COS 226 Bob's Notes 1: Union-Find and Percolation (Version 2)
Fall 2010
R. Tarjan, Preceptor

This is the first of a set of notes on ideas that we talked about in precept, important (and maybe not-so-important) points we didn't have time to cover, and my observations. I'll write these notes as needed and as my time and energy permits. Unless otherwise indicated, I ignore constant factors and lower-order terms. The material in these notes is entirely optional and is intended to supplement the course. It presupposes that you have read the relevant sections of the text (Algorithms, Fourth Edition, Preliminary Edition, Fall 2010, R. Sedgewick and K. Wayne) and attended lectures and precepts. All the exercises, problems, and questions are food for your thoughts. If any of them excite you and you choose to work on them, I will be happy to give you feedback and answer questions. But remember that your first and main priority is the programming assignments.

Welcome to COS 226! I'm excited to have you in my precepts. Together we can find the elegance, the enjoyment in algorithms. Remember, you will get out of this course no less, and I hope much more, than you put into it. Given your other class and personal commitments, you can expect to be under time pressure in this course, sometimes severe. So: get started early on all the assignments. Read the book, carefully. Come to lectures and precepts. Ask questions. Do not be shy! Seek help when you are stuck or confused. All questions are intelligent. (But, of course, you will learn the most by trying something yourself first, possibly several times, so you understand a bit about what you don't understand. Then you will ask more intelligent questions.) ("All animals are equal but some animals are more equal than others." – George Orwell, Animal Farm, 1945, Chapter 10)

On to the technical stuff: The union-find problem, or what I prefer to call the "disjoint set union" problem. As in all problems in algorithms, this one has a rich design space, and by exploring the design space we may discover a method or methods that are just as good as, or even better than, the "textbook" method(s). The bottom line: YES, there is a better method. (Skip down if you can't wait to see it.)

I'll start by looking at two questions about union-find algorithms. The first is how to speed up the quick-find method. The drawback of this method is that the only way to find the items in a set, which must be done to change their id's when a union occurs, is to run through the entire id array, which takes N accesses. We would like to find these items in constant time per item. We can do this if we make each set into a linked list. This adds space but saves time. In discussing this in P01 I stumbled a bit, and I did not discuss it at all in P02. Here I'll explain briefly how this idea works.

We make each set into a circular list by adding an extra array next: next[i] is the item after i in its list. If i is a singleton, next[i] = i. (By the way, I favor parentheses instead of brackets to denote arrays, but I'll try to stick with the Java convention.) To do union(p, q), if id[p] != id[q], we run through the list containing p, changing all id's to id[q]. Then we combine the lists for p and q by changing two pointers. (How?)

Exercise: Implement quick-find with circular lists.

Use of circular lists reduces the time for a union from N to the size of one of the two sets put together. We can combine this idea with weighted union, by maintaining the size of each set (in yet another array, indexed by set id), and in each union relabeling the items in the smaller set. We call this "quick-find with weighted union."

Exercise: Implement quick-find with weighted union.

In the worst case, quick-find with circular lists still takes N time per union, but quick-find with weighted union has efficiency similar to that of quick-union with weighted union (but no path compression): the time for N – 1 or fewer unions (if there are N items there can be at most N – 1 unions that combine two different sets) is at most NlgN. (I use lgn to denote the base-two logarithm. This notation was proposed by Knuth. In computer science, the base-two logarithm is at least as important as the natural (base-e) logarithm, and much more important than the base-10 logarithm – though all are within constant factors of each other.)

Exercise: Prove that quick-find with weighted union takes O(NlgN) total time for unions and O(1) time per find.

This makes quick-find with weighted union superficially attractive. But if we dig deeper, we find inefficiency. Specifically, quick-find with circular lists (with or without weighted union) is like path compression, except that it changes the ids of ALL items in a set (when it does a union), rather than those some of the items in the set (when it does a find). Thus it does a form of "eager" path compression. The problem is that this method is TOO eager; it is better to change id's in a lazier manner, such as in path compression. Indeed, one can show that, on EVERY sequence of intermixed union and find operations, the method of quick-union with weighted union and path compression is at least as fast as the method of quick-find with weighted union, and sometimes much faster. That is, the former method dominates the latter.

