Project Report

on

**Minimum Spanning Trees**



**Fahad Nazmi (2K20/IT/48)**

**CONTENTS**

[**Basic Terminology** 3](#_Toc87878472)

[**Graphs** 3](#_Toc87878473)

[**Spanning Trees** 3](#_Toc87878474)

[**Minimum Spanning Trees** 4](#_Toc87878475)

[**Applications of Minimum Spanning Trees** 4](#_Toc87878476)

[**Network Design** 4](#_Toc87878477)

[**Image Segmentation** 5](#_Toc87878478)

[**Handwriting Recognition** 6](#_Toc87878479)

[**Circuit Design** 7](#_Toc87878480)

[**Jigsaw Puzzle Solver** 7](#_Toc87878481)

[**Other Practical Applications** 8](#_Toc87878482)

[**Algorithms** 9](#_Toc87878483)

[**Kruskal’s Algorithm** 9](#_Toc87878484)

[**Algorithm** 9](#_Toc87878485)

[**Implementation** 12](#_Toc87878486)

[**Prim’s Algorithm** 14](#_Toc87878487)

[**Algorithm** 14](#_Toc87878488)

[**Implementation (Brute Force)** 16](#_Toc87878489)

[**Implementation (Priority Queue)** 18](#_Toc87878490)

[**MATLAB Visualization** 19](#_Toc87878491)

[**Output** 20](#_Toc87878492)

[**MST of a Grid Network (Visualization)** 20](#_Toc87878493)

[**Implementation (Python)** 21](#_Toc87878494)

[**Output** 21](#_Toc87878495)

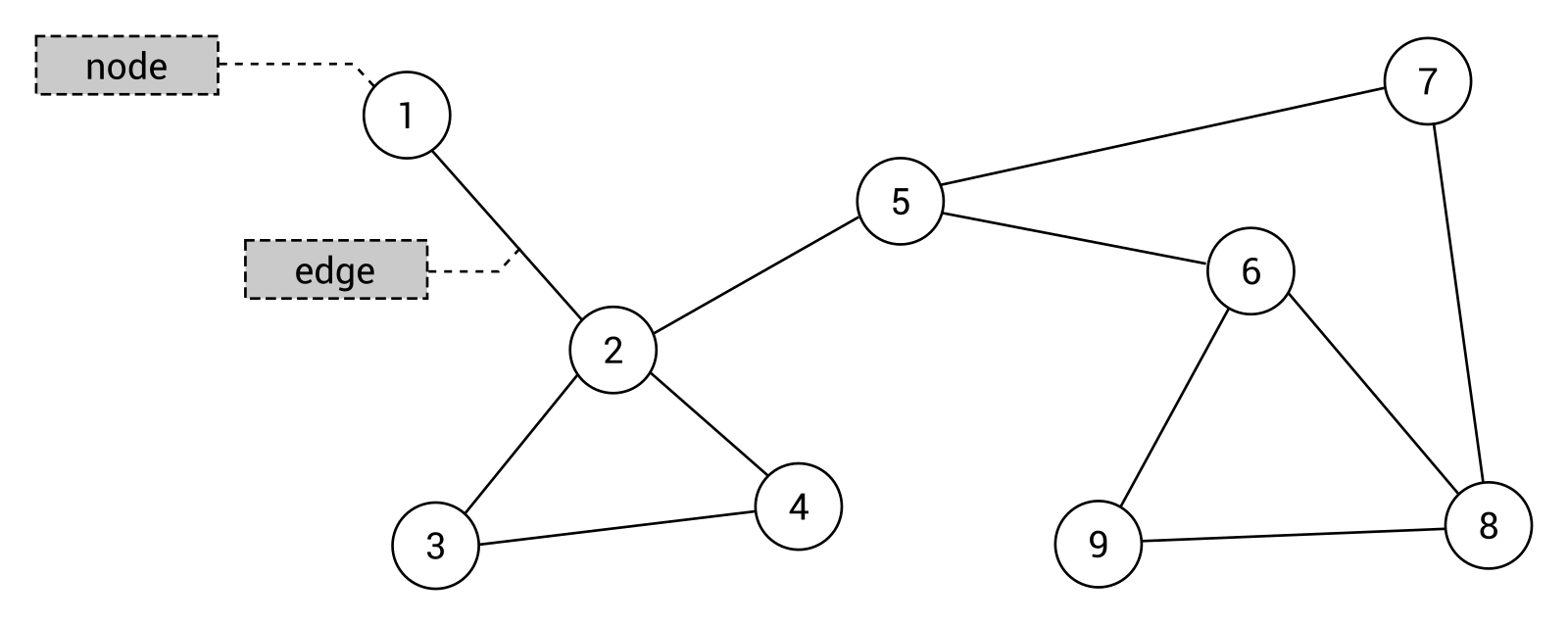
[**Results** 22](#_Toc87878496)

[**References** 24](#_Toc87878497)

