Union-Find

CS 214, Fall 2019

The following slides are from Robert Sedgewick and Kevin Wayne’s algorithms & data structures course at Princeton University. The original lectures may be found here.
Union-find abstractions

- Objects.
- Disjoint sets of objects.
- **Find queries:** are two objects in the same set?
- **Union commands:** replace sets containing two items by their union

**Goal.** Design efficient data structure for union-find.

- Find queries and union commands may be intermixed.
- Number of operations $M$ can be huge.
- Number of objects $N$ can be huge.
Quick-find [eager approach]

Data structure.
- Integer array \( id[] \) of size \( N \).
- Interpretation: \( p \) and \( q \) are connected if they have the same id.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( id[i] )</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

5 and 6 are connected
2, 3, 4, and 9 are connected
Quick-find [eager approach]

Data structure.
- Integer array $id[]$ of size $N$.
- Interpretation: $p$ and $q$ are connected if they have the same id.

Find. Check if $p$ and $q$ have the same id.

Union. To merge components containing $p$ and $q$, change all entries with $id[p]$ to $id[q]$.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[i]</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

- 5 and 6 are connected
- 2, 3, 4, and 9 are connected

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[i]</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

- 3 and 6 not connected

- union of 3 and 6
- 2, 3, 4, 5, 6, and 9 are connected

problem: many values can change
Quick-find example

3–4  0 1 2 4 4 5 6 7 8 9
4–9  0 1 2 9 9 5 6 7 8 9
8–0  0 1 2 9 9 5 6 7 0 9
2–3  0 1 9 9 9 5 6 7 0 9
5–6  0 1 9 9 9 6 6 7 0 9
5–9  0 1 9 9 9 9 9 7 0 9
7–3  0 1 9 9 9 9 9 9 0 9
4–8  0 1 0 0 0 0 0 0 0 0
6–1  1 1 1 1 1 1 1 1 1 1

problem: many values can change
Quick-find is too slow

Quick-find algorithm may take \( \sim MN \) steps to process \( M \) union commands on \( N \) objects

Rough standard (for now).
- \( 10^9 \) operations per second.
- \( 10^9 \) words of main memory.
- Touch all words in approximately 1 second.

Ex. Huge problem for quick-find.
- \( 10^{10} \) edges connecting \( 10^9 \) nodes.
- Quick-find takes more than \( 10^{19} \) operations.
- \( 300+ \text{ years} \) of computer time!

Paradoxically, quadratic algorithms get worse with newer equipment.
- New computer may be 10x as fast.
- But, has 10x as much memory so problem may be 10x bigger.
- With quadratic algorithm, takes 10x as long!
Quick-union [lazy approach]

Data structure.
- Integer array $id[]$ of size $N$.
- Interpretation: $id[i]$ is parent of $i$.
- Root of $i$ is $id[id[id[...id[i]...]]]$.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$id[i]$</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

3's root is 9; 5's root is 6.

keep going until it doesn't change
Quick-union [lazy approach]

Data structure.
- Integer array \( id[] \) of size \( N \).
- Interpretation: \( id[i] \) is parent of \( i \).
- Root of \( i \) is \( id[id[id[...id[i]...]]] \).

Find. Check if \( p \) and \( q \) have the same root.

Union. Set the id of \( q \)'s root to the id of \( p \)'s root.

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

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

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

```
0 1 9
2 4 5
3
```

3's root is 9; 5's root is 6
3 and 5 are not connected

```
0 1 9
2 4 5
3
7 8
```

only one value changes

```
0 1 9
2 4 5
p 3 6
q
```

keep going until it doesn't change
Quick-union example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3−4</td>
<td>0 1 2 4 4 5 6 7 8 9</td>
</tr>
<tr>
<td>4−9</td>
<td>0 1 2 4 9 5 6 7 8 9</td>
</tr>
<tr>
<td>8−0</td>
<td>0 1 2 4 9 5 6 7 0 9</td>
</tr>
<tr>
<td>2−3</td>
<td>0 1 9 4 9 5 6 7 0 9</td>
</tr>
<tr>
<td>5−6</td>
<td>0 1 9 4 9 6 6 7 0 9</td>
</tr>
<tr>
<td>5−9</td>
<td>0 1 9 4 9 6 9 7 0 9</td>
</tr>
<tr>
<td>7−3</td>
<td>0 1 9 4 9 6 9 9 0 9</td>
</tr>
<tr>
<td>4−8</td>
<td>0 1 9 4 9 6 9 9 0 0</td>
</tr>
<tr>
<td>6−1</td>
<td>1 1 9 4 9 6 9 9 0 0</td>
</tr>
</tbody>
</table>

