Union-Find

- naïve linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context
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. GALLER 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.
Disjoint-sets data type: applications

Applications.

- Percolation.
- Kruskal’s algorithm.
- Connected components.
- Computing LCAs in trees.
- Computing dominators in digraphs.
- Equivalence of finite state automata.
- Checking flow graphs for reducibility.
- Hoshen–Kopelman algorithm in physics.
- Hinley–Milner polymorphic type inference.
- Morphological attribute openings and closings.
- Matlab’s BW-LABEL function for image processing.
- Compiling EQUVALENCE, DIMENSION and COMMON statements in Fortran.
- ...

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).

**UNION(3, 5)**

![Diagram of disjoint-sets data structure]

- Parent of 3 is 8
- Root
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)**
Disjoint-sets data structure

Array representation. Represent each set as a tree of elements.

- Allocate an array `parent[]` of length `n`.  
  - `parent[i] = j` means parent of element `i` is element `j`.

```
0  1  2  3  4  5  6  7  8  9
parent[]  7  5  7  8  8  7  5  7  8  8
```

Note. For brevity, we suppress arrows and self-loops in figures.
Naïve linking. Link root of first tree to root of second tree.

\textsc{union}(5, 3)
Naïve linking

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

\textsc{Union}(5, 3)
Naïve linking

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

**MAKE-SET**\(_{(x)}\)

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

**UNION**\(_{(x, y)}\)

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

**FIND**\(_{(x)}\)

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

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

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

\begin{itemize}
  \item \textit{height} = 2
  \item \textit{height} = 3
  \item \textit{height} = $n - 1$
\end{itemize}
Union–Find

- naïve 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).

\textbf{Union}(5, 3)

\begin{itemize}
  \item \textbf{size} = 4
  \begin{itemize}
    \item 8
    \begin{itemize}
      \item 9
      \begin{itemize}
        \item 4
        \begin{itemize}
          \item 3
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}

  \item \textbf{size} = 6
  \begin{itemize}
    \item 7
    \begin{itemize}
      \item 0
      \begin{itemize}
        \item 5
        \begin{itemize}
          \item 1
          \begin{itemize}
            \item 6
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}
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).

\textbf{Union}(5, 3)

![Diagram of trees with node sizes and Union operation]

size = 10
**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**\(_x\) \[
\]  

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

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

---

**FIND**\(_x\) \[
\]

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

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

**RETURN** \(_x\).

---

**UNION**\(_x, y\) \[
\]

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

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

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

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

\[\text{parent}[s] \leftarrow r.\]

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

**ELSE**

\[\text{parent}[r] \leftarrow s.\]

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

---

**MAKE-SET**\(_x\) \[
\]

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

---

**UNION**\(_x, y\) \[
\]

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

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

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

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

\[\text{parent}[s] \leftarrow r.\]

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

**ELSE**

\[\text{parent}[r] \leftarrow s.\]

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

---

**MAKE-SET**\(_x\) \[
\]

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

---

**UNION**\(_x, y\) \[
\]

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

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

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

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

\[\text{parent}[s] \leftarrow r.\]

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

**ELSE**

\[\text{parent}[r] \leftarrow s.\]

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

---
**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.** [$\text{height}(r) > \text{height}(s)$] \( \text{size}'[r] > \text{size}[r] \)

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

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

---

**Diagram:**

- Size = 8 (Height = 2)
- Size = 3 (Height = 1)
Link-by-size: analysis

**Property.** Using link-by-size, for every root node $r$: $size[r] \geq 2^{\text{height}(r)}$.

**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 2.** [ $\text{height}(r) \leq \text{height}(s)$ ]

\[
size'[r] = size[r] + size[s] \\
\geq 2 \cdot size[s] \quad \text{link-by-size} \\
\geq 2 \cdot 2^{\text{height}(s)} \quad \text{inductive hypothesis} \\
= 2^{\text{height}(s) + 1} \\
= 2^{\text{height}(r)}.
\]

![Tree diagram](attachment:image.png)
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 $\leq \lceil \log n \rceil$. □

\[ \lg n = \log_2 n \]

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 $= \lg n$.

**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 $k$. □
Section 5.1.4

Union–Find

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

![Diagram showing Union(5, 3) with ranks 1 and 2.](image)

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

![Diagram]

**Note.** For now, rank = height.
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 (\(x \neq \text{parent}[x]\))
\(x \leftarrow \text{parent}[x]\).
RETURN \(x\).

