CSC 212
Data Structures and Abstractions
Spring 2016

Lecture 01: Union-Find Algorithms
Previously …

Introduction to 212

Case Study: Union-Find

makeset, find, and union operations

quick-find algorithm
Implementing a quick union

Data Structure

```
0 -- 1

3 -- 4

6 -- 7

2

5

8
```
Implementing a quick **union**

**Data Structure**

integer array $p$ of length $n$
Implementing a quick **union**

**Data Structure**

integer array \( p \) of length \( n \)

\( p[i] \) is the parent of \( i \) in a tree
Implementing a quick **union**

**Data Structure**

integer array $p$ of length $n$

$p[i]$ is the parent of $i$ in a tree

```
0 0 2 7 6 8 7 7 8
```

```
0 1 2 3 4 5 6 7 8
```
Implementing a quick **union**

**Data Structure**

integer array $p$ of length $n$

$p[i]$ is the parent of $i$ in a tree

- Parent of 4 is 6
- Parent of 7 is 7
- Root of 4 is 7
- Root of 6 is 7
quick-union example

After 9 calls to makeset(…)

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
quick-union example
quick-union example

union(1, 0)
quick-union example
quick-union example

union(5, 8)
quick-union example
quick-union example

union(3, 7)
**quick-union example**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>2</th>
<th>7</th>
<th>4</th>
<th>8</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
quick-union example

union(4, 6)
quick-union example
quick-union example

union(4, 3)
quick-union example
quick-union example
Implementing a quick **union**

How to implement **makeset(i)**?
Implementing a quick union

How to implement makeset(i)?

trivial, \( p[i] = i \)
Implementing a quick `union`

How to implement `makeset(i)`?

trivial, $p[i] = i$

How to implement `find(i)`?
Implementing a quick union

How to implement $\text{makeset}(i)$?

 trivial, $p[i] = i$

How to implement $\text{find}(i)$?

 return root of tree containing $i$
Implementing a quick **union**

How to implement `makeset(i)`?
trivial, `p[i] = i`

How to implement `find(i)`?
return **root** of tree containing `i`

How to implement `union(a, b)`?
Implementing a quick union

How to implement `makeset(i)`?
- trivial, \( p[i] = i \)

How to implement `find(i)`?
- return root of tree containing \( i \)

How to implement `union(a, b)`?
- set \( p[\text{root of } a] \) to root of \( b \), or vice-versa
```cpp
void QuickUnion::makeset(i) {
    p[i] = i;
}

int QuickUnion::find(i) {
    while (i != p[i]) {
        i = p[i];
    }
    return i;
}

void QuickUnion::union(int a, int b) {
    p[find(a)] = find(b);
}
```
void QuickUnion::makeset(i) {
    p[i] = i;  // 1 write access (constant time)
}

int QuickUnion::find(i) {
    while (i != p[i]) {
        i = p[i];
    }
    return i;
}

void QuickUnion::union(int a, int b) {
    p[find(a)] = find(b);
}
```cpp
void QuickUnion::makeset(i) {
    p[i] = i;  // 1 write access (constant time)
}

int QuickUnion::find(i) {
    while (i != p[i]) {
        i = p[i];  // [2 * depth] read accesses
    }
    return i;
}

void QuickUnion::union(int a, int b) {
    p[find(a)] = find(b);  // (linear time in the worst case)
}
```
```cpp
void QuickUnion::makeset(i) {
    p[i] = i;
}

int QuickUnion::find(i) {
    while (i != p[i]) {
        i = p[i];
    }
    return i;
}

void QuickUnion::union(int a, int b) {
    p[find(a)] = find(b);
}
```

1 write access (constant time)

[2 * depth] read accesses (linear time in the worst case)

2 * cost of find (linear time in the worst case)
Computational Cost

<table>
<thead>
<tr>
<th></th>
<th>makeset</th>
<th>find</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>1</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>quick-union</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

ignore leading constants
# Computational Cost

<table>
<thead>
<tr>
<th></th>
<th>makeset</th>
<th>find</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>1</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>quick-union</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

Ignore leading constants

If trees are tall, both **union** and **find** are expensive!
Wait ... you said 'quick union'

