Bob's Notes for COS 226 Fall 2013
1: Introduction, Union-Find, and Percolation

Robert E. Tarjan
9/15/13

Introduction

Welcome to COS 226! This is the first of a series of occasional notes that I plan to write about various subjects in the course. I wrote a set of similar notes that last time I helped teach the course, in Fall 2010, which you can find here:

www.cs.princeton.edu/courses/archive/fall10/cos226/precepts.php

I will try to mostly avoid duplicating what is in those notes, so you may well find it useful to read them in addition to this new series. These notes are intended for your edification and enjoyment. In addition to discussing some of the course content and assignments, I plan to suggest interesting ideas to pursue beyond what we have time to cover in the course.

Here are Bob's rules for succeeding in COS226 (and in your other courses):

1. **Follow the rules.** Know what you need to do in the course and do it: attend lectures and precepts, get access to a copy of the textbook, do the assigned readings and all the assignments and exercises. Know the collaboration policy and obey it.
2. **Focus.** Understand exactly what you need to do in each assignment, and do it. Once you have a complete, correct solution, and given extra time, please do experiment with extensions and alternatives. But your time is precious and you may well find yourself overcommitted. If so, do what is important.
3. **Start early.** This will give you more time to finish the assignment and to revise your solution to improve it. Think before you code. Figure out your solution method and your implementation plan. Then try it out.
4. **Finish early.** Then you will have extra time to resolve unexpected problems and to experiment with extensions, alternatives, and extra ideas.

The speed and storage capacity of computer systems is enormous and continues to grow rapidly. This means that the design space for algorithmic solutions to any problem is both deep and rich. Give yourself a little time to explore the space before you settle on a solution. Paraphrasing Einstein, "Make everything as simple as possible. But not simpler." Find the keys to a solution, and strip away everything else.

Union-Find

Here are a few thoughts about three aspects of the union-find (UF) data structure: the interface, path compression and its variants, and union methods.

There is an inconsistency in the UF interface as presented in the book, having to do with whether *find* is a public or private method. On pages 221 and 222, *find* is defined as public, but on pages 224 and 226 it is defined as private. A UF data structure with *find* private is seriously crippled, making it hard to use in situations where it is the natural solution.

Given that *find* is public, what *find*(*x*) should return is a *canonical element* in the set containing *x*. This element is defined as follows: initially, each singleton set {i} has canonical element i. When a union occurs, the implementation gets to choose one of the canonical elements of the two sets combined as the canonical element of the new set (assuming the union actually combines two different sets). This is not the way the interface is defined in the book, but it is the way all the implementations given in the book actually work. In the tree-based implementations, for example, the canonical element is the root of the tree. Note the comment on the top of page 220: "…you can think of each component as being represented by one of its sites." Yes! But let's build this idea into the interface, so we can use it. In what follows, I'll assume that *find* is a public method that returns a canonical element. (If one wants to be absolutely safe and stick with the interface as presented, one can write a little code that maps between component ids and canonical elements, but it's much better if the interface is designed correctly in the first place.)

Given that each set has a canonical element accessible by a *find*, one can store information about each set with its canonical element. This ability is key to many applications, and gives a nice solution to the "backwash" problem in the percolation assignment, as I discuss further below.

Let's turn to path compression and its variants. The idea is to change parents during a *find* so that later *finds* take less time (fewer pointer hops). The original way of doing this is path compression, invented by Alan Tritter in the 1960's: when doing *find*(*x*), after locating the root *y* of the tree containing *x*, make *y* the parent of all ancestors of *x*. Here is an iterative implementation of *find* with path compression (caveat: I have not run this or any other code in these notes; use at your own risk):

```
public int find(int x)
{   int y = x;
    int z = id[x];
    while (id[y] != y) y = id[y];
    while (z != y)
    {   id[x] = y;
        x = z;
        z = id[x];
    }
    return y;
}
```

One can also implement find with path compression recursively:

```
public int find(int x)
{   if (id[x] != x) id[x] = find(id[x]);
    return id[x];
}
```

The recursive implementation unnecessarily updates *id*[*x*] if *x* is a child of a root. This can be prevented at the cost of increasing the code length by replacing the second line of code by "{if (*id*[*id*[*x*]] != *id*[*x*]) *id*[*x*] = *find*(*id*[*x*])." An advantage of the iterative version is that it uses O(1) extra space; there is no risk of recursion stack overflow. Which version is best depends on the definition of "best." Try them out.

