Prim's Minimum Spanning Tree

In this tutorial we will cover another algorithm which uses greedy approach/technique for finding the solution.

Let's start with a real-life scenario to understant the premise of this algorithm:

1. A telecommunications organization, has offices spanned across multiple locations around the globe.

Figure 1

2. It has to use leased phone lines for connecting all these offices with each other.
3. The cost(in units) of connecting each pair of offices is different and is shown as follows:

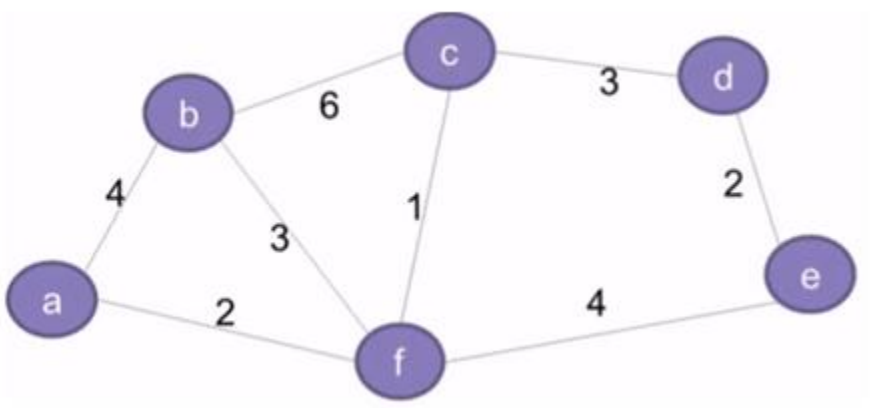

Figure 2

4. The organization, thus, wants to use minimum cost for connecting all its offices. This requires that all the offices should be connected using minimum number of leased lines so as to reduce the effective cost.
5. The solution to this problem can be implemented by using the concept of **Minimum Spanning Tree**, which is discussed in the subsequent section.
6. This tutorial also details the concepts related to Prim's Algorithm which is used for finding the minimum spanning tree for a given graph.

What is a Spanning Tree?

The network shown in the second figure basically represents a graph **G = (V, E)** with a set of vertices **V = {a, b, c, d, e, f}** and a set of edges **E = { (a,b), (b,c), (c,d), (d,e), (e,f), (f,a), (b,f), (c,f) }**. The graph is:

- Connected (there exists a path between every pair of vertices)

- Undirected (the edges do no have any directions associated with them such that (a,b) and (b,a) are equivalent)
- Weighted (each edge has a weight or cost assigned to it)

A spanning tree $G' = (V, E')$ for the given graph G will include:

- All the vertices (V) of G
- All the vertices should be connected by minimum number of edges (E') such that $E' \subset E$
- G' can have maximum $n-1$ edges, where n is equal to the total number of edges in G
- G' should not have any cycles. This is one of the basic differences between a tree and graph that **a graph can have cycles, but a tree cannot**. Thus, a tree is also defined as an **acyclic graph**.

Following is an example of a spanning tree for the above graph. Please note that only the highlighted edges are included in the spanning tree,

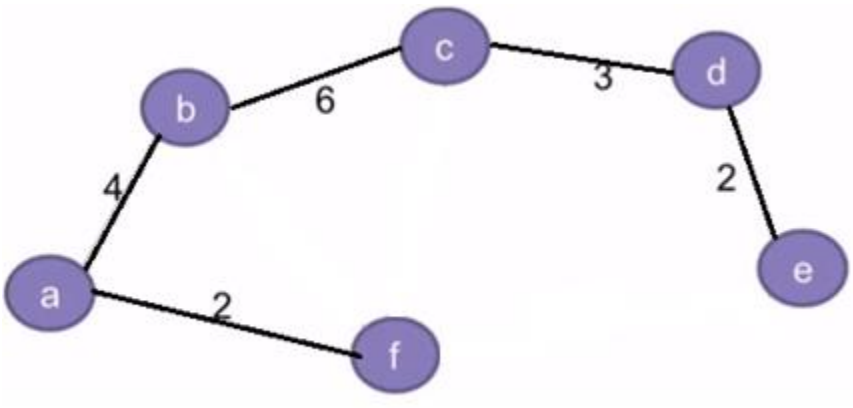

Figure 3

Also, there can be multiple spanning trees possible for any given graph. For eg: In addition to the spanning tree in the above diagram, the graph can also have another spanning tree as shown below:

