

Week3Lec2_Khush_2020101119

Main ideas:

- Greedy Algorithms
- Disjoint-Set Data structure

Greedy Algorithms

MST (Minimun Spanning Tree)

- Input Graph : $G = (V, E)$
- Edge Weights : w_e
- Tree produced : $T = (V, E')$

Kruskal's Algorithm

- greedy technique to determine a graph's MST.
- Algorithm
 - *Sort all the edges in non-decreasing order of their weight.*
 - *Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*
 - *Repeat step#2 until there are $(V-1)$ edges in the spanning tree.*

```
graph* kruskal(graph g[], int v){
    sort(g,w);
    graph T;
    for(int i=0;i<v;i++){
        if(does_form_cycle(g[i],T))
            continue;
        T.insert(g[i]);
    }
    return T; }
```

Time Complexity : $O(|E| \log |E|) + O(|E| \log^* |V|)$,

The CUT property

Let's assume edges X are part of a MST of $G = (V, E)$. Let us choose any subset of nodes S for which X does not cross between S and $V - S$, and let e be the lightest edge across this partition. Then $X \cup e$ is part of some Minimum Spanning Tree.

Proof.

If e is a part of T , then it is trivial.

But if e is not a part of T , let us construct another Minimum Spanning Tree, T' , which contains $X \cup \{e\}$. Now consider $T \cup \{e\}$. But since T is a spanning tree, it has only one edge connecting S and $V - S$. Let this edge be e' . Since e is the lightest edge connecting S and $V - S$, $w_e \leq w_{e'}$.

$$\begin{aligned} \text{weight}(T \cup \{e\} - \{e'\}) &= \text{weight}(T) + w_e - w_{e'} \\ &= \text{weight}(T) \end{aligned}$$

$T \cup \{e\} - \{e'\}$ is also a tree because it contains no cycle and has $n - 1$ edges. Thus, $T \cup \{e\} - \{e'\}$ is also a MST of G . Therefore, the cut property has been proven, and consequently, Kruskal's algorithm is correct.

Disjoint Set Union

- A **disjoint-set data structure** is a data structure that stores a collection of disjoint sets. It is also known as a **union-find data structure** or **merge-find set**. In the same way, it records a partition of a set into disjoint subsets.
- Consists of 3 operations
 - **makeset(v)**: Create a new set consisting only of the element v . The rank of a node v is the height of the subtree whose root is v . While rank is not needed to construct a DSU, it is one of the two very useful optimizations that we'll see.

```
function makeset(v):  
    parent(v) = v  
    rank(v) = 0
```

Here, the rank of a subtree is its height.

- **find(v)**: Return the 'representative' or the 'root' of the set containing v, which is an element in the same set as v. The naïve algorithm would be as follow.

```
function find(v):
    if v = parent(v):
        return v
    return parent(v)
```

The worst-case time complexity of the above function is $O(n)$ and an average of $O(\log n)$. A slight modification, known as the 'path compression' optimisation, can bring it down to an amortised $O(1)$ when used in conjunction with the union-rank optimisation.

```
function find(v):
    if v = parent(v): return v
    parent(v) = find(parent(v)) return parent(v)
```

- **union(u, v)**: Unify two sets

```
function union(u, v): a = find(u)
b = find(v) if a = b: return
if rank(a) > rank(b):
    parent(b) = parent(a) else:
    parent(a) = parent(b) if rank(a) = rank(b): rank(b) = rank(b) + 1
```

Properties of Ranks

1. For all x, $\text{rank}(x) < \text{rank}(\pi(x))$
2. Any root node of rank k has at least 2^k nodes in its tree.
3. If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.

Time Complexities

- Makeset $\Rightarrow O(1)$
- Find $\Rightarrow O(\log n)$

- Union $\Rightarrow O(\log n)$
- Overall $\Rightarrow O((|E| + |V|) \log |V|)$

Optimizing Find

Change the search function so that instead of all parent pointers following to the root of the tree, they are attached directly to the root node, making it $O(\log * n)$.