Union/Find (Disjoint Sets datastructure)

Let's step away from graphs for a moment and solve the following problem. Consider a collection of n people, each of whom belongs to a particular political party. We need a datastructure to store the assignment of people to parties. The datastructure needs to support just these 2 operations:

- int FIND(int x):
  Given a person, x, returns the leader of x's party.
- void UNION(int x, int y)
  Given two persons, x and y, merges x's and y's parties together under a single leader.

We are going to represent the structure by an array of size n. Let's assign each person a number in the range [0, n) and impose some extra structure upon the parties. Each party member will have a direct superior. The leader will be her own superior. The array, uf[], will contain, for each person, the number of that person's direct superior. This effectively gives us a "party tree", rooted at the leader, where each node has a link to its parent.

We can use this to implement both FIND() and UNION() in linear time.

Example 1: Slow UNION/FIND

```
int FIND( int x ) {
    if( uf[x] == x ) return x;  // x is the leader
    return FIND( uf[x] );
}

void UNION( int x, int y ) {
    uf[FIND( x )] = FIND( y );
}
```

FIND() simply follows the uf[] links up until it reaches the leader and returns. UNION points x's leader to y's leader, effectively merging the two trees. From now on, calling FIND(z) on any member of x's party will return the leader of y's party, as desired. The choice of leader for the new, merged party is unimportant; we could switch x and y and get an equivalent result.

The running time is not so great. UNION() is a one-liner, but it calls FIND() twice, so we need to optimize FIND() to improve the overall performance. Clearly, the limiting factor here is the depth of these trees. We would like to keep the trees as shallow as possible to minimize the number of recursive calls in FIND().

One technique for doing this is called "path compression". Suppose that we call FIND(x), and it returns z - the leader of x's party. What we could do is reset uf[x] to z, so that the next time we call FIND(x), it would return in just 2 iterations. In fact, when executing FIND(x), we will meet all the nodes along the path from x to z, so we can reset all of their superiors to be the root, z.

Example 2: FIND with path compression

```
int FIND( int x ) {
    if( uf[x] != uf[uf[x]] ) uf[x] = FIND( uf[x] );
    return uf[x];
}
```

The first line checks whether the path from x to the leader has length at least 2. If it does, then we reset uf[x] to point to the leader (otherwise, it's already pointing to the leader). Finally, we simply return uf[x].

Well, this doesn't seem like an improvement - FIND(x) still takes linear time because we have to follow the path from x to the leader, and we might meet all of the other nodes along the way. This isn't a fair analysis though because if we call FIND(x) again, its running time will be constant. We need amortized analysis. Suppose we have n people and n different parties to start with (everyone is in a separate party of one). Then we execute k UNION and FIND operations, in some unknown order, with some unknown parameters. It can be shown that with path compression, this will only require O(k*log (n)) time. The main idea for the proof is to count the number of times a given node can be reassigned to point to the root and the number of nodes at a given level in the trees.

If we add another improvement, we can reduce this to "almost linear" time. This one is called "union-by-rank". We will need an extra array that, for each node, will store a rank. The ranks start off at 1, and rank[x] simply equals to the depth of the tree rooted at x. Since in UNION(x, y), we have a choice whether to point x's leader at y's or y's leader at x's, we can now pick the shallower tree and point it at the deeper one, using ranks. This will prevent any tree from becoming deeper than log(n). Therefore, using union-by-rank by itself, without path compression, will also guarantee a running time of O(k*log (n)) for k operations.

If we combine the two, the analysis gets rather hairy because path compression can make deep trees shallow again without changing the rank of the root element. Nevertheless, the effort is certainly worth it because we get a running time of O((k+n)log*(n)), where the log*() (log-star) function is defined as follows:

- log*(x) = 0 if x <= 1;
- log*(x) = 1 + log*(log(x)) otherwise

log*(x) is the number of times you need to take the log() of x repeatedly until you get it down to 1. For all "reasonable" values of n (n smaller than 2^64), log*(n) is less than 5. One proof of this time bound can be found in the Big White book (Cormen *et al.*, "Introduction to Algorithms"), chapter 21 in the 2nd edition. An implementation of UNION() that uses both improvements may look like this.

