AOA EXPERIMENT 5

Name: Mohammed Arhaan Ali
Div: S11
Roll No: 02

Aim: To study and implement Minimum Cost Spanning using Prim's and Kruskal's algorithms.

Theory:

Kruskal's Minimum Spanning Tree (MST) Algorithm

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.

Introduction to Kruskal's Algorithm:
Here we will discuss Kruskal's algorithm to find the MST of a given weighted graph.
In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a Greedy Algorithm.

How to find MST using Kruskal's algorithm?

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

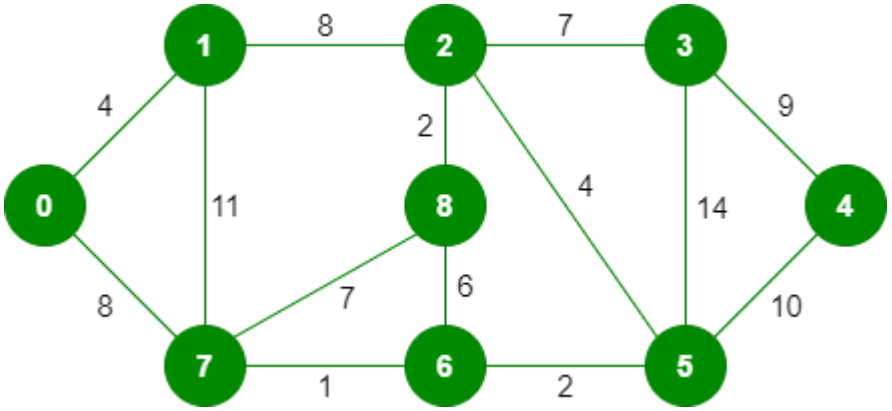
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. The Greedy Choice is to pick the smallest weight edge

that does not cause a cycle in the MST constructed so far. Let us understand it with an example:

Illustration:

Below is the illustration of the above approach:
Input Graph:

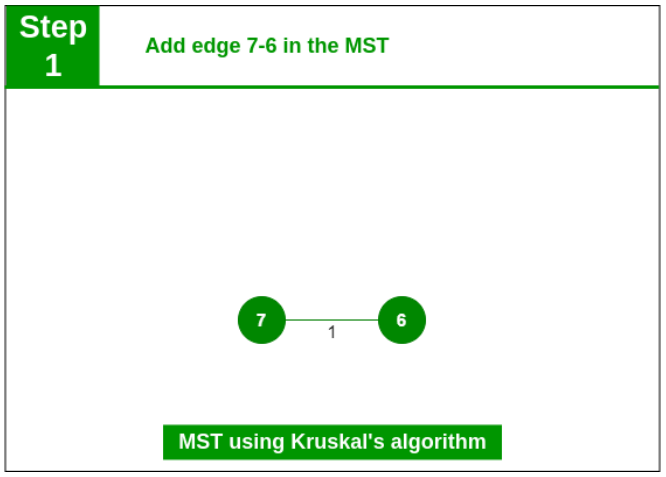


The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.
After sorting:

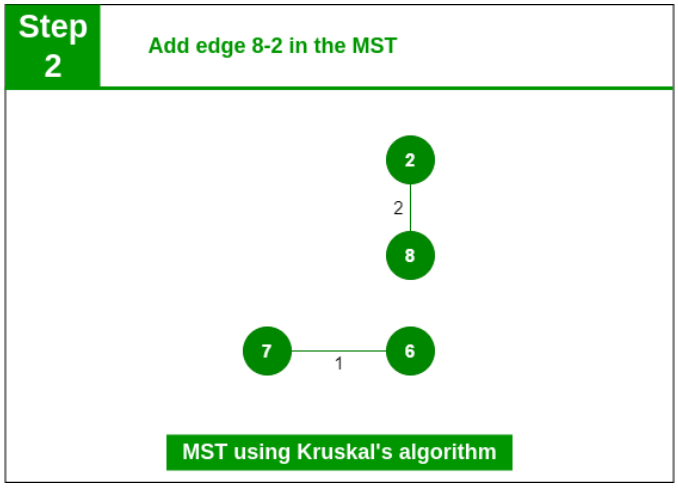
Weight	Source	Destination
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3

