CSM 61B
Spring 2018

Disjoint Sets

Mentoring Special Topics: March 19, 2018

1 Introduction

The Basics

A disjoint-set or union-find data structure is a method to keep track of a set of items that belong to one or more non-overlapping (i.e. disjoint) sets.

Java has no DisjointSet interface, but if it did, it would probably look like this, with two methods:

```
public interface DisjointSet<E> {
    void connect(E a, E b)
    boolean isConnected(E a, E b)
}
```

Disjoint sets could be used, for example, to keep track of connectedness in a social network. Imagine starting out with a set of people, {Michael, Will, Max, Thong, Sruthi, Sean}. They all do not know each other, so they are each in their own disjoint set: {Michael}, {Will}, {Max}, {Thong}, {Sruthi}, {Sean}.

We write a class that implements the DisjointSet interface for People objects. First, we want to model that Michael meets Sruthi, Thong meets Will, and Thong meets Max. The new disjoint set data structure would represent the sets {Michael, Sruthi}, {Thong, Will, Max}, {Sean}. Each person appears exactly once! The sequence of function calls that would represent these meetings would be the following:

```
connect(Michael, Sruthi)
connect(Thong, Will)
connect(Thong, Max)
```

Connecting two items means merging the set that the first item belongs to with the set the second item belongs to. We can see what is going on in our data structure using isConnected:

```
isConnected(Sruthi, Michael) // This is true
isConnected(Will, Max) // This is true
isConnected(Michael, Will) // This is false
```

Then, if Will meets Sruthi, the final data structure would represent the sets {Michael, Sruthi, Thong, Will, Max}, {Sean}.

```
connect(Will, Sruthi) // This represents Sruthi and Will meeting
isConnected(Michael, Will) // Now this is true
```

For simplicity, in this worksheet we will only be looking at disjoint sets where the items in question are integers, e.g. a set of integers something like {0, 1, 2, 3, 4, 5, 6}, that belong to perhaps three disjoint sets: {0, 1, 2, 4}, {3, 5}, and {6}. Our interface would then just look like this:

```
public interface DisjointSet {
    void connect(int a, int b)
    boolean isConnected(int a, int b)
}
```

Notice that there is no delete method! Once you connect two sets, they cannot be disconnected. Implementing efficient deletions into UnionFind data structures is a question that has piqued the interest of a lot of important computer scientists, and you can read more about it in the Appendix.

When are disjoint sets useful?

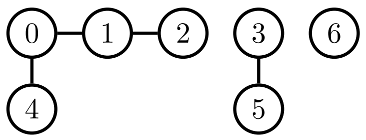
With slight modifications, the disjoint set data structure introduced above can be used to find cycles in a graph. For a basic implementation of a findCycle function, see the Appendix. Disjoint sets will also make a reappearance in 61B when we learn about Kruskal's algorithm.

2 Quick Find and Quick Union

QuickFind looks for a way to improve the run time of checking whether two items are connected, i.e. `isConnected`. Recall from lecture that a natural choice for this improvement is to have an `int[]`, where each index corresponds to an item in our set, and the value for that index represents an "id" of the connected component this item is a part of. If an item of a connected component joins a new set, every item in that connected component must also be in the new set.

2.1 Fill out the following methods according to the given schema for QuickFind. Every item should start off in its own connected component i.e. id[index] = index.

```
public class QuickFindDS implements DisjointSets {
    private int[] id;
    public QuickFindDS(int N) {






    }

    public void connect(int p, int q) {






    }

    public boolean isConnected(int p, int q) {






    }
}
```



Connected components:
$\{0, 1, 2, 4\}, \{3, 5\}, \{6\}$

id[]	0	0	0	3	0	3	6
	0	1	2	3	4	5	6

2.2 Given the example data and implementation from the previous page, answer the following questions. Throughout these questions, assume `connect(p, q)` will make item `p` become a child of item `q`.

