Disjoint-sets data type

Goal. Support three operations on a collection of disjoint sets.
- **MAKE-SET(x)**: create a new set containing only element \( x \).
- **FIND(x)**: return a canonical element in the set containing \( x \).
- **UNION(x, y)**: replace the sets containing \( x \) and \( y \) with their union.

Performance parameters.
- \( m \) = number of calls to **MAKE-SET**, **FIND**, and **UNION**.
- \( n \) = number of elements = number of calls to **MAKE-SET**.

Dynamic connectivity. Given an initially empty graph \( G \), support three operations.
- **ADD-NODE(u)**: add node \( u \).
- **ADD-EDGE(u, v)**: add an edge between nodes \( u \) and \( v \).
- **IS-CONNECTED(u, v)**: is there a path between \( u \) and \( v \)?

Disjoint-sets data type: applications

Original motivation. Compiling **EQUIVALENCE**, **DIMENSION**, and **COMMON** statements in Fortran.

An Improved Equivalence Algorithm

BERNARD A. GAIGER AND MICHAEL J. FISHER
University of Michigan, Ann Arbor, Michigan

An algorithm for assigning storage on the basis of **EQUIVALENCE**, **DIMENSION** and **COMMON** declarations is presented. The algorithm is based on a tree structure, and has reduced computation time by 40 percent over a previously published algorithm by identifying all equivalence classes with one scan of the **EQUIVALENCE** declarations. The method is applicable in any problem in which it is necessary to identify equivalence classes, given the element pairs defining the equivalence relation.

Note. This 1964 paper also introduced key data structure for problem.
**Union-Find**

- naïve linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context

---

**Disjoint-sets data structure**

**Parent-link representation.** Represent each set as a tree of elements.
- Each element has an explicit parent pointer in the tree.
- The root serves as the canonical element (and points to itself).
- \( \text{Find}(x) \): find the root of the tree containing \( x \).
- \( \text{Union}(x, y) \): merge trees containing \( x \) and \( y \)
  (by making one root point to the other root).

\( \text{Union}(3, 5) \)

---

**Array representation.** Represent each set as a tree of elements.
- Allocate an array \( \text{parent}[i] \) of length \( n \).
- \( \text{parent}[i] = j \) means parent of element \( i \) is element \( j \).

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
7 & 5 & 7 & 8 & 8 & 7 & 5 & 7 & 8 & 8 \\
\end{array}
\]

---

**Note.** For brevity, we suppress arrows and self-loops in figures.
**Naïve linking**

**Naïve linking.** Link root of first tree to root of second tree.

\[
\text{UNION}(5, 3)
\]

![Diagram of UNION(5, 3)](image)

**Naïve linking**

**Naïve linking.** Link root of first tree to root of second tree.

\[
\text{UNION}(5, 3)
\]

![Diagram of UNION(5, 3)](image)

**Naïve linking**

**Naïve linking.** Link root of first tree to root of second tree.

\[
\text{MAKE-SET}(x)
\]

\[
\begin{align*}
\text{parent}[x] & \leftarrow x. \\
\text{FIND}(x) & \text{ WHILE } (x \neq \text{parent}[x]) \\
& x \leftarrow \text{parent}[x]. \\
\text{RETURN } x.
\end{align*}
\]

\[
\text{UNION}(x, y)
\]

\[
\begin{align*}
r & \leftarrow \text{FIND}(x). \\
s & \leftarrow \text{FIND}(y). \\
\text{parent}[r] & \leftarrow s.
\end{align*}
\]

**Naïve linking: analysis**

**Theorem.** Using naïve linking, a UNION or FIND operation can take \(\Theta(n)\) time in the worst case, where \(n\) is the number of elements.

