Kruskal's Algorithm

This algorithm was described by Joseph Bernard Kruskal, Jr. in 1956. It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.

Invitation to Algorithmic Graph Theory

Description

Kruskal's algorithm initially places all the nodes of the original graph isolated from each other, to form a forest of single node trees, and then gradually merges these trees, combining at each iteration any two of all the trees with some edge of the original graph. Before the execution of the algorithm, all edges are sorted by weight (in non-decreasing order). Then begins the process of unification: pick all edges from the first to the last (in sorted order), and if the ends of the currently picked edge belong to different subtrees, these subtrees are combined, and the edge is added to the answer. After iterating through all the edges, all the vertices will belong to the same sub-tree, and we will get the answer.

The simplest implementation

The following code directly implements the algorithm described above, and is having $O(M \log M + N^2)$ time complexity. Sorting edges requires $O(M \log N)$ (which is the same as $O(M \log M)$) operations. Information regarding the subtree to which a vertex belongs is maintained with the help of an array $tree_{id}$[] - for each vertex v, $tree_{id}$[v] stores the number of the tree , to which v belongs. For each edge, whether it belongs to the ends of different trees, can be determined in $O(1)$. Finally, the union of the two trees is carried out in $O(N)$ by a simple pass through $tree_{id}$[] array. Given that the total number of merge operations is $N - 1$, we obtain the asymptotic behavior of $O(M \log N + N^2)$.

```cpp
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<int> tree_id(n);
vector<Edge> result;
for (int i = 0; i < n; i++)
    tree_id[i] = i;

sort(edges.begin(), edges.end());

for (Edge e : edges) {
    if (tree_id[e.u] != tree_id[e.v]) {
        cost += e.weight;
        result.push_back(e);

        int old_id = tree_id[e.u], new_id = tree_id[e.v];
        for (int i = 0; i < n; i++) {
            if (tree_id[i] == old_id)
                tree_id[i] = new_id;
        }
    }
}
```

Can we do better?

Yes, of course

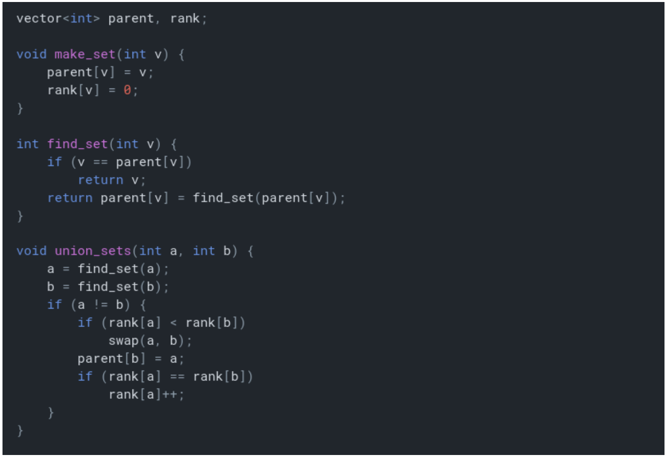
We can use the Disjoint Set Union (DSU) data structure to write a faster implementation of the Kruskal's algorithm with the time complexity of about $O(M \log N)$.

Description

Just as in the simple version of the Kruskal algorithm, we sort all the edges of the graph in non-decreasing order of weights. Then put each vertex in its own tree (i.e. its set) via calls to the $make_{set}$ function - it will take a total of $O(N)$. We iterate through all the edges (in sorted order) and for each edge determine whether the ends belong to different trees (with two $find_{set}$ calls in $O(1)$ each). Finally, we need to perform the union of the two trees (sets), for which the DSU $union_{sets}$ function will be called - also in $O(1)$. So we get the total time complexity of $O(M \log N + N + M) = O(M \log N)$.

Implementation

Here is an implementation of Kruskal's algorithm with Union by Rank.

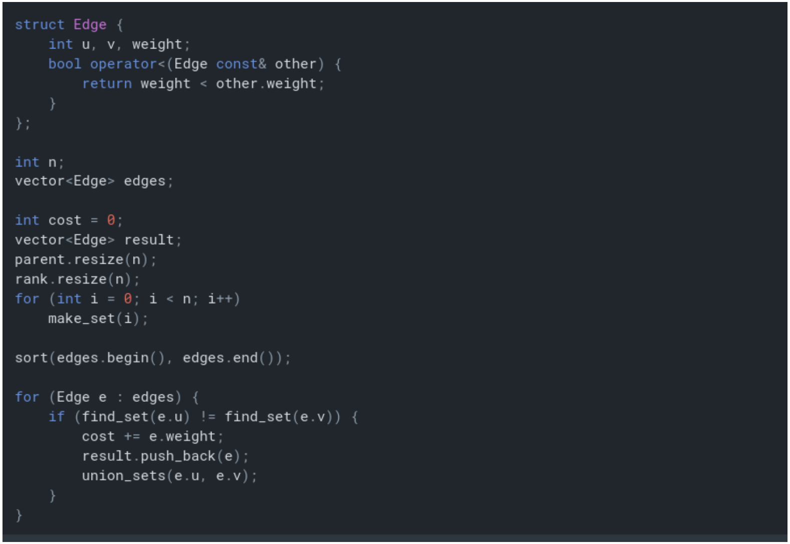


```cpp
vector<int> parent, rank;

void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```



```cpp
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<Edge> result;
parent.resize(n);
rank.resize(n);
for (int i = 0; i < n; i++)
    make_set(i);

sort(edges.begin(), edges.end());

for (Edge e : edges) {
    if (find_set(e.u) != find_set(e.v)) {
        cost += e.weight;
        result.push_back(e);
        union_sets(e.u, e.v);
    }
}
```

Note

Since the MST will contain exactly $N-1$ edges, we can stop the for loop once we found that many.