6.046
PROBLEM SET 8

JOHN WANG

Collaborators:

1. Problem 8-1

1.1. Problem A.

Problem 1.1. *Develop a data structure containing special trees satisfying the above constraints that supports the aforementioned operations. 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 develop a data structure which satisfies all of the constraints by creating a union operation. First, let each of the internal nodes store the number of children it contains and its height. Each node also contains parent and child pointers, and the root contains a pointer to the representative element. The union operation will make sure to update both of these criteria, and the Make-Set operation will create a node with height of 0 and number of children of 0.

Now, we can break the union of two trees into different cases. Let T_1 be the tree with greater height and T_2 be the tree with smaller height (choose arbitrarily if the heights are equal). Let h_1 and c_1 be the heights and number of children of the root for T_1 and h_2 and c_2 be the corresponding numbers for T_2. We assume that if two trees have been passed to the union function, then they hold the invariants of the constraints named in the problem.

Now, if $h_2 < h_1 - 1$, then we know from the invariants that each internal node with height h has at least k children whose heights are exactly $h-1$. Thus, we can append the root of T_2 to a child of the root of T_1, and we know this will not increase the height of the tree. Moreover, the child of the root with a new node still satisfies its invariant because it contains more than k children with heights exactly $h-1$ by the original assumption.

If $h_2 = h_1 - 1$, then we can append T_2 to the root of T_1. We can do this without destroying the invariants because T_2 is now a child of the root of T_1, and it has height of $h_1 - 1$. This means each child of the root of T_1 still has height $h_1 - 1$.

If $h_2 = h_1$, then we have to look at the number of children in T_1 and T_2. If either $c_1 < k$ or $c_2 < k$, then put all of the children of the root of the tree with $c_i < k$ onto the root of the other tree. Discard the root of tree T_i (since it contains no more relevant information). The new tree will still satisfy all the constraints because the children in all of the roots have height $h_1 - 1$, and each internal node withheight h still has at least k children with height $h-1$.

Now, if both $c_1 \geq k$ and $c_2 \geq k$, then create a new root node, and set T_1 and T_2 as children of this new root node. This keeps the invariants because the root has at least two children, each with height $h-1$.

Finally, we update each of the number of children and height of the nodes which have been affected. We update the height of each node which has a new child by setting $h_{new} = \max\{h_{old}, h_j + 1\}$ for all j which are new children. We update the number of children of each node which has a new child by setting $c_{new} = c_{old} + \sum_j c_j$ for all j which are new children.

Since our analysis has shown that the union operation doesn't break the invariant, we know that the constraints will always be satisfied. This is because they are satisfied for each Make-Set operation and Union operation, and Find-Set does not mutate anything within the data structure.

Now we shall analyze the runtime. Clearly, Make-Set is $O(1)$ because we simply initialize a node and its information. Moreover, we know that Weak-Union is $O(k)$ because in any one of the cases enumerate above, we append at most k subtrees to another node. Finally, we know that $Find-Set$ runs in $O(\log_k(n))$ because all internal nodes have at least k children with heights exactly $h-1$. This means each level of the tree, except possibly the first level, will have at least k children for each node. However, since the first level is a constant and we know that there are n total objects in the tree, we must have a tree height of $h = O(\log_k n)$. This means that Find-Set, which will climb up parent pointers until it reaches the root, will take $O(\log_k n)$ running time. □

1.2. **Problem B.**

Problem 1.2. *Explain how you can use the data structure in (a) to achieve $o(m \lg n)$ run time on m total operations.*

Solution First, note that we can perform union in $O(k)+2O(\log_k n)$ time because we can use two find sets and a weak union to create a true union function. Now, we know that an adversary will attempt to choose the worst sequence of operations for our datastructure. Therefore, the adversary will not pick Make-Set operations, but will instead choose Find-Set and Union operations.

