



EXPERIMENT - 7

Algorithms Design and Analysis Lab

Aim

To implement minimum spanning trees algorithm and analyse its time complexity.

Syeda Reeha Quasar

14114802719

4C7

EXPERIMENT – 7

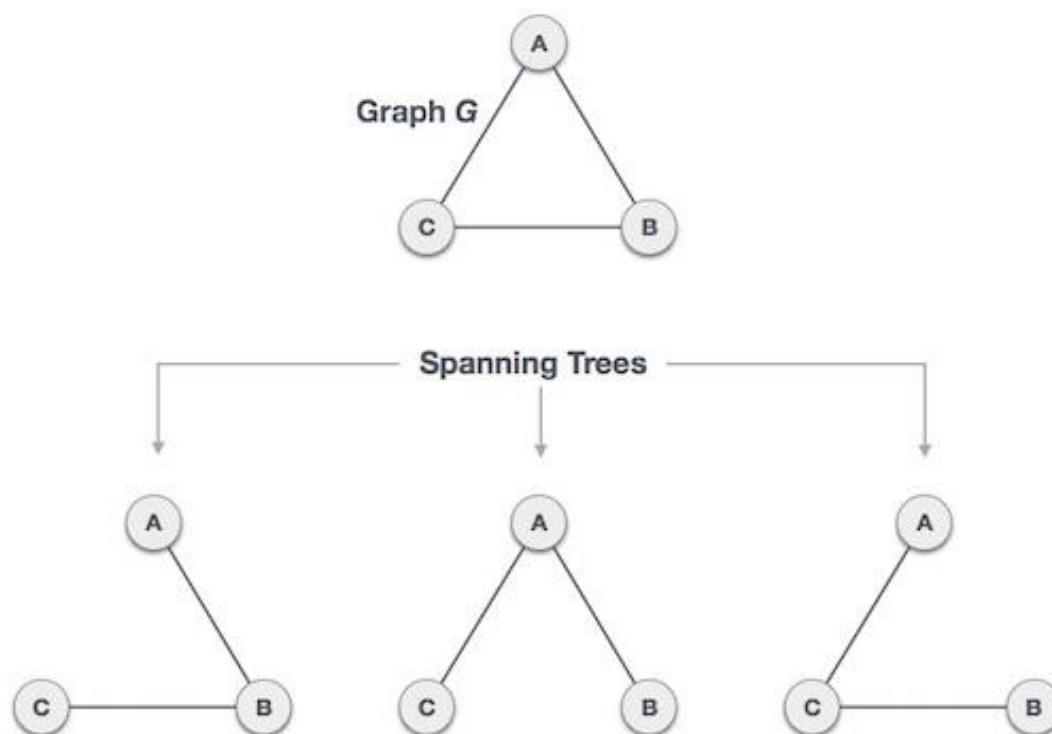
Aim:

To implement minimum spanning trees algorithm and analyse its time complexity.

Theory:

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.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G , have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has **$n-1$** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **$e - n + 1$** edges, we can construct a spanning tree.
- A complete graph can have maximum **n^{n-2}** number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are –

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

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.

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here –

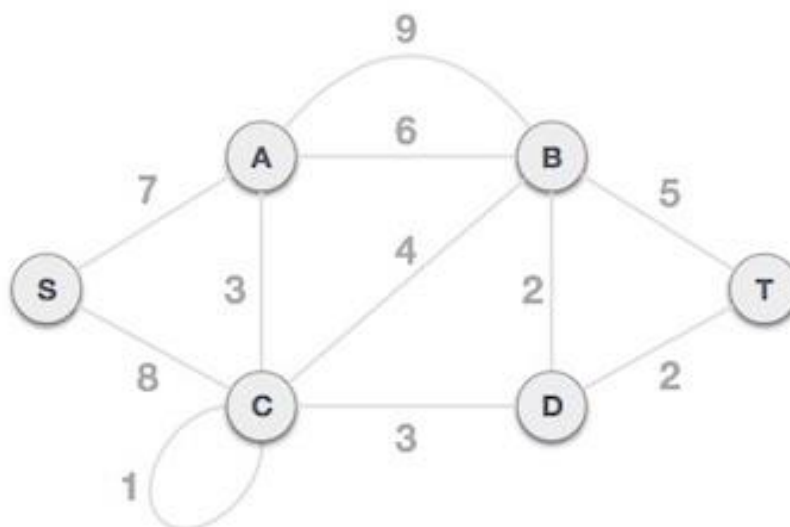
- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

