Disjoint Sets Study Guide

Author: Josh Hug

Overview

Algorithm Development. Developing a good algorithm is an iterative process. We create a model of the problem, develop an algorithm, and revise the performance of the algorithm until it meets our needs. This lecture serves as an example of this process.

The Dynamic Connectivity Problem. The ultimate goal of this lecture was to develop a data type that support the following operations on a fixed number *N* of objects:

- `connect(int p, int q)` (called `union` in our optional textbook)
- `isConnected(int p, int q)` (called `connected` in our optional textbook)

We do not care about finding the actual path between `p` and `q`. We care only about their connectedness. A third operation we can support is very closely related to `connected()`:

- `find(int p)`: The `find()` method is defined so that `find(p) == find(q)` iff `connected(p, q)`. We did not use this in class, but it's in our textbook.

Key observation: Connectedness is an equivalence relation. Saying that two objects are connected is the same as saying they are in an equivalence class. This is just fancy math talk for saying "every object is in exactly one bucket, and we want to know if two objects are in the same bucket". When you connect two objects, you're basically just pouring everything from one bucket into another.

Quick find. This is the most natural solution, where each object is given an explicit number. Uses an array `id[]` of length N, where `id[i]` is the bucket number of object `i` (which is returned by `find(i)`). To connect two objects `p` and `q`, we set every object in `p`'s bucket to have `q`'s number.

- `connect`: May require many changes to `id`. Takes $\Theta(N)$ time, as algorithm must iterate over the entire array.
- `isConnected` (and `find`): take constant time.

Performing M operations takes $\Theta(MN)$ time in the worst case. If M is proportional to N, this results in a $\Theta(N^2)$ runtime.

Quick union. An alternate approach is to change the meaning of our `id` array. In this strategy, `id[i]` is the parent object of object `i`. An object can be its own parent. The `find()` method climbs the ladder of parents until it reaches the root (an object whose parent is itself). To connect `p` and `q`, we set the root of `p` to point to the root of `q`.

- `connect`: Requires only one change to `id[]`, but also requires root finding (worst case $\Theta(N)$ time).
- `isConnected` (and `find`): Requires root finding (worst case $\Theta(N)$ time).

Performing M operations takes $\Theta(NM)$ time in the worst case. Again, this results in quadratic behavior if M is proprtional to N.

Weighted quick union. Rather than `connect(p, q)` making the root of `p` point to the root of `q`, we instead make the root of the smaller tree point to the root of the larger one. The tree's *size* is the *number* of nodes, not the height of the tree. Results in tree heights of $\lg N$.

- `connect`: Requires only one change to `id`, but also requires root finding (worst case $\lg N$ time).
- `isConnected` (and `find`): Requires root finding (worst case $\lg N$ time).

Warning: if the two trees have the same size, the book code has the opposite convention as quick union and sets the root of the second tree to point to the root of the first tree. This isn't terribly important (you won't be tested on trivial details like these).

Weighted quick union with path compression. When `find` is called, every node along the way is made to point at the root. Results in nearly flat trees. Making M calls to union and find with N objects results in no more than $O(M\log^* N)$ array accesses, not counting the creation of the arrays. For any reasonable values of N in this universe that we inhabit, $log^*(N)$ is at most 5. It is possible to derive an even tighter bound, mentioned briefly in class (known as the Ackerman function).

Example Implementations

You are not responsible for knowing the details of these implementations for exams, but these may help in your understanding of the concepts.

QuickFind

QuickUnion

WeightedQuickUnion

Weighted Quick Union with Path Compression

Recommended Problems

To do the Coursera problems, you will need to register for an account. These problems will not be graded, and your progress will not be tracked by the 61B staff in any way. Signing up does not obligate you to do anything at all (i.e. Kevin Wayne will not come to your house and ask in a sad voice why you have not finished his class).

C level

1. Coursera problems, though problem 3 is a bit of overkill for our course (but isn't bad to know).

B level

1. Problem 1 from the Princeton Fall 2011 midterm.
2. Problem 1 from the Princeton Fall 2012 midterm.
3. For the `WeightedQuickUnionUF` data structure, the runtime of the `union` and `connected` operations is $O(\log N)$. Suppose we create a `WeightedQuickUnionUF` object with N items and then perform M_U and M_C union and connected operations, give the runtime in big O notation. Answer:
4. Same as #3, but give an example of a sequence of operations for which the runtime is $\Theta(N + M_U + M_C)$.
5. (From Textbook 1.5.8) Does the following implementation of Quick-Find work? If not, give a counter-example:

```
public void connect(int p, int q) {
    if (connected(p, q)) return;

    // Rename p’s component to q’s name.
    for (int i = 0; i &lt; id.length; i++) {
        if (id[i] == id[p]) id[i] = id[q];
    }
    count -= 1;
}
```

A level

1. (From Textbook 1.5.10): In weighted quick-union, suppose that we set `id[find(p)]` to q instead of `id[find(q)]`. Would the resulting algorithm be correct?
2. If we're concerned about tree height, why don't we use height for deciding tree size instead of weight? What is the worst-case tree height for weighted-quick-union vs. heighted-quick-union? What is the average tree height?
3. Try writing weighted-quick-union-with-path-compression without looking at the code on the booksite. You may look at the API. Compare your resulting code with the textbook's code.