CSE 326: Data Structures

Disjoint Sets

Neva Cherniavsky
Summer 2006

Equivalence Relations

Relation $R$:
• For every pair of elements $(a, b)$ in a set $S$, $a R b$ is either true or false.
• If $a R b$ is true, then $a$ is related to $b$.
An equivalence relation satisfies:
1. (Reflexive) $a R a$
2. (Symmetric) $a R b$ iff $b R a$
3. (Transitive) $a R b$ and $b R c$ implies $a R c$

Examples of Equivalence Relations

• $\geq$: Is it reflexive, symmetric, and transitive?
• Electrical connectivity: Is it reflexive, symmetric, and transitive?
• Two cities in the same country: Is it reflexive, symmetric, and transitive?

Determining Equivalence Classes

• Divide set $S$ into subsets containing items related to each other
  › $\{\text{Paris, Lyon}\}$, $\{\text{Seattle, New York, Boston}\}$, $\{\text{London}\}$, $\{\text{Bombay, Calcutta}\}$
• Given the set, how do we determine these classes?
  › $\{\text{Paris}\}$, $\{\text{Lyon}\}$, $\{\text{Seattle}\}$, $\{\text{New York}\}$, $\{\text{Boston}\}$, $\{\text{London}\}$, $\{\text{Bombay}\}$, $\{\text{Calcutta}\}$

Solution: Union/Find

Algorithm:
• Start with sets $S_0, S_1, S_2, \ldots, S_k$
• Check: is $S_0$ related to $S_i$? (Does find return the same value?)
• If so, perform union
Applications:
• Graph theory problems (project phase C)
• Compiler checking type relations

Disjoint Union - Find

• Maintain a set of pairwise disjoint sets.
  › $\{\text{Paris}\}$, $\{\text{Lyon}\}$, $\{\text{Seattle, New York}\}$, $\{\text{Boston}\}$, $\{\text{London}\}$, $\{\text{Bombay, Calcutta}\}$
  › $\{3,5,7\}$, $\{4,2,8\}$, $\{9\}$, $\{1,6\}$
• Each set has a unique name, one of its members
  › $\{\text{Paris}\}$, $\{\text{Lyon}\}$, $\{\text{Seattle, New York}\}$, $\{\text{Boston}\}$, $\{\text{London}\}$, $\{\text{Bombay, Calcutta}\}$
  › $\{3,5,7\}$, $\{4,2,8\}$, $\{9\}$, $\{1,6\}$
Union

- Union(x, y) – take the union of two sets named x and y
  - {3, 5, 7}, {4, 2, 8}, {9}, {1, 6}
  - Union(5, 1)
    - {3, 5, 7, 1, 6}, {4, 2, 8}, {9}.

Find

- Find(x) – return the name of the set containing x.
  - {3, 5, 7, 1, 6}, {4, 2, 8}, {9}, {1, 6}
  - Find(1) = 5
  - Find(4) = 8

Example

<table>
<thead>
<tr>
<th>S</th>
<th>Union(7, 20)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1, 2, 8, 9, 13, 19}</td>
<td>( \text{Find}(8) = 7 )</td>
</tr>
<tr>
<td>{3}</td>
<td>( \text{Find}(14) = 20 )</td>
</tr>
<tr>
<td>{5}</td>
<td>( \text{Union}(7, 20) )</td>
</tr>
<tr>
<td>{6}</td>
<td>-</td>
</tr>
<tr>
<td>{11, 17}</td>
<td>-</td>
</tr>
<tr>
<td>{14, 20, 26, 27}</td>
<td>-</td>
</tr>
<tr>
<td>{15, 16, 21}</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>{22, 23, 24, 29, 39, 32}</td>
<td>-</td>
</tr>
<tr>
<td>{33, 34, 35, 36}</td>
<td>-</td>
</tr>
</tbody>
</table>

Implementing the DS ADT

- \( n \) elements.
  - Total Cost of: \( m \) finds, \( \leq n - 1 \) unions

  - Target complexity: \( O(m+n) \)
    - i.e. \( O(1) \) amortized

  - \( O(1) \) worst-case for find as well as union would be great, but...

    Known result: both find and union cannot be done in worst-case \( O(1) \) time

Up-Tree for DU/F

- Initial state: 1 2 3 4 5 6 7
  - Intermediate state: 1 3 7 2 5 4

  - Roots are the names of each set.

Find Operation

- \( \text{Find}(x) \) follow \( x \) to the root and return the root

\( \text{Find}(6) = 7 \)
Union Operation

• Union(i,j) - assuming i and j roots, point i to j.

Simple Implementation

• Array of indices

1 2 3 4 5 6 7

Union(1,7)

1 2 3 4
5 6 7

Union(up[], x, y)

//precondition: x and y are roots/
Up[x] := y

Constant Time!

Exercise

• Design Find operator
  › Recursive version
  › Iterative version

Find(up[], x)

//precondition: x is in the range 1 to size//
???

