Optimal Parallel Suffix Tree Construction*

Ramesh Hariharan†

April 1, 1997

Abstract

An $O(m)$-work, $O(m)$-space, $O(\log^4 m)$-time CREW-PRAM algorithm for constructing the suffix tree of a string s of length m drawn from any fixed alphabet set is obtained. This is the first known work and space optimal parallel algorithm for this problem. It can be generalized to a string s drawn from any general alphabet set to perform in $O(\log^4 m)$ time and $O(m \log |\Sigma|)$ work and space, after the characters in s have been sorted alphabetically, where $|\Sigma|$ is the number of distinct characters in s. In this case too, the algorithm is work-optimal.

Keywords. Suffix tree, strings, pattern matching, string periodicity.

1 Introduction

A suffix tree of a string s is a compacted trie of all suffixes of s. It is a powerful data structure which finds applications in many string processing algorithms. Some examples are string matching, finding all squares or repetitions in a string [AP83], computing substring statistics [AP85], approximate string matching [LV86], text compression [RPE81], analyzing genetic sequences [CHM86], etc. A very powerful feature of the suffix tree is that after construction and suitable preprocessing, the longest common prefix of any two substrings of s can be found in constant time; further, the preprocessing required for this step can be performed optimally in logarithmic time.

The first sequential algorithm for constructing the suffix tree was obtained by Weiner [W73]. This algorithm takes $O(m \log |\Sigma|)$ time, where $|\Sigma|$ is the number of distinct characters in s and $m = |s|$. McCreight [M76] gave a more efficient construction with the same asymptotic time bound. For strings from a fixed alphabet, both algorithms take linear, i.e., $O(m)$ time.

The first parallel algorithm for this problem was due to Landau and Vishkin [LV86]; it runs in $O(\log m)$ time and does $O(m^2)$ work. Apostolico et al. [AILSV86] used the

*Work supported by NSF grants CCR-8902221, CCR-8906949 and CCR-9202900.

†Max-Planck Institut für Informatik, Im Stadtwald, Saarbrücken, Germany, 66123. Email: ramesh@mpi-sb.mpg.de. Work done when the author was a student at the Courant Institute of Mathematical Sciences, New York University

technique of "naming" [KMR72] to obtain an algorithm which takes $O(\log m)$ time and does $O(m \log m)$ work in the arbitrary CRCW-PRAM model. This algorithm requires superlinear, i.e., $O(m^2)$ space, which can be reduced to $O(m^{1+\epsilon})$ at the expense of an $O(\frac{1}{\epsilon})$ factor in the time bound. A variant of this algorithm runs in $O(\log^2 m)$ time, does $O(m \log^2 m)$ work and takes linear space in the CREW-PRAM model.

Note that constructing the suffix tree of s implicitly involves ordering the characters in s by alphabet; therefore, no parallel algorithm can have a better work bound than the algorithm in [AILSV86] when the characters in s are drawn from a general and potentially infinite alphabet. However the algorithm of Apostolico et al. [AILSV86] runs in the above mentioned time, space and work bounds even for binary alphabet. The existence of a linear work algorithm for the case when the characters in s are drawn from a fixed alphabet has been an important open problem in string processing.

The Main Result. We give the first work-optimal parallel algorithm to construct the suffix tree of a binary string s. The algorithm does $O(m)$ work and takes $O(\log^4 m)$ time and $O(m)$ space in the CREW-PRAM model. This algorithm can be generalized to handle strings s drawn from any fixed alphabet set to perform in the same bounds. For strings s drawn from a general alphabet set, the algorithm can be generalized to perform in $O(\log^4 m)$ time, $O(m \log |\Sigma|)$ work, and $O(m \log |\Sigma|)$ space after the characters in s have been sorted by alphabet, where $|\Sigma|$ is the number of distinct characters in s. Since constructing the suffix tree of s implicitly involves ordering the characters in s by alphabet, no parallel algorithm can have a better work bound.

An Application. As an application, we obtain the first optimal work algorithm for multi-pattern matching, i.e., matching a set of patterns of collective size M against a text of size N, where the patterns and text are drawn over some fixed alphabet set. This algorithm takes $O(\log^4 M_l)$ time, $O(M + N)$ work and $O(M + N)$ space in the CREW-PRAM model, where M_l is the length of the longest pattern. Previously, the algorithm with the smallest work bound was due to Muthukrishnan and Palem [MP93]; this algorithm took $O(\log M_l)$ time, $O(N\sqrt{\log M_l} + M)$ work and $O(M^{1+\epsilon} + N)$ space in the arbitrary CRCW-PRAM model.

Other Recent Developments. Independent of this work, Sahinalp and Vishkin [SV93] obtained an algorithm for suffix tree construction which takes $O(\log^2 m)$ time , $O(m \log \log m)$ work and $O(m^{1+\epsilon})$ space in the arbitrary CRCW-PRAM model. A randomized version of their algorithm takes $O(\log^2 m)$ time, $O(m \log^* m)$ work and $O(m)$ space. Both algorithms work for strings drawn from an alphabet of size polynomial in m and use the "naming" technique. They also modify this algorithm for the case of fixed alphabet to run in $O(\log^2 m)$ time, $O(m)$ work and $O(m^{1+\epsilon})$ space in the arbitrary CRCW-PRAM model. Note that this algorithm is time-wise better than ours but is not space optimal and requires a stronger model; further the techniques used by them (symmetry breaking and naming) are very different from the combinatorial techniques we use. More recently, Farach and Muthukrishnan [FM93] have obtained a randomized algorithm which takes $O(\log m)$ time, $O(m)$ work and $O(m)$ space in the arbitrary CRCW-PRAM model for strings drawn from a constant sized alphabet.

Main Techniques Used. The algorithm of Apostolico et al. [AILSV86] for suffix tree

construction uses some interesting algorithmic techniques but does not use any of the properties intrinsic to strings. Our approach is quite different and exploits combinatorial and periodicity properties of strings along with the repetitive nature of the suffix tree. It is interesting to note that many parallel algorithms use superlinear space in order to exploit the fixed (or restricted) alphabet assumption [MP93, BD+91, KLP89] in order to reduce the work bound; in contrast, we exploit the fixed alphabet assumption to obtain an optimal work algorithm using just linear space. The following are two of the essential ingredients of the algorithm which may also be of independent interest.

• Our scheme hinges on an upper bound we show for the following combinatorial question. If i_x is the number of times substring x occurs in s, then what is the sum $\Sigma \min\{i_{1x}, i_{0x}\}$, over all binary strings x of length r, as a function of $m = |s|$ and r? Here $0x$ just denotes the string x preceded by the symbol 0, and similarly for $1x$. Note that this sum is closely related to the number of nodes in the suffix tree of s which have two suffix links incident upon them. We show that this sum is upper bounded by $\frac{m \log m}{r}$.

• We give a concurrent version of McCreight's sequential algorithm for suffix tree construction which constructs a *single* data structure in which the suffix trees of the binary strings $s_1, \ldots, s_k$ are merged. This algorithm runs in $O(\max_i\{|s_i|\} \log k)$ time, does $O(\Sigma_i |s_i|)$ work, and takes $O(\Sigma_i |s_i|)$ space. In this algorithm, one processor is assigned to each string s_i; each processor performs McCreight's sequential algorithm on its respective string. All processors work on the same data structure. This leads to two problems, the more critical of which is the fact that a processor which inserts a node x into this data structure might have to wait for the suffix link of x's parent to be in place before proceeding. Indeed long paths may appear in the data structure, with each node in this path waiting for the suffix link of its parent to be in place. We give a non-trivial amortization argument to show that the total work done is indeed linear, in spite of processors waiting for suffix links to be in place.

Algorithm Overview. The suffix tree construction algorithm uses the following scheme. Compacted tries are built for substrings of s of successively increasing lengths in $O(\log m)$ stages; the final trie will be the suffix tree. In the first stage, the substrings have size r, where r is polylogarithmic in m. In the subsequent stages, they have lengths $2r, 2^2 r, 2^3 r, \ldots$.

The first stage provides us with the first challenge in solving this problem as it is not clear how even this stage can be accomplished in $O(m)$ work. We use the concurrent version of McCreight's algorithm described above. s is split into pieces of length $2r$, each pair of adjacent pieces overlapping in exactly r characters. One processor is assigned to each piece. Each processor performs McCreight's sequential algorithm to insert all substrings of length r in its piece into a compacted trie. All processors insert substrings into the same data structure. This takes $O(r \log m)$ time and $O(m)$ work.

Each subsequent stage can be performed easily by sorting up to m items in each stage. However, this leads to an $O(m \log^2 m)$ work algorithm. The main challenge now is to get around the problem of sorting up to m items in each stage. In order to avoid performing this computation, we use the following scheme. Each stage proceeds in two steps.

Recall that in a given stage, all substrings of a particular length are processed. In the first step, the compacted trie is computed for a carefully chosen subset of these substrings. Using the combinatorial property mentioned above and exploiting periodicity properties, we show that the the number of such substrings which are not periodic with small periods is $O(\frac{m}{log^2 m})$, and that those substrings which are periodic with small periods can be grouped into $O(\frac{m}{log^2 m})$ families. We can then "sort" these chosen substrings using Cole's parallel merge sort algorithm [C88] so that the total work done in sorting is $O(\frac{m}{\log m})$ per stage, which is $O(m)$ over all stages.

In the second step, by exploiting the repetitive structure of the trie, we obtain the compacted trie for all substrings using the compacted trie for the substrings chosen above. These chosen substrings enable a graph defined on the leaves of the trie obtained in the previous stage to be partitioned into trees; this graph partitioning is critical to the efficient performance of the second step.

This paper is organized as follows. Section 2 gives the requisite definitions and describes some preliminary procedures, Section 3 describes a key combinatorial property of strings, Section 4 describes how the first stage is performed, and Sections 5, 6, 7, 8, and 9 describes the remaining stages.

2 Definitions and Preliminaries

The *compacted trie* of a set of strings $s_1, s_2, \ldots, s_k$, none of which is a prefix of another string in the set, is a tree T defined as follows. T has a leaf $lf(s_i)$ for each s_i, $1 \leq i \leq k$. Each internal node in T has at least two children. Associated with each edge e in T is a string. The strings associated with the edges on the path from the root to $lf(s_i)$ yield the string s_i when concatenated in path order. Further, for any two leaves $lf(s_i)$ and $lf(s_j)$ with least common ancestor x, the strings associated with the edges on the path from the root to x, when concatenated in path order, give the longest common prefix of s_i and s_j. It follows that if e and e' are edges leading from a node x to any two of its children, the strings associated with e and e' begin with distinct characters. In addition, the edges leading from x to its children are ordered from left to right in increasing lexicographic order of the associated strings. Fig.1. shows the compacted trie of strings $1011, 0100, 1101, 1110, 1010$.

The above definition of compacted tries is generalized to the case when some of the strings $s_1, \ldots, s_k$ are identical by having one leaf per distinct string rather than one leaf per string. All identical strings are associated with the same leaf in T.

Let s be a binary string of length m. The *suffix tree* ST of s is defined as follows. Let p be the string $s\$$, where \$ is a new symbol. Then, clearly, no suffix of p is a prefix of another suffix of p. ST is a compacted trie of all suffixes of p except the suffix '\$'. Fig.1. shows the suffix tree of the string 110110. The *r-suffix tree*, r-ST, of s, $r \geq 1$, is defined to be compacted trie of all substrings of p of length r and all suffixes of p of length at most r, excluding the suffix '\$'. Clearly, for any $r > m$, $ST = r$-ST.

In any r-ST, let $str(x)$ denote the string obtained by concatenating the strings associated with the edges on the path from the root to the node x in r-ST. An r-ST

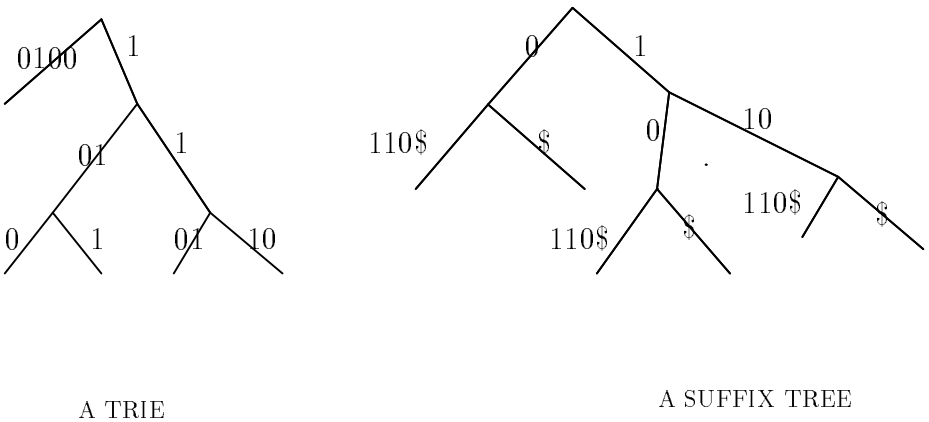

Figure 1: Tries and Suffix Trees

will be represented as follows. The nodes will be stored in contiguous array locations. Each node x of the tree has pointers to its children (which are at most 3 in number), a parent pointer $par(x)$, a suffix link pointer $suf(x)$, and a field $len(x)$ which equals $|str(x)|$. If x is an internal node then the suffix link pointer $suf(x)$ points to a node y such that $|str(y)| = |str(x)| - 1$ and $str(y)$ is a suffix of $str(x)$. For every internal node x in any r-ST, such a node y is guaranteed to exist. For the root $root$, $suf(root) = par(root) = root$ and $len(root) = 0$. The string $p[i \ldots j]$ associated with an edge e is denoted by $substr(e)$ and is represented by the pair of indices i, j.

Let the characters in p be indexed from $1 \ldots m+1$. Two strings u and v are said to be *consecutive substrings* of p if for some index j there is an occurrence of u in p beginning at index j and an occurrence of v in p beginning at index $j+1$.

A subtree of some r-ST T *induced by* a subset L of its leaves is the tree obtained by removing all subtrees of T containing only leaves outside L and then compacting paths of degree two nodes in the resulting tree into a single edge.

We impose the order $\phi < 0 < 1 < \$$ on the alphabet, where ϕ stands for the empty symbol, i.e., a blank. We say that $u < v$ for strings u, v if string u is lexicographically less than v. We say that $l_1 < l_2$ for nodes l_1, l_2 of some r-ST if $str(l_1) < str(l_2)$.

We assume the CREW-PRAM model of computation in the rest of this paper. The following primitives will be useful.

Comparing Leaves. The following primitive is due to Schieber and Vishkin [SV88]. A tree with m nodes can be preprocessed in $O(\log m)$ time and $O(m)$ work following which the relative order of any two leaves can be determined in constant time and work.

Computing Induced Tries. Given an ordered set S of substrings of p and an oracle which gives the length of the longest common prefix of u, v for any two consecutive strings $u, v \in S$, there is a scheme for constructing the compacted trie for the strings in S in $O(\log |S|)$ time and $O(|S|)$ work [FM93]. This trie is called the trie *induced by* the strings in S.

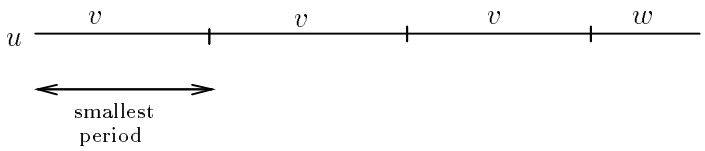

Figure 2: A Periodic String

We also need the concept of *periodicities* in strings. x is said to be a *period* of string u if for all j, $1 \leq j \leq |u| - x$, $u[j] = u[j + x]$. u is said to be *periodic* if its smallest period x is at most $\frac{|u|}{2}$; in fact, u is said to be periodic *with* period x (see Fig.2.). A string u is said to be *primitive* if for no u' and $k > 1$ is $u = (u')^k$. Lemma 2.1 is classic and follows from [LS62].

Lemma 2.1 *Suppose string u is periodic. Then there exists a primitive string v and a prefix w of v such that $u = v^k w$, for some $k \geq 2$. Further, $x \leq |u| - |v|$ is a period of u if and only if x is a multiple of $|v|$. In addition, the smallest period of the string $u' = ua$, where $a \neq u[|u| - |v| + 1]$, is at least $|u| - |v| + 1$.*

Our goal is to construct the suffix tree ST of s in polylogarithmic time and linear work, using just a linear amount of space. We do so by constructing the sequence of trees $r\text{-}ST, 2r\text{-}ST, 2^2r\text{-}ST, \ldots, 2^ir\text{-}ST$, where i is the smallest number such that $2^ir \geq m + 1$ and r, to be fixed later, is polylogarithmic in m. We assume that $m \geq 8$.

3 A Basic Property

In this section, we describe a basic combinatorial property of strings which will be critical to the performance of the algorithm. Consider any binary string v.

Definitions. For a binary string u of length less than $|v|$, let i_u denote the number of occurrences of u in v. For $1 \leq r \leq |v| - 1$, let $n_r = \Sigma \min\{i_{0u}, i_{1u}\}$, where the summation is over all binary strings u of length r. $\Sigma_{r=1}^{|v|-1} n_r$ is called the bifurcation number of v.

Lemma 3.1 *$n_r \leq n_{r-1}$, for $1 \leq r \leq |v| - 1$.*

Proof. Let z be a binary string of length $r - 1$. We show below that $\min\{i_{0z0}, i_{1z0}\} + \min\{i_{0z1}, i_{1z1}\} \leq \min\{i_{0z}, i_{1z}\}$. Since $n_r = \Sigma[\min\{i_{0z0}, i_{1z0}\} + \min\{i_{0z1}, i_{1z1}\}]$ and $n_{r-1} = \Sigma \min\{i_{0z}, i_{1z}\}$, where the summations are over all binary strings z of length $r - 1$, the lemma follows.

Note that $i_{0z0} + i_{0z1} \leq i_{0z}$ because if either $0z0$ or $0z1$ occurs starting at $v[i]$, then $0z$ occurs starting at $v[i]$. Similarly, $i_{1z0} + i_{1z1} \leq i_{1z}$. Therefore, $\min\{i_{0z0}, i_{1z0}\} + \min\{i_{0z1}, i_{1z1}\} \leq \min\{i_{0z0} + i_{0z1}, i_{1z0} + i_{1z1}\} \leq \min\{i_{0z}, i_{1z}\}$, as claimed. □

Lemma 3.2 *The bifurcation number $\Sigma_{r=1}^{|v|-1} n_r$ of v is at most $|v| \log |v|$.*

Proof. Consider the suffix tree ST of the v', the reverse of v. Each prefix of v corresponds to a distinct leaf in ST.

Consider those internal nodes x with at least two edges e and e' leading to children y and y' of x, respectively, such that the $substr(e)$ begins with a 0 and $substr(e')$ begins with a 1. Let l_x^0 be the number of leaves in the subtree rooted at y and let l_x^1 be the number of leaves in the subtree rooted at y'. Clearly, the bifurcation number of v is exactly the sum $\Sigma \min\{l_x^1, l_x^0\}$ over all such internal nodes in ST.

We call $\min\{l_x^1, l_x^0\}$ the *count* at node x. Clearly, for each leaf of ST, there are at most $\log |v|$ internal nodes at which it can contribute to the count. The lemma follows. □

Corollary 3.3 $n_r \leq \frac{|v| \log |v|}{r}$, *for* $1 \leq r \leq |v| - 1$.

Remark. Leszek Gąsieniec [G93] has shown that there exist strings v for which $n_r = \Theta(\frac{|v| \log |v|}{r})$, for $r = \Omega(\log |v|)$. So Corollary 3.3 is tight to within constant factors for $r = \Omega(\log |v|)$.

4 Optimal r-Suffix Tree construction in O(r log m) time

Given some $r \leq m$, we show how T_0, the r-ST of s, is constructed in $O(r \log m)$ time and $O(m)$ work on a common CRCW-PRAM. Recall that the r-ST is a compacted trie of all substrings of p of length r and all suffixes of p of length at most r, except the suffix '\$'. The value of r will be fixed later at some power of $\log m$.

4.1 The Algorithm

The procedure for constructing T_0 has two steps.

Step 1. In this step, a tree T_0' is built in $O(r \log m)$ time and $O(m)$ work. s is partitioned into pieces of length $2r$ (the rightmost piece could be shorter), each pair of adjacent pieces overlapping in exactly r characters. There are $O(\frac{m}{r})$ such pieces. Let these pieces be denoted by the strings $s_1, s_2, \ldots, s_k$. Let p_i be the string $s_i\$_i$, where $\$_i \neq \$_j$, for any $i \neq j$. T_0' is the compacted trie of all suffixes of each of the p_i's, except the suffixes $\$_i$. In other words, T_0' is a single data structure containing the suffix trees of all the s_i's.

To construct T_0', one processor is assigned to string p_i. The processor associated with p_i inserts the suffixes of p_i, except $\$_i$, in left to right order into T_0' using the sequential algorithm of McCreight [M76]. Each processor takes $O(r \log m)$ time and does $O(r)$ work. In fact, there are $O(r)$ *time-steps* and each processor takes $O(1)$ time and does $O(1)$ work in each time step; in addition, after each time step, reorganizing the processors in a manner to be described takes $O(\log m)$ time and $O(\frac{m}{r})$ work. Thus, the total time taken is $O(r \log m)$ and the total work done is $O(m)$. This construction will be described in detail shortly.

