Chapter 31

Union Find

By Sariel Har-Peled, December 17, 2012[1] Version: 0.2

Scene: It's a fine sunny day in the forest, and a rabbit is sitting outside his burrow, tippy-tapping on his typewriter.
Along comes a fox, out for a walk.
Fox: "What are you working on?"
Rabbit: "My thesis."
Fox: "Hmmm. What's it about?"
Rabbit: "Oh, I'm writing about how rabbits eat foxes."
Fox: (incredulous pause) "That's ridiculous! Any fool knows that rabbits don't eat foxes."
Rabbit: "Sure they do, and I can prove it. Come with me."
They both disappear into the rabbit's burrow. After a few minutes, the rabbit returns, alone, to his typewriter and resumes typing.
Scene inside the rabbit's burrow: In one corner, there is a pile of fox bones. In another corner, a pile of wolf bones. On the other side of the room, a huge lion is belching and picking his teeth.
(The End)
Moral: It doesn't matter what you choose for a thesis subject.
It doesn't matter what you use for data.
What does matter is who you have for a thesis advisor.
– – Anonymous

31.1 Union-Find

31.1.1 Requirements from the data-structure

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

(i) **makeSet**(x) - creates a set that contains the single element x.

(ii) **find**(x) - returns the set that contains x.

(iii) **union**(A,B) - returns 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]This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/3.0/` or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

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

31.1.3 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 31.1 for an example of how such a reversed tree looks like.

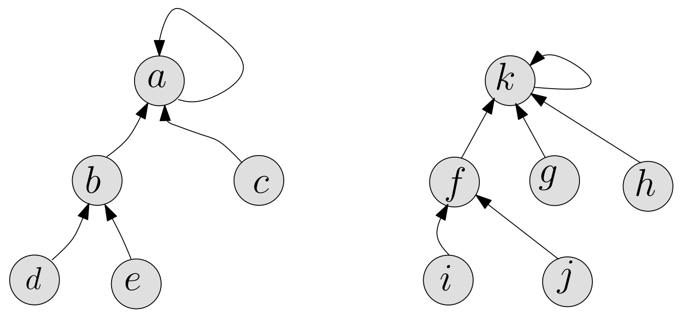
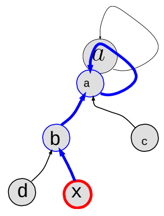


Figure 31.1: The Union-Find representation of the sets $A = \{a, b, c, d, e\}$ and $B = \{f, g, h, i, j, k\}$. The set A is uniquely identified by a pointer to the root of A, which is the node containing a.

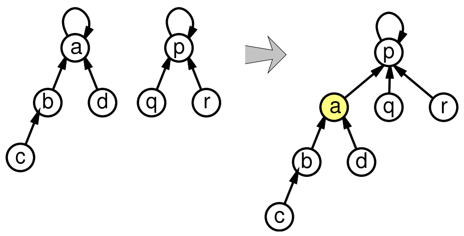
We implement the operations of the Union-Find data structure as follows:

(A) **makeSet**: Create a singleton pointing to itself:

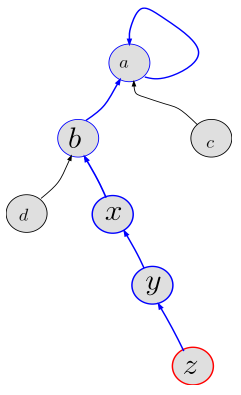
(B) **find**(x): We start from the node that contains x, and we start traversing up the tree, following the parent pointer of the current node, till we get to the root, which is just a node with its parent pointer pointing to itself.
Thus, doing a **find**(x) operation in the reversed tree shown on the right, involve going up the tree from $x \to b \to a$, and returning a as the set.



(C) **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. Example of how the new tree representing the new set is depicted on the right.



Note, that in the worst case, depth of tree can be linear in n (the number of elements stored in the tree), so the **find** operation might require $\Omega(n)$ time. To see that this worst case is realizable perform the following sequence of operations: create n sets of size 1, and



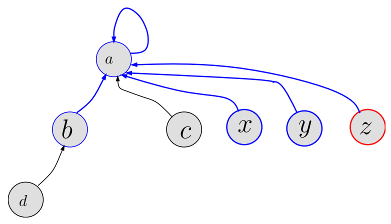


Figure 31.2: (a) The tree before performing **find**(z), and (b) The reversed tree after performing **find**(z) that uses path compression.

```
makeSet(x)
    $\overline{p}(x) \leftarrow x$
    $\mathrm{rank}(x) \leftarrow 0$
```

```
find(x)
    if $x \neq \overline{p}(x)$ then
        $\overline{p}(x) \leftarrow$ find$(\overline{p}(x))$
    return $\overline{p}(x)$
```

```
union($x$, $y$ )
    $A \leftarrow$ find$(x)$
    $B \leftarrow$ find$(y)$
    if $\mathrm{rank}(A) > \mathrm{rank}(B)$ then
        $\overline{p}(B) \leftarrow A$
    else
        $\overline{p}(A) \leftarrow B$
        if $\mathrm{rank}(A) = \mathrm{rank}(B)$ then
            $\mathrm{rank}(B) \leftarrow \mathrm{rank}(B) + 1$
```

Figure 31.3: The pseudo-code for the Union-Find data-structure that uses both path-compression and union by rank. For element x, we denote the parent pointer of x by $\overline{p}(x)$.

repeatedly merge 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 will 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”:

(i) ***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.