A Bad Case

1 2 3 4 5 6 7

Union(1,2)

Union(2,3)

Union(3,4)

Union(4,5)

Union(5,6)

Union(6,7)

1 2 3 4 5 6 7

Find(1) n steps!!

Now this doesn’t look good 😞

Can we do better? Yes!

1. Improve union so that find only takes \(O(\log n)\)
   • Union-by-size
   • Reduces complexity to \(O(m \log n + n)\)

2. Improve find so that it becomes even better!
   • Path compression
   • Reduces complexity to almost \(O(m + n)\)
Weighted Union

- Weighted Union
  - Always point the smaller tree to the root of the larger tree

Example Again

Example Again

Analysis of Weighted Union

- With weighted union an up-tree of height $h$ has weight at least $2^h$.
- Proof by induction
  - Basis: $h = 0$. The up-tree has one node, $2^0 = 1$
  - Inductive step: Assume true for all $h' < h$.

Analysis of Weighted Union

- Let $T$ be an up-tree of weight $n$ formed by weighted union. Let $h$ be its height.
- $n \geq 2^h$
- $\log_2 n > h$
- $\text{Find}(x)$ in tree $T$ takes $O(\log n)$ time.
- Can we do better?

Worst Case for Weighted Union

- $n/2$ Weighted Unions
- $n/4$ Weighted Unions

Example of Worst Cast (cont’)

- After $n - 1 = n/2 + n/4 + \ldots + 1$ Weighted Unions

If there are $n = 2^k$ nodes then the longest path from leaf to root has length $k$. 
Elegant Array Implementation

Weighted Union

```c
W-Union(i, j : index)
// i and j are roots/
wi := weight[i];
wj := weight[j];
if wi < wj then
    up[i] := j;
    weight[i] := wi + wj;
else
    up[j] := i;
    weight[j] := wi + wj;
```

Union-by-size: Find Analysis

- Complexity of Find: $O(\max \text{ node depth})$
- All nodes start at depth 0
- Node depth increases:
  - Only when it is part of smaller tree in a union
  - Only by one level at a time
- Result: tree size doubles when node depth increases by 1

Find runtime = $O(\text{node depth}) = \ldots$

runtime for $m$ finds and $n-1$ unions =

Path Compression

- On a Find operation point all the nodes on the search path directly to the root.

Draw the result of Find(e):

Self-Adjustment Works
Path Compression Find

Path Compression Find

\[
\text{PC-Find}(i) \{
\begin{align*}
    r &:= i; \\
    \text{while } \text{up}[r] \neq 0 &\text{ do} \quad //\text{find root//} \\
    r &:= \text{up}[r]; \\
    \text{if } i \neq r &\text{ then} \quad //\text{compress path//} \\
    k &:= \text{up}[i]; \\
    \text{while } k \neq r &\text{ do} \\
    \text{up}[i] &:= r; \\
    i &:= k; \\
    k &:= \text{up}[k]; \\
    \text{return}(r)
\end{align*}
\]

Complex Complexity of Union-by-Size + Path Compression

Tarjan proved that, with these optimizations, \(p\) union and find operations on a set of \(n\) elements have worst case complexity of \(O(p \cdot \alpha(p, n))\).

For all practical purposes this is amortized constant time: \(O(p \cdot 4)\) for \(p\) operations!

\begin{itemize}
    \item Very complex analysis – worse than splay tree analysis etc. that we skipped!
\end{itemize}

Disjoint Union / Find with Weighted Union and PC

\begin{itemize}
    \item Worst case time complexity for a W-Union is \(O(1)\) and for a PC-Find is \(O(\log n)\).
    \item Time complexity for \(m \geq n\) operations on \(n\) elements is \(O(m \log^* n)\) where \(\log^* n\) is a very slow growing function.
    \item \(\log^* n < 7\) for all reasonable \(n\). Essentially constant time per operation!
    \item Using “ranked union” gives an even better bound theoretically.
\end{itemize}

Amortized Complexity

\begin{itemize}
    \item For disjoint union / find with weighted union and path compression.
        \begin{itemize}
            \item average time per operation is essentially a constant.
            \item worst case time for a PC-Find is \(O(\log n)\).
        \end{itemize}
    \item An individual operation can be costly, but over time the average cost per operation is not.
\end{itemize}

Find Solutions

Recursive

\[
\text{Find}(up[]) : \text{integer array, } x : \text{integer}) : \text{integer} \{
\begin{align*}
\text{if } \text{up}[a] = 0 &\text{ then return } x; \\
\text{else return } \text{Find}(up, \text{up}[a]);
\end{align*}
\]

Iterative

\[
\text{Find}(up[]) : \text{integer array, } x : \text{integer}) : \text{integer} \{
\begin{align*}
\text{//precondition: } x \text{ is in the range } 1 \text{ to size//} \\
\text{while } \text{up}[a] \neq 0 &\text{ do} \\
    a &:= \text{up}[a]; \\
\text{return } a;
\end{align*}
\]