Step 2. T_0' is post-processed to give T_0 in three steps, each of which takes $O(\log m)$ time and $O(m)$ work.

Step 2.1. T_0' is truncated as follows. All edges e in T_0' are considered in parallel. Suppose e is between a node x and its parent y such that $len(x) > r$ and $len(y) \leq r$. Then the edge e and the subtree rooted at x are removed; in addition, if $len(y) < r$ then a leaf x' is inserted with parent y such that $str(x')$ is the prefix of $str(x)$ of length r.

Step 2.2. Next, all subtrees containing only leaves x such that $str(x)$ is a suffix of $p_{k'}$, $k' \neq k$, are removed. We do not do the above operation for p_k, for the following technical reason: the symbol $\$_k$ can serve as $\$$, the last symbol of p (recall $p = s\$$).

Step 2.3. All chains of degree 2 are contracted. T_0 is the resulting tree.

The total time taken in Steps 1 and 2 is $O(r \log m)$ and the total work done is $O(m)$.

Step 1 Description. As stated earlier, the processor associated with p_i, $1 \leq i \leq k$, inserts all suffixes of p_i, except $\$_i$, in left to right order into T_0' using the sequential algorithm of McCreight [M76].

McCreight's Algorithm. For the sake of completeness, we briefly recapitulate McCreight's sequential algorithm for constructing the suffix tree T of a string s'. Let $p' = s'\$$. The algorithm inserts the various suffixes of p' in left to right order, i.e., in decreasing order of the lengths of the suffixes. In this process, the characters in p' are scanned from left to right in sequence and the nodes in the current suffix tree are scanned in some order. Three variables, i, j and x, are maintained at every step. i denotes the index of the rightmost character in p' which has been read. j denotes the index in p' at which the current suffix which the algorithm seeks to insert begins. x denotes the current node in T being scanned by the algorithm. This algorithm alternates between two phases, the *scanning phase* and the *rescanning phase*. Initially the algorithm starts in the scanning phase with $i = 1$, $j = 1$, and x being the root.

We describe a snapshot of the algorithm starting with a scanning phase and ending just before the next scanning phase.

In a scanning phase, the characters $p'[i], p'[i+1], \ldots$ are compared with the characters in the strings associated with the edges along the appropriate path starting at x until a mismatch occurs. Suppose a mismatch occurs when character $p'[i']$ is compared with a character in the substring associated with the edge e between node y and one of its children y'. Then the edge e is broken at the appropriate point and a node z is inserted as a child of y and parent of y'. A new leaf z' is inserted as a child of z; z' will be the leaf corresponding to the suffix j.

A rescanning phase begins now. Suffixes $j+1, j+2, \ldots$ are inserted one by one in this rescanning phase. Suffix $j+g$ is inserted as follows after suffix $j+g-1$ has been inserted, where $g \geq 1$. Suppose edge e_1 between node w and one of its children was broken in order to insert the suffix $j+g-1$; let f_1 be the node which was used to break edge e_1. Then the edge e_1' which has to be broken in order to insert suffix $j+g$ is found by traversing the appropriate path starting at node $suf(w)$ until a node w' is reached such that $len(w') \geq len(z) - g$. If $len(w') > len(z) - g$ then e_1' is the edge between w' and its parent; $suf(f_1)$ is then set to f_1', the node which is used to break e_1'; further, the

rescanning phase the continues with suffix $j+g+1$. If $len(w') = len(z) - g$ then the current rescanning phase comes to an end (without having inserted suffix $j+g$ yet) and the next scanning phase begins with $i = i'$, $j = j+g$ and $x = w'$; further, $suf(f_1)$ is set to w'.

An important fact to note is that in the rescanning phase, traversing an edge e takes constant time, while in the scanning phase, characters in the strings associated with e are compared in sequence and this takes time proportional to the number of comparisons. McCreight showed that the total time spent in the above rescanning phase is $O(h+i'-i)$, where h is the number of suffixes inserted in the rescanning phase; the entire algorithm can then be easily seen to take $O(|s'|)$ time.

Problems Encountered. Note that since all the $O(\frac{m}{r})$ processors which run McCreight's algorithm on the strings $p_1, \ldots, p_k$ work on a common data structure, these processors interfere with each other. The following two problems are encountered.

1. A number of processors may simultaneously attempt to break an edge e between node x and its child y and insert different nodes between these two nodes.

2. When a node y is inserted in T_0', the suffix link $suf(x)$ of the current parent node x of y may not have been determined. The analysis of McCreight's algorithm requires that this link be in place when y is inserted.

These problems are solved as follows.

Solution to Problem 2. After a node y is inserted as a child of x, the processor which inserted y *waits at* x until the suffix link of x is set before proceeding further. It needs to be shown that the total time spent by a processor waiting at various nodes is bounded by $O(r)$. This analysis will be described in Section 4.2.

Definition. We define the *current string length* of the processor associated with p_l, $1 \leq l \leq k$, at any instant as follows. If i is the index of the rightmost character in p_l scanned till that instant and j is the index at which the last suffix of p_l inserted begins then the current string length of the above processor is $i - j$.

The following fact holds for McCreight's algorithm.

Fact 1 *The current string length of the processor associated with* p_l *equals* $str(x) + 1$ *immediately before it inserts a node* x.

Solution to Problem 1. The run of the algorithm is divided into *time-steps*. As we will show in Section 4.2, there are $O(r)$ time-steps in all. At each time-step, each processor executes one step of McCreight's algorithm in $O(1)$ time. Following this the processors are organized into ordered lists in a manner to be described; this will take $O(\log m)$ time and $O(\frac{m}{r})$ work per time-step. Thus, the total time taken is $O(r \log m)$ and the total work done is $O(m)$.

At each time-step, two ordered lists are maintained for each node x in T_0'. The first list, l_1, contains those processors which either are at node x in the rescanning mode, or are in the scanning mode comparing characters in the strings associated with one

of the edges leading down from x. The second list, l_2, contains processors which are waiting at node x, i.e., waiting for the suffix link of node x to be set (see solution to Problem 2). If $suf(x)$ is defined then the list l_2 at x is empty. Each list is ordered by the current string lengths of the processors it contains. As we shall show, the only operations needed to update these lists at each time-step are to partition each list into contiguous sublists, divide each list into a constant number of non-contiguous sublists and to merge a constant number of lists into a single list. These operations can be accomplished in $O(\log m)$ time and $O(\frac{m}{r})$ work per time-step.

Consider a node x. At each *time-step* the following operations are performed, in addition to those that are routinely performed in McCreight's algorithm.

1. The list of processors in l_1 at node x is split into two ordered lists, one containing processors which seek to go down the edge whose associated substring begins with a 0 and the other containing processors which seek to go down the edge whose associated substring begins with a 1. Let L_0 be the former list and L_1 be the latter list. We concentrate our description on L_0. L_1 is processed similarly.

2. All processors in L_0 read in the pointer to the appropriate child y of x. Those processors in L_0 which are in the scanning phase compare their next character; those which are in the rescanning phase check whether the nodes they seek to insert are to be between x and y or not. Following this, an ordered sublist L' of L_0 comprising those processors which seek to break the edge e between x and y is obtained. Since processors in l_1 were ordered by their current string lengths, by Fact 1, the processors in L' appear in top to bottom order of the point at which they seek to break e. This also implies that all processors which seek to break e at the same point occur consecutively in L'.

3. L' is divided into sublists; each processor in a sublist seeks to break edge e at the same point. Let $x_1, \ldots, x_h$ be the distinct nodes, in order from top to bottom, which processors in L' seek to insert between x and y. Each sublist is further reorganized into groups of maximal size such that all processors in the same group have identical characters in the last scanned positions in their respective strings. Note that except for at most two groups (corresponding to last scanned characters 0,1, respectively) in each sublist, all other groups are singleton groups (because the $\$_i$'s are all mutually distinct).

Consider a particular x_i. Let α be the first processor in the sublist associated with x_i. α creates the node x_i and makes it a child of x_{i-1} if $i > 1$, and of x otherwise. In addition, if $i = h$, α makes y a child of x_h. Next, α sets $suf(z_i), suf(z_i')$ to x_i, where z_i, z_i' were the last nodes inserted by the processors in the sublist of L' associated with x_i (note that there are at most two such nodes). The first processor in each group in the sublist associated with x_i creates a new child of x_i; For the purpose of analysis, each such processor is said to have *inserted* x_i while all other processors in the above sublist are said to have *sought to insert* x_i.

4. For each x_i and newly created child v of x_i, let P denote the group of processors such that the first processor in P creates v. The first processor in P either continues its rescanning phase or starts a new rescanning phase. Consider processor α which is in P but not the first processor in p. If α is in the rescanning phase, then it begins a new scanning phase from x_i at the next time-step (In McCreight's algorithm, when the

node α seeks to insert already exists, α begins a new scanning phase from that node. The above situation is similar). Suppose α was already in the scanning phase. Further, suppose α is associated with p_l, $1 \leq l \leq k$. α sought to insert node x_i after performing an unsuccessful comparison involving $p_l[len(x_i)+1]$. α then continues its scanning phase in the next time-step by comparing $p_l[len(x_i)+2]$ with the second character in the string associated with the edge between x_i and v; note that $p_l[len(x_i)+1]$ is guaranteed to match the first character in this string.

5. The new lists l_1 and l_2 are obtained for the nodes $x, x_1, \ldots, x_h$ as follows. Let z, z' be the two nodes, if any, such that $suf(z) = x$ and $suf(z') = x$ (at least one of z, z' exists after the first suffix of each p_l, $1 \leq l \leq k$, has been inserted). Let y' be the parent of x at the beginning of the current time-step. Note that y' may no longer be the parent of x. The new list l_1 at x comprises processors derived from the old lists l_1 at x, y', z and z' and the old lists l_2 at z, z'. The new list l_1 at x_i, $1 \leq i \leq h$, comprises processors derived from the old lists l_1 at x, z_i and z_i', and the old lists l_2 at z_i and z_i' (z_i, z_i' were defined in Step 3). If $suf(x)$ is defined then the new list l_2 at x is empty; otherwise, the new list l_2 at x is obtained by inserting into the old list l_2 at x those processors in the sublist associated with x_1 which are the first processors in their respective groups. The new list l_2 at x_h is empty. The new list l_2 at x_i, $1 \leq i < h$, contains those processors in the sublist associated with x_{i+1} which are the first processors in their respective groups. Note that these processors are now in the rescanning phase.

Note that the processors in the old list l_1 at x contribute only to the new lists l_1 at $x, x_1, \ldots, x_h, y, suf(x)$. The old list l_2 at x either remains unchanged, or has some processors from the old list l_1 at x inserted into it, or becomes part of the new list l_1 at $suf(x)$. Further, recall that those processors in the old list l_1 at x which move to the the new lists l_1 at $x_1, \ldots, x_h$ appear in order in the old list l_1 at x, i.e., a processor which moves to the new list l_1 at x_j appears after a processor which moves to the new list l_1 at $x_{j'}$ if $j > j'$. In addition, note that the current string length of each processor in some old list l_1 can change by at most one and the current string length of each processor in some old list l_2 remains unchanged. It can easily be checked that all new lists can be derived from old lists by partitioning each old list into a number of contiguous sublists, dividing each old list into a constant number of non-contiguous sublists, and merging together a constant number of the lists. Therefore the new lists can easily be obtained in $O(\log m)$ time and $O(\frac{m}{r})$ work.

The following fact is noteworthy; it follows from point 4 above.

Fact 2 *Consider one scanning phase of a processor α which processes p_l, $1 \leq l \leq k$. Suppose this scanning phase begins with a comparison at $p_l[i]$ and ends with an unsuccessful comparison at $p_l[j]$. Then each character in $p_l[i \ldots j]$ is compared exactly once in this scanning phase, one character in each time step.*

Remark. We remark that an array storing, for each index j, $1 \leq j \leq m$, the leaf x of T_0 such that $p[j \ldots j+r-1] = str(x)$ can be obtained in the process of constructing T_0 without any time or work overhead. In addition, for each leaf x of T_0, a list of indices j

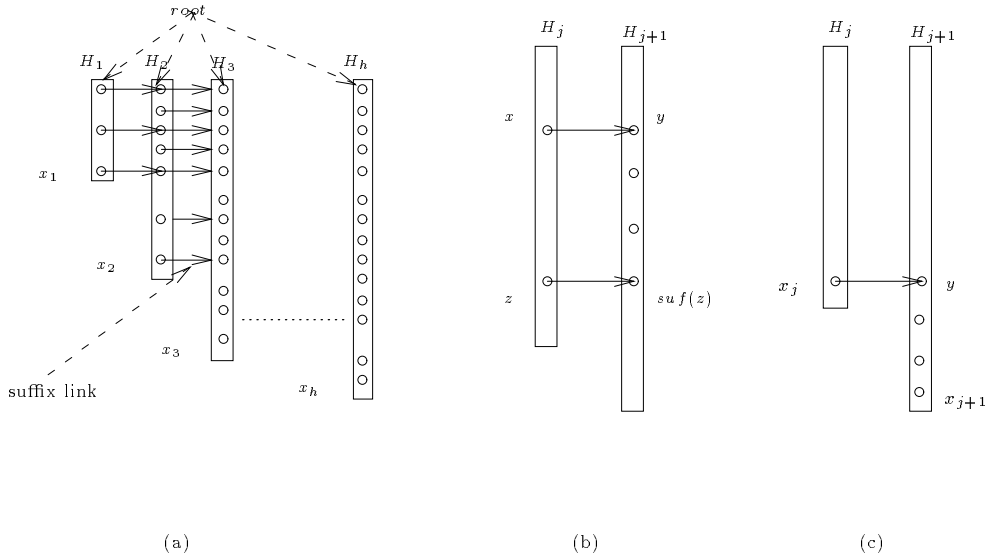

Figure 3: The sequences H_i.

such that $p[j \ldots j+r-1] = str(x)$ can also be obtained in the process of constructing T_0.

4.2 The Analysis

Since processors spend time waiting in the l_2 lists at various nodes in T_0', it is not obvious that the total number of time-steps is $O(r)$. We show that this is indeed the case.

Definitions. $p(x)$ is defined to be the parent of x when x is inserted. Let $root$ denote the root node. $p(root)$ is defined to be $root$. For a node x in T_0', let at_x denote the time-step in which node x is inserted in T_0'. Consider a particular processor α which is associated with some piece of s. Let $\rho_1, \rho_2, \ldots, \rho_h$, $h \leq 2r$, be the sequence of leaves inserted by α and let $x_i = p(\rho_i)$. $\rho_1, \ldots, \rho_h$ correspond to the suffixes (in decreasing length order, respectively) of the piece of s processed by α.

The Node Sequences H_i. We define node sequences H_i, for $1 \leq i \leq h$. The first node in each sequence is $root$. The last node in sequence H_i will be x_i. See Fig.3a. For each node x in sequence H_i, we will define a time-step $t_{x,i}$. $t_{root,i}$ is defined to be 1 for all i. If x follows y in H_i then $t_{x,i}$ will be greater than $t_{y,i}$ and x will be a strict descendant of y in T_0'. As we will show, the above definitions will guarantee that node x is in T_0' by time-step $t_{x,i}$, i.e., $at_x \leq t_{x,i}$.

Sequence H_1. The first node in H_1 is the root node $root$ and $t_{root,1} = 1$. Consider the sequence S of nodes finally on the path from $root$ to x_1 in that order, with both endpoints excluded. The first node z in S such that $at_z \leq t_{root,1} + 1$ is added to H_1 and $t_{z,1}$ is defined to be $t_{root,1} + 1$. Next, the first node z' in S following z such that $at_{z'} \leq t_{root,1} + 2$

is added to H_1 and $t_{z',1}$ is defined to be $t_{root,1} + 2$. This process is continued until no further nodes in S can be added to H_1. Then x_1 is added to H_1. $t_{x_1,1}$ is defined to be $t_{root,1} + len(x_1) - len(root) + 1 = len(x_1) + 2$. Note that if a processor traverses the path from *root* downwards towards x_1 starting at time-step 1, then the only nodes it can encounter are the nodes added to H_1 above.

The Remaining Sequences. Sequence H_{j+1}, $j \geq 1$, is defined next, assuming that H_j has already been defined. H_{j+1} will contain as a subsequence (not necessarily contiguous) the nodes $suf(x)$, where x is in H_j. Suppose H_{j+1} has been defined up to $suf(x)$ for some x in H_j. Let $y = suf(x)$. Let z be the node following x in H_j, if any.

There are two cases to consider. If z is defined then we describe how to augment H_{j+1} until $suf(z)$ has been included. If z is not defined then we describe how the rest of the sequence H_{j+1} is constructed.

First, suppose $x \neq x_j$, i.e., z is defined. See Fig.3b. Let $t = \max\{t_{y,j+1}, t_{z,j}\}$. Consider the sequence S of nodes finally on the path from y to $suf(z)$ in that order, with y and $suf(z)$ excluded. The first node z' in S such that $at_{z'} \leq t + 1$ is added to H_{j+1} and $t_{z',j+1}$ is defined to be $t + 1$. Next, the first node z'' in S following z' such that $at_{z''} \leq t + 2$ is added to H_{j+1} and $t_{z'',j+1}$ is defined to be $t + 2$. This process is continued until no further nodes in S can be added to H_{j+1}. Then $suf(z)$ is added to H_{j+1}. If S is non-empty then $t_{suf(z),j+1}$ is defined to be $t_{w,j+1} + 1$, where $w \in S$ is the node preceding $suf(z)$ in H_{j+1}. If S is empty then $t_{suf(z),j+1} = t + 1$. Note that if a processor traverses the path from y downwards towards $suf(z)$ starting at time-step t, then the only nodes it can encounter are the nodes added to H_{j+1} above.

Next, suppose z is not defined, i.e., $x = x_j$. If $y = x_{j+1}$ then H_{j+1} is fully defined. So suppose $y \neq x_{j+1}$. See Fig.3c. Then α begins a new scanning phase from y. In this case, the construction of H_{j+1} is similar to that of H_1. Let $t = t_{y,j+1}$. Consider the sequence S of nodes finally on the path from y to x_{j+1} in that order, with both endpoints excluded. The first node z' in S such that $at_{z'} \leq t + 1$ is added to H_{j+1} and $t_{z',j+1}$ is defined to be $t + 1$. Next, the first node z'' in S following z' such that $at_{z''} \leq t + 2$ is added to H_{j+1} and $t_{z'',j+1}$ is defined to be $t + 2$. This process is continued until no further nodes in S can be added to H_{j+1}. Then x_{j+1} is added to H_{j+1} and $t_{x_{j+1},j+1}$ is defined to be $t_{y,j+1} + len(x_{j+1}) - len(y) + 1$. Again, note that if a processor traverses the path from y downwards towards $suf(z)$ starting at time-step t, then the only nodes it can encounter are the nodes added to H_{j+1} above.

Analyzing the Sequences. The following key lemma holds for the above defined sequences. The proof of the lemma is given in the Appendix 1. Here we only sketch the intuition.

Lemma 4.1 *If $x \in H_i$, $1 \leq i \leq h$, then $at_x \leq t_{x,i}$.*

Intuition. The intuition behind this key lemma is the following. First, note that if x does not have a suffix link into it from a node in H_{i-1} and if $x \neq x_i$, the lemma is true by definition. We give the intuition here as to why the lemma is true for nodes which have suffix links from nodes in the previous sequence.

Let z_0, z_f be two consecutive nodes in sequence H_i. Assume that $at_{z_f} \leq t_{z_f,i}$ and $at_{suf(z_0)} \leq t_{suf(z_0),i+1}$. We show that $at_{suf(z_f)} < t_{suf(z_f),i+1}$.

We illustrate the intuition with the easier case, i.e., when $z_0 = p(z_f)$. Then the processor which inserts z_f will wait at z_0 at most until time-step $\max\{at_{z_f}, at_{suf(z_0)}\} \leq \max\{t_{z_f,i}, t_{suf(z_0),i+1}\}$. After this time-step, the above processor starts its search for $suf(z_f)$. By the way H_{i+1} is defined, the only nodes that the processor can encounter in this process are those in H_{i+1} strictly between $suf(z_0)$ and $suf(z_f)$; let δ_f be the number of such nodes. Therefore, by time-step $\max\{t_{z_f,i}, t_{suf(z_0),i+1} + \delta_f + 1\} \leq t_{suf(z_f),i+1}$, the above processor would either have located $suf(z_f)$, if it already exists, or inserted $suf(z_f)$, otherwise.

The harder case is when $z_0 \neq p(z_f)$. Suppose there exist nodes $z_1, \ldots, z_{f-1}$ such that $p(z_f) = z_{f-1}$, $p(z_{f-1}) = z_{f-2}$ and so on until $p(z_1) = z_0$. This is really the bad case because the processor that inserted z_f could be waiting at z_{f-1}, the processor that inserted z_{f-1} could be waiting at z_{f-2} and so on. However, the desired result can again be obtained by just repeating the argument of the previous paragraph with z_0, z_1, then z_1, z_2, and so on until z_{f-1}, z_f, and combining the results of each of these arguments.

