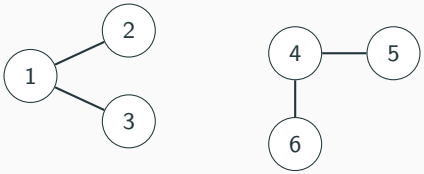
Union-Find

Disjoint Sets with Union by Rank and Path Compression

Motivation: Connectivity as Edges Arrive

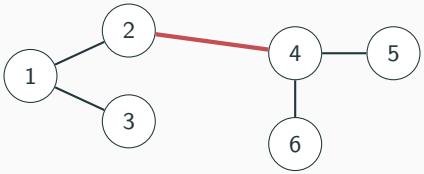
Edges of an undirected graph arrive one at a time. At any moment we want to answer: are u and v in the same connected component?



- Is 3 connected to 5? No.
- Rerunning DFS after each edge costs $O(n + m)$ per query. Too slow.

Motivation: Connectivity as Edges Arrive

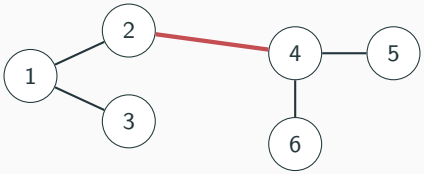
Edges of an undirected graph arrive one at a time. At any moment we want to answer: are u and v in the same connected component?



- Is 3 connected to 5? After adding the edge $\{2, 4\}$: yes.
- Rerunning DFS after each edge costs $O(n + m)$ per query. Too slow.

Motivation: Connectivity as Edges Arrive

Edges of an undirected graph arrive one at a time. At any moment we want to answer: are u and v in the same connected component?



- Is 3 connected to 5? After adding the edge $\{2, 4\}$: yes.
- Rerunning DFS after each edge costs $O(n + m)$ per query. Too slow.

Used in Kruskal's minimum spanning tree algorithm, image segmentation, network connectivity, equivalence of variables in compilers, ...

The Disjoint-Set Interface

Maintain a partition of $\{1, \ldots, n\}$ into disjoint sets. Each set has one representative.

- `makeset(x)`: create the new set $\{x\}$.
- `find(x)`: return the representative of the set containing x.
- `union(x, y)`: merge the sets containing x and y.

x and y are in the same set $\iff$ `find(x) == find(y)`.



Shaded vertices are the representatives.

First Attempt: Store the Label

Keep an array `label[x]` = representative of x.

x	1	2	3	4	5	6
`label[`x`]`	1	1	1	4	4	6

- `makeset(x)`: set `label[x] = x`. $O(1)$
- `find(x)`: return `label[x]`. $O(1)$
- `union(x, y)`: relabel every element whose label is `label[y]`. $O(n)$

First Attempt: Store the Label

Keep an array `label[x]` = representative of x.

x	1	2	3	4	5	6
`label[`x`]`	1	1	1	4	4	6

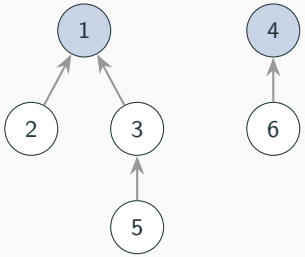
- `makeset(x)`: set `label[x] = x`. $O(1)$
- `find(x)`: return `label[x]`. $O(1)$
- `union(x, y)`: relabel every element whose label is `label[y]`. $O(n)$

$n-1$ unions cost $\Theta(n^2)$ in total. Can we make union cheap without making find expensive?

A Forest of Parent Pointers

Sets as Trees

Store each set as a rooted tree. Every element points to its parent. The root is the representative and points to itself.



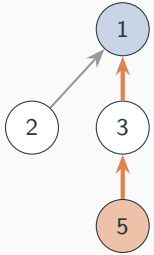
x	1	2	3	4	5	6
`parent[`x`]`	1	1	1	4	3	4
`rank[`x`]`	2	0	1	1	0	0

- `makeset(x)`: `parent[x] = x`, `rank[x] = 0`. A one-node tree.
- `find(x)`: follow parent pointers up to the root. Cost = depth of x.
- `rank(x)`: the height of the tree rooted at x. Here `rank(1)` $= 2$, `rank(4)` $= 1$.
- `union(x, y)`: find both roots and make one point to the other. Cost = two finds $+O(1)$.

makeset and find

```
function makeset!(parent, rank, x)
    parent[x] = x
    rank[x] = 0
end

function find(parent, x)
    while parent[x] != x
        x = parent[x]
    end
    return x
end
```



`find(5)` walks $5 \to 3 \to 1$ and returns 1.