Problem: trees can get tall
Quick-union is also too slow

Quick-find defect.
• Union too expensive (N steps).
• Trees are flat, but too expensive to keep them flat.

Quick-union defect.
• Trees can get tall.
• Find too expensive (could be N steps)
• Need to do find to do union

<table>
<thead>
<tr>
<th>algorithm</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick-find</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>Quick-union</td>
<td>N*</td>
<td>N</td>
</tr>
</tbody>
</table>

* includes cost of find

worst case
Improvement 1: Weighting

Weighted quick-union.
- Modify quick-union to avoid tall trees.
- Keep track of size of each component.
- Balance by linking small tree below large one.

**Ex.** Union of 5 and 3.
- Quick union: link 9 to 6.
- Weighted quick union: link 6 to 9.
Weighted quick-union example

3–4  0 1 2 3 3 5 6 7 8 9
4–9  0 1 2 3 3 5 6 7 8 3
8–0  8 1 2 3 3 5 6 7 8 3
2–3  8 1 3 3 3 5 6 7 8 3
5–6  8 1 3 3 3 5 5 7 8 3
5–9  8 1 3 3 3 3 5 7 8 3
7–3  8 1 3 3 3 3 5 3 8 3
4–8  8 1 3 3 3 3 5 3 3 3
6–1  8 3 3 3 3 3 5 3 3 3

no problem: trees stay flat
Weighted quick-union: Java implementation

Java implementation.
- Almost identical to quick-union.
- Maintain extra array \( \text{sz}[i] \) to count number of elements in the tree rooted at \( i \).

Find. Identical to quick-union.

Union. Modify quick-union to
- merge smaller tree into larger tree
- update the \( \text{sz}[i] \) array.

```java
if (sz[i] < sz[j]) {
    id[i] = j;
    sz[j] += sz[i];
} else {
    sz[i] < sz[j] {
    id[j] = i;
    sz[i] += sz[j];
}
```
Weighted quick-union analysis

Analysis.
• Find: takes time proportional to depth of $p$ and $q$.
• Union: takes constant time, given roots.
• Fact: depth is at most $\lg N$. [needs proof]

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Union</th>
<th>Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick-find</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>Quick-union</td>
<td>$N^*$</td>
<td>$N$</td>
</tr>
<tr>
<td>Weighted QU</td>
<td>$\lg N^*$</td>
<td>$\lg N$</td>
</tr>
</tbody>
</table>

* includes cost of find

Stop at guaranteed acceptable performance? No, easy to improve further.
Improvement 2: Path compression

Path compression. Just after computing the root of $i$, set the id of each examined node to $\text{root}(i)$. 
Path compression.

- Standard implementation: add second loop to `root()` to set the id of each examined node to the root.
- Simpler one-pass variant: make every other node in path point to its grandparent.

```java
public int root(int i)
{
    while (i != id[i])
    {
        id[i] = id[id[i]];
        i = id[i];
    }
    return i;
}
```

In practice. No reason not to! Keeps tree almost completely flat.
Weighted quick-union with path compression

no problem: trees stay VERY flat
Theorem. Starting from an empty data structure, any sequence of $M$ union and find operations on $N$ objects takes $O(N + M \lg^* N)$ time.

- Proof is very difficult.
- But the algorithm is still simple!

Linear algorithm?
- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

Amazing fact:
- In theory, no linear linking strategy exists
Summary

Ex. Huge practical problem.
• 10^{10} edges connecting 10^9 nodes.
• WQUPC reduces time from 3,000 years to 1 minute.
• Supercomputer won't help much.
• Good algorithm makes solution possible.

Bottom line.
WQUPC makes it possible to solve problems that could not otherwise be addressed