We use Lemma 4.1 as follows to complete the analysis. The remaining portion is akin in spirit to the analysis of McCreight's algorithm.

Definitions. Let π be a sequence comprising nodes derived from the sequences $H_1, \ldots, H_h$, defined as follows. Associated with each node x in π is a value $seq(x)$ which is i if x is derived from H_i. π begins at x_h and ends at *root*. See Fig.4a. $seq(x_h) = h$ and the seq values of the nodes in π are non-increasing. If x is a node in π, $seq(x) = i$, $x \neq root$, then the node y in π which follows x is determined as follows. If $i = 1$ then y is the node preceding x in H_1 and $seq(y) = 1$. Suppose $i > 1$. If there is no node $x' \in H_{i-1}$ such that $suf(x') = x$ (i.e., $x = x_i$), then y is the node preceding x in H_i and $seq(y) = i$. Otherwise, suppose that there is a node $x' \in H_{i-1}$ such that $suf(x') = x$. Let z' be the node which immediately precedes x' in H_{i-1} and let $z = suf(z')$; Then if $t_{z,i} > t_{x',i-1}$, $y = z$, otherwise, $y = x'$. In the former case $seq(y) = i$ and in the latter case $seq(y) = i - 1$.

π is divided into maximal subsequences such that for all consecutive x, y's in a particular subsequence, $seq(y) = seq(x) - 1$, i.e., $suf(y) = x$. Note that the last subsequence is a singleton subsequence containing only *root* as $suf(root) = root$.

If x, y are nodes in some H_i, $i > 1$, y preceding x, then $\delta_i(x, y)$ is defined to be the number of nodes in H_i between y and x.

Lemma 4.2 *Let $y_1, y_2, \ldots, y_j$ be one of the maximal subsequences of π defined above and suppose $seq(y_1) = e$, i.e., $seq(y_2) = e - 1, \ldots, seq(y_j) = e - j + 1$ (see Fig.4b). Let z be the node which follows y_j in π, i.e., $seq(z) = e - j + 1$. If $j = 1$ then $t_{y_1,e} \leq t_{z,e} + 2(len(y_1) - len(z))$. If $j > 1$ then $t_{y_1,e} \leq t_{z,e-j+1} + 2(len(y_1) - len(z)) + 4j - 3$.*

Proof. First, we show that when $t_{y_f,e-f+1} \leq t_{z,e-j+1} + 2(len(y_j) - len(z)) + 2(j - f + 1) - (len(y_f) - len(y'_f)) - 1$, for $f = j \ldots 1$, where y'_f is the node preceding y_f in H_{e-f+1}. This claim is shown by induction on f, $f = j \ldots 1$, in the subsequent paragraphs. The lemma follows when $j = 1$ because then $y'_1 = z$ and $len(y_1) - len(z) > 0$ and therefore,

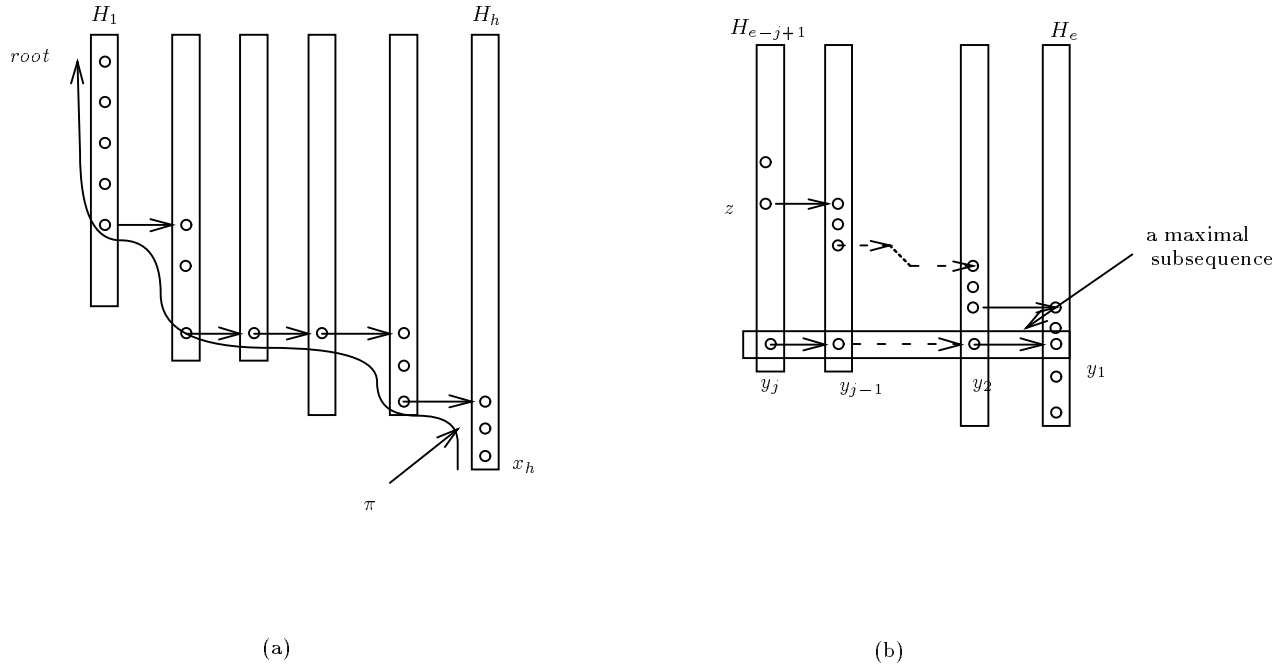

Figure 4: The path π and a maximal subsequence.

$t_{y_1,e} \leq t_{z,e} + (len(y_1) - len(z)) + 1 \leq t_{z,e} + 2(len(y_1) - len(z))$; for $j > 1$, the lemma follows because $t_{y_1,e} \leq t_{z,e-j+1} + 2(len(y_j) - len(z)) + 2j - (len(y_1) - len(y'_1)) - 1 \leq t_{z,e-j+1} + 2(len(y_j) - len(z)) + 2j - 1 \leq t_{z,e-j+1} + 2(len(y_1) + j - 1 - len(z)) + 2j - 1 \leq t_{z,e-j+1} + 2(len(y_1) - len(z)) + 4j - 3$.

As the base case, consider $f = j$. Then $y'_j = z$. If $y_j = x_{e-j+1}$ then $t_{y_j,e-j+1} \leq t_{z,e-j+1} + (len(y_j) - len(z)) + 1$. If $y_j \neq x_{e-j+1}$ then by the construction of π, $t_{y_j,e-j+1} = t_{z,e-j+1} + \delta_{e-j+1}(y_j, z) + 1 \leq t_{z,e-j+1} + (len(y_j) - len(z)) + 1$.

Next, assume that $t_{y_f,e-f+1} \leq t_{z,e-j+1} + 2(len(y_j) - len(z)) + 2(j - f + 1) - (len(y_f) - len(y'_f)) - 1$, $f > 1$, and consider $t_{y_{f-1},e-f+2}$. $t_{y_{f-1},e-f+2} = t_{y_f,e-f+1} + \delta_{e-f+2}(y_{f-1}, suf(y'_f)) + 1 \leq t_{y_f,e-f+1} + (len(y'_{f-1}) - len(suf(y'_f))) + 1 \leq t_{z,e-j+1} + 2(len(y_j) - len(z)) + 2(j - f + 1) - (len(y_f) - len(y'_f)) - 1 + (len(y'_{f-1}) - len(suf(y'_f))) + 1$. Since $len(y_{f-1}) - len(y'_{f-1}) = (len(y_f) - len(y'_f)) - (len(y'_{f-1}) - len(suf(y'_f)))$, we get $t_{y_{f-1},e-f+2} \leq t_{z,e-j+1} + 2(len(y_j) - len(z)) + 2(j - (f-1) + 1) - (len(y_{f-1}) - len(y'_{f-1}))) - 1$, as claimed. □

Lemma 4.3 $t_{x_h,h} \leq 14r$.

Proof. Let u_i, v_i be the extreme nodes in the ith subsequence and let $size(i)$ be the number of nodes in the ith subsequence. Let f be the number of subsequences. $u_f, v_f = root$ and $u_1 = x_h$.

The sum $\Sigma[4size(i) - 3]$ over all non-singleton subsequences is clearly $5h$. Then, by Lemma 4.2, $t_{x_h,h} - t_{root,seq(root)} \leq \Sigma_{1 \leq i < f}[2(len(u_i) - len(u_{i+1}))] + 5h \leq 2len(x_h) + 5h \leq 4r + 10r = 14r$. □

Corollary 4.4 *The total number of time-steps taken by processor α is at most $14r$.*

Theorem 4.5 *There exists an algorithm which constructs the r-ST of s in $O(r \log m)$ time and $O(m)$ work.*

5 Completing the Suffix Tree

Given an r-suffix tree T_0 of s, we show how to compute the complete suffix tree ST of s. This is done in $\log m - \log r$ iterations. In the ith iteration, the $2^i r$-ST of s is obtained.

Definitions. Let T_i denote the $2^i r$-ST of s. Let $leaf_i(j)$ be the leaf in T_i such that there is an occurrence of $str(leaf_i(j))$ beginning at index j in p. For any leaf $l \in T_i$, let $indices_i(l)$ be the set of indices j such that there is an instance of $str(l)$ beginning at j, i.e., $indices_i(l) = \{j | leaf_i(j) = l\}$.

A Naive Algorithm. First, we describe a naive algorithm which computes ST in $O(m \log^2 m)$ work.

At the end of iteration $i-1$, for each index j, $1 \leq j \leq m$, we keep track of $leaf_{i-1}(j)$. Consider the ith iteration.

T_{i-1} is preprocessed in $O(m)$ work and $O(\log m)$ time for order queries on leaves (see Section 2). Following this, the relative order and longest common prefix of any two substrings of p of length at most $2^i r$ can be determined in constant time and work.

Next, all leaves l of T_{i-1} such that $str(l)$ is not a suffix of p are processed in parallel. Consider one such leaf l. The substrings of p of length $2^i r$ beginning at indices in $indices_{i-1}(l)$ are sorted; subsequently, one representative is selected from each equivalence class of identical substrings. Each representative substring is said to *represent* all the substrings in its equivalence class. Let $L_i(l)$ denote the ordered list of these representative substrings. The trie induced by the strings in $L_i(l)$ is constructed using the induced trie construction algorithm mentioned in Section 2 and the root of this trie is merged with l (note that following this merger, l may no longer be a node in T_i).

Computing $L_i(l)$ takes $O(\log m)$ time and $O(m \log m)$ work over all leaves l of T_{i-1}. Computing the trie induced by the strings in $L_i(l)$ takes $O(\log m)$ time and $O(m)$ work, over all leaves l of T_{i-1}. The overall algorithm is made to run in $O(\log^2 m)$ time and $O(m \log^2 m)$ work by starting with $r = 1$ and performing $O(\log m)$ of the above iterations.

Reducing the Work. There are two main components in the above algorithm which lead to superlinear work. The first is obtaining the ordered list $L_i(l)$ of representative substrings. The second involves preprocessing $T_0, T_1, \ldots$ for order queries.

In our scheme too, we construct, for each leaf l of T_{i-1}, the trie induced by the strings in $L_i(l)$ in iteration i. This is done as before by first obtaining $L_i(l)$ for each leaf l of T_{i-1} and then using the induced trie construction algorithm. In order to restrict the total work done to $O(m)$, we use the fact that ST has a highly repetitive structure. Exploiting this repetitive structure, we show how to obtain the ordered lists $L_i(l)$ for all leaves l of T_{i-1}, by sorting and preprocessing (for order queries) only $O(\frac{m}{\log^2 m})$ items rather than up to m items. In the process, we make critical use of the property defined in Corollary

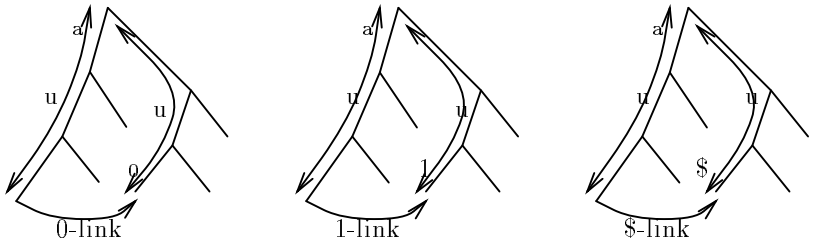

Figure 5: 1-,0-, and $-links ($u$ is a string and $a = 0/1$).

3.3 and also of periodicity properties of strings. The overall algorithm then takes linear work.

The New Scheme. T_0 is constructed with $r = 2\lceil log^3 m \rceil$ in $O(\log^4 m)$ time and $O(m)$ work. Note that by Corollary 3.3, $n_r = O(\frac{m}{\log^2 m})$. Next, $\log m$ iterations are performed, each iteration taking $O(\log^3 m)$ time. In iteration i, the ordered list $L_i(l)$ is computed for all leaves l of T_{i-1} in two steps. First, $L_i(l)$ is obtained for a subset of the leaves l of T_{i-1}; these leaves are called *sources*. Next, $L_i(l)$ is obtained for all leaves of T_{i-1} using the computation in the first step and the repetitive nature of ST. ST is repetitive in the following sense. If nodes x, y are such that $y = suf(x)$ and there are no other suffix links pointing to y then the subtree of ST rooted at y is "identical" to that at x. If nodes x, y, z are such that $y = suf(x)$ and $y = suf(z)$ then the subtree of ST rooted at y is a "merger" of the subtrees rooted at x and z.

The issues which need to be addressed next are how sources are chosen, how $L_i(l)$ is obtained efficiently for sources l, and how $L_i(l)$ is obtained for all remaining leaves l. We discuss each of the above three issues in turn. First, we need some definitions and primitives.

Definitions. Let $\mathcal{H}_{i-1}$ be the set of leaves of T_{i-1}. We define *0-links*, *1-links*, and *$-links* as follows (see Fig.5). The *0-link* of $l \in \mathcal{H}_{i-1}$ points to the leaf l' in $\mathcal{H}_{i-1}$, if any, such that $str(l)$ and $str(l')$ are consecutive substrings of p and $str(l')$ ends in 0. The *1-link* of $l \in \mathcal{H}_{i-1}$ points to the leaf l' in $\mathcal{H}_{i-1}$, if any, such that $str(l)$ and $str(l')$ are consecutive substrings of p and $str(l')$ ends in 1. The *$-link* of $l \in \mathcal{H}_{i-1}$ points to the leaf l' in $\mathcal{H}_{i-1}$, if any, such that $str(l)$ and $str(l')$ are consecutive substrings of p and $str(l')$ ends in \$. *0-links*, *1-links*, and *$-links* are collectively called *next-links*. Let J_{i-1} denote the digraph whose vertices are the leaves in $\mathcal{H}_{i-1}$ and whose edges are the next-links among these leaves.

In Section 6, we consider the digraph $G = J_0 = (\mathcal{H}_0, E)$, where E is the set of next-links between vertices in $\mathcal{H}_0$, and show how to choose $O(\frac{m \log m}{r}) = O(\frac{m}{\log^2 m})$ leaves of T_0, called *origin leaves*. In iteration i, source leaves will be determined using these origin leaves.

In Section 7, we show how to obtain $L_i(l)$ for source leaves in iteration i. In Section 8 we show how to obtain $L_i(l)$ for the rest of the leaves in iteration i. In both sections, for each leaf l of T_{i-1}, we show how a list $L'_i(l)$ with the following description is obtained

first. $L'_i(l)$ is an ordered list of one representative string from each equivalence class of the set of substrings of p of length $k.2^{i-1}r$ which begin at indices in $indices_{i-1}(l)$, for some k, $2 \leq k \leq 3$. The value of k is different for different leaves l. In particular, for source leaves, $k = 3$. Note that if $k = 2$ for leaf l, $L'_i(l) = L_i(l)$. Clearly, if the longest common prefix of adjacent strings in $L'_i(l)$ is known, which will indeed be the case, $L_i(l)$ can easily be obtained from $L'_i(l)$ in constant time and $O(|L'_i(l)|)$ work. Henceforth, we refer to $L'_i(l)$ as *the list at* l.

In Section 9, we show how some data structures defined in Sections 7 and 8 are maintained.

6 Choosing Origin Leaves

Recall from the remark at the end of Section 4.1 that an array storing $leaf_0(j)$, for each index j, $1 \leq j \leq m$, was obtained while constructing T_0. The next-links for the set of leaves $\mathcal{H}_0$ in T_0 can be set up easily from the above information and therefore, G can be obtained easily. Equivalently, G can be obtained during the construction of T_0 itself.

To select origin leaves, we remove edges from G until each connected component in the resulting graph is a rooted tree of $O(\log^2 m)$ height. There will be $O(\frac{m}{\log^2 m})$ such trees. The roots of each such tree are chosen as origin leaves.

In iteration i, a similar (but implicit) removal of edges from graph J_{i-1} will be performed. These removals will result in connected components which are rooted trees of $O(\log^2 m)$ height; the roots of these trees will be the sources in iteration i.

Before describing how origin leaves are determined, we need the following definitions and lemmas.

Definitions. Consider an index j, $2 \leq j \leq m$. Let $u = p[j \ldots j + r - 1]$, $v = p[j-1]u$ and let v' equal v with the first character complemented. j is called a *β-index* if both the following conditions hold.

1. v and v' occur at least once each in p.
2. Either v occurs fewer times than v' in p, or v occurs as many times as v' and $p[j-1] = 0$.

For technical reasons, index 1 is also defined to be a β-index. The leaf $leaf_0(j)$ is called a β-node in G if j is a β-index. The leaf $leaf_{i-1}(j)$ is called a β-node in J_{i-1} if j is a β-index.

Recall from the remark in Section 4.1 that for each leaf x of T_0, a list of indices j such that $p[j \ldots j + r - 1] = str(x)$ was obtained in the process of constructing T_0. From this information and using the next-links among the leaves of T_0, all β-indices and β-nodes in G can be determined easily using the following lemma.

Lemma 6.1 *With the exception of $leaf_0(1)$, a node in G is a β-node only if it has in-degree 2 in G. Further, any node in J_h which has in-degree 2 is a β-node, where $0 \leq h \leq \log m$. In addition, if there are edges from nodes l', l'' to l in J_h, $0 \leq h \leq \log m$,*

then either all $j > 1$ such that $leaf_0(j-1) = l'$ or all $j > 1$ such that $leaf_0(j-1) = l''$ are β-indices.

Proof. For the first part of the lemma, consider a node l in G, $l \neq leaf_0(1)$. Suppose that l is a β-node. Then both $0str(l)$ and $1str(l)$ are substrings of p. It follows that there exist nodes l', l'' such that there are edges from l', l'' to l in G and $str(l')$ and $str(l'')$ are prefixes of $0str(l)$ and $1str(l)$, respectively, of length r. Therefore, l has in-degree 2 in G.

For the second part of the lemma, consider a node l in J_h and suppose l has in-degree 2 in J_h. Let l', l'' be the nodes from which there are edges to l in G. Then there exists indices $j, j' > 1$ such that $leaf_h(j) = leaf_h(j') = l$, $p[j \ldots j+r-1] = p[j' \ldots j'+r-1]$ is a prefix of $str(l)$, $p[j-1 \ldots j+r-2]$ is a prefix of $str(l')$, $p[j'-1 \ldots j'+r-2]$ is a prefix of $str(l'')$, $p[j-1] = 0$ and $p[j'-1] = 1$. It follows that either j or j' is a β-index and therefore, l is a β-node.

The third part of the lemma is true by definition. □

Lemma 6.2 *The number of β-indices and β-nodes in G is $O(\frac{m \log m}{r}) = O(\frac{m}{\log^2 m})$.*

Proof. The number of β-indices $j > 1$ is clearly bounded by $\Sigma \min\{i_{0w}, i_{1w}\}$, where the summation is over all binary strings w of length r (see Section 3). The lemma then follows from Corollary 3.3 applied to string s. □

Edge removals from G are performed in three steps, Step 1–3.

Step 1. A graph G_1 is obtained from G in this step. For all β-indices $j > 1$, the edge from $leaf_0(j-1)$ to $leaf_0(j)$ is removed. Step 1 takes $O(m)$ work and $O(1)$ time. Note that by Lemma 6.1, $leaf_0(j)$ has in-degree 2 in G and therefore, in-degree 1 in G_1. Let $E_1 \subset E$ denote the set of edges removed in Step 1. Note that all edges in E_1 are incident upon β-nodes. Then $G_1 = (\mathcal{H}_0, E - E_1)$.

Definition. A *connected component* of a digraph is any maximal subgraph containing at least one vertex x from which there is a path to all other vertices in that subgraph; x is called the *root* of the connected component. Two nodes in a digraph are *adjacent* if there is an edge from one to the other. Two nodes in a digraph are distance j apart if the shortest path from one to the other contains exactly j edges.

