Disjoint set (Union-Find)

For Kruskal's algorithm for the minimum spanning tree problem, we found that we needed a data structure for maintaining a collection of disjoint sets. That is, we need a data structure that can handle the following operations:

- MAKESET(x) - create a new set containing the single element x
- UNION(x, y) - replace two sets containing x and y by their union.
- FIND(x) - return the name of the set containing the element x

Naturally, this data structure is useful in other situations, so we shall consider its implementation in some detail.

Within our data structure, each set is represented by a tree, so that each element points to a parent in the tree. The root of each tree will point to itself. In fact, we shall use the root of the tree as the name of the set itself; hence the name of each set is given by a canonical element, namely the root of the associated tree.

It is convenient to add a fourth operation LINK(x, y) to the above, where we require for LINK that x and y are two roots. LINK changes the parent pointer of one of the roots, say x, and makes it point to y. It returns the root of the now composite tree y. With this addition, we have UNION(x, y) = LINK(FIND(x),FIND(y)), so the main problem is to arrange our data structure so that FIND operations are very efficient.

Notice that the time to do a FIND operation on an element corresponds to its depth in the tree. Hence our goal is to keep the trees short. Two well-known heuristics for keeping trees short in this setting are UNION BY RANK and PATH COMPRESSION. We start with the UNION BY RANK heuristic. The idea of UNION BY RANK is to ensure that when we combine two trees, we try to keep the overall depth of the resulting tree small. This is implemented as follows: the rank of an element x is initialized to 0 by MAKESET. An element's rank is only updated by the LINK operation. If x and y have the same rank r, then invoking LINK(x, y) causes the parent pointer of x to be updated to point to y, and the rank of y is then updated to $r+1$. On the other hand, if x and y have different rank, then when invoking LINK(x, y) the parent point of the element with smaller rank is updated to point to the element with larger rank. The idea is that the rank of the root is associated with the depth of the tree, so this process keeps the depth small. (**Exercise:** Try some examples by hand with and without using the UNION BY RANK heuristic.)

The idea of PATH COMPRESSION is that, once we perform a FIND on some element, we should adjust its parent pointer so that it points directly to the root; that way, if we ever do another FIND on it, we start out much closer to the root. Note that, until we do a FIND on an element, it might not be worth the effort to update its parent pointer, since we may never access it at all. Once we access an item, however, we must walk through every pointer to the root, so modifying the pointers only changes the cost of this walk by a constant factor.

```
procedure MAKESET(x)
    p(x) := x
    rank(x) := 0
end


function FIND(x)
    if x ≠ p(x) then
      p(x) := FIND(p(x))
      return(p(x))
end


function LINK(x,y)
    if rank(x) > rank(y) then x ↔ y
    if rank(x) = rank(y) then rank(y) := rank(y) + 1
    p(x) := y
    return(y)
end


procedure UNION(x,y)
    LINK(FIND(x),FIND(y))
end
```

In our analysis, we show that any sequence of m UNION and FIND operations on n elements take at most $O((m+n)\log^* n)$ steps, where $\log^* n$ is the number of times you must iterate the $\log_2$ function on n before getting a number less than or equal to 1. (So $\log^* 4 = 2, \log^* 16 = 3, \log^* 65536 = 4$.) We should note that this is not the tightest analysis possible; however, this analysis is already somewhat complex!

Note that we are going to do an *amortized analysis* here. That is, we are going to consider the cost of the algorithm over a sequence of steps, instead of considering the cost of a single operation. In fact a single UNION or FIND operation could require $\Theta(\log n)$ operations. (**Exercise:** Prove this!) Only by considering an entire sequence of operations at once can obtain the above bound. Our argument will require some interesting accounting to total the cost of a sequence of steps.

Digression on Amortized Analysis: Dynamic Arrays

To introduce the idea of amortized analysis we consider a simple problem. (Note that this problem is completely unrelated to the Union-Find Problem we were discussing.

The problem we consider is that of maintaining an array of unknown length when the operating system only allocates arrays of fixed length. In our problem we simply INSERT elements $A[1]$, $A[2]$, and so on upto $A[n]$ into this array, but n is not known at the beginning.

Our solution starts be creating an array of length 1. When the second INSERT happens, it asks the OS for an array of size 2, copies over the old array and insert new element in 2nd location. In general, the current array will have size 2^i. When the $2^i + 1$th INSERT occurs the OS will be asked to allocate an array of size 2^{i+1} (double the current size). The old array of size 2^i will be copied over and the new element inserted into location $2^i + 1$. The next $2^i - 1$ inserts happen normally in this array, and then again we do the doubling and copying.