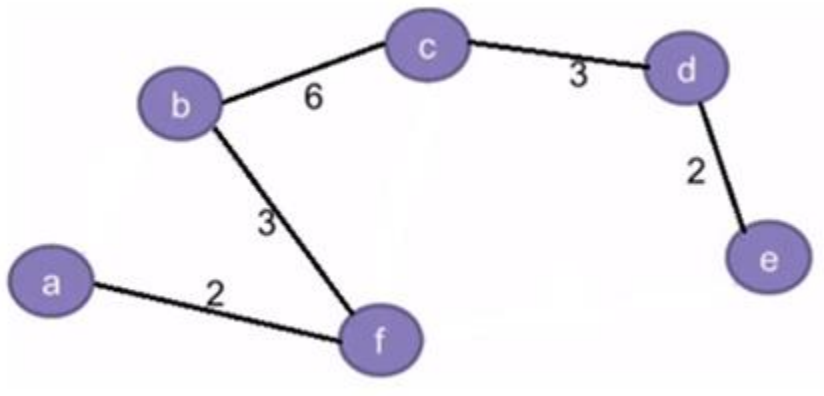

Figure 4

By convention, the total number of spanning trees for a given graph can be defined as:

${}^{n}C_{m} = n!/(m!*(n-m)!)$, where,

- n is equal to the total number of edges in the given graph
- m is equal to the total number of edges in the spanning tree such that $m <= (n-1)$.

Hence, the total number of spanning trees(S) for the given graph(second diagram from top) can be computed as follows:

- **n = 8**, for the given graph in Fig. 2
- **m = 5**, since its corresponding spanning tree can have only 5 edges. Adding a 6th edge can result in the formation of cycles which is not allowed.
- So, **$S = {}^{n}C_{m} = {}^{8}C_{5} = 8!/ (5! * 3!) = 56$**, which means that **56** different variations of spannig trees can be created for the given graph.

What is a Minimum Spanning Tree?

The cost of a spanning tree is the total of the weights of all the edges in the tree. For example, the cost of spanning tree in Fig. 3 is **(2+4+6+3+2) = 17** units, whereas in Fig. 4 it is **(2+3+6+3+2) = 16** units.

Since we can have multiple spanning trees for a graph, each having its own cost value, the objective is to find the spanning tree with minimum cost. This is called a **Minimum Spanning Tree(MST)**.

Note: There can be multiple minimum spanning trees for a graph, if any two edges in the graph have the same weight. However, if each edge has a distinct weight, then there will be only one minimum spanning tree for any given graph.

Problem Statement for Minimum Spanning Tree

Given a weighted, undirected and connected graph **G**, the objective is to find the minimum spanning tree **G'** for G.

Apart from the Prim's Algorithm for minimum spanning tree, we also have Kruskal's Algorithm for finding minimum spanning tree.

However, this tutorial will only discuss the fundamentals of **Prim's Algorithm.**

Since this algorithm aims to find the spanning tree with minimum cost, it uses **greedy approach** for finding the solution.

As part of finding the or creating the minimum spanning tree fram a given graph we will be following these steps:

- Initially, the tree is empty.
- The tree starts building from a random source vertex.
- A new vertex gets added to the tree at every step.
- This continues till all the vertices of graph are added to the tree.