Both arrays have one entry per element. `rank` only matters at roots.

How Should union Link the Roots?

Suppose `union(x, y)` always sets `parent[find(y)] = find(x)`. Then `union(2, 1)`, `union(3, 1)`, ..., `union(n, 1)` gives:



- Each union hangs the old tree below the new singleton.
- The tree becomes a path of height $n - 1$.
- `find(1)` now costs $\Theta(n)$, and n such finds cost $\Theta(n^2)$.

Two fixes:

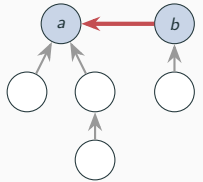
1. Union by rank: keep trees shallow when merging.
2. Path compression: flatten trees when searching.

Union by Rank

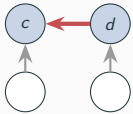
Union by Rank

Greedy rule: hang the root of smaller rank below the root of larger rank. Only a tie makes the tree taller.

```
function union!(parent, rank, x, y)
    rx, ry = find(parent, x), find(parent, y)
    rx == ry && return
    if rank[rx] < rank[ry]
        rx, ry = ry, rx
    end
    parent[ry] = rx
    if rank[rx] == rank[ry]
        rank[rx] += 1
    end
end
```



ranks $2, 1$:
rank stays 2



ranks $1, 1$:
rank of c becomes 2

Ranks stay equal to heights: below a taller root, the shorter tree does not increase the height; two equal heights h give height $h + 1$.

Union by Rank Keeps Trees Shallow

Lemma

With union by rank, a tree whose root has rank r contains at least 2^r elements. So every rank, and every height, is at most $\log_2 n$, and `find` costs $O(\log n)$.

Union by Rank Keeps Trees Shallow

Lemma

With union by rank, a tree whose root has rank r contains at least 2^r elements. So every rank, and every height, is at most $\log_2 n$, and `find` costs $O(\log n)$.

Proof by induction on the number of operations.

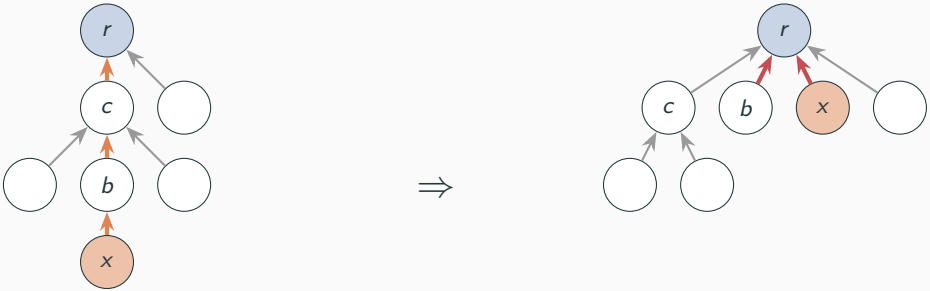
- `makeset`: rank 0, one element $= 2^0$.
- `union` with unequal ranks: the root of larger rank r keeps its rank and gains elements, so it still has $\geq 2^r$.
- `union` with equal ranks r: the new root has rank $r + 1$ and at least $2^r + 2^r = 2^{r+1}$ elements.

Since a tree has at most n elements, $2^r \leq n$, so $r \leq \log_2 n$. □

Path Compression

Path Compression

During `find(x)` we visit every vertex on the path from x to the root. Make all of them point directly to the root.



before `find(x)` after `find(x)`

The find costs the same this time, but every later find from x, b, or their descendants is faster.

Find with Path Compression

```
function find(parent, x)
    if parent[x] != x
        parent[x] = find(parent, parent[x])
    end
    return parent[x]
end
```

- The recursion walks up to the root.
- On the way back, each vertex on the path gets the root as its new parent.
- One extra assignment per visited vertex, so the cost is still $O(\text{depth})$.

Compression can make a tree shorter, but we do not update ranks. From now on `rank(x)` is only an upper bound on the height. The lemma still holds: its proof never used heights.