# **Basic Terminology**

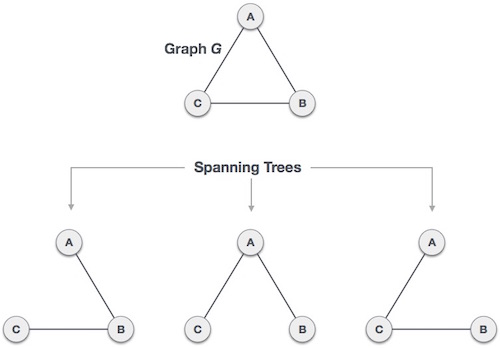
## **Graphs**

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

****

A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

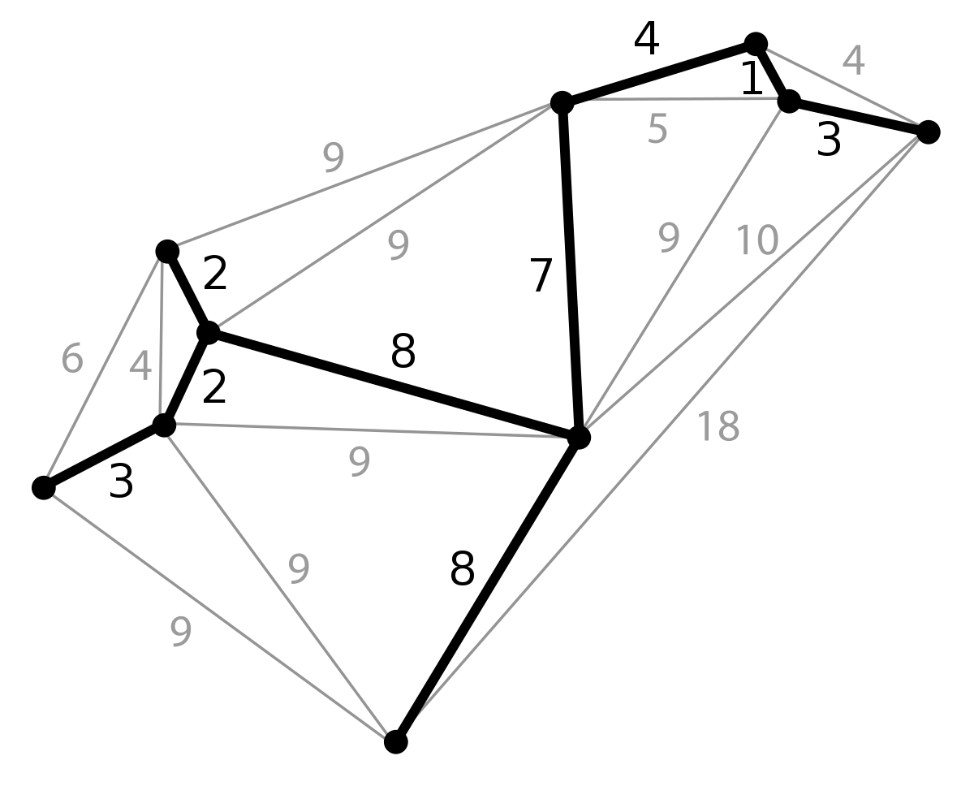
## **Spanning Trees**

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

## **Minimum Spanning Trees**

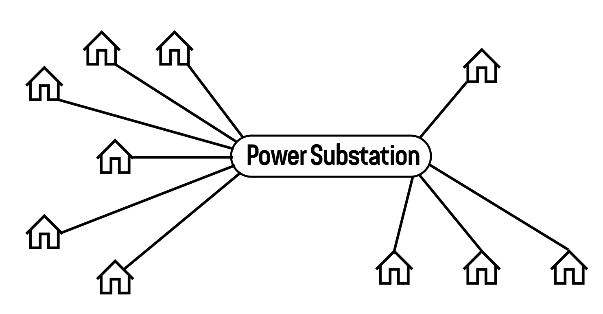
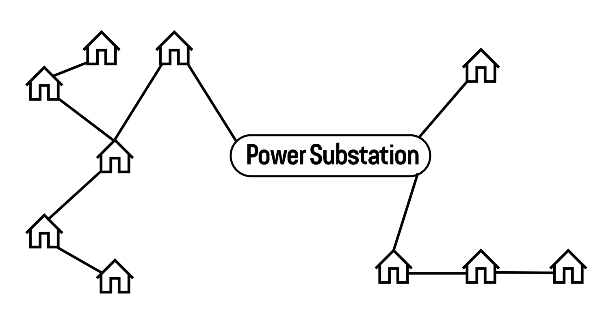
In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges

.

# **Applications of Minimum Spanning Trees**

## **Network Design**

Minimum spanning trees have direct applications in the design of networks, including computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids.

One example would be a telecommunications company laying cable to a new neighbourhood. If it is constrained to bury the cable only along certain paths (e.g., along roads), then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, thus would represent the least expensive path for laying the cable.

1. (b)

In Figure (a), although all the houses are connected directly to Power Substation, such kind of a network is usually more expensive to set up. In Figure (b), we use a minimum spanning tree so that all houses are directly or indirectly connected to the Power Substation, thereby minimising the total setup cost.

## **Image Segmentation**

Image segmentation strives to partition a digital image into regions of pixels with similar properties, e.g., homogeneity. The higher-level region representation simplifies image analysis tasks such as counting objects or detecting changes, because region attributes (e.g. average intensity or shape[[2]](https://en.wikipedia.org/wiki/Minimum_spanning_tree-based_segmentation#cite_note-ipsv97-2)) can be compared more readily than raw pixels.