KRUSKAL'S ALGORITHM

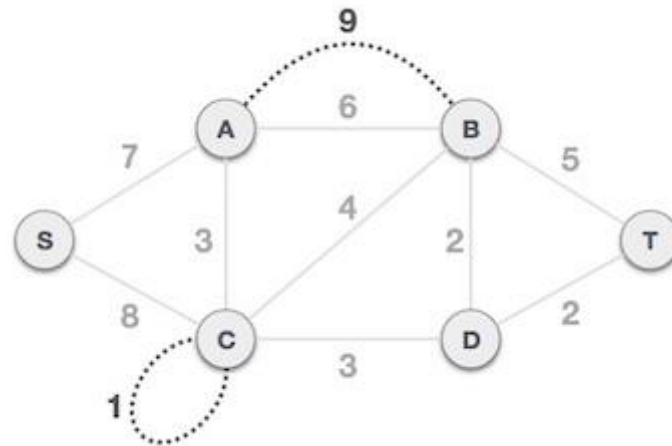
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

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

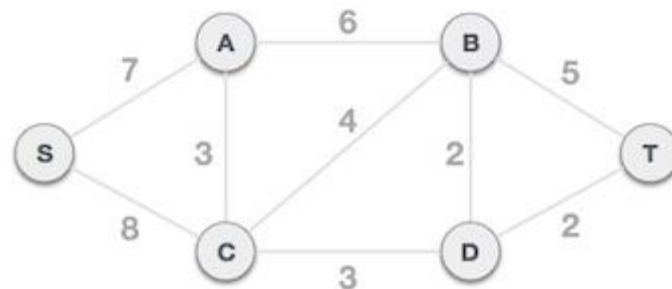


Step 1 - Remove all loops and Parallel Edges

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.



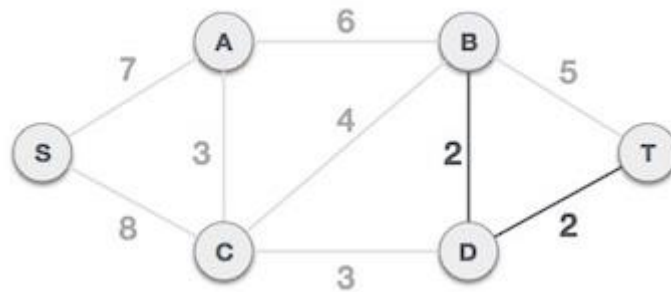
Step 2 - Arrange all edges in their increasing order of weight

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

| B, D | D, T | A, C | C, D | C, B | B, T | A, B | S, A | S, C |
|------|------|------|------|------|------|------|------|------|
| 2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 |

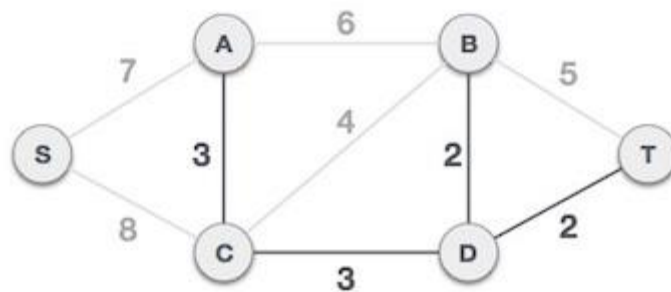
Step 3 - Add the edge which has the least weightage

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.

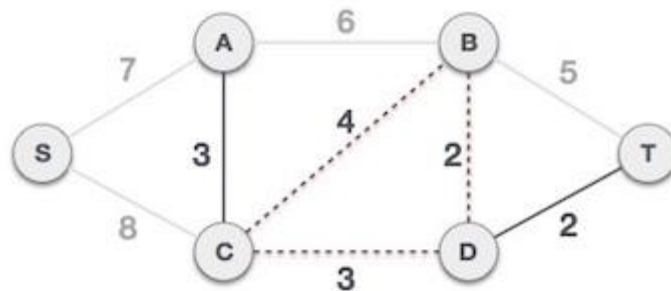


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

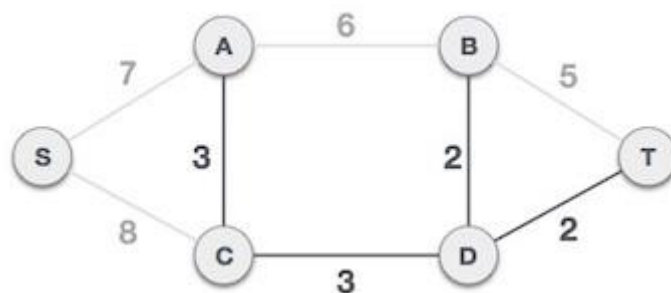
Next cost is 3, and associated edges are A,C and C,D. We add them again –



