Union-Find

Caio De Naday Hornhardt

May 2022

1 Introduction

At an early age, we learn about addition of numbers. Multiplication, which is iterated addition, comes just after. And multiplication is so ubiquitous that we cannot imagine mathematics without it.

A bit older, we learn about iterated multiplication. Powers of numbers are slightly less frequent than multiplication, but still very frequent. They appear in the modeling of a wide variety of phenomena.

But we never hear about iterated power of numbers. Why is that? One possible answer is that our world is too small for it. The exponential is already a very rapidly increasing function, if we iterate it we get something that is simply too fast. Let's iterate 2^x, for example: $2^1 = 2$, $2^2 = 4$, $2^4 = 16$, $2^{16} = 65536$, $2^{65536} = \ldots$, well, there are less than 2^{300} atoms in the known universe, so 2^{65536} is a really huge number. If there was a phenomenon that would follow such function, there is no way it would fit in our universe.

Of course, this is assuming we are looking for phenomena in the physical world. In the realm of abstraction, such a thing may appear. "If you make some irrelevant theory," the reader may think. However, it turns out that this crazy function is hidden in an innocent-looking algorithm that solves an innocent-looking problem. A problem that does arises in real-world applications.

2 The Problem

The Union-Find problem is the following: we have a set and we are given a list of pairs of elements of this set. The elements of each pair are then declared equivalent. The goal is to have an algorithm that tells us if any two elements are equivalent. For that we keep updating the equivalence classes: for each element of a pair we *find* the corresponding equivalence classes and then we perform a *union* operation.

3 The operations

The version of the algorithm we are going to see is the following: we will represent each equivalence class (or set, for simplicity) by a tree. All our classes, then,

form a forest. The root of each tree will be the chosen representative for the class. Each element will be given an integer called rank, which is related to its position in the tree it is.

The algorithm is then given by a sequence of the following routines:

We chose here to use the *rank* version of Union-Find:

```
function MakeSet(x) is
    if x is not already in the forest then
        x.parent := x
        x.rank := 0
    end if
end function

function Find(x) is
    if x.parent = x then
        return x
    else
        root := Find(x.parent)
        x.parent := root // Path Compression
        return root
    end if
end function

function Union(x, y) is
    root_x := Find(x)
    root_y := Find(y)

    if root_x = root_y then
        return
    end if

    if root_x.rank >= root_y.rank then
        big   := root_x
        small := root_y
    else
        big   := root_y
        small := root_x
    end if

    if big.rank = small.rank then
        big.rank := big.rank + 1
    end if
end function
```

4 Historical Note

The $\mathcal{O}(m \log^\star n)$ time-bound was discovered by J. E. Hopcroft and J. D. Ullman in [HU73]. In this paper, they say that they were actually trying to prove a $\mathcal{O}(m)$ time-bound, but M. J. Fischer found an error in their proof. R. E. Tarjan then showed in ?? that their goal was impossible.

On a different front, but also in [HU73], Hopcraft and Ullman present another algorithm to solve the same problem. It was an adaptation of an algorithm presented in [SL69], by R. E. Stearns and D. J. Lewis, that solved the problem in $\mathcal{O}(m \underbrace{\log \log \ldots \log}_{k} n)$ time, where k was an extra parameter that could be arbitrarily set to any natural number. Hopcraft and Ullman managed to eliminate this extra parameter and proved that the resulting algorithm had $\Theta(m \log^\star n)$ time. This could very well be the inspiration for guessing the $\mathcal{O}(m \log^\star n)$ time bound for the `Union-Find`.

5 Proof of the $\mathcal{O}(m \log^\star n)$ time bound

We consider a list of m operations `MakeSet`, `Find` and `Union`, n of which are `MakeSet`. Our goal is to find an upper bound for the execution time of these m operations.

