Disjoint Sets and the Union/Find Problem

1 Equivalence Relations

A binary relation R on a set S is a subset of the Cartesian product $S \times S$. If $(a, b) \in R$ we write aRb and say a ***relates to*** b. Relations can have many properties. An equivalence relation is a symmetric, reflexive, and transitive binary relation. All equivalence relations are isomorphic to the equality relation, and so equivalence may be thought of like the equality relation. An equivalence relation creates a partition of the set S into subsets $S_1, S_2, \ldots, S_n$ such that each set S_i is the transitive closure of the set of items equivalent to a member of S. Conversely, any partition defines the equivalence R defined by aRb iff a and b are in the same subset.

1.1 Examples of equivalence relations

- Two fractions are equivalent if they reduce to the same irreducible fraction.
- For any positive number n, the relationship $p \,\tilde{}\, q$ iff n divides $(p-q)$ is an equivalence relation. This is called *congruence modulo n*.
- Two cities are equivalent if it is possible to walk from one to the other without crossing a bridge.
- Two people are in the same clique if they know each other directly or if each of them knows someone who is in the clique. The world is divided into cliques. There is a conjecture that the whole world consists of a single clique in which the separation is at most six edges.
- Two cells in a maze are equivalent if there is a path from one to the other.
- Two stars are equivalent if they lie in the same galaxy.

2 Dynamic Equivalence Problem

A problem arising in many application areas is the ***dynamic equivalence problem***. Given two members of a set S, we need to know whether they are equivalent. One of the earliest instances of this problem arose in the development of the first compiler, the Fortran compiler. Fortran has an `EQUIVALENCE` declaration that tells the compiler that two or more entities share the same logical storage locations. (C and C++ have this feature too. It is called a ***union***.) As the compiler processes the source code, it needs to construct the sets of equivalent variables. The problem is dynamic because each `EQUIVALENCE` statement is, in effect, a union operation, combining two or more disjoint and inequivalent sets of variables into a single set. As it processes the code, the compiler also needs to identify which variables are sharing the same locations, which means deciding whether two variables are in the same equivalence class.

If S has n elements, we could create an n by n Boolean array to represent this information, by setting `R[i][j] = true` iff $(i, j) \in R$. That solves only half the problem. We also need to change the relationships dynamically so that, for example, we add the relation aRb or remove the relation aRb. Again, the Boolean matrix will provide a fast solution, since we can just turn off the appropriate bits.

Now add the requirement that we should be able to find the set of all elements that are related to a member by obtaining the name of the set to which it belongs and displaying all members of that set. The Boolean matrix no longer provides a fast solution, since we will need to compute the ***transitive closure***[1] of the matrix, at best super-quadratic running time. Alternatively when we add a new relation aRb, we could complete the matrix so that it is transitive, but that turns that operation into an $O(N)$ operation.

A more efficient solution is to represent the equivalence classes as disjoint sets and arbitrarily choose one member of the set as the name of the set. In the dynamic equivalence problem, there are only two operations that must be defined and implemented on this collection of sets: ***find*** and ***union***. Given an element $x \in S$, *find(x)* returns the name of the set containing x. Given the names of two sets x and y in S, *union(x,y)* forms the set union of x and y. The reason that union is important is that union can add new relations to S: if we want to add aRb, we form *union(find(a), find(b))* if a and b are in different sets to start.

If an algorithm can get to see the entire sequence of union and find operations before it processes any of them, as if it were reading them from a file and storing and analyzing them first, it is called an ***offline algorithm***. If an algorithm does not have this opportunity, and must process each instruction immediately when it sees it, without knowing which operations might follow, it is called an ***online algorithm***. This would be analogous to interactive input, in which a user enters the sequence of instructions at a terminal and the algorithm must respond immediately to each entered instruction. The problem we solve in this chapter is the online, disjoint set union/find problem, which is formalized as follows:

Union/Find Problem: Given N disjoint, singleton sets$\{0\}$, $\{1\}$, $\ldots$, $\{N-1\}$, process a sequence of `find()` and `union()` operations on-line, meaning one after the other. The names of the elements are arbitrary; although they are numbers they have no numeric properties, and we could just as easily name them $a, b, \ldots, z$ except that there are just *26* letters but infinitely many integers.

