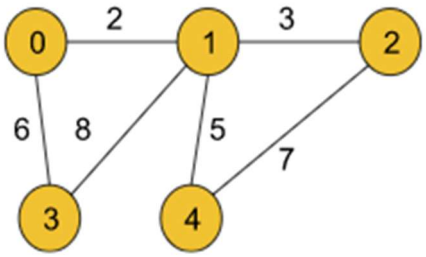
Kruskal's Algorithm – Minimum Spanning Tree : G-47

Problem Statement: Given a weighted, undirected, and connected graph of V vertices and E edges. The task is to find the sum of weights of the edges of the Minimum Spanning Tree.

Example 1:

Input Format:

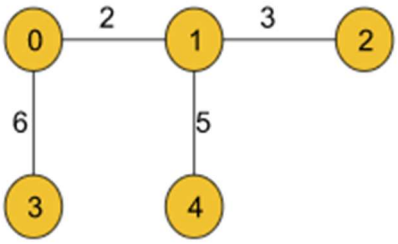
```
V = 5, edges = { {0, 1, 2}, {0, 3, 6}, {1, 2, 3}, {1, 3, 8}, {1, 4, 5}, {4, 2, 7}}
```



Result: 16

Explanation: The minimum spanning tree for the given graph is drawn below:

MST = {(0, 1), (0, 3), (1, 2), (1, 4)}



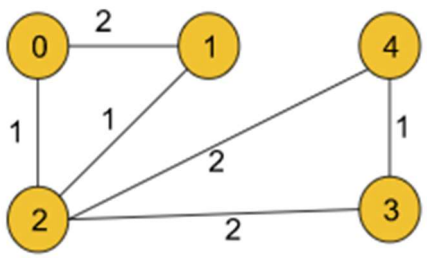
Sum of edge weights = 16

Example 2:

Input Format:

```
V = 5,
```

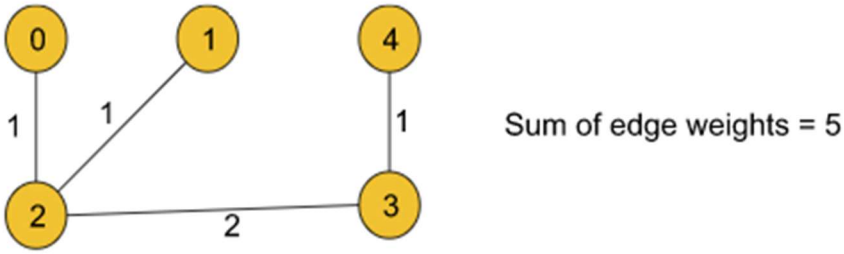
```
edges = { {0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}}
```



Result: 5

Explanation: The minimum spanning tree is drawn below:

MST = {(0, 2), (1, 2), (2, 3), (3, 4)}



Disclaimer. *Don't jump directly to the solution, try it out yourself first.*

Problem link.

Solution:

In the previous article on the minimum spanning tree, we had already discussed that there are two ways to find the minimum spanning tree for a given weighted and undirected graph. Among those two algorithms, we have already discussed Prim's algorithm.

In this article, we will be discussing another algorithm, named ***Kruskal's algorithm***, that is also useful in finding the minimum spanning tree.

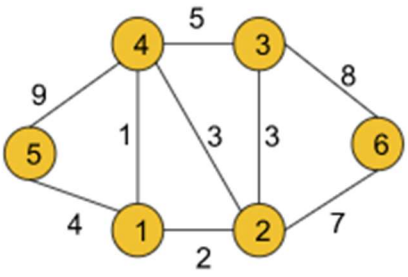
Approach:

We will be implementing Kruskal's algorithm using the ***Disjoint Set data structure*** that we have previously learned.

Now, we know Disjoint Set provides two methods named **findUPar()**(*This function helps to find the ultimate parent of a particular node*) and **Union**(*This basically helps to add the edges between two nodes*). To know more about these functionalities, do refer to the article on Disjoint Set.

