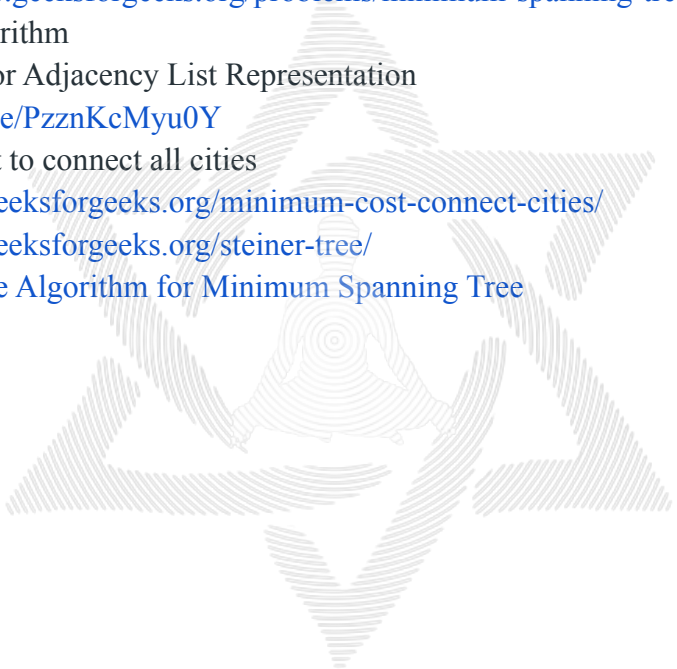
Graphs
Lesson 4

1. What is MST?
2. How many edges does a minimum spanning tree have?
3. Total number of Spanning Trees in a Graph
   https://www.geeksforgeeks.org/total-number-spanning-trees-graph/
   https://youtu.be/NcOYysnlx-0
4. Prim's Minimum Spanning Tree
   https://practice.geeksforgeeks.org/problems/minimum-spanning-tree/1
5. Kruskal's algorithm
6. Prim's MST for Adjacency List Representation
   https://youtu.be/PzznKcMyu0Y
7. Minimum cost to connect all cities
   https://www.geeksforgeeks.org/minimum-cost-connect-cities/
8. https://www.geeksforgeeks.org/steiner-tree/
9. Reverse Delete Algorithm for Minimum Spanning Tree



Spanning tree

If we have a graph containing V vertices and E edges, then the graph can be represented as:

G(V, E)

If we create the spanning tree from the above graph, then the spanning tree would have the same number of vertices as the graph, but the vertices are not equal. The edges in the spanning tree would be equal to the number of edges in the graph minus 1.

Suppose the spanning tree is represented as:

G`(V`, E`)

where,

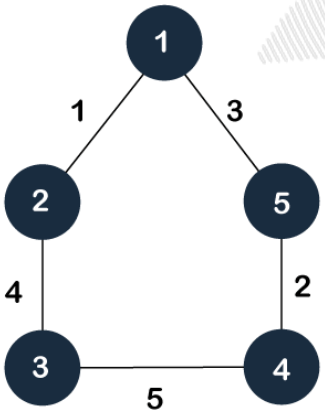
V=V`

E`ϵ E -1

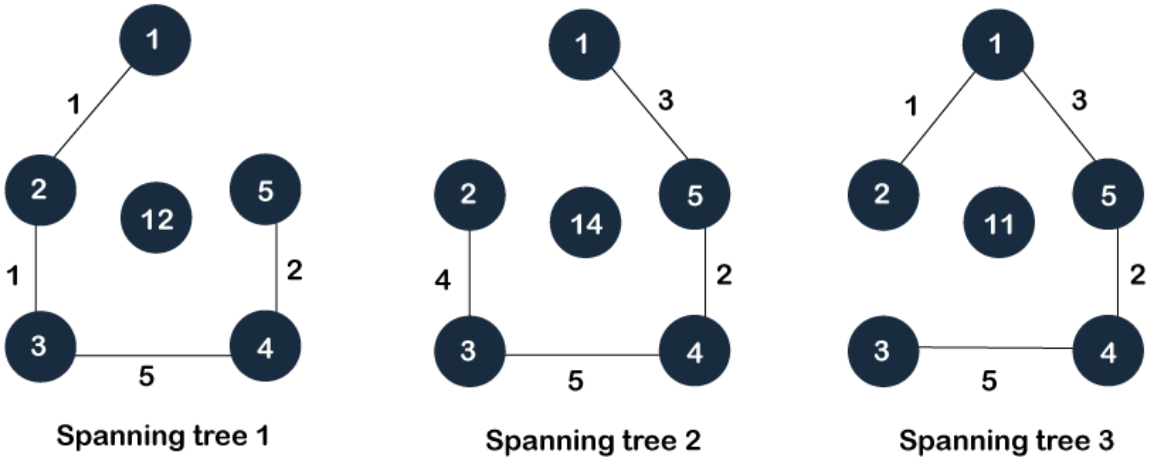
E`=|V| - 1

Let's understand through an example.

