Union-find algorithms

Consider the following problems:

(1) How do you determine whether a vertex x is connected to a vertex y in a graph?

(2) Let vertices correspond to objects and edges mean "is in the same set as". How do you determine whether x is in the same set as y?

(3) Equivalence Problem:

A relation T is defined on a set S if for every pair of elements (a,b), a, b ∈ S, a R b is either true or false. If a R b is true, then a “is related to” b.

An equivalence relation is a relation R satisfying three properties
1. (Reflexive) a R a, for all a ∈ S
2. (Symmetric) a R b if and only if b R a
3. (Transitive) a R b and b R c implies that a R c.

An equivalence class of an element a ∈ S is the subset of S that contains all the elements that are related to S. Every member of S appears in exactly one equivalence class.

Example:
≤ is not an equivalence relation as it is not symmetric.
Let two cities be related if they are in the same country; this is an equivalence relation.
Consider how to solve such problems if we must accept new relations arbitrarily intermixed with questions about which elements are related.

Initially:
N sets, each with one element; the sets are disjoint; all relations are false.
Si = {i} for i = 0 to N-1.

The addition of a new relation is called a union operation and the queries are called find operations.

Find returns the name of the set (i.e., equivalence class) containing a given element.

Are a and b related?
1) t = Find(a)
2) u = Find(b)
3) t == u?

Union merges two equivalence classes into a new equivalence class.

Algorithm: “disjoint set union/find”.

Strategy: find is O(1) or union is O(1); but both cannot be O(1)!
Implementing union and find

Array Implementation:

Array: | x | y |
     (0) (1)

Find: Equivalence class of element 0 is x. // O(1)
Union (0,1): scan the array and change all x’s to y’s // O(N)

Trick: Keep track of the size of each equivalence class and change the name of the smaller class to the larger
⇒ total time for N – 1 merges is O(N lg N). (Each class can have its name changed at most lg N times.)
⇒ total time for M finds and N-1 unions is O(M + N lg N)

Goal: total time for M finds and N-1 unions is a little more than O(M + N)
Represent each set by a tree.

$s[i]$ represents parent of $i$; initially: $s[i] = -1$ for $0 \leq i < 8$.

- the root is used to name the set. If $i$ is a root, $s[i] = -1$.

union(4,5):

union(6,7):

union(4,6):

\[
\begin{array}{cccccccc}
-1 & -1 & -1 & -1 & -1 & 4 & 4 & 6 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}
\]
public class DisjSets
{
    public DisjSets( int numElements )
    {
        /* Figure 8.7 */
        public void union( int root1, int root2 )
        {
            /* Figures 8.8 and 8.13 */
            public int find( int x )
            {
                /* Figures 8.9 and 8.15 */

                private int [ ] s;

                /**
                 * Construct the disjoint sets object.
                 * @param numElements the initial number of disjoint sets.
                 */
    public DisjSets( int numElements )
    {
        s = new int [ numElements ];
        for( int i = 0; i < s.length; i++ )
            s[ i ] = -1;

    /**
     * Union two disjoint sets.
     * For simplicity, we assume root1 and root2 are distinct
     * and represent set names.
     * @param root1 the root of set 1.
     * @param root2 the root of set 2.
     */
    public void union( int root1, int root2 )
    {
        s[ root2 ] = root1;
    }
The worst case time is $O(N)$ since the worst case depth is $N - 1$.

"Average case" is hard to define, but it is not fast enough: sequence of $M$ operations is $O(MN)$. 

```java
1     /**
2     * Perform a find.
3     * Error checks omitted again for simplicity.
4     * @param x the element being searched for.
5     * @return the set containing x.
6     */
7     public int find( int x )
8     {
9         if( s[ x ] < 0 )
10             return x;
11         else
12             return find( s[ x ] );
13     }
```
**Smart Unions:**

*Union-by-size:* make the smaller tree a subtree of the larger, breaking ties by any method.

Result of previous example when union-by-size is used plus union(3,4)

union(3,4) without union-by-size:
If unions done by size, then depth of any node is never more than \( \lg N \).

Find is \( O(\lg N) \).

Sequence of \( M \) operations is \( O(M\lg N) \)

Worst case tree after 16 unions between equal-sized tree:

(Note: this is a binomial tree from Ch. 6).

Implementing this approach:

Keep track of the size of each tree: array entry of root contains \textit{negative} of tree size. When union is performed, check the sizes; sum the sizes.

Sequence of \( M \) operations is average case \( O(M) \), worst case \( O(M\lg N) \).
**Union-by-height**: Keep track of height instead of size. Make shallower tree a subtree of deeper tree.

Height increases only when 2 equally deep trees are joined (and then height goes up by one).

Implementation: store the negative of height, minus an additional 1 (so heights of 0 are negative).

```java
1     /**
2     * Union two disjoint sets using the height heuristic.
3     * For simplicity, we assume root1 and root2 are distinct
4     * and represent set names.
5     * @param root1 the root of set 1.
6     * @param root2 the root of set 2.
7     */
8     public void union( int root1, int root2 )
9     {
10        if( s[ root2 ] < s[ root1 ] ) // root2 is deeper
11           s[ root1 ] = root2;       // Make root2 new root
12        else
13           {
14              if( s[ root1 ] == s[ root2 ] )
15                  s[ root1 ]--;       // Update height if same
16                  s[ root2 ] = root1; // Make root1 new root
17           }
18        }
```
Path Compression

We’ve made unions as fast as possible. Now speed up finds.

Find(x): every node on the path from x to the root has its parent changed to the root.

Given:

Nodes 12, 13, 14, and 15, are closer to root. Costs time, but future finds will be faster?
If path compress is used with arbitrary unions: sequence of M operations requires worst case O(M lg N) time.

If path compress is used with union-by-size, sequence of M operations is average case linear time, as is using union-by-size by itself. But worst case time is improved to “almost linear” (see below).

If union-by-height is used, it is not clear how to recomputed heights efficiently. Instead, use estimated heights, called ranks. Worst case is almost linear: O(N g(N)), where g(N) <= 5 for all N <= 2^{65536}. 

```java
/**
 * Perform a find with path compression.
 * Error checks omitted again for simplicity.
 * @param x the element being searched for.
 * @return the set containing x.
 */

public int find(int x)
{
    if (s[x] < 0)
        return x;
    else
        return s[x] = find(s[x]);
}
```
A program which does a sequence of \( n \) union/finds in any order, called a union-find program, has a \( \Theta(n^2) \) worst case performance since a sequence of unions can result in a long chain of vertices. (\( \Theta(n^2) \) refers to the number of operations on tree links, e.g., change, comparison. Thus, \( n/2 \) unions followed by \( n/2 \) finds \( \Rightarrow n/2 + n/2(n/2 + 1) \) link operations.)

A union-find program of size \( n \) does \( \Theta(n\log n) \) link operations in the worst case if the weighted union and straightforward find are used. A union-find program of size \( n \) does \( \Theta(n\log n) \) link operations in the worst case if the find-with-path-compression and the unweighted union are used. A union-find program of size \( n \) does \( \Theta(n\cdot g(n)) \) link operations in the worst case if the find-with-path-compression and the weighted union are used. Function \( g \) grows very slowly. In fact, \( g(n) \leq 5 \) for all \( n \leq 2^{65536} \).