Input Data will be:

A **Cost Adjacency Matrix** for out graph **G**, say `cost`

Output will be:

A Spanning tree with minimum total cost

Algorithm for Prim's Minimum Spanning Tree

Below we have the complete logic, stepwise, which is followed in prim's algorithm:

Step 1: Keep a track of all the vertices that have been visited and added to the spanning tree.

Step 2: Initially the spanning tree is empty.

Step 3: Choose a random **vertex**, and add it to the spanning tree. This becomes the **root node**.

Step 4: Add a new vertex, say **x**, such that

a. **x** is not in the already built spanning tree.

b. **x** is connected to the built spanning tree using minimum weight edge. (Thus, **x** can be adjacent to any of the nodes that have already been added in the spanning tree).

c. Adding **x** to the spanning tree should not form cycles.

Step 5: Repeat the Step 4, till all the vertices of the graph are added to the spanning tree.

Step 6: Print the total cost of the spanning tree.

Example for Prim's Minimum Spanning Tree Algorithm

Let's try to trace the above algorithm for finding the Minimum Spanning Tree for the graph in Fig. 2:

Step A:

a. Define `key[]` array for storing the key value(or cost) of every vertex. Initialize this to ∞(infinity) for all the vertices

b. Define another array `booleanvisited[]` for keeping a track of all the vertices that have been added to the spanning tree. Initially this will be **0** for all the vertices, since the spanning tree is empty.

c. Define an array `parent[]` for keeping track of the parent vertex. Initialize this to **-1** for all the vertices.

d. Initialize minimum cost, **minCost = 0**

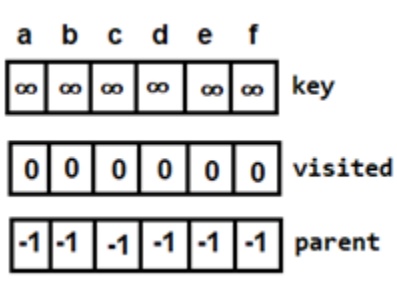

Figure 5: Initial arrays when tree is empty

Step B:

Choose any random vertex, say **f** and set `key[f]=0`.

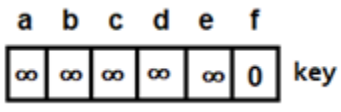

Figure 6: Setting key value of root node

Since its key value is minimum and it is not visited, add **f** to the spanning tree.

Figure 7: Root Node

Also, update the following:

- `minCost = 0 + key[f] = 0`
- This is how the `visited[]` array will look like:

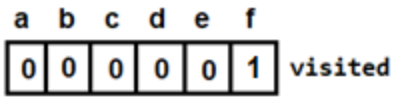

Figure 8: visited array after adding the root node

- Key values for all the adjacent vertices of **f** will look like this(key value is nothing but the cost or the weight of the edge, for **(f,d)** it is still infinity because they are not directly connected):

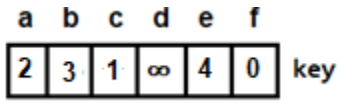

Figure 9: key array after adding the root node

Note: There will be no change in the `parent[]` because **f** is the root node.

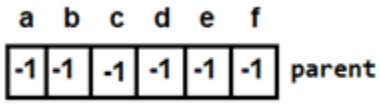

Figure 10: parent array after adding the root node

Step C:

The arrays `key[]` and `visited[]` will be searched for finding the next vertex.

- **f** has the minimum key value but will not be considered since it is already added (`visited[f]==1`)
- Next vertex having the minimum key value is **c**. Since `visited[c]==0`, it will be added to the spanning tree.

