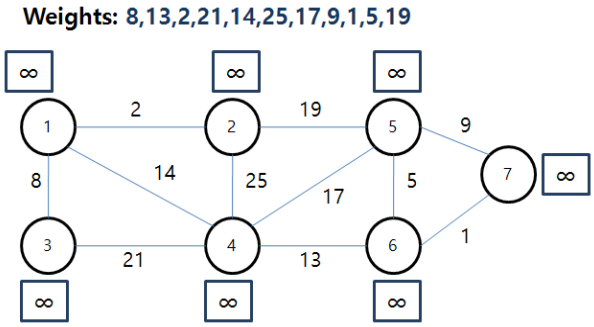

Figure 3.2: Stage 1

One by one the edges are selected from the edge list. Here edge with weight 8 is selected.The weights of node 1 and 3 are updates as they are greater then 8.

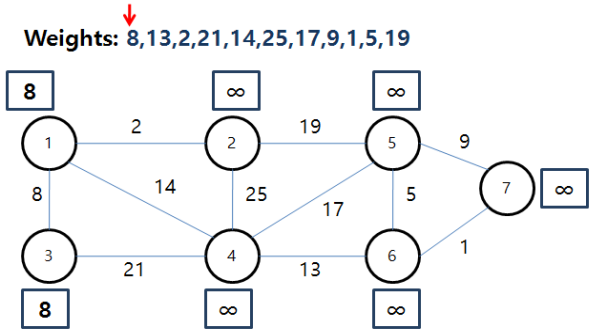

Figure 3.3: Stage 2

Edge with weight 13 is selected. Weights of nodes 4 and 6 are updated to 13.

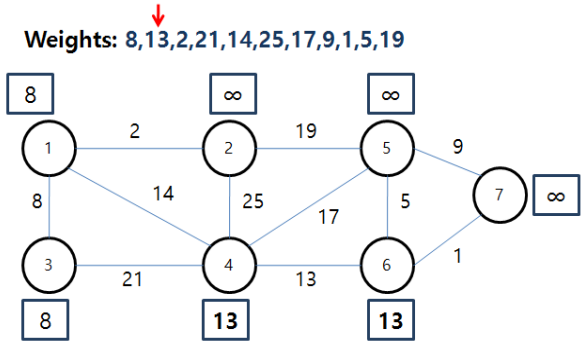

Figure 3.4: Stage 3

Node weights of 1 and 2 are updated to 2 from 8 and infinity respectively.

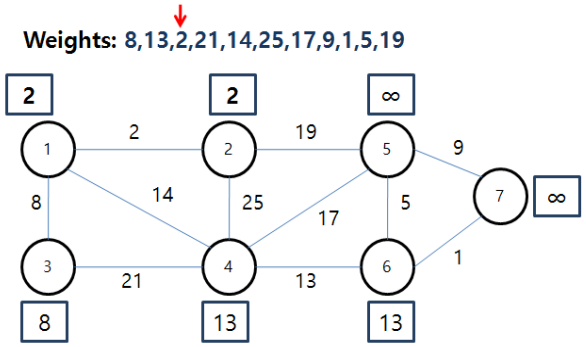

Figure 3.5: Stage 4

In Figure 3.6 it is observed that the node weights of 3 and 4 are not updated as they are not greater then 21.

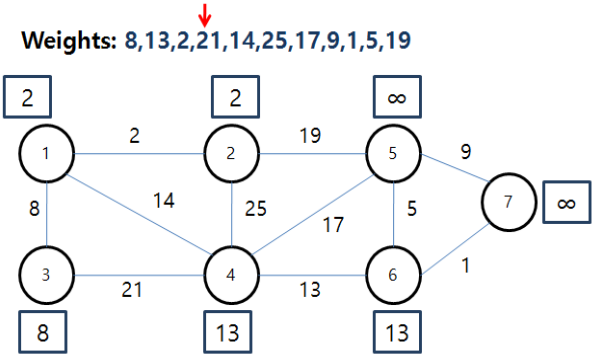

Figure 3.6: Stage 5

Similar to Figure 3.6 the node weights of 1 and 4 do not update due to being less then 14 (Figure 3.7)

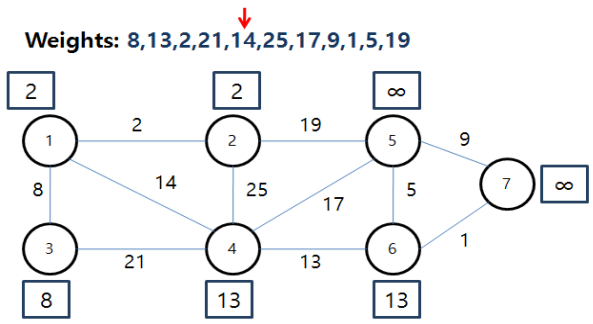

Figure 3.7: Stage 6

The node weights of nodes 2 and 4 also don not get updated as they are smaller then 25.

