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.
1.5 Union-Find

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

Given a set of $N$ objects.

- **Union command**: connect two objects.
- **Find/connected query**: is there a path connecting the two objects?

union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
union(2, 1)

connected(0, 7)  
connected(8, 9)  
union(5, 0)
union(7, 2)
union(6, 1)
union(1, 0)
connected(0, 7)  

![Graph diagram]
Connectivity example

Q. Is there a path connecting $p$ and $q$?

A. Yes.
Applications involve manipulating objects of all types.

- Pixels in a digital photo.
- Computers in a network.
- Friends in a social network.
- Transistors in a computer chip.
- Elements in a mathematical set.
- Variable names in Fortran program.
- Metallic sites in a composite system.

When programming, convenient to name objects 0 to N – 1.

- Use integers as array index.
- Suppress details not relevant to union-find.

- can use symbol table to translate from site names to integers: stay tuned (Chapter 3)
**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.

\[
\begin{align*}
\{0\} & \quad \{1, 4, 5\} & \quad \{2, 3, 6, 7\}
\end{align*}
\]

3 connected components
Implementing the operations

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

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

\[ \text{union}(2, 5) \]

\( \{0\} \{1\ 4\ 5\} \{2\ 3\ 6\ 7\} \)

3 connected components

\( \{0\} \{1\ 2\ 3\ 4\ 5\ 6\ 7\} \)

2 connected components
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
public class UF {
    UF(int N) {
        // initialize union-find data structure with N objects (0 to N – 1)
    }
    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 count() {
        // number of components
    }
}
```
Dynamic-connectivity client

- Read in number of objects $N$ from standard input.
- Repeat:
  - read in pair of integers from standard input
  - if they are not yet connected, connect them and print out pair

```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)) {
            uf.union(p, q);
            StdOut.println(p + " " + q);
        }
    }
}
```
1.5 UNION-FIND

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Quick-find  [eager approach]

Data structure.
- Integer array $\text{id}[]$ of length $N$.
- Interpretation: $p$ and $q$ are connected iff they have the same id.

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

0, 5 and 6 are connected
1, 2, and 7 are connected
3, 4, 8, and 9 are connected
Quick-find [eager approach]

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

$$id[] = \begin{bmatrix} 0 & 1 & 1 & 8 & 8 & 0 & 0 & 1 & 8 & 8 \end{bmatrix}$$

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

$$id[6] = 0; id[1] = 1$$
6 and 1 are not connected

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

$$id[] = \begin{bmatrix} 1 & 1 & 1 & 8 & 8 & 1 & 1 & 1 & 8 & 8 \end{bmatrix}$$

Problem: many values can change
Quick-find demo
Quick-find demo

```
id[]
0 1 1 1 8 8 1 1 1 8 8
```

Diagram:

```
0 --1 --2
  |
  v
 5 --6 --7
```

```
3 --4
  |
  v
 8 --9
```
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;
    }
}
```

- Set id of each object to itself (N array accesses)
- Check whether p and q are in the same component (2 array accesses)
- Change all entries with id[p] to id[q] (at most 2N + 2 array accesses)
Quick-find is too slow

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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</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>

order of growth of number of array accesses

Union is too expensive. It takes $N^2$ array accesses to process a 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!

Quadratic algorithms don't scale with technology.

- New computer may be 10x as fast.
- But, has 10x as much memory ⇒ want to solve a problem that is 10x as big.
- With quadratic algorithm, takes 10x as long!
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Quick-union  [lazy approach]

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

```
<table>
<thead>
<tr>
<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>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>
```

keep going until it doesn’t change (algorithm ensures no cycles)

root of 3 is 9
Quick-union  [lazy approach]

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

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

Union.  To merge components containing \( p \) and \( q \),
set the id of \( p \)'s root to the id of \( q \)'s root.
Quick-union demo

![Quick-union demo](image-url)
Quick-union demo
Quick-union: Java implementation

```java
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);
        int j = root(q);
        id[i] = j;
    }
}
```

- Set id of each object to itself (N array accesses)
- Chase parent pointers until reach root (depth of i array accesses)
- Check if p and q have same root (depth of p and q array accesses)
- Change root of p to point to root of q (depth of p and q array accesses)
Quick-union is also too slow

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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</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>

† includes cost of finding roots

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).
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Improvement 1: weighting

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each tree (number of objects).
- Balance by linking root of smaller tree to root of larger tree.

reasonable alternatives: union by height or "rank"

always chooses the better alternative

might put the larger tree lower
Weighted quick-union demo
Weighted quick-union demo
Quick-union and weighted quick-union example

Quick-union and weighted quick-union (100 sites, 88 union() operations)

average distance to root: 1.52
average distance to root: 5.11
Weighted quick-union: Java implementation

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.

