

BCB 567/CprE 548
Fall 2007
Homework 4
Solutions

I am going to start this solution page with a note on this homework and answering application questions using suffix trees in general. I am seeing a lot of very convoluted and complicated answers to simple problems. You should have noticed by now that the solutions to the various homework problems utilize tree traversals. For the benefit of those who do not have a lot of computer science background, or just need a refresher, I am going to talk a bit about traversals and what they are good for.

- **Top-down Traversal:** A top-down operation on a tree has the following property. The operation can be performed on a node as long as the operation has been performed on its parent. Top down traversals can make certain types of operations more efficient. For example, lets say you wanted to find the lowest ancestor of every leaf with property X . You *could* start at each leaf and traverse up the tree until you find the said ancestor. However, this would take $O(Lh)$ time in the worst case, where L is the number of leaves and h is the height of the tree. If instead you used a top down tree traversal to answer the question, you would take $O(N)$ time, where N is the number of nodes in the tree.

How to perform this operation in a top down manner? It is simple. Remember to describe a top down operation on a tree, we only have to answer the question: Given the answer for the parent, what is the answer for the child? In this case, we use the following logic. If I have property X , the lowest ancestor with property X is myself. Otherwise, copy the answer from my parent. It is that simple.

- **Bottom-up Traversals:** A bottom up operation on a tree has the following property. The operation can be performed on a node as long as the operation has been performed on all of its children first. Like top down operations, bottom up operations are made for certain problems. For example, lets say that we label all the leaves with property X or property Y . We wish a node to have property X only if all of the leaves in its subtree have property X and property Y only if all of the leaves in its subtree have property Y . The bad way to solve this problem would be to independently traverse the subtrees of each node to come to the answer. This would take $O(N^2)$ in the worst case.

Obviously, the answer can be answered very quickly in a bottom up manner. Given the answer for all of the children, we can quickly decide on the answer for the parent. In this case, if all children have property X , then the parent has property X . If all children have property Y , then the parent has property Y . It is that simple.

Many suffix tree applications make use of either (or both) of these two operations, as well as possibly finding some specific leaf (usually leaf 1) and then tracing a path to the root. It is important to understand these two operations and how to use them to truly master the use of the suffix tree.

1. Notice that if a string S is periodic in α , then the suffix starting at index $1 + |\alpha|$ is a prefix of suffix 1. Moreover, the inverse of this relationship is also true. If a suffix j is a prefix of suffix 1, this implies that the string is periodic with period length $j - 1$.

To see why this is the case, consider the following definition of a period. A string S is periodic with period k if and only if $S[i] = S[i + k]$ for all i . Now consider what it means for a suffix of a tree to also be a prefix of a tree. If this is true for suffix $k + 1$, then $S[1] = S[1 + k]$ and $S[i] = S[i + k]$ for any i . Thus the two statements are equivalent.

Let β be suffix j . We call β a *border* of S if it occurs at both the beginning and end of S . We have just shown that if $S = \alpha\beta$ and β is a border of S , then α is a period of S . Therefore, to find the smallest period of S , we can solve the simpler problem of finding the largest border of S . This can be done easily using suffix trees. Consider the suffix tree of S and the path from the leaf 1 to the root. Let u be the lowest node on this path with a child with edge label $\$$. Then $\text{path-label}(u)$ is a border of S and S has period of length $n - \text{str-depth}(u)$. Moreover, because u is the lowest such node, then this is the shortest period of S .

Our algorithm proceeds as follows. Start at leaf labeled 1 (this can be found either by DFS or by finding the pattern S in the suffix tree. Walk up the path until discovering the first node u with child edge labeled $\$$. Set α equal to the first $n - \text{str-depth}(u)$ characters of S .

2. We can easily identify a unique substring of S . This is a string that starts at the root and then continues until we enter the edge label of a leaf. Let u be the parent of a leaf node v , let α be the $\text{path-label}(u)$, and β be the edge label of $u \rightarrow v$ minus the $\$$, $|\beta| > 0$. Let $\beta_1, \beta_2, \dots, \beta_{|\beta|}$ be the unique prefixes of beta. Because in a suffix tree (not a generalized suffix tree) each leaf has exactly one label, we know that all substrings of S constructed by concatenating α and β_i are unique in S .

