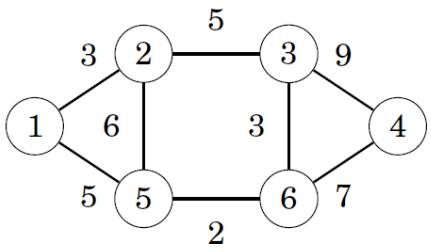
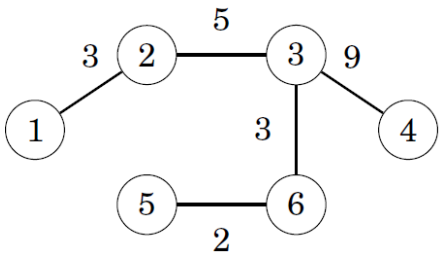
Spanning Trees

A spanning tree of a graph consist of all the nodes and some of the edges of the graph so that there exist a path between any two nodes. Like tree , spanning trees are connected and acyclic.

Consider the following graph



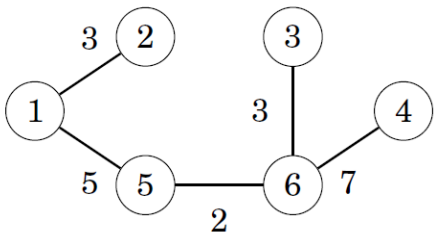
One spanning tree for the graph is as follows :



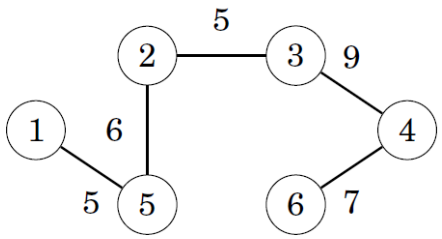
The weight of a spanning tree is the sum of its edge weights. For example, the weight of the spanning tree is 3+5+9+3+2=22.

Minimum Spanning Tree

A **minimum spanning tree** is a spanning tree whose weight is as small as possible. The weight of a minimum spanning tree for the example graph is 20 and such tree can be constructed as follows:



A **maximum spanning tree** is a spanning tree whose weight is as large as possible. The weight of a maximum spanning tree for the example graph is 32 and can be constructed as follows:



> If graph is a **complete graph** with n vertices then total no of spanning tree is n^(n-2) .
>
> If graph isn't **complete graph** then we can use Kirchoff's theorem to calculate spanning trees.

We can use the two algorithm discussed below to construct minimum spanning trees. Both are greedy algorithm The same algorithm can also find the maximum spanning trees by processing the edge in reverse order.

Kruskal's Algorithm

In **Kruskal's Algorithm** the initial spanning tree only contains the nodes of the graph and doesn't contain any edges. Then the algorithm goes through the edge ordered by their weights, and always add an edge to the tree if it doesn't create a cycle.

In Kruskal's algorithm, most time consuming operation is **sorting** because the total complexity of the Disjoint-Set operations will be O (E l o g V) , which is the overall Time Complexity of the algorithm.

Algorithm

1. *The first step of the algorithm is to sort the edges in increasing order of their weights.*
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.*
3. *Repeat step#2 until there are (V-1) edges in the spanning tree.*

Implementation

A class is created to store the information related to edge. It contains the information of source and destination to which the edge is connected.

```
class Edge{
    public:
    int src;
    int dest;
    int weight;
};
```

Initialize an output array of type edge to store the result of mst

```
Edge *output=new Edge[v-1];
```