Need improvements ...

weighted quick-union

path compression
Implementing a weighted quick-union

Always link smaller trees to larger trees

http://algs4.cs.princeton.edu/15uf/
Implementing a weighted quick-union

Always link smaller trees to larger trees will need an extra array to keep track of size

http://algs4.cs.princeton.edu/15uf/
void QuickUnion::makeSet(int i) {
    p[i] = i;
    size[i] = 1;
}

int QuickUnion::find(int i) {
    while (i != p[i]) {
        i = p[i];
    }
    return i;
}

void QuickUnion::union(int a, int b) {
    root_a = find(a);
    root_b = find(b);
    if (root_a == root_b) return;
    if (size[root_a] < size[root_b]) {
        p[root_a] = root_b;
        size[root_b] += size[root_a];
    } else {
        p[root_b] = root_a;
        size[root_a] += size[root_b];
    }
}
void QuickUnion::makeset(int i) {
    p[i] = i;
    size[i] = 1;
}

int QuickUnion::find(int i) {
    while (i != p[i]) {
        i = p[i];
    }
    return i;
}

void QuickUnion::union(int a, int b) {
    root_a = find(a);
    root_b = find(b);
    if (root_a == root_b) return;
    if (size[root_a] < size[root_b]) {
        p[root_a] = root_b;
        size[root_b] += size[root_a];
    } else {
        p[root_b] = root_a;
        size[root_a] += size[root_b];
    }
}

1 write access (constant time)
void QuickUnion::makeSet(int i) {
    p[i] = i;
    size[i] = 1;
}

int QuickUnion::find(int i) {
    while (i != p[i]) {
        i = p[i];
    }
    return i;
}

void QuickUnion::union(int a, int b) {
    root_a = find(a);
    root_b = find(b);
    if (root_a == root_b) return;
    if (size[root_a] < size[root_b]) {
        p[root_a] = root_b;
        size[root_b] += size[root_a];
    } else {
        p[root_b] = root_a;
        size[root_a] += size[root_b];
    }
}
void QuickUnion::makeset(i) {
    p[i] = i;
    size[i] = 1;
}

int QuickUnion::find(i) {
    while (i != p[i]) {
        i = p[i];
    }
    return i;
}

void QuickUnion::union(int a, int b) {
    root_a = find(a);
    root_b = find(b);
    if (root_a == root_b) return;
    if (size[root_a] < size[root_b]) {
        p[root_a] = root_b;
        size[root_b] += size[root_a];
    } else {
        p[root_b] = root_a;
        size[root_a] += size[root_b];
    }
}
void QuickUnion::makeSet(int i) {
    p[i] = i;
    size[i] = 1;
}

int QuickUnion::find(int i) {
    while (i != p[i]) {
        i = p[i];
    }
    return i;
}

void QuickUnion::union(int a, int b) {
    root_a = find(a);
    root_b = find(b);
    if (root_a == root_b) return;
    if (size[root_a] < size[root_b]) {
        p[root_a] = root_b;
        size[root_b] += size[root_a];
    } else {
        p[root_b] = root_a;
        size[root_a] += size[root_b];
    }
}
Logarithmic?

When \textbf{find} traverses from a node \textit{x} to the \textbf{root} each time we follow an arrow to a subtree of size at least double the size of the previous subtree
Logarithmic?

When **find** traverses from a node \( x \) to the **root** each time we follow an arrow to a subtree of size at least double the size of the previous subtree.

How many arrows do we need to follow (at most) in the **worst case**?
Logarithmic?

When \textbf{find} traverses from a node $x$ to the \textbf{root} each time we follow an arrow to a subtree of size at least double the size of the previous subtree

How many arrows do we need to follow (at most) in the \textbf{worst case}? $\log_2 n$
Logarithmic?

When `find` traverses from a node \( x \) to the root each time we follow an arrow to a subtree of size at least double the size of the previous subtree.