The following lemma holds now (see Fig.6).

Lemma 6.3 *Every connected component of G_1 has at least one root. Each such connected component has exactly one cycle unless the root is unique and equals $leaf_0(1)$, in which case it has no cycles. If such a cycle exists then it comprises all the roots of the component and contains at least one β-node.*

Proof. Let x be the leaf in T_0 such that $str(x)$ is a prefix of p. Clearly there is a path in G from x to every other vertex in G. Each vertex in G, except possibly x, has in-degree at least 1 and at most 2. Each vertex in G_1, except possibly x, has in-degree exactly 1. This implies the following.

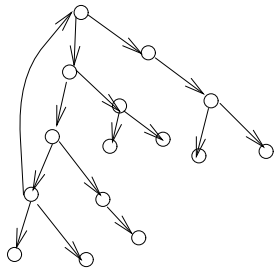

Figure 6: Connected components of G_1.

If a connected component of G_1 does not have any cycles then the root is unique and must be x, which is a β-node. If a connected component CC has a cycle C then C is the only cycle in CC; further, all nodes in C are roots and no node outside C can be a root of CC. If x is in C then the lemma follows. If x is not in C, some node y in C must have an edge in E_1 incident upon it as there is a path from x to every node in C in G. From Lemma 6.1, y is a β-node. $\Box$

Step 2. One edge is removed from each cycle in G_1 to give a graph G_2. This requires finding the cycles first which is done as described below in $O(m)$ work and $O(\log^2 m)$ time. Let $E_2 \subset E - E_1$ denote the set of edges removed in Step 2.

Finding Cycles in G_1. Note that no optimal algorithm is known for finding the biconnected components of a graph. Instead, using the fact that the number of nodes with in-degree 2 in G_1 is small, we obtain a new graph G' of size $O(\frac{m \log m}{r}) = O(\frac{m}{\log^2 m})$ from G_1; each cycle of G' corresponds to a unique cycle in G_1. We then run the biconnected components finding algorithm [TV84] on G' in order to find the only cycle, if any, in each connected component of G', in $O(\log^2 m)$ time and $O(m)$ work.

G' is obtained as follows. First the node $leaf_0(1)$ and all those nodes in G_1 which have in-degree or out-degree more than 1 in G are removed from G_1. By Lemmas 6.3 and 6.1, at least one node is removed from every cycle in G_1. Clearly, each connected component of the resulting graph is a simple chain of nodes. These chains are identified easily in $O(\log m)$ time and $O(m)$ work. G' is just the graph G_1 with each such chain condensed into one node. G' is obtained easily in $O(\log m)$ time and $O(m)$ work.

Lemma 6.4 shows that G' has $O(\frac{m \log m}{r}) = O(\frac{m}{\log^2 m})$ vertices. Since G' has degree at most 2, it has $O(\frac{m}{\log^2 m})$ edges, as required. Further, since $leaf_0(1)$ and all those nodes in G_1 which have in-degree 2 in G are also in G', it follows from Lemmas 6.3 and 6.1 that each cycle in G_1 corresponds to a unique cycle in G'.

Lemma 6.4 *G' has $O(\frac{m \log m}{r} + 1) = O(\frac{m}{\log^2 m})$ nodes.*

Proof. It follows from Lemma 6.3 that each connected component of G' is a tree with at most one extra edge. Let the term *chain node* denote a node in G' which represents a condensed chain of vertices in G_1. Clearly, two chain nodes cannot be adjacent in G'.

We bound the number of nodes in G' which have out-degree 0, 1, 2 and 3 in G' individually. There is at most 1 node which could have out-degree 3 in G and hence in G', specifically, the vertex x such that $str(x)$ is a suffix of s (recall $p = s\$$). To see this, note that nodes x such that $str(x)$ is a suffix of p have only \$-links going out from them. Further, nodes x such that $str(x)$ is neither a suffix of s or p do not have an outgoing \$-link.

First, we show that the number of nodes with out-degree 0 in G' is $O(\frac{m \log m}{r}) = O(\frac{m}{\log^2 m})$. Note that with the exception of at most 1 node (the node corresponding to the suffix of $p = s\$$ of length 2, recall that the suffix \$ of p is excluded from T_0, by definition), each node has out-degree at least 1 in G. It follows from Step 1 that, but for 1 node with out-degree 0 in G', each node in G' with out-degree 0 in G' is adjacent in G to a β-node. By Lemma 6.2, the number of nodes with out-degree 0 in G' is $O(\frac{m \log m}{r} + 1) = O(\frac{m}{\log^2 m})$.

Since each connected component of G' is a tree with at most one extra edge, the number of nodes with out-degree 2 in G' is proportional to the number of nodes with out-degree 0 in G'.

It remains to bound the number of nodes x with out-degree 1 in G'. Let z be a node in G, if any, which has out-degree 3 in G. Recall from above that there is at most one such node. There are only $O(1)$ nodes x such that either $x = z$ or x is adjacent to z in G. In addition, the number of nodes x which are at most distance 2 in G from a node y with in-degree 2 in G is $O(\frac{m \log m}{r}) = O(\frac{m}{\log^2 m})$; this follows from Lemmas 6.1, 6.2 and the fact that G has constant degree. Therefore, we assume that $x \neq z$, i.e., x has out-degree 1 or 2 in G, x is not adjacent to z in G, and that neither x nor any of the nodes within distance 2 from x in G has in-degree 2 in G. We now show that such an x must be adjacent in G' to a node with out-degree 2 in G'. The lemma then follows from the fact that G' is a constant degree graph.

First, suppose x is not a chain-node, i.e., it has either in-degree or out-degree more than 1 in G. From the assumptions above, it must have out-degree 2 in G. Recall that x has out-degree 1 in G'. Therefore one of the edges exiting x in G was removed in Step 1, i.e., x is adjacent in G to a β-node, a contradiction. Second, suppose x is a chain-node. Let y be the node to which there is an edge from x in G'. Since x has out-degree 1 in G', y exists. Since y is not a chain node, it has either in-degree or out-degree more than 1 in G. From the assumptions above, y must have out-degree 2 in G. If y has out-degree 2 in G' as well then we are done. Otherwise, one of the edges exiting y in G was removed in Step 1, i.e., y is adjacent in G to a β-node and x is distance at most 2 in G from a β-node, a contradiction. □

It follows from Lemma 6.3 that the connected components of G_2 are rooted trees. The distance of each node in each tree from its respective root is found in $O(\log m)$ time and $O(m)$ work.

Step 3. A graph G_3 is obtained from G_2 by removing the sole edge incident into each node x in G_2 which is distance a multiple of $\lceil \log^2 m \rceil$ from the root of the tree forming its connected component in G_2. Let $E_3 \subset E - E_1 - E_2$ denote the set of edges removed in Step 3.

Note that $G_3 = (\mathcal{H}_0, E - E_1 - E_2 - E_3)$. Clearly, the connected components of G_3

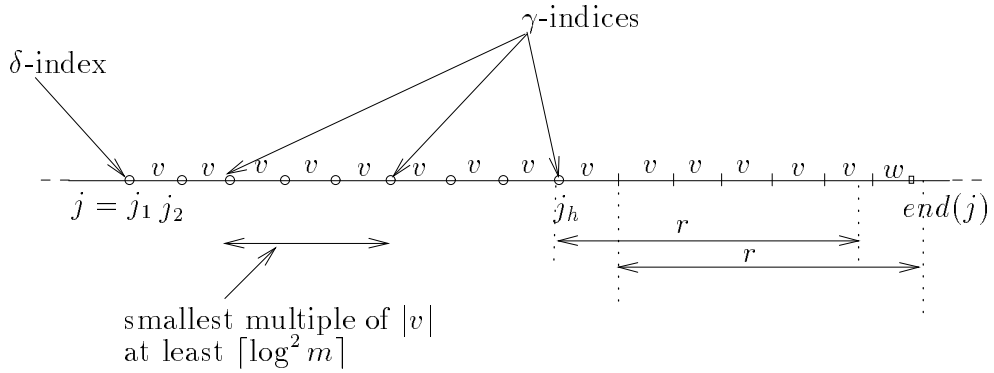

Figure 7: The family $F(j)$, $|v| = per(j)$, w a prefix of v.

are rooted trees with $O(\log^2 m)$ height. The roots of these trees are chosen to be the required origin leaves.

The following fact will be used repeatedly later.

Fact 3 *All edges in E_1 are incident upon β-nodes in G. Edges in E_2, E_3 are incident upon nodes in G which are origin leaves, i.e., which are roots of the connected components of G_3.*

6.1 Properties of Origin Leaf Indices

Next, the origin leaves chosen above will be divided into two categories. The sum of the sizes of the sets $indices_0(l)$ over all origin leaves in the first category will be $O(\frac{m}{\log^2 m})$. For origin leaves l in the second category, $str(l)$ will be periodic with period $O(\log^2 m)$.

The categories to be defined will be based upon some properties of the various indices j such that $leaf_0(j)$ is an origin leaf. Such indices are classified as follows.

α-,γ- and δ-Indices. Let A be the set of indices j such that $leaf_0(j)$ is an origin leaf. The sequence of indices in A (considered in increasing order) is divided into maximal contiguous subsequences with the following property: if $j_1, \ldots, j_h$ is one such subsequence then there is no β-index k, $j_1 < k \leq j_h$ and, in addition, $leaf_0(j_1) = leaf_0(j_2) = \cdots = leaf_0(j_h)$.

Definitions. α-,γ- and δ-indices are defined as follows. Consider an origin leaf x of T_0. If for all $j \in A$ such that $leaf_0(j) = x$, the subsequences containing j are singleton sequences then all such j's are defined to be α-indices and x is called an *α-origin leaf.* Suppose there exists $j \in A$ such that $leaf_0(j) = x$ and the subsequence containing j is not singleton. x is called a *non-α-origin leaf.* All those j such that $leaf_0(j) = x$ and j is the first index in its subsequence are δ-indices. The indices in the subsequence of δ-index j constitute the family $F(j)$. Lemma 6.5 shows that $str(x)$ is periodic with period $per(j) \leq \lceil \log^2 m \rceil$ in this case (see Fig.7). For each δ-index j, $F(j) = \{j_1, \ldots, j_h\}$,

$end(j)$ is the smallest index greater than j_h such that $p[end(j)] \neq p[end(j) - per(j)]$. Those indices $k \in F(j)$ such that $j_h - k$ is a multiple of the smallest multiple of $per(j)$ which is at least $\lceil \log^2 m \rceil$, are defined to be γ-indices.

If $j' \in F(j)$ we say that $fam(j') = j$.

By Lemma 6.5, for each δ-index j, $F(j) = \{j_1, \ldots, j_h\}$, $end(j)$ can be computed by finding the LCA of $leaf_0(j_1)$ and $leaf_0(j_h + per(j))$ in constant time after processing T_0 for LCA queries in $O(\log m)$ time and $O(m)$ work. Clearly, all α-, γ-, δ-indices and the families $F(j)$ for δ-indices j along with $per(j)$ and $end(j)$ can be obtained in $O(m)$ work and $O(\log m)$ time.

The above defined indices satisfy the following properties which the algorithm will exploit. The proofs of the following lemmas appear in Appendix 2.

Lemma 6.5 *Suppose j is a δ-index and $F(j) = \{j_1, j_2, \ldots, j_h\}$, $h > 1$, $j = j_1$. Let $per(j)$ be the smallest period of $p[j \ldots j+r-1]$. Then $j_2 - j_1 = j_3 - j_2 = \cdots = j_h - j_{h-1} = per(j) \leq \lceil \log^2 m \rceil$. Further $j_h + r - 1 < end(j) \leq j_h + per(j) + r - 1$ and $p[j \ldots end(j)-1]$ is periodic with period $per(j) \leq \lceil \log^2 m \rceil$ while $p[j_1 \ldots end(j)], p[j_2 \ldots end(j)], \ldots, p[j_h \ldots end(j)]$ all have periods greater than $\lceil \log^2 m \rceil$.*

The following corollaries can easily be derived from Lemmas 6.5 and 2.1, keeping in mind the structure illustrated in Fig.7.

Corollary 6.6 *If j, j' are two δ-indices such that $leaf_0(j) = leaf_0(j')$ then $per(j) = per(j')$. Further, for any k, k', $k \in F(j), k' \in F(j')$, if $end(j') - k' \leq end(j) - k$ then $p[k' \ldots end(j') - 1]$ is a prefix of $p[k \ldots end(j) - 1]$.*

Corollary 6.7 *Let j be a δ-index and $F(j) = \{j_1, j_2, \ldots, j_h\}$, where $j = j_1$ and $h > 1$. Let g, h' be such that $1 \leq g$, $1 \leq h' \leq h$. The following facts hold.*

1. *$str(leaf_g(j_{h'}))$ has period $per(j) \leq \lceil \log^2 m \rceil$ if $j_{h'} + 2^g r - 1 < end(j)$ and period greater than $\lceil \log^2 m \rceil$, otherwise.*
2. *If $h' > 1$ then $leaf_g(j_{h'}) = leaf_g(j_1)$ if and only if $j_{h'} + 2^g r - 1 < end(j)$, i.e., $str(leaf_g(j_{h'}))$ has period $per(j) \leq \lceil \log^2 m \rceil$.*
3. *If $h' < h$ and $leaf_g(j_{h'}) \neq leaf_g(j_{h'+1})$ then $leaf_g(j_{h'}), leaf_g(j_{h'+1}), leaf_g(j_{h'+2}), \ldots, leaf_g(j_h)$ are all distinct and $str(leaf_g(j_{h'+1})), str(leaf_g(j_{h'+2})), \ldots, str(leaf_g(j_h))$ all have periods greater than $\lceil \log^2 m \rceil$.*

Lemma 6.8 *Consider indices k, k', $k \in F(j)$, $k' \in F(j')$, where j, j' are δ-indices such that $leaf_0(j) = leaf_0(j')$. If $p[end(j)] = p[end(j')] = 1$ then $p[k \ldots end(j)] < p[k' \ldots end(j')]$ if and only if $end(j) - k > end(j') - k'$. If $p[end(j)] = 1$ and $p[end(j')] = 0$ then $p[k \ldots end(j)] > p[k' \ldots end(j')]$.*

Next, we bound the number of α-, δ- and γ-indices.

Lemma 6.9 *The number of α-indices is $O(\frac{m}{\log^2 m})$.*

Lemma 6.10 *For each δ-index $j > 1$, there exists a β-index k, $\max\{j', j - per(j)\} < k \leq j$, where j' is the largest index in A such that $j' < j$. Thus the number of δ-indices is $O(\frac{m \log m}{r}) = O(\frac{m}{\log^2 m})$.*

Lemma 6.11 *The number of γ-indices is $O(\frac{m \log m}{r}) = O(\frac{m}{\log^2 m})$.*

Lemma 6.12 *Suppose $j' \in F(j)$ for some δ-index j. Then j' is a γ-index if and only if $per(j)\lfloor \frac{end(j)-j'-r}{per(j)} \rfloor$ is divisible by the smallest multiple of $per(j)$ which is at least $\lceil \log^2 m \rceil$.*

7 Computing lists at sources

Recall from the end of Section 5 that for sources l, $L'_i(l)$ is an ordered list of one representative string from each equivalence class of the set of substrings of p of length $3.2^{i-1}r$ which begin at those indices $j \in indices_{i-1}(l)$. Also note that we have yet to describe how exactly sources are defined.

In Section 7.1, we define sources. In Section 7.2, we describe some data structures and auxiliary information required to be maintained by the algorithm. Section 7.3 describes the list computation algorithm for sources.

7.1 Defining Sources

Source Leaves. A leaf l of T_{i-1} is called a *source leaf* or *source* if one of the following is true.

1. For some j, $leaf_{i-1}(j) = l$, j is a β-index and there is at most one next-link incident upon l.
2. For some j, $leaf_{i-1}(j) = l$ and j is either an α-index or a γ-index.
3. For some j, $leaf_{i-1}(j) = l$, j is a δ-index and $j + 2^{i-1}r - 1 < end(j)$.

Since there are only $O(\frac{m}{\log^2 m})$ α-, β-, γ- and δ-indices, the list of all source leaves l can be obtained in $O(\frac{m}{\log^2 m})$ work and $O(\log m)$ time, given the data structures to be described in Section 7.2.

Lemmas 7.1, 7.2, 7.3 and 7.4 describe some important properties of sources.

Lemma 7.1 *Consider a source l. If source l has no next-links incident into it then the only index j such that $leaf_0(j) = l$ is $j = 1$. If there exists a β-index $j \neq 1$ such that $leaf_{i-1}(j) = l$ and l has only one next-link incident into it then all k such that $leaf_{i-1}(k) = l$ are β-indices. If there exists an α-index j such that $leaf_{i-1}(j) = l$, then all k such that $leaf_{i-1}(k) = l$ are α-indices.*

Proof. We consider each case in turn. The lemma is true in the first case because for each index $k > 1$, there is a next-link from $leaf_{i-1}(k-1)$ to $leaf_{i-1}(k)$. Consider the second case. The definition of β-indices implies that all indices k' such that $leaf_0(k'-1) = leaf_0(j-1)$ and $leaf_0(k') = leaf_0(j)$ are β-indices. Since l has only one incident next-link, $leaf_{i-1}(k-1) = leaf_{i-1}(j-1)$ for all indices k such that $leaf_{i-1}(k) = leaf_{i-1}(j)$. Since the set of those indices k such that $leaf_{i-1}(k-1) = leaf_{i-1}(j-1)$ and $leaf_{i-1}(k) = leaf_{i-1}(j)$ is a subset of the set of indices k' above, the lemma follows for this case. Consider the third case. The set of indices k such that $leaf_{i-1}(k) = leaf_{i-1}(j)$ is a subset of the set of indices k' such that $leaf_0(k) = leaf_0(j)$. The lemma follows for this case from the definition of α-indices. □

Lemma 7.2 *Consider a source l. If there exists a δ-index j such that $leaf_{i-1}(j) = l$ and $j + 2^{i-1}r - 1 < end(j)$ then $str(l)$ is periodic with period $per(j) \leq \lceil \log^2 m \rceil$; further, $leaf_{i-1}(k) = l$ if and only if $k \in F(j')$ for some δ-index j' such that $leaf_0(j) = leaf_0(j')$, and $k + 2^{i-1}r - 1 < end(j')$.*

Proof. The first part of the lemma follows immediately from the Corollary 6.7. Consider the second part. Since the set of all indices k such that $leaf_{i-1}(k) = l$ is a subset of the set of all indices k' such that $leaf_0(k') = leaf_0(j)$, by the definition of δ-indices, $k \in F(j')$ for some δ-index j' and $leaf_0(j') = leaf_0(k) = leaf_0(j)$. By Corollary 6.6, $per(j) = per(j')$. Since $str(l)$ is periodic with period $per(j) = per(j')$, it follows from Corollary 6.7 that $leaf_{i-1}(k) = l$ for $k \in F(j')$ if and only if $k + 2^{i-1}r - 1 < end(j')$. The lemma follows. □

Lemma 7.3 *If $l \in \mathcal{H}_{i-1}$ is not a source leaf and its ancestor $l' \in \mathcal{H}_0$ is an origin leaf then l' is a non-α-origin leaf.*

Proof. Suppose for a contradiction that l' is an α-origin leaf. Then all indices j such that $leaf_0(j) = l'$ are α-indices. It follows that all indices j such that $leaf_{i-1}(j) = l$ are α-indices and therefore l is a source leaf, a contradiction. □

Definitions. We say that a source l is *periodic* if it satisfies the conditions in Lemma 7.2, i.e., there exists a δ-index j such that $leaf_{i-1}(j) = l$ and $j + 2^{i-1}r - 1 < end(j)$, and *aperiodic* otherwise.

Lemma 7.4 *Consider a source l. Suppose there exists an γ-index j such that $leaf_{i-1}(j) = l$, but no δ-index j' such that $leaf_{i-1}(j') = l$. Then $str(l)$ has period greater than $\lceil \log^2 m \rceil$ and $leaf_{i-1}(k) = l$ only if k is a γ-index satisfying $k + 2^{i-1}r - 1 \geq end(fam(k))$; in addition, the value $end(fam(k)) - k$ is the same for all such k.*

Proof. There exists an γ-index j such that $leaf_{i-1}(j) = l$, but no δ-index j' such that $leaf_{i-1}(j') = l$. Then $leaf_{i-1}(fam(j)) \neq leaf_{i-1}(j) = l$. Clearly, all k such that $leaf_{i-1}(k) = l$ belong to the family of some δ-index; by Corollary 6.6, the value $per(fam(k))$ is the same for all such k. By Corollary 6.7, $str(l)$ has period greater than $\lceil \log^2 m \rceil$. It follows from Corollary 6.7 that for all k such that $leaf_{i-1}(k) = l$, $k + 2^{i-1}r - 1 \geq end(fam(k))$. By Lemma 6.5, $p[k \ldots end(fam(k)) - 1]$ is periodic with

period $per(fam(k)) \leq \lceil \log^2 m \rceil$ and $end(fam(k)) - k \geq r \geq per(fam(k))$ for each such k; therefore, the value $end(fam(k)) - k$ is the same for all such k. From Lemma 6.12, it follows that if any one such k is a γ-index (which is indeed the case as j is a γ-index) then so are all such k's. □

Some Machinery for Periodic Sources. We set up some machinery that will be required for handling periodic sources in Section 7.3. We need the following definition.