7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

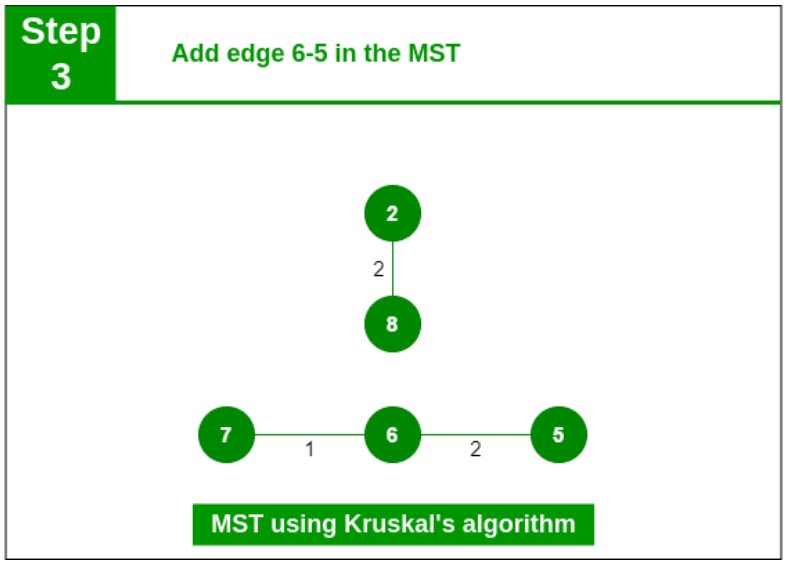
Now pick all edges one by one from the sorted list of edges
Step 1: Pick edge 7-6. No cycle is formed, include it.



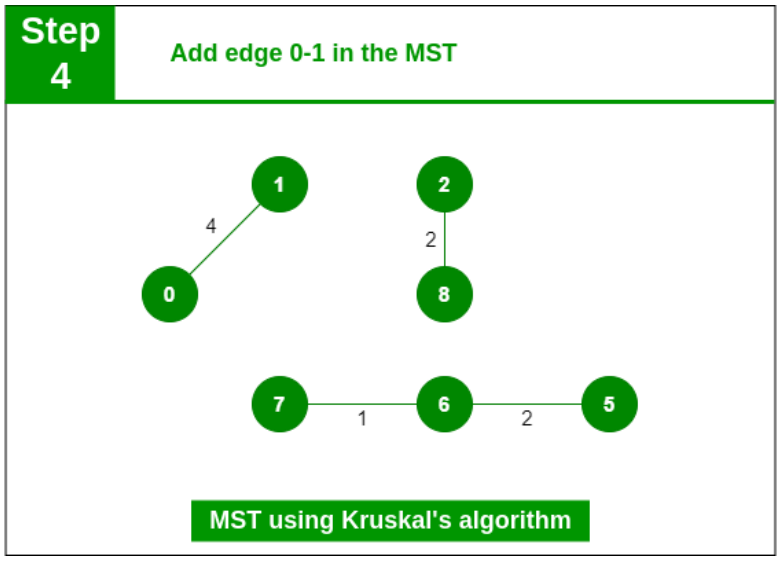
Step 2: Pick edge 8-2. No cycle is formed, include it.



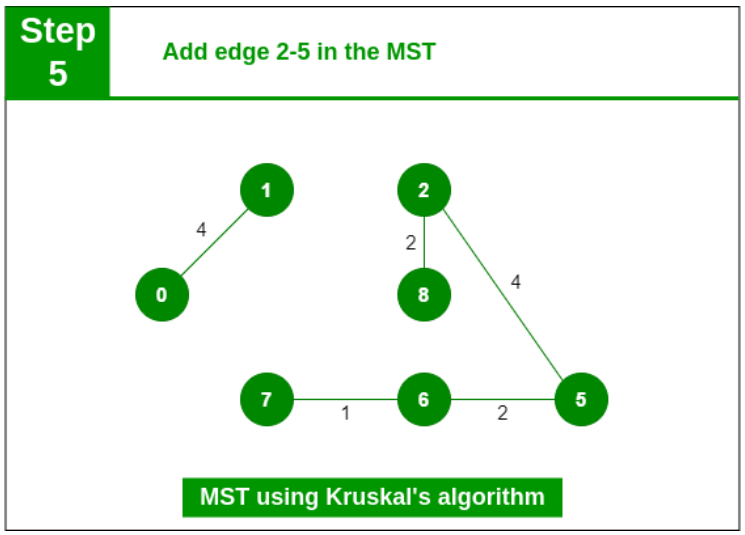
Add edge 8-2 in the MST
Step 3: Pick edge 6-5. No cycle is formed, include it.



