CSE 326: Data Structures
Lecture #17
The Dynamic (Equivalence) Duo: Weighted Union & Path Compression
Henry Kautz
Winter Quarter 2002

Today’s Outline
- Making a "good" maze
- Disjoint Set Union/Find ADT
- Up-trees
- Weighted Unions
- Path Compression

What’s a Good Maze?
1. Connected
2. Just one path between any two rooms
3. Random

The Maze Construction Problem
- Given:
  - collection of rooms: \( V \)
  - connections between rooms (initially all closed): \( E \)
- Construct a maze:
  - collection of rooms: \( V' = V \)
  - designated rooms in, \( i \in V \), and out, \( o \in V \)
  - collection of connections to knock down: \( E' \subseteq E \)
    such that one unique path connects every two rooms

The Middle of the Maze
- So far, a number of walls have been knocked down while others remain.
- Now, we consider the wall between A and B.
- Should we knock it down? When should we not knock it?
Maze Construction Algorithm

While edges remain in E
- Remove a random edge \( (u, v) \) from E
  How can we do this efficiently?
- If \( u \) and \( v \) have not yet been connected
  - add \( u \) to \( E' \)
  - mark \( u \) and \( v \) as connected
  How to check connectedness efficiently?

Equivalence Relations

An equivalence relation \( R \) must have three properties
- reflexive: for any \( x \), \( xRx \) is true
- symmetric: for any \( x \) and \( y \), \( xRy \) implies \( yRx \)
- transitive: for any \( x \), \( y \), and \( z \), \( xRy \) and \( yRz \) implies \( xRz \)

Connection between rooms is an equivalence relation
- Why?

Disjoint Set Union/Find ADT

- Union/Find operations
  - create
  - destroy
  - union
  - find
- Disjoint set partition property: every element of a DS U/F structure belongs to exactly one set with a unique name
- Dynamic equivalence property: Union\((a, b)\) creates a new set which is the union of the sets containing \( a \) and \( b \)

Example

Construct the maze on the right
Initial (the name of each set is underlined):
\{a\} \{b\} \{c\} \{d\} \{e\} \{f\} \{g\} \{h\} \{i\}
Randomly select edge 1
Order of edges in blue

Example, First Step

\{a\} \{b\} \{c\} \{d\} \{e\} \{f\} \{g\} \{h\} \{i\}
find\((b) \Rightarrow b\)
find\((c) \Rightarrow c\)
find\((b) \neq \text{find}(c)\) so:
add 1 to \( E' \)
union\((b, c)\)
Order of edges in blue
**Example, Continued**

{a, b, c, d, f, g, h, i}

Order of edges in blue

**Up-Tree Intuition**

Finding the representative member of a set is somewhat like the opposite of finding whether a given key exists in a set.

So, instead of using trees with pointers from each node to its children; let’s use trees with a pointer from each node to its parent.

**Up-Tree Union-Find Data Structure**

- Each subset is an up-tree with its root as its representative member
- All members of a given set are nodes in that set’s up-tree
- Hash table maps input data to the node associated with that data

Up-trees are not necessarily binary!

**Find**

Just traverse to the root!

**Union**

Just hang one root from the other!

**For Your Reading Pleasure...**
The Whole Example (1/11)
union(b,e)

The Whole Example (2/11)
union(a,d)

The Whole Example (3/11)
union(a,b)

The Whole Example (4/11)
find(d) = find(e)
No union!

While we’re finding e, could we do anything else?

The Whole Example (5/11)
union(h,i)

The Whole Example (6/11)
union(c,f)
The Whole Example (7/11)

find(e)  
find(f)  
union(a,c)

Could we do a better job on this union?

The Whole Example (8/11)

find(f)  
find(i)  
union(c,h)

The Whole Example (9/11)

find(e) = find(h) and find(b) = find(c)  
So, no unions for either of these.

The Whole Example (10/11)

find(d)  
find(g)  
union(c,g)

The Whole Example (11/11)

find(g) = find(h)  
So, no union.  
And, we’re done!

Ooh… scary!  
Such a hard maze!

