Design and Analysis of Algorithms May 2, 2012
Massachusetts Institute of Technology 6.046J/18.410J
Profs. Dana Moshkovitz and Bruce Tidor Handout 21

Problem Set 8 Solutions

This problem set is due **at 9:00pm** on **Wednesday, May 2, 2012**.

Problem 8-1. Improving Union-by-Rank

Recall the Forest of Trees solution to the Union-Find data structure presented in class. Consider the version with the Union-by-Rank but not the Path Compression improvement. Let n be the total number of elements, and let m be the total number of operations. We gave run time bounds for the following operations:

- MAKE-SET(u): $\Theta(1)$
- WEAK-UNION(u, v): $\Theta(1)$ (where u, v are the roots of two trees)
- FIND-SET(u): $\Theta(\lg n)$
- UNION(u, v) = WEAK-UNION(FIND-SET(u), FIND-SET(v)): $\Theta(1) + 2\Theta(\lg n) = \Theta(\lg n)$

Thus, using this data structure, a series of m operations will take $O(m \lg n)$ time.

Notice that currently, WEAK-UNION is fast ($\Theta(1)$), while FIND-SET is slow ($\Theta(\lg n)$). If we can somehow make FIND-SET slightly faster, at the expense of making WEAK-UNION slightly slower, then the run time of UNION will be slightly faster, so we can do m operations in $o(m \lg n)$ time. This improvement can be achieved without using Path Compression at all. However, regular Union-by-Rank with Path Compression is still the best.

(a) We will develop a data structure over n elements that supports MAKE-SET in $\Theta(1)$, WEAK-UNION in $O(k)$, and FIND-SET in time $O(\log_k n)$, for any integer k.

To lead you off, you will still need a forest of trees; however, the trees in this forest should satisfy certain special properties:

1. Unlike trees in regular Union-by-Rank, the trees in our forest only store elements on its leaves. The root and all the internal nodes (nodes that are neither the root nor a leaf) don't represent elements in the data structure. The root stores the name of the representative element.
2. Let $height(x)$ of a node x be the length of the longest path from x to any leaf below x (ex. height of a leaf is 0). We require that all roots in the forest have height at least 1.
3. We also require that if a root has height $h > 1$, then it must have at least two children. Further, we will ensure that each child of the root must have height $h - 1$.

4. Finally, each internal node with height h must have at least k children whose heights are exactly $h-1$. It may have any number of children with height less than $h-1$. Consequently, each internal node will have at least k children, but possibly more.

Develop a data structure containing special trees satisfying the above constraints that supports the aforementioned operations, perhaps augmented with additional data on the nodes. Analyze how each of the operations is performed to ensure the properties of the trees are always satisfied, and show that they have the desired run time.

Solution: We will augment the data structure to store two additional pieces of information on the root of each tree (in addition to the name of the representative), which are its height h and how many children it has (note: not descendants). We also need child to parent pointers on all nodes.

To implement MAKE-SET(x), we just make a new root, and attach x to that root. Note this satisfies constraint 2. and takes O(1) time.

To implement FIND-SET(x), we simply traverse up the tree. It takes $\log_k n$ time because each internal node as at least k children (from property 4).

To implement WEAK-UNION(a, b), we look at the heights and number of childrens of the two roots. Without loss of generality, say a has no more height than b, and if their heights are equal, a has no more children than b.

There are three cases:

1. The height of a is strictly less than b. Let c be an arbitrary child of b. If a has less than k children, we can afford to redirect all of their pointers to c instead, and discard a (which is fine, because all data are on the leaf nodes only). Note that by property 3, all of a's children will have height 1 less than $h(a)$, which is less than or equal to $h(v)$. So doing this merge does not violate any of the constraints above.
   Otherwise, a must have at least k children. If $h(a) = h(b) - 1$, we can make a a child of b. This is okay because a already has at least k children of rank $h(a) - 1$, so this satisfies property 4 of any internal node. If $h(a) < h(b) - 1$, we can make a a child of c, while still satisfying all properties.
2. The heights of a and b are equal, and the number of children of a is smaller than k. We can afford to redirect all children of a to point to b, and increment b's number of children field.
3. The heights of a and b are equal, and the number of children of a is greater than k. In this case, we can create a new root c that points to both a and b, and store in c the name of the new set, the number of children, and pointers to its children. The height of c will be $h(a) + 1$. This ensures that c satisfies the requirements of property 4.

