Disjoint Union/Find

Equivalence Relations

Relation $R$:
1. For every pair of elements $(a, b)$ in a set $S$, $a R b$ is either true or false.
2. If $a R b$ is true, then $a$ is related to $b$.

An equivalence relation satisfies:
1. (Reflexive) $a R a$
2. (Symmetric) $a R b$ iff $b R a$
3. (Transitive) $a R b$ and $b R c$ implies $a R c$

A new question

• Which of these things are similar?
  { grapes, blackberries, plums, apples, oranges, peaches, raspberries, lemons }

• If limes are added to this fruit salad, and are similar to oranges, then are they similar to grapes?

• How do you answer these questions efficiently?

Equivalence Classes

• Given a set of things...
  { grapes, blackberries, plums, apples, oranges, peaches, raspberries, lemons, bananas }

• ...define the equivalence relation
  All citrus fruit is related, all berries, all stone fruits, and THAT’S IT.

• ...partition them into related subsets
  { grapes }, { blackberries, raspberries }, { oranges, lemons },
  { plums, peaches }, { apples }, { bananas }

Everything in an equivalence class is related to each other.

Determining equivalence classes

• Idea: give every equivalence class a name
  – { oranges, limes, lemons } = “like-ORANGES”
  – { peaches, plums } = “like-PEACHES”
  – Etc.
• To answer if two fruits are related:
  – FIND the name of one fruit’s e.c.
  – FIND the name of the other fruit’s e.c.
  – Are they the same name?

Building Equivalence Classes

• Start with disjoint, singleton sets:
  – { apples }, { bananas }, { peaches }, ...

• As you gain information about the relation, UNION sets that are now related:
  – { peaches, plums }, { apples }, { bananas }, ...

• E.g. if peaches R limes, then we get
  – { peaches, plums, limes, oranges, lemons }
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\}, \{1,6\}
  - Find(1) = 5
  - Find(4) = 8

Example

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

- Find(8) = 7
- Find(14) = 20
- Union(7,20)

Cute Application

- Build a random maze by erasing edges.

Cute Application

- Pick Start and End
Cute Application

• 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

A 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 $E = \{(1,2), (1,7), (2,8), \ldots\}$ 60 edges total.

<table>
<thead>
<tr>
<th>Start</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>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>End</td>
</tr>
</tbody>
</table>
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 $E$
$u := \text{Find}(x)$;
$v := \text{Find}(y)$;
if $u \neq v$ then
  Union($u,v$)
else
  add $(x,y)$ to Maze
All remaining members of $E$ together with Maze form the maze

Example Step

1. Pick (8,14)

Example

1. 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: find and union cannot both
  be done in worst-case $O(1)$ time
Implementing the DS ADT

• Observation: trees let us find many elements given one root…

• Idea: if we reverse the pointers (make them point up from child to parent), we can find a single root from many elements…

• Idea: Use one tree for each equivalence class. The name of the class is the tree root.

Up-Tree for DU/F

Initial state

Intermediate state

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

Simple Implementation

• Array of indices

Union

\[
\text{Union(up[] : integer array, x, y : integer)} : \{
\text{precondition: x and y are roots/}
\text{Up[x] := y}
\}
\]

Constant Time!
Exercise

• Design Find operator
  – Recursive version
  – Iterative version

Find(up[]): integer array, x: integer): integer {
  //precondition: x is in the range 1 to size/
  ???
}

A Bad Case

Now this doesn’t look good 😞
Can we do better? Yes!

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

2. Improve \textbf{find} so that it becomes even better!
   • Path compression
   • Reduces complexity to almost $\Theta(m + n)$

Weighted Union

• Weighted Union
  – Always point the smaller 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

- 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.
- Can we do better?

Worst Case for Weighted Union

Example of Worst Cast (cont’)

After \( n - 1 = \frac{n}{2} + \frac{n}{4} + \ldots + 1 \) Weighted Unions

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

Elegant Array Implementation

Weighted Union

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

Path Compression

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

```
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)
    }
```

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 ($2^{2^{300}}$)

$\alpha$ shows up in:
- Computation Geometry (surface complexity)
- Combinatorics of sequences

A More Comprehensible Slow Function

log* $x$ = number of times you need to compute log to bring value down to at most 1

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

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

```plaintext
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**

```plaintext
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;
}
```