Disjoint Union - Find

• 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

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

Find

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

Number the Cells

We have disjoint sets \(S = \{\{1\}, \{2\}, \{3\}, \ldots, \{36\}\}\) each cell is unto itself. We have all possible edges \(E = \{(1,2), (1,7), (2,8), (2,3), \ldots\\}\) 60 edges total.

Basic Algorithm

• \(S\) = set of sets of connected cells
• \(E\) = set of edges
• \(\text{Maze}\) = 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!
    \(\text{Union}(u,v)\)
  else // cells \(x\) and \(y\) were already connected, add this
    // edge to set of edges that will make up final maze.
    add \((x,y)\) to \(\text{Maze}\)
} 
```

All remaining members of \(E\) together with \(\text{Maze}\) form the maze

Example Step

```
Pick (8,14)
```

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

```
Example

\[
\begin{align*}
S &= \{1, 2, 7, 9, 13, 19\} \\
 S &= \{1, 2, 7, 8, 9, 13, 19, 14, 20, 26, 27\} \\
 F(8) &= 7 \\
 F(14) &= 20 \\
 U(7, 20) \\
 E(22, 24, 29, 39, 32, 33, 34, 35, 36)
\end{align*}
\]

Example at the End

\[
\begin{align*}
S &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36\}
\end{align*}
\]

Implementing the DS ADT

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

\[ O(m+n) \]
  *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

Find Operation

\[
\text{Find}(x) - \text{follow } x \text{ to the root and return the root}
\]

Roots are the names of each set.
**Union Operation**

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

**Union (1, 7)**

**Simple Implementation**

- Array of indices

<table>
<thead>
<tr>
<th></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><strong>up</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Up[x] == -1 means x is a root.

**Sample Implementations**

```c
int Find(int x) {
    while(up[x] != -1) {
        x = up[x];
    }
    return x;
}

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

runtime for **Union()**: 

runtime for **Find()**: 

runtime for m Finds and n-1 Unions:

**Find Solutions**

**Recursive**

```c
int Find(int x) {
    if up[x] == -1 {
        return x;
    } else {
        return Find(up[x]);
    }
}
```

**Iterative**

```c
int Find(int x) {
    while up[x] != -1 {
        x = up[x];
    }
    return x;
}
```

**A Bad Case**

1 2 3 4 5 6

Union(2, 1)

1 2 3 4 5 6

Union(3, 2)

1 2 3 4 5 6

Union(n, n-1)

1 2 3 4 5 6

Find(1) n steps!!

**Now this doesn’t look good**

Can we do better? **Yes!**

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

2. Improve **find** so that it becomes even better!
   - **Path compression**
   - Reduces complexity to almost O(m + n)
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 \( h' < h \).

Analysis of Weighted Union (cont)

Let \( T \) be an up-tree of weight \( n \) formed by weighted union. Let \( h \) be its height.

\[ n \geq 2^h \]
\[ \log_2 n \geq h \]

- Find(x) in tree \( T \) takes \( O(\log n) \) time.
  
Worst Case for Weighted Union

n/2 Weighted Unions

n/4 Weighted Unions

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

2
\[ \begin{array}{c}
1 & 3 & 4 \\
2 & 5 & 6 \\
7 & & \\
\end{array} \]

Weighted Union

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

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

Union-by-size: Find Analysis

- Complexity of \textit{Find}: \(O(\max \text{ node depth})\)
- 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

\[ \text{Find runtime} = O(\text{node depth}) = \]
运行时 \(m\) 找和 \(n-1\) 汇合 =

Nifty Storage Trick

- Use the same array representation as before
- Instead of storing \(-1\) for the root, simply store \(-\text{size}\)

[Read section 8.4, page 299]

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.

PC-Find(3)
Path Compression

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

```
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)
}
```

Self-Adjustment Works

```
Path Compression: Code
int Find(Object x) {
    // x had better be in
    // the set!
    int xID = hTable[x];
    int i = xID;
    // Get the root for
    // this set
    while(up[xID] != -1) {
        xID = up[xID];
    }
    // Change the parent for
    // all nodes along the path
    while(up[i] != -1) {
        temp = up[i];
        up[i] = xID;
        i = temp;
    }
    return xID;
}
```

Draw the result of Find(e):

Interlude: A Really Slow Function

*Ackermann’s function* is a really big function \( A(x, y) \) with inverse \( \alpha(x, y) \) which is really small.

How fast does \( \alpha(x, y) \) grow?

\( \alpha(x, y) = 4 \) for \( x \) far larger than the number of atoms in the universe \( (2^{300}) \)

\( \alpha \) shows up in:
- Computation Geometry (surface complexity)
- Combinatorics of sequences
A More Comprehensible Slow Function

\[ \log^* x = \text{number of times you need to compute} \]
\[ \log \text{ to bring value down to at most 1} \]

E.g. \( \log^* 2 = 1 \)
\( \log^* 4 = \log^* 2^2 = 2 \)
\( \log^* 16 = \log^* 2^{12} = 3 \) \( (\log \log \log 16 = 1) \)
\( \log^* 65536 = \log^* 2^{222} = 4 \) \( (\log \log \log \log 65536 = 1) \)
\( \log^* 2^{65536} = \ldots \ldots \ldots = 5 \)

Take this: \( \alpha(m,n) \) grows even slower than \( \log^* n \) !!

Complex Complexity of Union-by-Size + Path Compression

Tarjan proved that, with these optimizations, \( p \) union and find operations on a set of \( n \) elements have worst case complexity of \( O(p \cdot \alpha(p, n)) \)

For all practical purposes this is amortized constant time: \( O(p \cdot 4) \) for \( p \) operations!

- Very complex analysis – worse than splay tree analysis etc. that we skipped!

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.