Algorithm Analysis and Design

Pratyay Suvarnapathaki, 2020111016

Week 3, Lecture 1

The Minimum Spanning Tree (MST) Problem

The problem statement is quite simple.
Given a graph $G = (V, E)$ and its edge weights w_e,
A tree $T = (V, E')$ which spans the graph, but has the least total edge weight sum $\Sigma_{e \epsilon E'} w_e$
i.e. the **Minimum Spanning Tree** is to be found, as efficiently as possible.

Kruskal's Algorithm

Kruskal's Algorithm is a *greedy* approach, wherein the main idea is to repeatedly add the *lighest possible edge*, provided it doesn't result in the formation of a cycle.
Such an approach guarantees that the resultant edge weight sum has the least feasible value.
i.e.

1. Sort the set of all edges E in ascending order of edge weight.
2. Create a new graph
3. Till the number of edges in the new graph reaches $|V| - 1$,
    - Pick the next edge in the sorted set.
    - If it forms a cycle, reject it and continue
    - If it does not form a cycle, add it into the new graph and continue

At the end, this 'new graph' is nothing but the MST itself!

Pseudocode:

```
class Graph:
    def kruskal():
        Graph newGraph
        while (newGraph.NUM_EDGES < self.NUM_VERTICES – 1):
            e = self.sorted_edge_list[newGraph.NUM_EDGES++]
            if(forms_cycle(e))
                continue
            else
                newGraph.insert(e)
        return newGraph
```

The Cut Property

For a connected graph, a cut is a partition of the vertices of a graph into two disjoint subsets.
The cut property says that if one of the edges of the cut has weight smaller than any other edge in

the cut then it is in the MST.

Proof by Contradiction

- We assume there is some new edge e which is not in the MST.
- We then try to construct another hypothetical MST T' containing $E' \cup e$, where E' is the current set of MST edges.
- By definition of MST, this new edge e results in the creation of a cycle between some sets of vertices S and $V - S$, each of which will also have another edge e'(by definition of cycle).
- Thus, $T' = T \cup e - e'$
- We know that T' is a tree as it has V veritces and $E - 1$ edges.
- But the weight of T' = weight(T) $+w_{e'} - we$
- This, strangely enough, results in the creation of a new tree with smaller weight, thereby contradicting the minimality of the MST.

Hence, proved

Disjoint-Set Data Structure

Also known as union-find, the disjoint set data structure maintains a collection of two sets such that no element belongs to more than one set.
It supports two main operations:
find() and unite()
Each set has a representative, which can be found by following the chain starting at that element. Upon joining two sets, the two representatives are connected and one of them is made the representative of the new set. The efficiency of the data structure depends on how the sets are joined. Hence, the representative must be chosen carefully.\

The find prodcedure:
while (x != $\pi(x)$:
$x = pi(x)$
return x

The union procedure:
r_x=find (x)
r_y=find (y)
if $r_x = r_y$: return
if rank(r_x) > rank(r_y)
$\pi(r_y) = r_x$
else:
$\pi(r_x) = r_y$
if rank(r_x) = $rank$(r_y): rank(r_y) = $rank$(r_x) + 1

Properties of rank:\

1. For any x, rank (x) < rank $(\pi(x))$.
2. Any root node of rank k contains at least $2k$ nodes in its tree, owing to the union procedure.
3. If there are n elements overall, there can only be $n/2k$ nodes of rank k.

Time complexities of DSU operations:\

- Make_Set => $O(1)$
- Find => $O(\log n)$
- Union => $O(\log n)$
- Overall => $O((|E| + |V|) \log |V|)$

Making find() more efficient:
Instead of parent pointers *leading* to the root of the tree, make them point directly to the root.

```
if x != pi(x):
    pi(x) = find(pi(x))
return pi(x)
```

As a result of using a DSU, the final complexity obtained is:
sorting $|E|$ elements + $O(|E| \log^* |V|)$
