THE UNION-FIND STRUCTURE

Yanheng Wang

March 31, 2024

We are interested in maintaining a partition of $[n]$. Initially every element forms a class itself: $\{\{1\},\{2\},\dots,\{n\}\}$. Then a user may call union(i, j) to merge the class containing i and the class containing j. For example:

$$\begin{aligned}\{\{1\},\{2\},\{3\},\{4\}\} &\xrightarrow{\text{union}(2,3)} \{\{1\},\{2,3\},\{4\}\}\\ &\xrightarrow{\text{union}(1,4)} \{\{1,4\},\{2,3\}\}\\ &\xrightarrow{\text{union}(2,4)} \{\{1,2,3,4\}\}\end{aligned}$$

At any time, each class has a distinct identifier. Via find(i) the user may access the identifier of the class containing i. In particular, he could figure out whether two elements i, j live in the same class by comparing find(i) with find(j).

There is an elegant solution to the task, so compelling that it gets named *the* union-find structure. It handles union and find operations in amortised $O(\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function. Since $\alpha(10^{35000}) \leqslant 5$, we may treat $\alpha(n)$ essentially as a constant.

This structure stores each class in a separate tree. The root of the tree serves as the identifier. Merging two classes amounts to linking two trees together, and retrieving the identifier amounts to finding the root. A preliminary version is described below:

Algorithm 1

```
fn initialise()
    for $i = 1, \dots, n$ do
        parent$[i] := \bot$

fn find($i$)
    $x := i$
    while parent$[x] \neq \bot$ do
        $x :=$ parent$[x]$
    return $x$
```

```
fn union($i, j$)
    $x :=$ find($i$)
    $y :=$ find($j$)
    parent[$x$] $:= y$    {link tree x under tree y}
```

Unfortunately, this version is susceptible to adversarial operation sequence. Consider union$(i, i+1)$ for $i = 1, \ldots, n-1$ in order. The resulting tree is a path of length n, which makes find operations highly inefficient. To patch the issue, we introduce a *rank* value for each node. At least for now, the rank reflects the height of the subtree. Upon unions, we always link the root of smaller rank under the one of higher rank. This way the resulting tree is balanced.

Algorithm 2

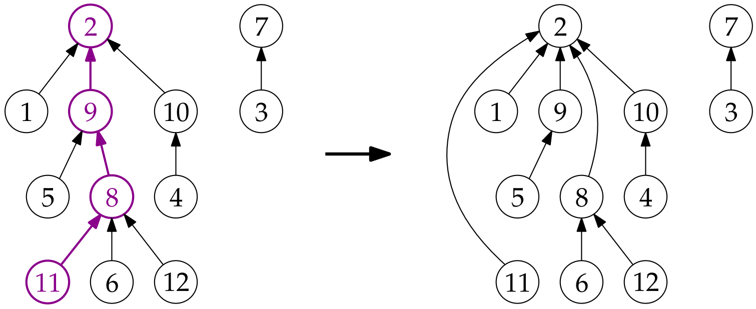
```
fn initialise()
    for $i = 1, \ldots, n$ do
        parent[$i$] $:= \perp$
        rank[$i$] $:= 0$
fn find($i$)
    $x := i$
    while parent[$x$] $\neq \perp$ do
        $x :=$ parent[$x$]
    return $x$
fn union($i, j$)
    $x :=$ find($i$)
    $y :=$ find($j$)
    if rank[$x$] $<$ rank[$y$] then
        parent[$x$] $:= y$
    else
        parent[$y$] $:= x$
        rank[$x$] $:= \max\{$rank[$x$], rank[$y$] $+ 1\}$
```

Note that every node x has at least $2^{\text{rank}[x]}$ descendants (including itself). Indeed, after initialisation every node has $1 \geqslant 2^0$ descendant. Throughout the time the number of descendents can only increase. But rank$[x]$ increases only at unions; in particular, only when rank$[y] =$

rank$[x]$. In that case, x receives the entire tree y as new descendants, thus counting $\geqslant 2^{\text{rank}[x]} + 2^{\text{rank}[y]} = 2^{\text{rank}'[x]}$ in total by induction.

It follows that the maximum rank (hence the height) is at most $\log n$. So the cost of each operation is $O(\log n)$. This is already strikingly good given the simplicity of the algorithm. But with one more trick we could continue pushing its limit.