Fig.: Image Segmentation

The possibility of [stitching](https://en.wikipedia.org/wiki/Image_stitching) together independent sub-images motivates adding connectivity information to the pixels. This can be viewed as a graph, the nodes of which are pixels, and edges represent connections between pixels.

Edges are considered in increasing order of weight; their endpoint pixels are merged into a region if this doesn't cause a cycle in the graph, and if the pixels are 'similar' to the existing regions' pixels. Detecting cycles is possible in near-constant time with the aid of a [disjoint-set data structure](https://en.wikipedia.org/wiki/Disjoint-set_data_structure).[[5]](https://en.wikipedia.org/wiki/Minimum_spanning_tree-based_segmentation#cite_note-hr00-5) Pixel similarity is judged by a heuristic, which compares the weight to a per-segment threshold. The algorithm outputs multiple disjunct MSTs, i.e., a forest; each tree corresponds to a segment.



Fig.: MST Segmentation by varying sensitivity parameter

## **Handwriting Recognition**

Document image layout analysis and text line

extraction are important tasks in optical character

recognition (OCR). Unlike that printed documents consist

mostly of rectangular blocks and straight lines, the text

lines in unconstrained handwritten documents are often

multi-skewed and curved, and the space between lines is

not obvious. Moreover, some text lines in handwritten

documents touch with each other. So, the techniques used

in printed document analysis, such as projection analysis,

Hough transform and recursive X-Y cut, do not perform

well on handwritten documents.

Document image layout analysis and text line

extraction are important tasks in optical character

recognition (OCR). Unlike that printed documents consist

mostly of rectangular blocks and straight lines, the text

lines in unconstrained handwritten documents are often

multi-skewed and curved, and the space between lines is

not obvious. Moreover, some text lines in handwritten

documents touch with each other. So, the techniques used

in printed document analysis, such as projection analysis,

Hough transform and recursive X-Y cut, do not perform

well on handwritten documents

Document image layout analysis and text line

extraction are important tasks in optical character

recognition (OCR). Unlike that printed documents consist

mostly of rectangular blocks and straight lines, the text

lines in unconstrained handwritten documents are often

multi-skewed and curved, and the space between lines is

not obvious. Moreover, some text lines in handwritten

documents touch with each other. So, the techniques used

in printed document analysis, such as projection analysis,

Hough transform and recursive X-Y cut, do not perform

well on handwritten documents

Document image layout analysis and text line

extraction are important tasks in optical character

recognition (OCR). Unlike that printed documents consist

mostly of rectangular blocks and straight lines, the text

lines in unconstrained handwritten documents are often

multi-skewed and curved, and the space between lines is

not obvious. Moreover, some text lines in handwritten

documents touch with each other. So, the techniques used

in printed document analysis, such as projection analysis,

Hough transform and recursive X-Y cut, do not perform

well on handwritten documents

Document image layout analysis and text line extraction are important tasks in optical character recognition (OCR). Unlike printed documents which consist mostly of rectangular blocks and straight lines, the text lines in unconstrained handwritten documents are often multi-skewed and curved, and the space between lines is not obvious. Moreover, some text lines in handwritten documents touch with each other. So, the techniques used in printed document analysis, such as projection analysis, Hough transform and recursive X-Y cut, do not perform well on handwritten documents.

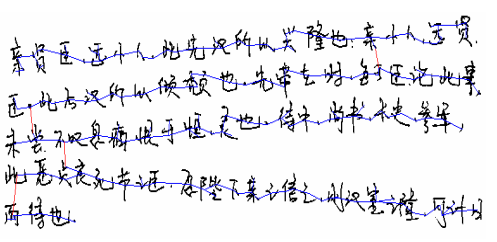
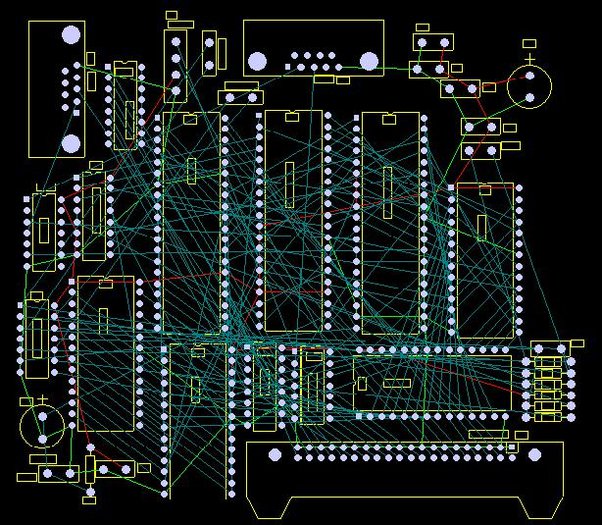


Fig.: MST of a document image

Each component is then viewed as a node in a graph (document graph). Each pair of nodes is linked by an edge with the distance between them as the weight. The distance measure is designed to encourage within-line links and discourage between-line links. From the weighted document graph, a minimum spanning tree (MST) is built using Kruskal’s algorithm.

