22: Union `Find`

CS 473u - Algorithms - Spring 2005

April 14, 2005

1 Union-`Find`

We want to maintain a collection of sets, under the operations of:

1. `MakeSet`(x) - create a set that contains the single element x.
2. `Find`(x) - return the set that contains x.
3. `Union`(A,B) - return the set which is the union of A and B. Namely $A \cup B$. Namely, this operation merges the two sets A and B and return the merged set.

1.1 Amortized analysis

We use a data-structure as a black-box inside an algorithm (for example `Union-Find` in Kruskal algorithm for computing minimum spanning tee). So far, when we design a data-structure we cared about worst case time for operation. Note however, that this is not necessarily the right measure. Indeed, we care about the *overall* running time spend on doing operations in the data-structure, and less about its running time for a single operation.

Formally, the *amortized running time* of an operation is the average time it takes to perform an operation on the data-structure. Formally, the amortized time of an operation is

$$\frac{\text{overall running time}}{\text{number of operations}}.$$

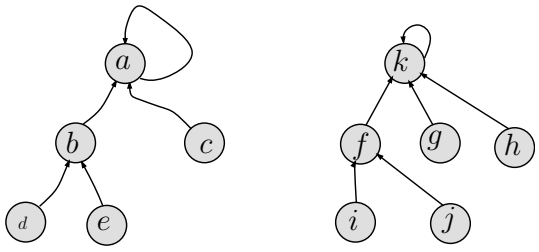


Figure 1: The `Union-Find` representation of the sets $A = \{a, b, c, d, e\}$ and $B = \{f, g, h, i, j, k\}$. The set A is uniquely identified by a pointer to the root of A, which is the node containing a.

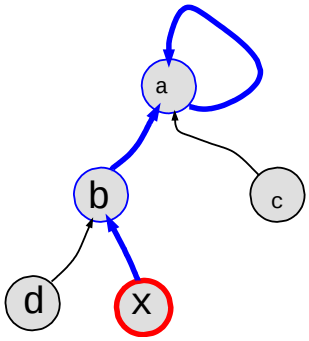
1.2 The data-structure

To implement this operations, we are going to use Reversed Trees. In a reversed tree, every element is stored in its own node. A node has one pointer to its parent. A set is uniquely identified with the element stored in the root of such a reversed tree. See Figure 1 for an example of how such a reversed tree looks like.

We implement the operations of the `Union-Find` data structure as follows:

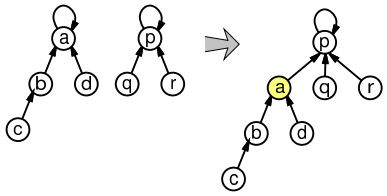
1. `MakeSet`: Create a singleton pointing to itself:

2. `Find`(x): We start from the node that contains x, and we start traversing up the tree, following the parent pointer of the current node, till we get to the root, which is just a node with its parent pointer pointing to itself. Thus, doing `Find`(x) in the following reversed:



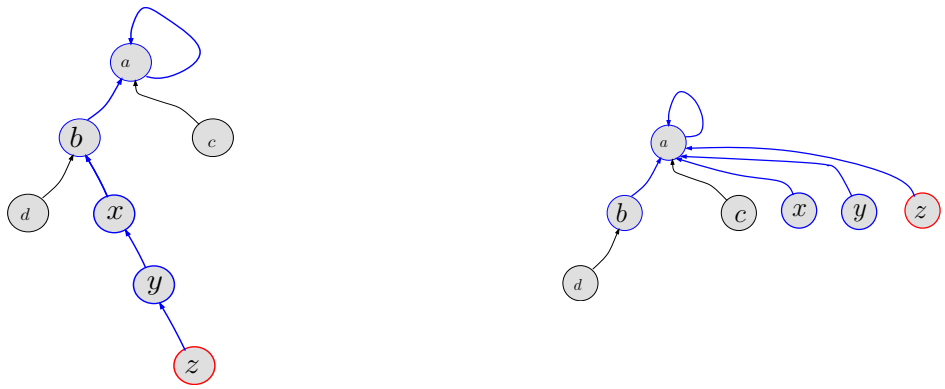
   Involve going up the tree from $x \to b \to a$, and returning a as the set.

3. `Union`(a, p): We merge two sets, by hanging the root of one tree, on the root of the other. Note, that this is a destructive operation, and the two original sets no longer exist.



Note, that in the worst case, depth of tree can be linear, so search time is $\Omega(n)$ (search time=length of path to the root). This is bad. To see why this is true, imagine creating n sets of size 1, and repeatedly merging the current set with a singleton. If we always merge (i.e., do `Union`) the current set with a singleton by hanging the current set on the singleton, the end result would be a reversed tree which looks like a linked list of length n. Doing a `Find` on the deepest element, would take linear time.