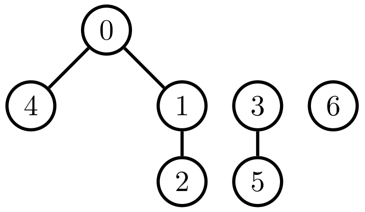
(a) What are the sequence of `connect` calls that will generate the diagram and its corresponding int array from the previous page? Can two different orderings produce the same result?

(b) What will the resulting diagram and `int[] id` look like after calling `connect(2, 5)`?

(c) What is the run time with respect to the number of items `N` of:

`QuickFindDS`: `connect`: `isConnected`:

Now let's try to optimize the `connect` (or union) function, which is the focus of QuickUnion. Recall that we change our set representation such that each value at each index represents the parent of that item. The difference here with this new system is that we no longer need to change every value of every item in a connected component if one of them were to join a new connected component. Rather, we only need to trace the items to their "top level" parent and replace one value with the other. In order to achieve this effect, we have an additional helper method that will find the "top level" parent of a given item.



Connected components:
$\{0, 1, 2, 4\}, \{3, 5\}, \{6\}$

p[]	0	0	1	3	0	3	6
	0	1	2	3	4	5	6

2.3 Implement the following methods according to this new schema for Quick-Union. The constructor and `find` methods are provided to you to get you started.

```
public class QuickUnionDS implements DisjointSets {
    private int[] parent;
    ...
    public void connect(int p, int q) {









    }

    public boolean isConnected(int p, int q) {









    }
}
```

```
public QuickUnionDS(int N) {
    parent = new int[N];
    for (int i = 0; i < N; i++) {
        parent[i] = i;
    }
}

public int find(int p) {
    while (p != parent[p])
        p = parent[p];
    return p;
}
```

2.4 Given the example data and implementation from the previous page, answer the following questions. Throughout these questions, assume `connect(p, q)` will make item `p` become a child of item `q`.

(a) What are the sequence of `connect` calls that will generate the diagram and its corresponding int array from the previous page? Can two different orderings produce the same result?

(b) What will the resulting diagram and `int[] parent` look like after calling `connect(5, 2)`?

(c) Draw the following diagram resulting from this sequence of `connect` calls: connect(6, 5) → connect(5, 4) → connect(4, 3) → connect(3, 2) → connect(2, 1) → connect(1, 0)

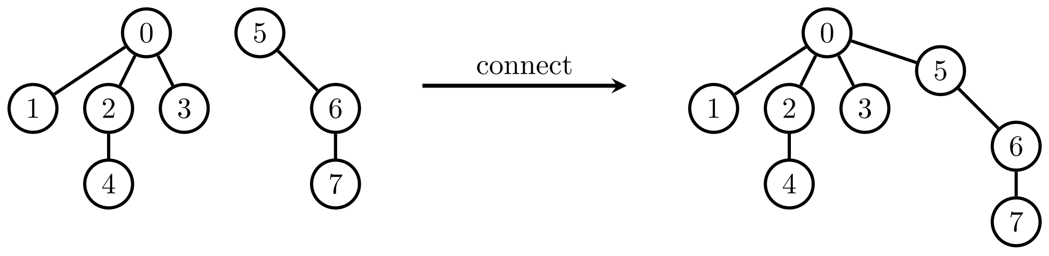
(d) What is the run time with respect to the number of items `N` of:

`find`: `connect`: `isConnected`:

3 Weighted Quick Union

As you may have noticed from Problems 2.4(c) and 2.4(d) of the previous section, our runtime isn't exactly the most efficient it could be! In our current implementation, something is happening that is a bottleneck and slowing us down. What do you think that is? As a hint, our Quick Union is represented as a tree, so what kinds of trees give us poor runtimes? Spindly ones! Traversing a spindly tree from leaf to parent is the slowest possible operation. Is there an operation that causes us to traverse from the leaf of a tree to the parent? The `find` method! We start at a node and climb up the tree until we reach the parent. Convince yourself that a spindly tree will cause this process to be very slow. So, we will improve our data structure by introducing the concept of *weight* or *size*.

