Disjoint sets data type

**Goal.** Support two operations on a set of elements:
- **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).** Merge the sets containing x and y.

**Dynamic connectivity.** Given an initial empty graph G on n nodes, support the following queries:
- **ADD-EDGE(u, v).** Add an edge between nodes u and v.  
  \[ \text{1 union operation} \]
- **IS-CONNECTED(u, v).** Is there a path between u and v?  
  \[ \text{2 find operations} \]

Disjoint sets data type: applications

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

**An Improved Equivalence Algorithm**

Bernard A. Galler and Michael J. Fischer
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.
Disjoint-sets data structure

**Representation.** Represent each set as a tree of elements.
- Each element has a parent pointer in the tree.
- The root serves as the canonical element.
- **FIND**(x). Find the root of the tree containing x.
- **UNION**(x, y). Make the root of one tree point to root of other tree.

**Link-by-size.** Maintain a subtree count for each node, initially 1.
Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

**Note.** For brevity, we suppress arrows and self loops in figures.
**Link-by-size**

**Link-by-size.** Maintain a subtree count for each node, initially 1.
Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

**MAKE-SET (x)**

\[\text{parent}(x) \leftarrow x.\]

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

**FIND (x)**

\[\text{WHILE} \ (x \neq \text{parent}(x)) \]

\[\quad x \leftarrow \text{parent}(x).\]

**RETURN x.**

**UNION-BY-SIZE (x, y)**

\[r \leftarrow \text{FIND} (x).\]

\[s \leftarrow \text{FIND} (y).\]

**IF** \(r = s\) **RETURN.**

**ELSE IF** \(\text{size}(r) > \text{size}(s)\)

\[\quad \text{parent}(s) \leftarrow r.\]

\[\quad \text{size}(r) \leftarrow \text{size}(r) + \text{size}(s).\]

**ELSE**

\[\quad \text{parent}(r) \leftarrow s.\]

\[\quad \text{size}(s) \leftarrow \text{size}(r) + \text{size}(s).\]

---

**Link-by-size: analysis**

**Property.** Using link-by-size, for every root node \(r\), \(\text{size}(r) \geq 2^{\log \text{size}(r)}\).

**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 tree rooted at** \(s\) **is linked into** \(r\).
- **Case 1.** \(\text{height}(r) > \text{height}(s)\)

\[\text{size}'(r) \geq \text{size}(r)\]

\[\geq 2^\text{height}(r) \quad \text{inductive hypothesis}\]

**Case 2.** \(\text{height}(r) \leq \text{height}(s)\)

\[\text{size}'(r) = \text{size}(r) + \text{size}(s)\]

\[\geq 2 \cdot \text{size}(s) \quad \text{link-by-size}\]

\[\geq 2 \cdot 2^\text{height}(s) \quad \text{inductive hypothesis}\]

\[= 2^\text{height}(r) + 1\]

\[= 2^\text{height}'(r). \qedhere\]

---

**Theorem.** Using link-by-size, any UNION or FIND operations 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 \(\leq \lfloor \log n \rfloor\).

\[\lfloor \log n \rfloor = \log_2 n\]
A matching lower bound

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

**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 $k$.

---

**Union-Find**

- **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 new root by 1.

---

**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 new root by 1.

---

**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 new root by 1.

\[
\text{MAKE-SET}(x) \\
\text{parent}(x) \leftarrow x. \\
\text{rank}(x) \leftarrow 0.
\]

\[
\text{UNION-BY-RANK}(x, y) \\
r \leftarrow \text{FIND}(x). \\
s \leftarrow \text{FIND}(y). \\
\text{IF } (r = s) \text{ RETURN.} \\
\text{ELSE IF } \text{rank}(r) > \text{rank}(s) \\
\text{parent}(s) \leftarrow r. \\
\text{ELSE IF } \text{rank}(r) < \text{rank}(s) \\
\text{parent}(r) \leftarrow s. \\
\text{ELSE} \\
\text{parent}(r) \leftarrow s. \\
\text{rank}(s) \leftarrow \text{rank}(s) + 1.
\]