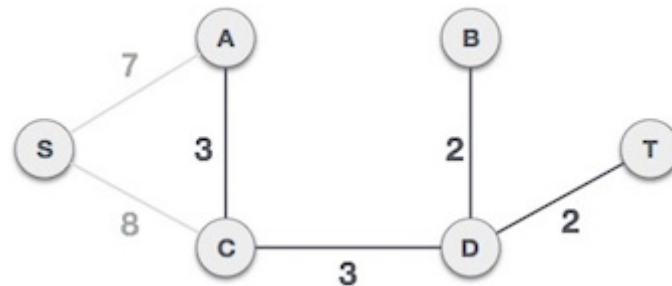
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



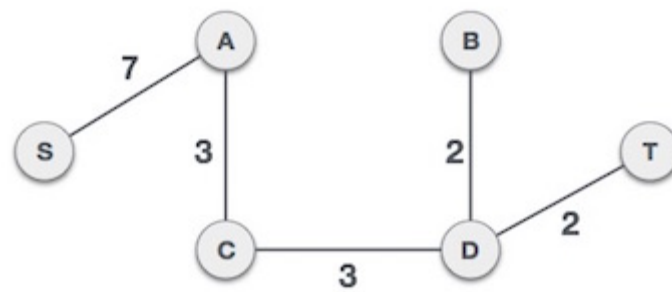
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



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.

Source Code:

```

#include <bits/stdc++.h>
using namespace std;

class DSU
{
    int *parent;
    int *rank;

public:
    DSU(int n)
    {
        parent = new int[n];
        rank = new int[n];
    }

```

```
        for (int i = 0; i < n; i++)
        {
            parent[i] = -1;
            rank[i] = 1;
        }
    }

    int find(int i)
    {
        if (parent[i] == -1)
            return i;

        return parent[i] = find(parent[i]);
    }

    void unite(int x, int y)
    {
        int s1 = find(x);
        int s2 = find(y);

        if (s1 != s2)
        {
            if (rank[s1] < rank[s2])
            {
                parent[s1] = s2;
                rank[s2] += rank[s1];
            }
            else
            {
                parent[s2] = s1;
                rank[s1] += rank[s2];
            }
        }
    }
};

class Graph
{
    vector<vector<int>> edgelist;
    int V;

public:
    Graph(int V)
    {
        this->V = V;
    }
};
```



```
    }

    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({w, x, y});
    }

    int kruskals_mst()
    {
        // 1. Sort all edges
        sort(edgelist.begin(), edgelist.end());

        // Initialize the DSU
        DSU s(V);
        int ans = 0;
        for (auto edge : edgelist)
        {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];

            // take that edge in MST if it does not form a cycle
            if (s.find(x) != s.find(y))
            {
                s.unite(x, y);
                ans += w;
            }
        }
        return ans;
    }
};

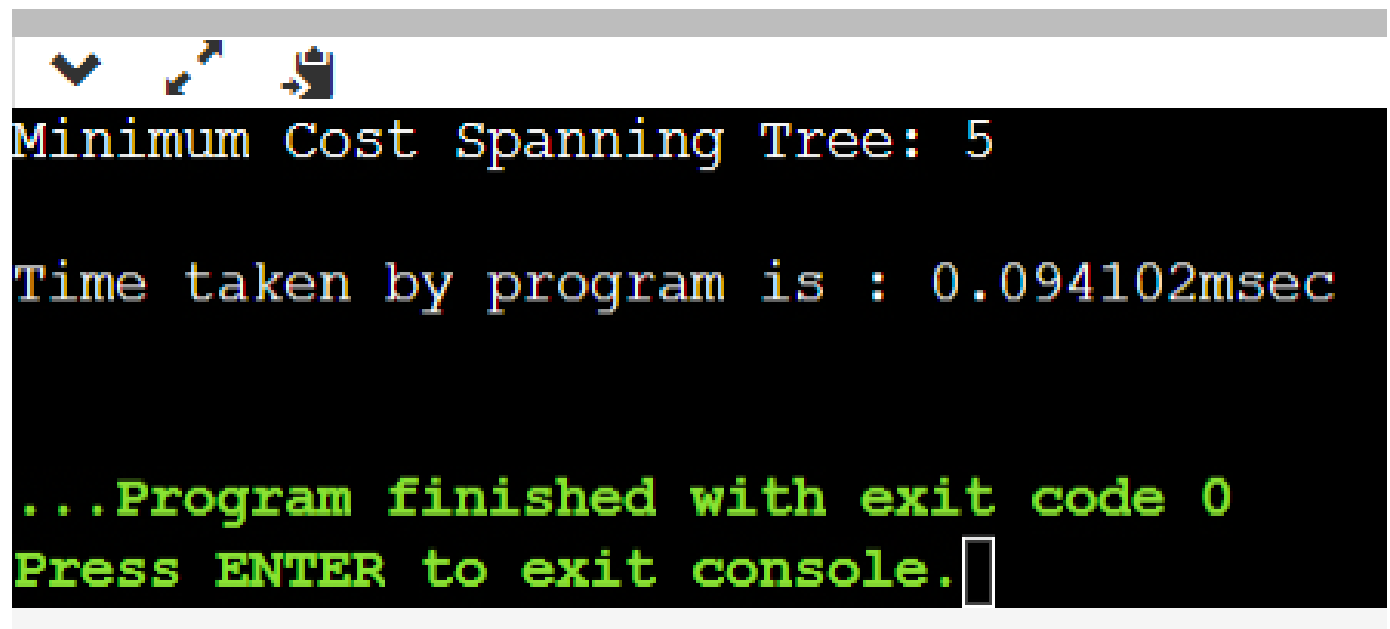
int main()
{
    Graph g(4);
    g.addEdge(0, 1, 1);
    g.addEdge(1, 3, 3);
    g.addEdge(3, 2, 4);
    g.addEdge(2, 0, 2);
    g.addEdge(0, 3, 2);
    g.addEdge(1, 2, 2);

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    cout << "Minimum Cost Spanning Tree: " << g.kruskals_mst() << endl;
```

```
    auto end = chrono::high_resolution_clock::now();  
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -  
start).count();  
  
    time_taken *= 1e-9 * 1000;  
  
    cout << "\nTime taken by program is : " << time_taken << setprecision(6);  
    cout << "msec" << endl;  
  
    return 0;  
}
```