Any implementation of path compression requires two passes over the *find* path, one to find the root, the other to update parents. There are alternative one-pass methods that do not compress find paths completely but shorten them sufficiently to provide both practical and theoretical efficiency. A good one is *path splitting*, which makes every node on a *find* path point to its grandparent. Here is Josh's implementation of *find* with path splitting, from lecture 1 (with variable *i* changed to *x*):

```
public int find(int x)
{  while (id[x] != x)
   {  id[x] = id[id[x]];
      x = id[x];
   }
   return x;
}
```

This code also does an unnecessary assignment, which can be prevented by replacing the second line by "{while (*id*[*id*[*x*]] != *id*[*x*]." Whether this improves performance is a question for experiments.

How do these methods perform? It turns out that each of these methods when used with any union method has a total time of $O((M + N)(\lg N)/\lg(M/N + 2))$ for $M > 0$ set operations on sets containing a total of $N > 1$ elements. This bound is better than that of union by weight without path compression, since it decreases as the ratio M/N of operations to elements increases. With union by weight, they perform even better: the bound is $O((M + N)\alpha(N, M/N))$, where α is an incredibly slowly growing function that is a functional inverse of Ackermann's function. For non-negative integers k and j, Ackermann's function A is defined recursively as follows:

$A(0, j) = j + 1$, $A(k + 1, 0) = A(k, 1)$; $A(k + 1, j + 1) = A(k, A(k + 1, j))$.

A good way to think about this function is as a set of single-variable functions indexed by k. That is, the 0th function is $A(0, j) = j + 1$, the 1st function is $A(1, j) = j + 2$ (not growing so fast yet), the 2nd function is $A(2, j) > 2j$ (getting more interesting), the 3rd function is $A(3, j) > 2^j$, the fourth function is $A(4, j)$, which is bigger than an exponential stack of j 2's (its inverse function is log*, to within an additive constant), and so on.

The *inverse Ackermann function* α is defined for any non-negative integer n and non-negative real number r by

$\alpha(n, r) = \min\{k > 0 | A(k, \text{floor}(r)) > n\}$.

For all practical purposes α is a small positive constant. Indeed, the analysis that gives the α bound on path compression can be tightened to show that on any example small enough that it could ever be run in practice, the number of parents changed by path compression with weighted union is at most $N + 2M$.

The bottom line is that path compression or splitting is easier to implement than weighted union and is at least as efficient, and path compression or splitting with weighted union is even more efficient.

THE BOTTOM LINE: As a first heuristic to improve the performance of the compressed tree data structure for union-find, use path compression or path splitting, not a union method.

Speaking of union methods, there is a very simple one that is likely to do just as well in the percolation application as union by weight. This is *randomized union*, defined as follows: Assign the integers 0 through $N - 1$ uniformly at random to the set elements. When linking two roots, make the larger-numbered root the parent of the smaller. One can prove that the expected performance of this method is as good as weighted union, to within a constant factor. (This is actually a very recent result, to appear in a forthcoming conference, SODA 2014. Many of the data structures you will learn about, even though basic, are still subjects of on-going research. Computer science is (still) a young and active field!)

In the percolation application, one does not have to randomly number the elements; one can use *any* element numbering, including the straightforward one. This is because the application opens sites randomly, so they can all be viewed as equivalent, if one ignores boundary effects: sites near the top or bottom or left or right side behave differently from those near the middle of the array, but it is reasonable to suppose that these effects are insignificant. Thus we arrive at the following recommended union rule for the percolation problem, *union by index*: when doing a union, make the root of larger number (with respect to whatever numbering scheme you have chosen for the sites) the parent of the smaller. Union by index is not only simpler than weighted union (no need to store, test, and update set sizes), it can be used to help solve the "backwash" problem, as I discuss below.