The `size` of a disjoint set is defined as the number of elements it contains. So, when we do a `connect()` operation, we will connect the root of the tree with the *smaller* size to the *larger* one. This improves the runtimes of our `connect()` and `isConnected()` operations to $O(\log(n))$ time! To keep track of the sizes of the trees, we will create a new array `size[]` where each root index will contain the size of its tree. The Weighted Quick Union process is exactly like the normal Quick Union process, except for this one extra step that we take when connecting two roots. Even though it is a small change, it has a huge impact on the runtime! An example is shown below:



parent[]	0	0	0	0	2	5	5	6
	0	1	2	3	4	5	6	7
size[]	5	1	2	1	1	3	2	1
	0	1	2	3	4	5	6	7

parent[]	0	0	0	0	2	0	5	6
	0	1	2	3	4	5	6	7
size[]	8	1	2	1	1	3	2	1
	0	1	2	3	4	5	6	7

3.1 Implement the following methods according to this new schema for Weighted Quick-Union. Start with the constructor and implement **find** and **connect**.

```
public class WeightedQuickUnionUF implements DisjointSets {

    public WeightedQuickUnionUF(int n) {

    }
    public int find(int p) {

    }

    public void connect(int p, int q) {

    }
}
```

3.2 Circle the letters corresponding to `id[]` arrays that *cannot* possibly occur during the execution of the weighted quick union algorithm

(a)

```
          i: 0 1 2 3 4 5 6 7 8 9
-----------------------------
A. id[i]: 8 0 4 0 0 4 0 4 2 0
B. id[i]: 4 1 8 2 1 5 1 1 4 5
C. id[i]: 3 3 6 9 3 6 3 4 1 9
D. id[i]: 2 1 1 1 1 1 1 2 1 7
```

(b)

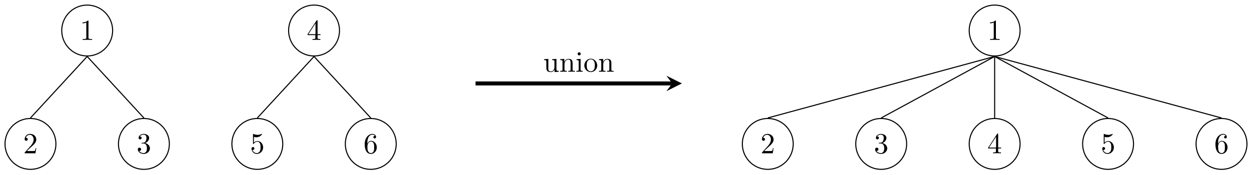
```
          i: 0 1 2 3 4 5 6 7 8 9
-----------------------------
A. id[i]: 0 1 2 1 1 8 6 7 8 9
B. id[i]: 4 4 1 0 8 0 0 4 6 4
C. id[i]: 9 9 3 0 0 2 8 6 8 9
D. id[i]: 5 5 5 9 5 9 8 2 9 9
```

3.3 Draw the resulting weighted quick union tree and write the resulting `id[]` after the calls to **connected**.

```
id[] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

connect(3,4)
connect(4,9)
connect(8,0)
connect(2,3)
connect(5,6)
connect(5,9)
connect(7,3)
connect(4,8)
connect(6,1)
```

4 Path Compression

The key to this next improvement on Weight Quick Union is to realize that once we call union on two objects, we will never un-union them. Thus there is no need to keep our trees bushy - we only need to keep track of each item's parent, not the path of unions to that parent. With **path compression**, we modify the tree by attaching each item directly as a child of its parent, creating a tree that only has 2 levels: the identifier for a set (parent) and all of the other items in that set (children).