Definitions. If $leaf_{i-1}(j) = l$ is a source then let $rep(j)$ denote the string which represents $p[j \ldots j + 3.2^{i-1}r - 1]$ in $L'_i(l)$.

Consider a periodic source l. Recall Lemma 7.2. There exists a δ-index $j \in A$ such that $j + 2^{i-1}r - 1 < end(j)$. Further, $str(l)$ is periodic with period $per(j) \leq \lceil \log^2 m \rceil$. In addition, $leaf_{i-1}(k) = l$ if and only if $k \in F(j')$, for some δ-index j', and $k + 2^{i-1}r - 1 < end(j')$.

Defining a_j, b_j. Let j be a δ-index in $indices_{i-1}(l)$. From the above, $j + 2^{i-1}r - 1 < end(j)$. We define a_j, b_j, $a_j \leq b_j$, as follows. By Lemma 6.5, $F(j)$ has the form $\{j, j + per(j), \ldots, j + h * per(j)\}$. a_j is defined to be the largest index in F_j, if any, such that $a_j + 3.2^i r - 1 < end(j)$. If a_j is not defined above then let $a_j = j$. b_j is defined to be the largest index in F_j, if any, such that $b_j + 2^{i-1}r - 1 < end(j)$ but $b_j + 3.2^i r - 1 \geq end(j)$. If b_j is not defined above then, since $j + 2^{i-1}r - 1 < end(j)$ and $per(j) < r$, it must be the case that $j + h * per(j) + 3.2^i r - 1 < end(j)$; in this case, let $b_j = j + h * per(j)$. Note that $a_j = b_j$ in this case.

Lemma 7.5 *a_j and b_j satisfy Properties 1, 2 and 3 defined below.*

1. *For all $j' \in F(j)$, $leaf_{i-1}(j') = l$ if and only if $j \leq j' \leq b_j$*
2. *For all $j' \in F(j)$, $j \leq j' \leq a_j$, $rep(j') = rep(a_j)$.*
3. *For all $j', j'' \in F(j)$, $a_j \leq j' < j'' \leq b_j$, $rep(j') \neq rep(j'')$.*

Proof. Recall that $str(l)$ has period at most $\lceil \log^2 m \rceil$.

Consider Property 1 first. Note that $b_j + 2^{i-1}r - 1 < end(j)$ and that for all indices $j' \in F(j)$ greater than b_j, $j' + 2^{i-1}r - 1 \geq end(j)$. By Corollary 6.7, for all $j' \in F(j)$, $leaf_{i-1}(j') = leaf_{i-1}(j) = l$ if and only if $j \leq j' \leq b_j$. Thus Property 1 holds.

Consider Property 2 next. For all $j' \in F(j)$, $j < j' \leq a_j$, $j' + 3.2^i r - 1 < end(j)$; it follows from Lemma 6.5 and the definition of $end(j)$ that $p[j' \ldots j' + 3.2^i r - 1] = p[j \ldots j + 3.2^i r - 1]$. Property 2 follows immediately.

Finally, consider Property 3 and assume that $a_j < b_j$. From the definition of a_j, it follows that $j' + 3.2^i r - 1 \geq end(j)$ for all $j' \in F(j)$, $j' > a_j$. Recall that by Lemma 6.5, $p[j \ldots end(j) - 1]$ is periodic with period $per(j)$; further, $p[end(j)] \neq p[end(j) - per(j)]$, and that $end(j) > b_j + r - 1 \geq b_j + per(j) - 1$. Property 3 follows. □

The following lemmas are key in the computation of $L'_i(l)$ when l is periodic.

Lemma 7.6 *Consider δ-index $j \in indices_{i-1}(l)$ such that $a_j \neq b_j$ and $p[end(j)] = 1$. If $a_j \leq k, k + per(j) \leq b_j$ and $k, k + per(j) \in F(j)$ then $rep(k + per(j)) > rep(k)$[1].*

[1]Recall the lexicographic order defined on strings in Section 2.

Proof. By the definition of a_j, $k + rep(j) + 3.2^i r - 1 \geq end(j)$. By Lemma 6.5, $p[k + per(j) \ldots end(j) - 1] = p[k \ldots end(j) - 1 - per(j)]$ and $p[end(j)] = 1 \neq p[end(j) - per(j)]$. Then $p[end(j) - per(j)] = 0$ and the lemma follows. □

Lemma 7.7 *Consider δ-indices $j, j' \in indices_{i-1}(l)$ such that $a_j \neq b_j$ and $a_{j'} \neq b_{j'}$. For k, k', $a_j \leq k < b_j$, $a_{j'} \leq k' < b_{j'}$, if $rep(k + per(j)) \geq rep(k' + per(j'))$ then $rep(k) \geq rep(k')$.*

Proof. Since $leaf_{i-1}(j) = leaf_{i-1}(j')$, and since $per(j) < r$, $p[k \ldots k + per(j)] = p[k' \ldots k' + per(j)]$. The lemma follows immediately. □

Lemma 7.8 *Consider δ-indices $j, j' \in indices_{i-1}(l)$ such that $a_j \neq b_j$, $a_{j'} \neq b_{j'}$, $p[end(j)] = p[end(j')] = 1$. For $k, k' \in indices_{i-1}(l)$, $a_j \leq k \leq b_j$, $a_{j'} \leq k' \leq b_{j'}$, $k \in F(j)$, $k' \in F(j')$, if $\frac{b_j - k}{per(j)} < \frac{b_{j'} - k'}{per(j')}$ then $rep(k) \geq rep(k')$.*

Proof. By Corollary 6.6, $per(j) = per(j') = per$, say. Further, by the definition of b_j and from Lemma 6.5, $b_j + 2^{i-1} r - 1 < end(j)$, $b_{j'} + 2^{i-1} r - 1 < end(j')$, $b_j + per + 2^{i-1} r - 1 \geq end(j)$, and $b_{j'} + per + 2^{i-1} r - 1 \geq end(j')$. Therefore, $|(end(j) - b_j) - (end(j') - b_{j'})| < per$. Also, $b_j - k \leq b_{j'} - k' - per$. Therefore, $(end(j) - k) - (end(j') - k') < 0$. By Lemma 6.8, $p[k \ldots end(j)] > p[k' \ldots end(j')]$. The lemma follows. □

Lemma 7.9 *Consider δ-indices $j, j' \in indices_{i-1}(l)$ such that $p[end(j)] = 1$ and $p[end(j')] = 0$. For all k, k', $k \in F(j)$, $k' \in F(j')$, $rep(k) \geq rep(k')$.*

Proof. Follows immediately from Lemma 6.8. □

7.2 Data Structures And Auxiliary Information

Data Structures. We assume that the following data structures are available at the end of iteration $i - 1$. We will show in Section 9 how these data structures are maintained after iteration i.

(1). An array storing, for each α-, β-, γ-, and δ-index j, a pointer to $leaf_{i-1}(j)$.

(2). Next-links between the leaves $\mathcal{H}_{i-1}$ of T_{i-1}. Each leaf knows both the next-links pointing into it and the next-links pointing away from it.

(3). A data structure which given any two indices k, k' enables the computation of the longest common prefix of $p[k \ldots k + 2^{i-1} r - 1]$ and $p[k' \ldots k' + 2^{i-1} r - 1]$ in $O(1)$ time and work. Note that using this data structure, the longest common prefix of $p[k \ldots k + g.2^{i-1} r - 1]$ and $p[k' \ldots k' + g.2^{i-1} r - 1]$ can be computed in $O(g)$ time and work.

(4). For each index j, a data structure which enables the computation of $next_{i-1}(j)$ and $prev_{i-1}(j)$ in constant time and work, where $next_{i-1}(j)$ is the smallest index at least j such that either $next_{i-1}(j)$ is a β-index or $leaf_{i-1}(next_{i-1}(j))$ is a source, and $prev_{i-1}(j)$ is the largest index at most j such that either $prev_{i-1}(j)$ is a β-index or $leaf_{i-1}(prev_{i-1}(j))$ is a source.

Some Required Auxiliary Information. Consider a source l. We show how to compute $L_i'(l)$ along with the following auxiliary information associated with the strings in $L_i'(l)$.

1. For each string $x \in L_i'(l)$, an index $ind(x)$ such that there is an occurrence of x beginning at index $ind(x)$, i.e., $leaf_{i-1}(ind(x)) = l$ and $p[ind(x) \ldots ind(x) + 3.2^{i-1}r - 1] = x$.

2. For each string $x \in L_i'(l)$, an array $ptr(x)$ of pointers. The size of $ptr(x)$ equals $\max\{next_{i-1}(j) - j \,|\, p[j \ldots j + 3.2^{i-1}r - 1] = x\}$. Since for all j', $j < j' < next_{i-1}(j)$, $leaf_{i-1}(j')$ is not a source, the size of the ptr arrays over all strings x in $L_i'(l)$ and over all sources l is $O(m)$.

We remark that the array $ptr(x)$ is necessary only to set up the data structure described in (3) above and does not play any part in computing $L_i'(l)$. Each pointer in $ptr(x)$ points to a list of indices. Only the first of these is set up now; subsequent pointers in $ptr(x)$ will be set up later when lists at non-sources are computed. The first pointer in $ptr(x)$, $x \in L_i'(l)$, points to a list $lst(x)$ of all indices j such that $p[j \ldots j + 3.2^{i-1}r - 1] = x$. The indices in this list may not appear explicitly if x is periodic. We will show how this list is set up in work which is linear over all iterations shortly. Indices in the list pointed to by the subsequent pointers will always appear explicitly; further, these indices will be β-indices.

7.3 The List Computation Algorithm

There are two cases in the computation of $L_i'(l)$ for source l, namely the case in which l is aperiodic and the case in which it is periodic.

Aperiodic Case. By Lemmas 7.1 and 7.4, either $indices_{i-1}(l) = \{1\}$ or $indices_{i-1}(l)$ contains only α-indices or only β-indices or only γ-indices j with the property that $j + 2^{i-1}r - 1 \geq end(fam(j))$. In each case, the number of indices in $indices_{i-1}(l)$ over all aperiodic sources l is $O(\frac{m}{\log^2 m})$ by Lemmas 6.9, 6.2 and 6.11. $L_i'(l)$ along with the ind and lst values for each string in $L_i'(l)$ is easily computed by sorting the strings of length $3.2^{i-1}r$ beginning at indices in $indices_{i-1}(l)$ using the data structure (3) described above to perform comparisons in constant time and work; this takes $O(\log m)$ time and $O(\frac{m}{\log m})$ work over all aperiodic sources l. The work done over all iterations in this step is clearly linear. The indices in the lst lists appear explicitly in this case.

Periodic Case. Recall the machinery developed in Section 7.1 for handling periodic sources. Note that from the definition of a_j, b_j and Lemma 7.5, to compute $L_i'(l)$, it suffices to consider only those indices $j' \in F(j)$ for which $a_j \leq j' \leq b_j$, for each δ-index $j \in indices_{i-1}(j)$.

$L_i'(l)$ is computed for periodic source l in $O(\log^2 m)$ time and work $O(\frac{m}{\log m} + \Sigma(b_j - a_j))$, where the summation is over all δ-indices $j \in indices_{i-1}(l)$. To see that this work is linear over the entire algorithm, note that for any δ-index j such that $F(j) = \{j_1, \ldots, j_h\}$ and any index k such that $j_1 \leq k \leq j_h$, in at most two iterations is k between the a_j and

b_j values for these iterations. We remark that the ind and lst values can be computed for all strings in $L'_i(l)$ in the process of computing $L'_i(l)$ in the same time and work bounds. Also note that the indices in $F(j)$ between j and a_j inclusive appear implicitly in $lst(rep(a_j))$ in the form of the pair (j, a_j). The sum of the lengths of the lst lists over all strings in the lists at all sources l is then $O(\frac{m}{\log^2 m} + \Sigma(b_j - a_j))$, where the summation is over all δ-indices j. The sum of these lengths is clearly linear over all iterations.

The following three step procedure is performed.

Step 1. δ-indices $j \in indices_{i-1}(l)$ such that $a_j = b_j$ are considered. An ordered list X of $rep(a_j)$'s is obtained by simply sorting the $rep(a_j)$'s using the data structure (3) described before to perform comparisons in constant time and work. This takes $O(\log m)$ time and $O(\frac{m}{\log m})$ work over all periodic sources l. The work done over all iterations in this step is clearly linear.

Step 2. All δ-indices $j \in indices_{i-1}(l)$ such that $a_j \neq b_j$ are considered and an ordered list Y of strings $rep(k)$ is obtained, where $k \in F(j)$ and $a_j \leq k \leq b_j$ for some such j. This is done in two stages.

First, only those j such that $p[end(j)] = 1$ are considered.

For each such j, let Y_j be the list of substrings of p of length $3.2^{i-1}r$ beginning at indices k, $k \in F(j)$, $a_j \leq k \leq b_j$, and in increasing order of k. Y_j is easily set up in $O(1)$ time and $O(b_j - a_j)$ work. By Lemma 7.6, the strings in Y_j are in increasing lexicographic order. All Y_j's are then merged together to give a list Y^1. This merger is achieved as follows.

All Y_j's are vertically aligned along their rightmost column. Lemma 7.7 implies that the order in any column is the same as the order in the next column to the right (some inequalities may change to equalities but this does not affect the order) and Lemma 7.8 shows that all items in a column precede or equal all items in columns to the right. The strings in each $\lceil \log m \rceil$th column are sorted in $O(\log m)$ time and $O(\Sigma(b_j - a_j))$ work, where the summation is over all δ-indices j being considered currently. Subsequently, merging the Y_j's is accomplished easily in $O(\log m)$ time and $O(\Sigma(b_j - a_j))$ work, where the summation is over all δ-indices j being considered currently.

Second, those j such that $p[end(j)] = 0$ are considered and a list Y^0 is obtained in the manner similar to Y^1. Lemma 7.9 implies that Y^0 and Y^1 can be merged by simply concatenating the two lists to give a new ordered list; Y is the list obtained by removing duplicate strings from this list.

Step 3. X and Y are merged in $O(\log m)$ time and $O(|X| + |Y|) = O(\frac{m}{\log^2 m} + \Sigma(b_j - a_j))$ work, where the summation is over all δ-indices $j \in A$. $L'_i(l)$ is the list obtained by removing duplicates in the resulting list.

8 Computing $L'_i(l)$ for Remaining Leaves

We show how $L'_i(l)$ is computed for those remaining leaves l of T_{i-1} for which $|L'_i(l)| > 1$, given $L'_i(l)$ for source leaves l. Note that there is no need to compute $L'_i(l)$ if $|L'_i(l)| = 1$.

Definition. A next-link is said to be *frozen* if either it is incident upon a source leaf or it leads from $leaf_{i-1}(j-1)$ to $leaf_{i-1}(j)$ for some β-index $j \neq 1$. Let J'_{i-1} be the graph obtained from J_{i-1} by removing those edges which correspond to frozen next-links. The following lemma is key; the proof appears in Appendix 3.

Lemma 8.1 *The connected components of the graph J'_{i-1} are rooted trees, whose roots are all sources. The height of each tree is at most $3\lceil \log^2 m \rceil$.*

Lemma 8.1 facilitates the computation of the lists at all nodes l in the same connected component in J'_{i-1} by a breadth-first scan of the associated rooted tree. Recall from Section 7 that the list at the root of this tree has already been obtained. We need the following lemma and definitions before describing the algorithm for computing the lists at non-source leaves of T_{i-1}.

Lemma 8.2 *The number of nodes in J_{i-1} with in-degree 2 is $O(\frac{m}{2^{i-1}\log^2 m})$.*

Proof. By Corollary 3.3, the number of binary strings x of length $2^{i-1}r$ such that both $0x$ and $1x$ occur in s is at most $\frac{m \log m}{2^{i-1}r} = O(\frac{m}{2^{i-1}\log^2 m})$. The lemma follows. $\Box$.

Definitions. Recall that for sources l, $L'_i(l)$ is an ordered list of one representative string from each equivalence class of the set of substrings of p of length $3.2^{i-1}r$ which begin indices in $indices_{i-1}(l)$. $L'_i(l)$ for non-sources l is defined in a similar manner as follows. If l is distance k from the root of its connected component in J'_{i-1} then $L'_i(l)$ is an ordered list of one representative string from each equivalence class of the set of substrings of p of length $3.2^{i-1}r - k$ which begin at indices in $indices_{i-1}(l)$; further, $rep(j)$ denotes the string which represents $p[j \ldots j + 3.2^{i-1}r - 1 - k]$ in $L'_i(l)$. Note that by Lemma 8.1, $k \leq 3\lceil \log^2 m \rceil \leq r$, therefore, $3.2^{i-1}r - k \geq 2.2^{i-1}r$, as claimed in the penultimate paragraph of Section 5.

Some Required Auxiliary Information. For each string x in each $L'_i(l)$ computed, where l is a non-source, the following auxiliary information is computed. We remark that only the first of these is intrinsic to the procedure described in this section. The remaining three are used only for maintaining the data structure (3) described before which enables longest common prefix queries to be answered in constant time.

1. An index $ind(x)$ such that x occurs beginning at index $ind(x)$ in p.

2. A list β-$list(x)$ of all β-indices j such that there is an occurrence of x beginning at j.

3. A string $ext(x)$ with the following properties: $y = ext(x)$ is the longest string such that $y \in L'_i(l')$ for some ancestor l' of l in the tree formed by the connected component of J'_{i-1} containing l and x is a suffix of y. Since l is not a source, it will be the case that $ext(x) = x$ if and only if all j such that x occurs starting at j are β-indices, i.e., $leaf_{i-1}(j-1)$ is not the parent of $leaf_{i-1}(j) = l$ in the tree formed by the connected component of J'_{i-1} containing l.

4. If $ext(x) = x$, a $3\lceil \log^2 m \rceil$ sized array $ptr(x)$ of pointers is maintained. Recall that a similar data structure was defined earlier in Section 7.2 for sources l. Clearly, over all

x, l, the sum of the sizes of the *ptr* arrays is linear by 3 above and by Lemma 6.2. If $ext(x) = x$ then $\beta\text{-}list(x)$ is non-empty by 3 above and the first pointer in $ptr(x)$ will be made to point to this list. In addition, if $ext(x) \neq x$ but $\beta\text{-}list(x)$ is non-empty, then a pointer in $ptr(ext(x))$ will be made to point to $\beta\text{-}list(x)$.

The *lst* list (see Section 7.2), the β-*list*, the *ext* values and the *ptr* arrays are used specifically to perform the following computation: given a string $x \in L'_i(l)$ for some leaf l of T_{i-1}, to determine the set of indices j at which occurrences of x in p begin in work proportional to the number of such indices. As mentioned earlier, this computation will be useful in maintaining data structure (3).

The List Computation Procedure. The following procedure is used to compute the lists at non-sources.

Preprocessing Step. First, each node x in J'_{i-1} with in-degree 2 in J_{i-1} scans and marks the path (this path is against the edge direction) from x to the root of the tree containing x in J'_{i-1}. A node of J'_{i-1} is called *open* if it lies on one such path. By Lemma 8.1 and 8.2, this takes $O(\log^2 m)$ time and $O(\frac{m}{2^{i-1}})$ work per iteration. The total work done in this process in $O(m)$ over all iterations. By Lemma 8.1 and 8.2, the total number of open leaves is at most $O(\frac{m}{2^{i-1}})$.

Phase Sequence. Next up to $3\lceil \log^2 m \rceil$ phases are performed. Each phase takes $O(\log m)$ time; the total time taken in the iteration is thus $O(\log^3 m)$. In the jth phase, $j \geq 1$, those leaves l' in T_{i-1} which are distance j from their respective roots in J'_{i-1} and which satisfy one of the following two properties are active: either l' is open or $|L'_i(l)| > 1$, where l is the parent of l' in the tree formed by the connected component of J'_{i-1} containing l, l'. Recall that since l' is distance j from the root of its connected component in J'_{i-1}, the strings in $L'_i(l')$ will have length $3.2^{i-1}r - j$.

It is important to note that $L'_i(l')$ is not computed if l' is not open and $|L'_i(l)| = 1$. This is fine because, by Lemma 8.3, for all nodes l'' in the subtree of the tree formed by the connected component of J'_{i-1} containing l', $|L'_i(l'')| = 1$.

Lemma 8.3 *Let T be the tree formed by the connected component of J'_{i-1} containing l, l'. If l' is not open and $|L'_i(l)| = 1$ then for all l'' in the subtree of T rooted at l', $|L'_i(l'')| = 1$.*

Proof. We show in the next paragraph that $|L'_i(l')| = 1$. Then, since none of the nodes in the subtree in question is open (by the definition of open), the lemma follows by repeating the argument in the next paragraph once for each level in the subtree.