Output:

A terminal window with a black background and white and green text. The output of the program is displayed line by line. The first line is "Minimum Cost Spanning Tree: 5". The second line is "Time taken by program is : 0.094102msec". The third line is "...Program finished with exit code 0". The fourth line is "Press ENTER to exit console." followed by a white cursor box.

```
Minimum Cost Spanning Tree: 5  
  
Time taken by program is : 0.094102msec  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

A terminal window with a black background and white and green text. The output of the program is displayed line by line. The first line is "Minimum Cost Spanning Tree: 5". The second line is "Time taken by program is : 0.094102msec". The third line is "...Program finished with exit code 0". The fourth line is "Press ENTER to exit console." followed by a white cursor box.

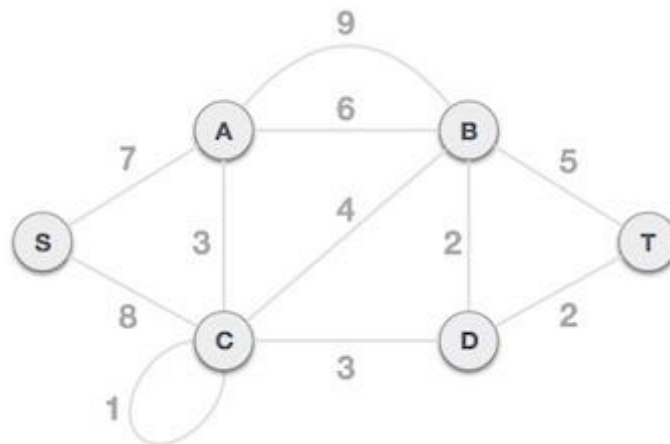
```
Minimum Cost Spanning Tree: 5  
  
Time taken by program is : 0.094102msec  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

