Impact of Great Algorithms

Internet. Web search, packet routing, distributed file sharing.
Biology. Human genome project, protein folding.
Computers. Circuit layout, file system, compilers.
Multimedia. CD player, DVD, MP3, JPEG, DivX, HDTV.
Transportation. Airline crew scheduling, map routing.
Physics. N-body simulation, particle collision simulation.
...
The Usual Suspects

Lectures. [Kevin Wayne]
- TTh 11-12:20, Friend 008.

Precepts. [Wolfgang Mulzer, Janet Yoon]
- Th 12:30, Friend 108.
- Th 3:30, Friend 108.
- Discuss programming assignments, exercises, lecture material.
- First precept meets 9/21.

Questionnaire

Please fill out questionnaire so that we can adapt course as needed.
- Who are you?
- Why are you taking COS 226?
- Which precept(s) can you attend?
- What do you hope to get out of it?
- What is your programming experience?

Coursework and Grading

7 programming assignments. 45%
- Due 11:55pm, starting Monday 9/25.
- Available via course website.

Weekly written exercises. 15%
- Due at beginning of Thursday lecture, starting 9/21.
- Available via course website.

Exams.
- Closed book with cheatsheet.
  - Midterm. 15%
  - Final. 25%

Staff discretion. Adjust borderline cases.

Course Materials

- Syllabus.
- Exercises.
- Lecture slides.
- Programming assignments.

Algorithms in Java, 3rd edition.
- Parts 1-4. [sorting, searching]
- Part 5. [graph algorithms]

- Strings and geometry handouts.
Union Find


Network Connectivity

in  out  evidence
3  4   3  4
4  9   4  9
8  0   8  0
2  3   2  3
5  6   5  6
2  9   (2–3–4–9)
5  9   5  9
7  3   7  3
4  8   4  8
5  6   (5–6)
0  2   (2–3–4–8–0)
6  1   6  1

What are critical operations we need to support?

- **Objects.**
  
  ![Grid Points](image)

- **Disjoint sets of objects.**

  ![Subsets of Connected Grid Points](image)

- **Find:** are objects 2 and 9 in the same set?

  ![Are Two Grid Points Connected?](image)

- **Union:** merge sets containing 3 and 8.

  ![Add a Connection Between Two Grid Points](image)
Union-Find Abstraction

What are critical operations we need to support?
- Objects.
- Disjoint sets of objects.
- Find: are two objects in the same set?
- Union: replace sets containing two items by their union.

Goal. Design efficient data structure for union and find.
- Number of operations M can be huge.
- Number of objects N can be huge.

Quick-Find [eager approach]

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

```
<table>
<thead>
<tr>
<th></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>\text{id}[\cdot]</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
```

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

```
3 and 6 not connected
```

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

```
<table>
<thead>
<tr>
<th></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>\text{id}[\cdot]</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
```

Quick-Find: Example

Applications involve manipulating objects of all types.
- Variable name aliases.
- Pixels in a digital photo.
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Metallic sites in a composite system.

When programming, convenient to name them 0 to N-1.
- Details not relevant to union-find.
- Integers allow quick-access to object-related info.

```
<table>
<thead>
<tr>
<th></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>\text{Example}</td>
<td>3-4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4-9</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>8-0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2-3</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>5-6</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>0</td>
</tr>
<tr>
<td></td>
<td>5-9</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>7-3</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>4-8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>6-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Quick-Find: Java Implementation

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

    public QuickFind(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }

    public boolean find(int p, int q) {
        return id[p] == id[q];
    }

    public void unite(int p, int q) {
        int pid = id[p];
        for (int i = 0; i < id.length; i++)
            if (id[i] == pid) id[i] = id[q];
    }
}
```

Problem Size and Computation Time

Rough standard for 2000.
- $10^9$ operations per second.
- $10^9$ words of main memory.
- Touch all words in approximately 1 second. [unchanged since 1950!]

Ex. Huge problem for quick find.
- $10^{10}$ edges connecting $10^9$ nodes.
- Quick-find might take $10^{20}$ operations. [~10 ops per query]
- 3,000 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: Example

```
3-4 0 1 2 4 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
```

Quick-Union: [lazy approach]

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

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.
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: 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;
    }
}
```

Summary

Quick-find defect.
- Union too expensive.
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.
- Finding the root can be expensive.
- Trees can get tall.

<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>tree height</td>
<td>N</td>
</tr>
</tbody>
</table>

Weighted Quick-Union: Example

- Union of 5 and 3.
- Quick union: link 9 to 6.
- Weighted quick union: link 6 to 9.
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. Same as quick-union, but merge smaller tree into larger tree, and update the sz[] array.

```java
if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else { 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>

Stop at guaranteed acceptable performance? No, can improve further.

Path Compression

Path compression. Just after computing the root of i, set the id of each examined node to root(i).

Weighted Quick-Union with Path Compression

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.
**Applications**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick-find</td>
<td>MN</td>
</tr>
<tr>
<td>Quick-union</td>
<td>MN</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>5 (M + N)</td>
</tr>
</tbody>
</table>

M union-find ops on a set of N elements

**Context**

**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 on Java cell phone beats QF on supercomputer!
**Other Applications**

- **Union-find applications.**
  - Hex.
  - Percolation.
  - Connectivity.
  - Image processing.
  - Least common ancestor.
  - Equivalence of finite state automata.
  - Hinley-Milner polymorphic type inference.
  - Kruskal’s minimum spanning tree algorithm.
  - Compiling equivalence statements in Fortran.

**Percolation**

**Q.** What is percolation threshold \( p^* \) at which charge carriers can percolate from top to bottom?

**A.** \(~0.592746\) for square lattices.

Percolation constant only known via simulation

**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.

**Reference:** [http://mathworld.wolfram.com/GameofHex.html](http://mathworld.wolfram.com/GameofHex.html)
Summary

Lessons.
- Start with simple, brute force approach.
  - don’t use for large problems
  - can’t use for huge problems
- Strive for worst-case performance guarantees.
- Identify fundamental abstractions: union-find.
- Apply to many domains.

might be nontrivial to analyze