Notes on Disjoint Set Union

These notes provide a simplified analysis of the amortized running time of the disjoint set union data structure compared to the one presented in the textbook. The running time analysis is worse. We only achieve $O(m \log\log n)$, $O(m \log\log\log n)$ and $O(m \log^* n)$ running times for m operations on n items instead of the $m\alpha(n)$ bound in the text. However, understanding this analysis will help you understand the one in the textbook.[1]

Read Chapter 21 of the textbook if you are not already familiar with the Disjoint Set data structure with Union by Rank and Path Compression. Make sure that you understand rank and how union by rank guarantees that a Find operation takes $O(\log n)$ time. We define size(x) to be the number of items in the subtree rooted at x (including x itself). We make the following observations about rank(x) and size(x).

Observations

1. If x is not a root, then rank$(x) <$ rank(parent(x)). Note that this is strictly less than.
2. If x becomes a non-root, then rank(x) becomes fixed forever.
3. If x is a root, size$(x) \geq 2^{\text{rank}(x)}$. This is an easy proof by induction on rank. When the rank of the root is 0, the tree has 1 item and the claim holds. A rank of $r+1$ can only be achieved by linking two trees with roots with rank r. This means the size of the tree doubles while the rank increases by 1.
4. For $r \geq 0$, there are at most $n/2^r$ items that ever achieve rank r during the entire run of the data structure (i.e., not just for a fixed point in time).

   Pf: This obviously holds for $r = 0$. Now, fix an $r > 0$. (Note: this is not a proof by induction.) When an item x achieves rank r, label all of its descendants with x. By above, at least 2^r items are labeled with x. Furthermore, note that none of the descendants of x have ever been labeled, because this has to be the first time that any of these nodes have a root with rank r. (In this labeling procedure, we are only considering rank r. The claim for each rank is done separately.) Also, none of the descendants of x will ever have a different root with rank r in the future. Thus, each item is labeled at most once during the entire run of the data structure. Since there are only n items and at least 2^r items are labeled by each label, at most $n/2^r$ items can ever achieve rank r. □
5. For all items x, rank$(x) \leq \lfloor \log n \rfloor$. Otherwise, size$(x) > n$.

Amortized Analysis 1: $O(m \log\log n)$

A sequence of m operations of the disjoint set data structure must include n MakeSet operations followed by some number of Link and Find operations. Assuming that Link is already given the roots of the two subtrees, this takes $O(1)$ time and we won't change that. The Find operation is more costly since it has to implement path compression. With the exception of the root and the child of the root, each node visited by the Find operation must be examined and in the second pass, must have its pointer adjusted to point to the root. This takes $O(1)$ time per node and the time must be "charged" to someone. We will either make the Find operation pay for these visits or make the node pay for the visit to itself. (The Find operation will pay for visiting the root and the child of the root, but that is just another $O(1)$.)

[1] I have lost track of the origins of this analysis. If anyone knows where it came from, please let me know.

So, we have two things to do:

1. Explain when we charge a visit to the Find operation and when we charge a visit to the node.
2. Add up the charges to the Find operation and the charges to the nodes.

How we charge a visit depends on the rank of a node and the rank of its parent. To do this we separate the rank values into *blocks*. (The definition of the blocks are a bit arbitrary, so don't try to make sense of them right away.) The blocks are named B_0, B_1, B_2, ... We want the size of the blocks to increase exponentially:

$$B_0 = \{0\}, \quad B_1 = \{1\}, \quad B_2 = \{2, 3\}, \quad B_3 = \{4, 5, 6, 7\},$$

$$B_4 = \{8, 9, 10, \ldots, 15\}, \quad B_5 = \{16, 17, 18, \ldots, 31\}, \quad B_6 = \{32, 33, 34, \ldots, 63\}, \quad \ldots$$

Note that for $i > 0$, the number of values in block B_i is 2^{i-1}. Also, there can be at most $\log\log n$ blocks since the largest rank is at most $\log n$.

Now, consider a node x that is visited during a Find operation. If rank(x) and rank(parent(x)) are in the same block (before path compression), then we charge the visit to x to x itself. Otherwise, rank(x) and rank(parent(x)) are in different blocks and we charge the visit to x to the Find operation.