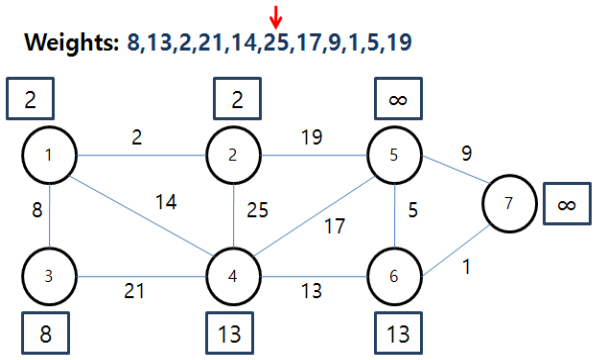

Figure 3.8: Stage 7

When edge with weight 17 is selected the node weight of node 4 does not change as it is less then 17 but the node weight of node 5 is updated from infinity to 17.

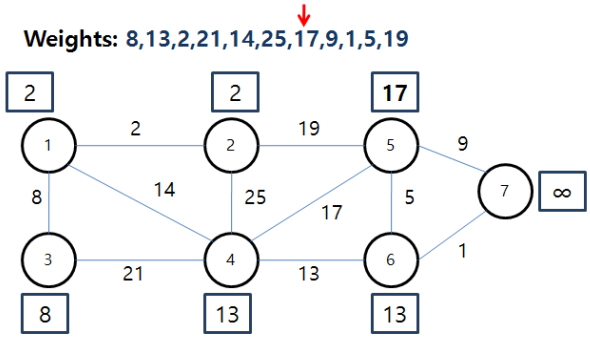

Figure 3.9: Stage 8

The node weights of nodes 5 and 7 are updated to 9 on the selection of edge with weight 9.

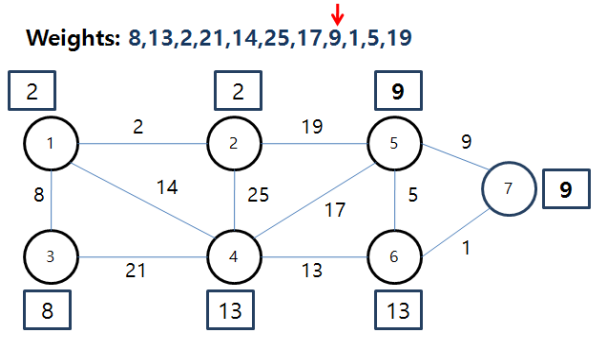

Figure 3.10: Stage 9

Node weights of nodes 6 and 7 are updated to 1 as it is greater than the edge weight 1.

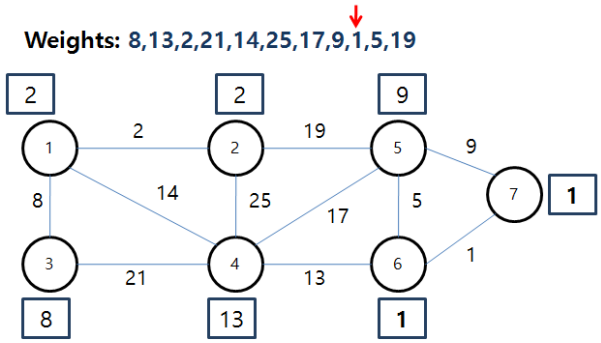

Figure 3.11: Stage 10

Selection of edge weight 9 updates the node weight of node 5 but not that of node 6.

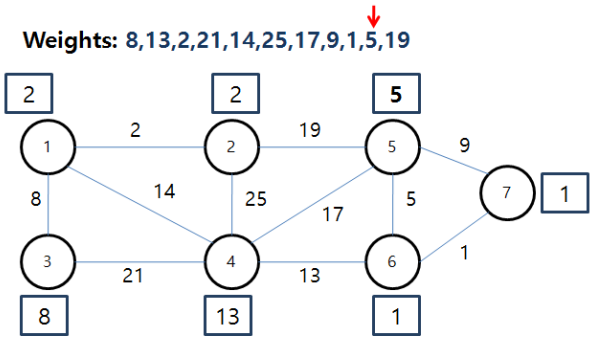

Figure 3.12: Stage 11

When edge weight with weight 19 is selected the node weights of nodes 2 and 5 do not change as they are smaller than 19.