PRIM'S SPANNING TREE Algorithm

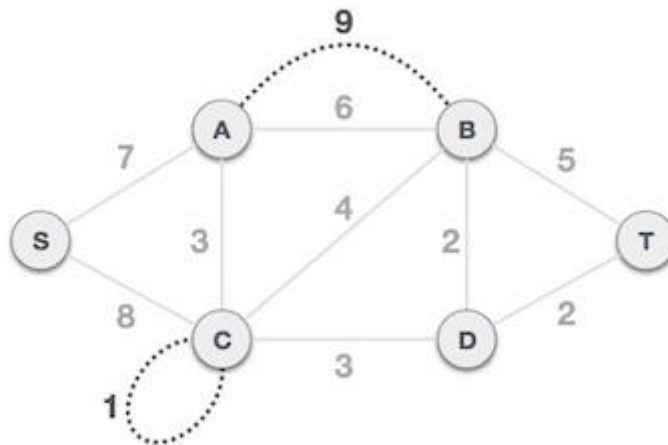
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

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.

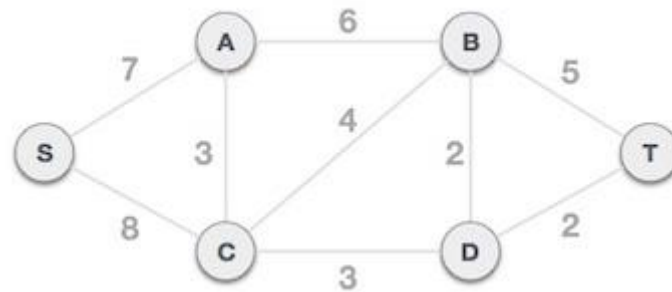
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



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.

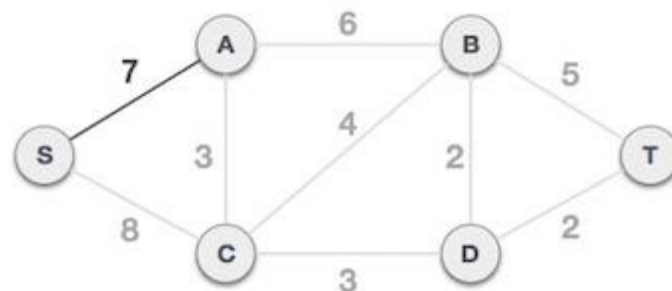


Step 2 - Choose any arbitrary node as root node

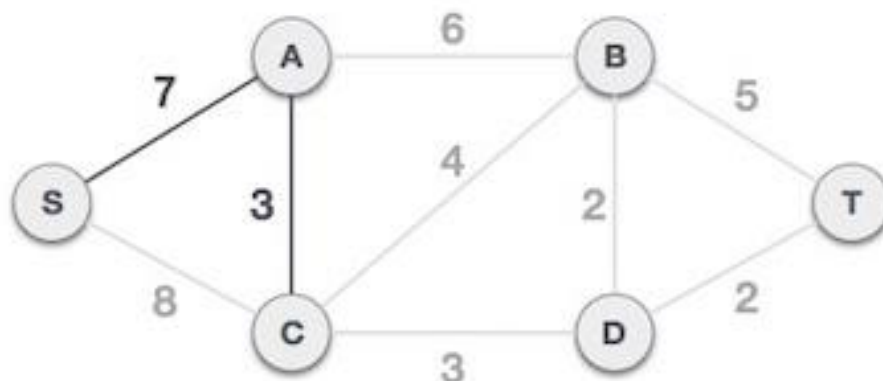
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any node can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then 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

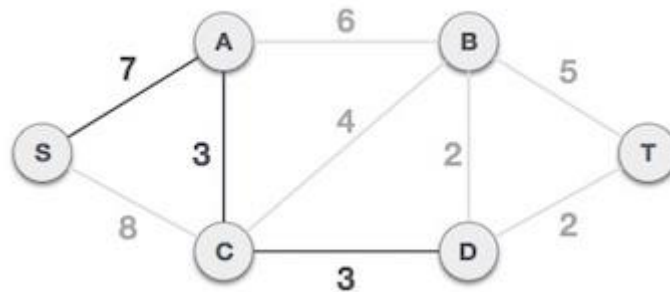
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



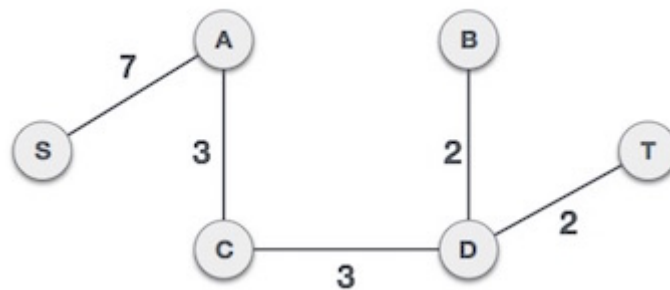
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



