Disjoint Union / Find

CSE 326
Data Structures
Lecture 14

Reading

• Reading
  › Chapter 8

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

Cute Application

• Build a random maze by erasing edges.
Cute Application

- Pick Start and End

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 $E = \{(1,2), (1,7), (2,8), (2,3), \ldots\}$ 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></td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</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 $E$
$v := \text{Find}(y)$;
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

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

Example

Pick (19,20)

<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></td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
</tbody>
</table>

Example at the End
Disjoint Union/Find - Lecture 14

**Up-Tree for DU/F**

- Initial state: 1, 2, 3, 4, 5, 6, 7
- Intermediate state:
  - Root names of each set:
    - 1, 2, 3, 4, 5, 6, 7

**Find Operation**

- Find(x) follows x to the root and returns the root.
  - Find(6) = 7

**Union Operation**

- Union(i, j) - assuming i and j are roots, point i to j.
  - Union(1, 7)

**Simple Implementation**

- Array of indices:
  - Up[] will be as shown:
    - Up[1] = 0 means x is a root.

**Union**

- Union(up[] : integer array, x, y : integer) :
  - //precondition: x and y are roots/
  - 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

```
1 2 3 ... n
Union(1,2)

1 2 3 ... n
Union(2,3)

... 

1 2 3 ... n
Union(n-1,n)
```

Find(1) n steps!!

Weighted Union

- **Weighted Union**
  - Always point the smaller tree to the root of the larger tree

```
1 2 3 ...
W-Union(1,7)
```

Example Again

```
1 2 3 ...
Union(1,2)

1 2 3 ...
Union(2,3)

... 

1 2 3 ...
Union(n-1,n)
```

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 2^{h-1}
```

Worst Case for Weighted Union

\[
\begin{array}{c}
n/2 \text{ Weighted Unions} \\
\end{array}
\]

\[
\begin{array}{c}
n/4 \text{ Weighted Unions} \\
\end{array}
\]
Example of Worst Cast (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

Weighted Union

```c
W-Union(i, j : 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.

Self-Adjustment Works

Path Compression Find

```c
PC-Find(x : index) {
    r := x;
    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)
    }
```
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.

Find Solutions

Recursive

```java
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

```java
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;
}
```