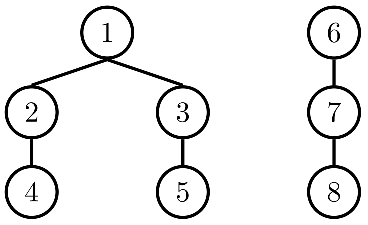


Path compression is implemented in `find` and is done by setting the node's parent to its root. This means that only calls to find and calls to functions that itself call find will utilize path compression. Below we define the find and union pseudo code for WQU with Path Compression. Notice that only `find` has changed. $\pi(x)$ signifies the parent of node x. It does not mean the root of the tree. Rank refers to the height of the tree as seen in Problem 3.3.

```
/* returns the root of x and performs path compression */
function find(x):
    if x != π(x): // if self is not parent
        π(x) = find(π(x)) // set parent to root of parent
    return π(x) // return parent

/* unions x and y and performs path compression on x and y through find */
function union(x, y):
    rx = find(x) // finds the root of x
    ry = find(y) // finds the root of y
    if rx == ry: // if already unioned, return
        return
    else if rank(rx) > rank(ry): // if rx higher rank, then set ry as child of rx
        π(ry) = rx
    else: // if ry higher rank, then set rx as child of ry
        π(rx) = ry
        if rank(rx) == rank(ry): // if equal rank, set as child of ry and increase rank of ry
            rank(ry) += 1
```

The runtime of performing a union-find operations on n items is $O(log^*(n))$, which for all practical purposes is constant (because log^* grows so slowly with n).

4.1 Given the disjoint set to the right, run the following `union` and `find` commands. Draw what the sets would look like after each command. Break ties by choosing the node with the lower number.



(a) `find(4)`

(b) `union(4, 8)`

(c) `find(7)`

(d) Suppose we started with all nodes disconnected (each in a separate set). Is our initial configuration as defined above possible by using WQU with Path Compression? If so, list in order the operations required. If not, explain why.

5 Path Compression Runtime

Note: The following problems regarding the amortized analysis of Path Compression are out of scope for CS 61B and would be considered a difficult CS 170 level problem. However the math is not very complex, and with enough effort, definitely possible for a 61B student.

The runtime for WQU with Path Compression can be analyzed with an amortization analogy or with a mathematical functioned called Inverse Ackerman function. Here we will be looking at the first method by proving smaller aspects of the amortization and building towards the complete proof.

5.1 **What Did the Drowning Computer Scientist Say?**
Answer: logloglogloglogloglog...

Iterated logarithm, denoted $\log^*$, is the number of times you need to apply $log(n)$ to some number n, $n \geq 0$ until the result is 1. It can be recursively defined as

$$\log^*(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^*(\log(n)) & \text{otherwise} \end{cases}$$

(a) Intuitively argue as to why $\log^*(n)$ is essentially constant for all practical values of n.

(b) Suppose you are given n balls, each numbered from 1 to n consecutively. You wish to separate these balls into bins such that bin contains the balls numbered from $k+1$ to 2^k, where k is a power of 2. For example, for $n = 2^{65535} = 2^{2^{16}}$,

$$\{1\}, \{2\}, \{3, 4\}, \{5, 6, 7, ..., 2^4\}, \{17, 18, ..., 2^{16}\}, \{2^{16}+1, 2^{16}+2, ..., 2^{2^{16}}\}$$

How many bins must you have to separate these n balls?
Hint: Apply $\log^*$ to the bounds of each bin.

(c) Prove that

$$\sum_{i=1}^{\alpha} \frac{n}{2^{k+i}} \leq \frac{n}{2^k} \qquad \forall \alpha \in \mathbb{N}$$

5.2 Universal Bin-sic Income

These following properties hold true for Weighted Quick Union with and without Path Compression. You do not need to memorize it for CS 61B, but may find it useful to try to prove them to better understand WQU.

1. Given n items, the maximum rank is $\log(n)$.
2. For any non root item x, $\text{rank}(x) < \text{rank}(\pi(x))$
3. Any root node of rank k has at least 2^k nodes in its tree.
4. If there are n elements overall, there can be at most $n = 2^k$ nodes of rank k.

(a) Using the solution from Problem 5.1c and the given properties, prove that there must be less than $\frac{n}{2^k}$ items of rank $> k$.

(b) Each item x_i has a rank r_i. Recall the bins scenario in Problem 5.1b. Let the range of all possible nonzero r_i, from 1 to $\log n$, be put into the bins. We assign some amount of money to each of the n items in this fashion: if an item x_i has rank r_i where r_i belongs to the bin

$$\{k+1, k+2, ..., 2^k\}$$

then we give that item 2^k dollars. Show that no more than $n \log^* n$ dollars is given out.

