Disjoint Set Union/Find Algorithms

Amod Jog

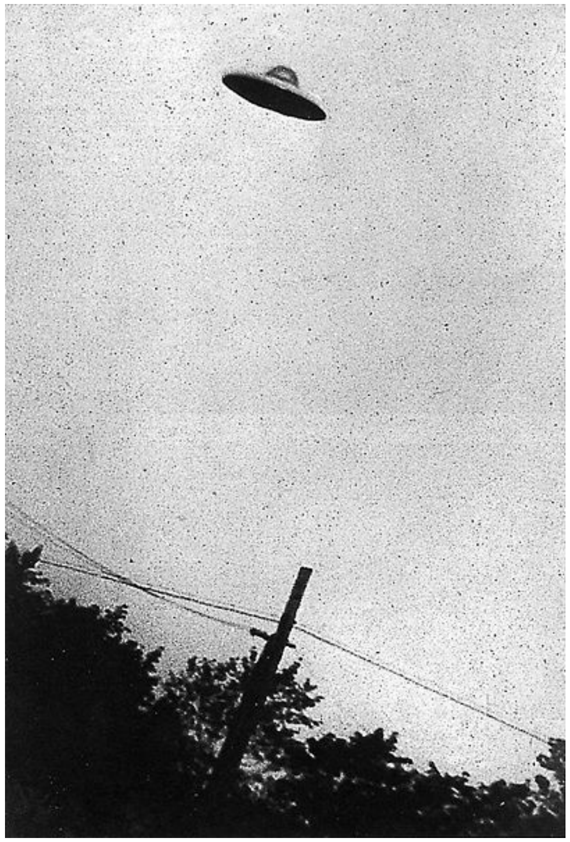
04/19

The Backdrop

- In a remote community close to Roswell, NM there lived 3 families
- The Reds, the Blues and the Greens
- The total population was 10

As Is the Norm…

- Entire population is abducted by aliens
- … and released the next morning!



The Cover-up

But They Took it Too Far..

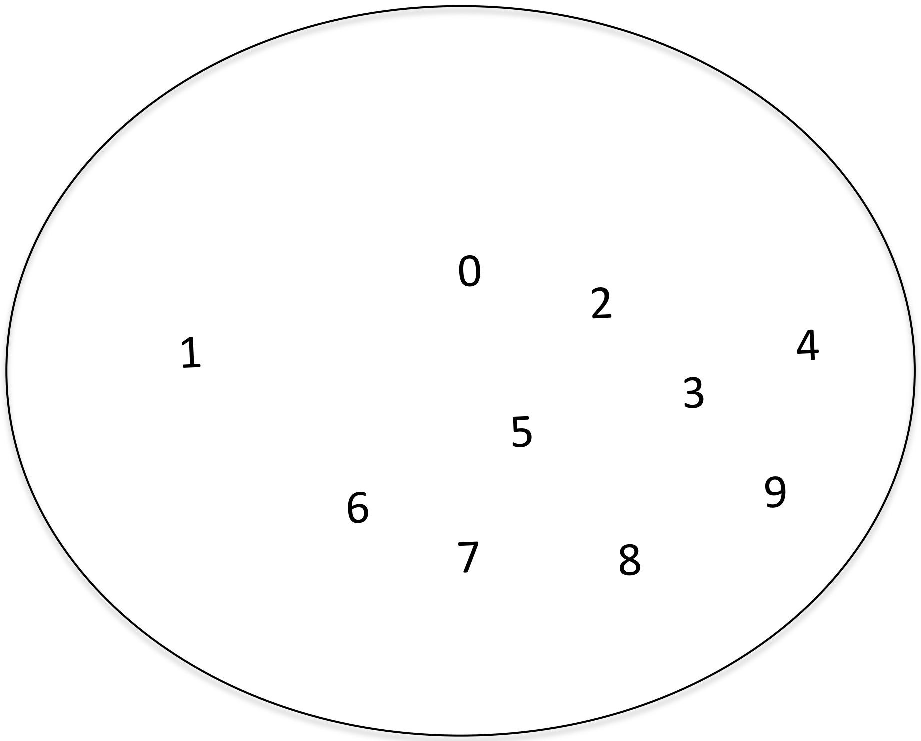
- None of Reds/Blues/Greens now remember who they are or who their family members are
- You set up a DNA lab for matching family members
- A person i is related to person j if their DNAs are close enough
- A person i is his own relative (reflexive)
- If i is j's relative then j is also i's relative (symmetric)
- If i and j are relatives and j and k are relatives, then i and k relatives too (transitive)
- This is an equivalence relation!

The Problem

- Partition the 10 members into 3 separate families based on their relations (as given by the DNA test!)
- Test all pairs? Too costly (~$1000 per test)
- Instead, use the equivalence relation to infer some relationships!
- If p is a relative of q and q is a relative of r, then no need to test p and r

