6.854
ADVANCED ALGORITHMS
PROBLEM SET 1

JOHN WANG

Collaborators: Jason Hoch, Ryan Liu, Varun Ganesan

1. Problem 1

Problem: Unlike regular heaps, Fibonacci heaps do not achieve their good performance by keeping the depth of the heap small. Demonstrate this by exhibiting a sequence of Fibonacci heap operations on n items that produce a heap-ordered tree of depth $\Omega(n)$.

Solution: Consider the following recursive series of operations. For the ith step of the recursion, we will assume there is a tree t_1 of depth i, composed of exactly i nodes. There is also a tree t_2 which is a single node such that $root(t_1) < root(t_2)$. We shall insert two nodes a and b such that $b < a < root(t_1) < root(t_2)$. Perform a delete-min operation on the set of trees. This operation will remove b since it is the minimum of the entire structure. This leaves us with 3 roots, namely $a, root(t_1)$, and $root(t_2)$. The delete-min operation will also perform a consolidation, so that a is merged with t_1, then the resulting tree is merged with t_2.

The resulting tree has a root of a, a left child of $root(t_1)$ and a right child of $root(t_2)$. Now, we perform a decrease key on $root(t_2)$ to a value lower than a. This will cut it off from the tree, and we will be left with a tree t_1' rooted at a and t_2'. We see that t_1' will have a depth of $i+1$ and t_2' will be a single node. Thus, we are back to our original datastructure with step $i+1$ and can recurse.

Note that we performed four operations: insert a, insert b, delete-min, decrease-key. Thus, we see that after n of these operations, we will have a tree of length $n/4 = \Omega(n)$. $\square$

2. Problem 2

Problem: Suppose that Fibonacci heaps were modified so that a node was cut only after losing k children. Show that this will improve the amortized cost of decrease key (to a better constant) at the cost of a worse cost for delete-min (by a constant factor).

Solution: First we will define our potential function as $\Phi = R + 2M/(k-1)$ where R is the number of roots and M is the number of mark bits. Examining the amortized cost of insert a_i is given by:

$$a_i = c + 1 + \Delta\Phi \tag{1}$$

Where c is the number of nodes cut on a given insert (due to cascading). The real cost is $c+1$ because c nodes are cut during cascading, each requiring constant time, and $+1$ because the node must be inserted into the data structure as well. The change is potential is given by:

$$\Delta\Phi = c + \frac{2(1-(k-1)(c-1))}{k-1} \tag{2}$$

Because c nodes are cut during the cascading, an additional c roots are created which accounts for the first c term in $\Delta\Phi$. Moreover, the number of mark bits decreases by $(k-1)(c-1)$ since we cut away c nodes, which means that $c-1$ of these nodes had $k-1$ mark bits already stored which were cleared when everything was cascaded. However, we added 1 mark bit to the last node in the cascading chain, which is why we have a change of $1-(k-1)(c-1)$ mark bits. Putting this into our expression, we obtain:

$$\begin{aligned} a_i &= 1 + c + c + \frac{2(1-(k-1)(c-1))}{k-1} && (3)\\ &= 1 + 2 + \frac{2}{k-1} && (4)\\ &= 3 + \frac{2}{k-1} && (5) \end{aligned}$$

Thus, when $k=2$, the cost for insert is 5, whereas when $k>2$, the cost for insert is less than 5. Thus, the change improves the amortized cost of decrease key to a better constant.

We are left to show that cutting nodes only after losing k children makes delete-min more expensive. We know that the real cost of delete min will be the number of roots r plus the maximum degree possible for the root. Since the potential function is only changed by the number of roots, we see that $\Delta\Phi = \text{max degree} - r$. Therefore, we see that the amortized cost is simply $a_i = O($ # of roots .

Now to find the maximum degree, we need to consider node x with children $y_0, y_1, \ldots, y_i$ in the order they were added. We know y_i becomes a child of x during consolidation and the degree of x is at least $i-1$ because $y_0, \ldots, y_{i-1}$ were already children of x. This means that y_i's degree could only have decreased by $k-1$ before it gets cut so that $deg(y_i) \geq i - 1 - (k-1) = i - k$. This means we can find a lower bound for the maximum degree, using S_m where S_m is the minimum number of nodes in a degree m subtree. This means:

$$\begin{aligned} S_m &\geq \sum_{i=k}^{m} S_{i-k} && (6)\\ S_m - S_{m-1} &\geq \sum_{i=k}^{m} S_{i-k} - \sum_{i=k}^{m-1} S_{i-k} = S_{m-k} && (7) \end{aligned}$$

Since $S_m - S_{m-1} = S_{m-k}$, we find that characterisitc polynomial can be written as $x^k - x^{k-1} = x^0$ or $x^k - x^{k-1} - 1 = 0$. When $k=2$, we have $x^2 - x - 1 = 0$ so that $x_1 = \phi$ is the dominant root. However, when $k>2$, we have $x_2 < \phi$ as the dominant root so that $\log_{x_1}(n) < \log_{x_2}(n)$. Since we know that the maximum degree when $k=2$ is less than the maximum degree when $k>2$, we see that delete-min has worse constants in the case when $k>2$. $\square$

3. Problem 3

On tradeoffs in the heap operations.