So, the question is how to further improve the performance of this data-structure. We are going to do this, by using two hacks:



(a) (b)

Figure 2: (a) The tree before performing `Find`(x), and (b) The reversed tree after performing `Find`(x) that uses path compression.

```
MakeSet(x)
    parent(x) ← x
    rank(x) ← 0
```

```
Find(x)
    if x ≠ parent(x) then
        parent(x) ← Find(parent(x))
    return parent(x)
```

```
Union(x, y )
    A ← Find(x)
    B ← Find(y)
    if rank(A)¿rank(B) then
        parent(B) ← A
    else
        parent(A) ← B
        If (rank(A) = rank(B)) then
            rank(B) ← rank(B) + 1
```

Figure 3: The pseudo-code for the `Union-Find` data-structure that uses both path-compression and union by rank.

Union by rank
Maintain for every tree, in the root, a bound on its depth (called rank). Always hang the smaller tree on the larger tree.
Path Compression
Since, anyway, we travel the path to the root during a find operation, we might as well hung all the nodes on the path directly on the root.

An example of the effects of path compression are depicted in Figure 2. For the pseudo-code of the `MakeSet`, `Union` and `Find` using path compression and union by rank, see Figure 3.

2 Analyzing the `Union`-Find Data-Structure

Definition 2.1 A node in the `Union-Find` data-structure is a *leader* if it is the root of a (reversed) tree.

Lemma 2.2 *Once a node stop being a leader (i.e., the node in top of a tree), it can never become a leader again.*

Proof: Note, that an element x can stop being a leader only because of a `Union` operation that hanged x on an element y. From this point on, the only operation that might change x parent pointer, is a `Find` operation that traverses through x. Since path-compression can only change the parent pointer of x to point to some other element y, it follows that x parent pointer will never become equal to x again. Namely, once x stop being a leader, it can never be a leader again. ■

Lemma 2.3 *Once a node stop being a leader than its rank is fixed.*

Proof: The rank of an element changes only by the `Union` operation. However, the `Union` operation changes the rank, only for elements that are leader after the operation is done. As such, if an element is no longer a leader, than its rank is fixed. ■

Lemma 2.4 *Ranks are monotonically increasing in the reversed trees, as we travel from a node to the root of the tree.*

Proof: It is enough to prove, that for every edge $u \rightarrow v$ in the data-structure, we have $\text{rank}(u) < \text{rank}(v)$. The proof is by induction. Indeed, in the beginning of time, all sets are singletons, with rank zero, and the claim trivially holds.

Next, assume that the claim holds at time t, just before we perform an operation. Clearly, if this operation is $\texttt{Union}(A, B)$, and assume that we hanged $\text{root}(A)$ on $\text{root}(B)$. In this case, it must be that $\text{rank}(\text{root}(B))$ is now larger than $\text{rank}(\text{root}(A))$, as can be easily verified. As such, if the claim held before the `Union` operation, then it is also true after it was performed.

If the operation is `Find`, and we traverse the path π, then all the nodes of π are made to point to the last node v of π. However, by induction, $\text{rank}(v)$ is larger than the rank of all the other nodes of π. In particular, all the nodes that get compressed, the rank of their new parent, is larger than their own rank. ■

Lemma 2.5 *When a node gets rank k than there are at least $\geq 2^k$ elements in its subtree.*

Proof: The proof is by induction. For $k = 0$ it is obvious. Next observe that a node gets rank k only if the merged two roots has rank $k - 1$. By induction, they have 2^{k-1} nodes (each one of them), and thus the merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes. ■

Lemma 2.6 *The number of nodes that get assigned rank k throughout the execution of the `Union-Find` data-structure is at most $n/2^k$.*

Proof: Again, by induction. For $k = 0$ it is obvious. We charge a node v of rank k to the two elements of rank $k - 1$ that were leaders that were used to create it – one of them is v having degree $k-1$, the other one is some other node u. After the merge v is of rank k and u is of rank $k - 1$ and it is no longer a leader (it can not participate in a union as a leader any more). Thus, we can charge this event to the two (no longer active) nodes of degree $k - 1$. Namely, u and v. By induction, we have $n/2^{k-1}$such nodes, and thus $\leq \left(n/2^{k-1}\right)/2 = n/2^k$ such nodes of degree k. ■

Lemma 2.7 *The time to perform a single `Find` operation when we perform union by rank and path compression is $O(\log n)$ time.*

Proof: The rank of the leader v of a reversed tree T, bounds the depth of a tree T in the `Union-Find` data-structure. By the above lemma, if we have n elements, the maximum rank is $\lg n$ and thus the depth of a tree is at most $O(\log n)$. ■

The main result we want to show, is the following.

Theorem 2.8 *If we perform a sequence of m operations over n elements, the overall running time of the* ***Union-Find*** *data-structure is $O((n+m)\log^* n)$.*

We remind the reader that $\log^*(n)$ is the number one has to take lg of a number to get a number smaller than two (there are other definitions, but they are all equivalent, up to adding a small constant). Thus, $\log^* 2 = 1$ and $\log^* 2^2 = 2$. Similarly, $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$. Similarly, $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$. Things get really exciting, when one considers

$$\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5.$$

However, $\log^*$ is a monotone increasing function. And $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$ is a huge number (considerably larger than the number of atoms in the universe). Thus, for all practical purposes, $\log^*$ returns a value which is smaller than 5. So Theorem 2.8 essentially states in the amortized sense, `Union-Find` takes constant time per operation (unless n is larger than β which is extremely unlikely).