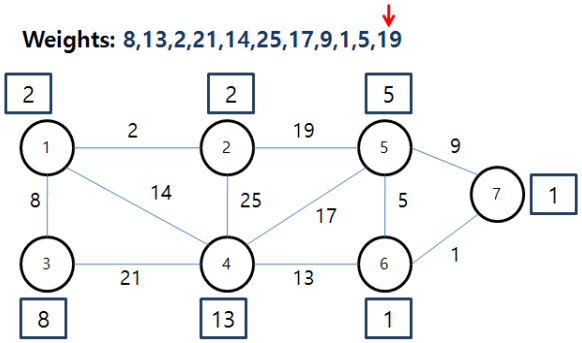

Figure 3.13: Stage 12

Now after all the edges are traversed,the node weights of the nodes contain the cheapest weight to be connected to other node. Using those cheapest weights of each node subsets are created.The edge list then updated with the edges not used in making of subsets.In figure 3.14 it is seen that two subsets are formed.

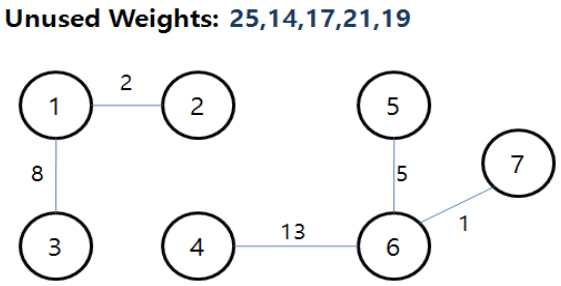

Figure 3.14: Stage 13

Kruskal's algorithm is then applied only on the unused edges.The edges in the edge list is sorted which is the first step of Kruskal's algorithm.

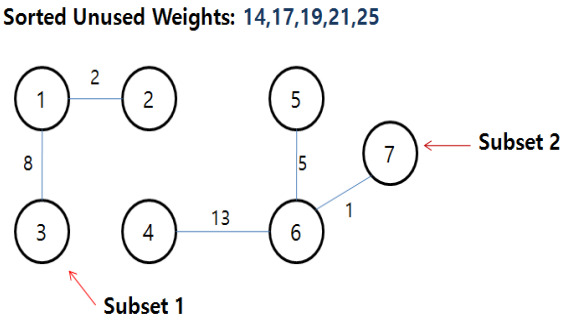

Figure 3.15: Stage 14

Each time an edge is selected from the sorted edge list it is check whether any cycle is formed or not. I cycle is formed then that edge is not considered.Here in Figure 3.16 when edge with weight 14 is selected both the subsets are connected. After traversing the edge list no other edge is found which can be considered due to formation of cycles.

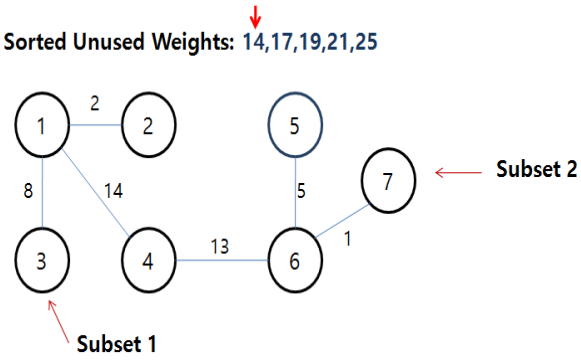

Figure 3.16: Minimum Spanning Tree

CHAPTER 4
EXPERIMENTAL RESULTS

4.1 Design of Experiments

Our proposed algorithm has been implemented in Java and compiled in Netbeans 8.0.2. The test is performed in windows 8. We have used a machine with processor intel core i5, clock speed 2.7 giga hz, ram 8 GB. Testing Graphs are generated randomly and the algorithm is implemented on those graphs in order to calculate the running times.

In our work, we have presented a number of experimental tests which are listed in Table 4.1 and 4.2 .

