Page No.:
Date:

Experiment 10

Aim : You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges diff amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures.

Theory:

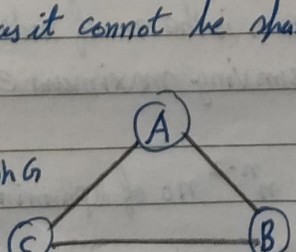
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach.
Prim's algorithm shares a similarity with the ~~st~~ shortest path first algorithms.
Prim's algorithm in contrast with Kruskal's algorithm, treats the nodes as a single tree & keeps on adding new nodes 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
ex:

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible no. of edges. Hence, a spanning tree does not have cycle and it connot be disconnected.

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

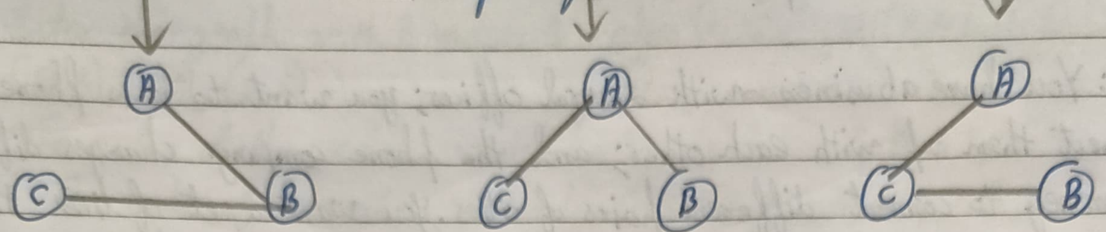
Graph G



Page No.:

Date:

Spanning Trees



We found three spanning trees off one complete graph. A complete undirected graph can have max n^{n-2} number of spanning trees, where n is the no. of nodes.
In the above addressed
ex:- n is 3, hence $3^{3-2} = 3$ spannings 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 no. of edges & 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 no 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 max n^{n-2} no of spanning tree

Thus, we can conclude that spanning trees are a subset of connected graph G &
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 are.

- Civil Network Planning
- Computer network Routing Protocol
- Cluster Analysis.

Let us understand this through a small ex. Consider, city network as a huge graph and now plans to deploy telephone lines is 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)

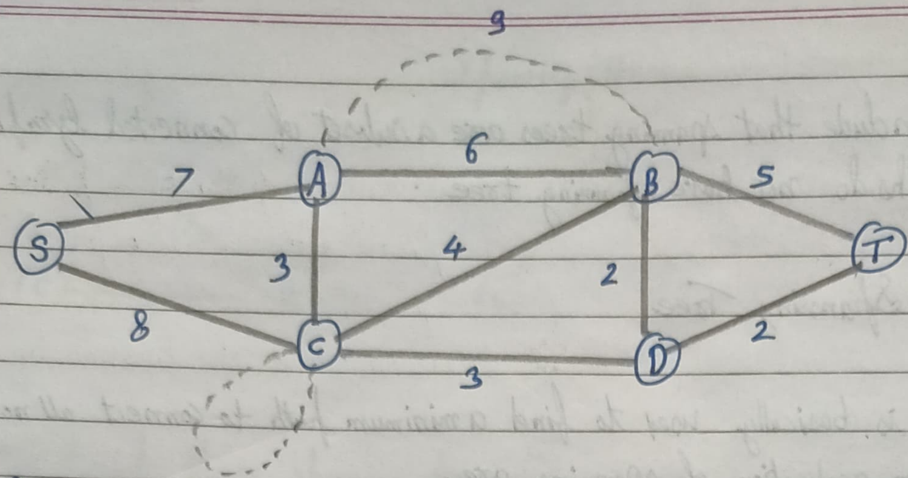
In a weighted graph, a minimum spanning tree that has minimum weight than all other spanning trees of the same graph. In real world situations, this weight can be measured as distance, congestion, traffic load or any ~~arbith~~ arbitrary value denoted to the edges.

