1.5 Union Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications

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

The scientific method.

Mathematical analysis.

Union-find applications

- Percolation.
- Games (Go, Hex).
- Dynamic connectivity.
- Least common ancestor.
- Equivalence of finite state automata.
- Hoshen-Kopelman algorithm in physics.
- Hinley-Milner polymorphic type inference.
- Kruskal’s minimum spanning tree algorithm.
- Compiling equivalence statements in Fortran.
- Morphological attribute openings and closings.
- Matlab’s `bwlabel()` function in image processing.

Percolation

A model for many physical systems:
- N-by-N grid of sites.
- Each site is open with probability p (or blocked with probability 1 − p).
- System percolates iff top and bottom are connected by open sites.
Depends on site vacancy probability $p$.

<table>
<thead>
<tr>
<th>$p$ low (0.4)</th>
<th>$p$ medium (0.6)</th>
<th>$p$ high (0.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>does not percolate</td>
<td>percolates?</td>
<td>percolates</td>
</tr>
</tbody>
</table>

**Percolation phase transition**

When $N$ is large, theory guarantees a sharp threshold $p^*$.

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

**Q.** What is the value of $p^*$?

**Monte Carlo simulation**

- Initialize $N$-by-$N$ whole grid to be blocked.
- Declare random sites open until top connected to bottom.
- Vacancy percentage estimates $p^*$.

**Dynamic connectivity solution to estimate percolation threshold**

**Q.** How to check whether an $N$-by-$N$ system percolates?
How to check whether an $N$-by-$N$ system percolates?

- Create an object for each site and name them $0$ to $N^2 - 1$.
- Sites are in the same set if connected by open sites.
- Percolates iff any site on the bottom row is connected to a site on the top row.

Brute-force algorithm: $N^2$ calls to `connected()`.

Clever trick. Introduce two virtual sites (and connections to the top and bottom).
- Percolates iff virtual top site is connected to virtual bottom site.

Efficient algorithm: only 1 call to `connected()`.
Clever trick. Introduce two virtual sites (and connections to top and bottom).
- Percolates iff virtual top site is connected to virtual bottom site.
- Open site is full iff connected to virtual top site.

Q. How to model as dynamic connectivity problem when opening a new site?
A. Connect new site to all of its adjacent open sites.

- dynamic connectivity
  - quick find
  - quick union
  - improvements
  - applications
Dynamic connectivity

Given a set of objects
• **Union**: connect two objects.
• **Connected**: is there a path connecting the two objects?

union(3, 4)
union(8, 0)
union(2, 3)
union(5, 6)
connected(0, 2) **no**
connected(2, 4) **yes**
union(5, 1)
union(7, 3)
union(1, 6)
union(4, 8)
connected(0, 2) **yes**
connected(2, 4) **yes**

more difficult problem: find the path

Connectivity example

Q. Is there a path from p to q?

A. Yes.

Modeling the objects

Dynamic connectivity applications involve manipulating objects of all types.
• Pixels in a digital photo.
• Computers in a network.
• Variable names in Fortran.
• Friends in a social network.
• Transistors in a computer chip.
• Elements in a mathematical set.
• Metallic sites in a composite system.

When programming, convenient to name sites 0 to N-1.
• Use integers as array index.
• Suppress details not relevant to union-find.

Modeling the connections

We assume "is connected to" is an equivalence relation:
• Reflexive: p is connected to p.
• Symmetric: if p is connected to q, then q is connected to p.
• Transitive: if p is connected to q and q is connected to r, then p is connected to r.

Connected components. Maximal set of objects that are mutually connected.

Modeling the connections
Implementing the operations

**Find query.** Check if two objects are in the same component.

**Union command.** Replace components containing two objects with their union.

### Union-find data type (API)

**Goal.** Design efficient data structure for union-find.
- Number of objects $N$ can be huge.
- Number of operations $M$ can be huge.
- Find queries and union commands may be intermixed.

```java
interface IUnionFind {
    void union(int p, int q); // add connection between p and q
    boolean connected(int p, int q); // are p and q in the same component?
    int find(int p); // component identifier for p (0 to N-1)
    int components(); // number of components
}
```

### Dynamic-connectivity client

- Read in number of objects $N$ from standard input.
- Repeat:
  - read in pair of integers from standard input
  - write out pair if they are not already connected

```java
public static void main(String[] args) {
    int N = StdIn.readInt();
    UF uf = new UF(N);
    while (!StdIn.isEmpty()) {
        int p = StdIn.readInt();
        int q = StdIn.readInt();
        if (uf.connected(p, q)) continue;
        uf.union(p, q);
        StdOut.println(p + " " + q);
    }
}
```

% more tiny.txt

```
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
```
Quick-find [eager approach]

Data structure.
• Integer array \(id[i]\) of size \(N\).
• Interpretation: \(p\) and \(q\) in same component iff they have the same id.

\[
\begin{array}{cccccccccc}
  i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
  id[i] & 0 & 1 & 9 & 9 & 9 & 6 & 6 & 7 & 8 \\
\end{array}
\]

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

\[
\begin{array}{cccccccccc}
  i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
  id[i] & 0 & 1 & 9 & 9 & 9 & 6 & 6 & 7 & 8 \\
\end{array}
\]

id\[3\] = 9; id\[6\] = 6
3 and 6 in different components

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

Union. To merge sets containing \(p\) and \(q\), change all entries with \(id[p]\) to \(id[q]\).