Now, let us set $k = \log\log n$ as the parameter for the trees will will be creating. If we do this, Make-Set remains $O(1)$, but Find-Set becomes $O(\log_{\log\log n} n)$ and Union becomes $O(\log\log n) + 2O(\log_{\log\log n} n) = O(\log_{\log\log n} n)$. Thus, both Find-Set and Union have the same asmyptotic running time. Moreover, the adversary will always choose these operations because they have a higher run time than Make-Set. Now consider m of these operations. They will take running time:

$$O(m)O(\log_{\log\log n} n) = O(m)O\left(\frac{\log n}{\log\log n}\right) \tag{1.1}$$

$$= O\left(\frac{m\log n}{\log\log n}\right) \tag{1.2}$$

Moreover, taking limits for little oh notation, we find:

$$\lim_{n\to\infty} \frac{\frac{m\log n}{\log\log n}}{m\log n} = \frac{1}{\log\log n} = 0 \tag{1.3}$$

This shows that the running time of m operations on this data structure is $o(m\log n)$. □

2. Problem 8-2

2.1. **Problem A.**

Problem 2.1. *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 We shall devise a polynomial time algorithm to solve this problem. First, we note that there is a convenient graphical representation for this information. We can let the classes and friends form vertices in a bipartite graph, with edges connecting pairs of classes and friends. Edges betwen friend i and class j denote that friend i has attended class j. Thus, the bipartite graph can be separated into two groups C and F, where $i \in F$ denotes a friend and $j \in C$ denotes a class.

Now notice that our problem asks for an output of k friends and classes which cover information regarding all of the available classes. This means, each $j \in C$ will have to be reached by the algorithm. The output will be a set of vertices R which consists of $j \in C$ and $i \in F$. A vertex $j \in C$ will correspond to you attending a class, while a vertex $i \in F$ will correspond to you talking to a friend. Notice that if any vertex j is not in R, then all of the edges leading out of j must go to a vertex in R in order to have a valid output. Thus, each edge must have be connected to a class $j \in R$, a friend $i \in R$, or both. This means that the problem is equivalent to returning a set R such that every edge in the graph has at least one endpoint in R and $|R|$ is minimized. Note that this is just the vertex cover problem.

Now, we shall solve the vertex cover problem on a bipartite graph using an algorithm that runs in polynomial time. First, we shall create a network by creating a source vertex s and a target vertex t. For each class $j \in C$, we shall create an edge connecting s and j. This edge will have a capacity of 1. Also, for each friend $i \in F$, we shall create an edge connecting i and t with a capacity of 1. Also, set the edges connecting C to F to have a capacity of 1. This creates a network where all edges have a capacity of 1, and where everything flows from s to t through the bipartite graph in the middle of the network.

Now, to find a vertex cover, we find a min-cut S on the network. This can be done using Ford-Fulkerson for a running time of $O(Ef^*)$ which in this case turns out to be $O((n_1+n_2)^3) = O(n_1^3+n_2^3)$ since we have maximum flow n_1+n_2. Now that we have a min-cut S, we define sets $C_1 = C \cap S$, $C_2 = C - C_1$, $F_1 = F \cap S$, and $F_2 = F - F_1$. Finally, we create the set B which contains the set of vertices in F_2 which have neighbors in C_1. The algorithm will then output $C_2 \cup F_1 \cup B$ as the final answer.

The running time of the algorithm is dominated by the use of Ford-Fulkerson, which requires $O(n_1^3+n_2^3)$ time. We also have to create the sets C_1, C_2, F_1, F_2, and B, and find the union $C_2 \cup F_1 \cup B$. Each of these

take $O((n_1+n_2)\alpha(n_1+n_2))$ using the disjoint-set data structure (where $\alpha(n)$ is the inverse of the Ackermann function). Thus, we see that the total running time is $O(n_1^3 + n_2^3)$, which is polynomial in the inputs.

Now, we shall prove correctness. First we prove that the output of the algorithm is a vertex cover.

Lemma 2.1. *The output of the above algorithm is a vertex cover.*

Proof. The output of $C_2 \cup F_1 \cup B$ covers all edges that have an endpoint either in C_2 or in F_1 because it contains all elements in both of these sets. Therefore, we only need to see if the output of the algorithm covers everything in the sets C_1 and F_2. We know, though, that B contains all the vertices in F_2 which have neighbors in C_1. Therefore, all of the edges in C_1 and F_2 are also covered. This shows that the output is a vertex cover. □

Now that we have shown this is a vertex cover, all that is left is to show that the output is a minimal vertex cover. Once this is proven, the algorithm will be clearly correct because we have previously shown that the problem boils down to finding a minimal vertex cover on a bipartite graph.

