7.1 Introduction and Problem Definition

In this lecture we'll talk about the Union-Find problem, which is also sometimes called the *disjoint sets* problem. Informally, for this problem we want to maintain a data structure of disjoint sets that allows to to take the union of sets, and to figure out which set a given element is in. There are many uses for this, particularly in graph algorithms (which we'll see later). For now, we're just doing it as an interesting data structure.

Slightly more formally, we want a data structure which supports the following operations:

1. Make-Set(x): create a new set containing just the element x, i.e. the set $\{x\}$.
2. Union(x, y): replace the set containing x (call it S) and the set containing y (call it T) with the single set $S \cup T$.
3. Find(x): return the representative for the set containing x

Let's examine these operations, particularly Find, in a little more detail. Suppose we're at some point in the operation of this data structure, so there is some collection of disjoint sets. We require every set to have a *unique* representative, which must be an element in the set. So if we call Find(x) and Find(y) and x and y are in the same set, the two calls must return the same representative (which could be x, y, or some other element z in the same set). And of course, since the representative must be in the set if x and y are in different sets then when we call Find on them we must get back different representatives. So we can think of the representative as being the "name" of the set.

We'll see a few ways of doing this reasonably efficiently, and in the homework you'll see an extremely efficient method. One nice thing about Union-Find is that even simple things work pretty well, but we don't hit a limit to improvement until we get very, very strong bounds.

Let's fix some notation to start: there are m operations total, n of which are Make-Set operations (so the "number of elements" is n, like usual).

7.2 Lists

One simple data structure would to keep a linked list for each set. In the simplest version, we have a list for set and each element is an element in the list. We will also add a pointer from each element to the head of the list (the head will be the representative). So Make-Set(x) is easy: we just set $x \to head = x$ and $x \to next = NULL$. Find(x) is also easy: we can just return $x \to head$.

But what about Union(x, y)? Let S be the list containing x, and let T be the list containing y. The obvious thing to do is traverse S until we get to the final element (say z), then set

$z \to next = y \to head$, then walk down T (starting from $y \to head$) setting all of the head pointers to $x \to head$. This takes $O(|S| + |T|)$ time. Since $|S|$ and $|T|$ might each be $\Omega(n)$, this gives a bound of only $O(n)$ on Unions (note that Make-Set and Find each take constant time).

Note that this can be bad even when $|S|$ or $|T|$ is small, say even a single element! Here's one obvious improvement: add T to the middle is S, rather than to the end. Instead of walking down S to get z to be the final element, we could set $z = x \to next$, then set $x \to next = y \to head$, then walk down T until we get to the final element w and set $s \to next = z$. This decreases the time to $O(|T|)$. Unfortunately, this can still be bad – clearly $|T|$ can be $\Omega(n)$.

Let's make one more modification. In the head node for each set we'll store the size of the set, i.e. the length of the list. This lets us always insert the shorter list into the longer one, so the time for a Union becomes $O(\min\{|S|, |T|\})$. Clearly this minimum can also be n, but now it turns out we can get a strong amortized bound!

Theorem 7.2.1 *The total running time is $O(m + n \log n)$ (recall that m is the total number of operations and n is the number of Make-Set operations).*

Proof: Find and Make-Set are clearly constant time, so their total cost is covered in the $O(m)$. So we need to argue that the total cost of all of the Union operations is $O(m + n \log n)$. In fact, we'll see that they only cost $O(n \log n)$.

To bound the total cost of all of the Unions, we can split up the cost among each element (this is sometimes called *charging* the elements). Suppose we do a union of S and T where $|T| \leq |S|$. Then this times time $O(|T|)$, so we can charge each element in T an $O(1)$ amount to pay for it (note that no elements in S get charge). For each element e, let $\alpha(e)$ be the total charge to e of all the operations. The total running time is thus

$$\sum_e \alpha(e),$$

so we just need to analyze how many times each element can be charged (note the similarity to what we did last class with the binary counter, where we switched to analyzing it bit-by-bit rather than operation-by-operation).