Think about the Find operation as it follows the pointers from a node y to the root of the tree. The ranks of the nodes along this path are strictly increasing. Now, think about the blocks that these rank values belong to. Every time the rank values enter a new block, the Find operation is charged for a visit. Since there are only $\log\log n$ blocks, the Find operation is charged for at most $\log\log n$ visits. Thus, the amortized running time of Find is $O(\log\log n)$.

Now, think about the charges to a node x. Each time a node x is charged, its parent's rank must increase by at least 1. (Recall that this is because x has a new parent which has larger rank than x's old parent and not because x's old parent had its rank increased.) After a while rank(parent(x)) is big enough to be in a different block from rank(x). Then, x will never be charged again. For example, suppose rank(x) is 16 which puts rank(x) in block B_5. Then rank(parent(x)) is at least 17. After 15 charges to x, rank(parent(x)) must be at least 32 which puts rank(parent(x)) in block B_6. Thus, after 15 charges, x will never be charged again. In general, a node with rank r, where $r \in B_i$, can be charged at most 2^{i-1} times. (Verify this yourself.) We could have claimed that a node with rank r is charged at most r times, but it will help us in our calculations below to make the weaker claim of 2^{i-1} instead. By treating all the ranks in block B_i the same, we can remove one of the summations.

Now, we can bound the total number of node charges to *every* node in the data structure:

$$\sum_{i=1}^{\log\log n} \sum_{r \in B_i} \left(\frac{n}{2^r}\right) 2^{i-1}.$$

Here the first summation sums over each block and the second summation sums over each rank in the block. The value $\frac{n}{2^r}$ is the number of nodes with rank r and 2^{i-1} is the number of times that a node with rank in B_i can be charged. Now a bit of arithmetic gives us

$$\begin{aligned}
\sum_{i=1}^{\log\log n} \sum_{r \in B_i} \left(\frac{n}{2^r}\right) 2^{i-1} &= \sum_{i=1}^{\log\log n} \sum_{r=2^{i-1}}^{2^i-1} \left(\frac{n}{2^r}\right) 2^{i-1} \le \sum_{i=1}^{\log\log n} \left[2^{i-1}\left(\frac{n}{2^{2^{i-1}}}\right) 2^{i-1}\right] \\
&= n \sum_{i=1}^{\log\log n} \left(\frac{2^{2i-2}}{2^{2^{i-1}}}\right) < n \sum_{i=1}^{\log\log n} 1 = n \log\log n.
\end{aligned}$$

Here's how we got rid of the second summation (the one over r). There are 2^{i-1} terms in the sum and the largest term is $\left(\frac{n}{2^{2^{i-1}}}\right)2^{i-1}$, when $r = 2^{i-1}$ Thus,

$$\sum_{r=2^{i-1}}^{2^i-1}\left(\frac{n}{2^r}\right)2^{i-1} \le 2^{i-1}\left(\frac{n}{2^{2^{i-1}}}\right)2^{i-1}.$$

Now of the m operations, n are MakeSet and the rest are either Find or Link. If we charge each MakeSet operation $\log\log n$ credits, this would pay for all of the node charges used during path compression for *all* of the Find operations. Then, MakeSet takes $O(\log\log n)$ amortized time, Find takes $O(\log\log n)$ amortized time and Link takes $O(1)$ real time. Thus, the total running time for m operations is $O(m\log\log n)$.

Amortized Analysis 2: $O(m\log\log\log n)$

The analysis in the previous section is not tight. We can see this because we used the estimate that

$$\left(\frac{2^{2i-2}}{2^{2^{i-1}}}\right) < 1.$$

This is a rather gross over estimate and suggest that we can improve the analysis by making the nodes pay for more of the time used for path compression. We do this by making the blocks larger. In our second analysis, we redefine the blocks by:

$$B_0 = \{0,1\}, \quad B_1 = \{2,3\}, \quad B_2 = \{4,5,6,\ldots,15\}, \quad B_3 = \{16,17,\ldots,255\},$$

$$B_4 = \{256,\ldots,2^{16}-1\}, \quad B_5 = \{2^{16},\ldots,2^{32}-1\}, \quad \ldots, \quad B_i = \{2^{2^{i-1}},\ldots,2^{2^i}-1\}.$$

