Massachusetts Institute of Technology
6.854/18.415: Advanced Algorithms
David Karger

Handout 5
September 28, 1999

Problem Set 3

Due: October 5, 1999.

Problem 1. Build an uncompressed suffix tree for "banana$". Show the structure and node traversal path for each suffix insertion. Mark the suffix links that are actually used as shortcuts in the efficient construction algorithm.

Problem 2. The **longest common substring** of two strings is simply the longest substring that appears in both. This problem is used as a common example for dynamic programming, which gives an $O(mn)$ time algorithm for handling two strings of length m and n. We will derive an $O(m+n)$ time algorithm.

() Draw the compressed suffix tree for "banana$"

() Add to your tree the set of suffixes of "cabana!" (this is slightly less messy than drawing the suffix tree for "banana$cabana!".)

() By eye, determine the longest common substring of "banana" and "cabana." Mark its corresponding node in your suffix tree. What is important about the subtree rooted at this node?

() Give an $O(m+n)$ time algorithm for finding the longest common substring in two strings of length m and n respectively.

Problem 3. Consider a data structure that supports the following operations on a forest of rooted trees with values at each node:

find-min(r) find the minimum value node in the tree with root r

decrease-key(v, x) decrease the value of node v to x

split(r) split the tree with root r by cutting off node r and deleting it (so all of the children of r become roots)

Give a data structure that supports m decrease-key operations, m find-min operations and n splits on a forest with n nodes in $O(m \log n)$ time. (Assume $m > n$.)

Hint: consider Euler tour trees.

Problem 4. The *least common ancestor* (sometimes called lowest common ancestor) of nodes v and w in an n node rooted tree T is the node furthest from the root that is an ancestor of both v and w.

The following algorithm solves the *offline* problem. That is, given a set of query pairs, it computes all of the answers quickly. It makes use a union-find data structure. (See Cormen, Leiserson, and Rivest Chapter 22.)

Offline LCA: Associate with each node an extra field "name". Process the nodes of T in postorder. To process a node, consider all of the query pairs it is a member of. For each pair, if the other endpoint has not yet been processed, do nothing. If the other endpoint has been processed do a find on it, and record the "name" of the result as the LCA of this pair. After considering all of the pairs, union the node with its parent, and set the "name" of the set representative to be the parent.

We leave it as an exercise (not to be turned in), that this algorithm is correct, and takes $O((n+m)\alpha(n))$ time. (If you haven't met α before, it is an inverse of Ackerman's function and grows VERY, VERY slowly—even slower than $\log * n$. It is only 4 on the number of particles in the universe.)

Of course, in some instances we would like to find least common ancestors *online*. That is, we aren't told all of the pairs up front; we get queries one at a time.

(a) 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 answered in $O(\log n)$ time. Aim for simple solution here, even if you solve part (b). **Hint:** path compression is messy for the persistent data structure, and is not necessary to achieve $O(\log n)$ time for union and find operations. Note also that nodes have arbitrary indegree, so path copying won't work.

* **(b)** Improve your solution to take $O(n)$ preprocessing time.

Problem 5. The previous is not the best online LCA algorithm. $O(1)$ query time is achievable, even if the queries arrive online. Let's get part-way there. Give an algorithm to preprocess an arbitrary rooted tree so that queries of the form "is node v an ancestor of node w" can be answered in $O(1)$ time. Hint: Consider Euler tours.

** **Problem 6.** Looking back at priority queues, we saw that very different structures applied when we considered the RAM model (which allows indirect indexing into an array) and derived bucket heaps, and when we considered the "pointer machine" model (which only allows fixed-size node structures with pointers to other nodes) and derived Fibonacci heaps. But we noticed today that we defined Fibonacci heaps to use an array of non-constant size (during consolidation of roots). Show that Fibonacci heaps can be made fully pointer based by finding a pointer machine scheme for consolidating the roots (you may need to maintain some extra pointers all the time to prepare for consolidation).