Algorithm Analysis and Design (CS1.301)

Monsoon 2021, IIIT Hyderabad
04 August, Saturday (Lecture 6)

Greedy Algorithms

Minimum Spanning Trees (Kruskal's Algorithm)

Given an undirected graph $G = (V, E)$ with edge weights $w_e > 0$, find a tree $T = (V, E')$, where $E' \subseteq E$, which minimises

$$\text{weight}(T) = \sum_{e \in E'} w_e.$$

Kruskal's algorithm consists of starting with an empty graph and repeatedly adding the next lightest edge that doesn't produce a cycle.
We now need to prove its correctness.

Proof of Correctness

First, consider the *cut property* of MSTs. Let a subset of edges $X \subseteq E$ be part of a MST of $G = (V, E)$, and $S \subseteq V$ is be any subset of nodes for which X does not cross between S and $V - S$. Let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.

To prove this, let the edges X be part of some MST T. If the new edge $e \in T$, then we are done; therefore assume $e \notin T$. We will construct a different MST T' containing $X \cup \{e\}$.
Note that if we add e to T, we create a cycle containing some other edge e' connecting S and $V - S$. Thus, let $T' = T \cup \{e\} - \{e'\}$. Now T' is connected (since e' is a cycle edge) and has the same number of edges as T – thus it is also a tree.
Further, the weight of T' must be $\leq$ that of T, since \$w_e $w_{e'}$. But T is an MST; therefore $\text{weight}(T) \leq \text{weight}(T')$ also. This shows us that T' and T have the same weight, and therefore T' is also an MST.

This property says that the MST is preserved at every step of Kruskal's algorithm; thus we can conclude that Kruskal's algorithm is correct.

Disjoint Set Implementation

```
procedure kruskal(G,w)
    for all u \in V: makeset(u)

    X = {}
    sort E by weight
    for all e = {u,v} \in E in increasing order:
        if find(u) != find(v):
            add e to X
            union(u,v)
```

This algorithm involved $|V|$ makeset operations, $2|E|$ find operations, and $|V|-1$ union operations.

For this, we will use the disjoint-set data structure, implemented in a directed-tree representation. The "name" of a set is its root node, which is returned by the `find` function; the *rank* of a node is the height of the subtree hanging from it; and $\pi(x)$ represents the parent of x.

```
procedure makeset(x)
    pi(x) = x
    rank(x) = 0

function find(x)
    while x != pi(x): x = pi(x)
    return x

procedure union(x,y)
    r_x = find(x)
    r_y = find(y)

    if r_x == r_y: return
    if r_x > r_y: pi(r_y) = r_x
    else: pi(r_x) = r_y

    if rank(r_x) = rank(r_y): rank(r_y) += 1
```

The rank function has has three important properties:
1. For any x, $\text{rank}(x) < \text{rank}(\pi(x))$.
2. Any node of rank k has at least 2^k nodes in its tree.
3. If there are n elements overall, there are at most $\frac{n}{2^k}$ nodes of rank k.

Therefore, the makeset operation is $O(1)$, the find operation is $O(\log n)$ and the union operation is $O(\log n)$. This makes the overall complexity $O((|E| + |V|) \log |V|)$.

Path Compression

We can improve this by using path compression. During each find, when a series of parent pointers is followed up to the root of a tree, we will change all these pointers so they point directly to the root.

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

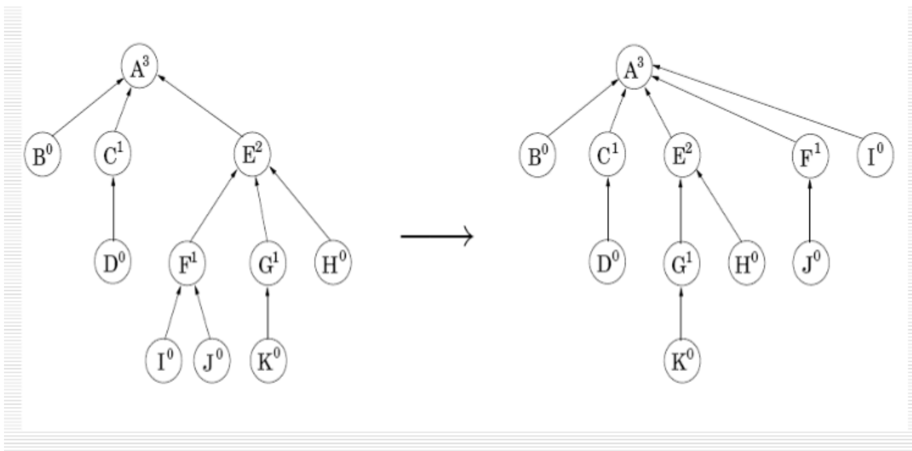

Figure 1: An Example of the Modified `find`

We now need to find the running time of this modified `find`.
The time taken by a specific find operation is simply the number of pointers followed. Let us first divide the natural numbers into intervals according to the $\log *$ function:

$$\{1\}, \{2\}, \{3,4\}, \{5,6,\ldots,16\}, \{17,18,\ldots,2^{16}=65536\}, \{65537, 65538, \ldots 2^{65536}\}, \ldots$$

Now, for any x, either the rank of $\pi(x)$ is in a higher interval than the rank of x, or it is in the same interval.

Firstly, there are at most $\log *n$ nodes of the first type.

Secondly, each time x is of the second type, its parent changes to one of higher rank. Therefore, if x's rank is in the interval $\{k+1, \ldots, 2^k\}$, then its parent's rank reaches a higher interval after at most 2^k times. After this, it is never in the same type again.

Therefore, the overall time for m `find` operations is $O(m \log *n)$ plus ($2^k \times$ the number of nodes of rank $> k$). Thus the amortised complexity of the modified `find` is $O(\log *n)$, which makes the complexity of Kruskal's algorithm $O(|E| \log *|V|)$.