Data Structures and Algorithms

CS245-2016S-14

Disjoint Sets

David Galles

Department of Computer Science
University of San Francisco
Disjoint Sets

• Maintain a collection of sets
• Operations:
  • Determine which set an element is in
  • Union (merge) two sets
• Initially, each element is in its own set
  • # of sets = # of elements
Elements will be integers (for now)

Operations:
- CreateSets(n) – Create n sets, for integers 0..(n-1)
- Union(x,y) – merge the set containing x and the set containing y
- Find(x) – return a representation of x’s set
  - Find(x) = Find(y) iff x,y are in the same set
Implementing Disjoint sets
• How should disjoint sets be implemented?
Implementing Disjoint Sets

- Implementing Disjoint sets (First Try)
  - Array of set identifiers: 
    Set[i] = set containing element i
  - Initially, Set[i] = i
Creating sets:
Creating sets: (pseudo-Java)

```java
void CreateSets(n) {
    for (i=0; i<n; i++) {
        Set[i] = i;
    }
}
```
Implementing Disjoint Sets

• Find:
14-7: Implementing Disjoint Sets

- Find: (pseudo-Java)

```java
int Find(x) {
    return Set[x];
}
```
14-8: Implementing Disjoint Sets

- Union:
Implementing Disjoint Sets

- **Union:** (pseudo-Java)

```java
void Union(x,y) {
    set1 = Set[x];
    set2 = Set[y];

    for (i=0; i < n; i=+)
        if (Set[i] == set2)
            Set[i] = set1;
}
```
14-10: **Disjoint Sets** \( \Theta() \)

- CreateSets
- Find
- Union
Disjoint Sets $\Theta()$

- CreateSets: $\Theta(n)$
- Find: $\Theta(1)$
- Union: $\Theta(n)$
Disjoint Sets $\Theta(\cdot)$

- CreateSets: $\Theta(n)$
- Find: $\Theta(1)$
- Union: $\Theta(n)$

We can do better! (At least for Union ...)

Implementing Disjoint Sets II

- Store elements in trees
- All elements in the same set will be in the same tree
- \text{Find}(x) \text{ returns the element at the root of the tree containing } x
  - How can we easily find the root of a tree containing \( x \)?
Implementing Disjoint Sets II

- Store elements in trees
- All elements in the same set will be in the same tree
- Find(x) returns the element at the root of the tree containing x
  - How can we easily find the root of a tree containing x?
  - Implement trees using parent pointers instead of children pointers
Trees Using Parent Pointers

- Examples:
Each element is represented by a node in a tree
Maintain an array of pointers to nodes
14-17: **Implementing Disjoint Sets II**

- Each element is represented by a node in a tree
- Maintain an array of pointers to nodes
14-18: Implementing Disjoint Sets II

- Find:
Implementing Disjoint Sets II

• Find:
  • Follow parent pointers, until root is reached.
    • Root is node with null parent pointer.
  • Return element at root
**Find: (pseudo-Java)**

```java
int Find(x) {
    Node tmp = Sets[x];
    while (tmp.parent != null)
        tmp = tmp.parent;
    return tmp.element;
}
```
14-21: **Implementing Disjoint Sets II**

- Union(x,y)
• **Union**($x,y$)
  • Calculate:
    • Root of $x$’s tree, $\text{root}_x$
    • Root of $y$’s tree, $\text{root}_y$
  • Set $\text{parent}(\text{root}_x) = \text{root}_y$
14-23: **Implementing Disjoint Sets II**

- **Union(x,y) (pseudo-Java)**

```java
def Union(x, y):
    rootx = Find(x)
    rooty = Find(y)
    Sets[rootx].parent = Sets[rooty]
```

14-24: **Removing pointers**

- We don’t need any pointers
- Instead, use index into set array

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
14-25: Removing pointers

\[
\begin{array}{cccccccc}
-1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

- Union(2,3), Union(6,8), Union(0,2), Union(2,6)
14-26: Removing pointers

- Union(2,3), Union(6,8), Union(0,2), Union(2,8)

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-1</td>
<td>3</td>
<td>8</td>
<td>-1</td>
<td>-1</td>
<td>8</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |   |
14-27: **Implementing Disjoint Sets III**

- Find: (pseudo-Java)

```java
int Find(x) {
    while (Parent[x] != -1)
        x = Parent[x]
    return x
}
```
14-28: **Implementing Disjoint Sets II**

- Union(x,y) (pseudo-Java)

```java
void Union(x,y) {
    rootx = Find(x);
    rooty = Find(y);
    Parent[rootx] = rooty;
}
```
So far, we haven’t done much to improve the run-time efficiency of Disjoint sets.

Two improvements will make a huge difference:

- Union by rank
- Path compression
14-30: Union by Rank

- When we merge two sets:
  - “Shorter” tree point to the taller tree
14-31: **Union by Rank**

- We need to keep track of the height of each tree
- How?
Union by Rank

- We need to keep track of the height of each tree
  - Store the height of the tree at the root
  - If a node $x$ is not a root, $Parent[x] =$ parent of $x$
  - If a node $x$ is a root, $Parent[x] = 0$ - # height of tree rooted at $x$
Examples
When we merge two trees, how do we know which tree to point at the other?
When we merge two trees, how do we know which tree to point at the other?

- The node with the larger (less negative) Parent[] value points to the node with the smaller (more negative) Parent[] value. Break ties arbitrarily.

How do we update the height of the new merged tree?
When we merge two trees, how do we know which tree to point at the other?

- The node with the larger (less negative) Parent[] value points to the node with the smaller (more negative) Parent[] value. Break ties arbitrarily.

How do we update the height of the new merged tree?

- If trees are different sizes, do nothing
- If trees are the same size, increase (decrease) new parent by 1.
14-37: Union by Rank

- Union(x,y) (pseudo-Java)

```java
void Union(x,y) {
    rootx = Find(x);
    rooty = Find(y);
    if (Parent[rootx] < Parent[rooty]) {
        Parent[rooty] = x;
    } else {
        if Parent[rootx] == Parent[rooty]
            Parent[rooty]--;
        Parent[rootx] = rooty;
    }
}
```
Path Compression

- After each call to `Find(x)`, change x’s parent pointer to point directly at root
- Also, change all parent pointers on path from x to root
Find: (pseudo-Java)

```java
int Find(x) {
    if (Parent[x] < 0)
        return x;
    else {
        Parent[x] = Find(Parent[x]);
        return Parent[x];
    }
}
```
14-40: Disjoint Set

- Time to do a Find / Union proportional to the depth of the trees
- “Union by Rank” tends to keep tree sizes down
- “Path compression” makes Find and Union causes trees to flatten
- Union / Find take roughly time $O(1)$ on average
Technically, \( n \) Find/Unions take time \( O(n \log^* n) \).

\( \log^* n \) is the number of times we need to take \( \log \) of \( n \)
to get to 1.

- \( \log 2 = 1, \log^* 2 = 1 \)
- \( \log(\log 4) = 1, \log^* 4 = 2 \)
- \( \log(\log(\log 16)) = 1, \log^* 16 = 3 \)
- \( \log(\log(\log(\log 65536))) = 1, \log^* 65536 = 4 \)
- \( \ldots \)
- \( \log^* 2^{65536} = 5 \)

- \# of atoms in the universe \( \approx 10^{80} \ll 2^{65536} \)
- \( \log^* n \leq 5 \) for all practical values of \( n \)