Lemma 2.2. *There does not exist a vertex cover of the graph which is of smaller size than the output of the algorithm.*

Proof. Let the capacity of the min cut S be given by c. Then we know the capacity must be the number of edges that go from S to the complement of S. This is just the number of edges from s to the complement of S (which is $|C_2|$), the number of edges from S to t (which is $|F_1|$), and the number of edges with one endpoint in C_1 and one endpoint in F_2 (which is $|B|$). Thus, we have:

$$c = |C_2| + |F_1| + |B| \tag{2.3}$$

However, we know that the output is of size less than or equal to $|C_2| + |F_1| + |B|$. The capacity c of the min cut is equal to the cost of the maximum flow in the graph. Now suppose that there is a vertex cover with size less than the capacity c of the min cut. Then this would imply that less than c vertices are needed to cover all of the edges in the bipartite graph. This would also imply that one could create a min cut with less than capacity c because one could then choose the vertices in the vertex cover as one group of the min-cut. This would contradict the fact that c is the minimum capacity cut, however. Therefore, the smallest vertex cover is of size c, which is the output of the algorithm. □

This lemma shows that the algorithm returns a minimal vertex cover, which is sufficient to solve the problem. □

2.2. **Problem B.**

Problem 2.2. *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 will show that the problem is NP-hard. We will do this by reducing the set cover problem to the class problem. Recall that in the set cover problem, we are given a universe U and sets $s_1, s_2, \ldots, s_m$ such that $\bigcup_{i=1}^{m} s_i = U$. We are asked then to minimize $I \subset \{1, 2, \ldots, m\}$ such that $\bigcup_{i \in I} s_i = U$. We will show that this problem can be reduced into the class problem.

Note that sets $s_1, \ldots, s_m$ can only contain elements $j \in \{1, \ldots, n\}$ where U is just the set of all j. We shall think of each of the elements j as classes, and each set $s_1, \ldots, s_m$ as friends. We see that each j can be in multiple sets s_i, which corresponds to a class being taken by multiple friends. Let us assume we have an algorithm for the class problem which will tell us how to pick out k friends and classes that cover all the classes offered, and returns null if this is impossible.

If we have this algorithm, then any set-cover problem can be converted into a class problem in the following manner. Convert all sets $s_1, s_2, \ldots, s_m$ into friends and all elements in each of the sets $j \in \{1, \ldots, n\}$ into classes. Now let the algorithm run with $k = 1$. If the algorithm returns null (i.e. it is impossible to obtain a set of k friends and classes that cover the input), try with $k = 2$. Keep incrementing k until a non-null output is received.

Once there is a non-null output, we will convert it back into a valid output for the set-cover problem. If the algorithm chooses to attend class (instead of talk to a friend who attended that class), we can simply pick a friend who attended that class and has not already been selected. This is because talking to that friend takes 1 hour, and talking to him will be at least as good as going to class, because talking to him, you can possibly get more information about other classes. Moreover, we will always have a friend who has

attended that class, because in the set-cover problem, we assume that the universe is completely covered by the union of all the sets, so that each class must have been attended by at least one friend. Once we have done this, we can turn each s_i in the output of the algorithm, along with the s_i that we have converted from attending class, and return them as the answer to the set-cover problem.

For correctness, we can clearly see that the output covers the universe U, so we only need to check to see if it is the smallest possible covering. We shall prove this by contradiction. Suppose there was a smaller cover than the one outputted by the algorithm above. Then this covering, when it was converted in terms of the class problem, would cover all of the classes, but in a lower k. However, since the above algorithm is run for all k until the algorithm returns a non-null solution, the algorithm should have returned this better solution. This is a contradiction, and thus, the algorithm is correct.

To show that the problem is NP-complete, we note that we have reduced the set-cover problem into the class problem with a polynomial number of runs of the class problem. This is because we must have $k < m$ since by definition $\bigcup_{i=1}^{m} s_i = U$. Since we have only made a polynomial number of moves, the runtime changes only by a multiplicative term of $O(m)$, which does not change the fact that the problem is NP-complete. Thus, we see that the class problem is also NP-complete. $\square$