Add edge 6-5 in the MST
Step 4: Pick edge 0-1. No cycle is formed, include it.

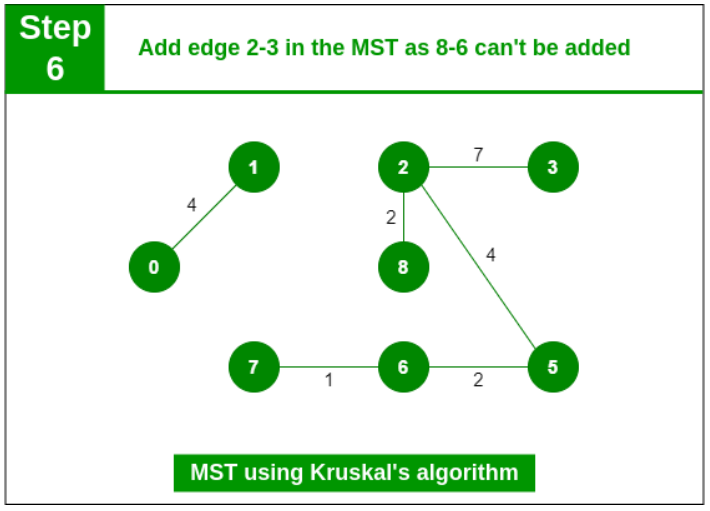


Add edge 0-1 in the MST
Step 5: Pick edge 2-5. No cycle is formed, include it.



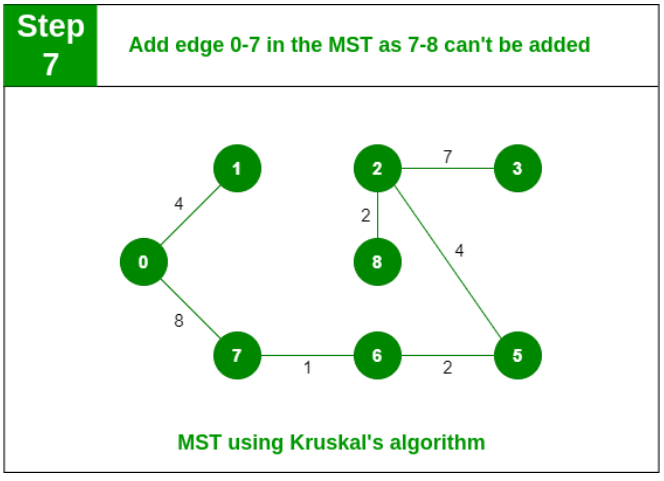
Add edge 2-5 in the MST

Step 6: Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.

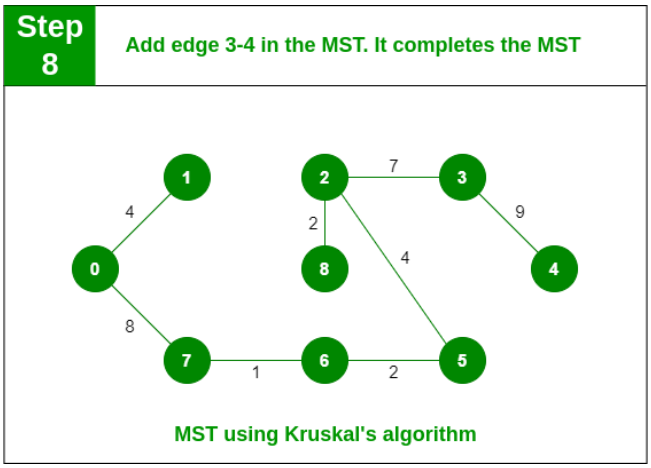


Add edge 2-3 in the MST

Step 7: Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it.



Add edge 0-7 in MST
Step 8: Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is formed, include it.