Since the largest rank is $\log n$, the number of blocks is bounded by $\log\log\log n$. The number of ranks in block B_i is $2^{2^i} - 2^{2^{i-1}} < 2^{2^i}$. Keeping everything else the same as before, a Find operation will be charged at most $\log\log\log n$ times and each node with rank in block B_i will be charged at most 2^{2^i} times (after which the rank of its parent will be in a different block). Then, the total number of node charges is:

$$\begin{aligned}
\sum_{i=1}^{\log\log\log n}\sum_{r\in B_i}\left(\frac{n}{2^r}\right)2^{2^i} &= n\sum_{i=1}^{\log\log\log n}\left[2^{2^i}\sum_{r=2^{2^{i-1}}}^{2^{2^i}-1}\left(\frac{1}{2^r}\right)\right]\\
&< n\sum_{i=1}^{\log\log\log n}\left[2^{2^i}\cdot 2^{2^i}\left(\frac{1}{2^{2^{2^{i-1}}}}\right)\right]\\
&= n\sum_{i=1}^{\log\log\log n}\left(\frac{2^{2^{i+1}}}{2^{2^{2^{i-1}}}}\right)\\
&< n\sum_{i=1}^{\log\log\log n} 1\\
&= n\log\log\log n.
\end{aligned}$$

Again we can charge $\log\log\log n$ credits to each MakeSet operation and get $O(m\log\log\log n)$ running time for m operations.

Amortized Analysis 3: $O(m \log^* n)$

Even, the $O(m \log\log\log n)$ analysis is not tight. We can make the running time even smaller by making the blocks even larger. There's good reason for us to stop at $O(m \log\log\log n)$. For example, let $n = 10^{80}$ which some physicists estimate to be the number of elementary particles in the universe. For $n = 10^{80}$, $\log\log\log n = 3.0096\ldots$, but $\log^* n = 5$. Thus, $O(m \log^* n)$ is a better bound only for *very, very* large n.

The blocks in this analysis are very large. We define block $B_i = \{b_i, \ldots, b_{i+1} - 1\}$ where the b_i are defined recursively by:

$$b_0 = 0, \quad b_{i+1} = 2^{b_i}.$$

If you unwind the recursion, you get that b_i is a stack of $i - 1$ 2's in the exponent:

$$b_i = 2^{\left.2^{2^{\cdots^{2}}}\right\}i-1}$$

As before we charge the time for path compression either to a node or to the Find operation. Since there are at most $\log^* n$ blocks, the amortized running time of Find is $O(\log^* n)$.

We estimate the total number of node charges as follows. First, recall that the number of nodes with rank equal to b_i is $\leq n/2^{b_i}$. Then, the number of nodes with rank *at least* b_i is

$$\frac{n}{2^{b_i}} + \frac{n}{2^{b_i+1}} + \frac{n}{2^{b_i+2}} + \frac{n}{2^{b_i+3}} + \cdots = \frac{n}{2^{b_i}}\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) \leq \frac{2n}{2^{b_i}}.$$

Thus, the number of *nodes* with rank in block B_i is upper bounded by $2n/2^{b_i}$ (this is an over estimate). Since each node with rank in block B_i can be charged at most 2^{b_i} times (this is a rough upper bound on $|B_i|$), we can bound the total number of node charges by:

$$\sum_{i=1}^{\log^* n} \left[\left(\frac{2n}{2^{b_i}}\right) 2^{b_i}\right] \quad = \quad 2n \sum_{i=1}^{\log^* n} 1 \quad = \quad 2n \log^* n.$$

Note that as before we sum over each block B_i. This time, however, we do not sum over the ranks $r \in B_i$. Instead, we count the number of nodes that have a rank that is in block B_i (which is bounded by $2n/2^{b_i}$ as described above) and multiply it by the number of node charges to such nodes.

Thus we have the upper bound of $O(m \log^* n)$ for m disjoint set operations. The textbook has a further improvement to $O(m\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function.