CSE 326: Data Structures
Disjoint Set Union/Find
(part 2)

Steve Seitz
Winter 2009

Data structure for disjoint sets?

- Represent: \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
- Support: find(x), union(x,y)

  - array of lists indexed by name (like hash table)
    find: \(O(n)\)  \hspace{1cm} union: \(O(n)\)
  - hash table over keys, each entry points to
    a regular array  \hspace{1cm} find: \(O(1)\)  \hspace{1cm} union: \(O(n)\)
  - linked list with name at front
    union: \(O(1)\)  \hspace{1cm} find: \(O(n)\)

Union/Find Trade-off

- Known result:
  - Find and Union cannot both be done in worst-case \(O(1)\) time with any data structure.
- We will instead aim for good amortized complexity.
- For \(m\) operations on \(n\) elements:
  - Target complexity: \(O(m)\)  \textit{i.e.} \(O(1)\) amortized

Tree-based Approach

- Each set is a tree
- Root of each tree is the set name.

\[
\begin{array}{c}
\{3,5,7\} \\
\{1,6\} \\
\{2,4\} \\
\end{array}
\]

- Allow large fanout (why?)
Up-Tree for DS Union/Find

Observation: we will only traverse these trees upward from any given node to find the root.

Idea: reverse the pointers (make them point up from child to parent). The result is an up-tree.

Initial state

```
1 2 3 4 5 6 7
```

Intermediate state

```
1  3

2

7

5 4

6
```

Roots are the names of each set.

Find Operation

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

\[ \text{Find}(6) = 7 \]

Union Operation

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

```
Union(1, 7)
```

Simple Implementation

- Array of indices

```
up = [-1  1  1  7  7  5 -1]
```

up[x] = -1 means x is a root.
Implementation

```c
void Union(int x, int y) {
    assert(up[x]<0 && up[y]<0);
    up[x] = y;
}

int Find(int x) {
    while(up[x] >= 0) {
        x = up[x];
    }
    return x;
}
```

runtime for Union: \(\Theta(1)\)

runtime for Find: \(\Theta(n)\)

Amortized complexity is no better.

A Bad Case

```
1 2 3 ... n
```

Union(1,2)

```
2 3 ...
```

Union(2,3)

```
: ...
```

Union(n-1,n)

```
1
```

Find(1) \(n\) steps!!

Two Big Improvements

Can we do better? Yes!

1. Union-by-size
   - Improve Union so that \(\text{Find}\) only takes worst case time of \(\Theta(\log n)\).

2. Path compression
   - Improve \(\text{Find}\) so that, with Union-by-size, \(\text{Find}\) takes amortized time of almost \(\Theta(1)\).
Union-by-Size

Union-by-size
- Always point the smaller tree to the root of the larger tree

Example Again

Analysis of Union-by-Size

- Theorem: With union-by-size an up-tree of height $h$ has size at least $2^h$.
- Proof by induction
  - Base case: $h = 0$. The up-tree has one node, $2^0 = 1$
  - Inductive hypothesis: Assume true for $h-1$
  - Observation: tree gets taller only as a result of a union.

\[
S(T) = S(T_1) + S(T_2) \
\geq 2^{h-1} + 2^{h-1} \
= 2(2^{h-1}) = 2^h
\]

Analysis of Union-by-Size

- What is worst case complexity of $\text{Find}(x)$ in an up-tree forest of $n$ nodes?

\[
n \geq 2^h \\
\log_2 n \geq h \\
\Theta(\log n)
\]

- (Amortized complexity is no better.)
Worst Case for Union-by-Size

n/2 Unions-by-size

n/4 Unions-by-size

Example of Worst Cast (cont’)

After \( n - 1 = \frac{n}{2} + \frac{n}{4} + \ldots + 1 \) Unions-by-size

If there are \( n = 2^k \) nodes then the longest path from leaf to root has length \( k \).

Array Implementation

Elegant Array Implementation

Can store separate size array:

\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\text{up size} & -1 & 1 & -1 & 7 & 7 & 5 & -1 \\
& 2 & 1 & & & & 4 \\
\end{array}
\]

Better, store sizes in the up array:

\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\text{up} & -2 & 1 & -1 & 7 & 7 & 5 & -4 \\
\end{array}
\]

Negative up-values correspond to sizes of roots.
Code for Union-by-Size

\[
S\text{-Union}(i, j)\{
  \text{// Collect sizes}
  si = -up[i];
  sj = -up[j];

  \text{// verify i and j are roots}
  \text{assert}(si >= 0 \&\& sj >= 0)

  \text{// point smaller sized tree to}
  \text{// root of larger, update size}
  \text{if } (si < sj) \{
    up[i] = j;
    up[j] = -(si + sj);
  \} \text{ else } \{
    up[j] = i;
    up[i] = -(si + sj);
  \}
\}
\]

Path Compression

- To improve the amortized complexity, we’ll borrow an idea from splay trees:
  - When going up the tree, improve nodes on the path!
  - On a Find operation point all the nodes on the search path directly to the root. This is called “path compression.”

Self-Adjustment Works

Draw the result of Find(5):
Code for Path Compression Find

PC-Find(i) {
  // find root
  j = i;
  while (up[j] >= 0) {
    j = up[j];
    root = j;
  }
  // compress path
  if (i != root) {
    parent = up[i];
    while (parent != root) {
      up[i] = root;
      i = parent;
      parent = up[parent];
    }
  }
  return(root)
}

Complexity of
Union-by-Size + Path Compression

- Worst case time complexity for...
  - ...a single Union-by-size is: $\Theta(1)$
  - ...a single PC-Find is: $\Theta(\log n)$

- Time complexity for $m \geq n$ operations on $n$ elements has been shown to be $O(m \log^* n)$.
  [See Weiss for proof.]
  - Amortized complexity is then $O(\log^* n)$
  - What is $\log^* n$?

The Tight Bound

In fact, Tarjan showed the time complexity for $m \geq n$ operations on $n$ elements is:

$\Theta(m \alpha(m, n))$

Amortized complexity is then $\Theta(\alpha(m, n))$.

What is $\alpha(m, n)$?

- Inverse of Ackermann’s function.

- For reasonable values of $m, n$, grows even slower than $\log^* n$. So, it’s even “more constant.”

Proof is beyond scope of this class. A simple algorithm can lead to incredibly hardcore analysis!