(ii) ***Path compression***: Since, anyway, we travel the path to the root during a **find** operation, we might as well hang all the nodes on the path directly on the root.

An example of the effects of path compression are depicted in Figure 31.2. For the pseudo-code of the **makeSet**, **union** and **find** using path compression and union by rank, see Figure 31.3.

We maintain a rank which is associated with each element in the data-structure. When a singleton is being created, its associated rank is set to zero. Whenever two sets are being merged, we update the rank of the new root of the merged trees. If the two trees have different root ranks, then the rank of the root does not change. If they are equal then we set the rank of the new root to be larger by one.

31.2 Analyzing the Union-Find Data-Structure

Definition 31.2.1. A node in the union-find data-structure is a ***leader*** if it is the root of a (reversed) tree.

Lemma 31.2.2. *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 31.2.3. *Once a node stop being a leader then 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 31.2.4. *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 $u \to v$ 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 **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 31.2.5. *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 since a singleton has a rank zero, and a single element in the set. 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 31.2.6. *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 u and v of rank $k-1$ that were leaders that were used to create the new larger set. 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 that the algorithm created at most $n/2^{k-1}$ nodes of rank $k-1$, and thus the number of nodes of rank k created by the algorithm is at most $\leq \left(n/2^{k-1}\right)/2 = n/2^k$. ■

Lemma 31.2.7. *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 $O(\log n)$. ■

Surprisingly, we can do much better.

Theorem 31.2.8. *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. Intuitively, Theorem 31.2.8 states (in the amortized sense), that the Union-Find data-structure takes constant time per operation (unless n is larger than β which is unlikely).

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

Definition 31.2.9. Let $\mathrm{Tower}(b) = 2^{\mathrm{Tower}(b-1)}$ and $\mathrm{Tower}(0) = 1$.

So, $\mathrm{Tower}(i)$ is just a tower of $2^{2^{2^{\cdot^{\cdot^{2}}}}}$ of height i. Observe that $\log^*(\mathrm{Tower}(i)) = i$.

Definition 31.2.10. For $i \geq 0$, let $\mathrm{Block}(i) = [\mathrm{Tower}(i-1)+1, \mathrm{Tower}(i)]$; that is

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

For technical reasons, we define $\mathrm{Block}(0) = [0, 1]$. As such,

$$\begin{aligned}
\mathrm{Block}(0) &= \left[0, 1\right] \\
\mathrm{Block}(1) &= \left[2, 2\right] \\
\mathrm{Block}(2) &= \left[3, 4\right] \\
\mathrm{Block}(3) &= \left[5, 16\right] \\
\mathrm{Block}(4) &= \left[17, 65536\right] \\
\mathrm{Block}(5) &= \left[65537, 2^{65536}\right] \\
&\vdots
\end{aligned}$$

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:

$$x_1 = x, x_2 = \overline{\mathrm{p}}(x) = \overline{\mathrm{p}}(x_1), ..., x_i = \overline{\mathrm{p}}(x_{i-1}), \ldots, x_m = \text{root of tree}.$$

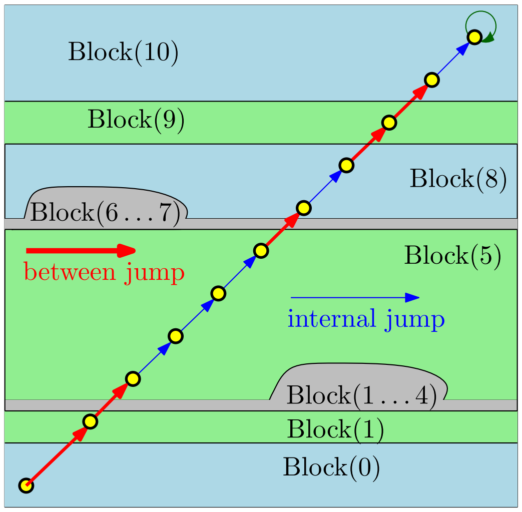
Clearly, we have for this sequence: $\mathrm{rank}(x_1) < \mathrm{rank}(x_2) < \mathrm{rank}(x_3) < \ldots < \mathrm{rank}(x_m)$, and the time it takes to perform **find**(x) is proportional to m, the length of the path from x to the root of the tree containing x.