The complete implementations is :

```
#include<bits/stdc++.h>
using namespace std;
class Edge{
    public:
    int src;
    int dest;
    int weight;
};
bool compare(Edge e1,Edge e2)
{
    return e1.weight<e2.weight;
}

//To store the rank, inorder to get a global parent for a connected component
class Set{
 public:
 int rank;
 int parent;
};
//retun the parent of the set
//In union rank algorithm we initially the parent of all element to those element
itself
int find(Set *set,int i){
  if(set[i].parent!=i){
      set[i].parent=find(set,set[i].parent);
  }
  return set[i].parent;
}
void Union(Set *set,int v1,int v2){
    int v1_root=find(set,v1);
    int v2_root=find(set,v2);
    //We will check rank of two set,if a particular set has lesser rank then
lower rank parent is updated
    if(set[v1_root].rank<set[v2_root].rank){
        set[v1_root].parent=v2_root;
    }else if(set[v1_root].rank>set[v2_root].rank){
        set[v2_root].parent=v1_root;
    }else{
        set[v2_root].parent=v1_root;
        set[v1_root].rank++;
    }
}
void Kruskals(Edge *input,int v,int e)
```

```
{
    //It will sort the input array on the basis of weight of edges
    sort(input,input+e,compare);
    //For a tree no of edges is vertex-1
    Edge *output=new Edge[v-1];
    //Initialised the set for all vertices
    Set *set = new Set[v];
    //for every set
    for(int i=0;i<v;i++){
        set[i].rank=0; //Initially all rank is zero
        set[i].parent=i; //Each node is a parent of own
    }
    int counter=0,i=0;
    while(counter<v-1){
        Edge currentEdge=input[i]; //We have taken the edge with minimum weight
        int sourceParent=find(set,currentEdge.src); //Find the parent of source
vertex
        int destinationParent=find(set,currentEdge.dest); //Find the parent of
distance vertex
        //if they arent same they dont belong to same set so there parent are
different
        if(sourceParent!=destinationParent){
            output[counter]=currentEdge;
            //Current set meh union kardia predefine function h
            Union(set,sourceParent,destinationParent);
            counter++; //increase the counter
        }
        i++;
    }
    for(int m=0;m<v-1;m++){
        cout<<output[m].src<<"---"<<output[m].dest<<" with weight"
<<output[m].weight<<endl;
    }
}
/*
7
8
0 3 4
0 1 6
1 2 5
3 2 7
3 4 2
4 5 4
5 6 1
4 6 3*/
int main()
{
    //No of vertices and no of edges
    int v,e;
    cin>>v>>e;
    Edge *input=new Edge[e];
    for(int i=0;i<e;i++){
        int s,d,w;
        cin>>s>>d>>w;
        input[i].src=s;
        input[i].dest=d;
        input[i].weight=w;
    }
```

```
    Kruskals(input,v,e);
    return 0;
}
```

Why does this work ? Why does the greedy strategy guarantee that we will find a minimum spanning tree?

Let us see what happens if the minimum weight edge of the graph is not included in the spanning tree. Then in that case between any two nodes the edge can have minimum weight than it current value which will help to decrease the weight of spanning tree. Hence the current tree isn't the minimum spanning tree.

So, here this work because, inorder to find minimum spanning tree we need to choose edges with minimum weight overall. An choosing minimum weight at each step would help us to achieve it.

Prims Algorithm

It is an alternative method to find a minimum spanning tree. The algorithm finds an arbitrary node to the tree. After this, the algorithm always choose a minimum-weight edge that adds a new node the tree. Finally all nodes have been added to the tree and minimum spanning tree has been found.

The time complexity of this algorithms is O(E log V)

> **Here we don't check cycle . Because we don't choose a vertex if it is already connected.**

The graph is stored in vector of vector with pairs.

```
vector<vector<pair<int,int>>>adj;
```

To add edge :

```
void addEdge(int u,int v,int w){
          adj[u].push_back(make_pair(v,w));
          adj[v].push_back(make_pair(u,w));
}
```

Choose any arbitrary node to start with. After that each time choose a vertex which can be reached from the visited node so far and edge weight of it is minimum.

```
 /*This give the vertex which hasn't been visited but can be visited only from
      the visited node.*/
      int findMinVertex(int *weight,bool *visited,int v){
          int minVertex = -1;
          for(int i=1;i<=v;i++){
              if(!visited[i] and (minVertex == -1 or weight[i]
<weight[minVertex])){
                  minVertex=i;
              }
          }
          return minVertex;
      }
```