**Find** 

\[
\text{WHILE } x \neq \text{parent}(x) \\
\text{s} \leftarrow \text{parent}(x). \\
\text{RETURN } x.
\]

**Link-by-rank: properties**

**Property 1.** If \(x\) is not a root node, then \(\text{rank}(x) < \text{rank}(\text{parent}(x))\).

\[\text{Pf.} \quad \text{A node of rank } k \text{ is created only by merging two roots of rank } k - 1.\]

**Property 2.** If \(x\) is not a root, then \(\text{rank}(x)\) will never change again.

\[\text{Pf.} \quad \text{Rank changes only for roots; a nonroot never becomes a root.}\]

**Property 3.** If \(\text{parent}(x)\) changes, then \(\text{rank}(\text{parent}(x))\) strictly increases.

\[\text{Pf.} \quad \text{The parent can change only for a root, so before linking } \text{parent}(x) = x; \text{After } x \text{ is linked-by-rank to new root } r \text{ we have } \text{rank}(r) > \text{rank}(x).\]

\[\text{\begin{figure}
\centering
\includegraphics[width=\textwidth]{tree1.png}
\caption{Example tree with ranks.}
\end{figure}}\]

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

\[\text{Pf.} \quad [\text{by induction on } k]\]

- **Base case:** true for \(k = 0\).
- **Inductive hypothesis:** assume true for \(k - 1\).
- A node of rank \(k\) is created only by merging 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 \(\leq \lfloor \log n \rfloor\).

\[\text{Pf.} \quad \text{Immediate from PROPERTY 1 and PROPERTY 4.}\]

**Property 6.** For any integer \(r \geq 0\), there are \(\leq n / 2^r\) nodes with rank \(r\).

\[\text{Pf.} \quad \begin{align*}
\bullet \text{ Any root node of rank } k \text{ has } \geq 2^k \text{ descendants.} & \quad [\text{PROPERTY 4}] \\
\bullet \text{ Any nonroot node of rank } k \text{ has } \geq 2^k \text{ descendants because:} & \\
& \quad - \text{it had this property just before it became a nonroot } \quad [\text{PROPERTY 4}] \\
& \quad - \text{its rank doesn’t change once it becomes a nonroot } \quad [\text{PROPERTY 2}] \\
& \quad - \text{its set of descendants doesn’t change once it became a nonroot} \\
\bullet \text{ Different nodes of rank } k \text{ can’t have common descendants.} & \quad [\text{PROPERTY 1}] \quad \end{align*}\]

\[\text{\begin{figure}
\centering
\includegraphics[width=\textwidth]{tree2.png}
\caption{Example tree with ranks.}
\end{figure}}\]
Link-by-rank: analysis

Theorem. Using link-by-rank, any UNION or FIND operations 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 PROPERTY 5, the height is $\leq \lfloor \lg n \rfloor$. □

**SECTION 5.1.4**

**UNION-FIND**

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

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

\[
\text{Find}(x) \\
\text{If } x \neq \text{parent}(x) \quad \text{parent}(x) \leftarrow \text{Find}(\text{parent}(x)). \\
\text{Return parent}(x).
\]

**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.

---

**Fact.** Path compression (with naive linking) can require \( \Omega(n) \) time to perform a single \textsc{Union} or \textsc{Find} operation, where \( n \) is the number of elements.

**Pf.** The height of the tree is \( n - 1 \) after the sequence of union operations: \textsc{Union}(1, 2), \textsc{Union}(2, 3), \ldots, \textsc{Union}(n - 1, n).