Since l' is not open, l' has in-degree 1 in J_{i-1}, i.e., there is exactly one next-link incident into l'. Recall that $|L'_i(l)| = 1$. Then l has out-degree 1 in J_{i-1}, i.e., there is exactly one next-link leading out of l. It then follows that the only strings in $L'_i(l')$ are suffixes of the strings in $L'_i(l)$ with length one less. Therefore, $|L'_i(l')| = 1$. □

Description of a Phase. Consider a particular phase. Let l' be a vertex which is active in this iteration and let l be its parent in the tree formed by the connected component to which l, l' belong in J'_{i-1}. l' is processed as follows.

$L'_i(l')$ is obtained from $L'_i(l)$ in $O(\log m)$ time and $O(|L'_i(l)| + |L'_i(l')| + \#\beta_{l'} \log m)$ work in a manner described below, where $\#\beta_{l'}$ is the number of β-indices j such that $leaf_{i-1}(j) = l'$. Since there are at least $|L'_i(l)| - 1 + |L'_i(l')| - 1$ internal nodes together in the subtrees of T_{i+1} rooted at l, l', the first two terms in the work done can be charged to internal nodes in these subtrees if even one of $|L'_i(l)|, |L'_i(l')| > 1$. Each internal node of the final suffix tree is charged a constant number of times in this process. If $|L'_i(l)| = |L'_i(l')| = 1$ then l' is open and the first two terms sum to $O(1)$ work which can be charged to l'; the total charge in this process in iteration i is $O(\frac{m}{2^{i-1}})$, which sums to $O(m)$ over all iterations. The last term in the work done is clearly linear over all iterations by Lemma 6.2.

Without loss of generality, assume that the next-link from l to l' is a 0-link. First, a list X comprising those strings x such that $y = ax$ for some $y \in L'_i(l)$, $y[2^{i-1}r + 1] = 0$, and $a = 0, 1$, is obtained. Clearly, the order of strings x in X is the same as the order of the corresponding strings y in $L'_i(l)$. For each $x \in X$, $ind(x) = ind(y) + 1$. This step takes $O(1)$ time and $O(|L'_i(l)|)$ work.

Second, all β-indices j, if any, such that $leaf_{i-1}(j) = l'$ are considered. Such indices exist only if l' has in-degree 2 in J'_{i-1}. An ordered list Y of the $rep(j)$'s is obtained in $O(\log m)$ time and $O(\#\beta_{l'} \log m)$ work using data structure (3) to perform comparisons in constant time and work. For each $x \in Y$, a β-index $ind(x)$ such that there is an occurrence of x beginning at index $ind(x)$, and $\beta\text{-}list(x)$, a list of all β-indices at which occurrences of x begin, are also obtained in this process.

Third, X and Y are merged to get $L'_i(l')$. If string x is in both X and Y then $ind(x)$ is chosen to be the value in X (this will prove to be critical later). This takes $O(\log m)$ time and $O(|L'_i(l')|)$ work.

Fourth, $ext(x)$ is obtained as follows. If $x \in X$ then $ext(x)$ is defined to be $ext(y)$, where $y \in L'_i(l)$ is such that $y = ax$, $a = 0, 1$. If $x \notin X$ then $x \in Y$ and $ext(x) = x$.

Fifth, the ptr arrays are updated as follows. Each $x \in L'_i(l)$ is considered. If $ext(x) = x$ then an array $ptr(x)$ of size $3\lceil \log^2 m \rceil$ is allocated. Finally, a pointer in $ptr(ext(x))$ is made to point to $\beta\text{-}list(x)$.

This completes the algorithm.

The following lemmas pertain to the auxiliary information maintained by the above algorithm and will be useful in setting up the data structures described earlier.

Lemma 8.4 *Suppose $x \in L'_i(l')$ and $z \in L'_i(l'')$, where l'' is an ancestor of l' in the connected component of J'_{i-1} containing l', l'' and x is a suffix of z. Then $ind(x) = ind(z) + |z| - |x|$.*

Proof. The lemma is clearly true if $ext(x) = x = z$, i.e., $l' = l''$. So assume that $ext(x) \neq x$. Clearly, $ext(x) = ext(z)$. Let l be the parent of l' in the tree formed by the connected component of J'_{i-1} containing l'. Then $ext(x) = ext(y)$ and $ind(x) = ind(y) + 1$, where $y \in L'_i(l)$ and $y = ax$ for some a, $a = 0, 1$. The lemma follows by a simple induction argument. □

Lemma 8.5 *Let $x \in L'_i(l')$ and suppose that $L'_i(l')$ is indeed computed by the above algorithm. Let G be the set of indices g in the lists pointed to by pointers in $ptr(ext(x))$ at the end of the phase in which $L'_i(l)$ is computed. For each $g \in G$, let k_g denote the distance between $leaf_{i-1}(g)$ and l' in J'_{i-1}. Occurrences of x in p begin at exactly the set of indices $\{g + k_g | g \in G\}$. In addition, for no $g, g' \in G$ is $g + k_g = g' + k_{g'}$.*

Proof. Suppose $ext(x) \in L'_i(l'')$. Note from the definition of $ext(x)$ that for all $g \in G$, the leaves $l'', leaf_{i-1}(g), l'$ are all in the same connected component of J'_{i-1} and that l'' is an ancestor of $leaf_{i-1}(g)$ and of l' in the tree T formed by this connected component. Recall that we are considering the instant at the end of the phase in which $L'_i(l')$ is computed. At this instant, $leaf_{i-1}(g)$ is a descendant of l'' and an ancestor of l' in T for all $g \in G$.

First, consider some $g \in G$. We show that there is an occurrence of x starting at index $g + k_g$. Let y denote the string $rep(g) \in L'_i(leaf_{i-1}(g))$. By definition, there is an occurrence of y beginning at index g in p. Since $g \in G$, y must be a suffix of $ext(x)$. Since $leaf_{i-1}(g)$ is an ancestor of l' in T, $|x| < |y|$ and, in particular, $|y| = |x| + k_g$. Since x is a suffix of $ext(x)$, x is a suffix of y, and therefore, there is an occurrence of x beginning at index $g + k_g$.

Second, consider an index j such that an occurrence of x begins at j. We show that for some $g < j$, $g \in G$ and $k_g = j - g$. Define g to be either the largest index less than or equal to j such that $leaf_{i-1}(g-1)$ is not the parent of $leaf_{i-1}(g-1)$ in T or the largest index less than or equal to j such that $rep(g) = ext(x)$, whichever is larger. Let y denote the string $rep(g) \in L'_i(leaf_{i-1}(g))$. Clearly, x is a suffix of y and $k_g = j - g$. Since $leaf_{i-1}(g)$ is on the path from l'' to l' in T and since x is a suffix of y and of $ext(x)$, y is a suffix of $ext(x)$. It remains to be shown that $g \in G$. If $y = ext(x)$ then $ext(y) = y$. Then, if $leaf_{i-1}(g)$ is the root then $g \in lst(y)$ and if $leaf_{i-1}(g)$ is not the root then g is a β-index and $g \in \beta\text{-}list(y)$; in either case, $g \in G$. Suppose $y \neq ext(x)$. Then $ext(y) = ext(x)$ and $leaf_{i-1}(g-1)$ is not the parent of $leaf_{i-1}(g)$ in T. It follows that g is a β-index and $g \in \beta\text{-}list(y)$, and therefore, $g \in G$.

Finally, suppose for a contradiction that there exist distinct $g, g' \in G$ such that $g + k_g = g' + k_{g'} = j$. From the previous paragraphs, there is an occurrence of x beginning at index j. Without loss of generality, assume that $g < g'$. Clearly, $k_g > k_{g'}$, i.e., $leaf_{i-1}(g)$ is a strict ancestor of $leaf_{i-1}(g')$ in T and $|rep(g)| > |rep(g')|$. Further, since we are considering the instant at the end of the phase in which $L'_i(l')$ is computed and since $g, g' \in G, leaf_{i-1}(g), leaf_{i-1}(g')$ are ancestors of l' in T. Therefore, $leaf_{i-1}(g), leaf_{i-1}(g+1), \ldots, leaf_{i-1}(g + k_g) = l'$ is a path in T. Then $leaf_{i-1}(g + k_g - k_{g'} - 1)$ and $leaf_{i-1}(g + k_g - k_{g'}) = leaf_{i-1}(g')$ are both in T. This implies that g' is not a β-index, which along with the fact that $leaf_{i-1}(g')$ is not the root gives the required contradiction. □

9 Maintaining Data Structures.

We describe how the following data structures mentioned earlier are maintained after each iteration i, assuming that they are available at the end of iteration $i - 1$. Data structures (1), (2), and (4) can clearly be obtained before the first iteration in $O(m)$

work and $O(\log m)$ time. Data structure (3) can be obtained before the first iteration by preprocessing T_0 for LCA queries in the same time and work bounds.

(1). An array storing, for each α-, β-, γ-, and δ-index j, a pointer to $leaf_i(j)$.

Since there are only $O(\frac{m}{\log^2 m})$ such indices and $l = leaf_{i-1}(j)$ is known for each such index j, $leaf_i(j)$ can be computed by binary searching $L_i(l)$ for each such j using the constant time comparisons facilitated by data structure (3). This takes $O(\log m)$ time and $O(\frac{m}{\log m})$ work per iteration, which is linear over all iterations.

(2). Next-links between the leaves $\mathcal{H}_i$ of T_i. Each leaf knows both the next-links pointing into it and the next-links pointing away from it.

Next-links between the leaves of T_i are set up as follows. All pairs of leaves l, l' of T_{i-1} such that there is a next-link from l to l' and either $|L_i(l)| > 1$ or $|L_i(l')| > 1$ are considered in parallel. Without loss of generality assume that the next-link from l to l' is a 0-link. For each string $0x \in L_i(l)$, the strings $x0, x1$ in $L_i(l')$ are found; this is done for all such x's by removing the first characters of the strings in $L_i(l)$ beginning with 0 and merging the resulting set of strings with $L_i(l')$ in $O(\log m)$ time and $O(|L_i(l)| + |L_i(l')|)$ work; this work is clearly linear over the entire algorithm. For $a = 0, 1, \$$, an a-link is set between the leaf representing string $0x$ in the subtree of T_i rooted at l and the leaf representing xa in the subtree of T_i rooted at l'.

Remark. Suffix links can easily be set up using next-links in the same time and work bounds as those required to set up next-links.

(3). A data structure which given indices k, k' finds the longest common prefix of $p[k \ldots k + 2^i r - 1]$ and $p[k' \ldots k' + 2^i r - 1]$, in $O(1)$ time and work.

This data structure has the following components. Component 3A can be maintained easily using (1) and the induced subtree construction algorithm in $O(\log m)$ time and $O(\frac{m}{\log m})$ work per iteration. Component 3C can be maintained easily as the L'_i lists are constructed without any time or work overhead. We will show how to maintain component 3B after the ith iteration.

3A. The subtree of T_i induced by leaves $leaf_i(j)$ is maintained and processed for LCA queries, where j is an α-, β-, or γ-index. These indices are called *special* indices i and the corresponding leaves are called *special* leaves.

3B. For each index j such that $leaf_i(j)$ is not a special leaf, a special index $c(j)$ is maintained such that $d(j)$, the length of the longest common prefix of $p[c(j) \ldots c(j) + 2^i r - 1]$ and $p[j \ldots j + 2^i r - 1]$ is maximized over all special indices. $d(j)$ is also maintained.

3C. For each index j such that $ind(x) = j$ for some string x in some $L'_i(l)$, l a leaf of T_{i-1}, a pointer from index j to the leaf l' of T_i such that $leaf_i(j) = l'$ is maintained.

The data structure in 3B is updated as the sets $L'_i(l)$ are computed in the ith iteration. Note that if $leaf_{i-1}(j)$ is not a source or a β-node then $leaf_{i-1}(j)$ is not a special leaf after iteration $i - 1$ and therefore, $c(j)$ and $d(j)$ would have been defined following iteration

$i-1$. It then suffices to consider those j such that $leaf_{i-1}(j)$ is either a source or a β-node. We consider each case in turn.

First, consider a leaf l of T_{i-1} which is a source such that $indices_{i-1}(l)$ has at least one special index. For each $x \in L'_i(l)$, if $lst(x)$ (recall $lst(x)$ was defined in Section 7.2) contains no special indices then for each $j \in lst(x)$, $c(j)$ is set to any special index in $lst(y)$, where $y \in L'_i(l)$ is the string such that the longest common prefix of y and x is the longest among all $z \in L'_i(l)$ such that $lst(z)$ contains at least one special index. $d(j)$ is set to be the longest common prefix of x and y. Note that whether or not $lst(x)$ contains any special indices can be computed while $lst(x)$ is set up without any additional overhed. Since $L'_i(l)$ is ordered, for all $x \in L'_i(l)$, the corresponding y can be found in $O(\log m)$ time and $O(|L'_i(l)|)$ work. Thus this step is accomplished in $O(\log m)$ time and $O(|L'_i(l)| + \#)$ work per source l per iteration, where $\#$ is the number of indices in $indices_{i-1}(l)$ for which $c(j)$ is defined in this iteration.

Second, consider a leaf l of T_{i-1} which is a β-node in J_{i-1} and consider the end of the phase in which $L'_i(l)$ is computed. All $x \in L'_i(l)$ such that there is no β-index j at which an occurrence of x begins are considered (i.e., β-$list(x)$ is empty). Consider one such x. String $y \in L'_i(l)$ is computed with the property that the longest common prefix of y and x is the longest among all $z \in L'_i(l)$ such that β-$list(z)$ is non-empty. Next, all j such that there is an occurrence of x beginning at j are determined using Lemma 8.5 in $O(\log m)$ time and work proportional to the number of such indices; $c(j)$ is set to any index in β-$list(y)$ and $d(j)$ is set accordingly. Thus this step is accomplished in $O(\log m)$ time and $O(|L'_i(l)| + \#)$ work per β-node l per iteration, where $\#$ is the number of indices in $indices_{i-1}(l)$ for which $c(j)$ is defined in this iteration.

Each index j is considered for setting $c(j)$ and $d(j)$ in at most two iterations (note that $L'_i(l)$ contains strings which are possibly longer but not twice as long as the corresponding strings in $L_i(l)$). Since there are at most $O(\frac{m}{\log^2 m})$ sources and β-nodes and since the subtree of T_{i+1} rooted at leaf l of T_{i-1} has at least $|L'_i(l)| - 1$ internal nodes, the total work done above is $O(m)$.

Computing Longest Common Prefixes. With the above data structures, the longest common prefix v of $x = p[k \ldots k + 2^i r - 1]$ and $y = p[k' \ldots k' + 2^i r - 1]$ can be computed as follows. First the longest common prefix of $p[k \ldots k + r - 1]$ and $p[k' \ldots k' + r - 1]$ is computed in constant time using T_0. If these two strings are different then the process ends. Suppose the two strings are the same.

Let $sp(h)$ denote the nearest special index to the right of index h. Without loss of generality, assume that $j = sp(k) - k \leq sp(k') - k'$. By Lemma 8.1, the fact that γ-indices occur less than $2\lceil \log^2 m \rceil$ apart in the family of any δ-index, and the fact that $m \geq 8$, $sp(k) \leq k + 5\lceil \log^2 m \rceil - 1 \leq k + r - 1$ and $sp(k') \leq k' + r - 1$. It follows that v can be easily determined after the longest common prefix v' of $p[sp(k) \ldots sp(k) + 2^i r - 1]$ and $p[k' + j \ldots k' + j + 2^i r - 1]$ is found in constant time as follows.

If $k' + j$ is a special index then this is easily done using 3A. Suppose $k' + j$ is not a special index. The main problem now is that $leaf_i(k' + j)$ is not known. However, recall that it can still be determined in constant time whether or not $leaf_i(k' + j)$ is a special leaf by simply checking whether or not $c(k' + j)$ has been computed ($c(k' + j)$ would have been computed if and only if $leaf_i(k' + j)$ is not a special leaf). There are two cases

next.

First, suppose $leaf_i(k'+j)$ is not a special leaf, i.e., $c(k'+j)$ has been computed. Then v' can be found in constant time by finding the LCA of $leaf_i(c(k'+j))$ and $leaf_i(sp(k))$ using 3A, and using $d(k'+j)$.

Second, suppose $leaf_i(k'+j)$ is a special leaf. Then $leaf_{i-1}(k'+j)$ is either a source or a β-node in J_{i-1}. In either case, the algorithm computes $L'_i(leaf_{i-1}(k'+j))$ even if this list has only one leaf. Recall that the main problem here is to get a pointer to $leaf_i(k'+j)$. Once this is available, the longest common prefix v' of $p[k'+j \ldots k'+j+2^i r-1]$ and $p[sp(k) \ldots sp(k)+2^i r-1]$ is found in constant time by finding the LCA of $leaf_i(sp(k))$ and $leaf_i(k'+j)$ using 3A. We show how to obtain this information. More specifically, we show how to compute a, b such that $p[b+a \ldots b+a+2^i r-1] = p[k'+j \ldots k'+j+2^i r-1]$, i.e., $leaf_i(k'+j) = leaf_i(b+a)$, and $leaf_i(b+a)$ can be computed in constant time from $b+a$.

By the definition of $prev_{i-1}(k'+j)$, $leaf_{i-1}(k'+j)$ and $leaf_{i-1}(prev_{i-1}(k'+j))$ are in the same connected component of J'_{i-1} and, in particular, $leaf_{i-1}(prev_{i-1}(k'+j))$ is the ancestor of $leaf_{i-1}(k'+j)$ in the tree formed by this connected component. Let $a = k'+j-prev_{i-1}(k'+j)$ and $b = ind(rep(prev_{i-1}(k'+j)))$. Then $|rep(prev_{i-1}(k'+j))| = |rep(k'+j)| + a$ and $rep(k'+j)$ is a suffix of $rep(prev_{i-1}(k'+j))$. a can be computed in constant time and work using data structure (4). To see that b can also be computed in the same time and work, note that $rep(h)$ can be computed for all indices h which have the form $prev_{i-1}(h')$ for some h', $1 \leq h' \leq m$, in constant time and work given rep values for all α-, β-, γ-, and δ-indices. This is because h must either be an α-, β-, γ- or δ-index, or $h \in F(g)$ for some δ-index g and $rep(h) = rep(g)$. Also note that the rep values for α-, β-, γ-, and δ-indices can easily be computed in $O(\log m)$ time and $O(\frac{m}{\log m})$ work.

Since $|rep(prev_{i-1}(k'+j))| = |rep(k'+j)|+a$ and $rep(k'+j)$ is a suffix of $rep(prev_{i-1}(k'+j))$, $p[b+a \ldots b+a+|rep(k'+j)|-1] = p[k'+j \ldots k'+j+|rep(k'+j)|-1]$. Since $|rep(k'+j)| \geq 2^i r$, $p[b+a \ldots b+a+2^i r-1] = p[k'+j \ldots k'+j+2^i r-1]$, i.e., $leaf_i(k'+j) = leaf_i(b+a)$. By Lemma 8.4, $b+a = ind(rep(b+a))$; by 3C, $leaf_i(b+a)$ is known. This shows the required claim.

(4). For each index j, a data structure which computes $next_i(j)$ and $prev_i(j)$ in constant time and work, where $next_i(j)$ is the smallest index at least j such that either $next_i(j)$ is a β-index or $leaf_i(next_i(j))$ is a source in the $i+1$th iteration, and $prev_i(j)$ is the largest index at most j such that either $prev_i(j)$ is a β-index or $leaf_i(prev_i(j))$ is a source in the $i+1$th iteration.

First, we show how $next_i(j)$ is computed in constant time. Let $k \geq j$ be the smallest index which is either in A (recall the definition of A from Section 6) or a β-index. If k is an α-, β- , γ-index or if k is in the family of some δ-index k' such that $k+2^i r-1 < end(k')$ then $next_i(j) = k$. Otherwise, $k \in A$, $k \in F(k')$ for some δ-index k' and $k+2^i r-1 \geq end(k')$; in this case, $next_i(j)$ is the nearest γ-index in $F(k')$ to the right of k.

Second, we show how $prev_i(j)$ is computed in constant time. Let $k \leq j$ be the largest index which is either in A or a β-index. If k is an α-, β- , γ-index or if k is in the family of some δ-index k' such that $k+2^i r-1 < end(k')$ then $prev_i(j) = k$. Otherwise, $k \in A$, $k \in F(k')$ for some δ-index k' and $k+2^i r-1 \geq end(k')$. Then there are two cases. If $k'+2^i r-1 < end(k')$ then $prev_i(j)$ is the larger of the largest index k'' in $F(k')$

such that $k'' + 2^i r - 1 < end(k')$ and the largest γ-index in $F(k')$ which is at most k. If $k' + 2^i r - 1 > end(k')$ and there exists a γ-index in $F(k')$ which is at most k then $prev_i(j)$ is the largest such γ-index. If $k' + 2^i r - 1 > end(k')$ and no such γ-index exists then $prev_i(j)$ is the largest β-index which is at most k'; the correctness of this definition follows from Lemma 6.10.

We conclude with the following theorems.

