Disjoint Sets

CSE 373 - Data Structures
May 20, 2002
Readings and References

• Reading
  › Chapter 8, *Data Structures and Algorithm Analysis in C*, Weiss

• Other References
Disjoint Set ADT

- **Find**: Given an element, return the “name” of its equivalence class
  - note that we are finding the equivalence class, not the element
- **Union**: Given the “names” of two equivalence classes, merge them into one class
  - may have a new name or one of the two old names
Disjoint Set Example

Equivalence Classes = \{1,4,8\}, \{2,3\}, \{6\}, \{7\}, \{5,9,10\}
Name of equivalence class underlined
Up-Tree Virtual Data Structure

- Each equivalence class (or discrete set) 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.
  - input string → integer

Up-trees are usually not binary!
Example of Find

Find: Just traverse from the node to the root.

```
find(e) = a
```

```
find(f) = c
```
Example of Union

Union: Just hang one root from the other. Now \( \text{find}(f) = c \)
and \( \text{find}(e) = c \)

union(c,a)
An Up-Tree Implementation

- Forest of up-trees can easily be stored in an array “$up$”
- If node names are positive integers or characters, can use a very simple, perfect hash function: $Hash(X) = X$
- $up[X] = \text{parent of } X$;  
  - $= 0$ if $X$ is a root

Array $up$:

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

![Diagram of up-trees](attachment:image.png)
Example of Find

Traverse to the root

Find(e) = a

Runtime = ?

Array up:

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

Hang one root from another

Union(c,a)

Runtime = ?

Now:
Find(f) = c
Find(e) = c

Array up:

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

Change a (from 0) to point to c (= 3)
Example: Union(b,e)
Example: Union(a,d)
Example: Union(a,b)
Example: Union(d,e)

Union(d,e) – But (you say) d and e are not roots!
May be allowed in some implementations – do Find first to get roots
Since Find(d) = Find(e), union already done!

But: while we’re finding e, could we do something to speed up Find(e) next time? (hold that thought!)
Example: Union(h, i)
Example: Union(c,f)

Union(c,f)

Diagram showing the union operation on disjoint sets.
Example: Union(c, a)
An Implementation of Find

```c
int Find(int X, DisjSet up) {
    // Assumes X = Hash(X_Element)
    // X_Element could be str/char etc.
    if (up[X] <= 0)    // Parent is flag value
        return X;      // so X is a root
    else              // else find root recursively
        return Find(up[X], up);
}
```

Runtime of Find: O(max height)
Height of tree depends on the previous Unions that built the particular tree
→ Best case: U(1,2), U(1,3), U(1,4), … O(1)
→ Worst case: U(2,1), U(3,2), U(4,3),… O(N)
An Implementation of Union

```c
void Union(DisjSet up, int X, int Y) {
    // Make sure X, Y are roots
    assert(up[X] == 0);
    assert(up[Y] == 0);
    up[Y] = X;
}
```

Runtime of Union: $O(1)$
Issue with Union\((c, a)\)

Could we do a better job on this Union? What happened to the depth of node e?
Speeding Up: Union-by-Size

- Can we speed things up by being clever about growing our up-trees?
  - Always make root of *larger* tree the new root
  - *Why?* Minimizes height of the new up-tree

Union(c,a) → Union-by-Size
Storing Size Information

- Instead of storing 0 in root, store up-tree size as **negative value** in root node

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1/a</th>
<th>2/b</th>
<th>3/c</th>
<th>4/d</th>
<th>5/e</th>
<th>6/f</th>
<th>7/g</th>
<th>8/h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>-5</td>
<td>1</td>
<td>-2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>
void Union(DisjSet up, int X, int Y) {
    // X, Y are roots containing (-size) of up-trees
    assert(up[X] < 0);
    assert(up[Y] < 0);

    if (-up[X] > -up[Y]) {// X is bigger than Y
        up[X] += up[Y]; // so X is new root
        up[Y] = X; // and Y points to X
    } else { // size of X ≤ size of Y
        up[Y] += up[X]; // so Y is new root
        up[X] = Y; // and X points to Y
    }
}

Union-by-Size Code
Union-by-Size: Analysis

- Finds are $O(\text{max up-tree height})$ for a forest of up-trees containing $N$ nodes
- Number of nodes in an up-tree of height $h$ using union-by-size is $\geq 2^h$

Pick up-tree with max height
- Then, $2^{\text{max height}} \leq N$
- max height $\leq \log N$
- Find takes $O(\log N)$

