Lecture 15

Graph Algorithms III: Union-Find

15.1 Overview

In this lecture we describe the *union-find* problem. This is a problem that captures the key task we had to solve in order to efficiently implement Kruskal's algorithm. We then give two data structures for it with good amortized running time.

15.2 Motivation

To motivate the union-find problem, let's recall Kruskal's minimum spanning tree algorithm.

Kruskal's Algorithm (recap):
Sort edges by length and examine them from shortest to longest. Put each edge into the current forest if it doesn't form a cycle with the edges chosen so far.

We argued correctness last time. Today, our concern is running time. The initial step takes time $O(|E| \log |E|)$ to sort. Then, for each edge, we need to test if it connects two different components. If it does, we will insert the edge, merging the two components into one; if it doesn't (the two endpoints are in the same component), then we will skip this edge and go on to the next edge. So, to do this efficiently we need a data structure that can support the basic operations of (a) determining if two nodes are in the same component, and (b) merging two components together. This is the union-find problem.

15.3 The Union-Find Problem

The general setting for the union-find problem is that we are maintaining a collection of disjoint sets $\{S_1, S_2, \ldots, S_k\}$ over some universe, with the following operations:

MakeSet(x)**:** create the set $\{x\}$.

Union(x, y)**:** replace the set x is in (let's call it S) and the set y is in (let's call it S') with the single set $S \cup S'$.

Find(x): return the unique ID for the set containing x (this is just some representative element of this set).

Given these operations, we can implement Kruskal's algorithm as follows. The sets S_i will be the sets of vertices in the different trees in our forest. We begin with MakeSet(v) for all vertices v (every vertex is in its own tree). When we consider some edge (v, w) in the algorithm, we just test whether Find(v) equals Find(w). If they are equal, it means that v and w are already in the same tree so we skip over the edge. If they are not equal, we insert the edge into our forest and perform a Union(v, w) operation. All together we will do $|V|$ MakeSet operations, $|V| - 1$ Unions, and $2|E|$ Find operations.

Notation and Preliminaries: in the discussion below, it will be convenient to define n as the number of MakeSet operations and m as the total number of operations (this matches the number of vertices and edges in the graph up to constant factors, and so is a reasonable use of n and m). Also, it is easiest to think conceptually of these data structures as adding fields to the items themselves, so there is never an issue of "how do I locate a given element v in the structure?".

15.4 Data Structure 1 (list-based)

Our first data structure is a simple one with a very cute analysis. The total cost for the operations will be $O(m + n \log n)$.

In this data structure, the sets will be just represented as linked lists: each element has a pointer to the next element in its list. However, we will augment the list so that each element also has a pointer directly to head of its list. The head of the list is the representative element. We can now implement the operations as follows:

MakeSet(x): just set `x->head=x`. This takes constant time.

Find(x): just return `x->head`. Also takes constant time.

Union(x, y): To perform a union operation we merge the two lists together, and reset the head pointers on one of the lists to point to the head of the other.

Let A be the list containing x and B be the list containing y, with lengths L_A and L_B respectively. Then we can do this in time $O(L_A + L_B)$ by appending B onto the end of A as follows. We first walk down A to the end, and set the final `next` pointer to point to `y->head`. This takes time $O(L_A)$. Next we go to `y->head` and walk down B, resetting head pointers of elements in B to point to `x->head`. This takes time $O(L_B)$.

Can we reduce this to just $O(L_B)$? Yes. Instead of appending B onto the end of A, we can just splice B into the middle of A, at x. I.e., let `z=x->next`, set `x->next=y->head`, then walk down B as above, and finally set the final `next` pointer of B to `z`.

Can we reduce this to $O(\min(L_A, L_B))$? Yes. Just store the length of each list in the head. Then compare and insert the shorter list into the middle of the longer one. Then update the length count to $L_A + L_B$.

We now prove this simple data structure has the running time we wanted.