**Pf.**

- In the worst case, FIND takes time proportional to the height of the tree.
- Height of the tree is \(n - 1\) after the sequence of union operations: \(\text{UNION}(1, 2), \text{UNION}(2, 3), \ldots, \text{UNION}(n - 1, n)\).
**UNION–FIND**

- naive linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context

**Link-by-size**

*Link-by-size.* Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

**MAKE-SET(x)**

\[
\text{size}[x] \leftarrow 1.
\]

**UNION(x, y)**

\[
\begin{align*}
\text{parent}[x] & \leftarrow x. \\
\text{size}[x] & \leftarrow 1. \\
\text{parent}[y] & \leftarrow x. \\
\text{size}[y] & \leftarrow \text{size}[x] + \text{size}[y].
\end{align*}
\]

**FIND(x)**

\[
\begin{align*}
\text{parent}[x] & \leftarrow x. \\
\text{size}[x] & \leftarrow \text{size}[x].
\end{align*}
\]

**WHILE** (x ≠ parent[x])

\[
\begin{align*}
x & \leftarrow \text{parent}[x].
\end{align*}
\]

**RETURN x.**

**UNION(5, 3)**

<table>
<thead>
<tr>
<th>size = 4</th>
<th>size = 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

**Link-by-size**

*Link-by-size.* Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).
**Property.** Using link-by-size, for every root node $r$: $\text{size}[r] \geq 2^{\text{height}(r)}$.

**Pf.** [by induction on number of links]
- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first $i$ links.
- Tree rooted at $r$ changes only when a smaller (or equal) size tree rooted at $s$ is linked into $r$.
- Case 1. [$\text{height}(r) > \text{height}(s)$] $\text{size}'[r] > \text{size}[r]$
  \[ \geq 2^{\text{height}(r)} \quad \text{inductive hypothesis} \]
  \[ = 2^{\text{height}(r)}. \]

**Theorem.** Using link-by-size, any UNION or FIND operation takes $O(\log n)$ time in the worst case, where $n$ is the number of elements.

**Pf.**
- The running time of each operation is bounded by the tree height.
- By the previous property, the height is $\leq \lfloor \log n \rfloor$.

\[ \log n = \log_2 n \]

**Note.** The UNION operation takes $O(1)$ time except for its two calls to FIND.

**A tight upper bound**

**Theorem.** Using link-by-size, a tree with $n$ nodes can have height $= \log n$.

**Pf.**
- Arrange $2^k - 1$ calls to UNION to form a binomial tree of order $k$.
- An order-$k$ binomial tree has $2^k$ nodes and height $k$.  

- $B_0 \quad B_1 \quad B_2 \quad B_3 \quad B_4$
**Union-Find**

- naïve linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context

**Link-by-rank**

**Link-by-rank.** Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.

**Union(5, 3)**

- rank = 1
- rank = 2

**Note.** For now, rank = height.

**Link-by-rank**

**Link-by-rank.** Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.

**Make-Set(x)**

- `parent[x] ← x.`
- `rank[x] ← 0.`

**Find(x)**

- **While** `(x ≠ parent[x])`
  - `x ← parent[x].`
- **RETURN** `x.`

**Union(x, y)**

- `r ← Find(x).`
- `s ← Find(y).`
- **IF** `(r = s)` **RETURN**.
- **ELSE IF** `(rank[r] > rank[s])`
  - `parent[s] ← r.`
- **ELSE IF** `(rank[r] < rank[s])`
  - `parent[r] ← s.`
- **ELSE**
  - `parent[r] ← s.`
  - `rank[s] ← rank[s] + 1.`
**Link-by-rank: properties**

**PROPERTY 1.** If \( x \) is not a root node, then \( rank[x] < rank[parent[x]]. \)

*Pf.* A node of rank \( k \) is created only by linking two roots of rank \( k - 1 \).

**PROPERTY 2.** If \( x \) is not a root node, then \( rank[x] \) will never change again.

*Pf.* Rank changes only for roots; a nonroot never becomes a root.

**PROPERTY 3.** If \( parent[x] \) changes, then \( rank[parent[x]] \) strictly increases.

*Pf.* The parent can change only for a root, so before linking \( parent[x] = x \).