Example 3: UNION with union-by-rank

```
bool UNION( int x, int y ) {
    int xx = FIND( x ), yy = FIND( y );
    if( xx == yy ) return false;

    // make sure rank[xx] is smaller
    if( rank[xx] > rank[yy] ) { int t = xx; xx = yy; yy = t; }

    // if both are equal, the combined tree becomes 1 deeper
    if( rank[xx] == rank[yy] ) rank[yy]++;

    uf[xx] = yy;
    return true;
}
```

As an extra bonus, UNION() now returns true when the parties of x and y are truly merged during the operation, and it returns false if they started off in the same party and this call had no effect (except for some path compression).

Minimum Spanning Trees (MST)

An unrooted tree (or simply, a tree) is an undirected, connected, acyclic graph. This means that there is exactly one path between every pair of vertices in a tree. A spanning tree of a given graph, G = (V, E), is a tree T = (V, E'), where E' is a subset of E.

A simple way to think about an MST is the following problem. You have a number of towns (vertices) connected by roads (edges) of various lengths. It's expensive to maintain roads, so your task is to eliminate some of the roads in such a way that it is still possible to drive from any town to any other town. Furthermore, the total length of the remaining roads is minimal.

The following greedy algorithm works for this problem. Start with no edges in the spanning tree. Take the shortest edge and add it to the tree - we will use it. Now take the rest of the edges in order of increasing length and add them to the tree as long as they connect two vertices not already connected by the previous edges. Repeat until all vertices are connected. Think of this as growing a forest of trees which get joined together until they become one big spanning tree. Here is an implementation.

Example 4: Kruskal's algorithm

```
int uf[128];

// a structure to represent an edge
struct Edge {
    // the two endpoints and the weight
    int u, v, w;

    // a comparator that sorts by least weight
    bool operator<( const Edge &e ) const { return w < e.w; }
};

// the graph represented as a list of edges
Edge edges[100000];

// the number of vertices and the number of edges
int n, m;

int kruskal() {
    // sort the first m entries in the 'edges' array
    sort( edges, edges + m );

    // initialize the union-find array
    for( int i = 0; i < n; i++ ) uf[i] = i;

    // the number of trees (parties), and the total weight
    int trees = n, sum = 0;

    for( int i = 0; i < m && trees > 1; i++ ) {
        if( UNION( edges[i].u, edges[i].v ) ) {
            // use edge i in the tree
            trees--;
            sum += edges[i].w;
        }
    }
    return sum;
}
```

We are using the last version of UNION(), the one that returns true when the call results in two different parties being merged. First, we sort the edges by increasing weight. Note that sort() works on an array, too. Instead of using an adjacency list or matrix, we simply need a list of edges. Then, we initialize the union-find structure to have each vertex in its own "party". Finally, we scan the list of edges and see if we can use any given edge to merge two different parties. If yes, then we decrement the number of trees (or "parties") and add the edge's weight to the total sum. The final sum is returned as the weight of the minimum spanning tree.

In the final loop, the algorithm selects some of the edges to build a forest, F, that eventually becomes the minimum spanning tree. To prove correctness, we will show the following invariant: "After each iteration of the final loop, F contains edges that are present in some minimum spanning tree of G." In the beginning, F is empty, so the invariant is trivially true. Suppose the invariant is true after iteration i-1. Then during iteration i, if edge number i is not added by the algorithm, we have nothing to prove. If it is added, then the fact that edges[] is sorted ensures us that the new edge (u,v) is the smallest edge that connects two disconnected vertices in V. Consider splitting the whole set of vertices in the graph into two parts, A and B. Let A contain u, and let B contain v. Furthermore, let there be no edge in F that crosses from A to B. This is possible because we are ensured that there is no path from u to v in F.

By assumption, the invariant was true at iteration i-1, so there exists some minimum spanning tree that contains F. In this tree (call it T'), there must be an edge that connects some vertex u' in A to some vertex v' in B. Since u' and v' are disconnected in F, the edge (u,v) must be no heavier than (u',v') (because of sorting). Now let's remove the edge (u',v') from T' and insert the edge (u,v) instead. This gives us a tree T, which is also a spanning tree. It must be a minimum spanning tree because its total weight is no larger than the weight of T'. Hence, adding the edge (u,v) to F satisfies the invariant; namely, there exists some minimum spanning tree (T) that contains all the edges in F and the edge (u,v).