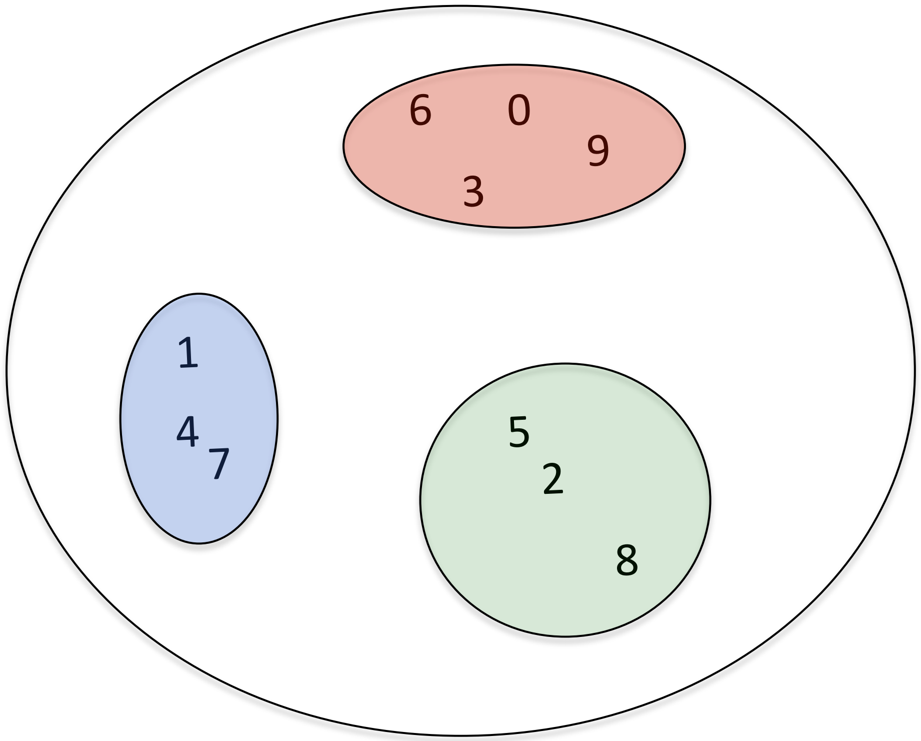
Input

- Lab takes all samples and starts testing them
- Declares results and updates relationships based on discovered relations
- Each person given an integer ID
- We can only query if if two samples are related or not



Expected Output

- Partition the input into disjoint sets
- Everything within a set is related
- Basically reunite the families!
- They may never know their previous names but at least they will be together



Online Algorithm

- Lab declares a result: p and q are related
- Relationships and sets get updated
- Lab declares next result, and so on until everyone is in a separate family
- We will assume that there is no single person family in our example

find() and union()

- If the DNA lab declares that p and q are related, find() will locate which sets p and q belong to
- Union() will create a new set which is a union of these two sets, if they are different
- Hence the algorithm is known as disjoint set union/find algorithm

Algorithm 1: Naïve Approach

- Store the name of the set to which an element belongs to, with the element
- Find() is O(1)
- Union() is O(N), need to scan the entire array to change set ID
- Maximum N-1 unions possible
- So total time for N-1 unions
  - $O(N^2)$
- For M finds and N-1 unions
  - $O(M + N^2)$
  - Can be made into O(M + NlogN)

Element ID	Set ID
0	0
1	1
..	
3	3
..	
6	3
9	3

Trees for Data Structures

- Root is the Set ID, all members from the same tree have the same root and hence the same set ID
- Find() looks for the parent, if it is the root, return the root, else return find(parent)
- Union(root1, root2) takes distinct roots (as sets) and makes one a subtree of the other, thus making a union

Performance of find() and union()

- Worst case, N-1 deep tree, find() takes O(N) time
- If there are M find() calls, O(MN) in the worst case
- Union() is O(1) since only the root pointer is changed
- Average case analysis is hard, Θ(MN) thought to be realistic enough

Smart Union()'s

- Earlier, union() between two roots was arbitrary
- Can result in deeper and deeper trees
- Bad, because find() needs a lot longer to find the root for the deeper nodes
- Smarter way to join two trees?
- Union-by-size!

Union-by-size

- Always make the smaller tree a subtree of the larger subtree
- Lowest depth for a node will be log N
- Therefore the worst performance for find() is O(log N), and for M finds, O(M log N)
- Union-by-height can also be done

Worst Case for Union-by-size

- When two trees with same sizes are joined every time
- Remember binomial queues?
- This will always be a worst case for union()
- Any way to improve find()?
- A smart way: Path Compression

Path Compression

- Recall that find(node) needs to return the root of node
- The deeper the node, the longer it will take
- So, bring the nodes closer to the root!
- Recall also that find() is recursive, if the parent is not the root, it calls find(parent) until it finds the root
- So, join all nodes on the path to the root, directly to the root

Path Compression Performance

- Can work for any union() strategy, independent of what we use for union()
- Union-by-rank: rank of a node is its estimated height
- Similar performance as union-by-size
- If path compression and union-by-size are both used, for M operations in the worst case it takes O(M* α(M,N)) time
- Proof is rather complicated (in the book)

(Painfully) Slowly Growing Functions

- Worst case performance using path compression and smart unions O(M* α(M,N))
- α(M,N) is inverse of Ackermann's function

  A(M,N)

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

- For all practical purposes α(M,N) <= 4, but still not a constant

Log * function

- For a single variable, N the inverse Ackerman function is log*(N)
- It is the number of times you need to take log of N to make it less than 1
- log*(2) = 1
- log*(4) = 2
- log*(16) = 3
- log*(65536) = 4
- log*(2^{65536}) = 5
- The number of hydrogen atoms in the observable Universe is ~2^{300}

Path Compression Performance

- The union/find algorithm using path compression and a union-by-size is
  O(M log*N)
- For practical sized inputs, it's less than O(5M)
- … but we cannot call it linear theoretically, it is *almost* linear

Moral of the Story

- The families were reunited in almost linear time and everyone lived happily ever after