The Prims implementation is :

```
 void Prims(){
          bool *visited = new bool[v+1];
          int *parent = new int[v+1];
          int *weight = new int [v+1];
          for(int i=0;i<=v;i++){
              visited[i] = false;
              weight[i] = inf;
          }
          //Parent and Weight Node for Arbitrary Node
          parent[1]=-1;
          weight[1]=0;

         for(int i=1;i<=v;i++){
              //Find the node which is not visited,distance is minimum
              int minVertex=findMinVertex(weight,visited,v);
              visited[minVertex]=true;

            for(auto neighbour:adj[minVertex]){
             /* I will explore the child from current vertex. If i get visited
vertex
             that is already part of my tree . So i shouldnt choose it.*/
              if(!visited[neighbour.first]){
                  /*A node can reach through multiple nodes. So we update it so
that
                  we can reach it through minimum edge weight.*/
                  if(weight[neighbour.first]>neighbour.second)//Comparing
weights
                  {
                      //Update the parent of the child
                      parent[neighbour.first]=minVertex;
                      //Update the weight
                      weight[neighbour.first]=neighbour.second;
                  }
              }
            }
          }
```

Prim's algorithm resembles Dijkstra's algorithm. The difference is that Dijkstra's algorithm always selects an edge whose distance from the starting node is minimum , but Prim's algorithm simply selects the minimum weight edge that adds a new node to the tree.

The complete implementation is :

```
#include<bits/stdc++.h>
using namespace std;
#define inf INT_MAX
class Graph{
    public:
      int v;
      //Vector of Vector
      //Index of array defines a vertex and pair defines destination vertex and
weight
      vector<vector<pair<int,int>>>adj;

      Graph(int v){
```

```
        this->v=v;
        adj.resize(v+1); // For 1 base indexing v+1
    }

    void addEdge(int u,int v,int w){
        adj[u].push_back(make_pair(v,w));
        adj[v].push_back(make_pair(u,w));
    }
    /*This give the vertex which hasn't been visited but can be visited only
from
    the visited node.*/
    int findMinVertex(int *weight,bool *visited,int v){
        int minVertex = -1;
        for(int i=1;i<=v;i++){
            if(!visited[i] and (minVertex == -1 or weight[i]
<weight[minVertex])){
                minVertex=i;
            }
        }
        return minVertex;
    }
    void Prims(){
        bool *visited = new bool[v+1];
        int *parent = new int[v+1];
        int *weight = new int [v+1];
        for(int i=0;i<=v;i++){
            visited[i] = false;
            weight[i] = inf;
        }
        //Parent and Weight Node for Arbitrary Node
        parent[1]=-1;
        weight[1]=0;

       for(int i=1;i<=v;i++){
            //Find the node which is not visited,distance is minimum
            int minVertex=findMinVertex(weight,visited,v);
            visited[minVertex]=true;

          for(auto neighbour:adj[minVertex]){
           /* I will explore the child from current vertex. If i get visited
vertex
            that is already part of my tree . So i shouldnt choose it.*/
            if(!visited[neighbour.first]){
                /*A node can reach through multiple nodes. So we update it so
that
                we can reach it through minimum edge weight.*/
                if(weight[neighbour.first]>neighbour.second)//Comparing
weights
                {
                    //Update the parent of the child
                    parent[neighbour.first]=minVertex;
                    //Update the weight
                    weight[neighbour.first]=neighbour.second;
                }
            }
          }
        }
```

```
            for(int i=1;i<=v;i++){
                cout<<i<<" --"<<parent[i]<<" with weight "<<weight[i]<<endl;
            }
        }
};


int main()
{
    int n,e;
    cin>>n>>e;
    Graph g(n);
    int u,v,w;
    for(int i=0;i<e;i++){
        cin>>u>>v>>w;
        g.addEdge(u,v,w);
    }
    g.Prims();
}
```

