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

Handout 8
September 28, 1999

# Problem Set 3 Solutions

**Problem 1.** The uncompressed suffix tree for "banana-" is shown in Figures 1, 2 and 3. (Your TA could not print \$ on the figures for reasons beyond his control.)

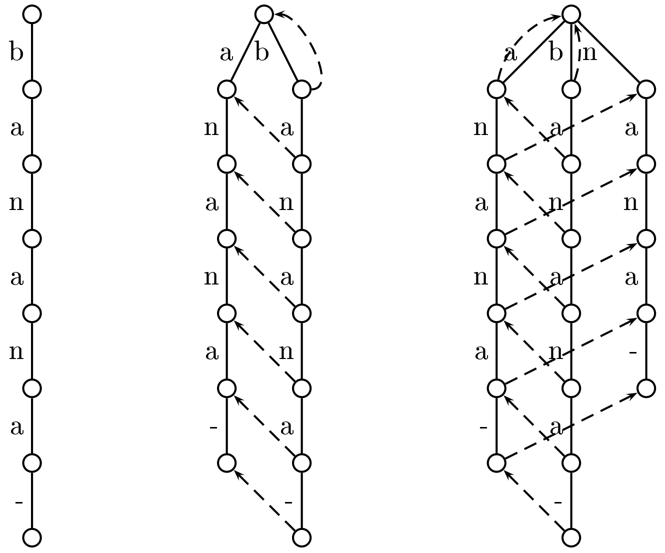

**Figure 1**: Suffix tree for "banana-": Step 1, 2 and 3

**Comments from graders:** Many solutions had minor errors in the figures.

**Problem 2.** A compressed suffix tree for the string "banana\$" is shown in Figure 4. For clarity, the edges are represented by substrings and not indices as in the actual algorithm. After adding suffixes of "cabana!", we get Figure 5. The solid node in Figure 5 corresponds to the longest common substring of banana and cabana.

A node is a substring of banana and cabana if it has both '\$' and '!' in its subtree of descendants. Among all such nodes, the marked node corresponding to the longest common substring has the maximum number of characters on its path to the root.

We can now design an algorithm for computing longest common substring in linear time. Let $w_1$ and $w_2$ be the two given strings. Let $w_1$ be $m$ characters long and $w_2$ be $n$ characters long.

**Lemma 1** *Consider the suffix tree containing suffixes of "$w_1\$w_2!$". String $w_x$ corresponding to node $x$ is a common substring iff the subtree rooted at $x$ has a suffix containing '\$' and another suffix not containing '\$'.*