After \( x \) is linked-by-rank to new root \( r \) we have \( rank[r] > rank[x]. \).

\[ \text{rank} = 0 \]

\[ \text{rank} = 1 \]

\[ \text{rank} = 2 \]

\[ \text{rank} = 3 \]

**Link-by-rank: properties**

**PROPERTY 4.** Any root node of rank \( k \) has \( \geq 2^k \) nodes in its tree.

*Pf.* [by induction on \( k \)]

- Base case: true for \( k = 0 \).
- Inductive hypothesis: assume true for \( k - 1 \).
- A node of rank \( k \) is created only by linking two roots of rank \( k - 1 \).
- By inductive hypothesis, each subtree has \( \geq 2^{k-1} \) nodes

\[ \Rightarrow \text{resulting tree has } \geq 2^k \text{ nodes.} \]

**PROPERTY 5.** The highest rank of a node is \( \leq \lceil \log n \rceil \).

*Pf.* Immediate from **PROPERTY 1** and **PROPERTY 4**.

**Link-by-rank: analysis**

**Theorem.** Using link-by-rank, any UNION or FIND operation takes \( O(\log n) \) time in the worst case, where \( n \) is the number of elements.

*Pf.*

- The running time of UNION and FIND is bounded by the tree height.
- By **PROPERTY 5**, the height is \( \leq \lceil \log n \rceil \).
**SECTION 5.1.4**

**UNION–FIND**

- naive linking
- link-by-size
- link-by-rank
- **path compression**
- link-by-rank with path compression
- context

**Path compression**

*Path compression.* When finding the root \( r \) of the tree containing \( x \), change the parent pointer of all nodes along the path to point directly to \( r \).
**Path compression**

Path compression. When finding the root $r$ of the tree containing $x$, change the parent pointer of all nodes along the path to point directly to $r$.

---

**Path compression**

Path compression. When finding the root $r$ of the tree containing $x$, change the parent pointer of all nodes along the path to point directly to $r$.

---

**Path compression**

Path compression. When finding the root $r$ of the tree containing $x$, change the parent pointer of all nodes along the path to point directly to $r$.

---

**Path compression**

Path compression. When finding the root $r$ of the tree containing $x$, change the parent pointer of all nodes along the path to point directly to $r$.

---

Note. Path compression does not change the rank of a node; so $\text{height}(x) \leq \text{rank}(x)$ but they are not necessarily equal.
Path compression

**Fact.** Path compression with naïve linking can require $\Omega(n)$ time to perform a single `UNION` or `FIND` operation, where $n$ is the number of elements.

**Pf.** The height of the tree is $n - 1$ after the sequence of union operations: `UNION(1, 2), UNION(2, 3), …, UNION(n–1, n)`. 

naïve linking: link root of first tree to root of second tree

**Theorem.** [Tarjan–van Leeuwen 1984] Starting from an empty data structure, path compression with naïve linking performs any intermixed sequence of $m \geq n$ `MAKE-SET`, `UNION`, and `FIND` operations on a set of $n$ elements in $O(m \log n)$ time.

**Pf.** Nontrivial (but omitted).

---

Link-by-rank with path compression: properties

**PROPERTY.** The tree roots, node ranks, and elements within a tree are the same with or without path compression.

**Pf.** Path compression does not create new roots, change ranks, or move elements from one tree to another. ●

---

Union–Find

- naïve linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context

---

Link-by-rank with path compression: properties

**PROPERTY.** The tree roots, node ranks, and elements within a tree are the same with or without path compression.

**Corollary.** Property 2, 4–6 hold for link-by-rank with path compression.

**PROPERTY 1.** If $x$ is not a root node, then $\text{rank}[x] < \text{rank}[	ext{parent}[x]]$.

**PROPERTY 2.** If $x$ is not a root node, then $\text{rank}[x]$ will never change again.

**PROPERTY 3.** If $\text{parent}[x]$ changes, then $\text{rank}[	ext{parent}[x]]$ strictly increases.

**PROPERTY 4.** Any root node of rank $k$ has $\geq 2^k$ nodes in its tree.

**PROPERTY 5.** The highest rank of a node is $\leq \lceil \log n \rceil$.

**PROPERTY 6.** For any integer $k \geq 0$, there are $\leq n / 2^k$ nodes with rank $k$.

**Bottom line.** Property 1–6 hold for link-by-rank with path compression. (but we need to recheck Property 1 and Property 3)
**Link-by-rank with path compression: properties**

**PROPERTY 3.** If \( \text{parent}[x] \) changes, then \( \text{rank}[\text{parent}[x]] \) strictly increases.  
**Pf.** Path compression can make \( x \) point to only an ancestor of \( \text{parent}[x] \).

