CS41: Extra Notes on Union-Find

The following notes provide additional information on the union-find data structure used to implement Kruskal's Algorithm, to supplement Homework 4. For the curious, it explains how some of the improvements to the data structure are implemented in order to speed up FIND operations. You do not need to know these details in order to do the homework, and you won't be expected to know the details of the Union-Find implementation described here on an exam.

Basic Union-Find Pseudocode

Below is the pseudocode provided at the end of Homework 4, implementing the basic version of the Union-Find data structure using a collection of trees whose root nodes are used to identify a set (all nodes in the same tree are considered to be in the same set). Finding the set containing an element then amounts to finding the root node of the tree it's in, which we can do by following a chain of pointers from each node to its parent, until we reach the root node. We identify the root node by setting its parent to be itself.

```
CreateDisjointSets(S)
1  for each element x ∈ S
2      π(x) = x

Find(x)
1  if π(x) == x
2      return x
3  return Find(π(x))

Union(T1, T2)
1  // T1 and T2 must be roots of different trees
2  π(T1) = T2
```

Improvements

As mentioned in the homework, FIND may take time $\Omega(n)$ in the worst case (see optional part (f) of problem 3). The homework mentions two main improvements that speed up FIND operations:

1. **Union By Rank.** The first idea is to keep track of the height of each tree (or, more precisely, for each node x, the height of the subtree rooted at x). Each element x has a value called the *rank*, initialized to 0, corresponding to this height:

   ```
   CreateDisjointSets(S)
   1  for each element x ∈ S
   2      π(x) = x
   3      rank(x) = 0
   ```

The union-by-rank idea says that when merging two trees, if they have different ranks, the tree with the larger *rank* should be the root of the merged tree. The root of the lower-rank tree therefore has its parent set to the root of the higher-rank tree. As long as the trees don't have the same rank, this doesn't increase the overall height at all. Only when both trees have the same rank does the overall height increase, in which case it increases by one (if both trees have height r, the root of the merged tree will now have a child whose height is r, so the overall height is $r + 1$). Pseudocode is shown below:

```
UNION(T1, T2)
1  // T1 and T2 must be roots of different trees
2  if rank(T1) < rank(T2)
3        π(T1) = T2
4  else
5        π(T2) = T1
6        if rank(T1) == rank(T2)
7              rank(T1) = rank(T1) + 1
```

Running Time: With this strategy, merging two trees of rank r gives a tree of rank $r + 1$ (and otherwise the ranks do not change). A fun exercise to try is to prove the following property by induction: *A tree whose root node has rank r contains at least 2^r elements.* A consequence of this is that a tree with k nodes has height at most $\log_2 k$, and so FIND takes time at most $O(\log(k))$. Thus, with n elements, union-by-rank ensures all trees have height at most $\log_2(n)$ and thus FIND always takes time at most $O(\log n)$.

Thus, union-by-rank alone gets the running time of FIND down to $O(\log n)$ (while CREATEDISJOINTSETS stays $O(n)$ and UNION is still $O(1)$).

2. **Path Compression.** The key insight to path compression is that after a successful FIND operation on a node x, we have the root node s of its tree and can simply set $\pi(x) = s$ so that future FIND operations are free (at least until future UNION operations potentially move it further down the tree again). Pseudocode is given below.

```
FIND(x)
1  if π(x) ≠ x
2        π(x) = FIND(π(x))
3  return π(x)
```

In the above, we first find the root of x (recursively, as before, by calling FIND($\pi(x)$)), and then set the parent of x to point directly to the root (i.e., x becomes a direct child of the root); then, we return $\pi(x)$, which is now the root of x.

A nice detail to note here: because of the recursive nature of FIND, not only does this set the parent of x to the root, but it also does so *for every node on the path from x to the root* since each recursive call also updates the parent of its node to the root. Thus, the entire *path* from x to the root gets *compressed* to one edge, and likewise for every node on that path.

Notice, however, that we don't update the rank of nodes anywhere in this process. Once we compress paths, the *rank* of a root node is no longer necessarily equal to the height of the

compressed tree. This is one reason why "rank" is a better term than "height". We can instead think of the rank as indicating the height of the hypothetical uncompressed version of the tree, and thus also as providing a lower bound on the number of nodes in the tree (that is, at least 2^{rank} nodes).

Analysis: Although we don't update the rank of any node during the path compression, all of the analysis for union-by-rank above still holds: as discussed above, the rank tells us what the height of the tree *would* be if we didn't compress any paths; we can then reason about the ranks and numbers of nodes in each tree by reasoning about this hypothetical uncompressed version (after all, it has the same number of nodes, all with the same ranks as before, we've just added some "shortcuts" for speeding up FIND). That is to say, when using union-by-rank, path compression helps us shortcut paths to speed up future FINDs without changing anything else, so the asymptotic running time should be *at least as good* as union-by-rank without path compression (the added work of updating the parent pointers only changes things by a constant factor), or $O(\log n)$.

Of course, a single FIND operation may still need $\Theta(\log n)$ time, since until FIND(x) gets called for the first time, the path from the node x to the root is still uncompressed, and the uncompressed path length could be $\Theta(\log n)$ in the worst case. The benefit of path compression is rather a *future* benefit. Reasoning about this requires an *amortized* analysis (you may recall seeing amortized running times with ArrayLists: adding an element to an ArrayList may take $O(n)$ time due to resizing, but the *amortized* running time is $O(1)$ since a sequence of n insertions takes a total time of only $O(n)$ rather than $O(n^2)$).

The amortized analysis for FIND is quite complicated, so we won't see it here. It was first analyzed in 1973, when it was shown that with n elements total, the amortized running time of FIND is $O(\log^* n)$, where $\log^*$ is the *iterated logarithm* and is equal to the number of times we need to repeatedly take the log of n before we get down to 1 (or less). This grows much more slowly than the regular log: for logarithms, $\log(2x) = \log x + 1$ (i.e., doubling x increases the log by one), whereas for iterated logarithms, $\log^*(2^x) = \log^*(x) + 1$ (that is, doubling x is not enough, and we need to go all the way up to 2^x before the iterated log increases by one). For any realistic value of n, $\log^*(n) \leq 5$ (e.g., the $\log^*$ of the number of atoms in the observable universe is 5); so this is *effectively* as good as constant in practice.

The amortized analysis was later improved to $O(\alpha(n))$. The function $\alpha(n)$ is the inverse of the *Ackermann function*, which itself appears in other areas of CS theory. The Ackermann function is defined recursively and grows absurdly fast (far faster than exponential functions), and its *inverse* $\alpha(n)$ thus grows absurdly slowly, even more slowly than $\log^*(n)$.