Both Together

Theorem (Tarjan, 1975)

With union by rank and path compression, any sequence of m operations on n elements takes $O(m\,\alpha(n))$ time.

- $\alpha(n)$ is the inverse Ackermann function. It grows absurdly slowly: $\alpha(n) \leq 4$ for every n smaller than the number of atoms in the universe.
- So in practice each operation is amortized constant time.
- The proof is beyond this course; the $O(\log n)$ bound from union by rank alone is the one to remember.

Example

Example Trace

Union by rank (ties: the root of y goes below the root of x), path compression.

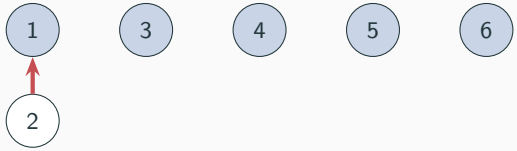


x	1	2	3	4	5	6
`parent[`x`]`	1	2	3	4	5	6
`rank[`x`]`	0	0	0	0	0	0

Start: `makeset(1)`, ..., `makeset(6)`.

Example Trace

Union by rank (ties: the root of y goes below the root of x), path compression.

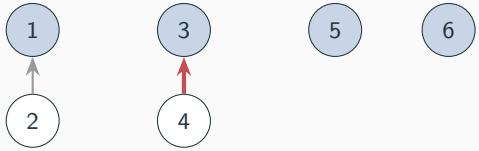


x	1	2	3	4	5	6
`parent[`x`]`	1	**1**	3	4	5	6
`rank[`x`]`	**1**	0	0	0	0	0

`union(1, 2)`: ranks tie at 0, so 2 goes below 1 and `rank(1)` becomes 1.

Example Trace

Union by rank (ties: the root of y goes below the root of x), path compression.

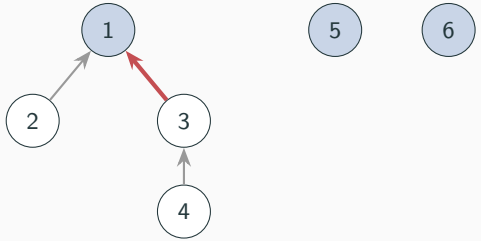


x	1	2	3	4	5	6
`parent[`x`]`	1	1	3	**3**	5	6
`rank[`x`]`	1	0	**1**	0	0	0

`union(3, 4)`: 4 goes below 3, `rank(3)` becomes 1.

Example Trace

Union by rank (ties: the root of y goes below the root of x), path compression.

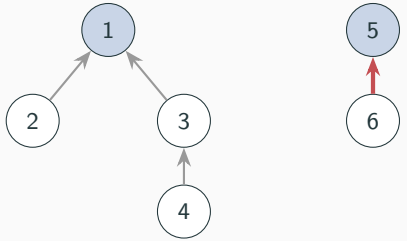


x	1	2	3	4	5	6
`parent[`x`]`	1	1	**1**	3	5	6
`rank[`x`]`	**2**	0	1	0	0	0

`union(2, 4)`: roots 1 and 3 tie at rank 1, so 3 goes below 1 and `rank(1)` becomes 2.

Example Trace

Union by rank (ties: the root of y goes below the root of x), path compression.

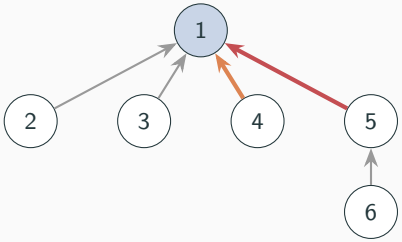


x	1	2	3	4	5	6
`parent[`x`]`	1	1	1	3	5	**5**
`rank[`x`]`	2	0	1	0	**1**	0

`union(5, 6)`: 6 goes below 5, `rank(5)` becomes 1.

Example Trace

Union by rank (ties: the root of y goes below the root of x), path compression.

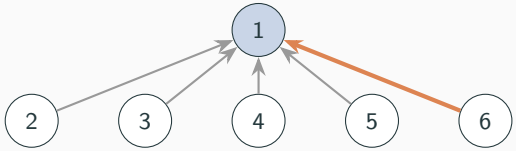


x	1	2	3	4	5	6
`parent[`x`]`	1	1	1	**1**	**1**	5
`rank[`x`]`	2	0	1	0	1	0

`union(6, 4)`: `find(4)` compresses $4 \to 3 \to 1$ (orange). Root 5 (rank 1) goes below root 1 (rank 2).

