Disjoint Set Operations:
“UNION-FIND” Method

CSE 373
Data Structures

Equivalence Relations

- A relation $R$ is defined on set $S$ if for every pair of elements $a, b \in S$, $a R b$ is either true or false.
- An equivalence relation is a relation $R$ that satisfies the 3 properties:
  - Reflexive: $a R a$ for all $a \in S$
  - Symmetric: $a R b$ iff $b R a$; $a, b \in S$
  - Transitive: $a R b$ and $b R c$ implies $a R c$

Equivalence Classes

- Given an equivalence relation $R$, decide whether a pair of elements $a, b \in S$ is such that $a R b$.
- The equivalence class of an element $a$ is the subset of $S$ of all elements related to $a$.
- Equivalence classes are disjoint sets

Dynamic Equivalence Problem

- Starting with each element in a singleton set, and an equivalence relation, build the equivalence classes
- Requires two operations:
  - Find the equivalence class (set) of a given element
  - Union of two sets
- It is a dynamic (on-line) problem because the sets change during the operations and Find must be able to cope!

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

- **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
  - Find(9) = ?

### An Application

- Build a random maze by erasing edges.

### An Application (ct’d)

- Pick Start and End

### An Application (ct’d)

- Repeatedly pick random edges to delete.

### Desired Properties

- None of the boundary is deleted
- Every cell is reachable from every other cell.
- There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.
A Cycle (we don’t want that)

A Good Solution

Good Solution : A Hidden Tree

Number the Cells

We have disjoint sets $S = \{ (1), (2), (3), (4), \ldots, (36) \}$, 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
- 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 $E$
  - if $u \neq v$ then
    - Union(u,v) //knock down the wall between the cells (cells in the same set are connected)
  - else
    - add $(x,y)$ to Maze //don’t remove because there is already a path between $x$ and $y$

All remaining members of $E$, together with Maze form the maze

Example Step

<table>
<thead>
<tr>
<th>Pick (8,14)</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 4 6 8 10 12</td>
<td>1,2,7,8,9,13,19</td>
</tr>
<tr>
<td>3 5 7 9 11 13</td>
<td>3</td>
</tr>
<tr>
<td>14 16 18 20 22 24</td>
<td>10</td>
</tr>
<tr>
<td>15 17 19 21 23 25</td>
<td>11,12</td>
</tr>
<tr>
<td>14 26 27 28 29 30</td>
<td>14,20,26,27</td>
</tr>
<tr>
<td>31 32 33 34 35 36</td>
<td>15,16,21</td>
</tr>
<tr>
<td>41 42 43 44 45 46</td>
<td></td>
</tr>
<tr>
<td>47 48 49 50 51 52</td>
<td></td>
</tr>
<tr>
<td>53 54 55 56 57 58</td>
<td></td>
</tr>
<tr>
<td>59 60 61 62 63 64</td>
<td></td>
</tr>
<tr>
<td>65 66 67 68 69 70</td>
<td></td>
</tr>
<tr>
<td>71 72 73 74 75 76</td>
<td></td>
</tr>
<tr>
<td>77 78 79 80 81 82</td>
<td></td>
</tr>
<tr>
<td>83 84 85 86 87 88</td>
<td></td>
</tr>
<tr>
<td>89 90 91 92 93 94</td>
<td></td>
</tr>
<tr>
<td>95 96 97 98 99 100</td>
<td></td>
</tr>
</tbody>
</table>

Start  End

Start  End
### Example

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

1. **Find** (8) = 7
2. **Find** (14) = 20
3. **Union** (7, 20)

### Find Operation

- **Find** (x) follow x to the root and return the root (which is the name of the class).

**Find (6) = 7**

### Union Operation

- **Union** (i, j) - assuming i and j roots, point i to j.

**Union** (1, 7)
### Simple Implementation

- **Array of indices** (Up[i] is parent of i)
  
  Up[i] = 0 means x is a root.

### Union

Union(up[], x, y) :

//precondition: x and y are roots//
Up[x] := y

Constant Time!

### Find

Find(x) :

//precondition: x is in the range 1 to size//
???

 UP

if up[x] = 0 then return x
else

### Weighted Union

- **Weighted Union** (weight = number of nodes)
  - Always point the smaller tree to the root of the larger tree

### Example Again

Find(1) constant time

Find(1) n steps!!
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 $h' < h$.

\[
W(T) \geq 2^{h-1} \times 2^h = 2^h
\]

Worst Case for Weighted Union

- Let $T$ be an up-tree of weight $n$ formed by weighted union. Let $h$ be its height.
  - $n \geq 2^h$
  - $\log_2 n \geq h$
  - $\text{Find}(x)$ in tree $T$ takes $O(\log n)$ time.
  - Can we do better?

Example of Worst Cast (cont’)

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

Find

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

Elegant Array Implementation

- Weighted Union

\[
W-\text{Union}(i,j: \text{index})
\]

//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;
\]
Path Compression

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

Path Compression Find

```plaintext
PC-Find(i : index) {
    r := i;
    while up[r] ≠ 0 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

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!

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.
Find Solutions

Recursive
Find(up[]) : integer array, x : integer) : integer {
  //precondition: x is in the range 1 to size/
  if up[x] = 0 then return x
  else return Find(up,up[x]);
}

Iterative
Find(up[]) : integer array, x : integer) : integer {
  //precondition: x is in the range 1 to size/
  while up[x] ≠ 0 do
    x := up[x];
  return x;
}