**Theorem.** [Tarjan-van Leeuwen 1984] Starting from an empty data structure, path compression (with naive linking) performs any intermixed sequence of \( m \geq n \) find and \( n - 1 \) union operations 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.
**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 \( \text{rank}(x) < \text{rank}(\text{parent}(x)) \).
**Pf.** Path compression only increases rank of parent.

**Property 2.** If \( x \) is not a root, then \( \text{rank}(x) \) will never change again.
**Property 3.** If \( \text{parent}(x) \) changes, then \( \text{rank}(\text{parent}(x)) \) 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 \( \leq \lfloor \log n \rfloor \).
**Property 6.** For any integer \( r \geq 0 \), there are \( \leq n / 2^r \) nodes with rank \( r \).

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

---

**Iterated logarithm function**

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

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

<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, ..., 16 \}
- \{ 17, 18, ..., 2^{16} \}
- \{ 65537, 65538, ..., 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.
- By Property 6, the number of nodes with rank \( \geq k + 1 \) is at most \( \frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k} \).
- 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: \( \text{parent}(x) \) is a root \( \Rightarrow \) only happens for one link per FIND.
- Case 1: \( \text{rank}(\text{parent}(x)) \) is in a higher group than \( \text{rank}(x) \).
- Case 2: \( \text{rank}(\text{parent}(x)) \) is in the same group as \( \text{rank}(x) \).

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, \( \text{rank}(\text{parent}(x)) \) strictly increases. [Property 1]
- Therefore, if \( \text{rank}(x) \) 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 \( \text{rank}(\text{parent}(x)) \) is in a higher group than \( \text{rank}(x) \), it remains so because:
  - \( \text{rank}(x) \) does not change once it ceases to be a root. [Property 2]
  - \( \text{rank}(\text{parent}(x)) \) 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-size with path compression performs any intermixed sequence of \( m \geq n \) FIND and \( n - 1 \) UNION operations in \( O(m \log^* n) \) time.

<table>
<thead>
<tr>
<th>UNION-FIND</th>
</tr>
</thead>
<tbody>
<tr>
<td>link-by-size</td>
</tr>
<tr>
<td>link-by-rank</td>
</tr>
<tr>
<td>path compression</td>
</tr>
<tr>
<td>link-by-rank with path compression</td>
</tr>
<tr>
<td>context</td>
</tr>
</tbody>
</table>
Link-by-size with path compression

Theorem. [Fischer 1972] Link-by-size with path compression performs any intermixed sequence of \( m \geq n \) FIND and \( n - 1 \) UNION operations in \( O(m \log \log n) \) time.

Efficiency of a Good But Not Linear Set Union Algorithm

Robert Endre Tarjan
University of California, Berkeley, California

Abstract. Two types of instructions for manipulating a family of disjoint sets which partition a universe of \( n \) elements are considered: \textsc{find}(p), which computes the name of the (unique) set containing element \( p \); and \textsc{union}(a,b), which chooses a new set name \( c \) and assigns \( a \) and \( b \) new names \( c \). A known algorithm for implementing such instructions is linear in the worst case. It is shown here that, if \( \text{find}(p) \) and \( \text{find}(q) \) are the minimum times required by a sequence of \( m \) \textsc{find}(p) and \( n \) \textsc{union}(a,b) instructions, then \( m \text{\text{\textsc{find}}}(n) + n \text{\text{\textsc{union}}}(a,b) \) is related to a functional inverse of Ackermann’s function and is very slow growing.

Ackermann function

Ackermann function. A computable function that is not primitive recursive.

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

Zum Hilbertschen Aufbau der reellen Zahlen

Von Wilhelm Ackermann in Göttingen.