**UNION**(*x*, *y*)

\(r \leftarrow \text{FIND}(x)\).
\(s \leftarrow \text{FIND}(y)\).

IF \((r = s)\) RETURN.
ELSE IF (rank[\(r\)] > rank[\(s\)])
\(\text{parent}[s] \leftarrow r\).
ELSE IF (rank[\(r\)] < rank[\(s\)])
\(\text{parent}[r] \leftarrow s\).
ELSE
\(\text{parent}[r] \leftarrow s\).
rank[\(s\)] ← rank[\(s\)] + 1.
**Link-by-rank: properties**

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

**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 $\text{rank}[x]$ will never change again.

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

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

**Pf.** The parent can change only for a root, so before linking $\text{parent}[x] = x$. After $x$ is linked-by-rank to new root $r$ we have $\text{rank}[r] > \text{rank}[x]$. □
**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
  $\Rightarrow$ resulting tree has $\geq 2^k$ nodes. ■

**PROPERTY 5.** The highest rank of a node is $\leq \lfloor \lg n \rfloor$.

**Pf.** Immediate from PROPERTY 1 and PROPERTY 4. ■
**PROPERTY 6.** For any integer \( k \geq 0 \), there are \( \leq n / 2^k \) nodes with rank \( k \).

**Pf.**

- Any root node of rank \( k \) has \( \geq 2^k \) descendants. [PROPERTY 4]
- Any nonroot node of rank \( k \) has \( \geq 2^k \) descendants because:
  - it had this property just before it became a nonroot [PROPERTY 4]
  - its rank doesn’t change once it became a nonroot [PROPERTY 2]
  - its set of descendants doesn’t change once it became a nonroot
- Different nodes of rank \( k \) can’t have common descendants. [PROPERTY 1]
Theorem. Using link-by-rank, any \textsc{Union} or \textsc{Find} operation takes $O(\log n)$ time in the worst case, where $n$ is the number of elements.

Pf.

\begin{itemize}
  \item The running time of \textsc{Union} and \textsc{Find} is bounded by the tree height.
  \item By \textsc{Property 5}, the height is $\leq \lfloor \lg n \rfloor$.
\end{itemize}
Section 5.1.4

Union–Find

- naïve linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context
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. 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. 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. 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. 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. 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 $r$.

```
FIND(x)
IF (x ≠ parent[x])
    parent[x] ← FIND(parent[x]).
RETURN parent[x].
```

Note. Path compression does not change the rank of a node; so $height(x) ≤ rank[x]$ 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), \ldots, 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).
Section 5.1.4

**Union-Find**

- naive 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.

**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 \( \text{rank}[x] < \text{rank}[\text{parent}[x]] \).

**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 \( \text{rank}[\text{parent}[x]] \) 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 \( k \geq 0 \), 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)
**Property 1.** If \( x \) is not a root node, then \( \text{rank}[x] < \text{rank}[	ext{parent}[x]] \).

*Proof.* 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 \( \text{rank}[	ext{parent}[x]] \) strictly increases.
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>

*iterated lg function*

**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} \}
- \ldots

**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
  \[
  \frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}
  \]
- 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-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.
**Union-Find**

- naïve linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context
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.
Theorem. [Hopcroft–Ullman 1973] 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 $O(m \log^* n)$ time.
Theorem. [Tarjan 1975] 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 $O(m \alpha(m, n))$ time, where $\alpha(m, n)$ is a functional inverse of the Ackermann function.

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: $FIND(x)$ computes the name of the (unique) set containing element $x$. $UNION(A, B, C)$ combines sets $A$ and $B$ into a new set named $C$. A known algorithm for implementing sequences of these instructions is examined. It is shown that, if $t(m, n)$ is the maximum time required by a sequence of $m \geq n$ $FIND$s and $n - 1$ intermixed $UNION$s, then $k_1m\alpha(m, n) \leq t(m, n) \leq k_2m\alpha(m, n)$ for some positive constants $k_1$ and $k_2$, where $\alpha(m, n)$ is related to a functional inverse of Ackermann’s function and is very slow-growing.
Ackermann function

**Ackermann function.** [Ackermann 1928] A computable function that is not primitive recursive.

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

---

**Zum Hilbertschen Aufbau der reellen Zahlen.**

Von

Wilhelm Ackermann in Göttingen.