How many arrows do we need to follow (at most) in the worst case? \( \log_2 n \)
## Computational Cost

<table>
<thead>
<tr>
<th></th>
<th>makeset</th>
<th>find</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>1</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>quick-union</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>weighted quick-union</td>
<td>1</td>
<td>log n</td>
<td>log n</td>
</tr>
</tbody>
</table>

ignore leading constants
## Computational Cost

<table>
<thead>
<tr>
<th>Makeset</th>
<th>Find</th>
<th>Union</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>weighted quick-union</td>
<td>1</td>
<td>log n</td>
</tr>
</tbody>
</table>

Processing \(m\) union operations takes \(m \times \log_2 n\) accesses (\(m\) calls to `union`)

*ignore leading constants*
## Computational Cost

<table>
<thead>
<tr>
<th></th>
<th>makeset</th>
<th>find</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>1</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>quick-union</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>weighted quick-union</td>
<td>1</td>
<td>log n</td>
<td>log n</td>
</tr>
</tbody>
</table>

Processing \( m \) union operations takes \( m \times \log_2 n \) accesses (\( m \) calls to \texttt{union})

If \( m \sim n \), \textbf{linearithmic time} (\( n \log_2 n \) accesses)
Is linearithmic running time any good?

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>$N^2$</th>
<th>$N \log N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>100</td>
<td>33</td>
</tr>
</tbody>
</table>
Is linearithmic running time any good?

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>$N^2$</th>
<th>$N \log N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>10,000</td>
<td>664</td>
</tr>
</tbody>
</table>
Is linearithmic running time any good?

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>$N^2$</th>
<th>$N \log N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>10,000</td>
<td>664</td>
</tr>
<tr>
<td>$n = 1000$</td>
<td>1,000,000</td>
<td>9,965</td>
</tr>
</tbody>
</table>
Is linearithmic running time any good?

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>( N^2 )</th>
<th>( N \log N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 10 )</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>( n = 100 )</td>
<td>10,000</td>
<td>664</td>
</tr>
<tr>
<td>( n = 1000 )</td>
<td>1,000,000</td>
<td>9,965</td>
</tr>
<tr>
<td>( n = 10000 )</td>
<td>100,000,000</td>
<td>132,877</td>
</tr>
</tbody>
</table>
Is linearithmic running time any good?

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>$N^2$</th>
<th>$N \log N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 10</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>n = 100</td>
<td>10,000</td>
<td>664</td>
</tr>
<tr>
<td>n = 1000</td>
<td>1,000,000</td>
<td>9,965</td>
</tr>
<tr>
<td>n = 10000</td>
<td>100,000,000</td>
<td>132,877</td>
</tr>
<tr>
<td>n = 100000</td>
<td>10,000,000,000</td>
<td>1,660,964</td>
</tr>
</tbody>
</table>
Is linearithmic running time any good?

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>$N^2$</th>
<th>$N \log N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>10.000</td>
<td>664</td>
</tr>
<tr>
<td>$n = 1000$</td>
<td>1.000.000</td>
<td>9.965</td>
</tr>
<tr>
<td>$n = 10000$</td>
<td>100.000.000</td>
<td>132.877</td>
</tr>
<tr>
<td>$n = 100000$</td>
<td>10.000.000.000</td>
<td>1.660.964</td>
</tr>
<tr>
<td>$n = 1000000$</td>
<td>1.000.000.000.000</td>
<td>19.931.568</td>
</tr>
</tbody>
</table>
Is linearithmic running time any good?

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>(N^2)</th>
<th>(N \log N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 10</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>n = 100</td>
<td>10.000</td>
<td>664</td>
</tr>
<tr>
<td>n = 1000</td>
<td>1.000.000.000</td>
<td>9.965</td>
</tr>
<tr>
<td>n = 10000</td>
<td>100.000.000.000</td>
<td>132.877</td>
</tr>
<tr>
<td>n = 100000</td>
<td>10.000.000.000.000</td>
<td>1.660.964</td>
</tr>
<tr>
<td>n = 1000000</td>
<td>1.000.000.000.000.000</td>
<td>19.931.568</td>
</tr>
<tr>
<td>n = 10000000</td>
<td>100.000.000.000.000.000</td>
<td>232.534.966</td>
</tr>
</tbody>
</table>
Is linearithmic running time any good?