The question is, “Which of these strings is minimal?” The answer is obvious: $\alpha\beta_1$ is minimal, because if we remove the last character, we are left with α which is the path label of u , and therefore a right maximal repeat in S and not unique. We see that for each leaf with edge label of length at least 2, there is exactly one such minimal unique substring, and that for each leaf with edge label $\$$, there is no such minimal unique substring.

It is easy to prove the reverse direction, that each minimal unique substring corresponds to path in the suffix tree ending at the first character in the label of a leaf node, but we will not do that here.

We have observed that there is a one-to-one correspondence between minimal unique substrings and leaves in the suffix tree. The $O(n)$ algorithm for enumerating the minimal unique substrings in S follows. We will output tuples (i, j) for each string, where i is the starting index in S and j is the ending index. Perform a tree traversal. For each leaf i encountered in the traversal, look at its edge label: (k, j) . If $j - k > 0$, then the string starting at position i and ending at position $n - (j - k)$ is a minimal unique substring. If the length of this string $(n - j + k - i + 1)$ is greater than or equal to l , then output $(i, n - j + k)$ as an answer and continue.

3. This question can be solved in a manner similar to the method used for enumerating all right maximal common substrings between two strings S_1 and S_2 . For each node u in the

generalized suffix array, we will store a length two binary array X_u . We will fill the binary array so that the following property holds:

- $X_u[0] = 1$ if and only if a suffix of A occurs in the subtree rooted at u .
- $X_u[1] = 1$ if and only if a suffix of any string $s \in S$ occurs in the subtree rooted at u .

Now we can fill this array in $O(1)$ time per node using a bottom up traversal of the tree. The leaves are filled by looking at the leaf label(s). $X_u[i] = 1$ if and only if there exists a child of u , which we call v , such that $X_v[i] = 1$.

Now we will store another variable for each node u , $primer(u)$. $primer(u)$ holds the minimum primer length for a node. This is the length of the shortest prefix α of $path-label(u)$, such that $|\alpha| > k$ and α does not occur in any s_i .

Once we have finished calculating the values of X_u for all nodes, we can calculate $primer(u)$ for all nodes in the following top down manner. First initialize $primer(u) = \infty$ at all nodes. Now, for each node u with parent v , do the following. If $X_u[0] = 1$ and $X_u[1] = 0$ and $str-depth(u) \geq k$, then set $primer(u) = \min(primer(v), \max(str-depth(v) + 1, k))$. Otherwise do nothing.

Now each leaf u labeled (A, i) has stored in the variable $primer(u)$ the length of the shortest substring of A of length at least k and starting at position i that does not occur in any $s \in S$. It has the value ∞ if such a substring does not exist.

4. The answer to this problem is very similar to problem 2. The question is almost asking us to find the minimal unique substrings for a generalized suffix tree with $l = 1$. However, there are some notable differences. First, we wish to report the length of the minimum such substring for each string. Second, we allow a substring to be a repeat within a string, as long as it is unique to that string.

First, construct the generalized suffix tree for S . We will augment the suffix tree with the value $label(u)$ for each node. $label(u) = i$ if the subtree rooted at u contains only suffixes for i . $label(u) = null$ otherwise. Initialize $label(u) = null$ for all nodes. Then, using a bottom tree traversal, update $label(u) = i$ if $label(v) = i$ for all children v of u .

We will enumerate the lengths of all *minimal* unique substrings as tuples of the form (i, j) where i is the string identifier and j is the length of the substring. Perform a DFS of the tree. For each node u encountered in the traversal, such that $parent(u) = v$, $label(u) \neq null$, $label(u) \neq label(v)$, and $edge-label(v, u) \neq '$'$, we report a minimal unique substring: $(label(u), str-depth(v) + 1)$.