Note that each `MakeSet` operation take constant time, and each `Union` operation takes the time of the two `Find`'s it runs plus a constant time. Hence we only have to know how long it takes to run all `Find`'s (including those inside the `Union`'s). And the time to run each `Find` is proportional to the number of edges child-parent it follows.

Lemma 1. *During any moment during the execution, let v be a vertex on a tree whose root has rank k. Any new ancestor to v will have rank strictly greater than k.*

Proof. Let u be the root of the tree where v is. If the next operation is a `MakeSet` or a `Find`, v continues to be in a tree whose root is u, v has no new ancestor (even if its parent was changed) and the rank of u continue to be k. If the next operation is a `Union` that keeps u as a root, v still has no new ancestor, but maybe the rank of u has increased. If the next operation is a `Union` that makes u no longer a root, then v gets only one new ancestor, namely the new parent of u, which has rank strictly greater then k.

We have proved so far that the result is valid if only one operation is made. Since the rank of root of the tree where v is after this operation is greater or equal to k in all cases, the result will still hold after any number of operations. □

Corollary 2. *An ancestor of a non-root vertex v has always a rank strictly greater than the one of v.*

Proof. The vertex v was a root at a certain moment. Let k be the rank of v right before the operation where it ceases to a root. From then on, the rank of v

will continue to be k, since no operation modifies the rank of a non-root. By Lemma 1, every ancestor of v will have rank strictly grater then k. □

Lemma 3. *If a root has rank k, then the corresponding tree has at least 2^k elements.*

Proof. By induction. The case $k = 0$ is obvious. The only moment when the rank can change is when we link two roots with same rank k by a `Union` operation. In this case, the root of the new tree has rank $k + 1$ and, by the induction hypothesis, the new tree has at least $2^k + 2^k = 2^{k+1}$ elements. Since the amount of descendants of a root never decreases, we have our result. □

Corollary 4. *All the ranks are at most $\log n$.*

Proof. Let k be the rank of any vertex. By Lemma 3, we must have $2^k \leq n$ and, hence, $k \leq \log n$. □

Note that the results (and proofs!) we have so far do not depend on the Path Compression strategy. From them we get:

Proposition 5. *The execution time of the m operations, using or not the Path Compression strategy, is $\mathcal{O}(m \log n)$.*

Proof. By Corollary 2, each `Find` operation follows at most as many child-parent links as the rank of the root it finds, and, by Corollary 4, this rank is at most $\log n$. Since each one of the m operations runs 0, 1 or 2 `Find`'s plus something of constant time, each operation has a time-bound of $\mathcal{O}(\log n)$. □

We encourage the reader who is interested to prove that, if we do not use Path Compression, then execution takes time $\Theta(m \log n)$.

Following lemmas sound more technical. They still do not depend on the Path Compression strategy, but it forces us to be more careful in stating and proving them, since a non-root vertex can lose vertices.

Lemma 6. *Every element of rank k has or had, at a previous moment when it already had rank k, at least 2^k descendants.*

Proof. Follows directly from Lemma 3, since only roots can have their ranks changed. □

Lemma 7. *Let v be a vertex and k be an integer. During the execution of all m operations, v can be a descendant of at most one vertex with rank k.*

Proof. Consider the moment just after v becomes descendant of o vertex u with rank k for the first time. Since only links to roots are created and only roots can have their rank changed, at this moment u is a root. By Lemma 1, all new ancestors to v will have rank strictly greater than k, concluding the proof. □

Proposition 8. *At any moment during the execution, we have at most $\frac{n}{2^k}$ vertices with rank k.*

Proof. Let $v_1, \ldots, v_\ell$ be the elements with rank k at a given moment. Let D_i be the set of elements that are or were previously descendants of v_i at a moment when it already had rank k. By Lemma 6, each D_i has at least 2^k elements and, by Lemma 7, these sets are pairwise disjoint. Since we have at most n vertices in total, we must have $2^k \cdot \ell \leq n$ elements and, hence, $\ell \leq \frac{n}{2^k}$. □

Having established the results above, we are in position for the time complexity analysis itself. For that, we are going to separate our vertices in a clever way.

