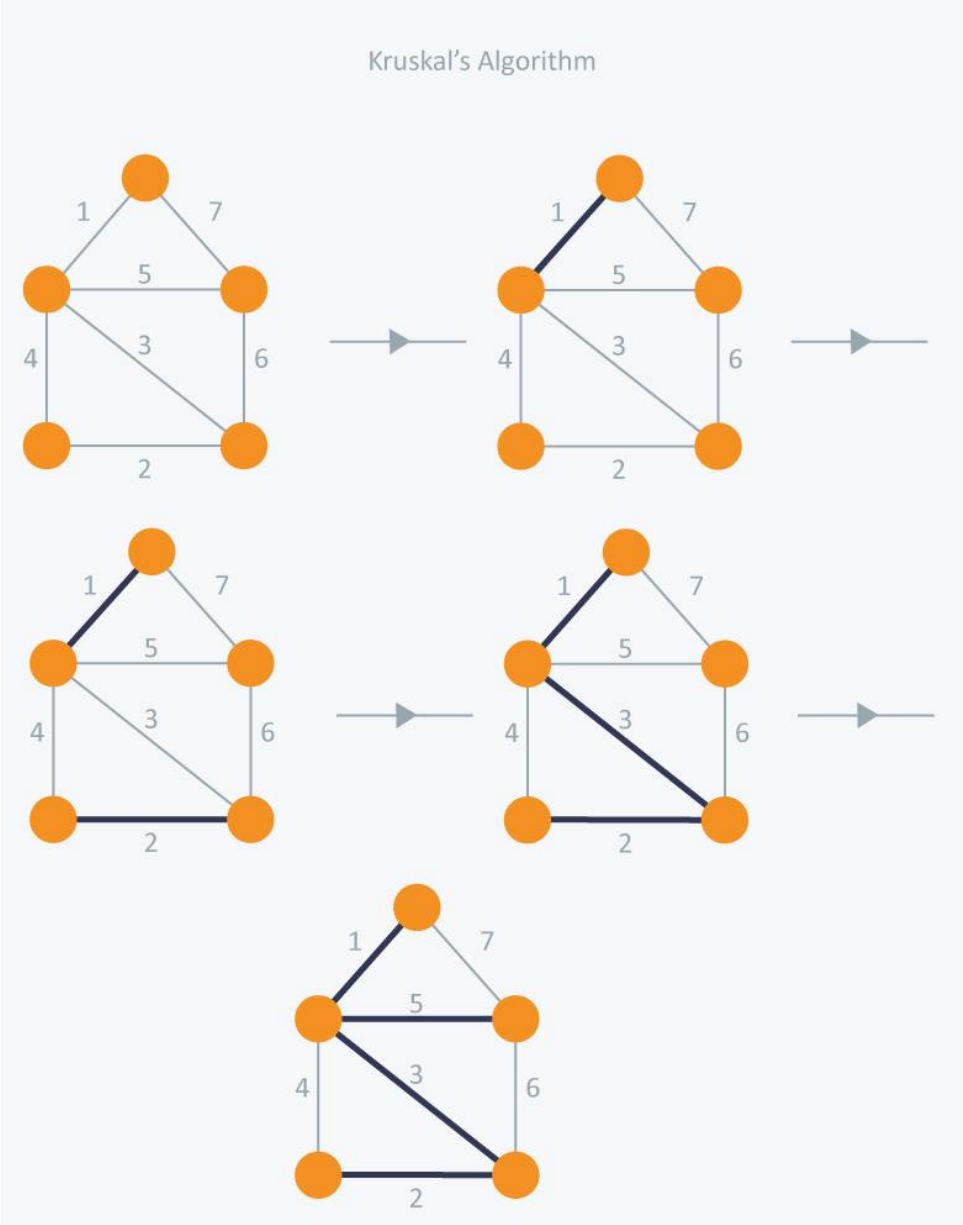
AA LAB ASSIGNMENT | KRUSKAL'S ALGORITHM

TITLE: Analysis, Proof of Analysis and Implementation of **Kruskal's MST Algorithm**

MECHANISM

Kruskal's algorithm is based on the idea of a forest. A forest is a collection of disjoint trees. Initially, the algorithm creates a forest with each vertex as a separate tree. Then, it repeatedly selects the minimum weight edge that connects two disjoint trees and adds it to the MST. The algorithm stops when all the vertices are connected to the MST.



ALGORITHM / PSEUDOCODE

Input: A connected, weighted graph G with vertices V and edges E.
Output: The minimum spanning tree of G.

1. Sort the edges of G in non-decreasing order of weight.
2. Initialize an empty set S to represent the MST.
3. For each vertex v ∈ V, create a new disjoint set containing only v.
4. For each edge (u, v) ∈ E, in non-decreasing order of weight: a. If u and v are not in the same set, add the edge (u, v) to S and merge the sets containing u and v.
5. Return S as the MST.