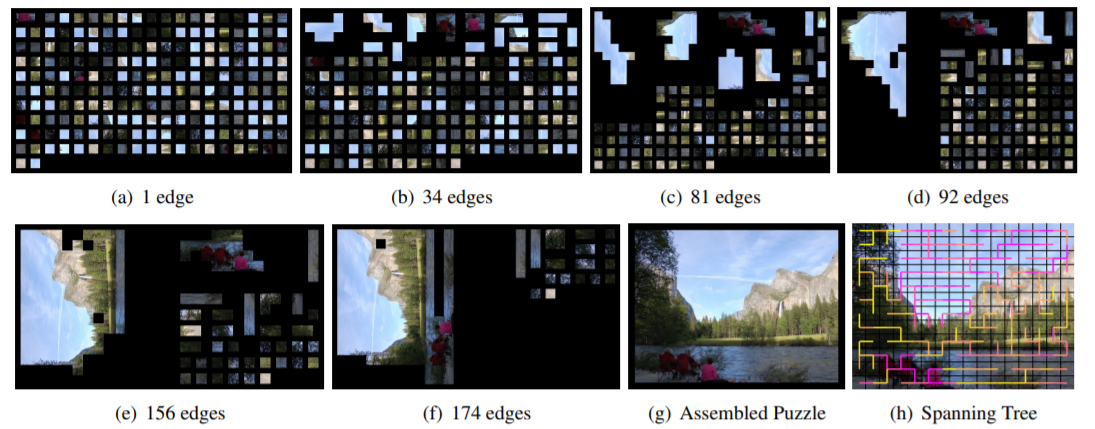
## **Circuit Design**

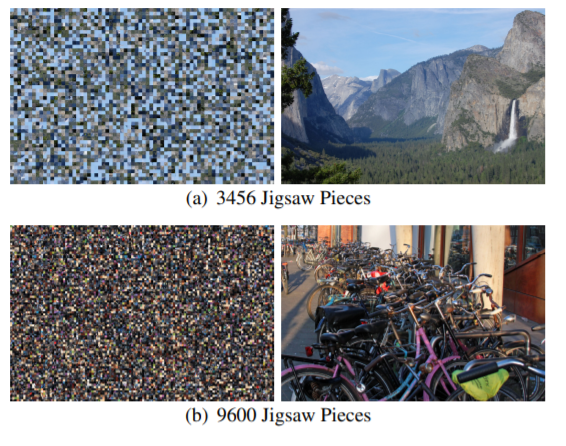
If you are designing a PCB with a CAD system, when placing components on the board it can be helpful for the system to display connections between that component and already placed components. These connections are drawn as straight lines, called a rat’s nest, and need to be updated as a real-time display. Kruskal’s algorithm was used to calculate the rat’s nest as a minimum spanning tree. Not only did this work very well, it also allowed to include routing constraints, needed for ECL, where certain pins are transmitters and others are receivers.



## **Jigsaw Puzzle Solver**

A Minimum Spanning Tree-based reassembly technique that greedily merges components while respecting the geometry of the puzzle has been proposed. With this, puzzles can be assembled whether or not the orientation of the pieces is known.





## **Other Practical Applications**

* Taxonomy
* Cluster Analysis
* Constructing Trees for broadcasting in computer networks
* Curvilinear Feature Extraction in computer vision
* Circuit Design
* Regionalization of socio-geographic areas
* Minimax process control
* Measuring homogeneity of two-dimensional materials
* Comparing ecotoxicology data
* Topological Observability in Power Systems

# **Algorithms**

## **Kruskal’s Algorithm**

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph. Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

This algorithm first appeared in Proceedings of the American Mathematical Society, pp. 48–50 in 1956, and was written by Joseph Kruskal.

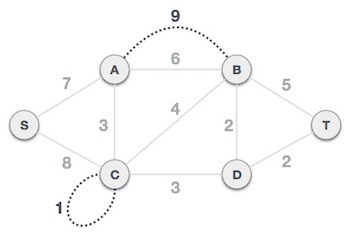
### **Algorithm**

To understand Kruskal's algorithm let us consider the following example −



**Step 1 - Remove all loops and Parallel Edges**

Remove all loops and parallel edges from the given graph.

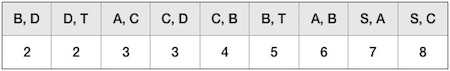


In case of parallel edges, keep the one which has the least cost associated and remove all others.



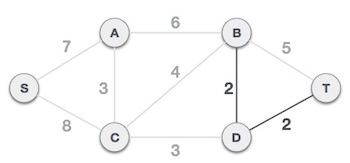
**Step 2 - Arrange all edges in their increasing order of weight**

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).



