CSE 332: Data Structures & Parallelism
Lecture 23: Disjoint Sets

Ruth Anderson
Autumn 2018
Aside: Union-Find aka Disjoint Set ADT

- **Union(x,y)** – take the union of two sets named x and y
  - Given sets: \{3, 5, 7\}, \{4, 2, 8\}, \{9\}, \{1, 6\}
  - **Union(5,1)**
    - Result: \{3, 5, 7, 1, 6\}, \{4, 2, 8\}, \{9\},
  
To perform the union operation, we replace sets x and y by \((x \cup y)\)

- **Find(x)** – return the name of the set containing x.
  - Given sets: \{3, 5, 7, 1, 6\}, \{4, 2, 8\}, \{9\},
  - **Find(1)** returns 5
  - **Find(4)** returns 8

- We can do Union in constant time.
- We can get Find to be amortized constant time
  (worst case O(log n) for an individual Find operation).
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

*can there be more unions?*
Data Structure for the DS ADT

• **Observation:** trees let us find many elements given one root…

• **Idea:** if we reverse the pointers (make them point up from child to parent), we can find a single root from many elements…

• **Idea:** Use one tree for each equivalence class. The name of the class is the tree root.
Up-Tree for Disjoint Union/Find

Initial state: 1 2 3 4 5 6 7

After several Unions:

Roots are the names of each set.
Find Operation

Find(x) - follow x to the root and return the root

Find(6) = 7
Union Operation

Union(x, y) - assuming x and y are roots, point y to x.

Union(1, 7)
Simple Implementation

- Array of indices

<table>
<thead>
<tr>
<th>up</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Up[x] = 0 means x is a root.
Implementation

```c
int Find(int x) {
    while(up[x] != 0) {
        x = up[x];
    }
    return x;
}

void Union(int x, int y) {
    up[y] = x;
}
```

runtime for Union():

runtime for Find():

runtime for m Finds and n-1 Unions:
A Bad Case

Union(x,y) – “point y to x”

Union(2,1)

Union(3,2)

Union(n,n-1)

Find(1) n steps!!
Now this doesn’t look good 😞

Can we do better? Yes!

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

2. Improve find so that it becomes even better!
   - Path compression
   - Reduces complexity to almost \( \Theta(m + n) \)
Weighted Union/Union by Size

- Weighted Union
  - Always point the *smaller* (total # of nodes) tree to the root of the larger tree

![Diagram showing weighted union operation](image)
Example Again

W-Union(2,1)

W-Union(3,2)

W-Union(n,2)

Find(1) constant time
Analysis of Weighted Union

With weighted union an up-tree of height $h$ has weight \textit{at least} $2^h$.

- **Proof by induction**
  - \textbf{Basis}: $h = 0$. The up-tree has one node, $2^0 = 1$
  - \textbf{Inductive step}: Assume true for all $h' < h$.

Minimum weight up-tree of height $h$ formed by weighted unions

$$W(T) \geq W(T_1) + W(T_2) \geq 2^{h-1} + 2^{h-1} = 2^h$$
Analysis of Weighted Union (cont)

Let T be an up-tree of weight n formed by weighted union. Let h be its height.

\[ n \geq 2^h \]
\[ \log_2 n \geq h \]

- 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 \( \frac{n}{2} + \frac{n}{4} + \ldots + 1 \) Weighted Unions:

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

```
  1
  2
  3
  4
  5
  6
  7

table:

up weight
1  2  3  4  5  6  7
-1 1 -1  7  7  5 -1
2  1  4
```
Weighted Union

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

new runtime for Union():

new runtime for Find():

runtime for \( m \) finds and \( n-1 \) unions =
Nifty Storage Trick

• Use the same array representation as before

• Instead of storing $-1$ for the root, simply store $-\text{size}$

[Read section 8.4]
How about Union-by-height?

- Can still guarantee $O(\log n)$ worst case depth

*Left as an exercise!*

- Problem: Union-by-height doesn’t combine very well with the new find optimization technique we’ll see next
Now this doesn’t look good 😞
Can we do better? Yes!

1. DONE: Improve union so that find only takes $\Theta(\log n)$
   - Union-by-size
   - Reduces complexity to $\Theta(m \log n + n)$

2. NOW: Improve find so that it becomes even better!
   - Path compression
   - Reduces complexity to almost $\Theta(m + n)$
Path Compression

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

• On a Find operation point all the nodes on the search path directly to the root.
Draw the result of $\text{Find}(e)$:
Self-Adjustment Works

PC-Find(x)
Path Compression Find

PC-Find(i : index) {
    r := i;
    while up[r] ≠ -1 do //find root//
        r := up[r];
    if i ≠ r then  //compress path//
        k := up[i];
        while k ≠ r do
            up[i] := r;
            i := k;
            k := up[k]
        return(r)
    }
}
int Find(Object x) {
    // x had better be in
    // the set!
    int xID = hTable[x];
    int i = xID;

    // Get the root for
    // this set
    while(up[xID] != -1) {
        xID = up[xID];
    }

    // Change the parent for
    // all nodes along the path
    while(up[i] != -1) {
        temp = up[i];
        up[i] = xID;
        i = temp;
    }

    return xID;
}
Interlude: A Really Slow Function

Ackermann’s function is a really big function $A(x, y)$ with inverse $\alpha(x, y)$ which is really small.

How fast does $\alpha(x, y)$ grow?

$\alpha(x, y) = 4$ for $x$ far larger than the number of atoms in the universe ($2^{300}$)

$\alpha$ shows up in:

– Computation Geometry (surface complexity)
– Combinatorics of sequences
A More Comprehensible Slow Function

\[ \log^* x = \text{number of times you need to compute } \log \text{ to bring value down to at most 1} \]

E.g. \( \log^* 2 = 1 \)
\[ \log^* 4 = \log^* 2^2 = 2 \]
\[ \log^* 16 = \log^* 2^{2^2} = 3 \quad (\log \log \log 16 = 1) \]
\[ \log^* 65536 = \log^* 2^{2^{2^2}} = 4 \quad (\log \log \log \log 65536 = 1) \]
\[ \log^* 2^{65536} = \ldots \ldots \ldots \ldots \ldots = 5 \]

Take this: \( \alpha(m,n) \) grows even slower than \( \log^* n \) ！！
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!

• Very complex analysis
Disjoint Union / Find
with Weighted Union and PC

• Worst case time complexity for a W-Union is $O(1)$ and for a PC-Find is $O(\log n)$.

• Time complexity for $m \geq n$ operations on $n$ elements is $O(m \log^* n)$ where $\log^* n$ is a very slow growing function.
  – $\log^* n < 7$ for all reasonable $n$. Essentially constant time per operation!

• Using “ranked union” gives an even better bound theoretically.
Amortized Complexity

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