Definition 31.2.11. A node x is in the ith block if $\mathrm{rank}(x) \in \mathrm{Block}(i)$.

We are now looking for ways to pay for the **find** operation, since the other two operations take constant time.

Observe, that the maximum rank of a node v is $O(\log n)$, and the number of blocks is $O(\log^* n)$, since $O(\log n)$ is in the block $\mathrm{Block}(c \log^* n)$, for c a constant sufficiently large.

In particular, consider a **find** (x) operation, and let π be the path visited. Next, consider the ranks of the elements of π, and imagine partitioning π into which blocks each element rank belongs to. An example of such a path is depicted on the right. The price of the **find** operation is the length of π.



Formally, for a node x, $\nu = \mathrm{index}_{\mathrm{B}}(x)$ is the index of the block that contains $\mathrm{rank}(x)$. Namely, $\mathrm{rank}(x) \in \mathrm{Block}\big(\mathrm{index}_{\mathrm{B}}(x)\big)$. As such, $\mathrm{index}_{\mathrm{B}}(x)$ is the ***block of*** x.

Now, 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 ith block into a node in the $(i+1)$th block, we can never go back to the ith block (i.e., visit elements with rank in the ith block). As such, we can charge the visit to nodes in π that are next to a element in a different block, to the number of blocks (which is $O(\log^* n)$).

Definition 31.2.12. Consider a path π traversed by a **find** operation. Along the path π, an element x, such that $\overline{\mathrm{p}}(x)$ is in a different block, is a ***jump between blocks***.

On the other hand, a jump during a **find** operation inside a block is called an ***internal jump***; that is, x and $\overline{\mathrm{p}}(x)$ are in the same block.

Lemma 31.2.13. *During a single* **find**(x) *operation, the number of jumps between blocks along the search path is* $O(\log^* n)$.

Proof: Consider the search path $\pi = x_1, \ldots, x_m$, and consider the list of numbers $0 \leq \mathrm{index}_{\mathrm{B}}(x_1) \leq \mathrm{index}_{\mathrm{B}}(x_2) \leq \ldots \leq \mathrm{index}_{\mathrm{B}}(x_m)$. We have that $\mathrm{index}_{\mathrm{B}}(x_m) = O(\log^* n)$. As such, the number of elements x in π such that $\mathrm{index}_{\mathrm{B}}(x) \neq \mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x))$ is at most $O(\log^* n)$. ■

Consider the case that x and $\overline{\mathrm{p}}(x)$ are both the same block (i.e., $\mathrm{index}_{\mathrm{B}}(x) = \mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x))$ and we perform a find operation that passes through x. Let $r_{\mathrm{bef}} = \mathrm{rank}(\overline{\mathrm{p}}(x))$ before the **find** operation, and let r_{aft} be $\mathrm{rank}(\overline{\mathrm{p}}(x))$ after the **find** operation. Observe, that because of path compression, we have $r_{\mathrm{aft}} > r_{\mathrm{bef}}$. Namely, when we jump inside a block, we do some work: we make the parent pointer of x jump forward and the new parent has higher rank. We will charge such internal block jumps to this "progress".

Lemma 31.2.14. *At most* $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$ **find** *operations can pass through an element x, which is in the ith block (i.e.,* $\mathrm{index}_{\mathrm{B}}(x) = i$*) before* $\overline{\mathrm{p}}(x)$ *is* no longer *in the ith block. That is* $\mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x)) > i$.

Proof: Indeed, by the above discussion, the parent of x increases its rank every-time an internal jump goes through x. Since there at most $|\text{Block}(i)|$ different values in the ith block, the claim follows. The inequality $|\text{Block}(i)| \leq \text{Tower}(i)$ holds by definition, see Definition 31.2.10. ■

Lemma 31.2.15. *There are at most $n/\text{Tower}(i)$ nodes that have ranks in the ith block throughout the algorithm execution.*

Proof: By Lemma 31.2.6, we have that the number of elements with rank in the ith block is at most

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

■

Lemma 31.2.16. *The number of internal jumps performed, inside the ith block, during the lifetime of the union-find data-structure is $O(n)$.*

Proof: An element x in the ith block, can have at most $|\text{Block}(i)|$ internal jumps, before all jumps through x are jumps between blocks, by Lemma 31.2.14. There are at most $n/\text{Tower}(i)$ elements with ranks in the ith block, throughout the algorithm execution, by Lemma 31.2.15. Thus, the total number of internal jumps is

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

■

We are now ready for the last step.

Lemma 31.2.17. *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, by Lemma 31.2.16. There are $O(\log^* n)$ blocks. As such there are at most $O(n \log^* n)$ internal jumps. ■

Lemma 31.2.18. *The overall time spent on m* **find** *operations, throughout the lifetime of a union-find data-structure defined over n elements, is $O((m+n)\log^* n)$.*

Theorem 31.2.8 now follows readily from the above discussion.