**Base case:** $h = 0$, tree has $2^0 = 1$ node

**Induction hypothesis:** Assume true for $h < h'$

**Induction Step:** New tree of height $h'$ was formed via union of two trees of height $h'-1$
Each tree then has $\geq 2^{h'-1}$ nodes by the induction hypothesis
So, total nodes $\geq 2^{h'-1} + 2^{h'-1} = 2^{h'}$

$\Rightarrow$ True for all $h$
Union-by-Height

• Textbook describes alternative strategy of Union-by-height
  › Keep track of height of each up-tree in the root nodes
  › Union makes root of up-tree with greater height the new root
• Same results and similar implementation as Union-by-Size
  › Find is $O(\log N)$ and Union is $O(1)$
Find and Path Compression

• M Finds on same element take $O(M \log N)$ time
  › Can we modify Find to have *side-effects* so that next Find will be faster?

• **Path Compression**
  › When we do a Find, we follow a path in the tree from the given element X all the way up to the root
  › The tree does not have to be a binary tree
  › So we can reroot the nodes on the path so that they are all direct children of the root of their tree
Example: Path Compression

Path compression! The next Find(e) will run faster.

Remember splay trees? Similar idea … self adjust to improve future performance based on actual usage.
Another Path Compression Example

Find(e)
Path Compression Code

```c
int Find(int X, DisjSet up) {
    // Assumes X = Hash(X_Element)
    // X_Element could be str/char etc.
    if (up[X] <= 0) // Parent is flag value
        return X; // so X is root
    else // else find root recursively
        return up[X] = Find(up[X],up);
}
```

Make all nodes along access path point to root
New running time of Find?

- Find still takes $O(\text{max up-tree height})$ worst case
- But what happens to the tree heights over time?
  - we are collapsing the tree by having each node point to its root
- What is the \textit{amortized} run time of Find if we do $M$ Finds?

```c
int Find(int X, DisjSet up) {
    // Assumes X = Hash(X_Element)
    // X_Element could be str/char etc.
    if (up[X] <= 0) // Parent is flag value
        return X; // so X is root
    else // else find root recursively
        return up[X] = Find(up[X], up);
}
```
Find Run Time Analysis

- What is the *amortized* run time of Find if we do M Finds?
  - (one or more) operations that take $O(\text{max height})$
  - M-(one or more) operations that take $O(1)$ constant time
  - amortized total cost is $O(1)$ constant time
Slow-growing functions

- How fast does log N grow? log N = 4 for N = 16 = 2^4
  - Grows quite slowly
- Let log^{(k)} N = log (log (log … (log N))) (k logs)
- Let log* N = minimum k such that log^{(k)} N ≤ 1
- How fast does log* N grow? log* N = 4 for N = 65536 = 2^{22^2}
  - Grows very slowly
- Ackermann created a really explosive function A(i, j) and its inverse α(M, N)
- How fast does α(M, N) grow? α(M, N) = 4 for M (≥ N) far larger than the number of atoms in the universe (2^{300})!!
  - grows very, very slowly (slower than log* N)
Find and Union Run Time Analysis

• When both path compression and Union-by-Size are used, the worst case run time for a sequence of $M$ operations (Unions or Finds)
  › Textbook proves $O(M \log^* N)$ time
  › R. E. Tarjan showed $\Theta(M \, \alpha(M,N))$
    • $\alpha(M, N) \leq 4$ for all practical choices of $M$ and $N$

• Amortized run time per operation
  › $= \frac{\text{total time}}{(\# \text{ operations})}$
  › $= \Theta(M \, \alpha(M,N))/M = \Theta(\alpha(M,N))$
  › for all practical purposes: $O(1)$ constant time
Disjoint Set and Union/Find

• Disjoint Set data structure arises in many applications where objects of interest fall into different equivalence classes or sets
  › Cities on a map, electrical components on chip, computers in a network, people related to each other by blood, etc.
• Two main operations: Union of two classes and Find class name for a given element
Disjoint Set and Union/Find

• Up-Tree data structure allows efficient array implementation
  › Unions take $O(1)$ worst case time, Finds can take $O(N)$
  › Union-by-Size reduces worst case time for Find to $O(\log N)$
  › Union-by-Size plus Path Compression allows further speedup
    • Any sequence of $M$ Union/Find operations results in $O(1)$ amortized time per operation (for all practical purposes)