Minimum Spanning - Tree Algorithm

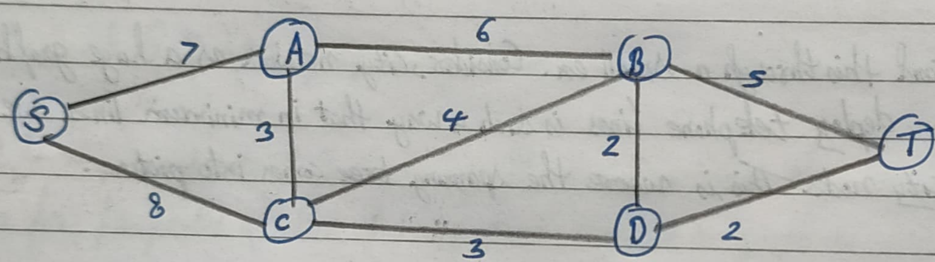
We Shall learn about two most important spanning tree algorithm here.

- Krushal Algorithm
- Prim's Algorithm.

Step-1 Remove all loops & parallel edges.



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

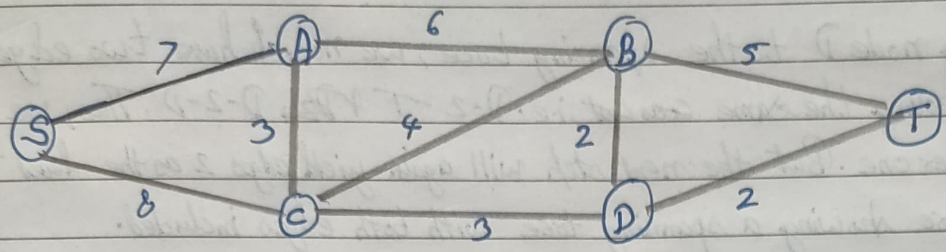


Step 2- Choose any arbitrary nodes as root node.

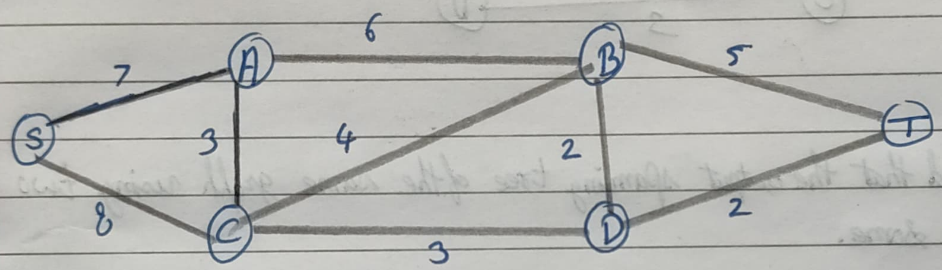
In the case, we choose S node as the root node of Prim's sporning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included & because it is connected then there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

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

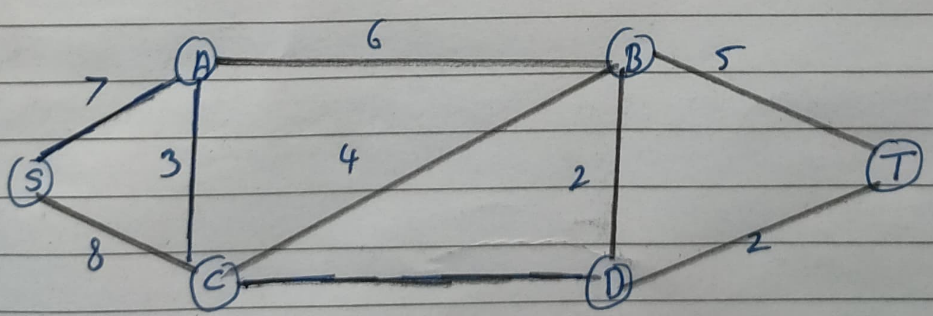
After choosing the root nodes, we see that S, A & S, C are two edges with weight 7 & 8, respectively.
We choose the edge S, A as it is lesser than the other.



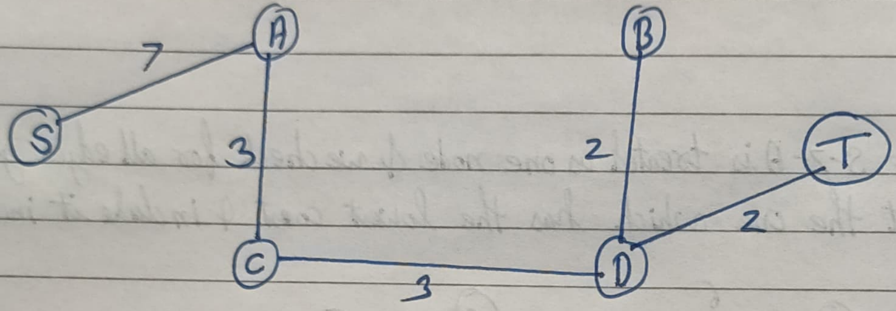
Now, the tree S-7-A is treated as one node & we check for all edges going out from it. We select the one which has the lowest cost & include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node & will check all the edges again. However, we will choose only the least cost edges. 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 ~~coas~~ cost, i.e. D-2-T & ~~D2~~ D-2-D. Thus, we can add either one. But the next step will again yield edges 2 as the least ~~coo~~ 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.