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

Union. Modify quick-union to:
- Link root of smaller tree to root of larger tree.
- Update the sz[] array.

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

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 $\lg N$. 

$N = 10$
$\text{depth}(x) = 3 \leq \lg N$
Weighted quick-union analysis

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 $\lg N$.

Pf. When does depth of $x$ increase?

Increases by 1 when tree $T_1$ containing $x$ is merged into another tree $T_2$.
- The size of the tree containing $x$ at least doubles since $|T_2| \geq |T_1|$.
- Size of tree containing $x$ can double at most $\lg N$ times. Why?
Weighted quick-union analysis

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 $\lg N$.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</th>
<th>union</th>
<th>connected</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>$N^\dagger$</td>
<td>N</td>
</tr>
<tr>
<td>weighted QU</td>
<td>N</td>
<td>$\lg N^\dagger$</td>
<td>$\lg N$</td>
</tr>
</tbody>
</table>

$\dagger$ includes cost of finding roots

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

Quick union with path compression. Just after computing the root of \( p \), set the id of each examined node to point to that root.
Improvement 2: path compression

Quick union with path compression. Just after computing the root of \( p \), set the id of each examined node to point to that root.
Improvement 2: path compression

Quick union with path compression. Just after computing the root of $p$, set the id of each examined node to point to that root.
**Improvement 2: path compression**

*Quick union with path compression.* Just after computing the root of $p$, set the id of each examined node to point to that root.
Improvement 2: path compression

Quick union with path compression. Just after computing the root of $p$, set the id of each examined node to point to that root.
Path compression: Java implementation

**Two-pass 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 (thereby halving path length).

```java
private 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: amortized analysis

**Proposition.** [Hopcroft-Ulman, Tarjan] Starting from an empty data structure, any sequence of $M$ union–find ops on $N$ objects makes $\leq c (N + M \lg^* N)$ array accesses.

- Analysis can be improved to $N + M \alpha(M, N)$.
- Simple algorithm with fascinating mathematics.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\lg^* N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>65536</td>
<td>4</td>
</tr>
<tr>
<td>265536</td>
<td>5</td>
</tr>
</tbody>
</table>

Linear-time algorithm for $M$ union-find ops on $N$ objects?

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

**Amazing fact.** [Fredman-Saks] No linear-time algorithm exists.

iterate log function

in "cell-probe" model of computation
Summary

**Bottom line.** Weighted quick union (with path compression) 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>$N + M \lg^* N$</td>
</tr>
</tbody>
</table>

$M$ union–find operations on a set of $N$ objects

**Ex.** [10$^9$ unions and finds with 10$^9$ objects]
- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
1.5 UNION-FIND

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

<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>
Likelihood of percolation

Depends on site vacancy probability $p$. 

- $p$ low (0.4) does not percolate
- $p$ medium (0.6) percolates?
- $p$ high (0.8) percolates
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 $p^*$.  

$N = 20$

135 open sites
Q. How to check whether an $N$-by-$N$ system percolates?
Dynamic connectivity solution to estimate percolation threshold

**Q.** How to check whether an $N$-by-$N$ system percolates?
- Create an object for each site and name them 0 to $N^2 - 1$. 

![Diagram showing dynamic connectivity solution](image-url)
Dynamic connectivity solution to estimate percolation threshold

Q. 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 same component if connected by open sites.
Dynamic connectivity solution to estimate percolation threshold

Q. 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 same component if connected by open sites.
   • Percolates iff any site on bottom row is connected to site on top row.

brute-force algorithm: $N^2$ calls to connected()
**Clever trick.** Introduce 2 virtual sites (and connections to top and bottom).
- Percolates iff virtual top site is connected to virtual bottom site.

Dynamic connectivity solution to estimate percolation threshold

- Efficient algorithm: only 1 call to connected()
Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?

\[ N = 5 \]

- open site
- blocked site

open this site
Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?
A. Mark new site as open; connect it to all of its adjacent open sites.

$N = 5$

open site

blocked site

open this site

up to 4 calls to union()
Percolation threshold

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

Fast algorithm enables accurate answer to scientific question.
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.
1.5 Union-Find

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