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.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with minimum key value, from the set of
vertices not yet included in MST
int minKey(int key[], bool mstSet[])
```

```
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " \t" << graph[i][parent[i]] << " \n";
}

// Function to construct and print MST for a graph represented using adjacency
matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices included in MST
    bool mstSet[V];

    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {
        int u = minKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
```

```

        parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

int main()
{
    /* Let us create the following graph
        2 3
    (0)--(1)--(2)
    | / \ |
    6| 8/  \5 |7
    | / \ |
    (3)-----(4)
        9      */
    int graph[V][V] = {{0, 2, 0, 6, 0},
                      {2, 0, 3, 8, 5},
                      {0, 3, 0, 0, 7},
                      {6, 8, 0, 0, 9},
                      {0, 5, 7, 9, 0}};

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    primMST(graph);

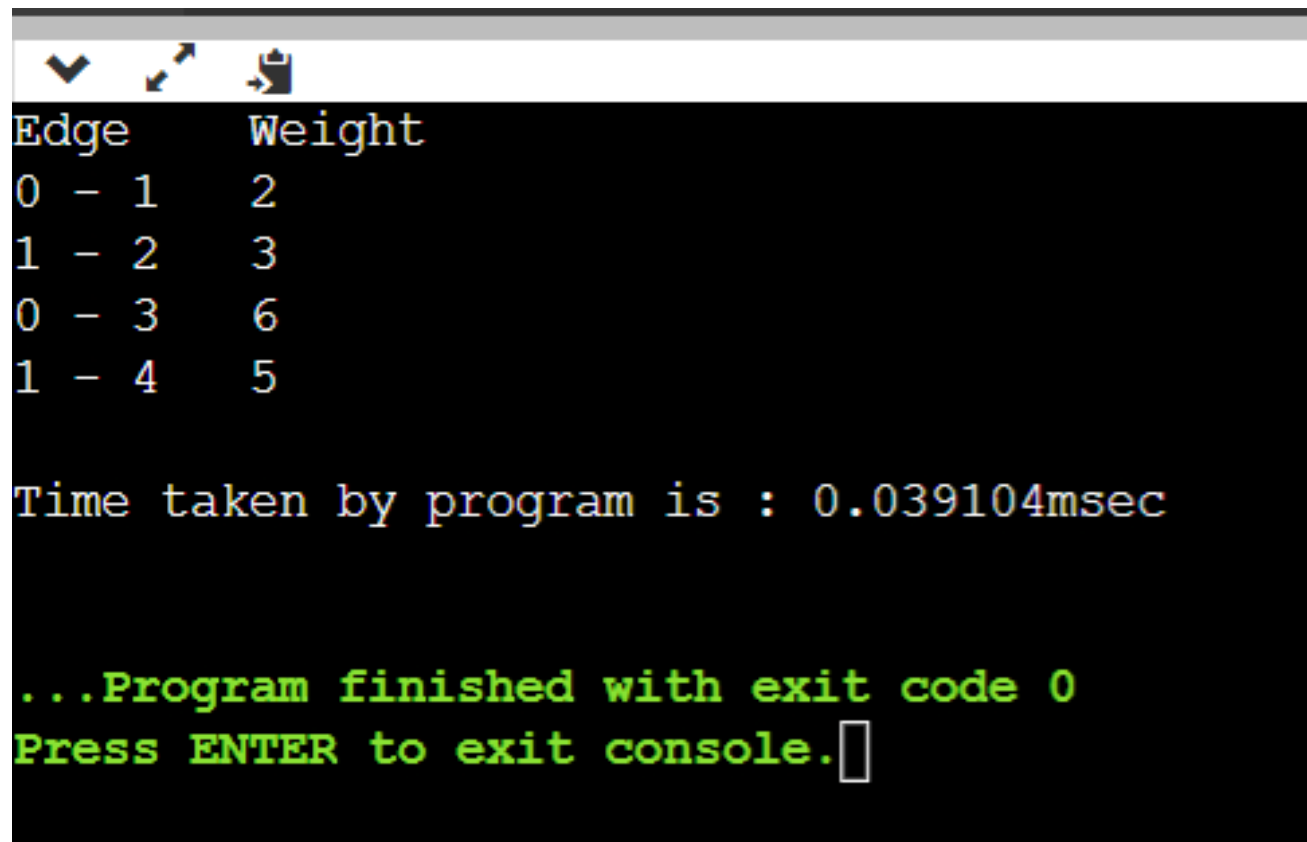
    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}