Add edge 3-4 in the MST
Note: Since the number of edges included in the MST equals to (V – 1), so the algorithm stops here

Program:

```
#include <stdio.h>
#include <stdlib.h>

// Comparator function to use in sorting
int comparator(const void* p1, const void* p2)
{
    const int(*x)[3] = p1;
    const int(*y)[3] = p2;

    return (*x)[2] - (*y)[2];
}

// Initialization of parent[] and rank[] arrays
void makeSet(int parent[], int rank[], int n)
{
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
```

```
}

// Function to find the parent of a node
int findParent(int parent[], int component)
{
    if (parent[component] == component)
        return component;

    return parent[component]
        = findParent(parent, parent[component]);
}

// Function to unite two sets
void unionSet(int u, int v, int parent[], int rank[], int n)
{
    // Finding the parents
    u = findParent(parent, u);
    v = findParent(parent, v);

    if (rank[u] < rank[v]) {
        parent[u] = v;
    }
    else if (rank[u] > rank[v]) {
        parent[v] = u;
    }
    else {
        parent[v] = u;

        // Since the rank increases if
        // the ranks of two sets are same
        rank[u]++;
    }
}

// Function to find the MST
void kruskalAlgo(int n, int edge[n][3])
{
```

```
    // First we sort the edge array in ascending order
    // so that we can access minimum distances/cost
    qsort(edge, n, sizeof(edge[0]), comparator);

    int parent[n];
    int rank[n];

    // Function to initialize parent[] and rank[]
    makeSet(parent, rank, n);

    // To store the minimun cost
    int minCost = 0;

    printf(
        "Following are the edges in the constructed MST\n");
    for (int i = 0; i < n; i++) {
        int v1 = findParent(parent, edge[i][0]);
        int v2 = findParent(parent, edge[i][1]);
        int wt = edge[i][2];

        // If the parents are different that
        // means they are in different sets so
        // union them
        if (v1 != v2) {
            unionSet(v1, v2, parent, rank, n);
            minCost += wt;
            printf("%d -- %d == %d\n", edge[i][0],
                edge[i][1], wt);
        }
    }

    printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

// Driver code
int main()
{
```

```
    int edge[5][3] = { { 0, 1, 10 },
                       { 0, 2, 6 },
                       { 0, 3, 5 },
                       { 1, 3, 15 },
                       { 2, 3, 4 } };

    kruskalAlgo(5, edge);

    return 0;
}
```

Output:

```
PS C:\Users\arhaa\OneDrive\Desktop\AOA> cd "c:\Users\arhaa\OneDriv
 { .\kruskal }
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
PS C:\Users\arhaa\OneDrive\Desktop\AOA>
```

Theory:

Introduction to Prim's algorithm:

We have discussed Kruskal's algorithm for Minimum Spanning Tree. Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way.

The algorithm 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, and 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, find a cut, pick

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

How does Prim's Algorithm Work?

The working of Prim's algorithm can be described by using the following steps:
Step 1: Determine an arbitrary vertex as the starting vertex of the MST.
Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
Step 3: Find edges connecting any tree vertex with the fringe vertices.
Step 4: Find the minimum among these edges.
Step 5: Add the chosen edge to the MST if it does not form any cycle.
Step 6: Return the MST and exit
Note: For determining a cycle, we can divide the vertices into two sets [one set contains the vertices included in MST and the other contains the fringe vertices.]

How to implement Prim's Algorithm?

Follow the given steps to utilise the Prim's Algorithm mentioned above for finding MST of a graph:

- Create a set mstSet that keeps track of vertices already included in MST.
- Assign a key value to all vertices in the input graph. Initialise all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.
- While mstSet doesn't include all vertices
    - Pick a vertex u that is not there in mstSet and has a minimum key value.
    - Include u in the mstSet.
    - Update the key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices.
        - For every adjacent vertex v, if the weight of edge u-v is less than the previous key value of v, update the key value as the 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 that are not yet

included in MST, the key value for these vertices indicates the minimum weight edges connecting them to the set of vertices included in MST. Below is the implementation of the approach:

Program:

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

// 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[]
int printMST(int parent[], int graph[V][V])
{
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i,
            graph[i][parent[i]]);
```

```
}

// 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];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first
    // vertex.
    key[0] = 0;

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

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {

        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
```

```
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent
            // vertices of m mstSet[v] is false for vertices
            // not yet included in MST Update the key only
            // if graph[u][v] is smaller than key[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);
}

// Driver's code
int main()
{
    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 } };

    // Print the solution
    primMST(graph);

    return 0;
}
```

Output:

```
PS C:\Users\arhaa\OneDrive\Desktop\AOA> cd "c:\Users\arhaa\OneDr
rim }
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
PS C:\Users\arhaa\OneDrive\Desktop\AOA>
```

Conclusion: Thus we have successfully implemented Prim's and Kruskal's algorithms.