Disjoint Sets

CSE 373
Data Structures & Algorithms
Ruth Anderson
Spring 2008

Today’s Outline

• Admin:
  – HW #4 due – Thurs 5/03 at 11:59pm
    • Print out of code
    • Write-up

• Disjoint Sets (Chapter 8)

Disjoint Set-Definition

• Set
  – A collection of distinct objects (unique in that set)
  – Sorted? Operations?
• Disjoint sets
  – A member of a set is unique among all sets
  – Example: {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}
  Or {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

Building Mazes

• Build a random maze by erasing edges.
Building Mazes (2)

- Pick Start and End

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.)

Building Mazes (3)

- Repeatedly pick random edges to delete.

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 $E = \{(1,2), (1,7), (2,8), (2,3), \ldots\}$ 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>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td></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 // 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 Maze
- All remaining members of $E$ together with Maze form the maze

Example

Example Step

Pick (8,14)

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

Example

S: $\{1,2,7,8,9,13,19\}$

$\text{Find}(8) = 7$

Union(7,20)

Example at the End

S: $\{1,2,3,4,5,6,7, \ldots, 36\}$
Implementing the DS ADT

- $n$ elements.
- Total Cost of: $m$ finds, $\leq n-1$ unions
- Target complexity: $O(m+n)$
  \[ \text{i.e. } O(1) \text{ amortized} \]
- $O(1)$ 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

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

Union Operation

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

---

Simple Implementation

- Array of indices
  
  Up[x] = 0 means x is a root.

  ![Diagram](image)

---

Implementation

```c
int Find(int x) {
    while (up[x] != 0) {
        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:
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**

```
1 2 3 n ...
```
`Union(2,1)`
```
1 2 3 n ...
```
`Union(3,2)`
```
1 2 3 n ...
```
`Union(n,n-1)`
```
1 2 3 n ...
```
Find(1): n steps!!

---

**Weighted Union**

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

```
1 2 3 4 5 6 7
```
W-Union(1,7)
```
1 2 3 4 5 6
```
```
1 2 3 4
```
W-Union(2,1)
```
1 2 3
```
W-Union(3,2)
```
```
1 2
```
W-Union(n,2)
```
```
1
```
Find(1): constant time

---

**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 $h' < h$.

```
T
```
Minimum weight up-tree of height $h$ formed by weighted unions
```
W(T) \geq 2h = 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.

$$n \geq 2^h$$
$$\log_2 n \geq h$$

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