Suppose we want to create the spanning tree of the graph, which is shown below:



As we know, that spanning tree contains the same number of vertices as the graph, so the total number of vertices in the graph is 5; therefore, the spanning tree will also contain the 5 vertices. The edges in the spanning tree are equal to the number of vertices in the graph minus 1; therefore, the number of edges is 4. Three spanning trees can be created, which are shown below:

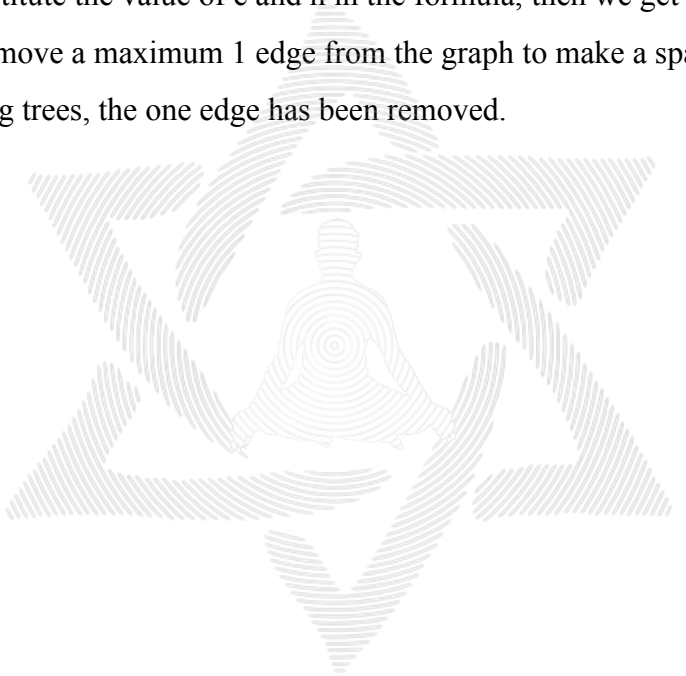


Properties of Spanning tree

- A connected graph can contain more than one spanning tree. The spanning trees which are minimally connected or we can say that the tree which is having a minimum total edge weight would be considered as the minimum spanning tree.
- All the possible spanning trees that can be created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.
- The spanning tree does not contain any cycle. Let's understand this property through an example.
- As we can observe in the above spanning trees that one edge has been removed. If we do not remove one edge from the graph, then the tree will form a cycle, and that tree will not be considered as the spanning tree.
- The spanning tree cannot be disconnected. If we remove one more edge from any of the above spanning trees as shown below:
  The above tree is not a spanning tree because it is disconnected now.
- If two or three edges have the same edge weight, then there would be more than two minimum spanning trees. If each edge has a distinct weight, then there will be only one or unique minimum spanning tree.
- A complete undirected graph can have n^{n-2} number of spanning trees where n is the number of vertices in the graph. For example, the value of n is 5 then the number of spanning trees would be equal to 125.

- Each connected and undirected graph contains at least one spanning tree.
- The disconnected graph does not contain any spanning tree, which we have already discussed.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges. Let's understand this property through an example.

  A complete graph is a graph in which each pair of vertices are connected.
- According to this property, the maximum number of edges from the graph can be formulated as (e-n+1) where e is the number of edges, n is the number of vertices. When we substitute the value of e and n in the formula, then we get 1 value. It means that we can remove a maximum 1 edge from the graph to make a spanning tree. In the above spanning trees, the one edge has been removed.



Total number of Spanning Trees in a Graph

If a graph is a complete graph with n vertices, then the total number of spanning trees is n(n-2) where n is the number of nodes in the graph. In the complete graph, the task is equal to counting different labeled trees with n nodes for which have Cayley's formula.

What if the graph is not complete?

Follow the given procedure :-

STEP 1: Create Adjacency Matrix for the given graph.

