CSE 326: Data Structures
Disjoint Sets ADT

Hannah Tang and Brian Tjaden
Summer Quarter 2002

What’s a Good Maze?

Maze Construction Algorithm

- Given:
  - A collection of rooms \( V \)
  - Connections between the rooms (initially all closed): \( E \)
  - We want to build a collection of connections to knock down, \( E' \subseteq E \), such that one unique path connects every two rooms

\[
\text{While edges remain in } E \{
    (A, B) = \text{RemoveRandomWall()}
    \text{if( } A \text{ and } B \text{ have not been connected ) } \{
        \text{Add } (A, B) \text{ to } E'
        \text{Mark } A \text{ and } B \text{ as connected}
    \}
\}
\]

The Problem, Formally

- “If \( A \) and \( B \) have not yet been connected”
  - Are two elements in the same set?
- “Mark \( A \) and \( B \) as connected”
  - Form the union of two sets

Disjoint Sets ADT

- \( \text{Find}(x) \)
  - Returns set identifier
  - \( \text{Find}(x) = \text{Find}(y) \) if \( x \) and \( y \) are in the same set
- \( \text{Union}(A, B) \)
  - Arguments are set identifiers
  - How do we union the sets containing \( x \) and \( y \)?
- \( \text{MakeNewSet}(item) \)
  - Create a new set containing only \( item \)

Disjoint Sets Formal Properties

- Equivalence property
  - Every element of a DS belongs to exactly one set
- Dynamic equivalence property
  - The set of an element can change after execution of a union
Disjoint Sets Even More Formally

- Given a set $U = \{a_1, a_2, \ldots, a_n\}$
- Maintain a partition of $U$, a set of subsets of $U \{S_1, S_2, \ldots, S_k\}$ such that:
  - Each pair of subsets $S_i$ and $S_j$ are disjoint: $S_i \cap S_j = \emptyset$
  - Together, the subsets cover $U$: $U = \bigcup_{i=1}^{k} S_i$
  - Each subset has a unique name

Our Modified Maze Construction Algorithm

While edges remain in $E$

$(A, B) = \text{RemoveRandomWall}()$

if( Find(A) != Find(B) )

$E' = E \cup (A, B)$

Union( Find(A), Find(B) )

Example

Construct the maze on the right

Initially (the name of each set is underlined):

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

Order of edges in blue

Example, continued

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

find(b) ⇒ b
find(e) ⇒ E
find(b) ≠ find(e) so:
add 1 to $E'$
union(b, e)

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

Order of edges in blue

DS ADT Tree Representation

- Maintain a forest of up-trees
- Each set is a tree
- The root of a tree is the set identifier

Find Implementation

Find(x)

- Walk parents of $x$ to the root

Runtime:
Union Implementation

Union(A, B)
- Join the two trees
- Since A and B are already the roots of a tree, this is easy!

Runtime:

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(b)
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.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
<th>(e)</th>
<th>(f)</th>
<th>(g)</th>
<th>(h)</th>
<th>(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>
up-index:

| 1 | 0 | 1 | 0 | 1 | 2 | 1 | 1 | 7 |

Implementation

typedef ID int;

ID Find(Object x) {
    ID xID = HTable[x];
    while(up[xID] != -1) {
        xID = up[xID];
    }
    return xID;
}

ID Union(ID x, ID y) {
    up[y] = x;
}

typedef ID int;

Improving Union

Could we do a better job on this union?

Weighted Union Code

ID Union(ID x, ID y) {
    // If up[x] and up[y] aren’t both
    // -1, this algorithm is in trouble
    if (weight[x] > weight[y]) {
        up[y] = x;
        weight[x] += weight[y];
    } else {
        up[x] = y;
        weight[y] += weight[x];
    }
}

new runtime for Union():

new runtime for Find():

Weighted Union Find Analysis

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

• $\therefore$ 2max height $= n$ and max height $= \log n$
• So, find takes $O(\log n)$

Base case: $h = 0$, tree has $2^0 = 1$ node
Induction hypothesis: assume true for $h < h'$
A merge can only increase tree height by one over the smaller tree. So, a tree of height $h'-1$ was merged with a larger tree to form the new tree. Each tree then has $2^{h'-1}$ nodes by the induction hypotheses for a total of at least $2^h$ nodes. QED.

Improving Find

Wait - what’s there to improve?

While we’re finding y, could we do anything else?
Path Compression!

**Path Compression Code**

```java
ID Find(Object x) {
  // x had better be in
  // the set!
  ID xID = hTable[x];
  ID i = xID;
  // Get the root for
  // this set
  while(up[xID] != -1) {
    xID = up[xID];
  }
  // Change the parent for
  // all nodes along the path
  while(up[i] != -1) {
    temp = up[i];
    up[i] = xID;
    i = temp;
  }
  return xID;
}
```

(New?) runtime for Find():

Interlude: A Tour of Slow Functions

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>64</th>
<th>1024</th>
<th>32768</th>
<th>$2^{20}$</th>
<th>$2^{30}$</th>
<th>$2^{20}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>log</td>
<td>1</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>30</td>
<td>220</td>
</tr>
<tr>
<td>log log</td>
<td>0</td>
<td>2.6</td>
<td>3.9</td>
<td>4.6</td>
<td>4.9</td>
<td>4.9</td>
<td>20</td>
</tr>
<tr>
<td>log log log</td>
<td>0</td>
<td>1.4</td>
<td>1.9</td>
<td>2.1</td>
<td>2.3</td>
<td>2.3</td>
<td>70</td>
</tr>
<tr>
<td>log*</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4.4</td>
<td>4.5</td>
<td>6</td>
</tr>
</tbody>
</table>

Let $\log^k n = \log (\log (\log (\ldots (\log n))))$

$k$ times

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

An Even Slower Function

Ackermann created a really big function $A(x, y)$ with the 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 ($2^{300}$)

$\alpha$ shows up in:

- Computation Geometry (surface complexity)
- Combinatorics of sequences

Complex Complexity of
Weighted Union + Path Compression

Tarjan proved that $m$ weighted union and find operations on a set of $n$ elements have worst case complexity of $O(m \cdot \alpha(m, n))$

For all practical purposes this is amortized constant time:

$O(m \cdot 4)$ for $m$ operations!

In some practical cases, one or both is unnecessary, because trees do not naturally get very deep.

Disjoint Sets ADT Summary

- Also known as Union-Find or Disjoint Set Union/Find
- Simple, efficient implementation
  - With weighted union and path compression
  - Great asymptotic bounds
  - Kind of weird at first glance, but lots of applications