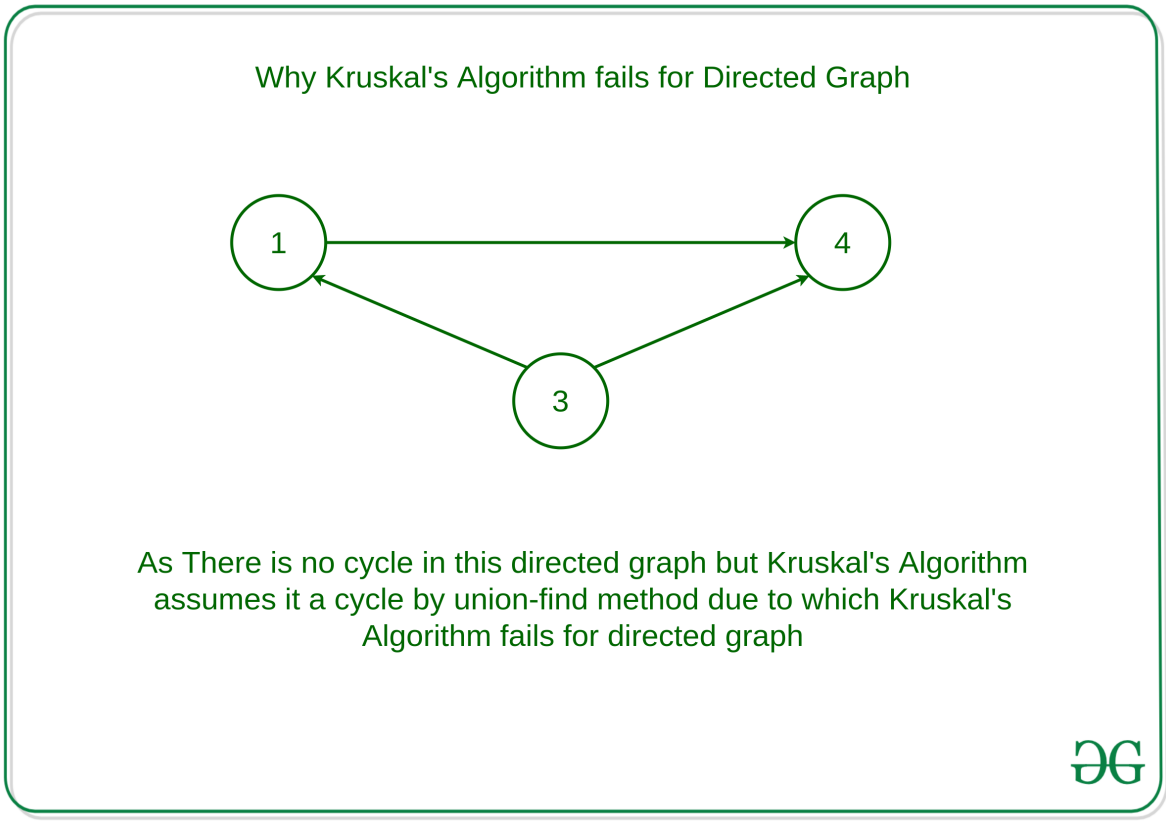
Important

Prims and Kruskal's Algorithm doesn't work for directed graph.

Why Prims Fail for directed graph ?

Prim's algorithm assumes that all vertices are connected. But in a directed graph, every node is not reachable from every other node. So, Prim's algorithm fails due to this reason.

Why Kruskal's Algorithm fails for directed graph ?



When should we use Prims or Kruskal ?

We should use Kruskal when the graph is sparse, i.e. small number of edges,like E=O(V),when the edges **are** already sorted or if **we can** sort them in linear time. **We should use Prim** when the graph is dense, i.e number of edges is high ,like $E=O(V^2)$.