Theorem 9.1 *There is a CREW-PRAM algorithm to construct the suffix tree of a binary string s of length m in $O(\log^4 m)$ time, $O(m)$ work and $O(m)$ space.*

An easy generalization of Theorem 9.1 is Theorem 9.2, obtained by simply encoding a general alphabet in binary.

Theorem 9.2 *There is a CREW-PRAM algorithm to construct the suffix tree of a string s of length m drawn from any fixed alphabet set which takes $O(\log^4 m)$ time, $O(m)$ work and $O(m)$ space. In addition, there is an CREW-PRAM algorithm to construct the suffix tree of a string s of length m drawn from a general alphabet set which takes in $O(\log^4 m)$ time, $O(m \log |\Sigma|)$ work and $O(m \log |\Sigma|)$ space, after the characters in s have been sorted by alphabet, where $|\Sigma|$ is the number of distinct characters in s.*

10 Concluding Remarks

The above algorithm is optimal in all respects except for the time. The time can possibly be improved to $O(\log^3 m)$ by pipelining the computation of lists at non-sources with the various iterations; there are a number of technical difficulties here and we do not know how exactly this can be done. On the other hand, based on the following rough argument, we believe that it will be hard to obtain a linear space, linear work algorithm which takes $o(\log^3 m)$ time on the CREW-PRAM and $o(\log^2 m \text{ polyloglog}(m))$ time on the CRCW-PRAM. It seems hard to avoid performing $O(\log m)$ iterations, each iteration requiring sorting. Since sorting has an $O(\log m)$ work overhead with linear space, the size of the set to be sorted in each iteration must be $O(\frac{m}{\log^2 m})$. It seems hard to "fill in" the rest of the tree without performing $O(\log^2 m)$ stages, each stage requiring merging of lists. We believe that our algorithm can be implemented on a common CRCW-PRAM in $O(\log^3 m \text{ polyloglog}(m))$ time in the same work and space bounds.

A thing to note is that while the algorithm of [AILSV86] requires $O(m \log m)$ work, the information it computes enables *on-line pattern matching*, i.e., finding all occurrences of a pattern of length k in a text of length m, given the suffix tree of the text, in $O(\log k)$ time. Our representation of the suffix tree would require $O(k)$ time to trace the path of the pattern through the suffix tree. Whether the suffix tree can be preprocessed in linear work and $polylog(m)$ time so that on-line queries can be answered in $O(\log k)$ time remains an important problem.

11 Acknowledgements

We thank Richard Cole for help with this manuscript, S. Muthukrishnan for discussions, and Leszek Gąsieniec for pointing out that Corollary 3.3 is essentially tight. We also thank the referees for their suggestions.

References

[A85] A. Apostolico. The myriad virtues of sub-word trees. Combinatorial Algorithms on Words, Editors: A. Apostilico and Z. Galil, NATO–ASI series F: Computer and System Sciences, Vol. 12, Springer–Verlag, 1985, pp. 85–96.

[AILSV86] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3, 1988, pp. 347–365.

[AP83] A. Apostolico, F. Preparata. Optimal off–line detection of repetitions in a string. *Theoretical Computer Science*, 22, 1983, pp. 297–315.

[AP85] A. Apostolico, F. Preparata. Structural properties of the string statistics problem. *Journal of Computer and System Sciences*, 31, 1985, pp. 394–411.

[BD+91] P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, T. Radzik, S. Saksena. Improved deterministic parallel integer sorting. *Information and Computation*, 94, 1991, pp. 29-47.

[CHM86] B. Clift, D. Haussler, R. McConnell, T.D. Schneider, G.D. Stormo. Sequence Landscapes. *Nucleic Acids Research* 4, 1, 1986, pp. 141–158.

[C88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17, 1988, pp. 770-785.

[CV86] R. Cole, U. Vishkin. Deterministic coin tossing and accelering cascades: micro and macro techiques for designing parallel algorithms. *Proc. of the 18th ACM Symposium on Theory of Computing*, 1986, pp. 206–219.

[FM93] M. Farach, S. Muthukrishnan. Optimal parallel randomized suffix tree construction, Private Communication, 1994.

[G93] Leszek Gąsieniec. Private Communication, 1993.

[HT84] D. Harel, R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13, 1984, 338–355.

[KLP89] Z. Kedem, G. Landau, K. Palem. Optimal parallel suffix-prefix matching algorithm and application. *Proc. of the 1st ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 388-398.

[KMR72] R. Karp, R.E. Miller, A.L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. *Proc. of the 4th ACM Symposium on Theory of Computing*, 1972, pp. 125–136.

[LV86] G. Landau, U. Vishkin. Introducing efficient parallelism into approximate string matching. *Proc. of the 18th ACM Symposium on Theory of Computing*, 1986, pp. 220–230.

[LS62] R. Lyndon and M. Schutzenberger. The equation $a^M = b^N c^P$ in a free group. *Michigan Math. J.*, 9, 1962, 289-298.

[M76] E. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 23, 1976, pp. 262–272.

[MP93] S. Muthukrishnan, K. Palem. Highly efficient parallel dictionary matching. *Proc. of the 5th ACM Symposium on Parallel Algorithms and Architechture*, 1993.

[RPE81] M. Rodeh, V.R. Pratt, S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28, 1981, pp. 16–24.

[SV88] B. Schieber, U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17, 1988, pp. 1253–1262.

[SV93] S.J. Sahinalp, U. Vishkin. Symmetry breaking for suffix tree construction. *Proc. of the 26th ACM Symposium on Theory of Computing*, 1994.

[TV84] R. Tarjan, U. Vishkin. Finding bi-connected components and computing tree functions in polylogarithmic parallel time. *Proc. of the 25th IEEE Symposium on Foundations of Computer Science*, 1984, pp. 12-20.

[W73] P. Weiner. Linear pattern matching algorithm. *Proc. of the 14th IEEE Symposium on Switching and Automata Theory*, 1973, pp. 1–11.

Appendix 1

We prove Lemma 4.1 of Section 4.2, which states: If $x \in H_i$, $1 \leq i \leq h$, then $at_x \leq t_{x,i}$. We need the following lemma first.

Lemma 11.1 *Let $z_0, \ldots, z_f$ be nodes in T_0' such that $z_0, z_f \in H_i$, for some i, $1 \leq i < h$, and each $z_{f'} = p(z_{f'+1})$, for $0 \leq f' \leq f-1$. Suppose $at_{z_f} \leq t_{z_f,i}$ and $at_{suf(z_0)} \leq t_{suf(z_0),i+1}$. Then $at_{suf(z_f)} \leq \max\{at_{z_f}, at_{suf(z_0)}\} + \delta_f + 1 \leq \max\{t_{z_f,i}, t_{suf(z_0),i+1}\} + \delta_f + 1$, where δ_f is the number of nodes in H_{i+1} between $suf(z_0)$ and $suf(z_f)$, excluding both endpoints. In addition, if α_f is the processor which inserts z_f, then by time-step $\max\{at_{z_f}, at_{suf(z_0)}\} + \delta_f + 1 \leq \max\{t_{z_f,i}, t_{suf(z_0),i+1}\} + \delta_f + 1$, α_f would have traversed the node $suf(z_f)$.*

Proof. Note that since $z_{f'} = p(z_{f'+1})$, $at_{z_{f'}} \leq at_{z_{f'+1}}$, for $0 \leq f' \leq f-1$. We show by induction on f', $1 \leq f' \leq f$, that $at_{suf(z_{f'})} \leq \max\{at_{z_{f'}}, at_{suf(z_0)}\} + \delta_{f'} + 1$, where $\delta_{f'}$ is the number of nodes in H_{i+1} between $suf(z_0)$ and $suf(z_{f'})$, both endpoints excluded.

Suppose $\alpha_{f'}$ is the processor which inserts $z_{f'}$. In addition, we show that by time-step $\max\{at_{z_{f'}}, at_{suf(z_0)}\} + \delta_{f'} + 1 \leq \max\{t_{z_f,i}, t_{suf(z_0),i+1}\} + \delta_f + 1$, $\alpha_{f'}$ would have traversed the node $suf(z_{f'})$. The lemma follows immediately.

First, consider the case $f' = 1$. The processor α_1 which inserted z_1 waits in the list l_2 at z_0 until the time step in which $suf(z_0)$ is inserted. It then either resumes the rescanning phase or starts a new rescanning phase; this event occurs immediately following time-step $\max\{at_{z_1}, at_{suf(z_0)}\}$. In this rescanning phase, α_1 traverses the path from $suf(z_0)$ to $suf(z_1)$, until it reaches $suf(z_1)$ (if $suf(z_1)$ does not already exist, it seeks to insert $suf(z_1)$). Since $\max\{at_{z_1}, at_{suf(z_0)}\} \leq \max\{at_{z_f}, at_{suf(z_0)}\} \leq \max\{t_{z_f,i}, t_{suf(z_0),i+1}\}$, by the definition of H_{i+1}, the only nodes α_1 can traverse in the above process are those δ_1 nodes in H_{i+1} which are strictly between $suf(z_0)$ and $suf(z_1)$. It then follows that $at_{suf(z_1)} \leq \max\{at_{z_1}, at_{suf(z_0)}\} + \delta_1 + 1$, as required. It also follows that by time-step $\max\{at_{z_1}, at_{suf(z_0)}\} + \delta_1 + 1$, α_1 would have traversed the node $suf(z_1)$.

Next, assume that the claim holds for some f', $1 \leq f' \leq f-1$. We show that it holds for $f'+1$ as follows.

Note that since $at_{suf(z_{f'})} \leq \max\{at_{z_{f'}}, at_{suf(z_0)}\} + \delta_{f'} + 1 \leq \max\{at_{z_f}, at_{suf(z_0)}\} + \delta_{f'} + 1 \leq \max\{t_{z_f,i}, t_{suf(z_0),i+1}\} + \delta_{f'} + 1$, $suf(z_{f'}) \in H_{i+1}$. Therefore, $\delta = \delta_{f'+1} - \delta_{f'} - 1$ is precisely the number of nodes in H_{i+1} which follow $suf(z_{f'})$ and have len values less than $suf(z_{f'+1})$.

Essentially by the argument for the case $f' = 1$, it follows that $at_{suf(z_{f'+1})} \leq \max\{at_{z_{f'+1}}, at_{suf(z_{f'})}\} + \delta + 1 \leq \max\{at_{z_{f'+1}}, at_{suf(z_0)}\} + \delta_{f'} + 1 + \delta + 1 \leq \max\{at_{z_{f'+1}}, at_{suf(z_0)}\} + \delta_{f'+1} + 1$, as required. It also follows that by time-step $\max\{at_{z_{f'+1}}, at_{suf(z_0)}\} + \delta_{f'+1} + 1$, $\alpha_{f'+1}$, the processor which inserted $z_{f'+1}$, would have traversed the node $suf(z_{f'+1})$. □

Proof of Lemma 4.1. We show this by induction.

First, consider the case of H_1. The lemma is clearly true for all nodes in H_1, except x_1. We show that it is true for node x_1 also. Recall that α seeks to insert x_1 at the end of the first scanning phase which begins at *root*. α compares one character in p_l at every time-step in this phase. By Fact 2, each character in $p_l[1 \ldots len(x_1)+1]$ is

compared exactly once in this scanning phase. x_1 is inserted in the same time-step in which the unsuccessful comparison at $p_l[len(x_1)+1]$ is performed. It follows that $at_{x_1} \leq at_{root} + len(x_1) + 1 = t_{x_1,1}$.

Next, assume that the lemma is true for all $x \in H_1, \ldots, H_j$. We show that it is true for all $x \in H_{j+1}$. By definition, it is true for all those nodes $x \in H_{j+1}$ which are neither equal to x_{j+1} nor to $suf(x')$ for some node $x' \in H_j$. We claim that the lemma is also true for all nodes $suf(x')$, where $x' \in H_j$, and also for x_{j+1}. We show the former claim first.

We perform an induction on the nodes in the sequence H_j. The lemma is clearly true for $suf(root) = root$. Assume that it is true for node $x = suf(x')$, where $x' \in H_j$, $x' \neq x_j$. Let y' be the node which follows x' in H_j. Let y denote the node $suf(y')$, $y \in H_{j+1}$. There are three cases, depending upon whether $p(y') = x'$, $p(y')$ is a strict ancestor [2] of x', or $p(y')$ is a strict descendant of x'.

Case 1. Suppose $p(y') = x'$. Then, by Lemma 11.1 with $z_0 = x'$, $z_1 = y'$ and $f = 1$, $at_y \leq \max\{t_{y',j}, t_{x,j+1}\} + \delta + 1$, where δ is the number of nodes in H_{j+1} which are strictly between x and y. Clearly, $at_y \leq t_{y,j+1}$ in this case.

Case 2. Suppose $p(y')$ is a strict descendant of x'. Then $at_{p(y')} > at_{x'}$, otherwise $p(y')$ must be in H_j, a contradiction. Therefore, $p(p(y'))$ cannot be a strict ancestor of x'. If it is a strict descendant of x' then $at_{p(p(y'))} > at_{x'}$, otherwise $p(p(y'))$ must be in H_j. Repeating this argument, we get a sequence of nodes $z_0, z_1, \ldots, z_f$, where $z_0 = x'$, $z_f = y'$, and each $z_{f'} = p(z_{f'+1})$, for $0 \leq f' \leq f-1$. By Lemma 11.1, $at_{suf(z_f)} \leq \max\{t_{suf(z_0),j+1}, t_{z_f,j}\} + \delta + 1$, where δ is the number of nodes in H_{j+1} between $suf(z_0)$ and $suf(z_f)$. In other words, $at_y \leq \max\{t_{x,j+1}, t_{y',j}\} + \delta + 1 = t_{y,j+1}$.

Case 3. Suppose $p(y')$ is a strict ancestor of x'. Then $at_{x'} > at_{y'} \geq at_{p(y')}$. Let z be the node with the largest len value in H_j such that $len(z) \leq len(p(y'))$. As in the previous paragraph, there is a sequence of nodes $z_0, z_1, \ldots, z_f$, where $z_0 = z$, $z_f = y'$, and each $z_{f'} = p(z_{f'+1})$, for $0 \leq f' \leq f-1$. By Lemma 11.1, $at_{suf(z_f)} \leq \max\{at_{suf(z_0)}, at_{z_f}\} + \delta + 1$, where δ is the number of nodes in H_{j+1} strictly between $suf(z_0)$ and $suf(z_f)$. In other words, $at_y \leq \max\{at_{suf(z)}, at_{y'}\} + \delta + 1$. It then suffices to show that $\max\{at_{suf(z)}, at_{y'}\} + \delta + 1 \leq t_{y,j+1}$. This is seen as follows.

Let $w_0, \ldots, w_g$ denote the nodes in H_j from z to y', in that order, where $w_0 = z$ and $w_g = y'$. $p(y')$ is an ancestor of $w_1, \ldots, w_g$ by the definition of z; therefore, $at_{w_{g'}} > at_{w_g}$, $1 \leq g' < g$. We show by induction on this sequence that $\max\{at_{suf(z)}, at_{w_g}\} + \delta_{g'} + 1 \leq t_{suf(w_{g'}),j+1}$, for all g', $1 \leq g' \leq g$, where $\delta_{g'}$ is the number of nodes in H_{j+1} strictly between $suf(z)$ and $suf(w_{g'})$. For $g' = 1$, $t_{suf(w_1),j+1} = \max\{t_{suf(w_0),j+1}, t_{w_1,j}\} + \delta_1 + 1 \geq \max\{at_{suf(z)}, at_{w_1}\} + \delta_1 + 1 \geq \max\{at_{suf(z)}, at_{w_g}\} + \delta_1 + 1$, where the first inequality follows from the induction hypotheses of the inductions being performed on the nodes of H_j and on the sequences H_i. Next, we assume that the claim is true for g', $1 \leq g' < g$, and show that it is true for $g'+1$. By the induction hypothesis of the induction being performed on g', $t_{suf(w_{g'+1}),j+1} = \max\{t_{suf(w_{g'}),j+1}, t_{w_{g'+1},j}\} + (\delta_{g'+1} - \delta_{g'} - 1) + 1 \geq t_{suf(w_{g'}),j+1} + (\delta_{g'+1} - \delta_{g'} - 1) + 1 \geq \max\{at_{suf(z)}, at_{w_g}\} + \delta_{g'} + 1 + (\delta_{g'+1} - \delta_{g'} - 1) + 1 \geq \max\{at_{suf(z)}, at_{w_g}\} + \delta_{g'+1} + 1$, as required.

[2] We assume that a node is an ancestor but not a strict ancestor of itself.

This proves the lemma for all $suf(x') \in H_{j+1}$ such that $x' \in H_j$. If $x_{j+1} = suf(x_j)$ then the proof is complete. Otherwise, if $x_{j+1} \neq suf(x_j)$, it remains to show the lemma for x_{j+1}. Let $y = suf(x_j)$. α seeks to insert x_{j+1} at the end of the scanning phase which begins at y.

First, we show that this scanning phase must have begun by the time-step $t_{suf(x_j),j+1}+1$. Consider Cases 1–3 above with $y' = x_j$. By Lemma 11.1, α must have traversed $suf(x_j)$ by time-step $\max\{at_{z_f}, at_{suf(z_0)}\} + \delta_f + 1$, where $z_f = x_j$, the value of z_0 depends upon which one of Cases 1–3 holds, and δ_f is the number of nodes in H_{j+1} strictly between $suf(z_0)$ and $suf(z_f)$. By the first part of this lemma, $\max\{at_{z_f}, at_{suf(z_0)}\} + \delta_f + 1 \leq \max\{t_{z_f,j}, t_{suf(z_0),j+1}\} + \delta_f + 1$. The right hand side clearly equals $t_{suf(z_f),j+1} = t_{suf(x_j),j+1}$ for Cases 1 and 2. For Case 3, it was shown that $\max\{at_{z_f}, at_{suf(z_0)}\} + \delta_f + 1 \leq t_{suf(z_f),j+1} = t_{suf(x_j),j+1}$.

Finally, consider this scanning phase. α compares one character in p_l at every time-step in this phase. By Fact 2, a distinct character in p_l is compared in each time-step in this scanning phase, including the time-step in which α inserts x_{j+1}. It follows that $at_{x_{j+1}} \leq t_{y,j+1} + len(x_{j+1}) - len(y) + 1 = t_{x_{j+1},j+1}$. □

Appendix 2

The proofs of the lemmas defined in the latter part of Section 6 are given here.

Proof of Lemma 6.5. Recall that $leaf_0(j_1) = leaf_0(j_2) = \cdots = leaf_0(j_h)$. Let x denote $leaf_0(j_1)$. Note that there are no β-indices or indices in A strictly between j_1 and j_2. By Fact 3 and Lemma 6.1, the path $leaf_0(j_1), leaf_0(j_1+1), \ldots, leaf_0(j_2-1)$ in G contains only vertices and edges in the same connected component of G_3. Since each connected component of G_3 is a rooted tree, any vertex in the same connected component in G_3 as x is distance at most $\lceil \log^2 m \rceil - 1$ from x in G_3. It follows that $per(j) \leq j_2 - j_1 \leq \lceil \log^2 m \rceil$. Since $3\lceil \log^2 m \rceil \leq r = 2\lceil \log^3 m \rceil$ for $m \geq 8$, $p[j_1 \ldots j_1 + r - 1]$ is periodic with period at most $\lceil \log^2 m \rceil$. Further, $j_2 - j_1$ is a period of $p[j_1 \ldots j_1 + r - 1]$. In fact, the smallest period $per(j)$ of $p[j_1 \ldots j_1 + r - 1]$ must equal $j_2 - j_1$, otherwise, by Lemma 2.1, it is easy to see that there must be some j', $j_1 < j' < j_2$, such that $p[j' \ldots j' + r - 1] = p[j_1 \ldots j_1 + r - 1]$, i.e., $j' \in A$, a contradiction.

Next, we show that $j_2 - j_1 = j_3 - j_2 = \cdots = j_h - j_{h-1} = per(j)$. Suppose for a contradiction that this is not true. Let k, $3 \leq k \leq h$ be the smallest number such that $j_k - j_{k-1} \neq per(j)$. Let v be the prefix of $str(x)$ of length $per(j)$. By Lemma 2.1, $str(x)$ has the form $v^g v'$, where v' is a prefix of v and $g \geq 2$. If for all l, $1 \leq l \leq per(j)$, $p[j_{k-1} + r - 1 + l] = p[j_{k-1} + r - 1 + l - per(j)]$ then clearly, $p[j_{k-1} + per(j) \ldots j_{k-1} + per(j) + r - 1] = p[j_{k-1} \ldots j_{k-1} + r - 1]$, i.e., $j_{k-1} + per(j) = j_k$, a contradiction. So assume that there exists an l, $1 \leq l \leq per(j)$, such that $p[j_{k-1} + r - 1 + l] \neq p[j_{k-1} + r - 1 + l - per(j)]$ and consider the smallest such l. Then, for all multiples l' of $per(j)$, $l' \leq \lceil \log^2 m \rceil$, $p[j_{k-1} + r - 1 + l] \neq p[j_{k-1} + r - 1 + l - l']$. This along with Lemma 2.1 implies that $j_k - j_{k-1} > r - per(j) \geq \lceil \log^2 m \rceil$. Since the connected components of G_3 are trees of height $\lceil \log^2 m \rceil - 1$, the path $leaf_0(j_{k-1}), leaf_0(j_{k-1}+1), \ldots, leaf_0(j_k)$ in G traverses an edge in $E_1 \cup E_2 \cup E_3$ which is not incident on x. By Fact 3, there must be a β-index or

an index in A between j_{k-1} and j_k, a contradiction. Therefore, $j_2 - j_1 = j_3 - j_2 = \cdots = j_h - j_{h-1} = per(j) \leq \lceil \log^2 m \rceil$. It follows that $p[j_1 \ldots j_h + r - 1]$ is periodic with period $per(j)$ (see Fig.7).