Program –

```
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX = 10;
class EdgeList;
//forward declaration
class Edge //USED IN KRUSKAL
{
    int u, v, w;

public:
    Edge() {}
    //Empty Constructor
    Edge(int a, int b, int weight)
    {
        u = a;
        v = b;
        w = weight;
    }
    friend class EdgeList;
    friend class PhoneGraph;
};
//---- EdgeList Class ----------
class EdgeList
{
    Edge data[MAX];
    int n;

public:
    friend class PhoneGraph;
    EdgeList()
    {
        n = 0;
    }
    void sort();
    void print();
};
//----Bubble Sort for sorting edges in increasing weights' order
void EdgeList::sort()
{
    Edge temp;
    for (int i = 1; i < n; i++)
        for (int j = 0; j < n - 1; j++)
            if (data[j].w > data[j + 1].w)
            {
                temp = data[j];
                data[j] = data[j + 1];
                data[j + 1] = temp;
            }
}
void EdgeList::print()
{
    int cost = 0;
```

```
    for (int i = 0; i < n; i++)
    {
        cout << "\n"
             << i + 1 << " " << data[i].u << "--" << data[i].v << " = " << data[i].w;
        cost = cost + data[i].w;
    }
    cout << "\nMinimum cost of Telephone Graph = " << cost;
}
// Phone Graph Class
class PhoneGraph
{
    int data[MAX][MAX];
    int n;

public:
    PhoneGraph(int num)
    {
        n = num;
    }
    void readgraph();
    void printGraph();
    int mincost(int cost[], bool visited[]);
    int prim();
    void kruskal(EdgeList &spanlist);
    int find(int belongs[], int vertexno);
    void unionComp(int belongs[], int c1, int c2);
};
void PhoneGraph::readgraph()
{
    cout << "Enter Adjacency(Cost) Matrix: \n";
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            cin >> data[i][j];
    }
}
void PhoneGraph::printGraph()
{
    cout << "\nAdjacency (COST) Matrix: \n";
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << setw(3) << data[i][j];
        }
        cout << endl;
    }
}
int PhoneGraph::mincost(int cost[], bool visited[]) //finding vertex with minimum cost
{
    int min = 9999, min_index; //initialize min to MAX value(ANY) as temporary
    for (int i = 0; i < n; i++)
    {
        if (visited[i] == 0 && cost[i] < min)
        {
```

```
            min = cost[i];
            min_index = i;
        }
    }
    return min_index; //return index of vertex which is not visited and having minimum cost
}
int PhoneGraph::prim()
{
    bool visited[MAX];
    int parents[MAX];
    int cost[MAX]; //saving minimum cost
    for (int i = 0; i < n; i++)
    {
        cost[i] = 9999; //set cost as infinity/MAX_VALUE
        visited[i] = 0; //initialize visited array to false
    }
    cost[0] = 0;      //starting vertex cost
    parents[0] = -1; //make first vertex as a root
    for (int i = 0; i < n - 1; i++)
    {
        int k = mincost(cost, visited);
        visited[k] = 1;
        for (int j = 0; j < n; j++)
        {
            if (data[k][j] && visited[j] == 0 && data[k][j] < cost[j])
            {
                parents[j] = k;
                cost[j] = data[k][j];
            }
        }
    }
    cout << "Minimum Cost Telephone Map:\n";
    for (int i = 1; i < n; i++)
    {
        cout << i << " -- " << parents[i] << " = " << cost[i] << endl;
    }
    int mincost = 0;
    for (int i = 1; i < n; i++)
        mincost += cost[i]; //data[i][parents[i]];
    return mincost;
}
// ------- Kruskal's Algorithm
void PhoneGraph::kruskal(EdgeList &spanlist)
{
    int belongs[MAX]; //Separate Components at start (No Edges, Only vertices)
    int cno1, cno2;   //Component 1 & 2
    EdgeList elist;
    for (int i = 1; i < n; i++)
        for (int j = 0; j < i; j++)
        {
            if (data[i][j] != 0)
            {
                elist.data[elist.n] = Edge(i, j, data[i][j]); //constructor for initializing
edge
                elist.n++;
```

```
            }
        }
    elist.sort(); //sorting in increasing weight order
    for (int i = 0; i < n; i++)
        belongs[i] = i;
    for (int i = 0; i < elist.n; i++)
    {
        cno1 = find(belongs, elist.data[i].u); //find set of u
        cno2 = find(belongs, elist.data[i].v); ////find set of v
        if (cno1 != cno2)                      //if u & v belongs to different sets
        {
            spanlist.data[spanlist.n] = elist.data[i]; //ADD Edge to spanlist
            spanlist.n = spanlist.n + 1;
            unionComp(belongs, cno1, cno2); //ADD both components to same set
        }
    }
}
void PhoneGraph::unionComp(int belongs[], int c1, int c2)
{
    for (int i = 0; i < n; i++)
    {
        if (belongs[i] == c2)
            belongs[i] = c1;
    }
}
int PhoneGraph::find(int belongs[], int vertexno)
{
    return belongs[vertexno];
}
// MAIN PROGRAM
int main()
{
    int vertices, choice;
    EdgeList spantree;
    cout << "Enter Number of cities: ";
    cin >> vertices;
    PhoneGraph p1(vertices);
    p1.readgraph();
    do
    {
        cout << "\n1.Find Minimum Total Cost(By Prim's Algorithm)"
             << "\n2.Find Minimum Total Cost(by Kruskal's Algorithms)"
             << "\n3.Re-Read Graph(INPUT)"
             << "\n4.Print Graph"
             << "\n0. Exit"
             << "\nEnter your choice: ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            cout << " Minimum cost of Phone Line to cities is: " << p1.prim();
            break;
        case 2:
            p1.kruskal(spantree);
            spantree.print();
```

```cpp
            break;
        case 3:
            p1.readgraph();
            break;
        case 4:
            p1.printGraph();
            break;
        default:
            cout << "\nWrong Choice!!!";
        }
    } while (choice != 0);
    return 0;
}
```