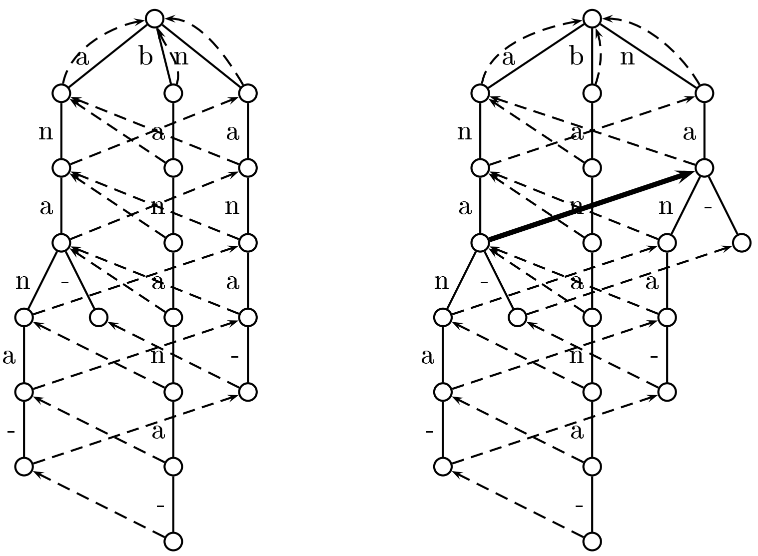

**Figure 2**: Suffix tree for “banana-”: Step 4 and 5

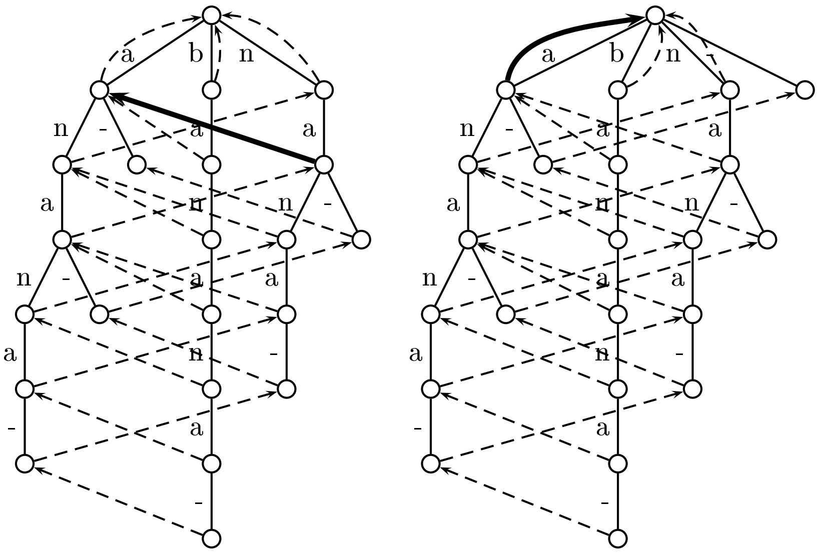

**Figure 3**: Suffix tree for “banana-”: Step 6 and 7

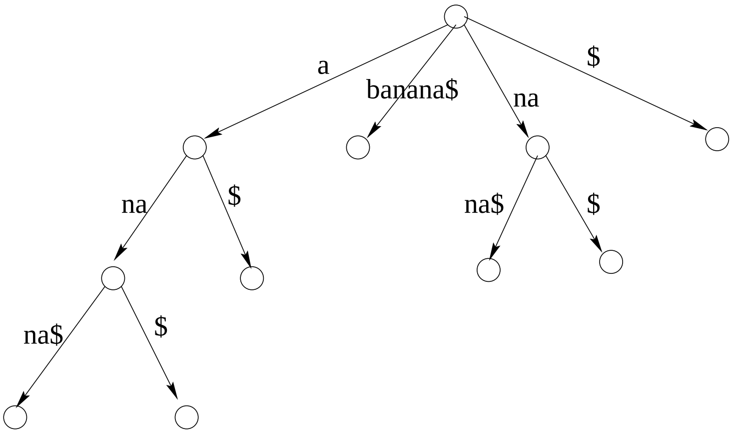

**Figure 4**: Compressed suffix tree for "banana$"

*Proof.* Let the suffixes be $s_1$ and $s_2$ respectively. Both of them start with $w_x$. The occurrence of $w_x$ in $s_1$ happens before the '$' character. Therefore $w_x$ is a substring of $w_1$. Using a similar argument about $s_2$, we know that $w_x$ is a substring of $w_2$.

Suffixes $s_1, \ldots, s_{m+1}$ contain '$' and suffixes $s_{m+2}, \ldots s_{m+n+2}$ do not. We construct a suffix tree for $w_1\$w_2!$. From this suffix tree we compute the longest common substring as follows.

(a) Mark every suffix tree node that has in its subtree a suffix containing "$". This can be done in linear time by performing a postorder traversal of the tree: when we examine a node, we have already checked all its children; mark the node if any of its children is marked. Do the same to mark every node with a suffix not containing "$".

(b) With one more tree traversal, find the deepest node marked with both features. Just maintain a "current depth" counter; increment it by the length of any edge traversed downward and decrement by the length of any edge traversed upward.

Our algorithm requires two linear time traversals of the linear-size suffix tree, so is linear.

**Comments from graders:** Some solutions had super-linear time computation of nodes corresponding to common substrings.

**Problem 3.** The forest of rooted trees can be represented as Euler tour trees. Vertices and edges of the forest are stored separately. Each vertex points to its active copy in the Euler tour tree. Each edge points to its two occurrences in the Euler tour tree. We use a splay tree implementation for the Euler tour trees. We will now augment this data structure to handle find-min and decrease-key queries.

**Definition 1** *The value of a node $x$ representing vertex $v$ is the value of vertex $v$ if the $x$ is the active copy and $\infty$ otherwise.*

Every node in the tree holds *a pointer* called the min pointer to the minimum value node in its descendants (including itself). Thus the root of the Euler tour tree holds a pointer to the the minimum value node in that tree.