Since $p[j_h \ldots j_h + r - 1]$ has period $per(j)$, $end(j) \geq j_h + r - 1$. Further, $end(j) < j_h + per(j) + r - 1$, otherwise $j_h + per(j)$ will be in $F(j)$ as well. Clearly, $p[j \ldots end(j) - 1]$ is periodic with period $per(j)$. By Lemma 2.1, $p[j_1 \ldots end(j)], p[j_2 \ldots end(j)], \ldots, p[j_h \ldots end(j)]$ have periods at least $r - per(j) + 1 > \lceil \log^2 m \rceil$. $\Box$.

Proof of Lemma 6.8. Recall from Corollary 6.6 that $per(j) = per(j')$.

First, suppose $p[end(j)] = p[end(j')] = 1$.

Suppose $end(j) - k > end(j') - k'$. We show that $p[k \ldots end(j)] < p[k' \ldots end(j')]$. By Corollary 6.6, $p[k' \ldots end(j') - 1]$ is a proper prefix of $p[k \ldots end(j) - 1]$. Since $p[end(j')] = 1$, it suffices to show that $p[k + end(j') - k'] = 0$. By the definition of $end(j')$, $p[end(j') - per(j')] = 0$. Since $p[k' \ldots end(j') - 1]$ is a proper prefix of $p[k \ldots end(j) - 1]$, $p[k + end(j') - k' - per(j')] = 0$. By Lemma 6.5 and the fact that $k + end(j') - k' < end(j)$, $p[k + end(j') - k' - per(j) + per(j)] = p[k + end(j') - k' - per(j)] = 0$, as claimed.

Now, suppose $p[k \ldots end(j)] < p[k' \ldots end(j')]$. We show that $end(j) - k > end(j') - k'$. Suppose for a contradiction that $end(j) - k \leq end(j') - k'$. By Corollary 6.6, $p[k \ldots end(j) - 1]$ is a prefix of $p[k' \ldots end(j') - 1]$. Since $p[end(j)] = p[end(j')] = 1$, it follows that these two strings are not of the same length, otherwise, $p[k \ldots end(j)] = p[k' \ldots end(j')]$. Therefore, $end(j) - k < end(j') - k'$. From the previous paragraph, with the roles of k, k' interchanged, it follows that $p[k \ldots end(j)] > p[k' \ldots end(j')]$, a contradiction.

Next, suppose $p[end(j)] = 1 \neq p[end(j')] = 0$. If $end(j) - k = end(j') - k'$ then the lemma immediately follows from Corollary 6.6. There are two cases next.

First, suppose $end(j') - k' < end(j) - k$. By Corollary 6.6, $p[k' \ldots end(j') - 1]$ is a proper prefix of $p[k \ldots end(j) - 1]$. Since $p[end(j')] = 0$, it suffices to show that $p[k + end(j') - k'] = 1$. By the definition of $end(j')$, $p[end(j') - per(j')] = 1$. Since $p[k' \ldots end(j') - 1]$ is a proper prefix of $p[k \ldots end(j) - 1]$, $p[k + end(j') - k' - per(j')] = 1$. By Lemma 6.5 and the fact that $k + end(j') - k' < end(j)$, $p[k + end(j') - k' - per(j) + per(j)] = p[k + end(j') - k' - per(j)] = 1$, as claimed.

Second, suppose $end(j') - k' > end(j) - k$. By Corollary 6.6, $p[k \ldots end(j) - 1]$ is a proper prefix of $p[k' \ldots end(j') - 1]$. Since $p[end(j)] = 1$, it suffices to show that $p[k' + end(j) - k] = 0$. By the definition of $end(j)$, $p[end(j) - per(j)] = 0$. Since $p[k \ldots end(j) - 1]$ is a proper prefix of $p[k' \ldots end(j') - 1]$, $p[k' + end(j) - k - per(j)] = 0$. By Lemma 6.5 and the fact that $k' + end(j) - k < end(j')$, $p[k' + end(j) - k - per(j') + per(j')] = p[k' + end(j) - k - per(j')] = 0$, as claimed. $\Box$

Proof of Lemma 6.9. Consider an α-index j which is not one of the two rightmost indices in A. Let k, k', $k' > k$, be the next two indices in A to the right of j. We show that either there is a β-index between j and k' inclusive, or $k' - j \geq \lceil \log^2 m \rceil$. The lemma then follows from Lemma 6.2.

If j is a β-index then the above claim is clearly true. Suppose j is not a β-index. By the definition of an α-index, either there is a β-index between j and k (k included) or $leaf_0(j) \neq leaf_0(k)$, otherwise j and k would be in the same maximal subsequence. The

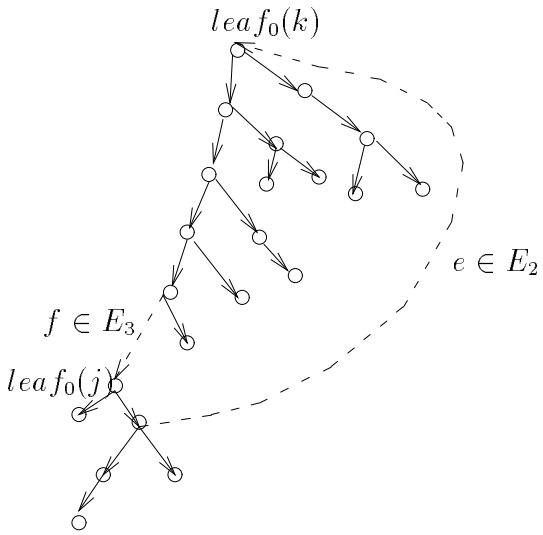

Figure 8: A connected component of G_1.

above claim follows in the first case too. So suppose that there is no β-index between j and k inclusive. Then since $leaf_0(j) \neq leaf_0(k)$, $leaf_0(j), leaf_0(k)$ are in different connected components of G_3. Consider the first edge e in G on the path from $leaf_0(j)$ to $leaf_0(k)$ which is missing in G_3 and which connects vertices in different connected components of G_3. Then e is either in E_1, E_2, or E_3. We consider each case separately. Recall Fact 3.

If $e \in E_1$ then there is a β-index between k and j inclusive, which is a contradiction. If $e \in E_3$, the claim that $k' - j > k - j \geq \lceil \log^2 m \rceil$ follows from the definition of the set E_3. Suppose $e \in E_2$. Since all edges in E_2 are incident upon origin leaves and since there are no indices in A strictly between j and k, e must be incident upon $leaf_0(k)$. Since $leaf_0(j) \neq leaf_0(k)$ and $e \in E_2$, $leaf_0(j)$ and $leaf_0(k)$ are in the same connected component in G_2 but in different connected components in G_3; further, $leaf_0(k)$ must be the root of its connected component in G_2 (see Fig.8). Since $leaf_0(j)$ is the root of its connected component in G_3, it follows that the edge f in G_2 incident upon $leaf_0(j)$ is in E_3. For all edges $e' \in G$ such that e' leads out of one of the vertices in the same connected component in G_3 as $leaf_0(k)$, $e' \notin E_2$ (because if e' was in E_2 then e' would be incident upon $leaf_0(k)$ and therefore $e' = e$, which is a contradiction because the other endpoint of e is not is the same connected component of G_3 as $leaf_0(k)$). It follows that either there is a β-index between k and k' (k' included) or $k' - k \geq \lceil \log^2 m \rceil$, as claimed. □

Proof of Lemma 6.10. If j is a β-index the lemma is clearly true. So assume that j is not a β-index. If $j - per(j) < 1$ and j is the smallest index in A (i.e., j' is not defined) then the lemma is true as 1 is a β-index. So suppose that either $j - per(j) \geq 1$ or j' is defined.

Since $leaf_0(j)$ is a non-α-origin leaf, there exists a non-singleton maximal subsequence (as defined in Section 6.1) of indices associated with this leaf. Let e be the first index in one such subsequence. By definition, e is a δ-index. Note that $leaf_0(e) = leaf_0(j)$ and $|F(e)| > 1$. There are no β-indices or indices in A strictly between e and $e + per(e)$. By Fact 3, it follows that all edges on the path $leaf_0(e), leaf_0(e+1), \ldots, leaf_0(e+per(e)-1)$ in G are in $E - (E_1 \cup E_2 \cup E_3)$; therefore $leaf_0(e), leaf_0(e+1), \ldots, leaf_0(e+per(e)-1)$ is a path in G_3. Since $leaf_0(e) = leaf_0(e + per(e))$, there is an edge between $leaf_0(e + per(e)-1)$ and $leaf_0(e)$ in G, i.e., $leaf_0(e), leaf_0(e+1), \ldots, leaf_0(e+per(e)-1), leaf_0(e)$ is a cycle C in G. Since there are no β-indices between $e+1, e+per(e)$ inclusive, none of the edges in C is in E_1.

Let $k \leq j$ be the largest index such that $leaf_0(k)$ is one of $leaf_0(e), leaf_0(e + 1), \ldots, leaf_0(e + per(e) - 1)$ but $leaf_0(k-1)$ is not. We show that $k > j - per(j)$ if $j - per(j) \geq 1$. In addition, we show that $k > j'$ if j' is defined and greater than $j - per(j)$. Finally, we show that k is a β-index. The lemma follows.

Suppose for a contradiction that $j - per(j) \geq 1$ and either k does not exist or $k \leq j - per(j)$. Since $p[j \ldots j + r - 1]$ is periodic with period $per(j)$ by Lemma 6.5 and since $r \geq 3per(j)$, it follows that for all k', $1 \leq k' \leq per(j)$, $leaf_0(j-k') = leaf_0(e+per(e)-k')$. In particular, by Corollary 6.6, $leaf_0(j-per(j)) = leaf_0(e+per(e)-per(j)) = leaf_0(e) = leaf_0(j)$. Since none of the edges in C is in E_1, no index h such that $leaf_0(h)$ and $leaf_0(h-1)$ are both in C can be a β-index. Therefore, $j - per(j) + 1, \ldots, j$ are not β-indices. Since there are no β-indices between $j - per(j) + 1$ and j inclusive and since there are no indices in A strictly between $j - per(j)$ and j, j cannot be a δ-index, a contradiction.

Next, suppose for a contradiction that j' is defined and greater than $j - per(j)$ and that either k does not exist or $k \leq j'$. Using the argument in the previous paragraph, for all k', $1 \leq k' \leq j - j'$, $leaf_0(j - k') = leaf_0(e + per(e) - k')$. Then since $j' \in A$, $leaf_0(j')$ must be an origin leaf. Therefore, $leaf_0(j') = leaf_0(j) = leaf_0(e)$. As in the previous paragraph, $j'+1, j'+2, \ldots, j$ are not β-indices. Since there are no β-indices between $j'+1$ and j inclusive and since there are no indices in A strictly between j' and j, j cannot be a δ-index, a contradiction.

To see that k is a β-index, recall that $leaf_0(e), leaf_0(e+1), \ldots, leaf_0(e + per(e) - 1), leaf_0(e)$ form a cycle C in G. Since $leaf_0(k-1)$ is outside C and there is edge in G from $leaf_0(k-1)$ to $leaf_0(k)$ which is in C, $leaf_0(k)$ has in-degree 2 in G. From Lemma 6.1, it follows that $leaf_0(k)$ is a β-node in G. Let h be an index such that $leaf_0(k) = leaf_0(e + h)$, $1 \leq h \leq per(j)$. By Lemma 6.1, either $e + h$ or k is a β-index. Since there are no β-indices between $e+1, e+per(e)$ inclusive, k is a β-index, as claimed. □

Proof of Lemma 6.11. The lemma follows from Lemma 6.10 and the fact that γ-indices in the family of a δ-index appear at least $\lceil \log^2 m \rceil$ distance apart. □

Proof of Lemma 6.12. Let j_h be the largest index in $F(j)$. By definition, j' is a γ-index if and only if $j_h - j'$ is divisible by l, the smallest multiple of $per(j)$ which is at least $\lceil \log^2 m \rceil$. By Lemma 6.5, $j_h + r - 1 < end(j) \leq j_h + per(j) + r - 1$ and therefore, $j_h - j' \leq end(j) - j' - r < j_h + per(j) - j'$. It follows that $\lfloor \frac{j_h - j'}{per(j)} \rfloor = \lfloor \frac{end(j) - j' - r}{per(j)} \rfloor$.

Therefore $j_h - j'$ is divisible by l if and only if $per(j)\lfloor\frac{end(j)-j'-r}{per(j)}\rfloor$ is divisible by l. □

Appendix 3

We give the proof of Lemma 8.1 here.

Proof. We claim that all non-source vertices in J'_{i-1} have in-degree exactly 1 and that all source vertices have in-degree 0; the first part of the lemma follows from this claim. The claim is shown as follows. All nodes in J_{i-1}, except possibly $leaf_{i-1}(1)$, have in-degree at least 1 and at most 2. By Lemma 6.1, any node x with in-degree 2 in J_{i-1} is a β-node in J_{i-1}, i.e., there is a β-index $j > 1$ such that $leaf_{i-1}(j) = x$. Clearly, x will have in-degree 1 in J'_{i-1}. It follows that all nodes x in J'_{i-1} have in-degrees at most 1; in addition, the in-degree is 0 if and only if either x is a source or $x = leaf_{i-1}(j)$ for some β-index $j > 1$ and x has in-degree 1 in J_{i-1} or x has in-degree 0 in J_{i-1} (i.e., $leaf_{i-1}(1) = x$); by the definition of source, x is a source in the last two cases too.

Next, we show that the heights of the trees forming the connected components of J'_{i-1} are bounded by $3\lceil\log^2 m\rceil$. We need the following definition and a preliminary fact. For any leaf z of T_{i-1}, let z^0 denote the leaf of T_0 of which z is a descendant. Also note that if there is an edge from z to z' in J'_{i-1} then for no β-index $j > 1$ is $leaf_{i-1}(j-1) = z$ and $leaf_{i-1}(j) = z'$. We claim that for no β-index $j > 1$ is $leaf_0(j-1) = z^0$ and $leaf_0(j) = z'^0$. Assume for a contradiction that such a j exists. Then, by Lemma 6.1, all indices $j' > 1$ such that $leaf_0(j'-1) = z^0$ and $leaf_0(j') = z'^0$ are β-indices. Therefore, all indices j' such that $leaf_{i-1}(j'-1) = z$ and $leaf_{i-1}(j') = z'$ are β-indices, a contradiction.

Let y be a node in J'_{i-1}. Let w be the root of the tree in J'_{i-1} containing y. It suffices to show that the length of the path in J'_{i-1} from w to y is at most $3\lceil\log^2 m\rceil$.

Let z, $z \neq w$, be the nearest node to y, if any, on the path from w to y in J'_{i-1} such that z^0 is an origin leaf. We show that z is indeed defined and that the length of the path from z to y in J'_{i-1} is at most $\lceil\log^2 m\rceil$. Let $y_k, y_{k-1}, \ldots, y_1$, $w = y_k, y = y_1$, be the nodes, in order, on the path from w to y in J'_{i-1}. Clearly, each y_h^0 has an edge to y_{h-1}^0 in $G = J_0$, $2 \leq h \leq k$. None of these edges is in E_1 as edges in E_1 only lead from $leaf_0(g-1)$ to $leaf_0(g)$ for β-indices $g > 1$. If none of these edges is in either E_2 or E_3 then $y_1^0, y_2^0, \ldots, y_k^0$ is a path in G_3 and therefore $k \leq \lceil\log^2 m\rceil$, as required. So assume for the rest of the proof that one of these edges is in $E_2 \cup E_3$. Since edges in E_2, E_3 are incident into origin leaves (recall Fact 3), z is defined. It also follows that there is a path from z^0 to y^0 in G_3 and therefore, the length of the path from z to y is at most $\lceil\log^2 m\rceil$.

It now suffices to show that the length of the path from w and z in J'_{i-1} is at most $2\lceil\log^2 m\rceil$. Suppose for a contradiction that this is not so.

Since z is not a source, by Lemma 7.3, z^0 must be a non-α-origin leaf. Let e be a δ-index such that $leaf_0(e) = z^0$ and $|F(e)| > 1$. Such an index exists by the definition of a non-α-origin leaf. Let d be an index such that $leaf_{i-1}(d) = z$. Since $leaf_0(d) = z^0$, $d \in F(d')$ for some δ-index d'. By Lemma 6.5, the prefix of $str(z)$ of length r is periodic with period $per(e) \leq \lceil\log^2 m\rceil$. By the definition of sources, if $d' + 2^{i-1}r - 1 < end(d')$ and $leaf_{i-1}(d') = z$ then z would have been a source; it follows that either $d' + 2^{i-1}r - 1 \geq end(d')$ or $leaf_{i-1}(d') \neq z$. In the former case, $d + 2^{i-1}r - 1 \geq end(d')$. In the latter

case also, by Corollary 6.7, $d + 2^{i-1}r - 1 \geq end(d')$. This implies that $str(z)$ has period greater than $\lceil \log^2 m \rceil$.

Next, we show that for every node v on the path from w to z, there exists an f, $e \leq f < e + per(e)$ such that $str(v)[1 \ldots r] = p[f \ldots f + r - 1]$. This is shown by induction. As the base case, the claim holds when $v = z$ with $f = e$. Assume that the claim is true for some v' on the above path and v is the node preceding v' on this path. Let f', $e \leq f' < e + per(e)$, be the value such that $str(v')[1 \ldots r] = p[f' \ldots f' + r - 1]$. Let $f = f' - 1$ if $f' > e$ and $f = e + per(e) - 1$, otherwise. Let j be an index such that $leaf_{i-1}(j) = v$ and $leaf_{i-1}(j+1) = v'$. Since $p[f' \ldots f' + r - 1] = p[j + 1 \ldots j + 1 + r - 1]$ and since $p[e \ldots e + r - 1] = p[e + per(e) \ldots e + per(e) + r - 1]$, $p[f \ldots f + r]$ and $p[j \ldots j + r]$ can differ only in the first character. Suppose for a contradiction that these two strings actually differ in the first character. Then either $f + 1$ is a β-index or $j + 1$ is a β-index. $j + 1$ cannot be a β-index, otherwise the edge from v to v' in J_{i-1} would have been removed in J'_{i-1}. By the definition of a non-α-origin leaf, $f + 1$ cannot be a β-index, a contradiction.

Consider the set S of those nodes on the path from w to z whose distances from z are multiples of $per(e)$ and at most $2\lceil \log^2 m \rceil$. Let $x_1, \ldots, x_h$ be the nodes in S ordered by decreasing distance from z. In order to obtain a contradiction, we show that one of the nodes in S must be a source; this contradicts the fact that sources have in-degree 0 in J'_{i-1}.

It follows from the paragraph before the previous one that if $x = leaf_{i-1}(j) \in S$ for some index j, then the $str(x), str(z)$ match on the first r characters, i.e., $leaf_0(j) = z^0$. Since z^0 is a non-α-origin leaf, it follows that each j such that $leaf_{i-1}(j) \in S$ is in the family of some δ-index, i.e., $fam(j)$ is defined. There are two cases next. First, suppose $str(x)$ is periodic with period $per(e)$ for some $x \in S$. Then let j be such that $leaf_{i-1}(j) = x$ and let $j' = fam(j)$. By Corollary 6.6, $per(e) = per(j')$. Further, by Corollary 6.7, $j' + 2^{i-1}r - 1 \leq j + 2^{i-1}r - 1 < end(j')$. It follows that $leaf_{i-1}(j') = x$ and that x is a source, as required. Second, suppose that $str(x)$ has period greater than $per(e)$ for all $x \in S$. Let k_g denote the length of the longest prefix of $str(x_g)$ which has period $per(e)$, where $1 \leq g \leq h$. Clearly, $k_g \geq r$ for all g, $1 \leq g \leq h$; further, since $2\lceil \log^2 m \rceil \leq r$, $k_g - k_{g+1} = per(e)$. It follows that there exists some $x_g \in S$ such that $per(j)\lfloor \frac{k_g - r}{per(e)} \rfloor$ is divisible by the smallest multiple of $per(e)$ which is greater than $\lceil \log^2 m \rceil$. Let j be an index such that $leaf_{i-1}(j) = x_g$. By Corollary 6.6, $per(e) = per(fam(j))$. By Lemma 6.12, j is a γ-index. It follows that x_g is a source, as required. $\square$