Lecture 6 (4th Sept)

Greedy Algorithms:

Greedy algorithms follow an approach that chooses the option or piece with the most obvious and immediate benefit, without worrying about future consequences.

- This behavior is myopic in nature, easy and convinient.

Minimum Spanning Tree:

Minimum spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

$$A\ tree\ T = (V, E')\ that\ minimizes\ weight(T) = \sum w_e$$

Greedy Approach To form MST by Kruskal's Algorithm:

- Select the lightest edge from the edge set
- Repeatedly add the next lightest edge that doesn't produce a cycle.

Few properties to keep in mind about Trees:

- A tree with n nodes has n-1 edges.
- Any connected, undirected graph G=(V,E) with |E|=|V|-1 is a tree.
- An undirected graph is a tree iff there is a unique path between pair of nodes.

Cut Property:

A cut is a partition of the vertices in two groups.

According to the cut property, if there is an edge in the cut set which has the smallest edge weight among all other edges in the cut set, then the edge should be included in some minimum spanning tree.

This property justifies kruskal's Algorithm.

Kruskal's Algorithm Implementation:

We start with a collection of disjoint sets, each containing the node of a particular component.

```
def makeset(x) : creates a singleton set just containing x

def find(x) : returns the set to which x belongs

def union(x,y): merge the sets containing x and y

def kruskal (G, w)  # Graph G and edge weights w
    for all u ∈V:   # vertex set
       makeset (u)
X = {}
sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
     add edge {u, v} to X
     union(u, v)
```

$$Time\ Complexity = O((|E| + |V|) * log|V|)$$

Data Structure for disjoint sets:

- We can store the set as a directed tree

- Nodes of the tree would be elements of the set, in no order, each with parent pointers eventually leading to root.
- The root element can be name of the set
- Thus each node has parent pointer, and a **rank** which is the height of the subtree hanging from that node

Building the tree:

To build, we make use of 2 functions and the union code:

```
def makeset(x)    # constant time operation
    π(x) = x
    rank(x) = 0
def find(x)       # follows parent pointers to the root of the tree
    while x = π(x) : x = π(x)
    return x
```

Now, to merge two trees, our goal to maintain computational efficiency:

- we make the root of the shorted tree point to the root of the longer tree to keep overall height short.
- Computing height is done by using rank numbers of their root nodes

```
def union(x, y)
    rx = find(x)
    ry = find(y)
    if rx = ry: return
    if rank(rx) > rank(r y):
        π(ry) = rx
    else:
        π(rx) = ry
        if rank(rx) = rank(ry) : rank(ry) = rank(ry) + 1
```

Properties regarding rank:

- For any node, rank of node is always lesser than rank of parent node. This makes sense as height of tree is always greater than height of its subtree
- Any root node of rank k has at least 2^k nodes in its tree.
- The maximum rank for a tree is $logn$ where n is the no of nodes. If there are n nodes, there can be at most $\frac{n}{2^k}$.

Path Compression:

If the edges given are sorted, we can try to increase efficiency.

The data structure part becomes the bottleneck. Thus we need to increase efficiency of find() and union() functions faster than logn.

We achieve this by intending to keep the trees short by introducing a small alteration in the find() function:

```
def find(x):
  if x = π(x) : π(x) = find(π(x))
  return π(x)
```

Now during each find, when a series of parent pointers is followed up to the root of a tree, we change the pointers so that they directly point to the root. This only slightly increases time for find() and the cost of the final algorithm is slightly more than $O(1)$.

Hence the time complexity becomes smaller than O(logn).