So how large can $\alpha(e)$ be? By definition, e is only charged when it is in the smaller of the two sets being unioned. This means that whenever e is charged, the size of the set containing it at least doubles! Thus $\alpha(e) \leq O(\log n)$, and thus the total running time of all of the Union operations is at most $O(n \log n)$. ■

Is this analysis tight? Let's give an example to show that it is, i.e. the Union operations might really cost $\Omega(n \log n)$ using this algorithm. Suppose we first do all n Make-Set operations, then use $n/2$ Unions to create $n/2$ sets of size 2, then $n/4$ unions to create $n/4$ sets of size 4, etc. Since the running time of Union(S, T) is $\Omega(\min\{|S|, |T|\})$, the time to union two equal sized sets is at least the size of the sets. Thus the first set of Unions takes time $\frac{n}{2} \cdot 1$, the second set takes time $\frac{n}{4} \cdot 2$, the third set takes times $\frac{n}{8} \cdot 4$, etc. At level i of this process, the total work is $\frac{n}{2^i} \cdot 2^{i-1} = n/2$. Since there are $\log n$ levels, this means the total running time is $\Omega(n \log n)$.

7.3 Forests

How can we improve on this? Here's a crazy (and not particularly good) idea: let's make Finds slower and Unions faster. Note that this is a terrible idea, since we might execute an arbitrary number of Finds but the number of Unions is at most $n-1$ (i.e. m can be arbitrarily larger than n). Nevertheless, let's see what happens when we do this.

The basic idea we're going to use is to switch from lists to trees. This is not a crazy idea – clearly every list is a tree, so it makes sense that by allowing more general structures we might be able to use the extra flexibility to get better running time bounds. And in the end, that's what we'll do.

So now each set is supposed to be a tree, and our data structure overall is a forest. In the simplest version, all of the algorithms are straightforward. Make-Set(x) is simple – we can just set $x \to parent = x$ and return x. For Find(x), we can simply follow parent pointers until we hit the root (which will be the representative). For Union(x, y) we can follow pointers until we reach the root of the tree containing x (call it z) and the root of the tree containing y (call it w). Then we can set w's parent pointer to z.

Is this a good algorithm? Obviously not. We could, for example, call Make-Set n times and then call Union $n-1$ times to gradually build up a single large set, which goes from size 1 to n. So in the ith iteration of this, we'll call Union on one large set (of size i) and one set of size 1. If the small set is the first parameter of the Union call, then this turns our forest into one big long linked list! And now we don't even have head pointers, just parent pointers, so each Find can take time $\Omega(n)$.

7.3.1 Union By Rank

Recall the fix we used for lists – we inserted the smaller one into the larger one. Can we do something similar with forests? The answer turns out to be "yes", but we need to be a little careful. In particular, instead of tracking *size*, we will track *depth*. For now, let's define the *rank* of a node to be its distance from its furthest descendent, i.e. its height in the tree (we will later change this definition slightly, which is why we call it the rank and not the depth or the height). Note that this is easy to initialize in Make-Set to just be equal to 0.

If every node keeps track of its rank, this gives us extra power when performing a Union. In particular, the pseudocode for Union(x, y) will now be:

1. Let $z \leftarrow$ Find-Set(x) and $w \leftarrow$ Find-Set(y)
2. If $z \to rank < w \to rank$ then set $z \to parent$ to w
3. If $w \to rank < z \to rank$ then set $w \to parent$ to z
4. If $z \to rank == w \to rank$ then
   (a) Set $z \to parent$ to w
   (b) Increment $w \to rank$ by 1.

This procedure is called Union-by-Rank, and intuitively it prevents the formation of a very long chain. Let's actually prove this. Recall that the height of a node is the maximum over all of its descendants of the length of the path to that descendant.

Lemma 7.3.1 *The rank of a node is always equal to its height.*

Proof: We proceed by induction on the operations of the algorithm. Initially there are no elements so the lemma is trivially true. Now suppose that it is true up to some point in time t. Find has no effect on ranks, so it would still be true after a call to Find. Make-Set creates a tree consisting of a single node of rank 0, which is the height of that node and thus all ranks are still equal to heights.

