Disjoint Union - Find

- Maintain a set of pairwise disjoint sets.
  - \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
- Required operations
  - Union – merge two sets to create their union (original sets need not be preserved)
  - Find – determine which set a given item appears in (in particular, be able to quickly tell whether two items are in the same set)

Set Representation

- Maintain a set of pairwise disjoint sets.
  - \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
- Each set has a unique name, one of its members
  - \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}

Union

- Union(x,y) – take the union of two sets named x and y
  - \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
  - Union(5,1)
    - \{3,5,7,1,6\}, \{4,2,8\}, \{9\},
Find

• Find(x) – return the name of the set containing x.
  – \{3, 5, 7, 1, 6\}, \{4, 2, 8\}, \{9\},
  – Find(1) = 5
  – Find(4) = 8

An Example: Building Mazes

• Build a random maze by erasing edges.

Building Mazes (2)

• Pick Start and End

Building Mazes (3)

• Repeatedly pick random edges to delete.
**Desired Properties**

- None of the boundary is deleted
- Every cell is reachable from every other cell.
- Only one path from any one cell to another (There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.)

**A Cycle**

**A Good Solution**

**A Hidden Tree**
Number the Cells
We have disjoint sets $S = \{ \{1\}, \{2\}, \{3\}, \{4\}, \ldots, \{36\} \}$ each cell is a set by itself. Also a set of all possible edges $E = \{ (1,2), (1,7), (2,8), (2,3), \ldots \}$ 60 edges total.

Start

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
| 31| 32| 33| 34| 35| 36| End

Basic Algorithm
- $S$ = set of sets of connected cells
- $E$ = set of edges not yet examined
- Maze = set of maze edges (initially empty)

While there is more than one set in $S$ {
    pick a random edge $(x,y)$ and remove from $E$
    \[ u := \text{Find}(x); \]
    \[ v := \text{Find}(y); \]
    if $u \neq v$ then // removing edge $(x,y)$ connects previously non-
        // connected cells $x$ and $y$ - leave this edge removed!
        Union($u,v$)
    else // cells $x$ and $y$ were already connected, add this
        // edge to set of edges that will make up final maze,
        add edge $(x,y)$ to Maze
}
All remaining members of $E$ together with Maze form the maze

Example Step

Pick $(8,14)$

Example

S \{1,2,7,8,9,13,19\} \{3\} \{4\} \{5\} \{6\} \{10\} \{1\} \{11,17\} \{12\} \{14,20,26,27\} \{15,16,21\}
Find(8) = 7
Find(14) = 20
Union(7,20)

S \{1,2,7,8,9,13,19,14,20,26,27\} \{3\} \{4\} \{5\} \{6\} \{10\} \{1\} \{11,17\} \{12\} \{14,20,26,27\} \{15,16,21\}
\{22,23,24,29,30,32\} \{34,35,36\}
Example

Pick (19,20)

Example at the End

Implementing the DS ADT

- \( n \) elements,
  
  Total Cost of: \( m \) finds, \( \leq n-1 \) unions

- Target complexity: \( O(m+n) \)
  
  \( i.e. \ O(1) \) amortized

- \( O(1) \) worst-case for find as well as union would be great, but…

  Known result: both find and union cannot be done in worst-case \( O(1) \) time

Attempt #1

- Hash elements to a hashtable
- Store set identifier for each element as data

  \( \text{runtime for find:} \)

  \( \text{runtime for union:} \)

  \( \text{runtime for } m \text{ finds, } n-1 \text{ unions:} \)
Attempt #2

- Hash elements to a hashtable
- Store set identifier for each element as data
- *Link* all elements in the same set together

runtime for find:

runtime for union:

runtime for m finds, n-1 unions:

Attempt #3

- Hash elements to a hashtable
- Store set identifier for each element as data
- *Link* all elements in the same set together
- Always update identifiers of *smaller* set

runtime for find:

runtime for union:

runtime for m finds, n-1 unions:

[Read section 8.2]

Up-Tree for Disjoint Union/Find

Initial state: 1 2 3 4 5 6 7

After several Unions:

Roots are the names of each set.