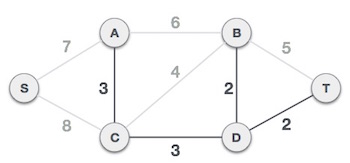
**Step 3 - Add the edge which has the least weightage**

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

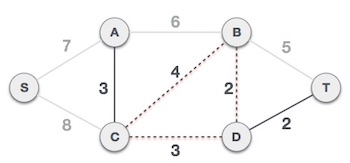


The least cost is 2 and edges involved are B, D and D, T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

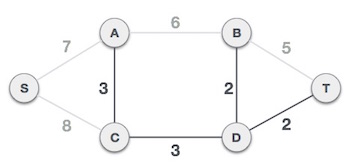
Next cost is 3, and associated edges are A, C and C, D. We add them again −



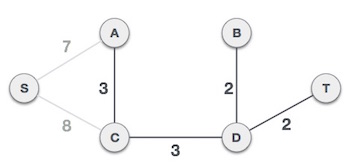
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



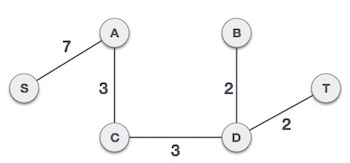
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S, A we have included all the nodes of the graph and we now have minimum cost spanning tree.

### **Implementation**

#include<bits/stdc++.h>

using namespace std;

const int MAX = 1e4;

int parent[MAX], nodes, edges;

pair <long long, pair<int, int> > p[MAX];

void initialize()

{

    for(int i = 0;i < MAX;++i)

        parent[i] = i;

}

int Find(int *k*)

    {

*// if `k` is root*

        if (parent[*k*] == *k*) {

            return *k*;

        }

*// recur for the parent until we find the root*

        return Find(parent[*k*]);

    }

void Union(int *x*, int *y*)

{

    int p = Find(*x*);

    int q = Find(*y*);

    parent[p] = parent[q];

}

long long kruskal(pair<long long, pair<int, int> > *p*[])

{

    int x, y;

    long long cost, minimumCost = 0;

    for(int i = 0;i < edges;++i)

    {

*// Selecting edges one by one in increasing order from the beginning*

        x = *p*[i].second.first;

        y = *p*[i].second.second;

        cost = *p*[i].first;

*// Check if the selected edge is creating a cycle or not*

        if(Find(x) != Find(y))

        {

            minimumCost += cost;

            Union(x, y);

        }

    }

    return minimumCost;

}

int main()

{

    int x, y;

    long long weight, cost, minimumCost;

    initialize();

    cout<<"\nEnter Number of Nodes : ";

    cin >> nodes;

    cout<<"\nEnter number of Edges : ";

    cin >> edges;

    for(int i = 0;i < edges;++i)

    {

        cout<<"\nEnter Source Vertex for Edge "<< i+1 <<" : ";

        cin >> x;

        cout<<"Enter Destination Vertex for Edge "<< i+1 <<" : ";

        cin >> y;

        cout<<"Enter Weight of Edge "<< i+1 <<" : ";

        cin>> weight;

        p[i] = make\_pair(weight, make\_pair(x, y));

    }

*// Sort the edges in the ascending order of weights*

    sort(p, p + edges);

    minimumCost = kruskal(p);

    cout << "\n\nCost of MST of graph : "<<minimumCost << endl;

    return 0;

}

## **Prim’s Algorithm**

Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959. Therefore, it is also sometimes called the Jarník's algorithm, Prim–Jarník algorithm, Prim–Dijkstra algorithm or the DJP algorithm.

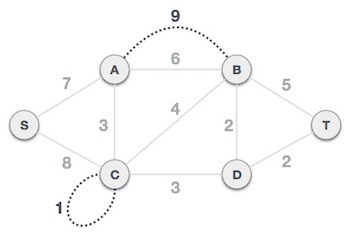
### **Algorithm**

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



**Step 1 - Remove all loops and parallel edges**



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

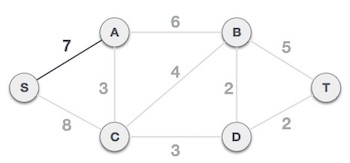


**Step 2 - Choose any arbitrary node as root node**

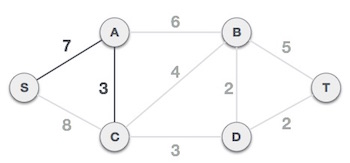
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So, the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

**Step 3 - Check outgoing edges and select the one with less cost**

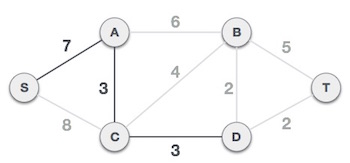
After choosing the root node **S**, we see that S, A and S, C are two edges with weight 7 and 8, respectively. We choose the edge S, A as it is lesser than the other.



