1.5 UNION-FIND

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

Subtext of today’s lecture (and this course)

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 not.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.

Dynamic connectivity problem

Given a set of N objects, support two operation:
- Connect two objects.
- Is there a path connecting the two objects?

connect 4 and 3  
connect 3 and 8  
connect 6 and 5  
connect 9 and 4  
connect 2 and 1

are 0 and 7 connected?  ×
are 8 and 9 connected?  ✓
connect 5 and 0  
connect 7 and 2  
connect 6 and 1  
connect 1 and 0  
are 0 and 7 connected?  ✓
A larger connectivity example

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

A. Yes.

Modeling the objects

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 a 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.

Implementing the operations

Find. In which component is object $p$?
Connected. Are objects $p$ and $q$ in the same component?
Union. Replace components containing objects $p$ and $q$ with their union.

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 component. Maximal set of objects that are mutually connected.
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.
- Union and find operations may be intermixed.

```java
public class UF

UF(int N) // initialize union-find data structure
    with $N$ singleton objects (0 to $N-1$)

void union(int p, int q) // add connection between $p$ and $q$

int find(int p) // component identifier for $p$ (0 to $N-1$)

boolean connected(int p, int q) // are $p$ and $q$ in the same component?

public boolean connected(int p, int q)
{ return find(p) == find(q); }
```

1-line implementation of connected()
Quick-find [eager approach]

Data structure.
- Integer array `id[]` of length `n`.
- Interpretation: `id[p]` is the id of the component containing `p`.

Find. What is the id of `p`?
Connected. Do `p` and `q` have the same id?

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

Quick-find demo

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 int find(int p) {
        return id[p];
    }

    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>initialize</th>
<th>union</th>
<th>find</th>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</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 operations on $N$ objects.

Quadratic algorithms do not scale

**Rough standard (for now).**
- $10^9$ operations per second.
- $10^8$ 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!

Quick-union [lazy approach]

**Data structure.**
- Integer array `id[]` of length $N$.
- Interpretation: `id[i]` is parent of $i$.
- **Root** of $i$ is $id[id[...id[i]...]]$.
- keep going until it doesn’t change (algorithm ensures no cycles)

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

parent of 3 is 4
root of 3 is 9

**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]...]]]`.

Find. What is the root of `p`?
Connected. Do `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

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;
    }

    public int find(int i) {
        while (i != id[i]) i = id[i];
        return i;
    }

    public void union(int p, int q) {
        int i = find(p);
        int j = find(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)
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>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>N</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/connected too expensive (could be N array accesses).

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.

Weighted quick-union demo

Reasonable alternatives: union by height or “rank”
Weighted quick-union demo

![Weighted quick-union demo](image)

Quick-union and weighted quick-union example

![Quick-union and weighted quick-union example](image)

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/connected.** Identical to quick-union.

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

```java
int i = find(p);
int j = find(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.
- Union: takes constant time, given roots.

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

![Weighted quick-union analysis](image)
Weighted quick-union analysis

Running time.
- Find: takes time proportional to depth of \( p \).
- Union: takes constant time, given roots.

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

**Pf.** What causes the depth of object \( x \) to 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?

---

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.

---

Weighted quick-union analysis

Running time.
- Find: takes time proportional to depth of \( p \).
- 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>Find</th>
<th>Connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>( N )</td>
<td>( N )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>quick-union</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
</tr>
<tr>
<td>weighted QU</td>
<td>( N )</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
</tr>
</tbody>
</table>

† 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.

![Diagram of quick union with path compression](image1)

**Bottom line.** Now, `find()` has the side effect of compressing the tree.

---

**Path compression: Java implementation**

Two-pass implementation: add second loop to `find()` to set the $id[]$ of each examined node to the root.

**Simpler one-pass variant (path halving):** Make every other node in path point to its grandparent.

```java
public int find(int i) {
    while (i != id[i]) {
        id[i] = id[id[i]];  // 4
        i = id[i];
    }
    return i;
}
```

\[only\ one\ extra\ line\ of\ code\!\]

**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 $\leq c(N + M \log* N)$ array accesses.

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\log^* N$</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>$2^{65536}$</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.

---

**Summary**

**Key point.** Weighted quick union (and/or 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>$N + M \log N + \log* N$</td>
</tr>
</tbody>
</table>

Order of growth for $M$ 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.

---

**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

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

Likelihood of percolation

Depends on grid size $N$ and site vacancy probability $p$.

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 all sites in an $N$-by-$N$ 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?
A. Model as a dynamic connectivity problem and use union-find.

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

Sites are in same component iff connected by open sites.
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 iff connected by open sites.
- Percolates iff any site on bottom row is connected to any site on top row.

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

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

brute-force algorithm: $N^2$ calls to connected()

more efficient algorithm: only 1 call to connected()
Percolation threshold

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

Fast algorithm enables accurate answer to scientific question.

Subtext of today’s lecture (and this course)

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.