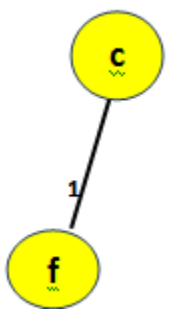

Figure 11: Adding vertex c

Again, update the following:

- `minCost = 0 + key[c] = 0 + 1 = 1`
- This is how the `visited[]` array will look like:

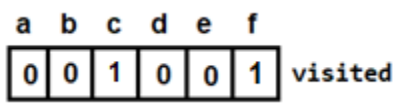

Figure 12: visited array after adding vertex c

- And, the `parent[]` array (**f** becomes the parent of **c**):

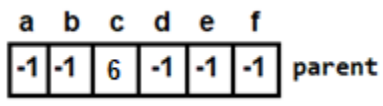

Figure 13: parent array after adding the root node

- For every adjacent vertex of **c**, say **v**, values in `key[v]` will be updated using the formula:

  `key[v] = min(key[v], cost[c][v])`

  Thus the `key[]` array will become:

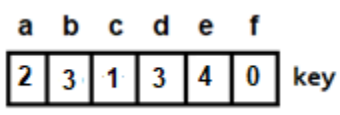

Figure 14: key array after adding the root node

Step D:

Repeat the Step C for the remaining vertices.

- Next vertex to be selected is **a**. And minimum cost will become `minCost=1+2=3`

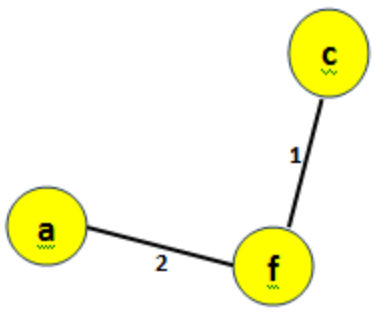

Figure 15: Adding vertex a

a	b	c	d	e	f	
1	0	1	0	0	1	visited
6	-1	6	-1	-1	-1	parent
2	3	1	3	4	0	key

Figure 16: Updated arrays after adding vertex a

- Next, either **b** or **d** can be selected. Let's consider **b**. Then the minimum cost will become $\text{minCost}=3+3=6$

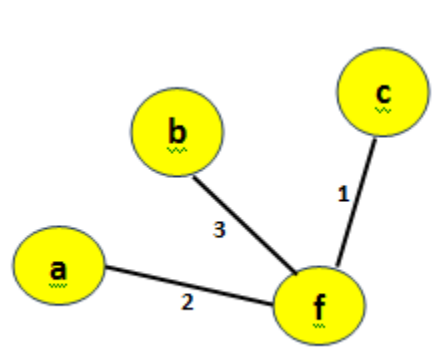

Figure 17: Adding vertex b to minimum spanning tree

a	b	c	d	e	f	
1	1	1	0	0	1	visited
6	6	6	-1	-1	-1	parent
2	3	1	3	4	0	key

Figure 18: Updated arrays after adding vertex b

- Next vertex to be selected is **d**, making the minimum cost $\text{minCost}=6+3=9$

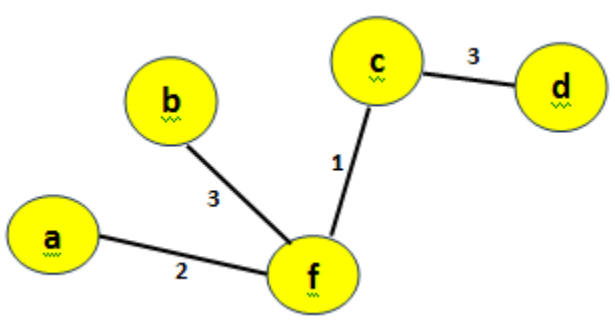

Figure 19: Adding vertex d

a	b	c	d	e	f	
1	1	1	1	0	1	visited
6	6	6	3	-1	-1	parent
2	3	1	3	2	0	key

Figure 20: Updated arrays after adding vertex d

- Then, **e** is selected and the minimum cost will become, $\text{minCost}=9+2=11$

