

## 11 Sets and the Union Find Algorithm

There are many instances in which we have a *set*, or a *bag*, or some similar object that must be manipulated.

A classic example would be a collection of sets. When working with sets, we need to be able to

- determine whether an element  $x$  is a member of a set  $S$
- take the union of two sets  $S$  and  $T$
- deal with the unordered-ness of elements in sets

Most algorithms rely on some sort of “less than” condition that allows one to bound a search, to keep elements in sorted order, and/or to imply a transitivity from a local property to a global property (for example, the root of a max-heap is not just the largest element of the three element subtree of itself and its two children, it’s also the global max of the entire heap).

In the absence of an ordering property, how do we handle data efficiently?

When in doubt, use a linked list, but that’s inherently inefficient. Consider doing set union of  $S$  and  $T$  represented as linked lists. Very expensive to remove the dups and make sure we only have each element occurring once.

**Definition 11.1.** An equivalence relation is a binary relation  $R$  on a set  $S$  that is

- *symmetric*:  $aRb \implies bRa$
- *reflexive*:  $aRa$  is true
- *transitive*:  $aRb$  and  $bRc \implies aRc$

**Examples:**

Equality of elements

Membership in a set

Congruence classes of the integers modulo an integer  $m$

Membership in a cycle/clique/connected component in a graph

Consider the last of these as a good dynamic example: lots of sensors dropped on a battlefield (or, what may be almost the same thing, the Chicago freeway system at rush hour). Sensors break, lose battery charge, get run over, so the network is dynamic.

Invariably, what we will want to have will be a *canonical representative* for each equivalence class. This will be our “handle” on the class, analogous to the root of a max-heap.

A **disjoint-set data structure** maintains a collection  $S = \{S_1, \dots, S_k\}$  of disjoint dynamic sets. Each set will be identified and manipulated through a *representative*. It doesn't really matter how the representative is chosen, as long as we can deal with it in a predictable way.

We need at least three operations:

- **Make-Set**( $x$ ) to create a new set  $S$  in the collection, with the only member of  $S$  being a new element  $x$
- **Union**( $S, T$ ) to create the union of sets  $S$  and  $T$  and return a representative for the union
- **Find**( $x$ ) to find the representative for the set of which  $x$  is a member

Linked lists work, but they are inefficient.

The **Union-Find** algorithm is an algorithm for constructing a data structure with online data for which testing the equivalence relation is “fast.”

We start with a graph  $(G, E)$  of nodes  $G$  and edges  $E$ . Each node starts as the root of its own tree.

The **Find** operation chases pointers from a node until it finds the root.

If we issue a **Find** for two nodes  $g$  and  $h$ , and we chase pointers to different roots, and yet these two nodes are supposed to be equivalent, then we do a **Union** operation to join the two trees together into one.

One thing that is a big deal is that the representation of the graph can have a major effect on the structure that is created, as the next examples show.

## 11.1 Example: Set Membership

Let's assume that we have sets  $\{A, B, C, D, E, F, G, H\}$  and  $\{J, K, L\}$  to represent in a data structure. If we issue "edges"  $AB, BC, CD, DE, EF, FG, GH, JK, KL$  in order, then we get the structures:

$A$     $B$     $C$     $D$     $E$     $F$     $G$     $H$     $J$     $K$     $L$

then

$A$     $C$     $D$     $E$     $F$     $G$     $H$     $J$     $K$     $L$

$B$   
then

$A$     $D$     $E$     $F$     $G$     $H$     $J$     $K$     $L$

$B$

$C$

then

A |

E |

F |

G |

H |

I |

J |

K |

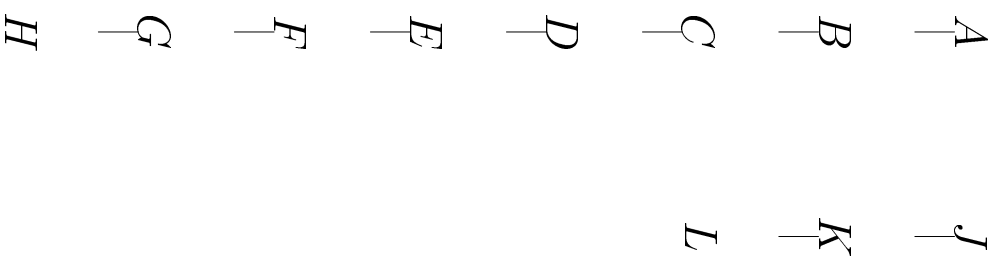
L |

B |

C |

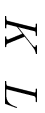
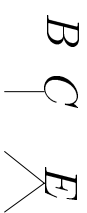
D

and eventually



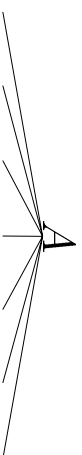
This “works” but is clearly inefficient. We chase more pointers than is perhaps necessary to get from  $G$  to  $A$ .

If instead we presented “edges”  $AB, CD, EF, GH, AC, EG, AE, JK, JL$ , then we might get



*H*

which would have better depth searching (“find”) characteristics. And if searching was a major priority, and this was a set operation so that membership in the tree was the highest priority issue, then probably a flat structure would be best:



*B C D E F G H*

*K L*

## 11.2 Example: Cycles in graphs

Now consider a graph with nodes

$$\{A, B, C, D, E, F, G, H\}$$

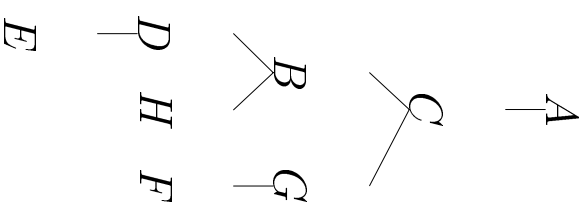
and edges

$$\{AC, BC, BD, BH, CD, CG, DE, DG, DH, FG, GH\}.$$

This graph has cycles with nodes  $BCD, BDH, CDG, DGH, BCDH, BCGH, BCDGH$ .

We can generate a **spanning set of cycles** such that the **sums of cycles** comprise all the cycles by using a union-find algorithm. (For sums of cycles, consider the following: trace out the cycles with an orientation, perhaps with  $BCD$  and  $BDH$  as  $B \rightarrow C \rightarrow D \rightarrow B$  and  $B \rightarrow D \rightarrow H \rightarrow B$ . Then the  $D \rightarrow B$  and the  $B \rightarrow D$  will “cancel”, leaving the cycle  $BCDH$  as  $B \rightarrow C \rightarrow D \rightarrow H \rightarrow B$ .)

If we present the edges as above, and do no path compression, then the final union-find tree for this connected component of the graph will be as follows.



As we add edges, when we try to add  $CD$ , this would produce the cycle  $BCD$  (we can tell this because the “find” will return the same root for the  $C$  node and the  $D$  node. Instead of adding the edge, we write out the cycle and discard the edge. Similarly, when we try to add the  $DG$  edge, we will get a cycle, and when we try to add the  $GH$  edge, we will get a cycle.

This gives a set of three cycles  $C_1, C_2, C_3$ . There are three sums of three cycles ( $C_1 + C_2, C_1 + C_3, C_2 + C_3$ ), and so six cycles in all.

**Issues:** We want low depth, but sometimes we also want the structure of the graph to be preserved. It is relevant to consider the number of nodes and number of edges, as well as the time to build the forest of trees compared with the time to search the forest of trees. And many variations exist on pointer jumping techniques for improving the “find” process.

**Path Compression:**