Um den Beweis für die von Cantor aufgestellte Vermutung zu erbringen, daß sich die Menge der reellen Zahlen, d. h. der zah lentheoretischen Funktionen, mit Hilfe der Zahlen der zweiten Zahlklasse auszählen läßt, benutzt Hilbert einen speziellen Aufbau der zah lentheoretischen Funktionen. Wesentlich bei diesem Aufbau ist der Begriff des Typs einer Funktion. Eine Funktion vom Typ 1 ist eine solche, deren Argumente und Werte ganze Zahlen sind, also eine gewöhnliche zah lentheoretische Funktion. Die Funktionen vom Typ 2 sind die Funktionenfunktionen. Eine derartige Funktion ordnet jeder zah lentheoretischen Funktion eine Zahl zu. Eine Funktion vom Typ 3 ordnet wieder den Funktionenfunktionen Zahlen zu, usw. Die Definition der Typen läßt sich auch ins Transfinite fortsetzen, für den Gegenstand dieser Arbeit ist das aber nicht von Belang.

---

**Note.** There are many inequivalent definitions.
Ackermann function

**Ackermann function.** [Ackermann 1928] A computable function that is not primitive recursive.

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

**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.**

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

**Ex.**

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

\[
2 \uparrow 65536 = 2^{2 \uparrow \ldots \uparrow 2^{65536}} \text{ (65536 times)}
\]

|   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | 11  | 12  | 13  | 14  | 15  | 16  | 17  | 18  | 19  | 20  | 21  | 22  | 23  | 24  | 25  | 26  | 27  | 28  | 29  | 30  | 31  | 32  | 33  | 34  | 35  | 36  | 37  | 38  | 39  | 40  | 41  | 42  | 43  | 44  | 45  | 46  | 47  | 48  | 49  | 50  | 51  | 52  | 53  |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| \(\alpha_1(n)\) | 1   | 1   | 2   | 2   | 3   | 3   | 4   | 4   | 5   | 5   | 5   | 6   | 6   | 7   | 7   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   |
| \(\alpha_2(n)\) | 0   | 1   | 2   | 2   | 3   | 3   | 3   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   |
| \(\alpha_3(n)\) | 0   | 1   | 2   | 2   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   |
| \(\alpha_4(n)\) | 0   | 1   | 2   | 2   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   |
Inverse Ackermann function

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

Property. For every \( n \geq 5 \), the sequence \( \alpha_1(n), \alpha_2(n), \alpha_3(n), \ldots \) converges to 3.

Ex. \([n = 9876!]\) \( \alpha_1(n) \geq 10^{35163}, \alpha_2(n) = 116812, \alpha_3(n) = 6, \alpha_4(n) = 4, \alpha_5(n) = 3. \)

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

Ex. \( \alpha(9876!) = 5. \)

Two-parameter inverse Ackermann. \( \alpha(m, n) = \min \{ k : \alpha_k(n) \leq 3 + m / n \} \).
A tight lower bound

**Theorem.** [Fredman–Saks 1989] In the worst case, any \textsc{Cell-Probe}(\log n) algorithm requires \(\Omega(m \alpha(m, n))\) time to perform an intermixed sequence of \(m\) \textsc{Make-Set}, \textsc{Union}, and \textsc{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.

---

The Cell Probe Complexity of Dynamic Data Structures

Michael L. Fredman
Bellcore and
U.C. San Diego

Michael E. Saks
U.C. San Diego,
Bellcore and
Rutgers University

1. Summary of Results

Dynamic data structure problems involve the representation of data in memory in such a way as to permit certain types of modifications of the data (updates) and certain types of questions about the data (queries). This paradigm encompasses many fundamental problems in computer science.

The purpose of this paper is to prove new lower and upper bounds on the time per operation to implement solutions to some familiar dynamic data structure problems including list representation, subset ranking, partial sums, and the set union problem. The main features of our lower bounds are:

1. They hold in the cell probe model of computation (A. Yao

---

register size from \(\log n\) to \(\log \log n\) only reduces the time complexity by a constant factor. On the other hand, decreasing the register size from \(\log n\) to 1 increases time complexity by a \(\log n\) factor for one of the problems we consider and only a \(\log \log n\) factor for some other problems.

The first two specific data structure problems for which we obtain bounds are:

**List Representation.** This problem concerns the representation of an ordered list of at most \(n\) (not necessarily distinct) elements from the universe \(U = \{1, 2, \ldots, n\}\). The operations to be supported are \textsc{report}(\(k\)), which returns the \(k\)th element of the list, \textsc{insert}(\(k, u\)) which inserts element \(u\) into the list between the
Path compaction variants

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

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-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 Weide are asymptotically optimal, whereas several other methods, including one proposed by Rem and advocated by Dijkstra, are slower than the best methods.