Table 4.1: Runtime Calculation for Less Dense Graph in Proposed Algorithm

Test Cases	Number of Nodes	Number of Edges	Used Edges	Unused Edges	Runtime
1	500	1000	498	502	1.00
2	1000	2000	998	1002	1.00
3	2000	4000	1997	2003	1.00
4	5000	11500	4997	6503	2.00
5	10000	20503	9997	10506	5.00
6	20000	30000	19998	10002	10.00
7	30000	65000	29997	35003	45.67
8	40000	87000	39997	47003	83.67
9	50000	100000	49998	50002	143.33

A pie chart given below shows the percentage of used and unused edges for a graph having 50000 nodes and 100000 edges after our proposed algorithm is applied on it. In the Figure 4.1, region (1) indicates the unused edges and region (2) indicates the used edges.

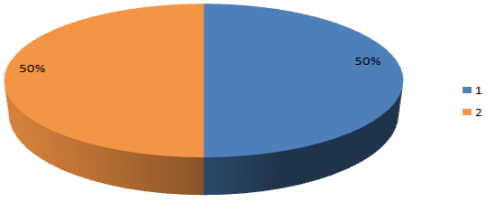

Figure 4.1: Used and Unused Edges for Less Dense Graph

From the above data in Table 4.1, it can be mentioned that if the number of edges in the graph

is almost double of the the number of nodes, then in those cases the number of edges in the edge list becomes approximately half the original edge list. It can be said that almost half of the edges are omitted at the time of generating subsets. The number of used and unused edges are approximately equal. Since sorting is done only on half of the edges it takes less time to give results in some cases. However for large graphs this law is not maintained due to the excess loops used in the code.

Table 4.2: Runtime Calculation for Dense Graph in Proposed Algorithm

Test Cases	Number of Nodes	Number of Edges	Used Edges	Unused Edges	Runtime
1	500	3000	493	2507	1.00
2	1000	4000	996	3004	1.33
3	2000	7000	1996	5004	1.67
4	5000	18990	4996	13994	4.00
5	10000	40000	9996	30004	15.33
6	20000	60000	19997	40003	35.67
7	30000	80000	29997	50003	66.00
8	40000	90000	39997	50517	108.00
9	50000	200000	99995	100005	930.00

A pie chart given below shows the percentage of used and unused edges for a graph having 50000 nodes and 100000 edges after our proposed algorithm is applied on it.vIn the Figure 4.2, region (1) indicates the unused edges and region (2) indicates the used edges.

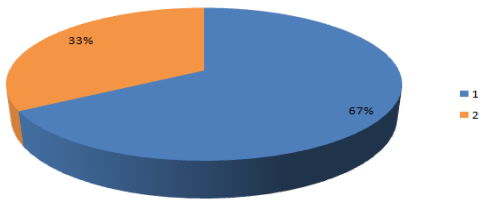

Figure 4.2: Used and Unused Edges for Denser Graph

Unfortunately our algorithm is not so efficient in case of denser graphs where the number of edges exceeds the double of the number of nodes. It could have been efficient give the use of excess loops in the code was omitted. The problem arises because the number of edges on which sorting is to performed does not reduce too much. The number of edges eliminated during making of subsets is not close to half the original number of edges.

4.2 Comparative performance of Kruskal's and Proposed Algorithm

Kruskal's Algorithm sorts the edges in order O(ElogE) and detects cycles for all the edges in O(ElogE) where E is the number of edges in the graph.So the complexity of Kruskal's Algorithm is O(ElogE).

Our proposed algorithm eliminates some edges initially at the time of generating subsets. It is done in the order O(E).The number of edges becomes then m on which Kruskal's Algorithm is applied.The complexity for this becomes O(mlogm). Therefore total complexity is O(E+mlogm).

Table 4.3: Comparison of Runtime for Proposed Algorithm Kruskal Algorithm (Less Dense Graph)

Test Cases	Number of Nodes	Number of Edges	Runtime (Proposed)	Runtime (Kruskal)
1	500	1000	1.00	1.33
2	1000	2000	1.00	1.33
3	2000	4000	1.00	1.33
4	5000	11500	2.00	2.33
5	10000	20503	5.00	4.67
6	20000	30000	10.00	10.33
7	30000	65000	45.67	39.33
8	40000	87000	83.67	74.00
9	50000	100000	143.33	114.67

Table 4.4: Comparison of Runtime for Proposed Algorithm Kruskal Algorithm (Dense Graph)