Find Operation

Find(x) - follow x to the root and return the root

Find(6) = 7
Union Operation

Union(x, y) - assuming x and y are roots, point y to x.

```
Union(1, 7)
```

Simple Implementation

- Array of indices

```
up = [0, 1, 0, 7, 7, 5, 0]
```

```
Union(1, 7)
```

Implementation

```c
int Find(int x) {
    while(up[x] != 0) {
        x = up[x];
    }
    return x;
}
```

```c
void Union(int x, int y) {
    up[y] = x;
}
```

```
runtime for Union():
```

```
runtime for Find():
```

```
runtime for m Finds and n-1 Unions:
```

Now this doesn’t look good 😞

Can we do better? Yes!

1. Improve union so that find only takes \( \Theta(\log n) \)
   - Union-by-size
   - Reduces complexity to \( \Theta(m \log n + n) \)

2. Improve find so that it becomes even better!
   - Path compression
   - Reduces complexity to almost \( \Theta(m + n) \)
A Bad Case

Weighted Union

- Weighted Union
  - Always point the smaller (total # of nodes) tree to the root of the larger tree

Example Again

Analysis of Weighted Union

With weighted union an up-tree of height $h$ has weight at least $2^h$.

- Proof by induction
  - Basis: $h = 0$. The up-tree has one node, $2^0 = 1$
  - Inductive step: Assume true for all $h' < h$. 

Minimum weight up-tree of height $h$ formed by weighted unions

$W(T) \geq 2^{h-1} + 2^{h-1} = 2^h$
Analysis of Weighted Union (cont)

Let $T$ be an up-tree of weight $n$ formed by weighted union. Let $h$ be its height.
\[
\begin{align*}
\text{n} & \geq 2^h \\
\log_2 n & \geq h
\end{align*}
\]

- Find($x$) in tree $T$ takes $O(\log n)$ time.
  - Can we do better?

Worst Case for Weighted Union

<table>
<thead>
<tr>
<th>n/2 Weighted Unions</th>
<th>n/4 Weighted Unions</th>
</tr>
</thead>
</table>

Example of Worst Cast (cont’)

After $n/2 + n/4 + \ldots + 1$ Weighted Unions:

If there are $n = 2^k$ nodes then the longest path from leaf to root has length $k$.

Array Implementation

<table>
<thead>
<tr>
<th>up weight</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Weighted Union

W-Union(i, j : index)
//i and j are roots
wi := weight[i];
wj := weight[j];
if wi < wj then
up[i] := j;
weight[j] := wi + wj;
else
up[j] := i;
weight[i] := wi + wj;
}

new runtime for Union():
new runtime for Find():
runtime for m finds and n-1 unions =

Union-by-size: Find Analysis

• Complexity of Find: O(max node depth)
• All nodes start at depth 0
• Node depth increases:
  – Only when it is part of smaller tree in a union
  – Only by one level at a time
Result: tree size doubles when node depth increases by 1

Find runtime = O(node depth) =
runtime for m finds and n-1 unions =

Nifty Storage Trick

• Use the same array representation as before

• Instead of storing -1 for a root, simply store -size

[Read section 8.4, page 276]

How about Union-by-height?

• Can still guarantee O(log n) worst case depth

Left as an exercise!

• Problem: Union-by-height doesn’t combine very well with the new find optimization technique we’ll see next
Path Compression

- On a Find operation point all the nodes on the search path directly to the root.

Draw the result of Find(e):

Self-Adjustment Works
Path Compression Find

```text}
PC-Find(i : index) {
  r := i;
  while up[r] ≠ -1 do //find root//
    r := up[r];
  if i ≠ r then //compress path//
    k := up[i];
    while k ≠ r do
      up[i] := r;
      i := k;
      k := up[k]
  return(r)
}
```
Disjoint Union / Find with Weighted Union and PC

- Worst case time complexity for a W-Union is $O(1)$ and for a PC-Find is $O(\log n)$.
- Time complexity for $m \geq n$ operations on $n$ elements is $O(m \log^* n)$ where $\log^* n$ is a very slow growing function.
  - $\log^* n < 7$ for all reasonable $n$. Essentially constant time per operation!
- Using “ranked union” gives an even better bound theoretically.

Amortized Complexity

- For disjoint union / find with weighted union and path compression.
  - average time per operation is essentially a constant.
  - worst case time for a PC-Find is $O(\log n)$.
- An individual operation can be costly, but over time the average cost per operation is not.