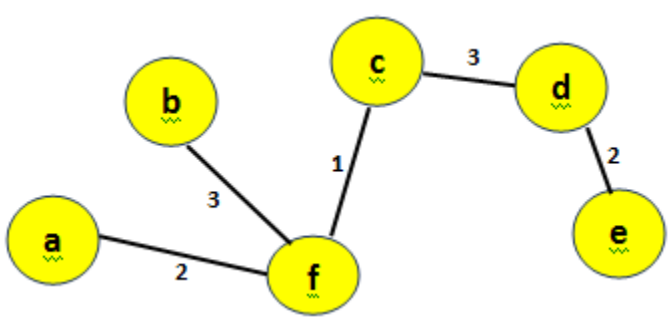

Figure 21: Adding vertex e. This is the final minimum spanning tree

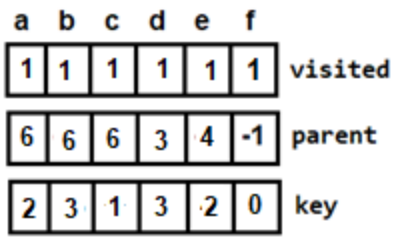

Figure 22: Updated arrays after adding vertex e (final arrays)

- Since all the vertices have been visited now, the algorithm terminates.
- Thus, Fig. 21 represents the Minimum Spanning Tree with total **cost=11**.

Implementation of Prim's Minimum Spanning Tree Algorithm

Now it's time to write a program in C++ for the finding out minimum spanning tree using prim's algorithm.

```
#include<iostream>


using namespace std;


// Number of vertices in the graph
const int V=6;


// Function to find the vertex with minimum key value
int min_Key(int key[], bool visited[])
{
    int min = 999, min_index;  // 999 represents an Infinite value


    for (int v = 0; v < V; v++) {
        if (visited[v] == false && key[v] < min) {
          // vertex should not be visited
            min = key[v];
                min_index = v;
        }
    }
    return min_index;
}
```

```
// Function to print the final MST stored in parent[]
int print_MST(int parent[], int cost[V][V])
{
    int minCost=0;
     cout<<"Edge \tWeight\n";
    for (int i = 1; i< V; i++) {
          cout<<parent[i]<<" - "<<i<<" \t"<<cost[i][parent[i]]<<" \n";
          minCost+=cost[i][parent[i]];
    }
     cout<<"Total cost is"<<minCost;
}

// Function to find the MST using adjacency cost matrix representation
void find_MST(int cost[V][V])
{
    int parent[V], key[V];
    bool visited[V];

    // Initialize all the arrays
    for (int i = 0; i< V; i++) {
        key[i] = 999;    // 99 represents an Infinite value
        visited[i] = false;
        parent[i]=-1;
    }

    key[0] = 0;  // Include first vertex in MST by setting its key vaue
to 0.
    parent[0] = -1; // First node is always root of MST

    // The MST will have maximum V-1 vertices
```

```
    for (int x = 0; x < V - 1; x++)

    {

        // Finding the minimum key vertex from the

        //set of vertices not yet included in MST

        int u = min_Key(key, visited);


        visited[u] = true;  // Add the minimum key vertex to the MST


        // Update key and parent arrays

        for (int v = 0; v < V; v++)

        {

            // cost[u][v] is non zero only for adjacent vertices of u

            // visited[v] is false for vertices not yet included in MST

            // key[] gets updated only if cost[u][v] is smaller than
key[v]

            if (cost[u][v]!=0 && visited[v] == false && cost[u][v] <
key[v])

            {

                parent[v] = u;

                key[v] = cost[u][v];

            }

        }


    }


    // print the final MST

     print_MST(parent, cost);

}


// main function

int main()

{
```

```
    int cost[V][V];

     cout<<"Enter the vertices for a graph with 6 vetices";

    for (int i=0;i<V;i++)

    {

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

        {

                cin>>cost[i][j];

        }

    }

     find_MST(cost);



        return 0;

}
```