**PROPERTY 1.** If \( x \) is not a root node, then \( \text{rank}[x] < \text{rank}[\text{parent}[x]] \).  
**Pf.** Path compression doesn’t change any ranks, but it can change parents. If \( \text{parent}[x] \) doesn’t change during a path compression, the inequality continues to hold; if \( \text{parent}[x] \) changes, then \( \text{rank}[\text{parent}[x]] \) strictly increases.

![before path compression](image1)

![after path compression](image2)

---

**Iterated logarithm function**

**Def.** The iterated logarithm function is defined by:

\[
\lg^* n = \begin{cases} 
0 & \text{if } n \leq 1 \\
1 + \lg^* (\lg n) & \text{otherwise}
\end{cases}
\]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \lg^* n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>[3, 4]</td>
<td>2</td>
</tr>
<tr>
<td>[5, 16]</td>
<td>3</td>
</tr>
<tr>
<td>[17, 65536]</td>
<td>4</td>
</tr>
<tr>
<td>[65537, ( 2^{65536} )]</td>
<td>5</td>
</tr>
</tbody>
</table>

**Note.** We have \( \lg^* n \leq 5 \) unless \( n \) exceeds the # atoms in the universe.

---

**Analysis**

Divide nonzero ranks into the following groups:

- \( \{ 1 \} \)
- \( \{ 2 \} \)
- \( \{ 3, 4 \} \)
- \( \{ 5, 6, \ldots, 16 \} \)
- \( \{ 17, 18, \ldots, 2^{16} \} \)
- \( \{ 65537, 65538, \ldots, 2^{65536} \} \)
- ...

**Property 7.** Every nonzero rank falls within one of the first \( \lg^* n \) groups.  
**Pf.** The rank is between 0 and \( \lfloor \lg n \rfloor \). [PROPERTY 5]

---

**Creative accounting**

**Credits.** A node receives credits as soon as it ceases to be a root. If its rank is in the interval \( \{ k+1, k+2, \ldots, 2^k \} \), we give it \( 2^k \) credits.

**Proposition.** Number of credits disbursed to all nodes is \( \leq n \lg^* n \).  
**Pf.**

- All nodes in group \( k \) have rank \( \geq k + 1 \).
- By PROPERTY 6, the number of nodes with rank \( \geq k + 1 \) is at most
  \[
  \frac{n}{2^k + 1} + \frac{n}{2^{k+1} + 2} + \cdots \leq \frac{n}{2^k}
  \]
  
  Thus, nodes in group \( k \) need at most \( n \) credits in total.
- There are \( \leq \lg^* n \) groups. [PROPERTY 7]  ●
Running time of FIND

Running time of FIND. Bounded by number of parent pointers followed.

- Recall: the rank strictly increases as you go up a tree. [PROPERTY 1]
- Case 0: parent[x] is a root ⇒ only happens for one link per FIND.
- Case 1: rank[parent[x]] is in a higher group than rank[x].
- Case 2: rank[parent[x]] is in the same group as rank[x].

Case 1. At most \( \lg^* n \) nodes on path can be in a higher group. [PROPERTY 7]

Case 2. These nodes are charged 1 credit to follow parent pointer.

- Each time \( x \) pays 1 credit, rank[parent[x]] strictly increases. [PROPERTY 1]
- Therefore, if rank[x] is in the group \( \{ k+1, \ldots, 2^k \} \), the rank of its parent will be in a higher group before \( x \) pays \( 2^k \) credits.
- Once rank[parent[x]] is in a higher group than rank[x], it remains so because:
  - rank[x] does not change once it ceases to be a root. [PROPERTY 2]
  - rank[parent[x]] does not decrease. [PROPERTY 3]
  - thus, \( x \) has enough credits to pay until it becomes a Case 1 node.

Link-by-rank with path compression

Theorem. Starting from an empty data structure, link-by-rank with path compression performs any intermixed sequence of \( m \geq n \) MAKE-SET, UNION, and FIND operations on a set of \( n \) elements in \( O(m \log^* n) \) time.