Notice that all the operations in WEAK-UNION can be done in $O(1)$ or $O(k)$. So we are good.

(b) Explain how you can use the data structure in (a) to achieve $o(m \lg n)$ run time on m total operations.

Solution: To minimize the running time of m operations, we want to minimize the maximum of $O(k)$ and $O(\log_k n)$. To do so, we set them equal to each other and solve:

$$\begin{aligned} k = \log_k n &= \frac{\lg n}{\lg k} \\ k \lg k &= \lg n \\ k &= \Theta\left(\frac{\lg n}{\lg \lg n}\right) \end{aligned}$$

This means that the time for any operation is at most:

$$\begin{aligned} O(k) + O(\log_k n) &= O\left(\frac{\lg n}{\lg \lg n} + \frac{\lg n}{\lg k}\right) \\ &= O\left(\frac{\lg n}{\lg \lg n} + \frac{\lg n}{\lg \frac{\lg n}{\lg \lg n}}\right) \\ &= O\left(\frac{\lg n}{\lg \lg n} + \frac{\lg n}{\lg \lg n - \lg \lg \lg n}\right) \\ &= O\left(\frac{\lg n}{\lg \lg n}\right) \end{aligned}$$

So the total time for m operations is $O\left(\frac{m \lg n}{\lg \lg n}\right) = o(m \lg n)$.

Problem 8-2. Polynomial or NP-Hard?

Suppose that there are n_1 classes you are thinking of taking. You have a list of which of your n_2 friends are taking which classes. To figure out your schedule for the semester, you've decided to learn as much as you can about all of the classes. You can learn something about a class in two ways: (1) attend the class yourself, or (2) ask a friend who is taking the class what it's like. Attending a class takes one hour. Speaking to one friend also takes one hour, but during that hour you can quiz your friend about *all* of the classes they are taking.

(a) For each available class, you would like to either attend the class in question, or speak to *all* of your friends in the class to get their personal opinion. You only have k hours to devote to this process. Either devise a polynomial-time algorithm for picking out k friends and classes that satisfy these constraints, or show that the problem is NP-hard.

Solution: First, note that we can make several simplifying assumptions about this problem. If there are any classes that are not attended by any friends, then the only

way to learn about the class is to attend it. If there are any friends who are not attending any classes, then there is no point in talking to them. As a result, we can assume that each friend attends at least one class and every class is attended by at least one friend.

With these restrictions, there exists a polynomial-time algorithm for this problem that works by reduction to the maximum cut problem. We construct a graph as follows:

1. Create a source vertex s and a sink vertex t.
2. For each friend f, construct a vertex x_f, and add an edge from the source s to x_f with capacity 1.
3. For each class c, construct a vertex y_c, and add an edge from y_c to the sink t with capacity 1.
4. For each friend f, let C_f be the set of classes attended by f. For each $c \in C_f$, construct an edge from x_f to y_c with capacity ∞.

We then run Ford-Fulkerson on the graph, and then compute the resulting minimum cut. For each edge (s, x_f) in the cut, we talk to friend f. For each edge (y_c, t) in the cut, we attend class c. If this schedule requires more than k hours, there is no way to learn about all classes in k hours. If this schedule requires at most k hours, then the schedule will be feasible. The runtime required for this algorithm is at most $(n_1+n_2)^3$, which is polynomial in the size of the input.

We must analyze the correctness of this algorithm. We may do so by showing that every cut of the graph is a schedule of the same size, and every schedule is a cut of the graph of the same size. This, minimizing the capacity of the cut is the same as minimizing the number of hours required for the schedule.

Suppose that we have some cut $(S, V - S)$ of the graph whose capacity is finite. Then the only edges crossing from S to $V - S$ must be edges of capacity 1. Each such edge corresponds to either attending a class or seeing a friend. We must determine two things about this schedule: the number of hours required, and whether the schedule lets us learn about all classes. The number of hours required is the same as the capacity of the cut.

Suppose, for the sake of contradiction, that the schedule doesn't let us learn about some particular class c. Then we cannot be attending c, and there must be at least one friend f who attends c who is not on the schedule. Consider the path (s, x_f, y_c, t). At least one of the three edges in the path must cross from S to T. It cannot be the edge from x_f to y_c, because that edge has infinite capacity, and we assumed a cut of finite capacity. So it must be (s, x_f) or (y_c, t). But we've already established that we are neither talking to f, nor attending c. Hence, this is a contradiction. Our schedule lets us learn about all classes.

