Union-Find

The disjoint sets (union find) data structure organizes a collection of elements into disjoint sets. Every element is in a set, and each set has a name, or a representative, an element in the set that the set is referred to.

**Make-Set**(*x*) creates a new set with one member *x* and with representative element *x*. *x* cannot be in another set, as all sets are disjoint.

**Union**(*x*, *y*) unions the two sets containing *x* and *y*. Let *S_x* be the set that contains *x*, and *S_y* be the set that contains *y*. If these two sets are not disjoint, then they must be the same set (because non-disjoint sets are not allowed), and thus no change is made. Otherwise, a new set *S* is created with the elements in *S_x* ∪ *S_y*, with any valid representative element. The original sets *S_x* and *S_y* are destroyed.

**Find-Set**(*x*) returns a pointer to the representative element of the set containing *x*. If *x* and *y* belong to the same set, **Find-Set**(*x*) will return the same element as **Find-Set**(*y*).

There are two main ways to implement union-find: linked lists and forests. Using linked lists, each list corresponds to a set, the head of a list is the representative element of the set, and every element in the list points to the next element as well as the head of the list.

Using forests, we represent each set as a rooted tree. The root of the tree is the representative element, and each element points to its parent. The root’s parent is itself. Note that when implementing UF this way, you can just keep an array of parent pointers *p*[1...n] such that *p*[i] is the parent of *i*. This makes the implementation rather straightforward.

**Problem.** What is the running time of the above three operations if linked lists are used to represent sets? What if trees are used?

At first glance, it seems forests have suboptimal runtime. However, we can introduce two improvements that will make the runtime of using forests asymptotically better than using linked lists.

**Union by rank** We define the rank of a node to be the depth of the node - with the restriction that the rank does not decrease, even if the depth of the node decreases. Let *T_x*, *T_y* be the two roots of the trees *x* and *y* belong to. If the rank of *T_x* is smaller than *T_y*, then *T_x* becomes a child of *T_y*, and vice versa. If they have the same rank, then we arbitrarily make one the child of the other, and the rank of the parent is incremented. This has the effect of always joining a shorter tree to a larger tree.

**Path compression** Whenever we call **Find-Set**(*x*), we update *x*’s parent to be the root of the tree (the representative element). This has the effect of flattening a tree so that successive calls to **Find-Set**(*x*) take constant time.

It can be proven that a series of *m* **Make-Set**, **Union**, and **Find-Set** operations on *n* elements takes worst case *O(mα(n))* time, where *α(n)* is the inverse Ackermann function. \[ \alpha(n) \] grows incredibly slowly, and in practice, never exceeds the value 5. Thus the amortized time taken for a single operation is almost constant!

The proof is incredibly complex, and is covered in section 21.4 of the textbook.

[^1]: https://en.wikipedia.org/wiki/Ackermann_function#Inverse
Discussion Question

- Is it guaranteed that a call to \texttt{Find}(v) will always return the same result throughout the algorithm? If not, is it possible to modify the algorithm such that it does?