Exercise: Prove this.

THE BOTTOM LINE: Quick-find is NEVER the algorithm of choice, even with circular lists, and even when weighted union. Some version of quick-union with path compression is better.

Next I want to examine alternative quick union methods that use a union rule other than union by weight, with the goal of simplifying the method, or improving its efficiency, or reducing its space, or some combination of these. We explored this a bit in P01. Two intriguing ideas were suggested. The overall goal is to devise a method that keeps the time for a find small. As an estimate of the time to find an item, we use its depth: the number of links from it to the root of its current tree. Thus our goal is to keep all depths, or at least the maximum depth, small.

The first suggestion, made by Amy Zhou, is a local method that avoids storing any extra information, such as tree size, with each tree root. The idea is as follows: to do find(p, q), follow links (parent pointers, encoded as integers) from p until reaching a root p'; do the same for q, finding root q'. While following these two paths of links, count the links, giving the depth of p and of q (each in its current tree). If p is shallower, make p' point to q'; if q is shallower, make q' point to p'. The intuition here is that if, say, p is shallower, then after the union it will still be no deeper than q, so the maximum of the depths of p and q cannot grow; if instead we make p' point to q', then the maximum of the depths of p and q, namely the depth of q, grows by one.

This idea turns out to lead to a really good method, but before developing it let's look at a downside of this particular method, a downside shared by ANY method that does not store some kind of extra information about each tree, information that is accessible from its root. We are concerned about the worst case here. Suppose each union is of two items p and q that are roots. Then we have no basis on which to decide whether to link p to q or q to p. In particular, we cannot avoid the BAD case, in which we link the root of a big tree to a root which is a singleton. If we do enough such bad unions, we get a long path; with no path compression, worst-case finds take time, much worse than logarithmic or close-to-constant.

An alternative method, suggested by Santhosh Balasubramanian, is to keep track of the "width' of the tree using some measure of bushiness. Then we can store the width of each tree in its root, and base our decision about how to add the link in a union on the widths of the two trees. A simple measure of width is the number of children of the root (the number of nodes that point directly to the root). This idea raises two questions: can we maintain the width efficiently? (Answer, please.) Even if we add path compression? If yes, then should we make the root of the wider tree point to that of the narrower tree, or the other way around? The first alternative suffers from the singleton problem: If we link two trees, one of which contains only one node, then the singleton becomes the root of the new tree (now with width one); repeating this builds a long path and makes for slow finds. The better alternative is to make the root of the tree with smaller width (the root with fewer children) point to that of the tree with larger width.

Problem: Analyze the running time of finds if quick union uses union by width, (1) without path compression, and (2) with path compression.

Exercise: Implement quick union with union by width.

Now let us return to the idea of keeping the tree depth small, instead of the tree size or width. Because of the "linking with a singleton" problem, we want to keep the maximum depth small, not just the depth of individual nodes. We can do this by storing the maximum depth (instead of the tree size) with the tree root, in an array that we'll call "rank": rank([i] is the maximum depth of the tree whose root is i. (We don't need to maintain rank[i] once i becomes a non-root; we can just ignore this array position, or use it for something else. This is also true for sizes in the weighted union method.) Initially all nodes are singletons, so rank[i] = 0 for all i. To do union(p, q), find the roots p' and q' of the trees containing p and q, respectively. If rank[p'] < rank[q'], make p' point to q'. If rank[q'] < rank[p'], make q' point to p'. In both of these cases no rank update is necessary. The interesting case is if rank[p'] = rank[q']. In this case make p' point to q', and increase rank[q'] by one. We call this method "quick-union with union by rank."

Exercise: Show that this method correctly maintains the maximum depth of each tree.