Problem: Let P be a priority queue that performs insert, delete-min, and merge in $O(\log n)$ time, and performs make-heap in $O(n)$ time where n is the size of the resulting priority queue. Show that P can be modified to perform insert in $O(1)$ amortized time, without affecting the cost of delete-min or merge (i.e. $O(\log n)$ amortized time). Assume that the priority queue does not support an efficient decrease-key operation.

Solution: We will store a linked list L of priority queues. The original priority queue P will be at the front of the linked list. When performing an insert, we will create a new priority queue and append it to L. When merging a new priority queue P_{new}, we first consolidate all of the priority queues which are not P, hence all the priority queues which are single nodes and have been created by an insert, and create an auxiliary priority queue P_a. Next, we merge P_a with P so that the entire data structure is now composed of a single heap P. Then, we merge the new priority queue P_{new} with the existing queue P. To perform a delete-min, we first consolidate all of the roots into a single priority queue. This can be done with the same routine used in merge, i.e. creating a priority queue of the singular roots and merging that auxiliary priority queue with P. Then, delete min can be performed on P in the usual manner.

To analyze this data structure, we will introduce a potential function $\Phi = \#$ of roots. The real cost of an insert is just a constant $O(1)$, and the change in potential is 1, so $a_i = O(1)$ for insert. To examine merge and delete-min, we will first examine the consolidation subroutine. In the consolidation subroutine, a make-heap operation is performed on all single-node roots. The real cost of the make-heap is $O(c)$, where c is the number of single-node roots. The merge part of the consolidation requires $O(\log n)$ real cost. The change in potential is given by $-c$, since there are now c fewer roots. Thus, the amortized cost of the consolidation subroutine is $O(\log n) + c - c = O(\log n)$. This means that the merge operation, which will merge another tree in $O(\log n)$ (and won't affect the potential), will cost $O(\log n) + O(\log n) = O(\log n)$ amortized time. The delete-min operation will perform a merge, then a regular delete-min from the priority queue P, which will take $O(\log n) + O(\log n) = O(\log n)$ amortized time.

Thus, we have created a data structure will allows insert in $O(1)$ but retains the amortized cost of delete-min and merge. □

Problem: Using the above technique, show that even binary heaps can be modified to support insert in $O(1)$ amortized time while maintaining an $O(\log n)$ time bound for delete-min. Note that binary heaps do not support merge in $O(\log n)$ time.

Solution: We will start out with a linked list L of priority queues as before. When we insert a node, we will simply create a new root and append it to L. We will also keep a priority queue Q which holds the roots of all other priority queues, and at first it will only contain the root of P. To perform delete-min, we will first perform a make-heap on all of the single-node roots, and insert the new heap's root into Q. After the new heap's root has been inserted into Q, we perform a delete-min on Q, then a delete-min on the tree which was the root of Q.

We will use a potential function $\Phi = \#$ of roots for the analysis. Insert requires amortized time $a_i = 1+1$, since it requires 1 in real cost to create a new node and the potential changes by 1. For delete-min, we first perform a make-heap on all of the single-node roots. Let's say that there exist c single-node roots, then the make heap requires $O(c)$ time. Inserting the heap into Q will require $O(\log n)$ time, since Q has a maximum size of n and therefore a maximum depth of $O(\log n)$. Delete-min on Q also requires $O(\log n)$. The next delete-min on the resulting heap requires $O(\log n)$ time it must also be of size $O(n)$, and hence height $O(\log n)$. The total amortized cost of delete-min is therefore $a_i = 3O(\log n) + c - c = O(\log n)$.

We have therefore achieved insert in $O(1)$ amortized time and delete-min in $O(\log n)$ time for a binary heap. □

4. Problem 4

Problem: Keeping a mark bit around in Fibonacci heaps may be wasteful. Suppose that instead, each time I do a cut, I ip a coin to decide whether to cascade that cut to the parent. If the coin is unbiased, show that the expected behavior of these markless Fibonacci heaps is like that of standard ones. Can I bias the coin, so that a cascade is more than 50% likely, to achieve the eect of cascading after (say) one and a half children are cut? Can this be used to improve the time for deletemin at the cost of increasing the time for decrease key?

Solution: We will use a potential function with two parts. The first part will be the usual number of roots, and the second part will be initiated at 0 and will have the following formula for updating it whenever a coin flip for cascading is made:

$$\Delta\Phi_2 = \left\{ \begin{array}{cc} +1/(1-p) & \text{no cascading} \\ -1/p & \text{cascading} \end{array} \right\} \tag{8}$$

Here p denotes the probability of cascading. Now let us examine the runtime on the Fibonacci heap when p is the probability of cascading the cut to the parent. The runtimes of insert and merge will be unaffected by the change, since cuts are not performed for these two operations.