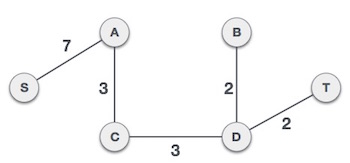
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e., D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

### **Implementation (Brute Force)**

#include <bits/stdc++.h>

using namespace std;

#define V 5

bool createsMST(int *u*, int *v*, vector<bool> *inMST*){

   if (*u* == *v*)

      return false;

   if (*inMST*[*u*] == false && *inMST*[*v*] == false)

      return false;

   else if (*inMST*[*u*] == true && *inMST*[*v*] == true)

      return false;

   return true;

}

void printMinSpanningTree(int *cost*[][V]){

   vector<bool> inMST(V, false);

   inMST[0] = true;

   int edgeNo = 0, MSTcost = 0;

   while (edgeNo < V - 1) {

      int min = INT\_MAX, a = -1, b = -1;

      for (int i = 0; i < V; i++) {

         for (int j = 0; j < V; j++) {

            if (*cost*[i][j] < min) {

               if (createsMST(i, j, inMST)) {

                  min = *cost*[i][j];

                  a = i;

                  b = j;

               }

            }

         }

      }

      if (a != -1 && b != -1) {

         cout<<"Edge "<<edgeNo++<<" : ("<<a<<" , "<<b<<" ) : cost = "<<min<<endl;

         MSTcost += min;

         inMST[b] = inMST[a] = true;

      }

   }

   cout<<"Cost of Minimum spanning tree ="<<MSTcost;

}

int main()

{

   int cost[][V] =

   {

      { INT\_MAX, 12, INT\_MAX, 25, INT\_MAX },

      { 12, INT\_MAX, 11, 8, 12 },

      { INT\_MAX, 11, INT\_MAX, INT\_MAX, 17 },

      { 25, 8, INT\_MAX, INT\_MAX, 15 },

      { INT\_MAX, 12, 17, 15, INT\_MAX },

   };

   cout<<"The Minimum spanning tree for the given tree is :\n";

   printMinSpanningTree(cost);

   return 0;

}

### **Implementation (Priority Queue)**

#include<bits/stdc++.h>

using namespace std;

int main(){

    int cost = 0;

    int N,M;

    cout<<"Enter Number of Vertices : ";

    cin>>N;

    cout<<"\nEnter Number of Edges : ";

    cin>>M;

    vector<pair<int,int> > adj[N];

    int a,b,wt;

    for(int i = 0; i<M ; i++)

    {

        cout<<"\nEnter source vertex for edge "<<i+1<<" : ";

        cin >> a;

        cout<<"\nEnter destination vertex for edge "<<i+1<<" : ";

        cin >> b;

        cout<<"\nEnter weight for edge "<<i+1<<" : ";

        cin >> wt;

        adj[a].push\_back(make\_pair(b,wt));

        adj[b].push\_back(make\_pair(a,wt));

    }

    int parent[N];

    int key[N];

    bool mstSet[N];

    for (int i = 0; i < N; i++)

        key[i] = INT\_MAX, mstSet[i] = false;

    priority\_queue< pair<int,int>, vector <pair<int,int>> , greater<pair<int,int>> > pq;

    key[0] = 0;

    parent[0] = -1;

    pq.push({0, 0});

    while(!pq.empty())

    {

        int u = pq.top().second;

        pq.pop();

        mstSet[u] = true;

        for (auto it : adj[u]) {

            int v = it.first;

            int weight = it.second;

            if (mstSet[v] == false && weight < key[v]) {

                parent[v] = u;

                key[v] = weight;

                cost += weight;

                pq.push({key[v], v});

            }

        }

    }

    for (int i = 1; i < N; i++)

        cout << parent[i] << " - " << i <<" \n";

    cout<<"Cost of Minimum Spanning Tree = "<<cost;

    return 0;

}

# **MATLAB Visualization**

MATLAB has a built-in function that can calculate the minimum spanning tree of a weighted graph.

Using the given lines of MATLAB commands, we can take any graph as input, plot it, calculate its minimum spanning tree, and then highlight the minimum spanning tree in the graph plot.

s = [1 1 1 2 5 3 6 4 7 8 8 8];

t = [2 3 4 5 3 6 4 7 2 6 7 5];

weights = [100 10 10 10 10 20 10 30 50 10 70 10];

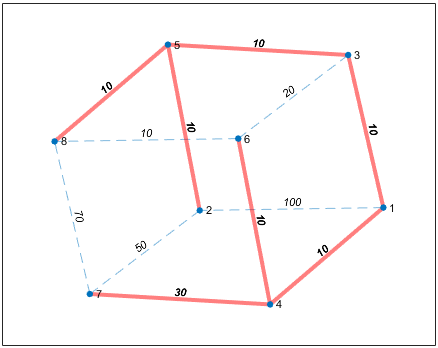
G = graph(s,t,weights);

p = plot(G,'EdgeLabel',G.Edges.Weight,"LineStyle","--");

[T,pred] = minspantree(G);

highlight(p,T,"EdgeColor","red","EdgeFontWeight","bold","LineWidth",3, "LineStyle","-")

## **Output**



# **MST of a Grid Network (Visualization)**

As discussed earlier, some applications of minimum spanning trees are

* Landing cables
* TV Network
* A network of pipes for drinking water or natural gas
* An electric grid
* Network for road/railways for connecting cities
* Irrigation Channels
* Placing microwave towers

In most of these applications, we can consider the system to be a grid network.

In a grid network of NxN nodes, each node is connected directly to its adjacent nodes.

We can assume the weight of each edge to be equal, hence, many minimum spanning trees can be formed.