The goal in this exercise is to show that the sequence of n INSERTS takes $O(n)$ time to compute. Most of the INSERTs (those that are not inserting the $2^i + 1$th elements take O(1) time. We call these the TYPE 1 INSERTS. The occasional INSERT (inserting the $2^i + 1$th element, referred to as TYPE 2 INSERTs below) take longer and in particular have worst case cost of $O(n)$. Yes we claim that the total cost of the n INSERTs is $O(n)$.

To see this we charge costs of different INSERTs differently. The TYPE 1 costs are associated directly to the INSERT operation. These lead to a cost of $O(1)$ per INSERT and thus a total of $O(n)$ cost for n INSERTS. The TYPE 2 INSERTs we charge first to the array that is allocated then, and then total them up separately later. When the array of size 2^i is created this thus leads to a charge of 2^i to this array which suffices to pay for the cost of this TYPE 2 INSERT. But now we have to account for all the charges charged to the arrays. We note that the arrays created have size 1, 2, 4 and so on till 2^k where $2^{k-1} < n \leq 2^k < 2n$ (where the final inequality is equivalent to the first one). The total TYPE 2 cost is at most the cost charged to these arrays which is $1 + 2 + 4 + \cdots 2^k \leq 2^{k+1} \leq 4n$. We thus find that the cost of all TYPE 2 INSERTS is also upper bounded by $O(n)$ leading to a total (TYPE 1 + TYPE 2) insertion cost of $O(n)$ for n INSERTs.

The above illustrates the principle of amortized analysis — even when single steps of an operation may take a lot of time, no sequence can consistently take a lot of time, and this can even be shown with relatively simple accounting.

Returning to Analysis of Union-Find

Returning to our goal of analyzing a sequence of m FINDs and UNIONs on n elements takes time $O((m+n)\log^* n)$. (We assume UNIONs are executed by two FINDS and a LINK and so the dominant cost is that of all the FINDs).

We now give the charging strategy and then prove that it works. The overview of this strategy is as follows. Suppose a FIND(v) leads to the path $v \to v_1 \to v_2 \to \cdots \to v_\ell$ where v_ℓ is the root of the tree containing v before the FIND is executed. We associate to each edge $v_i \to v_{i+1}$ a type in $\{$TYPE 1, TYPE 2$\}$. We show that the number of TYPE 1 edges in any path is $O(\log^* n)$ and so can be charged directly to the FIND operation. The charges for a TYPE 2 edge $v_i \to v_{i+1}$ we charge to v_i. Later we do (a careful) analysis to show that the total cost of TYPE 2 charges in a sequence of m UNIONs and FINDS is at most $O(n\log^* n)$ and this will yield the bound on overall.

To define TYPE 1 and TYPE 2 edges we need some more notation. To each element v, we let final-rank(v) be the rank(v) after the execution of all UNIONs and FINDs. We group elements based on their final rank. The different groups are [0], [1], [2], $(2,4]$, $(4,16]$, ... $(k,2^k]$,... We refer to a generic group as $(k,2^k]$. We say that v is in group $(k,2^k]$ if $k <$ final-rank$(v) \le 2^k$. Note that rank$(v) \le n$ (in fact a good exercise is to prove rank$(v) \le \log n$ for every v), we have that there are $O(\log^* n)$ groups with elements in them. Now returning to a FIND(v) leading to the path $v = v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$ we say that edge $v_i \to v_{i+1}$ is of TYPE 1 if $i+1 = \ell$ or v_i and v_{i+1} are in different groups. We say edge $v_i \to v_{i+1}$ is of TYPE 2 if $i+1 < \ell$ and v_i and v_{i+1} are in the same group. This completes the description of the charging strategy and we now turn to showing why the number of TYPE 1 charges and the number of TYPE 2 charges are each $O((m+n)\log^* n)$.

To do so, we first make a few observations about rank.

1. If $v \neq p(v)$ then rank$(p(v)) >$ rank(v) and final-rank$(v) =$ rank(v).
2. Whenever $p(v)$ is updated, rank$(p(v))$ increases. Specifically if before a FIND we have $p(v) = u$ and after the FIND $p(v) = w$ and $u \neq w$ ("$p(v)$ is updated") then after the FIND we have rank$(u) <$ rank(w).
3. The number of elements v with final-rank$(v) = k$ is at most $\frac{n}{2^k}$.
4. The number of elements v with final-rank$(v) \ge k$ is at most $\frac{n}{2^{k-1}}$.

