Union-Find Data Structures

Carola Wenk

Slides courtesy of Charles Leiserson with small changes by Carola Wenk
Disjoint-set data structure (Union-Find)

Problem:
• Maintain a dynamic collection of *pairwise-disjoint* sets $S = \{S_1, S_2, \ldots, S_r\}$.
• Each set $S_i$ has one element distinguished as the representative element, $rep[S_i]$.
• Must support 3 operations:
  • **MAKE-SET**($x$): adds new set $\{x\}$ to $S$ with $rep[\{x\}] = x$ (for any $x \notin S_i$ for all $i$)
  • **UNION**($x$, $y$): replaces sets $S_x$, $S_y$ with $S_x \cup S_y$ in $S$ (for any $x$, $y$ in distinct sets $S_x$, $S_y$)
  • **FIND-SET**($x$): returns representative $rep[S_x]$ of set $S_x$ containing element $x$
Union-Find Example

The representative is underlined

\[
\begin{align*}
\text{MAKE-SET}(2) & \quad S = \{\}\quad S = \{\{2\}\} \\
\text{MAKE-SET}(3) & \quad S = \{\{2\}, \{3\}\} \\
\text{MAKE-SET}(4) & \quad S = \{\{2\}, \{3\}, \{4\}\} \\
\text{FIND-SET}(4) = 4 & \quad S = \{\{2, 4\}, \{3\}\} \\
\text{UNION}(2, 4) & \quad S = \{\{2, 4\}, \{3\}\} \\
\text{FIND-SET}(4) = 2 & \quad S = \{\{2, 4\}, \{3\}, \{5\}\} \\
\text{MAKE-SET}(5) & \quad S = \{\{2, 4\}, \{3\}, \{5\}\} \\
\text{UNION}(4, 5) & \quad S = \{\{2, 4, 5\}, \{3\}\} 
\end{align*}
\]
Application: Dynamic connectivity

Suppose a graph is given to us incrementally by

- **ADD-VERTEX**(ν)
- **ADD-EDGE**(u, ν)

and we want to support connectivity queries:

- **CONNECTED**(u, ν):
  Are u and ν in the same connected component?

For example, we want to maintain a spanning forest, so we check whether each new edge connects a previously disconnected pair of vertices.
Application: Dynamic connectivity

Sets of vertices represent connected components. Suppose a graph is given to us incrementally by

- **Add-Vertex**(v) : **Make-Set**(v)
- **Add-Edge**(u, v) : if not **Connected**(u, v) then **Union**(u, v)

and we want to support connectivity queries:

- **Connected**(u, v): **Find-Set**(u) = **Find-Set**(v)
  Are u and v in the same connected component?

For example, we want to maintain a spanning forest, so we check whether each new edge connects a previously disconnected pair of vertices.
Disjoint-set data structure (Union-Find) II

• In all operations pointers to the elements $x, y$ in the data structure are given.

• Hence, we do not need to first search for the element in the data structure.

• Let $n$ denote the overall number of elements (equivalently, the number of MAKE-SET operations).
Simple linked-list solution

Store each set \( S_i = \{x_1, x_2, \ldots, x_k\} \) as an (unordered) doubly linked list. Define representative element \( rep[S_i] \) to be the front of the list, \( x_1 \).

\[
S_i: \quad x_1 \quad x_2 \quad \cdots \quad x_k
\]

\( rep[S_i] \)

- \( \Theta(1) \) • **MAKE-SET** \((x)\) initializes \( x \) as a lone node.
- \( \Theta(n) \) • **FIND-SET** \((x)\) walks left in the list containing \( x \) until it reaches the front of the list.
- \( \Theta(n) \) • **UNION** \((x, y)\) calls **FIND-SET** on \( y \), finds the last element of list \( x \), and concatenates both lists, leaving \( rep \) as **FIND-SET**\([x]\).
Simple balanced-tree solution

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as a balanced tree (ignoring keys). Define representative element $rep[S_i]$ to be the root of the tree.

- **MAKE-SET($x$)** initializes $x$ as a lone node. $\Theta(1)$
- **FIND-SET($x$)** walks up the tree containing $x$ until reaching root. $\Theta(\log n)$
- **UNION($x$, $y$)** calls **FIND-SET** on $y$, finds a leaf of $x$ and concatenates both trees, changing rep. of $y$. $\Theta(\log n)$

How?

$S_i = \{x_1, x_2, x_3, x_4, x_5\}$
Plan of attack

• We will build a simple disjoint-union data structure that, in an *amortized sense*, performs significantly better than $\Theta(\log n)$ per op., even better than $\Theta(\log \log n)$, $\Theta(\log \log \log n)$, ..., but not quite $\Theta(1)$.