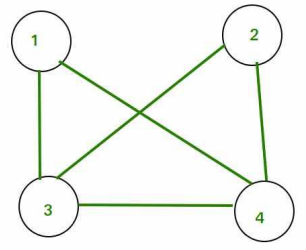
STEP 2: Replace all the diagonal elements with the degree of nodes. For eg. element at (1,1) position of adjacency matrix will be replaced by the degree of node 1, element at (2,2) position of adjacency matrix will be replaced by the degree of node 2, and so on.

STEP 3: Replace all non-diagonal 1's with -1.

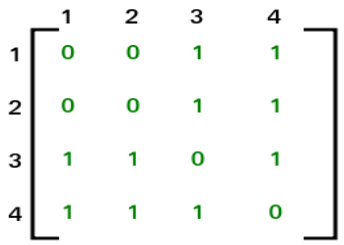
STEP 4: Calculate cofactor for any element.

STEP 5: The cofactor that you get is the total number of spanning tree for that graph.

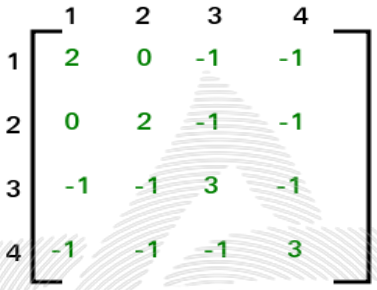
Consider the following graph:



The Adjacency Matrix for the above graph will be as follows:



After applying STEP 2 and STEP 3, adjacency matrix will look like



The cofactor for (1, 1) is 8. Hence the total no. of spanning trees that can be formed is 8.NOTE- Cofactor for all the elements will be same. Hence we can compute cofactors for any element of the matrix.

What is a Minimum Spanning Tree?

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a 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.

How many edges does a minimum spanning tree have?

A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Prim's Minimum Spanning Tree

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 sets 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 vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim's Algorithm Work? 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. They must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Algorithm

1) Create a set mstSet that keeps track of vertices already included in MST.

2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.

3) While mstSet doesn't include all vertices

….a) Pick a vertex u which is not there in mstSet and has a minimum key value.

….b) Include u to mstSet.

….c) Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

The idea of using key values is to pick the minimum weight edge from the cut. The key values are used only for vertices which are not yet included in MST; the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

How to implement the above algorithm?
We use a boolean array mstSet[] to represent the set of vertices included in MST. If a value mstSet[v] is true, then vertex v is included in MST, otherwise not. Array key[] is used to store key values of all vertices. Another array parent[] to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

```
1.  #include <bits/stdc++.h>
2.  using namespace std;
3.  #define I INT_MAX
4.  class Graph
5.  {
6.  private:
7.     int n;
8.     vector<vector<int>> adj;
9.  public:
10.    Graph(int n)
11.    {
12.       this->n=n;
13.       int i;
14.       vector<int> k;
15.       for(i=0; i<=n; i++)
16.          k.push_back(I);
17.       for(i=0; i<=n; i++)
18.          adj.push_back(k);
19.    }
20.    void add_edge(int u, int v, int weight)
21.    {
22.       adj[u][v]=weight;
23.       adj[v][u]=weight;
24.    }
25.    void minimum_spanning_tree()
26.    {
27.       vector<pair<int,int>> span;
28.       int minm=I;
29.       int i,j,u,v;
30.       for(i=1; i<=n; i++)
31.       {
32.          for(j=i; j<=n; j++)
```

```
33.         {
34.             if(minm>adj[i][j])
35.             {
36.                 minm=adj[i][j];
37.                 u=i;
38.                 v=j;
39.             }
40.         }
41.     }
42.     span.push_back({u,v});
43.     vector<int> near(n+1,-1);
44.     near[u]=0;
45.     near[v]=0;
46.     for(i=1; i<=n; i++)
47.     {
48.         if(near[i]!=0)
49.         {
50.             if(adj[u][i]>adj[v][i])
51.                 near[i]=v;
52.             else
53.                 near[i]=u;
54.         }
55.     }
56.     for(i=2; i<=n-1; i++)
57.     {
58.         minm=I;
59.         for(j=1; j<=n; j++)
60.         {
61.             if(near[j]!=0 && adj[j][near[j]]<minm)
62.             {
63.                 minm=adj[j][near[j]];
64.                 u=j;
65.                 v=near[j];
66.             }
67.         }
68.         span.push_back({v,u});
69.         near[u]=0;
```

```
70.          for(j=1; j<=n; j++)
71.          {
72.             if(near[j]!=0 && adj[j][near[j]]>adj[u][j])
73.             {
74.                near[j]=u;
75.             }
76.          }
77.       }
78.       int cost=0;
79.       for(i=0; i<n-1; i++)
80.       {
81.          cost+=adj[span[i].first][span[i].second];
82.          cout<<span[i].first<<"->"<<span[i].second<<endl;
83.       }
84.       cout<<"Cost of spanning tree is "<<cost<<endl;
85.    }
86. };
87. int main()
88. {
89.    cout<<"Enter no. of vertices and edges: ";
90.    int n,e;
91.    cin>>n>>e;
92.    Graph G(n);
93.    int u,v,weight;
94.    cout<<"Enter the edges and there weights:- "<<endl;
95.    for(int i=0; i<e; i++)
96.    {
97.       cin>>u>>v>>weight;
98.       G.add_edge(u,v,weight);
99.    }
100.       cout<<"Spanning Tree is "<<endl;
101.       G.minimum_spanning_tree();
102.       return 0;
103.    }
```

