Disjoint Sets
Union-Find Algorithm

CSE 373
Data Structures & Algorithms
Linda Shapiro
Spring 2013

Today’s Outline

• Announcements
• Today’s Topics:
  › Disjoint Sets (Weiss Chapter 8, except Section 6)

Equivalence Relations

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

Equivalence Classes

• Given an equivalence relation $R$, decide whether a pair of elements $a, b \in S$ is such that $a \mathrel{R} b$.
• 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)**

**Start**

**End**

**A Good Solution**

**Start**

**End**

**Good Solution : A Hidden Tree**

**Start**

**End**

**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.

<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>
</tbody>
</table>

| 789   | 10  | 11  | 12  |
| 13    | 14  | 15  | 16  |
| 17    | 18  |

**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

\( \text{Union}(u,v) \) //knock down the wall between the cells (cells in \( \text{the same set are connected} \))

else

\( \text{add } (x,y) \text{ to Maze} \) //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**

Pick \((8,14)\)

\( S = \{1,2,7,9,13,19\} \)

<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>
</tbody>
</table>

| 789   | 10  | 11  | 12  |
| 13    | 14  | 15  | 16  |
| 17    | 18  |

End
**Find Operation**

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

**Union Operation**

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

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

Union

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

Constant Time!

Find

- Design Find operator
  - Recursive version
  - Iterative version

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

if \text{up}[x] = 0 then return x
else return Find(\text{up}, \text{up}[x]);
}

Recursive

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

while \text{up}[x] \neq 0 do
x := \text{up}[x];
return x;
}

Iterative

A Bad Case

Weighted Union

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

Find(1) \text{n steps!!}
Example Again

\[ \begin{align*}
&1 \quad 2 \quad 3 \quad \cdots \quad 6 \\
&\quad \text{Union}(1, 2) \\
&2 \quad 3 \quad \cdots \quad 6 \\
&\quad \text{Union}(2, 3) \\
&\quad \quad \vdots \\
&3 \quad \cdots \quad 6 \\
&\quad \text{Union}(n-1, n) \\
&1 \quad 3 \quad \cdots \quad 6 \\
&\quad \text{Find}(1) \text{ constant time}
\end{align*} \]

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 \( h' < h \).

\[ n > 2^h \]

\[ \log_2 n > h \]

Find(x) in tree T takes \( O(\log n) \) time.

Can we do better?

Example of Worst Case (cont’)

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

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

Elegant Array Implementation

Can save the extra space by storing the complement of weight in the space reserved for the root.
Weighted Union

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

Path Compression

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

Self-Adjustment Works

```
PC-Find(i : index) {
    r := i;
    while up[r] \neq 0 do //find root//
        r := up[r];
    if i \neq r then //compress path//
        k := up[i];
        while k \neq r do
            up[i] := r;
            i := k;
            k := up[k];
        return(r)
    }
```

Disjoint Union / Find

- 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.