Minimum Spanning Tree (MST)

Given an undirected, weighted graph, find a subtree of this graph which connects all vertices together, without any cycles and has the least weight (i.e, total sum of weights of the all the edges is minimum) of all possible spanning trees. This spanning tree is known as minimum spanning tree.

An MST is unique if all the weights of the edges are distinct. Otherwise, there may be minimum spanning trees. The algorithms below give out one such particular MST of all possible MSTs for the given graph.

Prim's algorithm

Prim's algorithm builds the minimum spanning tree gradually by adding edges one at a time. At first, the MST consists only of a single vertex. This can be the root vertex or any arbitrary vertex. Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree. Now, we have selected two vertices. Let this be a set of selected vertices. Then, we select an edge with the minimum weight that has one end as a selected vertex from the set of selected vertices. This process of adding the edge with minimum weight with one end in selected vertices set is done until all vertices are selected or the MST consists of $V - 1$ edges.

The process of extracting minimum edge is $O(E)$ if done naively. This extraction is performed for $V - 1$ times to get a complete MST. Therefore, total complexity is $O(EV)$. This is really slow.

But if we use a data structure like `std::priority_queue` which is based on a binary heap, the extraction is done in $O(\log V)$. This approach uses a slightly different method which is shared below.

Below is an implementation of the algorithm in `prim.hpp`, which contains a `struct Prim` derived publicly from `class  weightedGraph` from `graph.hpp` mentioned before.

```
// prim.hpp

#ifndef PRIM_HPP
#define PRIM_HPP

#include "graph.hpp"

struct Prim : public weightedGraph {
    const long long int INF = 0x3f3f3f3f3f3f3f3f;

    /**
     * @brief Constructs a new Prim object.
     *
     * @param _V Number of vertices.
     * @param _directed Default = false. specify true if graph is directed.
     */
    Prim(Vertex _V, bool _directed = false) :
        weightedGraph(_V, _directed) {}

    /**
     * @brief Constructs a new Prim object from a weighted Graph object.
     *
     * @param G The weighted Graph Object to be copied.
```

```
     */
    Prim(weightedGraph G) : weightedGraph(G) {}

    /**
     * @brief Constructs a new Prim object from an old one.
     * It calls the weightedGraph copy constructor.
     *
     * @param copy The old Prim object to be copied.
     */
    Prim(const Prim &copy) : weightedGraph(copy) {}

    /**
     * @brief Solves the problem of minimum spanning tree using Prim's
     * algorithm.
     *
     * @return ll The sum of all the weights in the minimum spanning tree.
     */
    long long int PrimMST() {
        long long int sumMST = 0LL;
        int weightCount = 0;
        std::vector<bool> visited(V + 1);
        std::priority_queue<Edge, std::vector<Edge>,
                                    std::greater<Edge>> pQ;

        // first - Edge weight.
        // second - Vertex.
        pQ.push(Edge(0, 1));

        while (not pQ.empty()) {
            Edge E = pQ.top();
            pQ.pop();
            Vertex u = E.second;
            if (visited[u])
                continue;
            visited[u] = true;
            sumMST += 1LL * E.first;
            weightCount++;
            if (weightCount == V)
                break;
            for (Edge e : adj[u]) {
                int v = e.first;
                if (visited[v])
                    continue;
                pQ.push(Edge(e.second, v));
            }
        }

        return sumMST;
    }
};

#endif
```

For this approach,

- Time complexity: $O(E \log V)$.
- Space complexity: $O(V + E)$.

Kruskal's algorithm

Kruskal's algorithm, initially, considers every node of the graph, V different isolated forests. Then, the edges are sorted based on their weights. At every iteration, the two trees on the ends of an edge are merged into one. Gradually, after E iterations, several trees are merged into one. The idea is to color all trees with different color in the beginning and as they get merged, color them the same color.

We are going to use a data structure known as Disjoint Set Union or Union Find due to its operations of combining any two sets (union) and finding the set a particular element belongs to (find). With a technique called path compression, these operations are reduced to $O(\log n)$ to $O(1)$ on average. Since the focus is on the algorithm, we will move on to the implementation.

This is a implementation very different from the other implementations in this book. It does not make use of the header `graph.hpp`. A different `strcut Edge` is defined for the purpose of this algorithm as there is a need to sort the edges in the ascending order of their weights.

```
// kruskal.cpp

#include<vector>
#include<algorithm>

struct Edge {
    int u, v, weight;
    Edge(int a, int b, int w) :
        u(a), v(b), weight(w) {}
    bool operator<(const Edge& other) const {
        return (weight < other.weight);
    }
};

struct DisjointSetUnion {
    int n;
    int *parent, *size;
    DisjointSetUnion(int _n) : n(_n) {
        parent = new int[n + 1]();
        size = new int[n + 1]();
        for (int i = 1; i <= n; ++i) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    /**
     * @brief Finds the parent of the set containing vertex v
     * with path compression.
     *
```

```
     * @param v The vertex whose parent set is to be found.
     * @return int The parent of the set containing v.
     */
    int findSet(int v) {
        if (v == parent[v])
            return v;
        parent[v] = findSet(parent[v]);
        return parent[v];
    }

    /**
     * @brief Unions two sets containing a and b.
     * Calls the findSet function.
     *
     * @param a The vertex a
     * @param b The vertex b
     * @return true Returns true if a union was done.
     * @return false Returns false if no union was done,
     * i.e., params a and b already belonged to the same
     * set.
     */
    bool unionSets(int a, int b) {
        a = findSet(a);
        b = findSet(b);
        if (a != b) {
            if (size[a] < size[b])
                std::swap(a, b);
            parent[b] = a;
            size[a] += size[b];
            size[b] = 0;
            return true;
        }
        return false;
    }
};

/**
 * @brief Solves the problem of minimum spanning tree
 * using Kruskal's algorithm.
 *
 * @param edges Container of edges.
 * @param V Number of vertices.
 * @return long long int The sum of all the weights in the minimum spanning tree.
 */
long long int
kruskalMST(std::vector<Edge> edges, int V) {
    std::sort(edges.begin(), edges.end());
    long long int sumMST = 0LL;
    DisjointSetUnion dsu(V);
    for (Edge e : edges) {
        if (dsu.unionSets(e.u, e.v)) {
            sumMST += 1LL * e.weight;
        }
    }
    return sumMST;
```

```
}
```

The average time for both `unionSets(a, b)` and `findSet(v)` is $O(1)$. These function is called for $O(E)$ times . The construction of the structure takes $O(V)$ time. The `std::sort` sorts the container in $O(E \log E) \approx O(E \log V)$ time.

Time $= O(E + V + E \log V) = O(E \log V)$.

Therefore, for this algorithm,

- Time complexity: $O(E \log V)$.
- Space complexity: $O(V + E)$.