Test Cases	Number of Nodes	Number of Edges	Runtime (Proposed)	Runtime (Kruskal)
1	500	3000	1.00	1.00
2	1000	4000	1.33	1.67
3	2000	7000	1.67	1.33
4	5000	18990	4.00	4.00
5	10000	40000	15.33	14.00
6	20000	60000	35.67	32.67
7	30000	80000	66.00	61.67
8	40000	90000	108.00	91.00
9	50000	200000	930.00	895.00

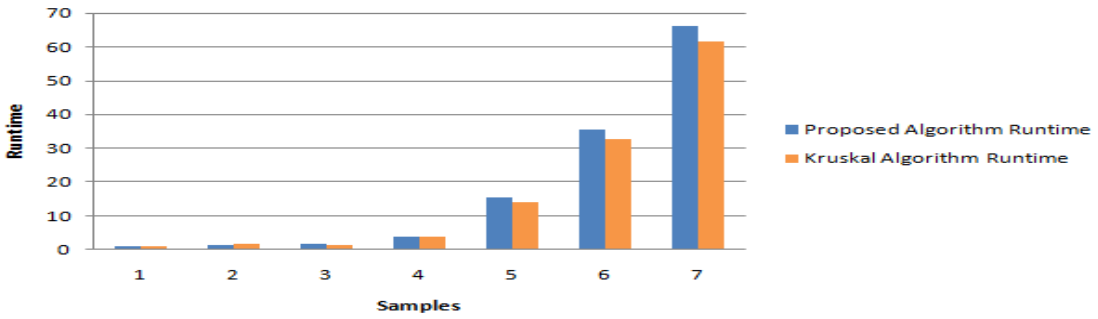

Figure 4.3: Used and Unused Edges for Denser Graph

In Table 4.3 the comparison of run time between Kruskal's and our proposed algorithm is shown through the experimental results. It is seen that in case of less dense graph where the

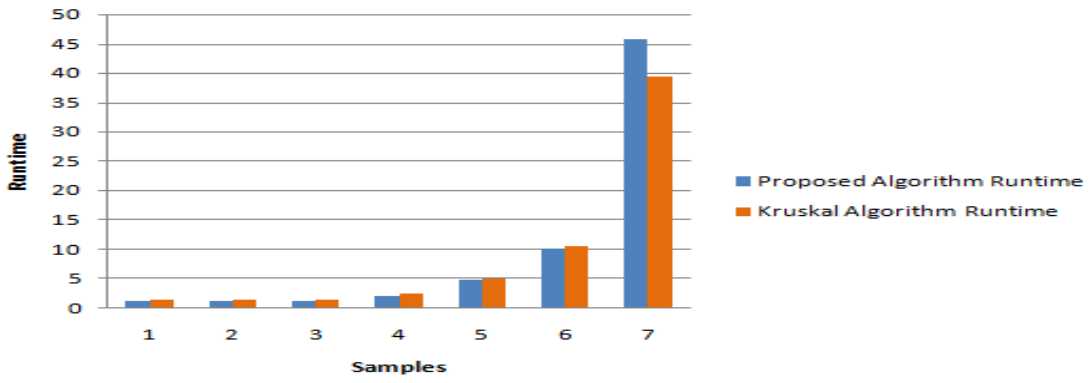

Figure 4.4: Used and Unused Edges for Denser Graph

number of edges in approximately double of the number of nodes our proposed algorithm gives better performance than Kruskal's Algorithm. However when the size of graph keeps on increasing our algorithm becomes inefficient due to the use of excess loops used in the code to find out the used and unused edges.

Table 4.4 shows the experimental results of our proposed algorithm and Kruskal's algorithm on dense graphs where number of edges is greater then twice the number of nodes. In such cases we have come to a conclusion that Kruskal's algorithm performs better than our proposed algorithm.

CHAPTER 5
CONCLUSION

In this paper, we have introduced a new approach of finding MST. Experimental results showed that proposed algorithm works on less dense graph efficiently.
Our proposed MST filters out a large number of edges before sorting edges. For this reason, complexity of sorting edges can be improved. In some test cases, this algorithm works better than Kruskal's algorithm. Proposed algorithm's run time increases for dense graph. But it is not always possible to say that one algorithm is better than another, as relative performance can vary depending on type data.
We hope that further research will improve the complexity of this algorithm for dense graph with better performance.

