18 (L) Kruskal's Algorithm, Union-Find data structure

Recall Kruskal's algorithm for computing MST: consider edges in order from smallest to biggest, keep each edge unless it would create a cycle with edges already chosen. How to implement this?

First idea:

- Initialize T to be an empty graph.
- Sort the edges.
- For each edge (u, v), run a DFS from u in T in and see if we can reach v. If so, adding (u, v) would create a cycle so discard it; otherwise, add (u, v) to T.

How fast is this? Sorting the edges takes $\Theta(E \log E) = \Theta(E \log V)$. ($E$ is $O(V^2)$ so $\log E$ is $O(\log V^2) = O(2 \log V) = O(\log V)$.) Running a DFS in T takes $\Theta(V + E) = \Theta(V)$ (since T will never have more edges than vertices), and in this algorithm we run a DFS once for each edge, for a total of $\Theta(VE)$, which dominates the $\Theta(E \log V)$ time to sort the edges. Of course, $\Theta(VE)$ can be as big as $\Theta(V^3)$ if there are a lot of edges. Can we do better?

Yes, we can! Notice that for each edge (u, v), doing a DFS is sort of overkill, in the sense that it actually tries to *find a path* from u to v, but we don't care about the path, only whether u and v are already connected or not. In this case—when we only care about which vertices are connected and not about the actual paths between them—we can test for connectivity much faster.

The basic idea is to come up with some sort of data structure to maintain a set of connected components. Given such a data structure, the algorithm looks something like this:

- Start every vertex in its own connected component.
- For each edge, test whether its vertices are in the same component (which means it would form a cycle) or in different components.
- If the endpoints are in different components, add the edge to T, and merge the two components into one.

This kind of data structure is called a *union-find* structure, and from the algorithm above we can see what operations it needs to support:

- MakeSets(n): create a union-find structure containing n singleton sets.
- Find(v): return the *name* of the set containing v. ("Name" could be anything, typically we will use integers.) To check whether two vertices u and v are in the same connected component, we can test if Find(u) = Find(v).
- Union(x, y): merge the two sets whose names are x and y.

This has lots of applications! (See homework.)

Given such a data structure, we can implement Kruskal's algorithm as follows:

Algorithm 9 KRUSKAL(G)

Require: Weighted, undirected, connected graph $G = (V, E)$.

```
1: T ← empty set of edges
2: Sort the edges of G by weight
3: U ← MakeSets(n)
4: for each edge e = (u, v) from smallest to biggest do
5:     if U.Find(u) ≠ U.Find(v) then
6:         Add e to T
7:         U.Union(u, v)
```

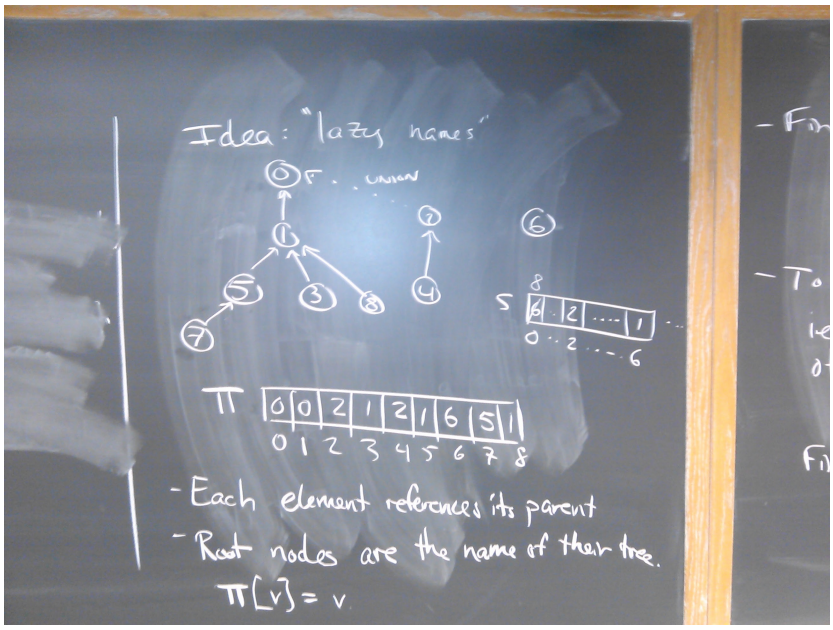
- Sorting edges by weight takes time $\Theta(E \log E) = \Theta(E \log V)$.
- We do $2|E|$ FIND operations.
- We do at most $|V| - 1$ UNION operations.

