Journal of Automata, Languages and Combinatorics **u** (v) w, x–y
© Otto-von-Guericke-Universität Magdeburg

COMPUTING ALL REPEATS USING SUFFIX ARRAYS [1]

F. Franěk, W. F. Smyth[2], Yudong Tang
Algorithms Research Group, Department of Computing and Software
McMaster University, Hamilton, Ontario, Canada
e-mail: {smyth, franek}@mcmaster.ca

ABSTRACT

We describe an algorithm that identifies *all* the repeating substrings (tandem, overlapping, and split) in a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Given the suffix arrays of $\boldsymbol{x}$ and of the reversed string $\widehat{\boldsymbol{x}}$, the algorithm requires $\Theta(n)$ time for its execution and represents its output in $\Theta(n)$ space, either as a reduced suffix array (called an NE array) or as a reduced suffix tree (called an NE tree). The output substrings $\boldsymbol{u}$ are ***nonextendible*** (NE); that is, any extension of some occurrence of $\boldsymbol{u}$ in $\boldsymbol{x}$, either to the left or to the right, yields a string ($\lambda\boldsymbol{u}$ or $\boldsymbol{u}\lambda$) that is unequal to the same extension of some other occurrence of $\boldsymbol{u}$. Thus the number of substrings output is the minimum required to identify all the repeating substrings in $\boldsymbol{x}$. The output can be used in a straightforward way to identify only repeating substrings that satisfy some proximity or minimum length condition.

Keywords: Repeats, suffix arrays, string algorithms, substring

1. Introduction

The computation of all the repeating substrings in a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a problem with various application areas, most notably data compression, cryptography, and computational biology. For repeating substrings that are ***tandem*** (that is, ***repetitions***), several $O(n \log n)$ algorithms [1, 4, 11] were discovered about 20 years ago; more recently, a repetitions algorithm [9] was published that, at least theoretically, executes in $\Theta(n)$ time in the common case that the alphabet is ***indexed*** — that is, treatable as a range of integers $1..\alpha \in O(n)$.

These successes with repetitions have encouraged researchers to seek algorithms that efficiently compute all repeating substrings $\boldsymbol{u}$, including in addition to tandem occurrences those that are ***split*** (of the form $\boldsymbol{uvu}$ for some nonempty $\boldsymbol{v}$) and ***overlapping*** (such as $\boldsymbol{u} = abaab$ in $\boldsymbol{x} = abaabaab$). The following definitions permit this problem to be stated more precisely:

[1]Supported in part by grants from the Natural Sciences and Engineering Research Council of Canada
[2]School of Computing, Curtin University, Perth, Australia

Definition 1 *A **repeat** in a string $\boldsymbol{x}$ is a tuple*

$$M_{\boldsymbol{x},\boldsymbol{u}} = (p;\ i_1, i_2, \ldots, i_r),\ r \geq 2,$$

where

$$\boldsymbol{u} = \boldsymbol{x}[i_1..i_1+p-1] = \boldsymbol{x}[i_2..i_2+p-1] = \cdots = \boldsymbol{x}[i_r..i_r+p-1].$$

*$\boldsymbol{u}$ is said to be a **repeating substring** of $\boldsymbol{x}$ and the **generator** of $M_{\boldsymbol{x},\boldsymbol{u}}$. As for repetitions, we call $p = |\boldsymbol{u}|$ the **period** of the repeat and r its **exponent**. If the tuple includes all the occurrences of $\boldsymbol{u}$ in $\boldsymbol{x}$, then $M_{\boldsymbol{x},\boldsymbol{u}}$ is said to be **complete** and is written $M^*_{\boldsymbol{x},\boldsymbol{u}}$.*

In its most general form, then, our task may be defined to be the computation of $M^*_{\boldsymbol{x},\boldsymbol{u}}$ for every repeating $\boldsymbol{u}$. This task is simplified by introducing the idea of "nonextendibility":

Definition 2 *A repeat $M_{\boldsymbol{x},\boldsymbol{u}} = (p;\ i_1, i_2, \ldots, i_r)$ is said to be **left-extendible** (respectively, **right-extendible**) if and only if*

$$(p+1;\ i_1-1, i_2-1, \ldots, i_r-1)\ \ \big(\textit{respectively, } (p+1;\ i_1, i_2, \ldots, i_r)\big)$$

*is a repeat. If $M_{\boldsymbol{x},\boldsymbol{u}}$ is neither left-extendible nor right-extendible, it is said to be **nonextendible**. We abbreviate these terms as **LE**, **RE** and **NE**, respectively.*

It is clear that it suffices to compute complete NE repeats: those that are extendible, either to left or right, do not need to be reported because any extendible repeating substring will be reported implicitly as a substring of the strings specified by an NE repeat.