Um das Beweis für die von Cantor aufgestellte Vermutung zu erbringen, daß die Menge der reellen Zahlen, d. h. der unbeschränkten Funktionen, unter Berücksichtigung der Zahl der maximalen und minimalen Lösungen endlich, benutzt Hilberts seinen Aufbau der reellen Zahlen. Wesentlich bei diesem Aufbau ist der auf die Klasse der Funktionen. Eine Funktion vom Typ 1 ist eine solche, daß Argumente und Werte ganze Zahlen sind, aber eine gewisse unscharfe Anzahl von Zahlen. Die Funktionen vom Typ 2 sind die Funktionenpunkte. Eine bestimmte Funktion erzeugt jede unbeschränkte Funktion von einer Zahl zu, einer Funktion von Typ 2 erzeugt wiederum die Funktionenpunkte Zahlen zu, und die Definition der Funktion fällt sich auch im Übrigen fortsetzen, für den Übergang dieser Arbeit ist das aber nicht von Belang.

Note. There are many inequivalent definitions.
Ackermann function. A computable function that is not primitive recursive.

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

Inverse Ackermann function.

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

“I am not smart enough to understand this easily.”
— Raymond Seidel

Inverse Ackermann function

Definition.

\[ \alpha_k(n) = \begin{cases} 
  1 & \text{if } n = 1 \\
  \lfloor n/2 \rfloor & \text{if } k = 1 \\
  1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} 
\end{cases} \]

Ex.

- \( \alpha_1(n) = \lceil n/2 \rceil \)
- \( \alpha_2(n) = \lceil \log n \rceil = \# \text{ of times we divide } n \text{ by two, until we reach 1.} \)
- \( \alpha_3(n) = \lceil \log^2 n \rceil = \# \text{ of times we apply the } \log \text{ function to } n, \text{ until we reach 1.} \)
- \( \alpha_4(n) = \# \text{ of times we apply the iterated } \log \text{ function to } n, \text{ until we reach 1.} \)

\[ \begin{array}{cccccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & \ldots & 2^{65536} & \ldots & 2^{65536} & \ldots & 2^{65536} \\
\alpha_1(n) & 1 & 1 & 2 & 3 & 4 & 4 & 5 & 5 & 6 & 6 & 7 & 8 & 8 & 8 & \ldots & 2^{65536} & \ldots & 2^{65536} & \ldots & 2^{65536} \\
\alpha_2(n) & 1 & 1 & 2 & 3 & 3 & 3 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & \ldots & 16 & \ldots & 65536 & \ldots & 2 \uparrow 65536 & \ldots & 65535 \\
\alpha_3(n) & 1 & 1 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & \ldots & 4 & \ldots & 5 & \ldots & 65536 & \ldots & 65536 \\
\alpha_4(n) & 1 & 1 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & \ldots & 3 & \ldots & 3 & \ldots & 4 & \ldots & 65536 \\
\end{array} \]

A matching lower bound

Theorem. [Fredman-Saks 1989] Any CPROBE(\log n) algorithm for the disjoint set union problem requires \( \Omega(m \alpha(m, n)) \) time to perform an intermixed sequence of \( m \geq n \) FIND and \( n - 1 \) UNION operations in the worst case.

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

A matching lower bound

Theorem. [Fredman-Saks 1989] Any CPROBE(\log n) algorithm for the disjoint set union problem requires \( \Omega(m \alpha(m, n)) \) time to perform an intermixed sequence of \( m \geq n \) FIND and \( n - 1 \) UNION operations in the worst case.

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.

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

### Linking variants

**Link-by-size.** Number of nodes in tree.

**Link-by-rank.** Rank of tree.

**Link-by-random.** Label each element with a random real number between 0.0 and 1.0. Link root with smaller label into root with larger label.

### Disjoint set union algorithms

**Theorem.** [Tarjan-van Leeuwen 1984] Link-by- \{ size, rank \} combined with \{ path compression, path splitting, path halving \} performs any intermixed sequence of \( m \geq n \) find and \( n - 1 \) union operations 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 prior methods proposed by van Leeuwen and van der Welle are asymptotically optimal, whereas several other methods, including one proposed by Fenn and advocated by Dijkstra, are slower than the best methods.