1 Union Find aka disjoint sets

Lets say you were given a bunch of random elements, and I wanted to be able to union groups together quickly, you would use **union-find**. Example of what it needs to do really quickly.

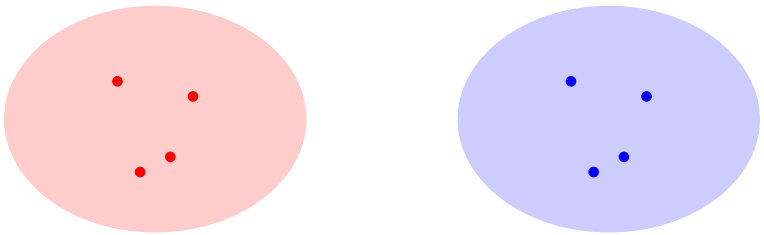


Figure 1: Two disjoint sets before union

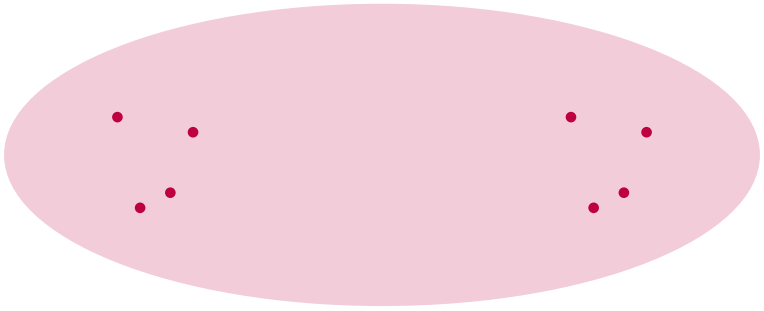


Figure 2: Sets merged after union operation

Note: There are a lot of ways to do this but most are inefficient. What is the brute force attempt? The best way would probably store a map of points to group ids. When you want to union two groups(a,b), you can pick a group(a) and update all mappings of group b to group a. We should not that this $O(n)$ time.

Union-find lets you do this close O(1) to O(logn) time. The basic idea of union-find is to use a tree and to only union on the parent.

Figure 3: Two disjoint sets with tree structures before union

The big idea behind union and find is that groups are stored as trees.

1.1 Writing union find

Union find is actually a very simple algorithm

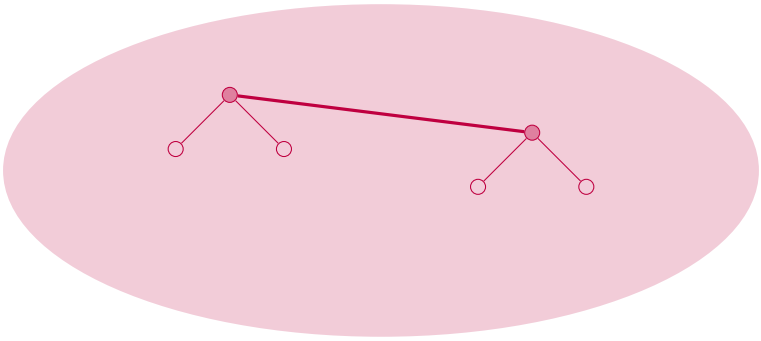


Figure 4: After union: red root becomes parent of blue root, forming a merged tree

Algorithm 1 Union-Find without Path Compression

```
1:  procedure INITIALIZE(n)
2:      for i ← 1 to n do
3:          parent[i] ← i
4:      end for
5:  end procedure
6:  function FIND(x)
7:      if parent[x] = x then
8:          return x
9:      else
10:         return FIND(parent[x])
11:     end if
12: end function
13: procedure UNION(a, b)
14:     rootA ← FIND(a)
15:     rootB ← FIND(b)
16:     if rootA ≠ rootB then
17:         parent[rootA] ← rootB
18:     end if
19: end procedure
```

Explaining the parts

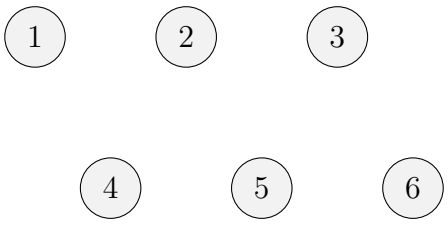
1.1.1 Initialization

Let us assume that every single element is a singleton or the set is just itself.

Algorithm 2 Union-Find without Path Compression

```
1:  procedure INITIALIZE(n)
2:      for i ← 1 to n do
3:          parent[i] ← i
4:      end for
5:  end procedure
```

All we are doing is saying the parent of an element e_i is e_i.