Now suppose that we have a schedule that lets us learn about all classes. We wish to show that this corresponds to a cut with capacity equal to the amount of time required for the schedule. For each friend f we talk to, remove (s, x_f) from the graph. For each class c we attend, remove (y_c, t) from the graph. Let S be the set of vertices reachable from s in this modified graph.

First, we wish to show that S is a cut: namely, that $t \notin S$. Suppose for the sake of contradiction that $t \in S$. Then there exists some path (s, x_f, y_c, t) from s to t in the modified graph. Consider the class c. Because (y_c, t) exists in the modified graph, we were not taking c in the original schedule. So it must be that we talked to all friends who were attending c, including f. But that would mean that (s, x_f) was removed from the graph, which contradicts the existence of the path (s, x_f, y_c, t). So S must be a cut that separates s from t.

Now we wish to calculate the capacity of the cut. Let (u, v) be an edge crossing the cut — in other words, $u \in S$ and $v \notin S$. The edge (u, v) was an edge in the original graph, so unless it was removed during the cut construction, it must also exist in the modified graph. But if it exists in the modified graph, then there must be a path from s to v: take the path from s to u, and add the edge (u, v) to the end of it. This is a contradiction. So (u, v) must have been one of the edges removed in the process of the cut construction. We remove a number of edges equal to the number of hours required by the schedule.

Hence, we have shown an equivalence between schedules that allow us to learn about all classes and cuts of the graph constructed using our reduction. This equivalence even holds for the size: the number of hours used by the schedule is equivalent to the capacity of the cut. So by finding the minimum cut, we are also finding the schedule with the fewest number of hours.

(b) To make your life easier, you've decided that you can learn enough about a class by either attending the class, or talking to *at least one* of your friends in the class. Again, you only have k hours to devote to this process. Either devise a polynomial-time algorithm for picking out k friends and classes that satisfy these constraints, or show that the problem is NP-hard.

Solution: We show that this problem is NP-hard by a reduction from VERTEX-COVER. Our reduction must take as input an undirected unweighted graph $G = (V, E)$ and a desired cover size ℓ. It should then do the following:

1. For each vertex $v \in V$, construct a friend f_v and add it to the set of friends F.
2. For each edge $e \in E$, construct a class c_e and add it to the set of classes C.
3. For each edge $e = (u, v) \in E$, make the friends f_u and f_v attend the class c_e.
4. Set the number of available hours k equal to ℓ.

First, we must examine the runtime of this reduction. Each of the first three steps involves a for-loop, and each iteration of the loop requires constant time. The total number of iterations is at most $|V| + |E|$. Therefore, the total runtime of the reduction is polynomial in the input size.

Next, we must show the correctness of the reduction. To do so, we must show two things: that any solution to the VERTEX-COVER problem would yield a solution to the classes problem, and vice versa.

Suppose that we have a solution to the VERTEX-COVER problem. Let S be the set of vertices in the solution. Then we can solve the corresponding classes problem as follows: for each vertex $v \in S$, talk to the friend f_v. Because $|S| = \ell = k$, this solution takes the correct number of hours. Does this also let us learn about all of our classes? Consider some edge $e = (u, v) \in E$, and the corresponding class c_e. Because S is a vertex cover, at least one of u or v must be in S. Therefore, we'll end up talking to at least one of f_u and f_v, and will therefore learn about c_e.

Now suppose that we have a solution to a classes problem constructed with this reduction. Let A be the set of friends visited, and let B be the set of classes attended. Now define the set S as follows:

$$S = \{v \mid f_v \in A\} \cup \{u \mid c_{(u,v)} \in B\}$$

This means that $|S| \leq |A| + |B| = k = \ell$. (If $|S| < \ell$, add arbitrary nodes until $|S| = \ell$.) Now we wish to show that S is a vertex cover. Consider some edge $e = (u, v) \in E$. In order to learn about class c_e, we must have $f_u \in A$, $f_v \in A$, or $c_e \in B$. Therefore, the constructed set S must either contain u or v. As a result, S is a vertex cover.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012