Exercise: Implement quick-union.

Exercise: Show that with union by rank, the maximum depth of a tree containing n nodes is at most lgn, and thus that the time per find is O(lgn), just as with weighted union.

Hint: Turn the proof of Proposition E on page 135 of the text "inside out."

Can we use path compression with union by rank? We can, but we must be careful: a path compression may reduce the maximum depth of a tree (we hope it does!) but we have no way to detect this. Thus when we do a path compression, we do NOT change any root ranks. Now the rank of the root of a tree is no longer the maximum depth of the tree, but it IS an upper bound on the maximum depth. (Prove this.) (This is the reason it is "rank" instead of "depth".) Quick-union with union by rank and path compression has the same almost-linear performance as quick-union with weighted union and path compression. With or without path compression, union by rank is better than weighted union: ranks change less often than sizes (a rank only changes in the equal-rank case of a union) and ranks are (exponentially!) smaller than sizes, so they need less space: lglgn bits rather than lgn bits. (Whether we can take advantage of this potential space savings is another question.)

THE BOTTOM LINE: Quick-find with union by rank and path compression is the algorithm that belongs in the "book." (The "book" is the book of best algorithms, written in the sky.)

The design of union-find algorithms, and their analysis, is a rich playground, and we could spend all semester on it. But this would leave us no time for other topics, even related ones such as percolation. For more on union-find, in particular the analysis of path compression, take COS 423!

I'll conclude with several comments about percolation. In the application of union-find to the problem of computing the percolation probability, the instances of union-find that result are not completely general: the set of edges determining the unions forms a graph that is a subset of a square grid. One might be able to design an implementation of union-find that is fast for this special case, even though it might not be fast on completely general instances. This raises the question of whether there is a linear-time union-find algorithm for the instances that arise in the percolation problem, either in the worst case or on the average. (Here the application defines the average-case model for us.) I suspect that the answer is yes to both versions of the question.

More significantly, one does not in fact need union-find to do the percolation computation, as Kevin pointed out to me. All that one needs to be able to do is to keep track of the set of full cells, those that are open and have a path to an open cell in the top row (or equivalently to the dummy top row). We can use incremental graph search to maintain this information. Each cell is in one of three states: blocked, open but not full, or full. We use no dummy rows. When a new site, say x, is opened, we apply the following recursive procedure, search, to x:

search(x): If x is in the top row or any neighbor if x is full, make x full. If x is now full and in the bottom row, stop: the system percolates. Otherwise, if x is now full, for each open neighbor y of x do search(y).

This method amounts to an incremental depth-first search. Why does it work? What is its running time?

As Kevin also pointed out to me and as I mentioned in precept, a solution to assignment 1 does not in fact compute the percolation probability in the original probability model, but in a slightly different model. The difference is subtle but important. In the original model, each site is open with probability p, and one wants to know the probability of percolation. What the assignment actually calculates is an estimate, for each k between 0 and the number of sites (say N^2), the probability of percolation given that k sites selected at random are open. The difference can be seen by considering k = N – 1. In the model where the number of open sites is k but they are selected randomly, the percolation probability is zero, since each path from the top to the bottom contains at least N sites. But if each site is open with probability k/N^2, the percolation probability is small but non-zero. This topic deserves much more discussion, but it is one for a course on probability (or average-case analysis).

Nader Al-Naji asked the following questions in an email message: Is there a way to open cells at random that avoids trying to open cells that are already open? This topic was also raised by Kevin in lecture 4. One solution is to use a randomized queue: see programming assignment 2. In this case one can simplify the space requirements of the data structure: it suffices that the maximum size of the data structure is proportional to N^2, so for example doubling and halving as in the array implementation of a stack (see lecture 4) is unnecessary.