## **Implementation (Python)**

*import* numpy

*from* numpy *import* \*

*import* networkx *as* nx

*from* networkx *import* \*

*import* matplotlib.pyplot *as* plt

N=30

G=nx.grid\_2d\_graph(N,N)

pos = dict( (n, n) *for* n *in* G.nodes() )

labels = dict( ((i, j), i + (N-1-j) \* N ) *for* i, j *in* G.nodes() )

nx.relabel\_nodes(G,labels,False)

inds=sorted(labels.keys())

vals=sorted(labels.values())

pos2=dict(zip(vals,inds))

nx.draw\_networkx(G, *pos*=pos2, *with\_labels*=False, *node\_size* = 15)

T=nx.minimum\_spanning\_tree(G)

plt.figure()

nx.draw\_networkx(T, *pos*=pos2, *with\_labels*=False, *node\_size* = 15)

plt.show()

## **Output**

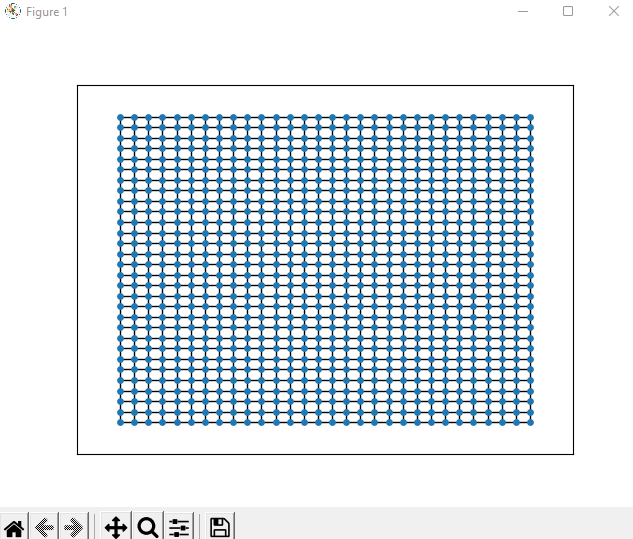
­­

Fig.1: A 30x30 Grid Network

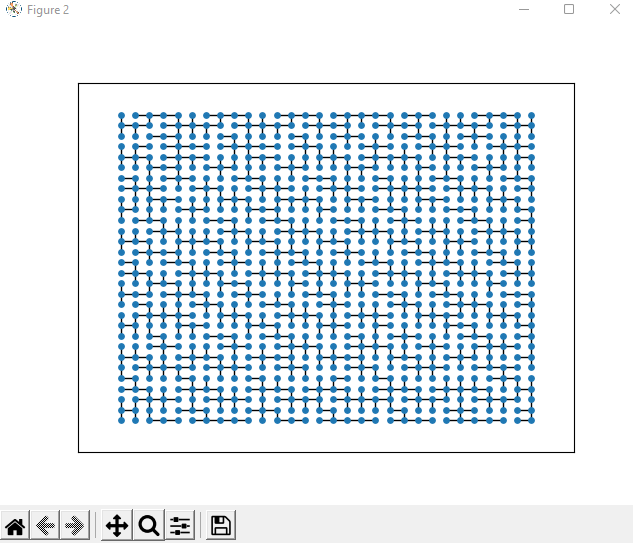
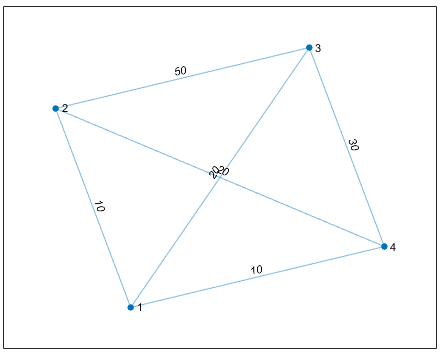