Suggested Experiment: Write your own implementations of union by index and path compression or splitting. Test union by index with and without path compression or splitting, and naïve union with and without path compression or splitting. Compare the performance of these methods with the required methods.

Note: If you do this experiment, it is entirely on your own time and is NOT part of the programming assignment. But I'm really curious about the results, and I'll report on any that you may care to produce.

Percolation

The main issue I want to address is the backwash problem. But let me just point out that the computation done by programming assignment 1 does not in fact estimate the percolation constant; it estimates a different but related constant. For a brief discussion of this, see my first set of notes from 2010 at the url given above.

The suggested way to speed up the test of whether there is a path of open sites from the top row to the bottom row is to create a top open dummy site, connect open sites in the top row to this site, create a bottom dummy site, connect all the open sites in the bottom row to this dummy site, and test for a path by testing whether the top and bottom dummy sites are in the same connected component. This has the unfortunate side effect that it makes the visualizer produce incorrect outcomes. The visualizer marks sites as full of they are connected to an open site in the top row. Connecting all the open sites in the top row to a dummy site does not affect this, but connecting those in the bottom row to a dummy site does, as shown by the example in the checklist of assignment 1. How does one fix this? The most obvious solution is to maintain two UF data structures, one of which the visualizer uses, and the other of which is used for the percolation test. This solution is obviously redundant in that the sequence of union operations is almost the same in the two structures; the "test" structure just has some extra unions involving the dummy sites. If one can write one's own UF implementation, then it is possible to take advantage of this redundancy and use a single augmented UF structure. (Indeed, this is another topic of ongoing research.) But this violates the rules of the assignment, which require using the UF structure as a black box.

Rather than use two dummy sites, one can add two entire dummy rows, a top row and a bottom row. While one is at it, why not add two dummy side columns as well? Thus the array becomes $N + 2$ by $N + 2$, with rows and columns from 1 to N being the "real" array, and rows 0 and $N + 1$ and columns 0 and $N + 1$ being the "dummy" rows and columns. Note that this offsets the real rows and columns by 1 from what is in the problem specification. Now one can begin by opening all the sites in the top and bottom rows. The sites in the top two dummy columns remain closed throughout. (The four corner sites can be either open or closed; might as well leave them closed.) Percolation occurs when site (0,1) is connected to site ($N + 1$, 1). This construction adds $4N + 4$ dummy sites, which is many more than the 2 added by the suggested solution to the testing problem, but this number is still insignificant compared to the N^2 real sites, and the regularity of the construction has one big advantage: all the real sites now have exactly four neighbors, so that the code to test whether neighbors are open does not need to have special cases to deal with the boundaries. This is an example of a general technique, of making boundary cases match the general case by appropriately padding or otherwise modifying the data structure. Making the root of a tree its own parent, rather than, for example, making its parent null or undefined, is another example of the same idea. Yet another example is the use of "sentinels" in maintaining linked lists.

Adding dummy rows and columns does not solve the backwash problem if the sites in the bottom dummy row are open. But if we use union by index (see the previous section of these notes), with sites numbered in increasing order row by row, top to bottom, we can modify this solution as follows: leave all sites in the bottom dummy row closed. After each site opening, find the *id* of the component containing site (0, 1). The system percolates if and only if this *id* is in the bottom real row. This idea works if we omit the bottom dummy row and/or the dummy side columns. It also works if we add only one dummy site, numbered smallest, connected to all the top open sites. Of course omitting the bottom dummy row and the dummy side columns makes site opening more complicated, as discussed above. Also, this solution is outside the rules of the assignment, since it requires modifying the UF implementation.

A third solution that assumes *find* is a public method is to store information about the state of each connected component with the component id. My notes from 2010 describe this solution; I'll repeat it here. Two bits of information suffice: a bit indicating whether the component contains an open site in the top row, and a bit indicating whether the component contains an open site in the bottom row. This solution avoids the need for any dummy sites, or for a second UF structure, or for a special implementation of UF.

Have any of you found another solution?