Currently, the find operation is the bottleneck. If one keeps calling find(i) for some deep node i, then we will repeatedly spend time traversing the path from i to the root. This redundant traversal can be eliminated by *path compression*: As we go through the path, we re-link every encountered node directly to the root. So next time if one calls find$()$ on any of these nodes, we can reach the root in one hop.



The final algorithm is given below. We had isolated two "atomic" operations compress(i,x) and link(x,y) that are useful in the analysis.

Algorithm 3

```
fn initialise()
    for $i = 1, \ldots, n$ do
        parent$[i] := \bot$
        rank$[i] := 0$
fn find$(i)$
    $x := i$
    while parent$[x] \neq \bot$ do
        $x :=$ parent$[x]$
    compress$(i, x)$
    return $x$
fn union$(i, j)$
    link(find$(i)$, find$(j)$)
```

```
fn compress($i$, $x$)
    {assume $i$ is a descendant of root $x$}
    while $i \neq x$ do
        parent[$i$] := $x$
        $i$ := parent[$i$]

fn link($x$, $y$)
    {assume $x$,$y$ are roots}
    if rank[$x$] < rank[$y$] then
        parent[$x$] := $y$
    else
        parent[$y$] := $x$
        rank[$x$] := max{rank[$x$], rank[$y$] + 1}
```

We remark that path compression does not update ranks. Hence, a rank value reflects the exact tree height no more, but instead an upper bound on height.

We aim at the following claim: For any interleaving sequence of m unions and m' finds, the total cost is $O((m+m')\,\alpha(n))$. So each operation runs in amortised $O(\alpha(n))$ time.

To untangle the analysis, let us replace every union(i, j) with three operations $x :=$ find(i), $y :=$ find(j) and link(x, y). Then, we further replace every find(i) with compress(i, x) where x is the root of node i at the time of calling. Now we obtain an interleaving sequence of $2m + m'$ compressions and m linkage. Apparently it is semantically equivalent to the original sequence, and their costs are of the same order (the only difference being the while-loop in the find operation, whose cost can be charged to compression anyway.)

The manoeuvre is helpful because linkage is *oblivious* to compressions. Indeed, a compression does not alter roots or ranks, so the execution of linkage is totally unaffected. Hence we may postpone all compressions after linkage, yet the cost remains the same. From now on, we study a sequence σ of m linkage *followed* by $2m+m'$ compressions.

Ranks Lemma. The following properties hold throughout σ.

(i) Ranks are strictly increasing along any leaf-to-root path.

(ii) For all $r \geqslant 0$, there are at most $n/2^r$ nodes of rank r.

Proof. Linkage can only increase the rank of a root, which shall never break monotonicity. Compression always points nodes to ancestors which has higher rank by induction, so monotonicity is preserved. These establish (i).

For property (ii), it suffices to consider the linking phase because compressions do not meddle with ranks. Suppose there are N nodes of rank r. Throughout linking phase, every such node has at least 2^r descendants. Moreover, no such node is an ancestor of another due to (i). So they have disjoint subtrees which cover at least $N \cdot 2^r$ nodes in total. Consequently $N \leqslant n/2^r$. □

Exercise. Show a variant of property (ii): For all $r \geqslant 1$, there are at most $m/(2^r - 1)$ nodes of rank r.

Let r_x be the rank of node x after the linking phase. (It equals the height of tree x at the end of the linking phase, but the height may decrease during the compression phase.) To analyse the cost of σ, we define two sets

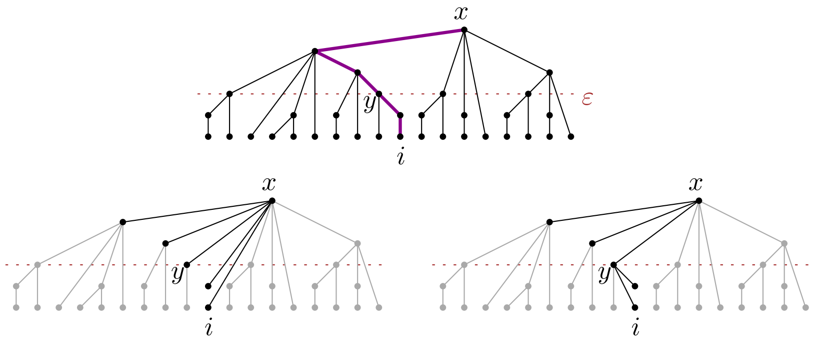
$$V_{\text{low}} := \{x \in [n] : r_x \leqslant \varepsilon\} \text{ and } V_{\text{high}} := \{x \in [n] : r_x \geqslant \varepsilon\}$$

where ε is a small number to be specified later. Then we separate three types of compressions:

1. those involving only V_{low};
2. those involving only V_{high};
3. and those crossing from V_{low} to V_{high}.