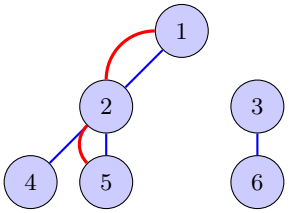
6 elements in their own set.

1.1.2 Find

The whole purpose of this is to find which group a thing is in. This is just a tree traversal going up from an arbitrary node trying to find the parent.

Algorithm 3 Union-Find without Path Compression

```
1: function FIND(x)
2:     if parent[x] = x then
3:         return x
4:     else
5:         return FIND(parent[x])
6:     end if
7: end function
```



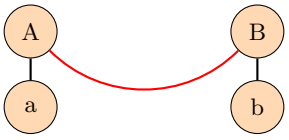
Assuming ur trying to find what group 5 is in, you just traverse up the tree to find the root. It is in group 1.

1.1.3 Union

The union is the core of the algorithm. Lets say you are given elements $a \in A$ and $b \in B$, and you want to union the two sets together. What you need to do is set a to b. But this can for pretty inefficient operations pushing this to $O(n)$.

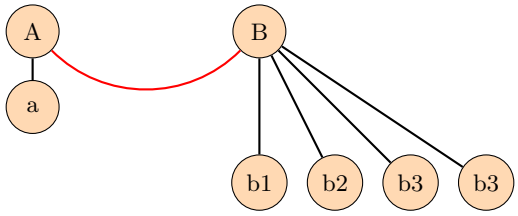
Algorithm 4 Union-Find without Path Compression

```
1: procedure UNION(a, b)
2:     rootA ← FIND(a)
3:     rootB ← FIND(b)
4:     if rootA ≠ rootB then
5:         parent[rootA] ← rootB
6:     end if
7: end procedure
```



2 Path Compression

The traditional naive method is to form the sets as trees and only merge the parent. But we have a common issue found in trees is they tend to form into linked lists reducing the efficiency if not balanced[for example RBT or AVL trees for BSTs]. So we need to have an efficient way to reduce the height of the tree. The great part about sets is you don't really need to care about what else is in the set other than what set the element is in.
Going back on this photo:



Which one would be more efficient, picking A as the root or B as the root? B is obviously the better choice because the amount of children B has is greater than A. By likelihood, we need to do operations on B more. However, our algorithm doesn't care and will arbitrarily union groups together.
There are a lot ways to optimize this, but the two main ways are union by **rank** and union by **size**.
Either choice still follows the programming paradigm of keeping an extra $O(n)$ amount of memory by having another array to store the value system.

2.1 New intialization

The following is the new initialization of the union find data structure **REGARDLESS** of rank or size.

Algorithm 5 Union-Find with Path Compression

```
1: procedure INITIALIZE(n)
2:     for i ← 1 to n do
3:         parent[i] ← i
4:         value[i] ← 1 // for the one element
5:     end for
6: end procedure
```

The value here is either rank or size.

2.2 New union function for rank

Rank is choosing by an upperbound or **heuristic** of how tall the tree is. Rank is not equivalent to the depth. We normally just increment by one on the tree that we add to. The depth of the tree is often a lot smaller than the rank.

Algorithm 6 Union-Find with Path Compression and Union by Rank

```
1:  procedure UNION(a, b)
2:      rootA ← FIND(a)
3:      rootB ← FIND(b)
4:      if rootA ≠ rootB then
5:          if rank[rootA] > rank[rootB] then
6:              parent[rootA] ← rootB
7:              rank[rootA] + 1
8:          else
9:              parent[rootB] ← rootA
10:             rank[rootB] + 1
11:         end if
12:     end if
13: end procedure
```

2.3 New union function for size

Size is the direct calculation of how big a tree is. We choose the larger tree.