By induction, the invariant is true when the loop terminates. At that point, assuming the original graph was connected, we have trees = 1, which means that we have a set of edges, F, that form a tree and are part of some minimum spanning tree. Hence, F is a minimum spanning tree.

Euler cycles (tours)

Recall that an Euler cycle is a closed walk that visits each edge exactly once. It turns out that there is a remarkably short, linear time algorithm for finding an Euler cycle, if one exists. Suppose G is a connected graph. Then *G has an Euler cycle if the degree of each vertex is even.* The degree of a vertex, v, is the number of times v appears as an endpoint of an edge in G. To prove this claim, we will demonstrate an algorithm that computes an Euler cycle in such a graph. The converse of the claim is also true and is a simple exercise to prove.

First, we will need to intruduce a function that finds a cycle (any cycle) in G, starting at a given vertex, u. It starts with an empty cycle and a graph where each vertex has an even degree and returns a cycle, removing all of its edges from the graph.

Example 5: Greedy Cycle Detector

```
void greedyCycle( int u ) {
    while( true ) {
        int v;
        for( v = 0; v < n; v++ ) if( graph[u][v] ) break;
```

```
        if( v < n ) {
            graph[u][v] = graph[v][u] = false;
            // add the edge (u,v) to the cycle
            u = v;
        } else break;
    }
}
```

The function is very simple – find an edge (u,v) and use it. Using an edge means removing it from the graph and adding it to the cycle. Here is an outline of a proof. Note that if u has an odd degree, then we will always find some edge (u,v) leaving u (because there is at least one edge). Otherwise, we will exit the main loop if the degree of u is zero. At first, u has even degree, just like every other vertex in the graph. After the first iteration, when we find an edge (u,v) and erase it, there will be exactly two vertices with odd degree – u and v, and we will be at v. In every subsequent iteration, there will be exactly 2 vertices with odd degrees – the starting vertex and the current vertex. Eventually, the loop must return to the starting vertex because it clearly terminates (we can't keep removing edges forever), and it must terminate at the original starting vertex – the only vertex that can have even degree when reached in the main loop.

To solve Euler Cycle, pick any starting vertex and call greedyCycle() to compute some cycle in the graph. If the cycle has used all the edges, it's an Euler cycle by definition. Otherwise, there is some vertex, v, on the cycle that has positive even degree. Call greedyCycle(v) on that vertex and insert the new found cycle in place of v in the original cycle to get a larger combined cycle. Repeat until we have no unerased edges left. Note that removing a cycle from a graph whose vertices all have even degree preserves the property of every vertex having even degree. It may, however, make the graph disconnected, but that is not a problem because each connected component of the remaining graph must share a vertex with the current cycle that we are building.

Here is an implementation that uses a linked list and builds all of the cycles recursively and at the same time. Using an adjacency list and some clever iterator manipulations, we can reduce its running time to O(m).

Example 6: Euler cycle in $O(mn)$

```
list< int > cyc;
void euler( list< int >::iterator i, int u ) {
    for( int v = 0; v < n; v++ ) if( graph[u][v] ) {
        graph[u][v] = graph[v][u] = false;
        euler( cyc.insert( i, u ), v );
    }
}
int main() {
    // read graph into graph[][] and set n to the number of vertices
    euler( cyc.begin(), 0 );
    // cyc contains an euler cycle starting at 0
}
```

The function euler() operates on a linked list. It has an iterator that points to the place in the list where we want to insert the next vertex. u is the vertex we have just arrived at. If u has no neighbours, we are done. Otherwise, we insert u into the linked list, erase the edge (u,v) and recurse on the neighbour, v. This recursive call could either lead us to the starting vertex (think of the first call to greedyCycle()) or back to u (giving a cycle to be inserted in place of u). These are the only two possibilities because only u and the starting vertex have odd degree at any given moment.

How would you change euler() if the graph were directed? What if it contained duplicate edges?