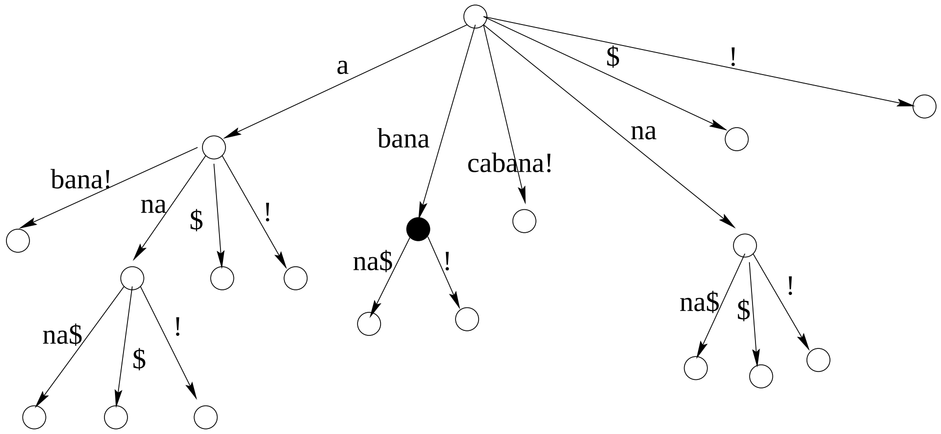

**Figure 5**: Compressed suffix tree having suffixes of "banana$" and "cabana!". The marked node shows the longest common substring.

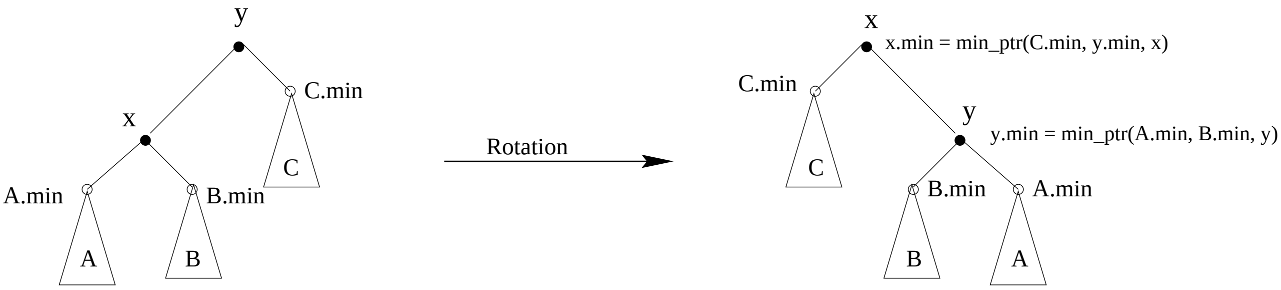

**Figure 6**: Updating min pointers on a rotation

After a rotation done on the tree, we can get back consistent values for best-descendant pointer as shown in Figure 6. A splay operation is composed of rotations. Therefore we can extend the splay operation so that consistent values of min pointers are maintained. Similarly we can update min pointers on splitting and joining edges. Thus all operations on our augmented splay trees will maintain consistent min pointers.

Find-min($r$) can be done by splaying $r$ to the root of the ET-tree and returning the min pointer. A decrease-key operation on node $x$ can be done by splaying $x$ and decreasing its key value. Notice that only $x$'s min pointer needs to be updated after this operation. Both these operations take $O(\log n)$ time.

The adjacency list of each vertex is maintained in the copy of the forest held in our data structure. Since our ET-tree supports split and join, splitting a root can be done by repeated removal of edges adjacent to $r$. At most $n - 1$ edges will be removed during the execution of the algorithm. Each edge removal takes $O(\log n)$ time and therefore the total time to do splits is $O(n \log n)$. The total running time of the algorithm is $O(m \log n)$ since $m > n$.

**Comments from graders:** Many solutions did not return the minimum node on find-min. Some solutions attempted to keep a separate heap. This idea does not work well since splitting the heap is a costly operation.