Out of all of the minimal unique substrings enumerated by the above method, we must find the minimum length substring for each string. We create an array of size m to store our answers and accomplish this with a single scan of the list of minimal unique substrings, keeping track of the minimum for each string.

The runtime and space of this algorithm is $O(\sum |s_i|)$, because we must first do a tree traversal calculate $label(u)$, do one more tree traversal to report all minimal unique substrings, and then finally traverse our enumerated list to find the minimum unique substrings for each string. The enumerated list of minimal unique substrings must be less than or equal to the number of leaves in the tree.

5. We will transform this problem using suffix trees. Let i be some position in S_2 . Then there exists some maximum length string α_i that is a substring of S_1 and is a substring of S_2 starting at i . If $|\alpha_i| > k$, then we can choose to start some T_j at position i and can then choose T_{j+1} to start at any index in the range $i + k + 1$ to $i + |\alpha_i| + 1$.

To make this problem easier to solve, we will construct a matrix M , such that:

$$M[i] = \begin{cases} |\alpha_i| & \text{if } |\alpha_i| \geq k \\ 0 & \text{otherwise} \end{cases}$$

The construction of this matrix can be done very quickly using generalized suffix array of the two strings. As we have done for many other applications, we will create a two entry binary array X_u at node u such that $X_u[0] = 1$ if and only if the subtree rooted at u contains a suffix from S_1 and $X_u[1] = 1$ if and only if the subtree rooted at u contains a suffix from S_2 . This can be filled in a bottom-up manner.

We will then do a top down traversal of the tree, and label each leaf with the string depth of the lowest ancestor of that leaf with both $M[0] = 1$ and $M[1] = 1$. These labels on the leaves of S_2 correspond exactly with entries in array M . Each entry in M can be interpreted as an interval starting at i and ending at $i + M[i] - 1$.

Therefore we have reduced the problem of k -cover to the following problem. Given a set of intervals, is there any selection of intervals such that the starting point of each selected interval is at least k units distant from the starting point of any other interval, and the entire range is covered?

We will show that that greedy algorithm works by transforming the problem to the reachability problem in graphs. Consider constructing the following graph: A node corresponds to each $M[i]$, with $M[i] > 0$. We also will create the node $M[n]$, even though $M[n] = 0$ unless $k = 1$. An edge occurs between $M[i]$ and $M[j]$ if and only if $j - k > k$ and $M[i] + i \geq j$. Now the problem of finding a k -cover is the same as finding a path between $M[1]$ and $M[n]$ in this graph. In fact, the shortest such path (which can be found using Dijkstra's algorithm) corresponds to the minimum sized k -cover.

However, we can't actually perform this transformation if we want to have an $O(n)$ algorithm, as the size of the graph might be as large as $O(n^2)$ in general. Therefore, we will transform the greedy shortest path algorithm work on the M matrix itself.

Consider the following algorithm: Create a matrix T . Let i_T be an index into T and i_M be an index into M . Perform the following scan algorithm:

- $i_M \leftarrow 1$;
- $i_T \leftarrow 1$;
- $T[i_T] \leftarrow 1$;
- While $i_T \leq n$
 - While $M[i_M] = 0$ or $T[i_M] = 0$, $i_M \leftarrow i_M + 1$.
 - if $i_M > i_T$ then return that there is no k -cover.
 - While $i_T < M[i_M] + i_M$
 - * if $i_T > i_M + k$ then $T[i_T] \leftarrow i_M$
 - * $i_T \leftarrow i_T + 1$
 - $i_M \leftarrow i_M + 1$
- Now we can enumerate the breakpoints in the k -cover.
 - $t \leftarrow T[n]$
 - while $t \neq 1$
 - * output t
 - * $t \leftarrow T[t]$
 - output t

By inspection of the algorithm, we see that each array element in M and T is traversed exactly once while filling T , and at most once when tracing back through T . Therefore, the algorithm runs in $O(|S_2|)$ time, the size of the each array. Because the arrays M and S are of size $O(|S_2|)$, and the suffix array is of size $O(|S_1| + |S_2|)$, the total space required is $O(|S_1| + |S_2|)$.