REFERENCES

[1] Nešetřil, Jaroslav, Eva Milková, and Helena Nešetřilová. "Otakar Borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history." Discrete mathematics 233.1-3 (2001): 3-36.

[2] Prim, Robert Clay. "Shortest connection networks and some generalizations." Bell Labs Technical Journal 36.6 (1957): 1389-1401. APA

[3] Kruskal, Joseph B. "On the shortest spanning subtree of a graph and the traveling salesman problem." Proceedings of the American Mathematical society 7.1 (1956): 48-50.

[4] Graham, Ronald L., and Pavol Hell. "On the history of the minimum spanning tree problem." Annals of the History of Computing 7.1 (1985): 43-57.

[5] Gabow, Harold N., et al. "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs." Combinatorica 6.2 (1986): 109-122.

[6] Fredman, Michael L., and Dan E. Willard. "Trans-dichotomous algorithms for minimum spanning trees and shortest paths." Journal of Computer and System Sciences 48.3 (1994): 533-551.

[7] Karger, David R., Philip N. Klein, and Robert E. Tarjan. "A randomized linear-time algorithm to find minimum spanning trees." Journal of the ACM (JACM) 42.2 (1995): 321-328.

[8] "Verification and sensitivity analysis of minimum spanning trees in linear time", "http://www.nzdl.org/gsdlmod?e=d-00000-00—off-0cstr–00-0—-0-10-0—0—0direct-10—4———-0-0l–11-en-50—20-about—00-0-1-00-0-0-11-1-0utfZz-8-10-0-0-11-10-0utfZz-8-00a=dcl=CL1.95d=HASH01ce2b9e98dde175120f238d.1", The New Zealand Digital Library, The University of Walkato.

[9] King, Valerie. "A simpler minimum spanning tree verification algorithm." Algorithmica 18.2 (1997): 263-270.

[10] Raidl, Günther R. "An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem." Evolutionary Computation, 2000. Proceedings of the 2000 Congress on. Vol. 1. IEEE, 2000.

[11] Buchsbaum, Adam L., et al. "Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators." Proceedings of the thirtieth annual ACM symposium on Theory of computing. ACM, 1998.

[12] Pettie, Seth, and Vijaya Ramachandran. ”An optimal minimum spanning tree algorithm.” Journal of the ACM (JACM) 49.1 (2002): 16-34.

[13] Mamun, Abdullah-Al, and Sanguthevar Rajasekaran. ”An efficient Minimum Spanning Tree algorithm.” Computers and Communication (ISCC), 2016 IEEE Symposium on. IEEE, 2016.

[14] Bazlamaçcı, Cüneyt F., and Khalil S. Hindi. ”Minimum-weight spanning tree algorithms a survey and empirical study.” Computers Operations Research 28.8 (2001): 767-785.

[15] Chazelle, Bernard. ”A faster deterministic algorithm for minimum spanning trees.” Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on. IEEE, 1997.

[16] Chazelle, Bernard. ”The soft heap: an approximate priority queue with optimal error rate.” Journal of the ACM (JACM) 47.6 (2000): 1012-1027.

[17] Chong, Ka Wong, Yijie Han, and Tak Wah Lam. ”Concurrent threads and optimal parallel minimum spanning trees algorithm.” Journal of the ACM (JACM) 48.2 (2001): 297-323.

[18] Bergantiños, Gustavo, and Juan J. Vidal-Puga. ”A fair rule in minimum cost spanning tree problems.” Journal of Economic Theory 137.1 (2007): 326-352.

[19] Haymond, R. E., J. P. Jarvis, and D. R. Shier. ”Computational methods for minimum spanning tree algorithms.” SIAM journal on scientific and statistical computing 5.1 (1984): 157-174.

[20] Johnson, Donald B. ”Priority queues with update and finding minimum spanning trees.” Information Processing Letters 4.3 (1975): 53-57.

[21] Yao, Andrew Chi-Chih. ”An$0(|E| \log\log |V|)$ algorithm for finding minimum spanning trees.” Information Processing Letters 4.1 (1975): 21-23.

[22] Gallager, Robert G., Pierre A. Humblet, and Philip M. Spira. ”A distributed algorithm for minimum-weight spanning trees.” ACM Transactions on Programming Languages and systems (TOPLAS) 5.1 (1983): 66-77.

[23] Fredman, Michael L., and Robert Endre Tarjan. ”Fibonacci heaps and their uses in improved network optimization algorithms.” Journal of the ACM (JACM) 34.3 (1987): 596-615.