At any moment of the execution, let B_k as the set of non-root vertices with rank in the interval $[k, 2^k)$. Note that once a vertex enters B_k, the vertex will never get out of there, since non-roots cannot have their ranks changed.

Lemma 9. *For each k, the amount of vertices in the set B_k is less then $\frac{n}{2^{k-1}}$.*

Proof. By Proposition 8, the number of vertices in B_k less or equal to

$$\begin{aligned}\frac{n}{2^k} + \frac{n}{2^{k+1}} + \cdots + \frac{n}{2^{2^k-1}} &< \frac{n}{2^k} + \frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \\ &= \frac{n}{2^k} \cdot (1 + \frac{1}{2} + \frac{1}{4} + \cdots) \\ &= \frac{n}{2^k} \cdot 2 = \frac{n}{2^{k-1}}.\end{aligned}$$

□

The following result is the first where the Path Compression strategy is used:

Lemma 10. *The number of times the m operations traverse links of the form $u \to v$, where v is the parent of u and both are in B_k, is less than $2n$.*

Proof. Fix $u \in B_k$. Every time a `Find` follows a child-parent link $u \to v$, it changes the parent of u to some ancestor v' of v. By Corollary 2, v' has a rank strictly greater than the one of v. Since the ranks of elements of B_k are in $[k, 2^k)$, the `Find`'s can follow at most $2^k - k - 1 < 2^k$ child-parent links starting at u before the parent of u is set to some vertex out of B_k. Since $u \in B_k$ was arbitrary and, by Lemma 9, B_k has less than $\frac{n}{2^{k-1}}$ elements, the amount of links as in the statement traversed by the `Find`'s is less than $\frac{n}{2^{k-1}} \cdot 2^k = 2n$. □

By the definition of B_k, every non-root vertex will fall in one of B_0, B_1, B_2, B_4, B_{16}, $B_{2^{16}}$, $B_{2^{2^{16}}}$ and so on (since $2^{16} = 65536$, it will be quite rare to need this last one or any other...). We will refer to these specific B_k's as *buckets*.

Lemma 11. *At most $\log^\star n$ of the buckets are non-empty.*

Proof. Let k denote the maximum rank achieved. By the definition of $\log^\star$, we have that B_0 together with next $\log^\star k$ buckets cover all the non-root vertices. By Corollary 4, $k \leq \log n$, so we have at most $1 + \log^\star(\log n) = \log^\star n$ non-empty buckets. □

Theorem 12. *The execution time of the m operations, with Path Compression, is $\mathcal{O}(m \log^\star n)$.*

Proof. We will count the number of child-parent links $u \to v$ that are followed by all the operations. To this end, we will consider separately the case where v is a root (i), the case where u and v lie in different buckets (ii) and the case where u and v lie in the same bucket (iii).

(i) Each `Find` gets to a root only once, so for the m operations we follow at most $2m$ of this sort.

(ii) By Corollary 2, each `Find` follows a sequence of vertices with increasing ranks, so we can start at most once in each bucket, and we can never start from the last. Hence, by Lemma 11, it can follow at most $\log^\star(n) - 1$. Therefore the m operations follow at most $2m \log^\star(n) - 2m$ of this sort.

(iii) Combining Lemmas 10 and 11, we have that we follow $2n \log^\star n$ of this sort. Note that, instead of considering each `Find` separately as in the previous cases, we are considering them in bulk.

Adding those numbers, we get that less than $2(m+n)\log^\star(n)$ child-parent links are traversed. Since $n \leq m$, the proof is complete. □

References

[HU73] J. E. Hopcroft and J. D. Ullman, *Set merging algorithms*, SIAM J. Comput. **2** (1973), 294–303. MR 329310

[SL69] R. E. Stearns and P. M. Lewis, *Property grammars and table machines*, Information and Control **14** (1969), 524–549. MR 249229