3 A Naive (Inefficient) Solution

One simple solution is to maintain a linear array `S` such that `S[i]` is the name of the set to which element `i` belongs. `find()` will be $O(1)$ but `union()` will be $O(N)$. To compute `union(A,B)` we would scan down the array changing all occurrences of the name `A` to `B` or vice versa. A sequence of *N-1* unions would take $O(N^2)$ steps. Some gain in performance can be achieved by maintaining the size of each set and when performing a union, always renaming the elements of the smaller set instead of the larger one. For example, if set `A` has *10* elements and set `B` has *5* elements, then when `union(A,B)` is processed, every element of `B` would be changed to belong to `A`. This implies that no element can have its set changed more than $\log N$ times, because each time its set is changed, it is part of a set that is at least double the size of the one it was in before the operation. As there are N elements, there are at most $\log N$ doublings that can take place. (There are no operations that undo the unions.) Therefore a sequence of N unions takes at most $O(N \log N)$ time, and a sequence of M finds and *N-1* unions take $O(M + N \log N)$ time.

[1] The transitive closure of an n by n matrix A is the matrix $A^+ = A + A^2 + A^3 + \cdots A^n$.

4 An Efficient Solution

In this approach, the `find()` operation will take more time and `union()` operation will be constant, but the total amortized time for *N-1* unions and M finds will be slightly more than $O(M + N)$.

4.1 Parent Trees

We represent a single set by a ***parent tree***, which is a tree in which the direction of edges is towards the root rather than towards the leaves.

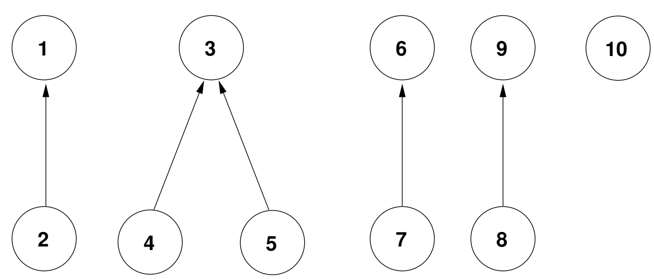
The element stored in the root of the tree is the name of the set containing all elements in the tree.

Since the name of the containing set is the only information we need to maintain for each node, and the root is the name, we can use an array to represent a collection of trees, i.e., a forest. Let `s[]` denote the array.

- If `s[x]`$\neq$ `-1, s[x]` is the index in `s[]` of the parent of `x` in the tree of which `x` is a member.
- The root `s[j]` of a tree will have `s[j] = -1` to indicate that it has no parent.

A `find(x)` operation will require traversing the path from `x` to the root of the tree. If the tree is maintained in an inefficient way, `find(x)` could be $O(N)$, but a very clever solution makes the running time much less.

The collection of sets $\{1, 2\}, \{3, 4, 5\}, \{6, 7\}, \{8, 9\}, \{10\}$ is represented by the following forest:



and the array would look like

	1	2	3	4	5	6	7	8	9	10
	-1	1	-1	3	3	-1	6	9	-1	-1

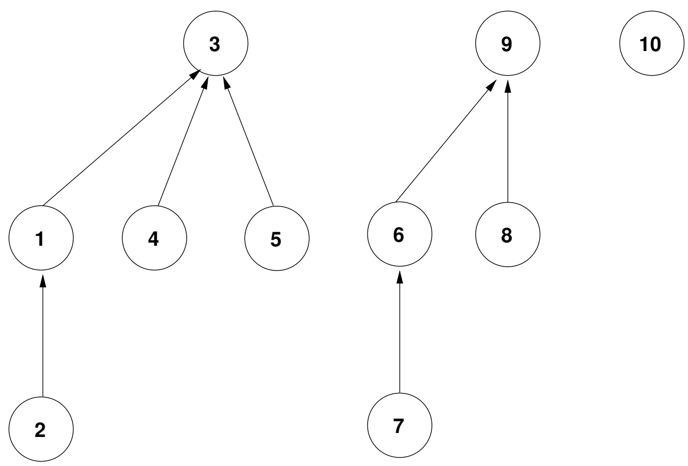
There is a way to include the size of each set without any additional storage cost. Since a `-1` indicates that the array element is the root of the tree, we can make any negative value indicate that it is a root, and its cardinality wil be the size of the set. So the above forest would be represented by the following array:

	1	2	3	4	5	6	7	8	9	10
	-2	1	-3	3	3	-2	6	9	-2	-1

4.2 Smart Union

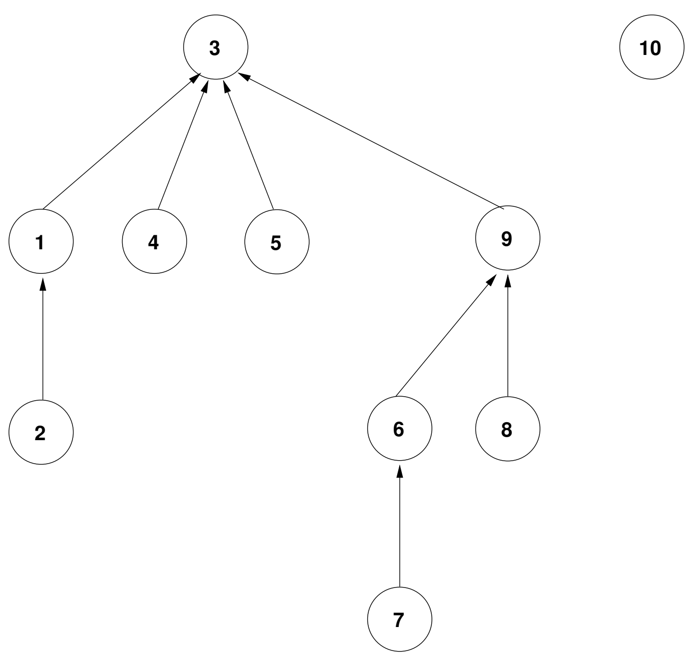
The naïve union operation will simply take the two trees and make one the child of the root of the other. A smarter solution makes the tree with fewer nodes the child of the larger tree. This is called ***union-by-size***. An alternative is to make the shorter tree the child of the deeper tree, which is called ***union-by-height***. Union-by-height is a slight modification of union-by-size.

We present the union-by-size algorithm. In the case that the two sets are the same size, either can be made the child of the other, so some fixed rule can be used. For example, after the smart union-by-size of the trees with roots *1* and *3*, followed by the union of the trees rooted at 6 and 9, the resulting forest and array would look like



	1	2	3	4	5	6	7	8	9	10
	3	1	-5	3	3	9	6	9	-4	-1

because the tree rooted at 3 has 5 nodes (-5) and node 1 now has node 3 as a parent. If we now perform the union of the 3 tree and the 9 tree, the resulting forest becomes



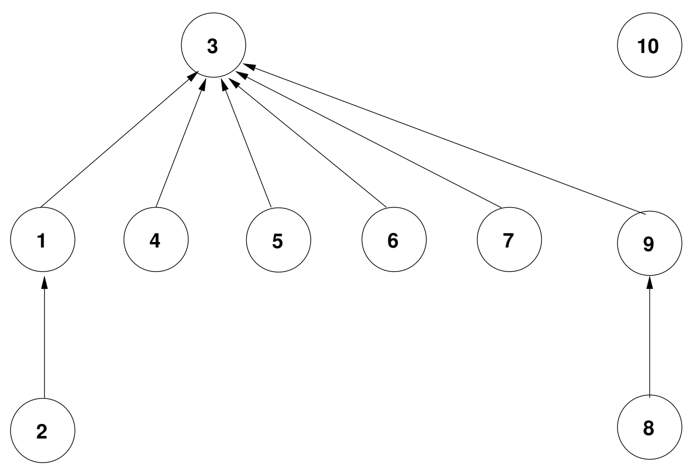
and the array would be

	1	2	3	4	5	6	7	8	9	10
	3	1	-9	3	3	9	6	9	3	-1

It is very trivial to implement either algorithm. The extra work to do this transforms the solution into an extremely efficient one, as we will see soon.