Theorem 15.1 *The above algorithm has total running time $O(m + n \log n)$.*

Proof: The Find and MakeSet operations are constant time so they are covered by the $O(m)$ term. Each Union operation has cost proportional to the length of the list whose head pointers get updated. So, we need to find some way of analyzing the total cost of the Union operations.

Here is the key idea: we can pay for the union operation by charging $O(1)$ to each element whose head pointer is updated. So, all we need to do is sum up the costs charged to all the elements over the entire course of the algorithm. Let's do this by looking from the point of view of some lowly element x. Over time, how many times does x get walked on and have its head pointer updated? The answer is that its head pointer is updated at most $\log n$ times. The reason is that we only update head pointers on the *smaller* of the two lists being joined, so every time x gets updated, the size of the list it is in at least doubles, and this can happen at most $\log n$ times. So, we were able to pay for unions by charging the elements whose head pointers are updated, and no element gets charged more than $O(\log n)$ total, so the total cost for unions is $O(n \log n)$, or $O(m + n \log n)$ for all the operations together. ■

Recall that this is already low-order compared to the $O(m \log m)$ sorting time for Kruskal's algorithm.

15.5 Data Structure 2 (tree-based)

Even though the running time of the list-based data structure was pretty fast, let's think of ways we could make it even faster. One idea is that instead of updating all the head pointers in list B (or whichever was shorter) when we perform a Union, we could do this in a lazy way, just pointing the head of B to the head of A and then waiting until we actually perform a find operation on some item x before updating its pointer. This will decrease the cost of the Union operations but will increase the cost of Find operations because we may have to take multiple hops. Notice that by doing this we no longer need the downward pointers: what we have in general is a collection of trees, with all links pointing *up*. Another idea is that rather than deciding which of the two heads (or roots) should be the new one based on the *size* of their sets, perhaps there is some other quantity that would give us better performance. In particular, it turns out we can do better by setting the new root based on which tree has larger *rank*, which we will define in a minute.

We will prove that by implementing the two optimizations described above (lazy updates and union-by-rank), the total cost is bounded above by $O(m \lg^* n)$, where recall that $\lg^* n$ is the number of times you need to take $\log_2$ until you get down to 1. For instance,

$$\lg^*(2^{65536}) = 1 + \lg^*(65536) = 2 + \lg^*(16) = 3 + \lg^*(4) = 4 + \lg^*(2) = 5.$$

So, basically, $\lg^* n$ is never bigger than 5. Technically, the running time of this algorithm is even better: $O(m\alpha(m, n))$ where α is the inverse-Ackermann function which grows even more slowly than $\lg^*$. But the $\lg^* n$ bound is hard enough to prove — let's not go completely overboard!

We now describe the procedure more specifically. Each element (node) will have two fields: a *parent* pointer that points to its parent in its tree (or itself if it is the root) and a rank, which is an integer used to determine which node becomes the new root in a Union operation. The operations are as follows.

MakeSet(x)**:** set x's rank to 0 and its parent pointer to itself. This takes constant time.

Find(x)**:** starting from x, follow the parent pointers until you reach the root, updating x and all the nodes we pass over to point to the root. This is called *path compression.*

The running time for Find(x) is proportional to (original) distance of x to its root.

Union(x, y)**:** Let Union(x, y) = Link(Find(x), Find(y)), where Link(root1,root2) behaves as follows. If the one of the roots has larger rank than the other, then that one becomes the new root, and the other (smaller rank) root has its parent pointer updated to point to it. If the two roots have *equal* rank, then one of them (arbitrarily) is picked to be the new root *and its rank is increased by 1.* This procedure is called *union by rank.*

Properties of ranks: To help us understand this procedure, let's first develop some properties of ranks.