Example Trace

Union by rank (ties: the root of y goes below the root of x), path compression.



x	1	2	3	4	5	6
`parent[`x`]`	1	1	1	1	1	**1**
`rank[`x`]`	2	0	1	0	1	0

`find(6)`: compresses $6 \to 5 \to 1$. The tree now has height 1, but `rank(1)` is still 2.

Union-Find in Julia

```
struct UnionFind
    parent::Vector{Int}
    rank::Vector{Int}
end
UnionFind(n) = UnionFind(collect(1:n), zeros(Int, n))
function find(uf::UnionFind, x)
    if uf.parent[x] != x
        uf.parent[x] = find(uf, uf.parent[x])
    end
    return uf.parent[x]
end
function union!(uf::UnionFind, x, y)
    rx, ry = find(uf, x), find(uf, y)
    rx == ry && return false
    if uf.rank[rx] < uf.rank[ry]; rx, ry = ry, rx; end
    uf.parent[ry] = rx
    uf.rank[rx] == uf.rank[ry] && (uf.rank[rx] += 1)
    return true
end
```

- `UnionFind(n)` performs `makeset(x)` for $x = 1, \ldots, n$.
- The struct is immutable but its vectors are not, so `union!` can modify them.
- `union!` returns whether two different sets were merged. Kruskal's algorithm uses exactly this to decide whether to keep an edge.

Application: Counting Components

Count the connected components of an undirected graph with n vertices and edge list edges.

```
function count_components(n, edges)
    uf = UnionFind(n)
    count = n
    for (u, v) in edges
        if union!(uf, u, v)
            count -= 1
        end
    end
    return count
end
```

- Start with n components.
- Each edge joining two different sets merges two components.
- Time $O(n + m\,\alpha(n))$, and edges can be processed as they arrive.

Exercise

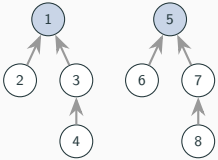
Start with `makeset(1)`, ..., `makeset(8)`. Use union by rank (ties: the root of y goes below the root of x) and path compression.

`union(1,2)`, `union(3,4)`, `union(5,6)`, `union(7,8)`,
`union(1,3)`, `union(5,7)`, `union(4,8)`, `find(6)`

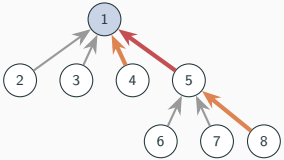
1. Draw the forest after the first six unions, with the rank of each root.
2. Draw the forest after `union(4,8)`, and again after `find(6)`. Give the final `parent` array and `rank(1)`. Does `rank(1)` equal the height?
3. Without path compression, what is the largest possible rank of a tree on 8 elements built by union by rank?

Solution

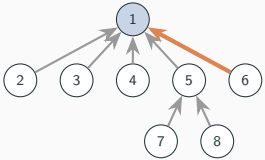
(a) after six unions



(b) after `union(4,8)`



(b) after `find(6)`



In `union(4,8)`, `find(4)` compresses 4 onto 1 and `find(8)` compresses 8 onto 5; the ranks tie at 2, so 5 goes below 1 and `rank(1)` $= 3$. Final `parent` $= [1, 1, 1, 1, 1, 1, 5, 5]$. The tree has height $2 <$ `rank(1)`: after compression, rank is only an upper bound.

(c) $\log_2 8 = 3$, by the lemma. It is reached by merging equal-rank trees: pairs, then pairs of pairs, then the two halves.

Summary

implementation	`find`	`union`
label array	$O(1)$	$O(n)$
parent forest, arbitrary link	$O(n)$	$O(n)$
union by rank	$O(\log n)$	$O(\log n)$
union by rank + path compression	$O(\alpha(n))$ amortized	

- Each set is a tree of parent pointers; its root is the representative.
- Union by rank: hang the root of smaller rank below the other; a rank-r tree has $\geq 2^r$ elements, so heights stay $\leq \log_2 n$.
- Path compression: point everything on a find path at the root.