Copy

The input graph is the same as the graph in Fig 2.

```
Enter the vertices for a graph with 6 vetices
0 4 0 0 0 2
4 0 6 0 0 3
0 6 0 3 0 1
0 0 3 0 2 0
0 0 0 2 0 4
2 3 1 0 4 0
Edge    Weight
5 - 1   3
5 - 2   1
2 - 3   3
3 - 4   2
0 - 5   2
Total cost is11
```

Figure 23: Output of the program

Time Complexity Analysis for Prim's MST

Time complexity of the above C++ program is **O(V2)** since it uses adjacency matrix representation for the input graph. However, using an adjacency list representation, with the help of binary heap, can reduce the complexity of Prim's algorithm to **O(ElogV)**.

Real-world Applications of a Minimum Spanning Tree

Finding an MST is a fundamental problem and has the following real-life applications:

1. Designing the networks including computer networks, telecommunication networks, transportation networks, electricity grid and water supply networks.

2. Used in algorithms for approximately finding solutions to problems like Travelling Salesman problem, minimum cut problem, etc.
    - The objective of a **Travelling Salesman problem** is to find the shortest route in a graph that visits each vertex only once and returns back to the source vertex.
    - A **minimum cut problem** is used to find the minimum number of cuts between all the pairs of vertices in a planar graph. A graph can be classified as planar if it can be drawn in a plane with no edges crossing each other. For example,

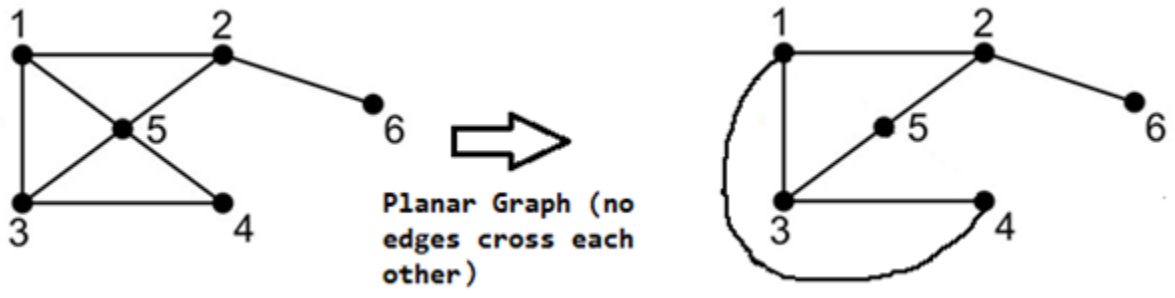

Figure 24: Planar Graph

3. Also, a cut is a subset of edges which, if removed from a planar graph, increases the number of components in the graph

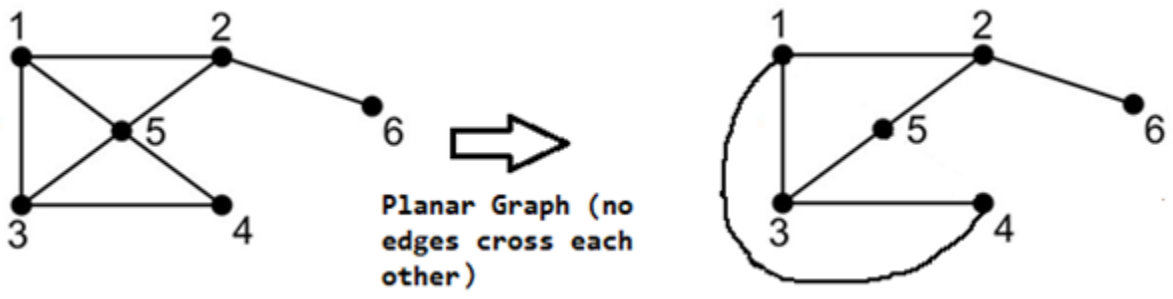

Figure 25: Cut-Set in a Planar Graph

4. Analysis of clusters.
5. Handwriting recognition of mathematical expressions.
6. Image registration and segmentation