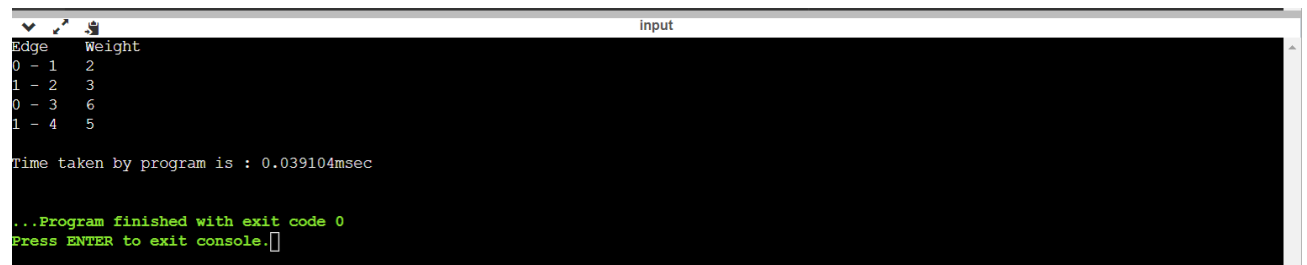
```

Output:

```
Edge      Weight
0 - 1     2
1 - 2     3
0 - 3     6
1 - 4     5

Time taken by program is : 0.039104msec

...Program finished with exit code 0
Press ENTER to exit console.
```



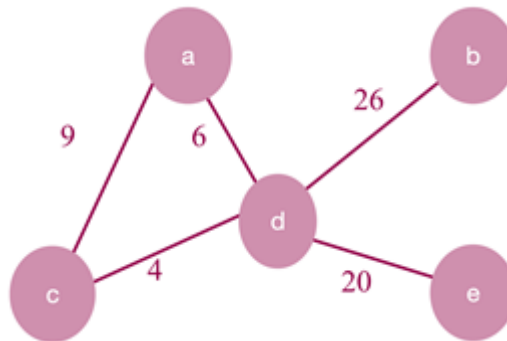
```
Edge      Weight
0 - 1     2
1 - 2     3
0 - 3     6
1 - 4     5