IMPLEMENTATION

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

// Structure to represent an edge in the graph
struct Edge {
    int u, v, w;
};

// Structure to represent a disjoint set
struct DisjointSet {
    int parent;
    int rank;
};

// Function to find the parent of a vertex in the disjoint set
int findParent(vector<DisjointSet>& ds, int i) {
    if (ds[i].parent != i) {
        ds[i].parent = findParent(ds, ds[i].parent);
    }
    return ds[i].parent;
}

// Function to merge two disjoint sets based on their rank
void merge(vector<DisjointSet>& ds, int x, int y) {
    int xroot = findParent(ds, x);
    int yroot = findParent(ds, y);

    if (ds[xroot].rank < ds[yroot].rank) {
        ds[xroot].parent = yroot;
    } else if (ds[xroot].rank > ds[yroot].rank) {
        ds[yroot].parent = xroot;
```

```
    } else {
        ds[yroot].parent = xroot;
        ds[xroot].rank++;
    }
}

// Function to find the minimum spanning tree of a graph
vector<Edge> kruskalMST(vector<Edge>& edges, int n) {
    vector<Edge> mst;
    vector<DisjointSet> ds(n);

    // Initialize disjoint sets
    for (int i = 0; i < n; i++) {
        ds[i].parent = i;
        ds[i].rank = 0;
    }

    // Sort edges by weight
    sort(edges.begin(), edges.end(), [](Edge& a, Edge& b) {
        return a.w < b.w;
    });

    // Add edges to MST until all vertices are connected
    int i = 0;
    while (mst.size() < n - 1) {
        Edge e = edges[i++];

        int x = findParent(ds, e.u);
        int y = findParent(ds, e.v);

        if (x != y) {
            mst.push_back(e);
            merge(ds, x, y);
        }
    }

    return mst;
}

// Driver code
int main() {
    int n, m;
    cin >> n >> m;

    vector<Edge> edges(m);
    for (int i = 0; i < m; i++) {
        cin >> edges[i].u >> edges[i].v >> edges[i].w;
    }
```

```
    vector<Edge> mst = kruskalMST(edges, n);

    cout << "\n\nMST:\n";
    for (int i = 0; i < mst.size(); i++) {
        cout << mst[i].u << " " << mst[i].v << " " << mst[i].w << endl;
    }

    return 0;
}
```

```
PS F:\#3 SIT\6. SEM 6\Advance Algo\Lab> cd "f:\#3 SIT\6. SEM 6\Adv
o kruskal } ; if ($?) { .\kruskal }
4 5
0 1 1
0 2 3
0 3 4

1 3 2
2 3 5


MST:
0 1 1
1 3 2
0 2 3
PS F:\#3 SIT\6. SEM 6\Advance Algo\Lab>
```

T(n) ANALYSIS WITH PROOF

The Union-Find operations take O(log V) time in the worst case, and since we perform these operations E times in the worst case, the total time taken by Union-Find operations is O(E log V).
Therefore, the total time complexity of Kruskal's algorithm is the sum of the time taken by the sorting step and the time taken by the Union-Find operations, which is:
T(n) = O(E log E) + O(E log V) = O(E log E)

SPACE COMPLEXITY ANALYSIS

To store the graph, we need to use an adjacency list or matrix, which takes O(V + E) space.
For the Union-Find operations, we need to maintain a parent array and a rank array, each of size V, which takes O(V) space.
Therefore, the total space complexity of Kruskal's algorithm is:
S(n) = O(V + E) + O(V) = O(V + E)

ADVANTAGES / DISADVANTAGES

Criteria	Advantages	Disadvantages
Algorithmic	- Can handle disconnected graphs and graphs with negative weights	- Slower than Prim's algorithm for dense graphs
	- Easy to implement	- Requires sorting edges, which can be time-consuming for large graphs
Performance	- Guaranteed to find the minimum spanning tree for a connected graph	- Requires additional space to store edges and the Union-Find data structure
	- Performs well on sparse graphs	- Worst-case time and space complexity can be high for dense graphs
	- Parallelizable and amenable to distributed computing	- May not be the most efficient algorithm for certain types of graphs or data

REAL LIFE APPLICATIONS:

Network design: Kruskal's algorithm can be used to design computer networks, such as connecting routers or nodes in a network.

Transportation planning: Kruskal's algorithm can be used to plan road and railway networks by determining the most efficient routes between different locations.

Image segmentation and clustering: Kruskal's algorithm can be used in image processing to segment and cluster images based on similarities between their pixels.

Supply chain optimization: Kruskal's algorithm can be used in logistics and transportation to optimize the supply chain by identifying the most efficient routes for delivering goods.

OPTIMIZATIONS AND ADVANCEMENTS:

1. Use a more efficient sorting algorithm, such as radix sort or counting sort.
2. Use a Union-Find data structure with path compression and ranking.
3. Implement the algorithm in a distributed or parallel computing environment.
4. Use heuristics or other optimization techniques, such as Boruvka's algorithm or the greedy algorithm, to improve the quality of the minimum spanning tree produced by the algorithm.