The fact that rank(v) does not change once it is not the root of its subtree follows from the definition of the algorithm. It is also clear that the rank of a node is strictly smaller than the rank of its parent when the LINK happen. And after that the rank of the child does not change while the rank of the parent may only increase. This yields

property (1) above. Property (2) follows from Property (1). For instance if we execute FIND(u) and the path under consideration is $u = v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$ and $v = v_i$ then the parent of v_i changes only if $i+1 < \ell$. In this case by Property (1) rank$(v_i) <$ *rank*$(v_{i+1}) <$ rank(v_ℓ) and so the new parent of v has larger rank than its old parent. (Another way the rank of the parent of v changes is if v was previously the root of a tree and now links to a new parent. Again by our algorithm the new parent has rank strictly larger than the old parent.) Property (3) requires a bit more care and we prove it below, but see that Property (4) follows immediately from Property (3) since the number of nodes with final-rank$(v) \geq k$ is upper bounded by $\sum_{t=k}^{\infty} n/2^t \leq n/2^{k-1}$.

To prove Property (3), let $S_v = \{u|$ u was in subtree rooted at v at the first instance when rank$(v) =$ final-rank$(v)\}$.

Claim 1: if final-rank$(u) =$ final-rank$(v) = k$ then S_u and S_v are disjoint and have size at least 2^k.

Proof: We prove by induction on rank that at any stage of the algorithm, a tree with root w has at least $2^{\text{rank}(w)}$ elements in the tree. To see this consider the first time that w reaches its current rank. Previously its rank must have been rank$(w) - 1$ and it must have been linked with another tree whose root had the same rank. By induction each tree has a least $2^{\text{rank}w-1}$ nodes and so after the LINK, the union of the trees has at least $2^{\text{rank}(w)}$. It follows that S_v has size at least 2^k.

Now to see that S_v and S_u are disjoint, suppose for contradiction that some element w was in the subtree rooted at u when u attained its final rank and also in v when v attained its final rank. Assume wlog that v attained its final rank later. This can only be possible if the entire subtree containing u was linked to the subtree rooted at v at some point after u attained its final rank. But at this stage rank(v) would need to be strictly larger than the final-rank(u) (by Property (1)). This yields the desired contradiction.

End of proof of Claim 1.

Given the Claim Property (3) is immediate since the sets associated with elements of final rank k are disjoint subsets of a universe of size n and each subset has size at least 2^k. It follows that the number elements of final rank at most k is at most $n/2^k$.

Exercise: Show that the maximum rank an item can have is $\log n$.

Now we turn to counting the TYPE 1 and TYPE 2 costs.

Claim 2: Every FIND has at most $\log^* n + 1$ TYPE 1 edges.

Proof: Fix a FIND(v) that explores the path $v = v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$.

We first note that for every $i < \ell$, we have final-rank$(v_i) <$ final-rank(v_{i+1}). This is so since by property (1)

we have that rank$(v_i) <$ rank(v_{i+1}) when the FIND(v) was executed. Since v_i is no longer a root, its rank does not change any more and so final-rank$(v_i) =$ rank(v_i). And rank(v_{i+1}) can only grow with time and so final-rank$(v_{i+1}) \geq$ rank(v_{i+1}). Putting these together yields final-rank$(v_i) <$ final-rank(v_{i+1}).

In particular this implies that the group numbers of elements are non-decreasing on the path $v = v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$ and so can increase at most $\log^* n$ times, since that is the total number of distinct groups with elements. Finally including the case $i+1 = \ell$ yields one more TYPE 1 edge and still we have total number of TYPE 1 edges in any single FIND is at most $\log^* n + 1$.

End of proof of Claim 2.

Thus the TYPE 1 cost per FIND is $O(\log^* n)$. Since the total number of FINDs is m we get that the total TYPE 1 cost of m FINDs is $O(m \log^* n)$. Finally we turn to the slightly more complex accounting of the number of TYPE 2 edges. For this the following claim is crucial.

Claim 3: For every v with final-rank$(v) \in (k, 2^k]$, the number of TYPE 2 edges of the form $v \to p(v)$ with final-rank$(p(v)) \leq 2^k$ encountered over all the m FINDs is at most 2^k.

Proof: Note that every time some FIND(u) goes through the element v with a TYPE 2 edge charged to v the parent of v is updated. And by Property (2) when the parent of v is updated its rank grows. This can only happen 2^k times before the rank of the parent exceeds 2^k. The claim follows.

End of proof of Claim 3.

Once we have this claim we are done by Property 4. The total number of all TYPE 2 edges associated with vertices in group $(k, 2^k]$ is $2^k \times (n/2^{k-1}) = O(n)$. The number of different groups is $O(\log^* n)$ and combining the two we get that the total cost of processing all TYPE 2 edges is $O(n \log^* n)$. We conclude:

Theorem 7.1 *An sequence of m UNIONs and FINDs on a universe of n elements takes $O((m_n) \log^* n)$ steps using the tree data structure with path compression and union by rank.*