However, for decrease-key, we will cascade up c nodes, then there must be an event where the coin flip decides not to cascade up the last node. For each node we cascade up, there will be a real cost of 1 to make the cut, and also a decrease in potential of $1/p$. Thus, the amortized cost of each cascade upwards is $1-1/p$. At the end, when the coin flip decides not to cascade up anymore, the final non-cascade will require a real cost of 1 and the potential will increase by $1/(1-p)$. Thus the amortized cost of last non-cascade will be $1+1/(1-p)$. Let us say we cascade up c nodes, then stop. The cost of a decrease-key will then be $c\left(1-\frac{1}{1-p}\right)+\left(1+\frac{1}{1-p}\right)$.

In order to find the expected amortized cost of decrease-key, we need to know the expected value of c. However, this is easy to calculate since we know p, the probability of cascading the cut to the parent. Thus, $E(c) = 1/p$ and we can rewrite our amortized cost of decrease-key as:

$$\frac{1}{p}\left(1-\frac{1}{1-p}\right)+\left(1+\frac{1}{1-p}\right) \tag{9}$$

In the case of $p = 1/2$ we can see that the expected amortized runtime of decrease-key will be 1. We can also figure out the cost for larger values of p. Differentiating, we see that the cost is an increasing function with respect to p:

$$\frac{da_i}{dp} = -\frac{1}{p^2}+\frac{2}{p^3}+\frac{1}{(1-p)^2} \tag{10}$$

This follows because $0 < p < 1$ so that $p^2 > p^3 > 0$ so that $\frac{1}{p^2} < \frac{2}{p^3}$. Thus, $\frac{da_i}{dp} > 0$ and the cost for decrease-key will increase as p rises. We have shown the first half of the problem–if $p > 50\%$, then the time for decrease-key increases. We are left to show that the time for delete-min will decrease.

We know that the real cost of delete min will be the number of roots r plus the maximum degree possible for the root. Since the potential function is only changed by the number of roots, we see that $\Delta\Phi = -r$. Therefore, we see that the amortized cost is simply $a_i = \#ofroots$. Now to find the expected degree, we need to consider node x with children $y_0, y_1, \ldots, y_i$ in the order they were added. We know y_i becomes a child of x during consolidation and the degree of x is at least $i-1$ because $y_0, \ldots, y_{i-1}$ were already children of x. This means that y_i's degree would only have decreased by $1/p$ in expectation before it gets cut so that $deg(y_i) \geq i-1-(1/p) = i-(1/p)$. This means we can find a lower bound for the expected maximum degree, using S_m where S_m is the minimum number of nodes in a degree m subtree. This is the same argument as in problem 2, but now we set $k = 1/p$.

Thus, the constant which is exponentiated will be the largest root of the characteristic polynomial $x^{1/p} - x^{(1/p)-1} - 1 = 0$. As $1/p$ increases, it is clear that the largest root x will decrease (by the discussion from problem 2). Therefore, we see that the number of roots is given by $O(\log_x(n))$, so that the cost of delete-min is also $O(\log_x(n))$. Sicne x increases as p increases, we see that the constants for delete-min will be improved as the probability of cascading increases. $\square$

5. Problem 5

Problem: Show how to use the techniques of persistent data structures to preprocess a tree in $O(n \log n)$ time so as to allow LCA queries to be answer in $O(\log n)$ time. Aim for a simple solution here, even if you solve part (b).

Solution: We will use a persistent version of the offline algorithm given in the problem. First, we associate with each node an extra field "name" and process the nodes of T is post order. However, we will make the tree into a persistent data structure by keeping a timestamped log in a fat node and using path copying for each union find we use. The order in which the nodes are processed by the algorithm in post order will be considered time in the persistent version of the data structure.

Thus, as we perform union-find operations in the preprocessing stage, we will be using fat nodes to store previous changes in the "name" attribute. In order to perform a query on nodes a and b, we will revert to time t where t is the last time either a or b had any changes. We will then move up the parent pointers until we find a parent pointer which is null at time t, then perform a find on the name of this node at time t.

This means that find will take $O(\log t)$ to find the correct tree to use and an additional $O(\log n)$ to find the node in that particular tree. Thus, there is a total real cost of $O(\log t) + O(\log n)$ to perform a find operation. To construct the data structure, we will have a real cost of $O(\log n)$ for each find and union operation, since find takes $O(\log n)$ and dominates the union operation. This means that preprocessing costs $O(\log n)$ per node. Since there are a total of n find operations in the sequence, we will have $O(n \log n)$ cost for the preprocessing.

We will use the potential function $\Phi = \#$ live nodes. Thus, the amortized time of find will be $a_i = O(\log t) + O(\log n) = O(\log n)$ because $t = O(n)$ as only one operation is done for each node in the preprocessing phase. Thus, we see that find will require $O(\log n)$ time. Next, we know that the preprocessing time will require $O(n)$ times the cost of make set, union, and find on a node. Since each of these costs $O(\log n)$ time in a union-find data structure, we see that the preprocessing time is $O(n)$.

This means that we can perform LCA queries in $O(\log n)$ and preprocess the tree in $O(n \log n)$ time. □

Problem: Improve your solution to take $O(n)$ preprocessing time.

Solution: N/A □