Forge Lab 3: Union-Find

Union-Find

Union-find is an efficient algorithm for determining whether two sets of elements overlap, and then merging them. It's often used as a component of larger algorithms that find wide application in networks, computer vision, and data analysis.

The two repeating operations that allow us to do this are:

union, which accepts two elements and, if they currently belong to different sets, merges those sets together; and

find, which returns a name for the partition subset that an input element belongs to.

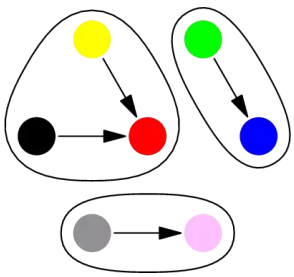
Disjoint-set Data Structure

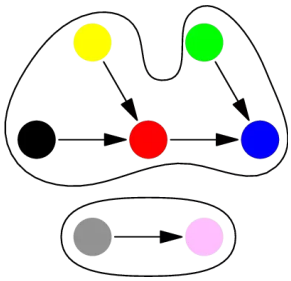
One way to support the above operations is an acyclic directed graph. Elements are represented by nodes in the graph. Each node has a parent, and the parent edges collectively define a path to a distinct root node that is unique to each partition.

Two nodes are therefore in the same partition subset when their root nodes are the same, and root nodes are their own parents. The **find** operation using this data structure would return the root node of a partition.

To union together two nodes A and B, we just set B's root's parent to A's root.

In the below example, we start out with three disjoint subsets, where each node points to its parent. A **union** has been performed in the second picture - the red node, previously the root of the red subset, now is not a root node and is instead a member of the blue root node's subset.





Union preserves Find

In this data structure, we always want the **union** operation to preserve the property that all connected nodes have the same root (we'll call it the **find** property).

In this lab, we'll model the data structure itself as well as two states: before a union operation, and after. Having the two states will allow us to prove that the union operation does indeed preserve the **find** property.

Copy the following template outlining our main sigs, and fill in **defineRoot**.

```
// State is just a point in time for our disjoint-set data structure

// We're only considering two states: pre-union and post-union
sig State {}
one sig BeforeUnion extends State {}
One sig AfterUnion extends State {}
// Our disjoint-sets consists of nodes, which each have:
sig Node {

      parent: set State -> Node, -- one parent (part of algorithm)

      root  : set State -> Node -- one root (abstraction for modeling)

}
pred oneParent {
      /* Fill */
}
pred oneRoot {
```

```
      /* Fill */

}

// Function for parents, makes binary operators easier (i.e.
n.^(findParents[s]))

fun findParents[s: State]: Node -> Node {

        {p1: Node,  p2: Node | p1->s->p2 in parent}

}

// Mock-recursively (inductively) define root of each node in each state

pred defineRoot {

      all s: State, n: Node | {

        -- any node that is its own parent is its own root

        -- otherwise, get its root from its non-self parent (recur)

        /* FILL */

}}
```

Now that we have our disjoint-set data structure, we can define:

- the union operation (join two disjoint subsets by, as said above, making one root have the other root as its parent), and
- our notion of a well-formed, "findable" state (where a node is connected to another node if and only if they have the same root, and the graph has no cycles).

Copy the following template, and fill in the definitions of union and find. There are some extra commands to help you sanity check your work, but the goal of this part is to get both checks to pass.

```
// A union event joins two pre-state nodes in the post-state

pred union {

      some n1, n2: Node | {


            let oldRoot = n2.root[BeforeUnion] |
```

```
            let newRoot = n1.root[BeforeUnion] | {

            -- set n1.root as parent of n2.root, no other parents altered

        /* FILL */

        }
}}
// See if the union operations look correct to you before formally checking

run {union and (all n: Node | one n.root[BeforeUnion])} for 5 Node, 2 State

// If our union operation is correct, find will preserved between pre and
post
-- find is true of a state if that state is *well formed*
pred find[s: State] { all n1: Node, n2: Node- n1 | {
        -- cycles should not exist in a clean find state
        (n1.parent[s] != n1 implies n1 not in n1.^(findParents[s]))
        -- find expects all connected nodes to have the same root,
        -- and all disjoint nodes to have different roots
        sameRoot[n1,n2,s] iff connected[n1,n2,s]
}}
pred sameRoot[n1: Node, n2: Node, s: State] {
        -- nodes n1 and n2 have the same root in state s
        /* FILL */
}
pred connected[n1: Node, n2: Node, s: State] {
```

```
        -- nodes n1 and n2 are connected if n1 can reach n2 and n2 can reach
n1

        -- hint: use (with other things) the ~ of the findParents function


        /* FILL */

}

unionPreservesFind : check { (find[BeforeUnion] and union) implies
find[AfterUnion] } for 5 Node, 2 State

pred isInitialState[s: State] {

      -- initial state if every node is its own parent

        /* FILL */

}

initialStateWellFormed : check {


      all s: State |


            isInitialState[s] implies find[s]} for 5 Node, 2 State
```

By proving that the union operation preserves the find property on the well-formed initial state BeforeUnion to the next state AfterUnion, we've proved inductively that union-find correctly determines if sets overlap and merges them.

Optional: Just for fun

This doesn't actually fully prove that union preserves find for **any possible set** of elements. Think about what is missing. Hint: it's nothing wrong with the argument, or the Forge specification. It's the bounds...

Check-off

Call over a TA and execute your **unionPreservesFind** and **initialStateWellFormed** assertions.

Addendum: Applications

Union-find is most commonly known as a component of Kruskal's algorithm, which finds minimum spanning trees (MSTs). It's used to decide whether or not to add a given edge to the spanning tree being built - if it connects two disjoint sets of nodes, it's added; otherwise, it would cause a cycle since it connects at least two nodes that are already connected.

One application of Kruskal's is the data analysis technique of *k-clustering*: grouping data points into k groups where the minimum distance between groups is maximized. We start with every data point as a node in a graph, each in their own "group". We merge groups by drawing an edge between the two most similar data points that are in different groups, and stop when we have k total groups.