• To reach this goal, we will introduce two key *tricks*. Each trick converts a trivial $\Theta(n)$ solution into a simple $\Theta(\log n)$ amortized solution. Together, the two tricks yield a much better solution.

• First trick arises in an augmented linked list. Second trick arises in a tree structure.
Augmented linked-list solution

Store $S_i = \{x_1, x_2, \ldots, x_k\}$ as unordered doubly linked list.

**Augmentation:** Each element $x_j$ also stores pointer $\text{rep}[x_j]$ to $\text{rep}[S_i]$ (which is the front of the list, $x_1$).

- $\text{FIND-SET}(x)$ returns $\text{rep}[x]$. $\Theta(1)$
- $\text{UNION}(x, y)$ concatenates lists containing $x$ and $y$ and updates the $\text{rep}$ pointers for all elements in the list containing $y$. $\Theta(n)$
Example of augmented linked-list solution

Each element $x_j$ stores pointer $rep[x_j]$ to $rep[S_i]$. 

$\text{UNION}(x, y)$

- concatenates the lists containing $x$ and $y$, and
- updates the $rep$ pointers for all elements in the list containing $y$.

$S_x :$ 

\begin{align*}
\cdots & \quad x_1 \quad \cdots \\
& \quad rep[S_x] \quad \cdots
\end{align*}

$S_y :$

\begin{align*}
\cdots & \quad y_1 \quad \cdots \\
& \quad rep[S_y] \quad \cdots
\end{align*}
Example of augmented linked-list solution

Each element $x_j$ stores pointer $rep[x_j]$ to $rep[S_i]$.

**UNION(x, y)**

- concatenates the lists containing $x$ and $y$, and
- updates the $rep$ pointers for all elements in the list containing $y$.

$S_x \cup S_y$:

```
<table>
<thead>
<tr>
<th>x_1</th>
<th>rep[S_x]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>y_1</th>
<th>y_2</th>
<th>y_3</th>
<th>rep[S_y]</th>
</tr>
</thead>
</table>
```
Example of augmented linked-list solution

Each element $x_j$ stores pointer $rep[x_j]$ to $rep[S_i]$. 

**UNION(x, y)**

- concatenates the lists containing $x$ and $y$, and
- updates the $rep$ pointers for all elements in the list containing $y$.

$S_x \cup S_y$

![Diagram showing augmented linked-list solution with elements $x_1$, $x_2$, $y_1$, $y_2$, $y_3$ and $rep[S_x \cup S_y]$]
Alternative concatenation

\textsc{Union}(x, y) \textit{could instead}

- concatenate the lists containing \( y \) and \( x \), and
- update the \textit{rep} pointers for all elements in the list containing \( x \).
Alternative concatenation

\textsc{Union}(x, y) could instead

- concatenate the lists containing \textit{y} and \textit{x}, and
- update the \textit{rep} pointers for all elements in the list containing \textit{x}. 

\begin{align*}
S_x \cup S_y : \\
x_1 &\leftarrow rep[S_y] \\
y_1 &\leftarrow \text{rep} \\
y_2 &\leftarrow \text{rep} \\
y_3 &\text{rep} \\
x_2 &\leftarrow \text{rep}[S_x]
\end{align*}
Alternative concatenation

\textsc{Union}(x, y) \text{ could instead}
\begin{itemize}
\item concatenate the lists containing y and x, and
\item update the \textit{rep} pointers for all elements in the list containing x.
\end{itemize}
**Trick 1: Smaller into larger**
*(weighted-union heuristic)*

To save work, concatenate the smaller list onto the end of the larger list. Cost = $\Theta$(length of smaller list).

Augment list to store its **weight** (# elements).

- Let $n$ denote the overall number of elements (equivalently, the number of **MAKE-SET** operations).
- Let $m$ denote the total number of operations.
- Let $f$ denote the number of **FIND-SET** operations.

**Theorem:** Cost of all **UNION**’s is $O(n \log n)$.

**Corollary:** Total cost is $O(m + n \log n)$. 
Analysis of Trick 1
(weighted-union heuristic)

**Theorem:** Total cost of UNION’s is $O(n \log n)$.

**Proof.** • Monitor an element $x$ and set $S_x$ containing it.
• After initial MAKE-SET$(x)$, $\text{weight}[S_x] = 1$.
• Each time $S_x$ is united with $S_y$:
  • if $\text{weight}[S_y] \geq \text{weight}[S_x]$:  
    – pay 1 to update $\text{rep}[x]$, and  
    – $\text{weight}[S_x]$ at least doubles (increases by $\text{weight}[S_y]$).
  • if $\text{weight}[S_y] < \text{weight}[S_x]$:  
    – pay nothing, and  
    – $\text{weight}[S_x]$ only increases.
Thus pay $\leq \log n$ for $x$. 
Disjoint set forest: Representing sets as trees

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an unordered, potentially unbalanced, not necessarily binary tree, storing only parent pointers. $rep[S_i]$ is the tree root.