The algorithm steps are as follows:

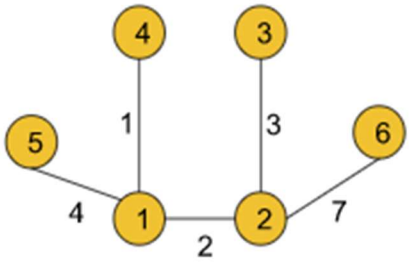
1. First, we need to extract the edge information(*if not given already*) from the given adjacency list in the format of (wt, u, v) where u is the current node, v is the adjacent node and wt is the weight of the edge between node u and v and we will store the tuples in an array.
2. Then the array must be sorted in the ascending order of the weights so that while iterating we can get the edges with the minimum weights first.
3. After that, we will iterate over the edge information, and for each tuple, we will apply the following operation:
    1. First, we will take the two nodes u and v from the tuple and check if the ultimate parents of both nodes are the same or not using the **findUPar()** function provided by the Disjoint Set data structure.
    2. ***If the ultimate parents are the same***, we need not do anything to that edge as there already exists a path between the nodes and we will continue to the next tuple.
    3. If the ultimate parents are different, we will add the weight of the edge to our final answer**(i.e. mstWt variable used in the following code)** and apply the **union operation(i.e. either *unionBySize(u, v) or unionByRank(u, v)*)** with the nodes u and v. The union operation is also provided by the Disjoint Set.
4. Finally, we will get our answer (in the mstWt variable as used in the following code) successfully.



Sorted edges according to weights:

(wt	u	v)
1	1	4
2	1	2
3	2	3
3	2	4
4	1	5
5	3	4
7	2	6
8	3	6
9	4	5

The minimum spanning tree with sum of edge weights = 17



Note: *Points to remember if the graph is given as an adjacency list we must extract the edge information first. As the graph contains bidirectional edges we can get a single edge twice in our array (For example, (wt, u, v) and (wt, v, u), (5, 1, 2) and (5, 2, 1)). But we should not worry about that as the Disjoint Set data structure will automatically discard the duplicate one.*

Note: *This algorithm mainly contains the Disjoint Set data structure used to find the minimum spanning tree of a given graph. So, we just need to know the data structure.*

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code:

- C++ Code
- Java Code

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

```cpp
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }
```

```
    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
class Solution
{
public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        // 1 - 2 wt = 5
        /// 1 - > (2, 5)
        // 2 -> (1, 5)

        // 5, 1, 2
        // 5, 2, 1
        vector<pair<int, pair<int, int>>> edges;
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                int adjNode = it[0];
                int wt = it[1];
                int node = i;

                edges.push_back({wt, {node, adjNode}});
            }
        }
        DisjointSet ds(V);
        sort(edges.begin(), edges.end());
        int mstWt = 0;
        for (auto it : edges) {
```

```
            int wt = it.first;
            int u = it.second.first;
            int v = it.second.second;

            if (ds.findUPar(u) != ds.findUPar(v)) {
                mstWt += wt;
                ds.unionBySize(u, v);
            }
        }

        return mstWt;
    }
};

int main() {

    int V = 5;
    vector<vector<int>> edges = {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2},
{3, 4, 1}, {4, 2, 2}};
    vector<vector<int>> adj[V];
    for (auto it : edges) {
        vector<int> tmp(2);
        tmp[0] = it[1];
        tmp[1] = it[2];
        adj[it[0]].push_back(tmp);

        tmp[0] = it[0];
        tmp[1] = it[2];
        adj[it[1]].push_back(tmp);
    }

    Solution obj;
    int mstWt = obj.spanningTree(V, adj);
    cout << "The sum of all the edge weights: " << mstWt << endl;
    return 0;
}
```

Output: The sum of all the edge weights: 5

Time Complexity: O(N+E) + O(E logE) + O(E*4α*2) where N = no. of nodes and E = no. of edges. O(N+E) for extracting edge information from the adjacency list. O(E logE) for sorting the array consists of the edge tuples. Finally, we are using the disjoint set operations inside a loop. The loop will continue to E times. Inside that loop, there are two disjoint set operations like findUPar() and UnionBySize() each taking 4 and so it will result in 4*2. That is why the last term O(E*4*2) is added.

Space Complexity: O(N) + O(N) + O(E) where E = no. of edges and N = no. of nodes. O(E) space is taken by the array that we are using to store the edge information. And in the disjoint set data structure, we are using two N-sized arrays i.e. a parent and a size array (as we are using unionBySize() function otherwise, a rank array of the same size if unionByRank() is used) which result in the first two terms O(N).