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:
  • \textsc{Make-Set}(x): adds new set \( \{x\} \) to \( S \) with \( rep[\{x\}] = x \) (for any \( x \notin S_i \) for all \( i \))
  • \textsc{Union}(x, y): replaces sets \( S_x, S_y \) with \( S_x \cup S_y \) in \( S \) (for any \( x, y \) in distinct sets \( S_x, S_y \))
  • \textsc{Find-Set}(x): returns representative \( rep[S_x] \) of set \( S_x \) containing element \( x \)
Union-Find Example

MAKE-SET(2)
S = {}  
MAKE-SET(3)
S = {{2}}  
MAKE-SET(4)
S = {{2}, {3}}  
FIND-SET(4) = 4
S = {{2}, {3}, {4}}  
UNION(2, 4)
S = {{2, 4}, {3}}  
FIND-SET(4) = 2
S = {{2, 4}, {3}}  
MAKE-SET(5)
S = {{2, 4}, {3}, {5}}  
UNION(4, 5)
S = {{2, 4, 5}, {3}}  

The representative is underlined.
Plan of attack

• We will build a simple disjoint-set 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 $rep[x_j]$ to $rep[S_i]$ (which is the front of the list, $x_1$).

Assume pointer to $x$ is given.

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

Each element $x_j$ stores pointer $\text{rep}[x_j]$ to $\text{rep}[S_i]$.

$\text{UNION}(x, y)$
- concatenates the lists containing $x$ and $y$, and
- updates the $\text{rep}$ pointers for all elements in the list containing $y$. 

\[ S_x : \begin{align*}
\text{rep} & \quad x_1 \quad \text{rep}[S_x] \\
x_2 \quad & \end{align*} \]

\[ S_y : \begin{align*}
\text{rep} & \quad y_1 \\
y_2 \quad & \quad y_3 \\
\text{rep}[S_y] & \end{align*} \]
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 $y$.

$S_x \cup S_y$:

```
rep

rep[$S_x$]

x1 --- x2

rep[$S_y$]

y1 --- y2 --- y3
```
Example of augmented linked-list solution

Each element \( x_j \) stores pointer \( rep[x_j] \) to \( rep[S_i] \).

\textbf{\textsc{Union}(x, y)}

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

\[ S_x \cup S_y : \]

\[ rep[S_x \cup S_y] \]

\[ \begin{array}{c}
  x_1 \\
  x_2 \\
  \end{array} \quad \begin{array}{c}
  y_1 \\
  y_2 \\
  y_3 \\
  \end{array} \quad \text{rep} \]
Alternative concatenation

\texttt{UNION}(x, y) 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 \( y \) and \( x \), and
- update the \textit{rep} pointers for all elements in the list containing \( x \).

\[ S_x \cup S_y : \]

\[ \text{rep} \]

\[ S_y \]

\[ \text{rep} \]

\[ S_x \]

\[ \text{rep} \]

\[ x_1 \rightarrow x_2 \]

\[ y_1 \rightarrow y_2 \rightarrow y_3 \]
Alternative concatenation

\textbf{UNION}(x, y) could instead

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

\[
S_x \cup S_y : \quad \text{rep}[S_x \cup S_y]
\]

\[
\text{rep}
\]
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($x$), $weight[S_x] = 1$.
• Each time $S_x$ is united with $S_y$:
  • if $weight[S_y] \geq weight[S_x]$:
    – pay 1 to update $rep[x]$, and
    – $weight[S_x]$ at least doubles (increases by $weight[S_y]$).
  • if $weight[S_y] < weight[S_x]$:
    – pay nothing, and
    – $weight[S_x]$ 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

• \textbf{Union-by-weight:} Merge tree with smaller weight into tree with larger weight.

• Variant of Trick 1 (see book): \textbf{Union-by-rank:} rank of a tree = its height

Example: \textbf{Union}((x_4, y_2))
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 \texttt{FIND-SET} operation and walk up a path $p$ to the root, we know the representative for all the nodes on path $p$.

\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 $\Theta(\text{depth}[x])$. 

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 $\Theta(\text{depth}[x])$. 

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.
Analysis of Tricks 1 + 2 for disjoint-set forests

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

**Proof:** Long, tricky proof by amortization. Omitted. See book for a proof sketch for $O(m \log^*(n))$ runtime.
Ackermann’s function $A$, and it’s “inverse” $\alpha$

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

$A_0(j) = j + 1$  \hspace{1cm} $A_0(1) = 2$
$A_1(j) \sim 2^j$ \hspace{1cm} $A_1(1) = 3$
$A_2(j) \sim 2^j 2^j > 2^j$ \hspace{1cm} $A_2(1) = 7$

$A_3(j) > 2^{2^j} \cdots 2^{2^j}$ \hspace{1cm} $A_3(1) = 2047$
$A_4(j)$ is a lot bigger. \hspace{1cm} $A_4(1) > 2^{2^{2^{2^{2047}}}}$

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

11/11/15

CMPS 2200 Intro. to Algorithms