- **MAKE-SET(x)** initializes $x$ as a lone node. $- \Theta(1)$
- **FIND-SET(x)** walks up the tree containing $x$ until it reaches the root. $- \Theta(depth[x])$
- **UNION(x, y)** calls **FIND-SET** twice and concatenates the trees containing $x$ and $y$… $- \Theta(depth[x])$
Trick 1 adapted to trees

- **UNION**\((x, y)\) can use a simple concatenation strategy: Make root **FIND-SET**\((y)\) a child of root **FIND-SET**\((x)\).
  \[ \Rightarrow \text{FIND-SET}(y) = \text{FIND-SET}(x). \]

- Adapt Trick 1 to this context:

  **Union-by-weight:**
  Merge tree with smaller weight into tree with larger weight.

- Variant of Trick 1 (see book):

  **Union-by-rank:**
  rank of a tree = its height
Trick 1 adapted to trees (union-by-weight)

• Height of tree is logarithmic in weight, because:
  • Induction on \( n \)
  • Height of a tree \( T \) is determined by the two subtrees \( T_1, T_2 \) that \( T \) has been united from.
  • Inductively the heights of \( T_1, T_2 \) are the logs of their weights.
  • If \( T_1 \) and \( T_2 \) have different heights:
    \[
    \text{height}(T) = \max(\text{height}(T_1), \text{height}(T_2)) = \max(\log \text{weight}(T_1), \log \text{weight}(T_2)) < \log \text{weight}(T)
    \]
  • If \( T_1 \) and \( T_2 \) have the same heights:
    (Assume \( 2 \leq \text{weight}(T_1) < \text{weight}(T_2) \))
    \[
    \text{height}(T) = \text{height}(T_1) + 1 = \log (2 \times \text{weight}(T_1)) \leq \log \text{weight}(T)
    \]
• Thus the total cost of any \( m \) operations is \( O(m \log n) \).
**Trick 2: Path compression**

When we execute a `FIND-SET` operation and walk up a path \( p \) to the root, we know the representative for all the nodes on path \( p \).

*Path compression* makes all of those nodes direct children of the root.

Cost of `FIND-SET(x)` is still \( \Theta(depth[x]) \).
Trick 2: Path compression

When we execute a \texttt{FIND-SET} operation and walk up a path \textit{p} to the root, we know the representative for all the nodes on path \textit{p}.

\textit{Path compression} makes all of those nodes direct children of the root.

Cost of \texttt{FIND-SET}(x) is still $\Theta(\text{depth}[x])$. 

\texttt{FIND-SET}(y_2)
Trick 2: Path compression

When we execute a FIND-SET operation and walk up a path \( p \) to the root, we know the representative for all the nodes on path \( p \).

Path compression makes all of those nodes direct children of the root.

Cost of FIND-SET(\( x \)) is still \( \Theta(\text{depth}[x]) \).

\[
\text{FIND-SET}(y_2)
\]
Trick 2: Path compression

• Note that UNION($x,y$) first calls FIND-SET($x$) and FIND-SET($y$). Therefore path compression also affects UNION operations.
Analysis of Trick 2 alone

**Theorem:** Total cost of FIND-SET’s is $O(m \log n)$.

**Proof:** By amortization. Omitted.
Ackermann’s function $A$, and its “inverse” $\alpha$

Define $A_k(j) = \left\{ \begin{array}{ll}
  j + 1 & \text{if } k = 0, \\
  A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1.
\end{array} \right.$ – iterate $j+1$ times

$A_0(j) = j + 1$

$A_1(j) \sim 2^j$

$A_2(j) \sim 2^j 2^j > 2^j$

$A_3(j) > 2^{2^{2^j}}$

$A_4(j)$ is a lot bigger.

Define $\alpha(n) = \min \{ k : A_k(1) \geq n \} \leq 4$ for practical $n$. 

$A_0(1) = 2$

$A_1(1) = 3$

$A_2(1) = 7$

$A_3(1) = 2047$

$A_4(1) > 2^{2^{2^{2^{2047}}}}$
Analysis of Tricks 1 + 2 for disjoint-set forests

**Theorem:** In general, total cost is $O(m \, \alpha(n))$.  

*(long, tricky proof – see Section 21.4 of CLRS)*