Kruskal's and Prim's Algorithm

Minimum Spanning Tree

- **Tree** : Graph without cycles.
- **Spanning Tree** : It is subgraph that is a tree and includes all vertices and is always connected.
- **Minimum Spanning Trees** : Its is a spanning tree with minimum total edge weight.
- **Properties of a Minimum Spanning Tree:**
    - A MST of a graph is unique, if the edge weights are all distinct. Otherwise, there may be multiple MSTs of a graph.
    - MST is also the tree with minimum product of weights of the edges.(Replace the weights of all the edges by logarithm and use any of the MST algo)
    - In a MST, the maximum edge-weight is the minimum possible from all the possible spanning trees of the graph. (Validity for Kruskal's Algo)
    - The *maximum spanning tree* of a graph can be obtained by simply changing the signs of the edges and then apply any of the MST algo.

Kruskal's Algorithm

Jist of Kruskal's Algo : Initially places all the nodes of the graph isolated, creating a forest of single-noded trees. Then it gradually merges those trees with some edge(which must be present in the original graph). Before the execution of merge, the edges are *sorted in ascending order* of the edge-weights. Then in the process of unification(merging), the edges from the sorted list(or array) is *picked one by one*; if the ends of the current edge(picked edge) belongs to different subtrees then the subtrees are combined and the edge contributes to the answer. Pick the edges starting from minimum weight edge, such that it doesn't form cycle.

- Given a weighted undirected graph; we need to find the ***Minimum spanning tree*** of the graph.
- Greedy Algorithm
- 3 Properties should be maintained:
    - Tree
    - Spanning Tree
    - Minimum Spanning Tree
- To maintain the properties:
    - Avoid Cycles
    - Cycle Detection
        - DFS
        - DSU *(standard)*
- **Algorithm**
    - Time Complexity :

- Sorting the edges: ElogE
- Iterate all the edges: E
- Checking for cycle for each edge (DSU): (log*)E
- Total = ElogE + E(log*)E = O(ElogE)

```
MST-KRUSKALS(G, w):
  A = []
  for each vertex v ∈ G.V
    MAKE-SET(v)
  sort edges of G.E in non-decreasing order by weight w
  for each edge (u, v) ∈ G.E
    if GET(u) != GET(v):
      A.push_back({u, v})
      UNION(u, v)
  return A

[Graph G having 2 properties V(vertices) and E(edges)]
```

- **Proof of Correctness**:
  Claim: *If E' is a set of edges at any intermediate stage of the Kruskal's Algorithm, then there exists a MST that contains all the edges of E'*

PROOF: (By Induction)
Suppose we want to add an edge *e* in a MST *T*, which will result in formation of a cycle.
If a cycle is formed in *T* then in order to validate the condition for spanning tree we need to remove an edge from MST *T*. Let the edge to be removed is *e'*.

Hence,
Incoming Edge: *e*
Outgoing Edge: *e'*
MST: *T*
Now,

- the edge-weight of *e'* cannot be smaller than edge-weight of *e*, otherwise *e* would have already been chosen by Kruskal's.
- *e'* also cannot have a greater edge weight than *e*, otherwise *T - e' + e* would be MST, but *T* is already a MST.
- Hence, **the edge-weight of *e* == *e'*.** Therefore, ***T - e' + e* is also MST containing all the edges from *F + e*.**
  Hence, the claim is true.
- **Implementation**

```
ll Get(std::vector<int>& parent, int a) {
    if (parent[a] == a) {
        return a;
```

```
    } else {
        return parent[a] = Get(parent, parent[a]);
    }
}

ll Union(std::vector<int>& parent, std::vector<int>& rank, int a, int b) {
    a = Get(parent, a);
    b = Get(parent, b);

    if (rank[a] == rank[b]) {
        rank[a]++;
    }
    if (rank[a] > rank[b]) {
        parent[b] = a;
    } else {
        parent[a] = b;
    }
}

class Edge {
public:
    int src, dest, wt;
};

bool cmp(Edge& e1, Edge& e2) {
    return (e1.wt < e2.wt);
}

std::vector<Edge> kruskals(std::vector<Edge>& edges, int n) {
    std::vector<Edge> ans;
    std::vector<int> parent(n), rank(n, 1);

    loop(i, 0, n - 1) {
        parent[i] = i;
    }

    std::sort(edges.begin(), edges.end(), cmp);

    for (int i = 0;i < edges.size() and ans.size() < n;i++) {
        int psrc = Get(parent, edges[i].src);
        int pdest = Get(parent, edges[i].dest);

        if (psrc != pdest) {
            ans.emplace_back(edges[i]);
            Union(parent, rank, psrc, pdest);
        }
    }
    return ans;
}
```

Prim's Algorithm

- Very Similar to Dijkstra's Algorithm(used for shortest path)
- Prim's Algorithm has the property that edges in answer set *A* always forms a single tree. The tree starts from an arbitrary root vertex *r* and spans all the vertices *V* in the Graph *G*. Each iteration adds a lighter vertex to the set *A*, connecting the tree *T* to an isolated vertex. Only the safe edges are added to the set *A*, hence, when the algorithm terminates, the edges in set *A* forms a MST.
- **Algorithm and Implementation**
    - Time Complexity
    - Bfs -> O(V+E)
    - Since pq insertion involved -> O((V+E)(logV))
    - Total : O((V+E)(logV))

```
class Edge{
public:
    int src, dest, wt;
    Edge(int s, int d, int w){
        this->src = s;
        this->dest = d;
        this->wt = w;
    }
};

std::vector<std::list<pii>> g;

std::vector<Edge> prims(int root, int vertices){
    std::priority_queue<pii, std::vector<pii>, std::greater<pii>> pq;
    std::vector<int> parent(vertices, -1), edgewt(vertices, INT_MAX);
    std::vector<int> visited(vertices, 0);

    edgewt[root] = 0;
    for(int i=0;i<vertices;i++){
        pq.push({edgewt[i], i});
    }

    std::vector<Edge> ans;
    while(not pq.empty()){
        pii curr = pq.top();
        pq.pop();

        if(visited[curr.second])
            continue;

        visited[curr.second] = true;

        for(auto &ne:g[curr.second]){
            if(ne.second < edgewt[ne.first] and !visited[ne.first]){
                edgewt[ne.first] = ne.second;
                parent[ne.first] = curr.second;
                pq.push({ne.second, ne.first});
            }
```

```
            }
        }

        for(int i=0;i<vertices;i++){
            if(parent[i] == -1)
                continue;
            // Edge e = Edge(parent[i], i, edgewt[i]);
            ans.emplace_back(Edge(parent[i], i, edgewt[i]));
        }

        return ans;
    }
```