Nifty storage trick

A forest of up-trees can easily be stored in an array.  
Also, if the node names are integers or characters, we can use a very simple, perfect hash.

up-index:

<table>
<thead>
<tr>
<th>a</th>
<th>1</th>
<th>b</th>
<th>2</th>
<th>c</th>
<th>3</th>
<th>d</th>
<th>4</th>
<th>e</th>
<th>5</th>
<th>f</th>
<th>6</th>
<th>g</th>
<th>7</th>
<th>h</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td>0</td>
<td>-1</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>-3</td>
<td>-1</td>
<td>7</td>
<td>0</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>
Implementation

```c
typedef ID int;
ID up[10000];

ID find(Object x)
{
    assert(HashTable.contains(x));
    ID xID = HashTable[x];
    while(up[xID] != -1) {
        xID = up[xID];
    }
    return xID;
}
```

```c
ID union(Object x, Object y)
{
    ID rootx = find(x);
    ID rooty = find(y);
    assert(rootx != rooty);
    up[rootx] = rooty;
}
```

```
typedef ID int;
ID up[10000];
runtime: \(O(\text{depth})\) or ...
runtime: \(O(1)\)
```

Room for Improvement: Weighted Union

- Always makes the root of the larger tree the new root
- Often cuts down on height of the new up-tree

```
Could we do a better job on this union?
```

```
Weighted union!
```

Weighted Union Code

```c
typedef ID int;

ID union(Object x, Object y)
{
    ID rootx = find(x);
    ID rooty = find(y);
    assert(rootx != rooty);
    if (weight[rootx] > weight[rooty]) {
        up[rooty] = rootx;
        weight[rootx] += weight[rooty];
    } else {
        up[rootx] = rooty;
        weight[rooty] += weight[rootx];
    }
}
```

```
typedef ID int;
```

```
new runtime of union:
```

```
new runtime of find:
```

```
runtime: \(O(\text{depth})\) or \(O(\text{log} n)\)
```

Weighted Union Find Analysis

- Finds with weighted union are \(O(\text{max up-tree height})\)
- But, an up-tree of height \(h\) with weighted union must have at least \(2^h\) nodes

```
Base case: \(h = 0\), tree has \(2^0 = 1\) node
```

```
Induction hypothesis: assume true for \(h < h'\)
```

```
and consider the sequence of unions.
```

```
Case 1: Union does not increase max height.
Resulting tree still has \(2^n\) nodes.
```

```
Case 2: Union has height \(h = 1\), where \(h = \text{height of each of the input trees.}\)
```

```
By induction hypothesis each tree has \(2^n\) nodes, so the
```
```
merged tree has at least \(2^{n+1}\) nodes. QED.
```

Alternatives to Weighted Union

- Union by height
- Ranked union (cheaper approximation to union by height)
- See Weiss chapter 8.

Room for Improvement: Path Compression

- Points everything along the path of a find to the root
- Reduces the height of the entire access path to 1

```
While we're finding e, could we do anything else?
```

```
Path compression!
```
Path Compression Example

Path Compression Code

```
ID find(Object x) {
    assert(HashTable.contains(x));
    ID xID = HashTable[x];
    ID hold = xID;
    while(up[xID] := -1) {
        xID = up[xID];
    }
    while(up[hold] := -1) {
        temp = up[hold];
        up[hold] = xID;
        hold = temp;
    }
    return xID;
}
```

Digression: Inverse Ackermann’s

Let \( \log^{(k)} n = \log (\log \ldots (\log n))^{\ldots} \) \( k \) times

Then, let \( \log^* n \) = minimum \( k \) such that \( \log^{(k)} n \leq 1 \)

**How fast does \( \log^* n \) grow?**

\[
\begin{align*}
\log^*(2) &= 1 \\
\log^*(4) &= 2 \\
\log^*(16) &= 3 \\
\log^*(65536) &= 4 \\
\log^*(2^{65536}) &= 5 \quad (a \ 20,000 \ digit \ number!) \\
\log^*(2^{2^{65536}}) &= 6
\end{align*}
\]

Complex Complexity of Weighted Union + Path Compression

- Tarjan (1984) proved that \( m \) weighted union and find operations with path compression on a set of \( n \) elements have worst case complexity \( O(m \cdot \log^*(n)) \)
  - *actually even a little better!*
- For all practical purposes this is amortized constant time

To Do

- Read Chapter 8
- Written homework #6 – out today
  - due Wednesday, Feb 20th in class
- Homework #6 (word counting project)
  - due Monday, Feb 25th by E-turnin midnight

Coming Up

- Graph Algorithms
  - Weiss Ch 9