Link-by-size with path compression

Theorem. [Fischer 1972] Starting from an empty data structure, link-by-size with path compression performs any intermixed sequence of \( m \geq n \) MAKE-SET, UNION, and FIND operations on a set of \( n \) elements in \( O(m \log \log n) \) time.
**Ackermann function.** [Ackermann 1928] A computable function that is not primitive recursive.

\[
A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\
  A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 
\end{cases}
\]

\[
\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\}
\]

"I am not smart enough to understand this easily."

— Raymond Seidel
Inverse Ackermann function

Definition.
\[ \alpha_k(n) = \begin{cases} \lceil n / 2 \rceil & \text{if } k = 1 \\ 0 & \text{if } n = 1 \text{ and } k \geq 2 \\ 1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} \end{cases} \]

Ex.
- \( \alpha_0(n) = \lceil n / 2 \rceil \)
- \( \alpha_1(n) = \lceil \log n \rceil = \# \text{ of times we divide } n \text{ by } 2, \text{ until we reach } 1. \)
- \( \alpha_2(n) = \lceil \log \log n \rceil = \# \text{ of times we apply the } \log \text{ function to } n, \text{ until we reach } 1. \)
- \( \alpha_3(n) = \# \text{ of times we apply the iterated } \log \text{ function to } n, \text{ until we reach } 1. \)

\[
\alpha_2(50) = 5, \quad \alpha_2(1000) = 12, \quad \alpha_2(1000000) = 19
\]

A tight lower bound

Theorem. [Fredman–Saks 1989] In the worst case, any \( \text{CELL-PROBE}(\log n) \) algorithm requires \( \Omega(m \alpha(m, n)) \) time to perform an intermixed sequence of \( m \) \( \text{MAKE-SET}, \text{UNION}, \) and \( \text{FIND} \) operations on a set of \( n \) elements.

Cell-probe model. [Yao 1981] Count only number of words of memory accessed; all other operations are free.

Path compaction variants

Path splitting. Make every node on path point to its grandparent.

<table>
<thead>
<tr>
<th>Inverse Ackermann function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition.</td>
</tr>
<tr>
<td>[ \alpha_k(n) = \begin{cases} \lceil n / 2 \rceil &amp; \text{if } k = 1 \ 0 &amp; \text{if } n = 1 \text{ and } k \geq 2 \ 1 + \alpha_k(\alpha_{k-1}(n)) &amp; \text{otherwise} \end{cases} ]</td>
</tr>
</tbody>
</table>

Property. For every \( n \geq 5, \) the sequence \( \alpha_1(n), \alpha_2(n), \alpha_3(n), \ldots \) converges to 3.
Ex. \( \lceil n = 98761 \rceil \) \( \alpha_3(98761) \geq 10^{35163}, \) \( \alpha_2(98761) = 116812, \alpha_1(98761) = 6, \alpha_2(98761) = 4, \alpha_3(98761) = 3. \)

One-parameter inverse Ackermann. \( \alpha(n) = \min \{ k : \alpha_k(n) \leq 3 \}. \)
Ex. \( \alpha(98761) = 5. \)

Two-parameter inverse Ackermann. \( \alpha(m, n) = \min \{ k : \alpha_k(n) \leq 3 + m / n \}. \)
**Path compaction variants**

**Path halving.** Make every other node on path point to its grandparent.

**Disjoint-sets data structures**

**Theorem.** [Tarjan–van Leeuwen 1984] Starting from an empty data structure, link-by- { size, rank } combined with { path compression, path splitting, path halving } performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of $n$ elements in $O(m \alpha(m, n))$ time.

**Worst-Case Analysis of Set Union Algorithms**

ROBERT E. TARJAN

AT&T Bell Laboratories, Murray Hill, New Jersey

AND

JAN VAN LEEUWEN

University of Utrecht, Utrecht, The Netherlands

Abstract. This paper analyzes the asymptotic worst-case running time of a number of variants of the well-known method of path compression for maintaining a collection of disjoint sets under union. We show that two one-pass methods proposed by van Leeuwen and van der Wiele are asymptotically optimal, whereas several other methods, including one proposed by Ross and advocated by Dijkstra, are slower than the best methods.