Algorithm 7 Union-Find with Path Compression and Union by Rank

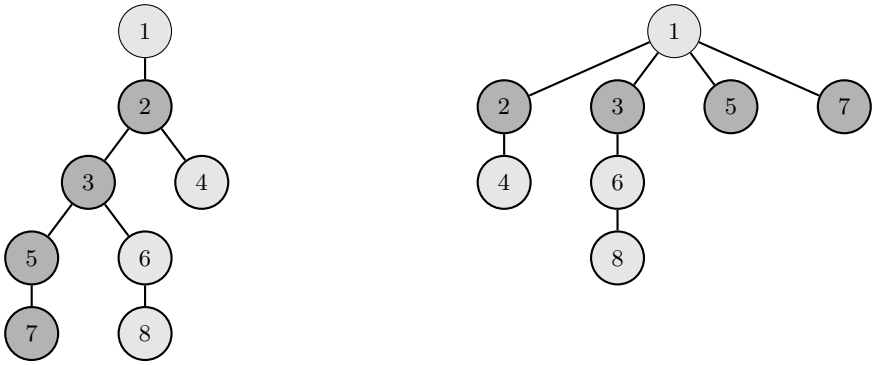
```
1:  procedure UNION(a, b)
2:      rootA ← FIND(a)
3:      rootB ← FIND(b)
4:      if rootA ≠ rootB then
5:          if size[rootA] > size[rootB] then
6:              parent[rootA] ← rootB
7:              size[rootA]+ = size[rootB]
8:          else
9:              parent[rootB] ← rootA
10:             size[rootB]+ = size[rootA]
11:         end if
12:     end if
13: end procedure
```

2.4 Better finds

If we constantly call finds, we can optimize the performance of our tree too by collapsing all the trees down to parent.

Algorithm 8 Union-Find with Path Compression and Union by Rank

```
1:  function FIND(x)
2:      if parent[x] = x then
3:          return x
4:      else
5:          parent[x] = FIND(parent[x])
6:          return parent[x]
7:      end if
8:  end function
```



Before and after one call of find on 7

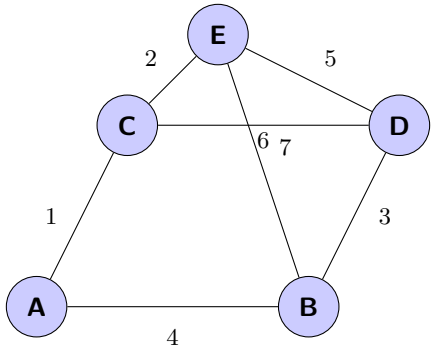
3 Kruskals and Minimum Spanning Trees

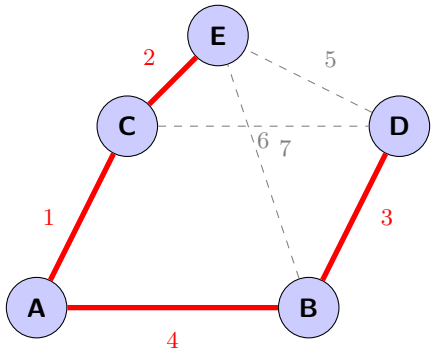
3.1 Minimum Spanning Trees

A minimum spanning tree on a graph G is defined as following:

1. It is a tree that **spans** all vertices
2. The sum of all the edges' weight is minimimized.

For example the following graph has the following minimum spanning tree:





Note: Prims algorithm We have prim's algorithm as a fall back. Prim's algorithm is basically just a modification of dijkstra's.
From last time

Algorithm 9 Dijkstra

```
1: procedure DIJKESTRA(G, s)
2:     initialize an empty priority queue Q
3:     mark s as visited
4:     enqueue s into Q
5:     while Q is not empty do
6:         v ← dequeue from Q
7:         process(v)
8:         for each neighbor u of v in G do
9:             if u is not visited then
10:                mark u as visited
11:                enqueue u into Q with distance d+edge weight
12:            end if
13:        end for
14:    end while
15: end procedure
```

Then prim is just the following:

Algorithm 10 Prim's

```
1: procedure PRIM'S(G, s)
2:     initialize an empty priority queue Q
3:     mark s as visited
4:     enqueue s into Q
5:     while Q is not empty do
6:         v ← dequeue from Q
7:         process(v)
8:         for each neighbor u of v in G do
9:             if u is not visited then
10:                mark u as visited
11:                enqueue u into Q with edge weight
12:            end if
13:        end for
14:    end while
15: end procedure
```

3.2 Kruksal's algorithm

If you know union find, then Kruskal's algorithm is pretty easy. We select the smallest possible edge such that it doens't form a cycle. The intuition here is the following:

1. We need it to be a tree. Essentially you cannot add edges if they form a cycle. We can use union find here.

2. It needs to be minimum. This is by our process of selection.

The proof of correctness is pretty long. Not really worth it unless you are interested.

4 Problems:

1. 2016 Gold https://usaco.org/index.php?page=viewproblem2cpid=669

This problem is literally just kruskals(or prims). Super simple. Given (x,y) cords calculate the MST with the distance formula of $x^2 + y^2$