Kruskal's Minimum Spanning Tree Algorithm

Below are the steps for finding MST using Kruskal's algorithm

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 the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

```
1.  #include <bits/stdc++.h>
2.  using namespace std;
3.  class Edge
4.  {
5.  public:
6.     int u;
7.     int v;
8.     int weight;
9.  };
10. class Union_Find
11. {
12. private:
13.    int n;
14.    int *s;
15. public:
16.    Union_Find(int n)
17.    {
18.       this->n=n;
19.       s=new int[n+1];
20.       int i;
21.       for(i=0; i<n+1; i++)
22.          s[i]=-1;
23.    }
24.    void Union(int u, int v)
25.    {
26.       if(s[u]<s[v])
27.       {
28.          s[u]=s[u]+s[v];
29.          s[v]=u;
```

```
30.       }
31.       else
32.       {
33.          s[v]=s[u]+s[v];
34.          s[u]=v;
35.       }
36.    }
37.    int Find(int u)
38.    {
39.      int x=u;
40.      while(s[x]>0)
41.          x=s[x];
42.      return x;
43.    }
44. };
45.
46. bool compare(Edge p, Edge q)
47. {
48.    return p.weight<q.weight;
49. }
50.
51. class Graph
52. {
53. private:
54.    int n,e;
55.    Edge *edges;
56.    int in;
57. public:
58.    Graph(int n, int e)
59.    {
60.       this->e=e;
61.       this->n=n;
62.       edges=new Edge[e];
63.       in=0;
64.    }
65.    void add_edge(int u, int v, int weight)
66.    {
```

```
67.       edges[in].u=u;
68.       edges[in].v=v;
69.       edges[in].weight=weight;
70.       in++;
71.    }
72.    vector<vector<int>> MST()
73.    {
74.       Union_Find U(n);
75.       vector<vector<int>> ans;
76.       sort(edges,edges+e,compare);
77.       int i=0,weight,u,v,j=0;
78.       while(i<n-1)
79.       {
80.          u=edges[j].u;
81.          v=edges[j].v;
82.          weight=edges[j].weight;
83.          j++;
84.          int p=U.Find(u);
85.          int q=U.Find(v);
86.          if(p!=q)
87.          {
88.             vector<int> span_edge;
89.             span_edge.push_back(u);
90.             span_edge.push_back(v);
91.             span_edge.push_back(weight);
92.             ans.push_back(span_edge);
93.             U.Union(p,q);
94.             i++;
95.          }
96.       }
97.       return ans;
98.    }
99. };
100.
101.    int main()
102.    {
103.       cout << "Enter the no. of vertices and edges in graph ";
```