1. The rank of a node is the same as what the height of its subtree would be if we didn't do path compression. This is easy to see: if you take two trees of *different* heights and join them by making the root of the shorter tree into a child of the root of the taller tree, the heights do not change, but if the trees were the *same* height, then the final tree will have its height increase by 1.
2. If x is not a root, then rank(x) is strictly less than the rank of x's parent. We can see this by induction: the Union operation maintains this property, and the Find operation only increases the difference between the ranks of nodes and their parents.
3. The rank of a node x can only change if x is a root. Furthermore, once a node becomes a non-root, it is never a root again. These are immediate from the algorithm.
4. There are at most $n/2^r$ nodes of rank $\geq r$. The reason is that when a (root) node first reaches rank r, its tree must have at least 2^r nodes (this is easy to see by induction). Furthermore, by property 2, all the nodes in its tree (except for itself) have rank $< r$, and their ranks are never going to change by property 3. This means that (a) for each node x of rank $\geq r$, we can identify a set S_x of at least $2^r - 1$ nodes of smaller rank, and (b) for any two nodes x and y of rank $\geq r$, the sets S_x and S_y are disjoint. Since there are n nodes total, this implies there can be at most $n/2^r$ nodes of rank $\geq r$.

We're now ready to prove the following theorem.

Theorem 15.2 *The above tree-based algorithm has total running time $O(m \lg^* n)$.*

Proof: Let's begin with the easy parts. First of all, the Union does two Find operations plus a constant amount of extra work. So, we only need to worry about the time for the (at most $2m$) Find operations. Second, we can count the cost of a Find operation by charging \$1 for each parent pointer examined. So, when we do a Find(x), if x was a root then pay \$1 (just a constant, so that's ok). If x was a child of a root we pay \$2 (also just a constant, so that's ok also). If x was lower, then the *very rough* idea is that (except for the last \$2) every dollar we spend is shrinking the tree because of our path compression, so we'll be able to amortize this cost somehow. For the remaining part of the proof, we're only going to worry about the steps taken in a Find(x) operation up until we reach the child of the root, since the remainder is just constant time per operation. We'll analyze this using the ranks, and the properties we figured out above.

Step 1: let's imagine putting non-root nodes into buckets according to their rank. Bucket 0 contains all non-root nodes of rank 0, bucket 1 has all of rank 1, bucket 2 has ranks 2 through $2^2 - 1$, bucket 3 has ranks 2^2 through $2^{2^2} - 1$, bucket 4 has ranks 2^{2^2} through $2^{2^{2^2}} - 1$, etc. In general, a bucket has ranks r through $2^r - 1$. In total, we have $O(\lg^* n)$ buckets.

The point of this definition is that the number of nodes with rank in any given bucket is at most $n/$(upper bound of bucket + 1), by property (4) of ranks above.

Step 2: When we walk up the tree in the Find(x) operation, we need to charge our steps to something. Here's the rule we will use: if the step we take moves us from a node u to a node v such that v is in the *same* bucket as u, then we charge it to node u (the walk-ee). But if v is in a higher bucket, we charge that step to x (the walk-er).

The easy part of this is the charge to x. We can move up in buckets at most $O(\lg^* n)$ times, since there are only $O(\lg^* n)$ different buckets. So, the total cost charged to the walk-ers (adding up over the m Find operations) is at most $O(m \lg^* n)$.

The harder part is the charge to the walk-ee. The point here is that first of all, the node u charged is not a root, so its rank is never going to change. Secondly, every time we charge this node u, the rank of its new parent (after path compression) is at least 1 larger than the rank of its previous parent by property 2. Now, it is true that increasing the rank of u's parent could conceivably happen $\log n$ times, which to us is a "big" number. *But*, once its parent's rank becomes large enough that it is in the next bucket, we never charge node u again as walk-ee. So, the bottom line is that the maximum charge any node u gets is the range of his bucket.

To finish the proof, we just said that a bucket of upper bound $B - 1$ has at most n/B elements in it, and its range is $\leq B$. So, the total charge to all walk-ees in this bucket over the entire course of the algorithm is at most $(n/B)B = n$. (Remember that the only elements being charged are non-roots, so once they start getting charged, their rank is fixed so they can't jump to some other bucket and start getting charged there too.) This means the *total* charge of this kind summed *over all buckets* is at most n times the number of buckets which is $O(n \lg^* n)$. ■ (Whew!)