Fig.2: MST of 30x30 Grid Network

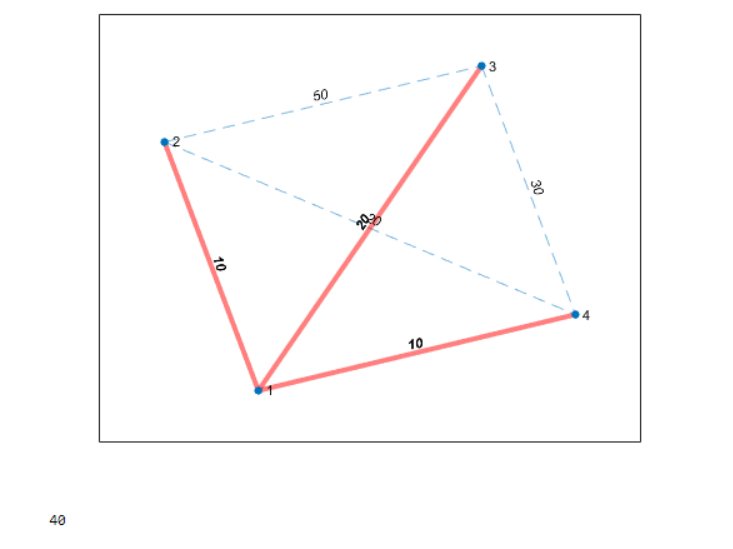
# **Results**

Input Graph

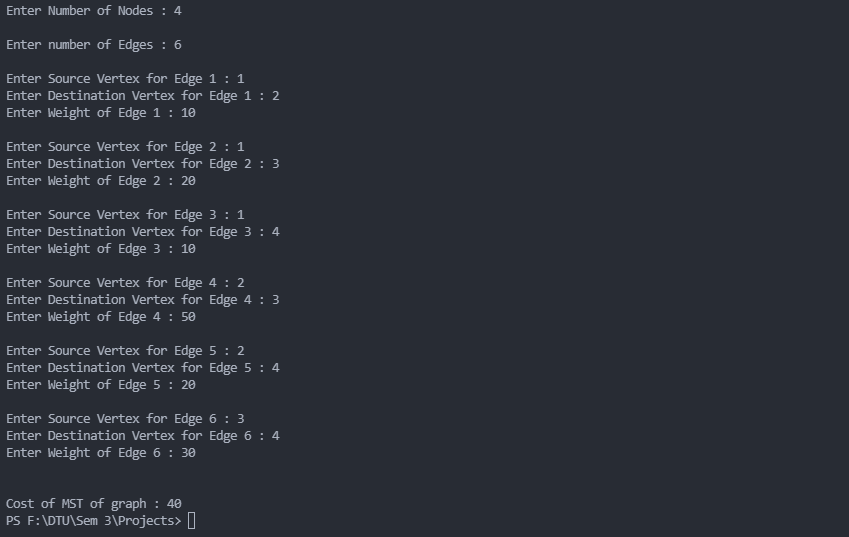


Output MST

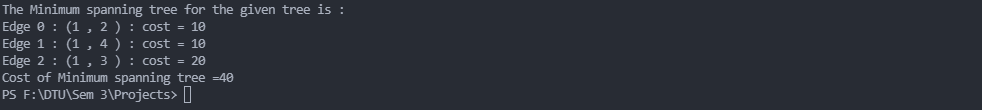
* MATLAB



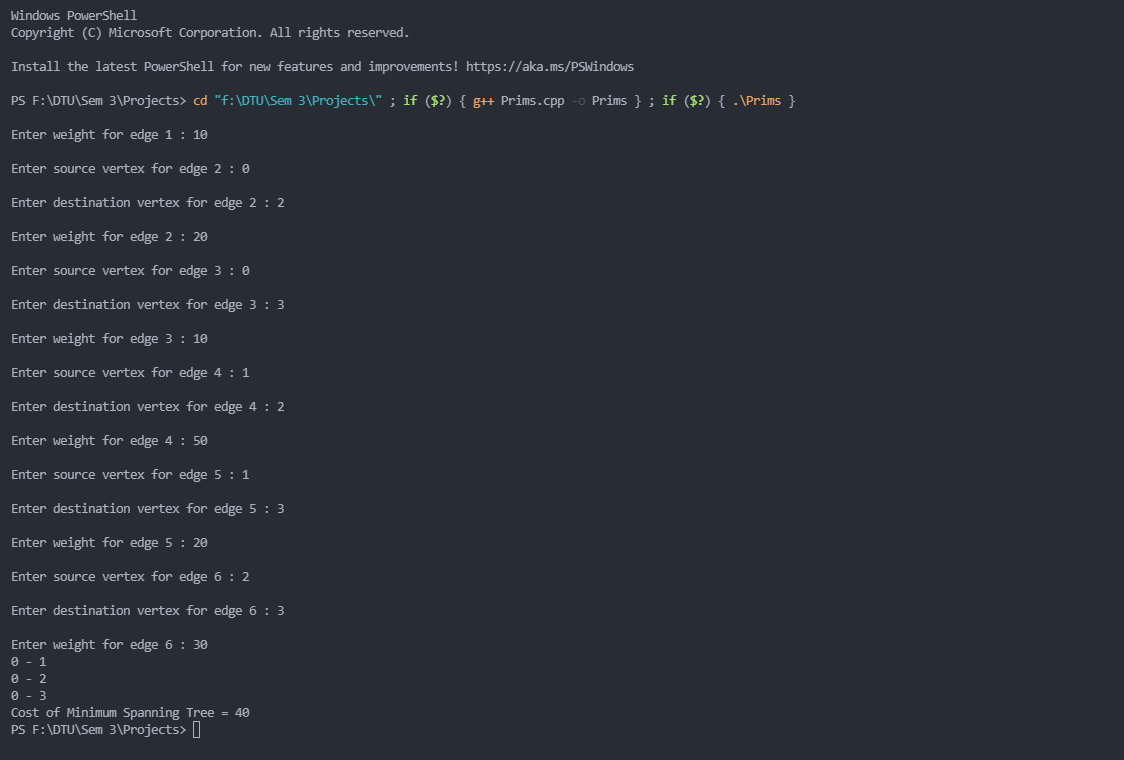
* Kruskal’s Algorithm



* Prim’s Algorithm (Brute Force Approach)



* Prim’s Algorithm (Priority Queue Approach)



# **References**

<https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/>

<http://chenlab.ece.cornell.edu/people/Andy/publications/Andy_files/Gallagher_cvpr2012_puzzleAssembly.pdf>

<https://en.wikipedia.org/wiki/Minimum_spanning_tree>

<https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>

<https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>

<https://en.wikipedia.org/wiki/Prim%27s_algorithm>