Now consider what happens on a call to Union(x, y). In the first or second case, where the ranks are unequal, the shorter tree becomes a child of the deeper tree. This means that no ranks change: the new root (say w) now has the descendants of z as its own descendants, but the maximum distance to them is at most $z \to rank + 1$ (by the inductive hypothesis) which is at most $w \to rank$.

In the last case, where the rank are equal, the new parent does indeed have a longer path to one of its new descendants, but we increase its rank by 1 so the invariant is maintained. ■

Lemma 7.3.2 *Any node of rank r has at least 2^r nodes in its subtree.*

Proof: We will again do induction on the algorithm. Initially there are no nodes so it is trivially true. It's also trivially true after a Find or Make-Set operation (Make-Set results in a new node of rank 0 with 1 nodes in its subtree, as claimed).

Suppose we do Union(x, y). In the first two cases, where the ranks are unequal, then clearly the invariant is maintained – the new child's subtree is exactly what it was before, and the new root has the same rank but even more nodes in its subtree. In the final case, where the two ranks are the same, we win by induction: we know inductively that if both w and z have rank r, they must have at least 2^r nodes in their subtrees (which are obviously disjoint). So when we make w the parent of z, the total number of nodes in w's new subtree is at least $2^r + 2^r = 2^{r+1}$, and the rank or w becomes $r + 1$. ■

Theorem 7.3.3 *The running time of any operation is $O(\log n)$*

Proof: The running time of Make-Set is constant, so clearly $O(\log n)$. To bound the running time of Find, note that Lemma 7.3.2 implies that every node has rank at most $\log n$, and then Lemma 7.3.1 implies that the height of the tree is at most $O(\log n)$. Clearly the running time of Find is simply the height of the tree, since Find just follow parent pointers until it hits the root. Thus Find takes time $O(\log n)$. Similarly, the Union operation just needs to execute two Finds and then do a constant amount of work (pointer switching and incrementing rank), so also takes time $O(\log n)$. ■

This theorem implies that the *total* work for m operations is at most $O(m \log n)$. This is worse than the $O(m + n \log n)$ that we got from using lists, but on the other hand this is a worst-case bound rather than an amortized bound. With lists, it is possible that single union could take $\Omega(n)$ time. That is not possible when we use Union-by-Rank and trees.

It turns out that this bound is tight: m operations really can take $\Omega(m \log n)$ time. An easy example of this is essentially the same set of operations from the bad example for lists. If we first

do all n Make-Sets, we can always Union together equal-sized sets. It is easy to argue inductively that that the heigh of each tree really is its rank, and that at the end of the process the root will have rank $\log n$. Then we can execute m Find commands and each one will really take $\log n$ time.

7.3.2 Path Compression

The main reason we had to use time $\Omega(\log n)$ was the time to travel from a leaf up to the root when doing a Find. This is in some sense unavoidable, but one key idea is that there's no reason for this to happen many times. We're not tied to binary trees (and Union-by-Rank definitely does not ensure that all trees are binary), so why not just move everything up to the root?

Slightly more formally, let's change the algorithm for Find(x) to be the following:

1. If $x \to parent == x$ return x (since x must be the root
2. Set $x \to parent$ to Find($x \to parent$), return $x \to parent$

It turns out that path compression on its own, even without Union-by-Rank, gives significant improvement over the trivial $\Omega(n)$ bound for operations. When we combine them, though, we get a huge improvement: m operations will take only $O(m\alpha(n))$ time, where α is what's known as the *Inverse Ackermann Function*. This function is a little complicated to define, but it grows ridiculously slowly. For example, α of the number of atoms in the universe is less than 4.

We won't prove this, as it is quite difficult. On the homework you'll need to prove a weaker bound, of $O(m \log^* n)$. Here $\log^* n$ is the iterated log function, i.e. the number of times we need to take the log before we get a number at most 1.