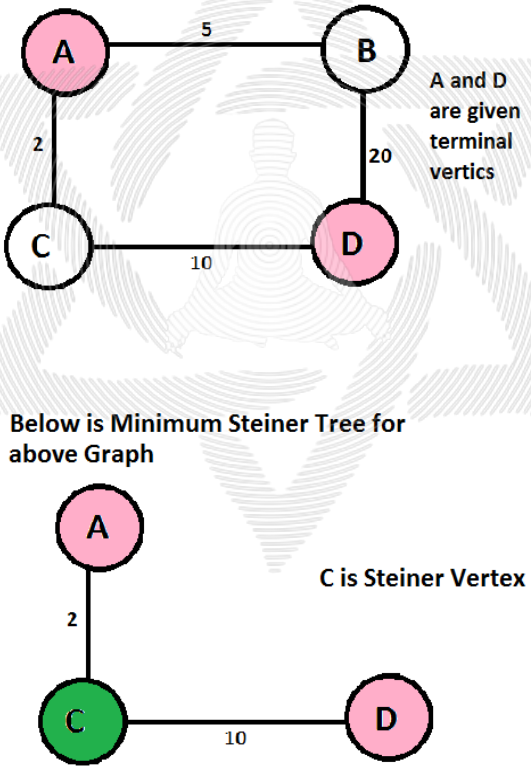
```
104.     int n,e;
105.     cin>>n>>e;
106.     Graph G(n,e);
107.     int i,u,v,weight;
108.     cout<< "Enter the edges and there corresponding weights:-"<<endl;
109.     for(i=0; i<e; i++)
110.     {
111.         cin>>u>>v>>weight;
112.         G.add_edge(u,v,weight);
113.     }
114.     vector<vector<int>> p=G.MST();
115.     int cost=0;
116.     cout<<"\nSpanning Tree is:-\n";
117.     for(i=0; i<n-1; i++)
118.     {
119.         cost+=p[i][2];
120.         cout<<p[i][0]<<" "<<p[i][1]<<" "<<p[i][2]<<endl;
121.     }
122.     cout<<"Cost of MST is "<<cost;
123.     return 0;
124. }
125.
```

Steiner Tree Problem

Given a graph and a subset of vertices in the graph, a steiner tree spans through the given subset. The Steiner Tree may contain some vertices which are not in a given subset but are used to connect the vertices of the subset.

The given set of vertices is called Terminal Vertices and other vertices that are used to construct Steiner trees are called Steiner vertices.

The Steiner Tree Problem is to find the minimum cost of a Steiner Tree. See below for an example.



Spanning Tree vs Steiner Tree

Minimum Spanning Tree is a minimum weight tree that spans through all vertices.

If the given subset (or terminal) vertices is equal to the set of all vertices in the Steiner Tree problem, then the problem becomes the Minimum Spanning Tree problem. And if the given subset contains only two vertices, then it is the shortest path problem between two vertices.

Finding the Minimum Spanning Tree is polynomial time-solvable, but the Minimum Steiner Tree problem is NP-Hard and related decision problems are NP-Complete.

Applications of Steiner Tree

Any situation where the task is to minimize cost of connection among some important locations like VLSI Design, Computer Networks, etc.

Shortest Path based Approximate Algorithm

Since the Steiner Tree problem is NP-Hard, there are no polynomial time solutions that always give optimal cost. Therefore, there are approximate algorithms to solve the same problem. Below is one simple approximate algorithm.

1) Start with a subtree T consisting of
   one given terminal vertex
2) While T does not span all terminals
   a) Select a terminal x not in T that is closest
      to a vertex in T.
   b) Add to T the shortest path that connects x with T

The above algorithm is (2-2/n) approximate, i.e., it guarantees that the solution produced by this algorithm is not more than this ratio of optimized solution for a given graph with n vertices.

Reverse Delete Algorithm for Minimum Spanning Tree

The Reverse Delete algorithm is closely related to Kruskal's algorithm. In the Reverse Delete algorithm, we sort all edges in decreasing order of their weights. After sorting, we one by one pick edges in decreasing order. We include the current picked edge if excluding the current edge causes disconnection in the current graph. The main idea is delete edge if its deletion does not lead to disconnection of the graph. Your task is to print the value of the total weight of the Minimum Spanning Tree formed.

Example 1:

Input:
V = 4, E = 5
Arr[] = {0, 1, 10, 0, 2, 6, 0, 3, 5, 1, 3, 15, 2, 3, 4}
Output:
19
Explanation:
The weight of the Minimum Spanning
Tree formed is 19.

Example 1:

Input:
V = 4, E = 3
Arr[] = {0, 1, 98, 1, 3, 69, 0, 3, 25}
Output:
192
Explanation:
The weight of the Minimum Spanning
Tree formed is 192.

Expected Time Complexity: O(V*E)

Expected Auxiliary Space: O(E)

Constraints:

1 <= V,E <= 1000

1 <= u,v <= V

1 <= w <= 100