Time taken by program is : 0.039104msec

...Program finished with exit code 0
Press ENTER to exit console.
```


Viva Questions

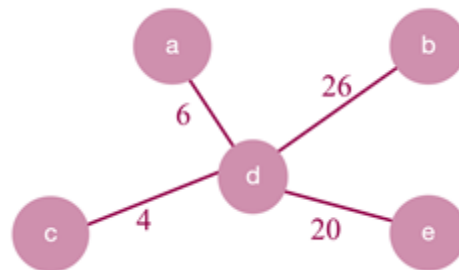
1. Consider the graph shown below. What are the edges in the MST of the given graph?



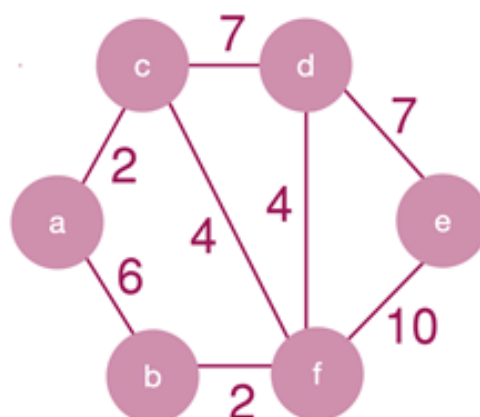
Ans.

(a-d)(d-c)(d-b)(d-e)

The minimum spanning tree of the given graph is shown below. It has cost 56.



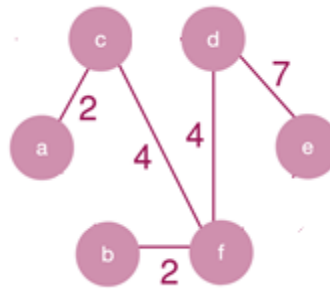
2 Consider the given graph.



What is the weight of the minimum spanning tree using the Kruskal's algorithm?

Ans.

Kruskal's algorithm constructs the minimum spanning tree by constructing by adding the edges to spanning tree one-one by one. The MST for the given graph is,



So, the weight of the MST is 19.

3. What is the time complexity of Kruskal's algorithm?

Ans.

Kruskal's algorithm involves sorting of the edges, which takes $O(E \log E)$ time, where E is a number of edges in graph and V is the number of vertices. After sorting, all edges are iterated and union-find algorithm is applied. union-find algorithm requires $O(\log V)$ time. So, overall Kruskal's algorithm requires $O(E \log V)$ time.

4. Worst case is the worst case time complexity of Prim's algorithm if adjacency matrix is used?

Ans.

Use of adjacency matrix provides the simple implementation of the Prim's algorithm. In Prim's algorithm, we need to search for the edge with a minimum for that vertex. So, worst case time complexity will be $O(V^2)$, where V is the number of vertices.

5. Prim's algorithm is also known as _____

Ans.

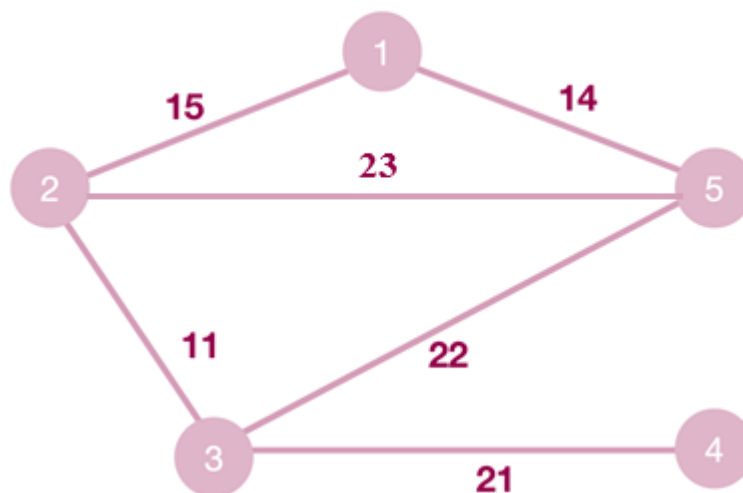
The Prim's algorithm was developed by Vojtěch Jarník and it was later discovered by the duo Prim and Dijkstra. Therefore, Prim's algorithm is also known as DJP Algorithm.

6. Prim's algorithm can be efficiently implemented using _____ for graphs with greater density.

Ans.

In Prim's algorithm, we add the minimum weight edge for the chosen vertex which requires searching on the array of weights. This searching can be efficiently implemented using binary heap for dense graphs. And for graphs with greater density, Prim's algorithm can be made to run in linear time using d-ary heap (generalization of binary heap).

7. Consider the graph shown below.

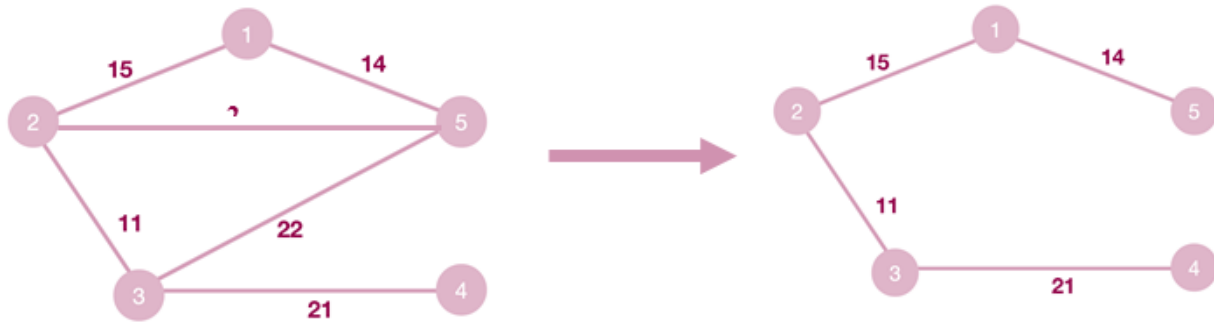


which of the following edges form the MST of the given graph using Prim's algorithm, starting from vertex 4.

Ans.

(4-3)(3-2)(2-1)(1-5)

The MST for the given graph using Prim's algorithm starting from vertex 4 is,



So, the MST contains edges (4-3)(3-2)(2-1)(1-5).

8. Prim's algorithm is a _____

Ans.

Prim's algorithm uses a greedy algorithm approach to find the MST of the connected weighted graph. In greedy method, we attempt to find an optimal solution in stages.