4.3 Path Compression

In most abstract data types, a clean line is drawn between accessors and mutators; operations that access data do not change the state of the object. For example, the `find` operation on search trees and hash tables does not modify those objects. Although `find` is an accessor, a significant improvement in running time can be achieved by allowing it to change the object. Robert Tarjan realized this when he invented the path compression algorithm for parent trees. When path compression is employed in the `find` algorithm, all of the nodes that are visited on the path from the node to the root are turned into children of the root. Thus, after the call `find(7)` on the set above, the forest will look like



and the array for this collection of trees is now

	1	2	3	4	5	6	7	8	9	10
	3	1	-9	3	3	3	3	9	3	-1

It is easy to implement these algorithms. Smart union is very easy. `find()` with path compression uses a clever bit of recursive coding to avoid the need for a stack. It is a little slower than a non-recursive algorithm because of the function call overhead. The `find()` algorithm would be different if union-by-height were used because it would have to recalculate the height of the tree. That is why it is easier to use union-by-size.

Disjoint Sets Class Interface

```
class DisjSets
{
    public:
        DisjSets( int);
        void union( int, int);
        int find(int);
    private:
        vector<int> s;
};
```

Disjoint Sets Class Implementation

```
/* Construct the disjoint sets object.
 * @param numElements is the initial number of disjoint sets.
 */
DisjSets::DisjSets( int numElements ) : s( numElements )
{
    for ( int i = 0; i < s.size( ); i++ )
        s[ i ] = -1;
}

/** Union two disjoint sets.
  * Assume root1 and root2 are distinct and  represent set names.
  * @param root1 is the root of set 1.
  * @param root2 is the root of set 2.
  */
void DisjSets::union( int root1,  int root2 )
{
    if (root1 != root2) {
        if ( s[ root2 ] < s[ root1 ] )  {
            // root2 is deeper
            s[root2] += s[root1];
            s[root1] = root2;
        } else {
            // root1 is deeper
            s[root1] += s[root2];
            s[root2] = root1;
        }
    }
}

/**
 * Perform a recursive find with path compression.
 * Error checks omitted again for simplicity.
 * @param x  is the element to be found
 * @return the set containing x.
 */
int DisjSets::find( int x )
{
    if ( s[ x ] < 0 )
        return x;
    else
```

```
        return s[ x ] = find( s[ x ] );
}
```

This implementation of `find()` makes a recursive call when `x` is not the root of its tree (`s[x]< 0`). In this case, `find()` is called with the parent of `x`, which is `s[x]`, simultaneously moving it one step closer to the root of the tree. The path compression takes place in the return statement, which assigns to `s[x]` the return value of the recursive call, which is the root of the tree to which `x` belongs. Thus, all nodes of the parent tree on the way from `x` to the root are made children of the root of that tree.

4.4 Analysis

The main result about this disjoint sets algorithm using union-by-size and path compression in the `find()` algorithm is that it is extremely fast. To make this precise, we need to make use of a function known as Ackermann's function.

Ackermann's function is a function of two variables with recursion in each variable. It is defined as follows:

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

You can think of this as defining a family of functions of one variable recursively, with each growing succeedingly faster than its predecessor, as follows:

$$\begin{aligned} A_0(n) &= n+1 \\ A_{m+1}(0) &= A_m(1) \\ A_{m+1}(n+1) &= A_m(A_{m+1}(n)) \end{aligned}$$

The function A_0 just adds one to its argument, i.e., $A_0(n) = n+1$. The base case of function $A_1(n)$ is obtained by applying rule 2 with $m = 0$ once:

$$A_1(0) = A_0(1) = 1+1 = 2$$

Then we can apply rules 2 and 3 repeatedly to get

$$\begin{aligned} A_1(n) &= A_0(A_1(n-1)) \\ &= A_1(n-1)+1 \\ &= A_1(n-2)+1+1 \\ &= A_1(n-3)+1+1+1 \\ &\vdots \\ &= A_1(0)+n \\ &= n+2 \end{aligned}$$