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>$N^2$</th>
<th>$N \log N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>10,000</td>
<td>664</td>
</tr>
<tr>
<td>$n = 1000$</td>
<td>1,000,000</td>
<td>9,965</td>
</tr>
<tr>
<td>$n = 10000$</td>
<td>100,000,000</td>
<td>132,877</td>
</tr>
<tr>
<td>$n = 100000$</td>
<td>10,000,000,000</td>
<td>1,660,964</td>
</tr>
<tr>
<td>$n = 1000000$</td>
<td>1,000,000,000,000</td>
<td>19,931,568</td>
</tr>
<tr>
<td>$n = 10000000$</td>
<td>100,000,000,000,000</td>
<td>232,534,966</td>
</tr>
<tr>
<td>$n = 100000000$</td>
<td>10,000,000,000,000,000</td>
<td>2,657,542,475</td>
</tr>
</tbody>
</table>
Is linearithmic running time any good?

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>N²</th>
<th>N log N</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 10</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>n = 100</td>
<td>10.000</td>
<td>664</td>
</tr>
<tr>
<td>n = 1000</td>
<td>1.000.000</td>
<td>9.965</td>
</tr>
<tr>
<td>n = 10000</td>
<td>100.000.000</td>
<td>132.877</td>
</tr>
<tr>
<td>n = 100000</td>
<td>10.000.000.000</td>
<td>1.660.964</td>
</tr>
<tr>
<td>n = 1000000</td>
<td>1.000.000.000.000</td>
<td>19.931.568</td>
</tr>
<tr>
<td>n = 10000000</td>
<td>100.000.000.000.000</td>
<td>232.534.966</td>
</tr>
<tr>
<td>n = 100000000</td>
<td>10.000.000.000.000.000</td>
<td>2.657.542.475</td>
</tr>
<tr>
<td>n = 1000000000</td>
<td>1.000.000.000.000.000.000</td>
<td>29.897.352.854</td>
</tr>
</tbody>
</table>
WQU with Path Compression

On every call to `find`, compress all pointers on the path traversed so that they all point to the root.
WQU with Path Compression

On every call to `find`, compress all pointers on the path traversed so that they all point to the root

```
find(2)
```
On every call to `find`, compress all pointers on the path traversed so that they all point to the root.
Applications
Maze Generation

From Algorithm Design and Applications, Goodrich & Tamassia
Maze Generation

Algorithm MazeGenerator($G$, $E$):

**Input:** A grid, $G$, consisting of $n$ cells and a set, $E$, of $m$ “walls,” each of which divides two cells, $x$ and $y$, such that the walls in $E$ initially separate and isolate all the cells in $G$

**Output:** A subset, $R$ of $E$, such that removing the edges in $R$ from $E$ creates a maze defined on $G$ by the remaining walls

while $R$ has fewer than $n - 1$ edges do

  Choose an edge, $(x, y)$, in $E$ uniformly at random from among those previously unchosen

  if $\text{find}(x) \neq \text{find}(y)$ then

    union($\text{find}(x)$, $\text{find}(y)$)

    Add the edge $(x, y)$ to $R$

return $R$

From Algorithm Design and Applications, Goodrich & Tamassia
Percolation

Percolation


http://michaeltoth.net/blog/posts/2014/10/Percolation/
Percolation

Once we have a NxN grid, how can we check whether the system percolates?
Percolation

Once we have a NxN grid, how can we check whether the system percolates?

first, build a union-find structure for the grid and ‘union’ open sites
Percolation

Once we have a NxN grid, how can we check whether the system percolates?

first, build a union-find structure for the grid and ‘union’ open sites
second, check for ‘connectivity’ between sites in the top row with sites in the bottom row (N² queries)
Percolation

Once we have a $N \times N$ grid, how can we check whether the system percolates?

first, build a union-find structure for the grid and ‘union’ open sites

second, check for ‘connectivity’ between sites in the top row with sites in the bottom row ($N^2$ queries)

can we do better?
Hex Game

From [http://ticc.uvt.nl/icga/games/hex/](http://ticc.uvt.nl/icga/games/hex/)