Union-Find Algorithms

- network connectivity
- quick find
- quick union
- improvements
- applications

Subtext of today's lecture (and this course)

Steps to developing a usable algorithm.
- Define the problem.
- Find an algorithm to solve it.
- Fast enough?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method

Mathematical models and computational complexity

READ Chapter One of Algs in Java

Network connectivity

Basic abstractions
- set of objects
- union command: connect two objects
- find query: is there a path connecting one object to another?
Union-find applications involve manipulating objects of all types.
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Variable name aliases.
- Pixels in a digital photo.
- Metallic sites in a composite system.

When programming, convenient to name them 0 to N-1.
- Hide details not relevant to union-find.
- Integers allow quick access to object-related info.
- Could use symbol table to translate from object names

Connected components

Connected component: set of mutually connected vertices

Each union command reduces by 1 the number of components

<table>
<thead>
<tr>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

3 = 10-7 components

Network connectivity: larger example

find(u, v) ?

Union-find abstractions

Simple model captures the essential nature of connectivity.
- Objects.
- Disjoint sets of objects.
- Find query: are objects 2 and 9 in the same set?
- Union command: merge sets containing 3 and 8.

Connected components

add a connection between two grid points

Connected component: set of mutually connected vertices
 Each union command reduces by 1 the number of components

<table>
<thead>
<tr>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

3 = 10-7 components

7 union commands
Network connectivity: larger example

find(u, v) ?
true

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 \( id[q] \).

Quick-find example

\[
\begin{array}{cccccccccccc}
3-4 & 0 & 1 & 2 & 4 & 5 & 6 & 7 & 8 & 9 \\
4-9 & 0 & 1 & 2 & 9 & 9 & 5 & 6 & 7 & 8 & 9 \\
8-0 & 0 & 1 & 2 & 9 & 9 & 5 & 6 & 7 & 0 & 9 \\
2-3 & 0 & 1 & 9 & 9 & 9 & 6 & 6 & 7 & 0 & 9 \\
5-6 & 0 & 1 & 9 & 9 & 9 & 6 & 6 & 7 & 0 & 9 \\
5-9 & 0 & 1 & 9 & 9 & 9 & 9 & 9 & 7 & 0 & 9 \\
7-3 & 0 & 1 & 9 & 9 & 9 & 9 & 9 & 7 & 0 & 9 \\
4-8 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 \\
6-1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

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+ 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 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.

Quick-union example

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

Problem: trees can get tall
Quick-union: Java implementation

```java
public class QuickUnion {
    private int[] id;
    public QuickUnion(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++) id[i] = i;
    }
    private int root(int i) {
        while (i != id[i]) i = id[i];
        return i;
    }
    public boolean find(int p, int q) {
        return root(p) == root(q);
    }
    public void unite(int p, int q) {
        int i = root(p);
        int j = root(q);
        id[i] = j;
    }
}
```

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>1</td>
</tr>
</tbody>
</table>

* includes cost of find

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.

network connectivity
quick find
quick union
improvements
applications
Weighted quick-union example

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

no problem: trees stay flat

Weighted quick-union: Java implementation

Java implementation.
- Almost identical to quick-union.
- Maintain extra array \( sz[] \) 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 \( sz[] \) array.

\[
\begin{align*}
\text{if } (sz[i] < sz[j]) & \{ id[i] = j; sz[j] += sz[i]; \} \\
\text{else } sz[i] < sz[j] & \{ id[j] = i; sz[i] += sz[j]; \}
\end{align*}
\]

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.

In practice. No reason not to! Keeps tree almost completely flat.

Weighted quick-union with path compression

**Theorem.** Starting from an empty data structure, any sequence of M union and find operations on N objects takes $O(N + M \log^* N)$ 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

WQUPC performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick-find</td>
<td>$M N$</td>
</tr>
<tr>
<td>Quick-union</td>
<td>$M N$</td>
</tr>
<tr>
<td>Weighted QU</td>
<td>$N + M \log N$</td>
</tr>
<tr>
<td>Path compression</td>
<td>$N + M \log N$</td>
</tr>
<tr>
<td>Weighted + path</td>
<td>$(M + N) \log^* N$</td>
</tr>
</tbody>
</table>

M union-find ops on a set of N objects

**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

Summary
Union-find applications

- Network connectivity.
- Percolation.
- Image processing.
- Least common ancestor.
- Equivalence of finite state automata.
- Hindley-Milner polymorphic type inference.
- Kruskal’s minimum spanning tree algorithm.
- Games (Go, Hex)
- Compiling equivalence statements in Fortran.

Percolation

A model for many physical systems

- N-by-N grid.
- Each square is vacant or occupied.
- Grid percolates if top and bottom are connected by vacant squares.

<table>
<thead>
<tr>
<th>model</th>
<th>system</th>
<th>vacant site</th>
<th>occupied site</th>
<th>percolates</th>
</tr>
</thead>
<tbody>
<tr>
<td>electricity</td>
<td>material</td>
<td>conductor</td>
<td>insulated</td>
<td>conducts</td>
</tr>
<tr>
<td>fluid flow</td>
<td>material</td>
<td>empty</td>
<td>blocked</td>
<td>porous</td>
</tr>
<tr>
<td>social interaction</td>
<td>population</td>
<td>person</td>
<td>empty</td>
<td>communicates</td>
</tr>
</tbody>
</table>

Percolation phase transition

Likelihood of percolation depends on site vacancy probability $p$

Experiments show a threshold $p^*$

- $p > p^*$: almost certainly percolates
- $p < p^*$: almost certainly does not percolate

Q. What is the value of $p^*$?
UF solution to find percolation threshold

- Initialize whole grid to be "not vacant"
- Implement "make site vacant" operation that does `union()` with adjacent sites
- Make all sites on top and bottom rows vacant
- Make random sites vacant until `find(top, bottom)`
- Vacancy percentage estimates $p^*$

```
1 1 1 1 1 not vacant
0 0 0 0 0 vacant
16 16 16 16 16
14 14 14 14 14
12 12 12 12 12
10 10 10 10 10
14 14 14 14 14
58 1 1 1 1
1 1 1 1 1
```

Q. What is percolation threshold $p^*$?
A. About 0.592746 for large square lattices.

Q. Why is UF solution better than solution in IntroProgramming 2.4?

Percolation

```
percolates does not percolate
```

Hex

Hex. [Piet Hein 1942, John Nash 1948, Parker Brothers 1962]
- Two players alternate in picking a cell in a hex grid.
- Black: make a black path from upper left to lower right.
- White: make a white path from lower left to upper right.

```
```

Subtext of today's lecture (and this course)

Steps to developing an usable algorithm.
- Define the problem.
- Find an algorithm to solve it.
- Fast enough?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method

Mathematical models and computational complexity

```
READ Chapter One of Algs in Java
```