The third function, $A_2(n)$ grows faster. Since $A_1(n)$ is $n+2$, we have $A_2(0) = A_1(1) = 1+2 = 3$, and

$$\begin{aligned}
A_2(n) &= A_1(A_2(n-1)) \\
&= A_2(n-1) + 2 \\
&= A_2(n-2) + 4 \\
&= A_2(n-3) + 6 \\
&\vdots \\
&= A_2(0) + 2n \\
&= 2n + 3
\end{aligned}$$

but by the time $m = 4$, the function is astronomical! The following table shows the values for the first few values of m and n.

A(m,n)	n=0	n=1	n=2	n-=3	n=4	n=5
m=0	1	2	3	4	5	6
m=1	2	3	4	5	6	7
m=2	3	5	7	9	11	13
m=3	5	13	29	61	125	253
m=4	13	65533	$2^{65536} - 3$	$2^{265536} - 3$	A(3,$2^{265536} - 3$)	A(3,A(4,4))
m=5	65533	A(4,65533)	A(4,A(4,65533))	A(4,A(5,2))	A(4,A(5,3))	A(4,A(5,4))
m=6	A(4,65533)	A(5,A(4,65533))	A(5,A(6,1))	A(5,A(6,2))	A(5,A(6,3))	A(5,A(6,4))

For all practical purposes, $A_4(n)$, when $n \geq 2$, is an unreachable number, and for $m > 4$ the functions A_m are hard to conceptualize. Now consider the diagonal of this table. It represents Ackermann's function with the two arguments being equal, $A(m,m)$. Let us define $A*(k) = A(k,k)$. Now let $\alpha(n)$ be the generalized inverse of $A*(k)$. This means that $\alpha(n)$ is the largest number k such that

$$A*(k) < n \leq A*(k+1)$$

For example, $\alpha(2) = 0$ because $A(0,0) = 1$ and $A(1,1) = 3$, and $\alpha(62) = 3$ because $A(3,3) = 61$ and $A(4,4) = A(3, 2^{265536} - 3)$. In fact, for all numbers n less than $A(3, 2^{265536} - 3)$, however ridiculously large that is, $\alpha(n) = 3$. In other words, for practical purposes, $\alpha(n)$ is a constant.

We can now state the main result regarding this algorithm.

Theorem. *The running time of a sequence of M unions and finds using path compression and union by size, in the worst case, on a collection of N sets is $O(M\alpha(N))$, where $\alpha(n)$ is the inverse of Ackermann's function.*

We do not include a proof of the theorem, which is quite lengthy. Because $\alpha(n)$ is essentially a constant, the theorem implies that a sequence of M unions and finds takes $O(M)$ time. Put another way, averaged out over the sequence, the time to perform a union or a find is constant.

5 An Application

One simple application of the disjoint set union/find problem is the generation of mazes. Imagine a rectangular M by N grid G in which walls surround all cells, including the perimeter cells. There is an equivalence relation R on the cells of the grid:

Definition. For two cells c and d, cRd is true if andf only if there is a path from c to d (or from d to c since paths do not have direction) in the grid.

Clearly for any cell c, cRc. Also cRd if and only if dRc. Lastly, this is transitive: If there is a path from a to b and there is a path from b to c, then there is a path from a to c. Thus aRb and bRc implies aRc. Therefore this is an equivalence relation and we can think of cells that are related to each other as being members of the same set.

To generate a maze, we create a grid in which every cell is surrounded by four walls. Thus, initially every cell is in a set by itself, so there are MN disjoint sets. If a wall is removed between cells *G[i,j]* and *G[i+1,j]* then these two cells are now part of the same set. This is a union operation. As walls are removed, the sizes of the sets increase.

One can generate a random maze by choosing a wall to remove randomly. By repeatedly removing a random wall until the entrance cell and the exit cell are connected, a maze is formed with at least one path from the entrance to the exit. By repeating until all cells are in the same set, meaning there is a path from any cell to any other cell, a more challenging maze is generated. Clearly, this means that the algorithm stops when the root of some set has MN members.