Output-

```
File  Edit  Selection  View  Go  Run  Terminal  Help          assignment10.cpp - assign 10 - Visual Studio Code
Get Started    assignment10.cpp ×
assignment10.cpp > Edge
#include <iostream>
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
orion@OMEN-15:/mnt/d/College/2 Second year/SY SEM 3/Data Structures and Algorithms (DSA)/Lab manual/assign 10$ ./assignment10
Enter Number of cities: 2
Enter Adjacency(Cost) Matrix:
5
4
3
2

1.Find Minimum Total Cost(By Prim's Algorithm)
2.Find Minimum Total Cost(by Kruskal's Algorithms)
3.Re-Read Graph(INPUT)
4.Print Graph
0. Exit
Enter your choice: 1
 Minimum cost of Phone Line to cities is: Minimum Cost Telephone Map:
1 -- 0 = 4
4
1.Find Minimum Total Cost(By Prim's Algorithm)
2.Find Minimum Total Cost(by Kruskal's Algorithms)
3.Re-Read Graph(INPUT)
4.Print Graph
0. Exit
Enter your choice: 2

1 1--0 = 3
Minimum cost of Telephone Graph = 3
1.Find Minimum Total Cost(By Prim's Algorithm)
2.Find Minimum Total Cost(by Kruskal's Algorithms)
3.Re-Read Graph(INPUT)
4.Print Graph
0. Exit
```

```
Enter your choice: 4

Adjacency (COST) Matrix:
  5  4
  3  2

1.Find Minimum Total Cost(By Prim's Algorithm)
2.Find Minimum Total Cost(by Kruskal's Algorithms)
3.Re-Read Graph(INPUT)
4.Print Graph
0. Exit
Enter your choice: 0

Wrong Choice!!!orion@OMEN-15:/mnt/d/College/2 Second year/SY SEM 3/Data Structures and Algorithms (DSA)/Lab manual/assign 10$
⊗ 0 ⚠ 0   pratikjade   Live Share
```