Minimum Spanning Tree Algorithms

A minimum spanning tree is the subset of edges in an undirected connected graph such that all vertices in the graph are connected to each other, and the sum of the weights of each edge in the subset is minimal.

To find the minimum spanning tree, there are two algorithms generally used: Prim's algorithm and Kruskal's algorithm.

Prim's Minimum Spanning Tree Algorithm

Prim's MST algorithm runs in $O(V^2)$ time. The strategy goes as thus:

1. Take your graph G with V vertices and E edges, and pick an arbitrary vertex to serve as the starting point. This point serves as the beginning of your (currently) incomplete MST.
2. Find the vertex A such that A is not originally connected to the incomplete MST and the edge weight required to connect A to the incomplete MST is smallest out of all possible options for A.
3. The edge used to connect A to the starting point is part of the MST. Repeat step 2 on the incomplete MST until all vertices are connected together.

In order to execute step 2 in O(V) time, keep a 1D primitive array of size V with each entry N in the array representing the minimum cost of 'attaching' the corresponding vertex N to the MST. Initially, each entry should be set to infinity with the exception of the starting point which is set to 0. Whenever a vertex is found, each vertex adjacent should have their entry updated by setting their entries in the array to the minimum of the connecting edge weight and the entry already in the array.

Kruskal's Minimum Spanning Tree Algorithm

Kruskal's MST algorithm runs in ElogE time (or ElogV, since the maximum value of E is V^2). The strategy goes as thus:

1. Take all E edges in the initial graph and sort them by edge weight (if some edges weighted the same, it does not matter how they are ordered), and initialize an empty graph H of V vertices.
2. Iterate through every edge in ascending order, and stick the edge into H if the two vertices in H are not connected without the edge.
3. Output H. This is your MST.

In order to check if two vertices are connected or not in H, it is necessary to use a new data structure, called the Disjoint Set Data Structure or the Union-Find Algorithm.

The disjoint set data structure is a collection of N elements, numbered from 1 to N. Initially, each element is inside its own set of 1 element. Two operations can be executed on the set of sets:

1. Check if two numbers are contained within the same set
2. Combine two sets

With this new data structure, it becomes vastly easier to keep track of connectivity between vertices in H - each vertex starts out connected only to itself (the base form of a disjoint set), and adding edges combines sets of connected vertices.

The disjoint set data structure must be constructed manually within C++ (it is not a built-in). There is an implementation of the disjoint set data structure that will run with a time complexity equivalent to the inverse of the ackermann function. The ackermann function grows extremely quickly (well beyond time complexities like n^n), and thus its inverse grows extremely slowly, such that for all intents and purposes it can be considered to be O(1).

The implementation of the disjoint set data structure is not really important to innately understand. Good CS problems involve modifying algorithms, thus necessitating a good understanding of the inner workings of such algorithms, but there are no such modifications needed with the disjoint set. Thus, copy-pasting the code to make a disjoint set is acceptable. The derivation of the strategy to make a disjoint set with the best efficiency will be covered in another PDF.

For the implementations of both algorithms below, the input format is as follows: two integers N and M on the first line, representing the number of vertices and the number of edges respectively, followed by M lines of three integers A, B, and C in that order, denoting an edge that connects the vertices A and B with a weight of C.

Prim's Implementation

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

#define f first
#define s second

vector<pair<int,int>> G[1010];
int N,M,k[1010];
bool c[1010];

int main(){
    cin >> N >> M;
    for(int i = 0; i < M; i++){
        int V1,V2,D;
        cin >> V1 >> V2 >> D;
        G[V1].push_back({V2,D});
        G[V2].push_back({V1,D});
    }
    for(int i = 0; i < 1010; i++){
        k[i] = 1e+9;
    }
    k[1] = 0;
    int tsum = 0;
    for(int i = 0; i < N; i++){
        pair<int, int> tot = {1e+9,0};
        for(int q = 1; q <= N; q++){
            if(c[q] == 0) tot = min(tot, {k[q],q});
        }
        tsum += tot.f;
        c[tot.s] = 1;
        for(pair<int, int> u : G[tot.s]){
            k[u.f] = min(k[u.f],u.s);
        }
    }
    cout << tsum << endl;
    return 0;
}
```

Kruskal's Implementation

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

class UF    {
    int *id, cnt, *sz;
public:
    UF(int N)   {
        cnt = N;
     id = new int[N];
     sz = new int[N];
        for(int i=0; i<N; i++)  {
            id[i] = i;
         sz[i] = 1;
     }
    }
    ~UF()  {
     delete [] id;
     delete [] sz;
    }
    int fnd(int p)    {
        int root = p;
        while (root != id[root])
            root = id[root];
        while (p != root) {
            int newp = id[p];
            id[p] = root;
            p = newp;
        }
        return root;
    }
    void mrg(int x, int y){
        int i = fnd(x);
        int j = fnd(y);
        if (i == j) return;
        if   (sz[i] < sz[j]){
           id[i] = j;
           sz[j] += sz[i];
     } else     {
           id[j] = i;
           sz[i] += sz[j];
     }
```

```
        cnt--;
    }
    bool ctd(int x, int y)    {
        return fnd(x) == fnd(y);
    }
    int count() {
        return cnt;
    }
};

#define f first
#define s second

int N,M;
vector<pair<int, pair<int, int>>> edges;

int main(){
    cin >> N >> M;
    for(int i = 0; i < M; i++){
        int V1,V2,D;
        cin >> V1 >> V2 >> D;
        edges.push_back({D,{V1,V2}});
    }
    sort(edges.begin(), edges.end());
    UF ds(N);
    int tsum = 0;
    for(pair<int, pair<int, int>> i : edges){
        if(!ds.ctd(i.s.f, i.s.s)){
            ds.mrg(i.s.f,i.s.s);
            tsum += i.f;
        }
    }
    cout << tsum << endl;
    return 0;
}
```

Problem Set

https://dmoj.ca/problem/ds2 (Test your homebrew disjoint set data structure here)
https://wcipeg.com/problem/graph3p3
https://wcipeg.com/problem/coci097p4
https://dmoj.ca/problem/ccc17s4
https://dmoj.ca/problem/ccc10s4
https://dmoj.ca/problem/ccc18s5