Quick-find example

\[
\begin{array}{cccccccccc}
  i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
  id[i] & 0 & 1 & 6 & 6 & 6 & 6 & 7 & 8 & 6 \\
\end{array}
\]

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

\[
\begin{array}{cccccccccc}
  i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
  id[i] & 0 & 1 & 6 & 6 & 6 & 6 & 7 & 8 & 6 \\
\end{array}
\]

id\[p\] and id\[q\] differ, so union() changes entries equal to id\[p\] to id\[q\] (in red)

id\[p\] and id\[q\] match, so no change
Quick-find: Java implementation

```java
public class QuickFindUF {
    private int[] id;
    public QuickFindUF(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }
    public boolean connected(int p, int q) {
        return id[p] == id[q];
    }
    public void union(int p, int q) {
        int pid = id[p];
        int qid = id[q];
        for (int i = 0; i < id.length; i++)
            if (id[i] == pid) id[i] = qid;
    }
}
```

Quick-find is too slow

Cost model. Number of array accesses (for read or write).

<table>
<thead>
<tr>
<th>algorithm</th>
<th>init</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
</tr>
</tbody>
</table>

Quick-find defect.
- Union too expensive.
- Trees are flat, but too expensive to keep them flat.
- Ex. Takes $N^2$ array accesses to process sequence of $N$ union commands on $N$ objects.

Quadratic algorithms do not scale

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.
- $10^9$ union commands on $10^9$ objects.
- Quick-find takes more than $10^{18}$ operations.
- 30+ 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 $id[id[id[...id[i]...]]]$.

**Find.** Check if $p$ and $q$ have the same root.

**Union.** To merge sets containing $p$ and $q$, set the id of $p$'s root to the id of $q$'s root.

**Quick-union example**

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

- 3 and 5 are in different components
Quick-union example

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

Quick-union: Java implementation

```java
public class QuickUnionUF {
    private int[] id;

    public QuickUnionUF(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 connected(int p, int q) {
        return root(p) == root(q);
    }

    public void union(int p, int q) {
        int i = root(p), j = root(q);
        id[i] = j;
    }
}
```

Quick-union is also too slow

**Cost model. Number of array accesses (for read or write).**

<table>
<thead>
<tr>
<th>algorithm</th>
<th>init</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>N</td>
<td>N†</td>
<td>N†</td>
</tr>
</tbody>
</table>

又好又慢

† includes cost of finding root

Quick-find defect.
- Union too expensive ($N$ array accesses).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.
- Trees can get tall.
- Find too expensive (could be $N$ array accesses).
**Weighted quick-union.**
- Modify quick-union to avoid tall trees.
- Keep track of size of each tree (number of objects).
- Balance by linking small tree below large one.

**Weighted quick-union examples**

**Data structure.** Same as quick-union, but maintain extra array `sz[i]` to count number of objects in the tree rooted at `i`.

**Find.** Identical to quick-union.

```
return root(p) == root(q);
```

**Union.** Modify quick-union to:
- Merge smaller tree into larger tree.
- Update the `sz[]` array.

```
int i = root(p);
int j = root(q);
if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```
Running time.
• Find: takes time proportional to depth of \( p \) and \( q \).
• Union: takes constant time, given roots.

**Proposition.** Depth of any node \( x \) is at most \( \lg N \).

![Diagram showing depth of node x is 3, \( \lg N \) is approximately 3.45]

Improvement 2: path compression

Just after computing the root of \( p \), set the id of each examined node to point to that root.

Quick union with path compression. Just after computing the root of \( p \), set the id of each examined node to point to that root.

**Q.** Stop at guaranteed acceptable performance?
**A.** No, easy to improve further.
Path compression: Java implementation

**Standard implementation**: add second loop to `find()` to set the `id[]` of each examined node to the root.

**Simpler one-pass variant**: halve the path length by making 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 example

![Weighted quick-union with path compression example](image)

Path compression: amortized analysis

**Proposition.** Starting from an empty data structure, any sequence of \(M\) union-find operations on \(N\) objects makes at most proportional to \(N + M \log^* N\) array accesses.

- Proof is very difficult.
- Can be improved to \(N + M \alpha(M, N)\).
- But the algorithm is still simple!

**Linear-time algorithm for \(M\) union-find ops on \(N\) objects?**

- Cost within constant factor of reading in the data.
- In theory, WQU is not quite linear.
- In practice, WQU is linear.

**Amazing fact.** No linear-time algorithm exists.

![Boo Tarjan](image)

Summary

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

<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>QU + path compression</td>
<td>(N + M \log N)</td>
</tr>
<tr>
<td>weighted QU + path compression</td>
<td>(N + M \log^* N)</td>
</tr>
</tbody>
</table>

Ex. \([10^6 \text{ unions and finds with } 10^9 \text{ objects}]\)

- WQU reduces time from 30 years to 6 seconds.
- Supercomputer won’t help much; good algorithm enables solution.