The general idea is as follows. Type-1 compressions are cheap because ε is small. Type-2 compressions are expensive in the beginning. But as $|V_{\text{high}}|$ is relatively small, all such nodes will soon be compressed to the vicinity of the root, and later compressions become cheap. Type-3 is a hybrid of the other two and can be treated likewise.

Specifically, let us consider every type-3 compress(i, x) in order. We replace it with a type-1 compress(i, y) plus a type-2 compress(y, x), where y is the unique node on path $i \rightsquigarrow x$ such that $r_y = \varepsilon$. Although the replacement produces a different tree, the cost does not change; moreover, it does not reduce the cost of future compressions. Therefore, after we have replaced all type-3 compressions, the cost of the resulting sequence is an upper bound on the original cost.



applying type-3 compress(i, x) replacing with types 1 and 2

It remains to study a sequence of m_1 type-1 and m_2 type-2 compressions, applied on an n-node forest of height h. Let $T(m_1+m_2, n, h)$ be the worst-case cost. By independence of the two types,

$$T(m_1+m_2,n,h) \;=\; T(m_1,|V_{\text{low}}|,\varepsilon) \;+\; T(m_2,|V_{\text{high}}|,h-\varepsilon) \qquad (✠)$$

We bound the first term trivially by $T(m_1, n, \varepsilon)$. What about the second term? The cost of each compression is one plus the number of parent changes. So the combined cost is m_2 plus the total number of parent changes in V_{high}. A node can change its parent up to $h-\varepsilon$ times by property (i). And with $\varepsilon := \log h$, the number of nodes is

$$|V_{\text{high}}| \leqslant \sum_{r=\varepsilon}^{h} \frac{n}{2^r} < \sum_{r=\varepsilon}^{\infty} \frac{n}{2^r} = \frac{n}{2^{\varepsilon-1}} = \frac{2n}{h}$$

by property (ii). Hence $T(m_2,|V_{\text{high}}|,h-\varepsilon) \leqslant m_2 + \frac{2n}{h}(h-\varepsilon) < m_2 + 2n$.

Summarising, we have

$$T(m_1+m_2,n,h) \;\leqslant\; T(m_1,n,\varepsilon) + m_2 + 2n.$$

Subtracting m_1+m_2 from both sides, we get

$$T(m_1+m_2,n,h) - (m_1+m_2) \leqslant (T(m_1,n,\varepsilon) - m_1) + 2n.$$

Recall $\varepsilon = \log h$, so the height parameter becomes $O(1)$ after $\log^\star h$ repeated applications. It implies

$$T(m_1+m_2,n,h) \leqslant m_1+m_2+2n\log^\star h, \qquad (♠)$$

so each operation costs $O(\log^\star h)$ amortised time. (If $m < n-1$ then we analyse each tree in the forest independently. Thus we can assume $m \geqslant n-1$ and $m_1 + m_2 = \Omega(n)$.)

We should not rejoice yet: the bound is weaker than our promise. But with this new insight we may refine the argument. In particular, we can now bound (✠) via (♠):

$$T(m_1+m_2,n,h) \leqslant T(m_1,n,\varepsilon) \;+\; m_2 + 2|V_{\text{high}}| \cdot \log^\star h.$$

Choosing threshold $\varepsilon := 1 + \log(\log^\star h) \leqslant \log^\star h$, we have

$$|V_{\text{high}}| < \frac{2n}{2^{\varepsilon-1}} = \frac{2n}{\log^\star h},$$

which implies

$$T(m_1+m_2,n,h) \leqslant T(m_1,n,\varepsilon) + m_2 + 2n.$$

The recursion has the same form as before, but the height decreases at a much faster rate. Easily we can solve

$$T(m_1+m_2,n,h) \leqslant m_1 + m_2 + 2n \log^{\star\star} h.$$

This strengthens (♠) with an additional star operator. We can continue this line of refinements and bound the amortised running time by $O(\log^{\star\cdots\star} n)$ for *any* number of star operators. Such function turns out to be asymptotically less than the inverse Ackermann function.