It would be useful to look on the inverse function to $\log^*$.

Definition 2.9 $\text{Tower}(b) = 2^{Tower(b-1)}$ and $\text{Tower}(0) = 1$.

So, $\text{Tower}(i)$ is just a tower of $2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}$ of height i.

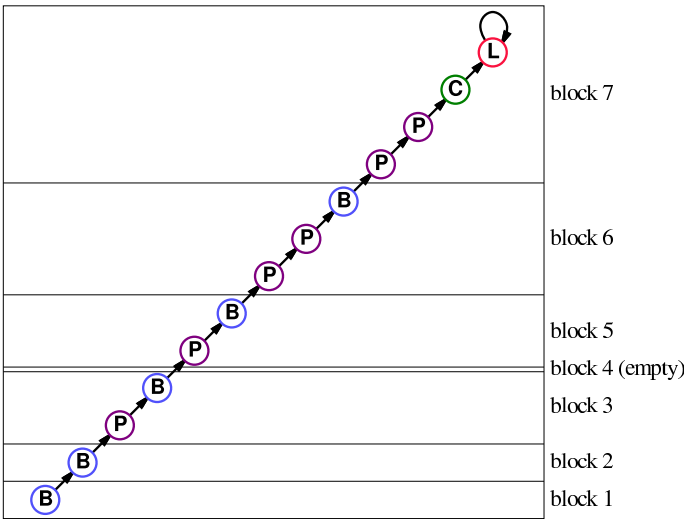
Definition 2.10 $Block(i) = [Tower(i-1)+1, Tower(i)]$
$Block(i) = \left[z, 2^{z-1}\right]$ for $z = Tower(i-1)+1$.

Observation 2.11 *The running time of* ***Find****(x) is proportional to the length of the path from x to the root of the tree that contains x. Indeed, we start from x and we visit the sequence:*
$x_1 = x$, $x_2 = parent(x) = parent(x_1)$,..., $x_i = parent(x_{i-1})$,...,$x_m = root$
Clearly, we have for this sequence: $rank(x_1) < rank(x_2) < rank(x_3) < \ldots < rank(x_m)$.
Note, that the time to perform find, is proportional to m.

Definition 2.12 A node x is in the i-th block if $rank(x) \in Block(i)$.

We are now looking for ways to pay for the find operation.

Observation 2.13 *The rank of a node v is $O(\log n)$, and the number of blocks is $O(\log^* n)$.*



Observation 2.14 *During a find operation, since the ranks of the nodes we visit are monotone increasing, once we pass through from a node v in the i-th block into a node in the $(i+1)$-th block, we can never go back to the i-th block (i.e., visit elements with rank in the i-th block).*

Lemma 2.15 *During a* **`Find`** *operation, the number of jumps between blocks is $O(\log^* n)$.*

Observation 2.16 *If x and $parent(x)$ are in the same block and we perform a find operation that passes through x. Let $r_{before} = rank(parent(x))$ before the find operation, and let r_{after} be $rank(parent(x))$ after the* **`Find`** *operation. Then because of path compression, we have $r_{after} > r_{before}$.*

Namely, when we jump inside a block, we do some work: we make the parent pointer jump forward.

Definition 2.17 A jump during a find operation inside the i-th block is called an *internal jump*.

Lemma 2.18 *At most $|Block(i)| \leq Tower(i)$ find operations can pass through an element x which is in the i-th block (i.e., $rank(x) \in Block(i)$) before $parent(x)$ is* no longer *in the i-th block.*

Lemma 2.19 *There are at most $n/Tower(i)$ nodes that have ranks in the i-th block throughout the algorithm execution.*

Proof: Clearly,

$$\sum_{i=Tower(i-1)+1}^{Tower(i)} \frac{n}{2^i} = n \cdot \sum_{i=Tower(i-1)+1}^{Tower(i)} \frac{1}{2^i} \leq \frac{n}{2^{Tower(i-1)}} = \frac{n}{Tower(i)}.$$

■

Lemma 2.20 *The number of inner block jumps performed inside the i-th block performed during the lifetime of the union-find data-structure is $O(n)$.*

Proof: An element x in the i-th block, can have $|Block(i)|$ jumps. There are $n/Tower(i)$ such elements. Thus, the total number of internal jumps is

$$|Block(i)| \cdot \frac{n}{Tower(i)} \leq Tower(i) \cdot \frac{n}{Tower(i)} = n.$$

■

We are now ready for the last step:

Lemma 2.21 *The number of internal jumps performed by the* `Union-Find` *data-structure overall is* $O(n \log^* n)$.

Proof: Every internal jump can be associated with the block it is being performed in. Every block contributes $O(n)$ internal jumps throughout the execution of the union-find data-structures. There are $O(\log^* n)$ blocks. As such there are at most $O(n \log^* n)$ internal jumps. ■

Lemma 2.22 *The overall time spent on* m `Find` *operations is* $O((m+n) \log^* n)$.

Theorem 2.8 now follows readily from the above discussion.