Hence, the overall time is $O(E \log V + T_{\text{MakeSets}} + ET_{\text{Find}} + VT_{\text{Union}})$. We would really like for T_{Find} and T_{Union} to be $O(\log V)$ (or better), which would make the whole thing $O(E \log V)$.

First idea: just keep a dictionary that maps each node to an "id" which identifies its set.

- FIND is $\Theta(1)$. (Assuming $\Theta(1)$ dictionary lookup.)
- But to do UNION we have to go through and change all the ids of one of the sets. This could be $O(n)$. Not good enough!

$\log n$, eh? This should make us think of trees. Idea: *forest* (multiple trees) of vertices where each points to its parent. (Parents don't need to know about their children.) We can represent this simply with an array/dictionary where $\pi[v] = p$ means p is the parent of v; by convention, $\pi[v] = v$ means v is a root. Each tree is a set; the root of the tree will be used as the name of the set. Nodes are given an id "lazily"—might not point directly to its id.



- To do FIND, just follow pointers up to the root. That is, given v, look up $\pi[v]$, then $\pi[\pi[v]]$, and so on, until finding a root.
- To do UNION(x, y), just merge the trees by setting one to be the parent of the other, that is, $\pi[x] \leftarrow y$. Note that this only works if x and y are already roots. (For convenience, one might also want to implement a version of UNION which can work on any nodes, by calling FIND on each first and then doing the union.)

Clearly UNION is $\Theta(1)$, hooray! But what about FIND? It seems like it might be $O(n)$ in the worst case, if we end up with an unbalanced tree. But if we are clever/careful in how we implement UNION, this won't happen!

- Keep track of the size of each set (*i.e.* in a separate array/dictionary).
- When doing UNION, always make the smaller set a child of the larger (and update the size of the larger in $\Theta(1)$).

Theorem 18.1. FIND *takes* $\Theta(\log n)$ *time.*

Proof. The distance from any node up to its root is, by definition, the number of times its set has changed names. But the name of a node's set only changes when it is unioned with a *larger* set. So each time a set changes names, its size must at least double. The total size of a set can't be larger than n; hence the most time any element can have its set change names, and therefore its maximum depth, is $O(\log n)$. ∎

One can also implement *path compression*: when doing FIND, update every node along the search path to point directly to the root. This does not make FIND asymptotically slower, and it speeds up subsequent FIND calls. One can show (although the proof is somewhat involved—it would probably take two lectures or so) that FIND then takes essentially $\Theta(\lg^* n)$ on average, where $\lg^* n$ is *iterated logarithm* of n, defined as the number of times the lg function must be iterated on n before reaching a result of 1 or less. This means that the largest number for which $\lg^* n = k$ is $n = 2^{2^{2^{\cdots}}}$ with k copies of 2 stacked up in a tower of exponents! So, for example, $\lg^* n \leq 5$ for all $n \leq 2^{2^{2^{2^2}}} = 2^{2^{2^4}} = 2^{2^{16}} = 2^{65536}$, a number with 19729 decimal digits (for comparison, the number of particles in the entire universe is estimated at around 10^{80}, a number with a measly 81 decimal digits). So although *in theory* $\lg^* n$ is not a constant—it does depend on n, and can get arbitrary large as long as n is big enough—*in practice*, in our universe, it is essentially a constant (and a rather small constant at that). This means that both FIND and UNION can be implemented to run in (essentially) constant time!

Note, however, that this does not change the asymptotic running time for Kruskal's algorithm, which is still dominated by the $\Theta(E \log V)$ time needed to sort the edges.