**Problem 4.** The offline algorithm for LCA traverses nodes in post-order and joins each node's component with its parent's component. When handling a query on $(v, w)$, the offline algorithm finds the name of $v$'s component when processing $w$ (if $v$ has already been processed). After processing all pairs incident on $w$, the algorithm "destroys" this version (say $V_w$) of the data structure. If we maintain $V_w$ in a partially persistent data structure, the query on the $(v, w)$ pair can be done online.

Specifically, we assign timestamp $t$ to the $t$th node in the post-order traversal. Let $\tau(v)$ denote the timestamp of node $v$. A version is maintained for each timestamp in $\{1, \ldots, n\}$. A LCA query on $(v, w)$, looks for the node with smaller timestamp (say $w$) and performs a find on $v$ in the $\tau(w)$th version of the data structure. We need to show how we can get a partially persistent union-find data structure with the following operations:

**Find**$(v, t)$**:** Find the name of $v$'s component in the $t$th version of the data structure.

**Union**$(w, p, t)$**:** Union the component with name $w$ and the component with name $p$ at timestamp $t$. The resulting component is named $p$. Timestamp $t$ has to be one more than the timestamp of the previous update.

(a) We use the union by rank heuristic of linking the smaller depth tree as the child of the larger tree's root. This achieves $O(\log n)$ depth trees. For clarity, we refer to parents (roots) in the union-find data structure as UF-parents(UF-roots).

**Lemma 2** *The UF-parent pointer of every node is initially null. During the execution of the algorithm, the UF-parent pointer of a node is updated at most once.*

*Proof.* The union by rank heuristic never changes the UF-parent pointer of a non-root node. Root nodes have null UF-parent pointer. The lemma follows.

We can augment the parent pointer with its time of creation, to make the traversal of this data structure partially persistent. Due to Lemma 2, the parent pointer is not changed again. To do a $find(v, t)$, we traverse UF-parent pointers from $v$ till we reach an edge with time-stamp more than $t$. The node we reach will be the UF-root of the component. We need however compute the name of the component.

So we maintain a log of operations done on the union-find data structure. The log is an array mapping time-stamps to the names of the components unioned. To compute the root node, we can lookup the name of the parent component corresponding to the time-stamp of the last edge traversed. It is evident that the find operation takes $O(\log n)$ time.

Union$(w, p, t)$ involves finding the UF-root of $w$'s and $p$'s components. This find operation works on the current version of the data structure. Then we do union by rank and timestamp the edge added with $t$. A log entry $(w, p)$ is added to the $t$th element in the log array. Each union operation takes $O(1)$ + time for 2 finds which is $O(\log n)$ time. So the preprocessing time is $O(n \log n)$.

(b) Notice that union is defined for nodes that are the names of their component. In the LCA algorithm this translates to the fact that a union of node $w$ to its parent $p$ has both $w$ and $p$ as the roots of their respective components.

So we maintain an ephemeral pointer from each root in the tree to the UF-root of its component. This cuts down the find cost while doing a union. The preprocessing is therefore $O(n)$.

**Comments from graders:** It is necessary to distinguish UF-roots and UF-parents from actual roots and parents. Some solutions had errors that did not recognize this difference. There were many solutions based on union-find data structures represented as linked lists and Euler tours. There were many simple and elegant solutions that did not use persistent data structures.

**Problem 5.** We construct an Euler tour sequence of the tree starting form its root. Each node holds the index of its first and last occurrence in the sequence.

**Lemma 3** *Node $v$ is the descendant of node $w$ iff the first and last indices of $v$ are within the first and last indices of $w$.*

*Proof.* Node $w$ is accessible from root only through its parent. So the first occurrence of $w$ corresponds to the edge connecting $w$'s parent to $w$. The edge from $w$ to its parent now becomes a cut-edge. Since we are interested in an Euler tour sequence, this cut-edge will not be traversed till all edges in $w$'s subtree are exhausted. Therefore all nodes occurring in-between the first and last occurrences of $w$ are descendants of $w$. This proves the "if" part of the lemma.

Since the (only) path from the root to $v$ passes through $w$, the first occurrence of $v$ in the Euler tour sequence is greater than that of $w$. The same argument applies for the reverse of the Euler tour sequence. We have shown the "only if" part of the lemma.

From Lemma 3, we have a data structure with $O(n)$-time preprocessing and constant time ancestor query.

**Comments from graders:** The solution to this problem is straightforward if Euler tours or dfs traversal is considered.