(c) There are two kinds of items x_i. Let a node be ”good” if the rank of x_i's parent $\pi(x_i)$ is in a higher bin than the rank of x_i is in. That means if x_i has rank $k+1$, then its parent has rank at least 2^k+1.

Now consider all the the nodes that are not ”good”. Explain why the maximum number of operations required until x_i's parent will be in a higher bin is $O(2^k)$.

(d) Does each node have enough money to turn itself into a ”good” node? What is an upper bound on the total amount of money that could be spend turning every node into a good node?

(e) If x_i and all of x_i's parents themselves are ”good”, then we say that x_i's path is partially compressed. Prove that if x_i is partially compressed, then it takes at most $O(\log^* n)$ operations to run `find(x)`. For example, if A's parent is B whose parent is C and A, B, C are all in different bins, then running `find(A)`will take no more than $O(\log^* n)$ operations.

5.3 Explanation of Results

The work you did in these previous problems was underlying math behind a complex analogy for amortization. By giving out $n \log^* n$ dollars, you can show that performing $n-1$ unions and some m finds, $m > n$, will run in $O(m \log^* n)$. This theorem is as stated:

Theorem. Starting from an empty data structure, weighted quick union with path compression performs any intermixed sequence of $m \geq n$ FIND and $n-1$ UNION operations in $O(mlog^*n)$ time.

Below, we will finish the proof of this theorem. A node x_i can be separated into 3 cases:

1. x_i is a root, and $x_i = \pi(x_i)$. Finding the parent thus is constant.
2. The rank of x_i's parent $\pi(x_i)$ is in a higher bin than that of x_i.
3. The rank of x_i's parent $\pi(x_i)$ is in the same bin as the rank of x_i.

The basic strategy here is to show that (1) it will take at most $n \log^* n$ work to get all nodes to be a Case 2 node. Then from there we can (2) show that path compression for a Case 2 node takes at most $\log^* n$ work.

When we path compress, we want to remove all intermediate parents of x_i and set x_i's immediate parent to the root. To turn all nodes from Case 3 to Case 2, we only need to particularly compress the path just enough so that every node on that path is in a different bin. What this means is that for every node x_i , we only want to compress x_i so that x_i's immediate parent's is in a higher rank bin. This involves changing the pointer of x_i's immediate parent at most 2^k times, where k is the rank of x_i. This is proven in Problem 5.2d and this operation costs 2^k dollars for x_i.

Each node is given a certain amount of money and is only responsible for using that money to get its parent into the next bin. That is, each node needs to have enough money to turn itself from a Case 3 into a Case 2. This is why we only have to worry about x_i getting its immediate parent p_i into the next bin. The node p_i can then use its money to get its immediate parent into the next higher bin. Thus because each node starts with 2^k dollars and it will cost at most 2^k dollars to move its parent up a bin, we can show that each node has enough money to become a Case 2. We know from Problem 5.2b that the total amount of money given out is $n \log^* n$, and we proved in problem 3d that this is enough money to partially compress every node. This (1) that it takes $O(n \log^* n)$ to make every node a Case 2.

Now assume that a node x_i has all of its intermediate parent in consecutively higher bins. Then to path compress this into just one parent, there need be at most $O(\log^* n)$ operations because there cannot be more than $\log^* n$ parents (the number of bins, one per parent). This proves (2) and the $O(\log^* n)$ operations upper bound is proven in Problem 5.2c.

Thus a FIND operation will take $\theta(1)$ time if the path is already compressed and $O(\log^* n)$ time if the path is particularly compressed. To get all nodes into that second state, it takes at most $n-1$ UNION operations at a cost of $O(n \log^* n)$, which is the total amount of money given out to every node. The total runtime for $m > n$ FIND operations provided $n-1$ UNION operations is $O(\text{cost of partial compression } + m * \text{ max cost of each find}) = O(n \log^* n + m * \log^* n) = O(m \log^* n)$. Amortized, this is $O(\log^* n)$ per operation which we showed in Problem 5.1a to be for all practical purposes constant.