But before adding such a data structure to the percolation implementation, it is worth asking how many random number generations would this save. That is, how much speed-up could we get from such a method? Let's look at this question. For simplicity, let me denote the total number of sites by n, and let the sites be numbered from 0 to n – 1. (In the percolation application, n = N^2.) Suppose k sites are already open. How many random numbers between 0 and n – 1, chosen with repetition, do we expect to generate before opening a new site? The probability of opening a new site is p = (n – k)/n, since of the n sites, n – k are currently blocked. The probability that exactly one try (one random number generation) is needed to open one more site is p. The probability that exactly two tries are needed is (p – 1)p. (Why?) The probability that exactly j tries are needed is (p – 1)^jp. (Why?) Thus the expected number of tries is p(1 + 2(1 – p) + 3(1 – p)^2 + 4(1 – p)^3 +… (Why?) = 1/p. (Exercise: verify this.)

Suppose we are not interested in percolation but instead we just want to open all the sites. How many tries in total would this take (on the average). One beauty of expectations (averages) is that they add. Thus the expected total number of tries to open all sites is the expected number to open the first site plus the expected number to open the second site plus the expected number to open the third site + … = 1 + n/(n – 1) + n/(n – 2) + … + n/2 + n = n(1 + ½ + 1/3 +… + 1/n) ~ nlnn, where lnn is the natural (base e) logarithm. (Exercise: verify this.) (Note: here is a use in algorithm analysis, probably the most important one, of the natural logarithm. We shall see this sum again in the analysis of quicksort.) This is NOT linear: if we wanted to open all the sites we would save a log factor by using the method suggested above. We have just solved the so-called "coupon collector's problem:" given n coupons, of which one comes in each cereal box, and each cereal box contains one coupon among the n, selected uniformly at random, how many boxes of cereal can you expect to buy to collect ALL the coupons? Collecting the last few coupons takes many tries. To get a feeling for this, check out http://www-stat.stanford.edu/~susan/surprise/Collector.html.

Fortunately we don't have to open all the sites, just enough so that the system percolates. We know that this is only a constant fraction of the sites, slightly less than 60%. The expected total number of tries to open this fraction of the sites is less than ~ nlgn – nln(.4n) = nln(2.5), which is about .91n (verify this computation), which means that we save about half a try per open site. We conclude that in this application, trying to avoid extra random number generations probably costs more than it saves.

One last technical comment about percolation, in response to a question asked by Clayton Raithel: if one uses the suggested trick of adding dummy open rows at the top and bottom to make the test for percolation more efficient, one encounters the "backwash" problem mentioned in the checklist for the assignment: when percolation occurs, all components that contain an open cell in the bottom row are declared "full" (and will be visualized as such), because they are connected through the dummy bottom row to an actual full cell in the bottom row. This is a symptom of a possibly more serious problem: the implementation does not actually keep track of all the connected components, since components with cells in the top row are combined, as are components with cells in the bottom row. This means that one cannot keep track of additional statistic, such as the number of components, or the size of the largest component. Here is what I think is a better and more disciplined way to keep track of whether the system percolates. Store two bits with each component root. One of these bits is true if the component contains a cell in the top row. The other bit is true if the component contains a cell in the bottom row. Updating the bits after a union takes constant time. Percolation occurs when the component containing a newly opened cell has both bits true, after the unions that result from the cell becoming open.

Unfortunately one must extend the union-find API to support such bits (or any other information associated with a set). For this and other reasons, I prefer the following alternative API: Each set has a unique canonical element, which the implementation gets to choose. The find method has only one argument, p, and it returns the canonical element of the set containing p. The union method has two arguments, both canonical elements. It combines the sets containing these canonical elements and chooses a canonical element for the new set. One also needs a method to initialize the data structure. With such an API, one can associate arbitrary information about the sets with their canonical elements.

Exercise: Redo the percolation assignment with bits to test for percolation, using the suggested alternate API for union-find.

Percolation has all kinds of amazing applications, such as the spread of disease, the spread of gossip on the web, and even the spread of intelligent life in the universe. For the last application, see http://www.geoffreylandis.com/percolation.htp.