Disjoint Sets

R. J. Renka

Department of Computer Science & Engineering
University of North Texas

11/05/2013
Disjoint Set Class

Problem
Given a set $S$ with $N$ elements, store a partitioning of $S$ into disjoint subsets such that operations union and find are efficiently implemented, where $\text{find}(x)$ returns the name of the set that contains $x \in S$, and $\text{union}(x,y)$ replaces the sets containing $x$ and $y$ by their union when $x$ and $y$ are in different sets; i.e., when $\text{find}(x) \neq \text{find}(y)$.

Example applications include Kruskal’s Algorithm and the partitioning of a set of variables into subsets associated with types by a compiler.

8.1 Equivalence Relations The subsets may be thought as equivalence classes associated with an equivalence relation $R$ on $S$, where $R$ is

1. Reflexive: $aRa$ for all $a \in S$,
2. Symmetric: $aRb$ if and only if $bRa$, and
3. Transitive: $aRb$ and $bRc$ implies that $aRc$. 
Assume the elements of $S$ are designated by 0 to $N - 1$. An array containing set names would give constant-time find but require $\Theta(N)$ time for union. A data structure that enables constant-time union represents each set in a forest as a tree stored implicitly in an array $S$ of length $N$. The only information needed for each node is a parent link with -1 as the parent of the root: $S[i] = \text{parent of } i$. The name of the set returned by find is the root of the tree.
**Figures 8.1-8.3**

**Figure 8.1** Eight elements, initially in different sets

![Diagram showing eight elements in different sets](image1)

**Figure 8.2** After union(4,5)

![Diagram showing elements after union(4,5)](image2)

**Figure 8.3** After union(6,7)

![Diagram showing elements after union(6,7)](image3)
Figure 8.4 After $\text{union}(4, 6)$

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<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>$4$</td>
<td>$4$</td>
<td>$6$</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>
</tr>
</tbody>
</table>

Figure 8.5 Implicit representation of previous tree
Figure 8.6

```cpp
1  class DisjSets
2  {
3    public:
4      explicit DisjSets( int numElements );
5
6      int find( int x ) const;
7      int find( int x );
8      void unionSets( int root1, int root2 );
9
10     private:
11     vector<int> s;
12  };

Figure 8.6 Disjoint set class interface
Disjoint set initialization routine

```cpp
1 /**
2 * Construct the disjoint sets object.
3 * numElements is the initial number of disjoint sets.
4 */
5 DisjSets::DisjSets( int numElements ) : s( numElements )
6 {
7     for( int i = 0; i < s.size( ); i++ )
8         s[ i ] = -1;
9 }
```

**Figure 8.7** Disjoint set initialization routine
```cpp
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set names.
 * root1 is the root of set 1.
 * root2 is the root of set 2.
 */
void DisjSets::unionSets( int root1, int root2 )
{
    s[ root2 ] = root1;
}

Figure 8.8 union (not the best way)
Figure 8.9

```cpp
1 /*
2   * Perform a find.
3   * Error checks omitted again for simplicity.
4   * Return the set containing x.
5 */
6 int DisjSets::find( int x ) const
7 {
8     if( s[ x ] < 0 )
9         return x;
10     else
11         return find( s[ x ] );
12 }
```

Figure 8.9 A simple disjoint set find algorithm
The time for $\text{find}(x)$ is proportional to the depth of $x$ — $O(N)$ in the worst case: make the large tree a subtree of a 1-element tree at each step.

A better approach is union-by-size — make the smaller tree a subset of the larger. Then the depth of any node is never more than $O(\log N)$ since a node starts at depth 0 and when its depth increases due to a union, it is placed in a tree that is at least twice as large; hence its depth can be increased at most $\log N$ times.

- Store negative size as the array entry of each root.
- A union sets the new size to the sum of the old sizes.
- A sequence of $M$ operations now has $O(M)$ average time and worst case $O(M \log N)$ time.
The forest above could result from the sequence union(4,5), union(6,7), union(4,6), union(3,4). Without the size heuristic, the tree could be deeper as depicted in Fig. 8.11.
**Figure 8.11** Result of an arbitrary union
Figure 8.12 Worst-case tree for $N = 16$
An alternative is **union-by-height** in which we store the negative height minus one instead of the negative size.

- Make the shallow tree a subtree of the deeper tree.
- Initial values are again -1.
- Height increases by 1 only when two trees of the same height are joined.

<table>
<thead>
<tr>
<th>size</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>4</th>
<th>-5</th>
<th>4</th>
<th>4</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>-3</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
Figure 8.13

```c
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set names.
 * root1 is the root of set 1.
 * root2 is the root of set 2.
 */

void DisjSets::unionSets( int root1, int root2 )
{
    if( s[ root2 ] < s[ root1 ] ) // root2 is deeper
        s[ root1 ] = root2;       // Make root2 new root
    else
    {
        if( s[ root1 ] == s[ root2 ] )
            s[ root1 ]--;          // Update height if same
        s[ root2 ] = root1;       // Make root1 new root
    }
}
```

Figure 8.13 Code for union-by-height (rank)
While linear on average ($O(M)$ for a sequence of $M$ operations) the worst case of $O(M \log N)$ can often occur; e.g., put all the sets in a queue and repeatedly dequeue the first two sets and enqueue the union.

Regardless of the method for union, the worst-case running time can be reduced by performing **path compression** during $\text{find}(x)$: change the parent to the root for every node on the path from $x$ to the root.

Path compression changes heights but rather than recompute them, they simply become estimated heights, referred to as ranks.
Figure 8.14: An example of path compression

The above tree is the result of path compression after `find(14)` on the generic worst-case tree of Fig. 8.12. Two link changes make nodes 12 and 13 one position closer to the root and nodes 14 and 15 two positions closer.
```cpp
/**
 * Perform a find with path compression.
 * Error checks omitted again for simplicity.
 * Return the set containing x.
 */

int DisjSets::find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return s[ x ] = find( s[ x ] );
}
```

**Figure 8.15** Code for disjoint set `find` with path compression