In the last few years, two algorithms [2, 8] have been published that employ suffix trees to compute all the NE repeats in a given string $\boldsymbol{x}$ defined on an ordered alphabet. There are several standard algorithms [13, 15, 16] that compute suffix trees in $O(n \log n)$ worst-case time: indeed, if alphabet size is assumed to be constant, then the $\log n$ term can be defined away and the claim made that these algorithms are "linear" in string length. Perhaps more importantly, their behaviour is in fact very close to linear in the average case.

It is not usually recognized that Crochemore's repetitions algorithm [4] also belongs in this collection of $O(n \log n)$-time algorithms: by continuing the refinement process until all of the equivalence classes become singletons, all the information required for suffix tree construction becomes available. For strings defined on an indexed alphabet, there does exist a genuine $\Theta(n)$ worst-case time suffix tree construction algorithm [7], but it is very complicated and the space requirements are consequently so great that its theoretical asymptotic time advantage may well be eaten away by additional memory swaps to and from disk. In general, the main difficulty with suffix tree use has been the space requirement, many times that of the string itself, so that for strings of several tens of millions of characters, their use can become impractical. As a result, related data structures such as directed acyclic word graphs (DAWGs) [3, 5, 6] and suffix arrays [12] have been devised, that provide much of the functionality of suffix trees but use less space. On the other hand, a recent very careful implementation of McCreight's algorithm [10] has again made suffix trees competitive with these alternative suffix structures, both in terms of construction time and use of space.

In its simplest form, a suffix array $\boldsymbol{\lambda}_{\boldsymbol{x}} = \boldsymbol{\lambda}[1..n]$ of a given string $\boldsymbol{x}[1..n]$ is an array of integers such that for every $i \in 1..n$, $\boldsymbol{\lambda}[i] = j$ if and only if $\boldsymbol{x}[j..n]\$$ is the i^{th} smallest suffix of $\boldsymbol{x}\$$, where the sentinel letter \$ is by convention lexicographically smaller than any other letter in the alphabet. Thus $\boldsymbol{\lambda}_{\boldsymbol{x}}$ specifies the starting positions of the suffixes of $\boldsymbol{x}$ in ascending lexicographical order; for example, if

$$\begin{array}{rcccccccccccccccccccc} & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & \\ \boldsymbol{x} & = & a & b & c & a & a & b & c & a & b & a & c & c & a & b & a & a & c & b & \$ \end{array} \tag{1}$$

then

$$\begin{array}{rccccccccccccccccccc} & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ \boldsymbol{\lambda}_{\boldsymbol{x}} & = & 4 & 15 & 13 & 8 & 1 & 5 & 16 & 10 & 18 & 14 & 9 & 2 & 6 & 3 & 12 & 7 & 17 & 11 \end{array} \tag{2}$$

For effective use of suffix arrays, a second array $\boldsymbol{\pi}_{\boldsymbol{x}}$ is normally also computed that gives the length (lcp) of the longest common prefix (LCP) of adjacent entries in $\boldsymbol{\lambda}_{\boldsymbol{x}}$; in our example,

$$\begin{array}{rccccccccccccccccccc} & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ \boldsymbol{\lambda}_{\boldsymbol{x}} & = & 4 & 15 & 13 & 8 & 1 & 5 & 16 & 10 & 18 & 14 & 9 & 2 & 6 & 3 & 12 & 7 & 17 & 11 \\ \boldsymbol{\pi}_{\boldsymbol{x}} & = & . & 2 & 1 & 3 & 2 & 4 & 1 & 2 & 0 & 1 & 2 & 1 & 3 & 0 & 2 & 4 & 1 & 1 \end{array} \tag{3}$$

reflecting the fact that $\text{lcp}(4, 15) = 2$ (as aa is the longest common prefix of $\boldsymbol{x}[4..18]\$$ and $\boldsymbol{x}[15..18]\$$), $\text{lcp}(15, 13) = 1$ (as a is the longest common prefix of $\boldsymbol{x}[15..18]\$$ and $\boldsymbol{x}[13..18]\$$), and so on. In general, we set $\boldsymbol{\pi}[1] = 0$, while for every $i \in 2..n$,

$$\boldsymbol{\pi}[i] = \text{lcp}\big(\boldsymbol{\lambda}[i-1], \boldsymbol{\lambda}[i]\big). \tag{4}$$

Suffix arrays were introduced in [12], where an $O(n \log n)$-time and space-efficient algorithm is described for their calculation. For many problems — in particular, as we shall see, for the problem discussed in this paper — suffix arrays provide the same functionality that suffix trees make available, and with a very much reduced use of space. The initial drawback was that direct suffix array construction required, according to [12], three to ten times the time needed for construction of the corresponding suffix tree. Of course suffix arrays can be computed from suffix trees, but then the space advantage is lost. Again more recent work has redressed the balance somewhat: the revised suffix array construction algorithm of Sadakane [14] appears to be competitive in time requirement with the fastest suffix tree construction algorithms.

In this paper we describe an alternate approach to the calculation of all the NE repeats in a string $\boldsymbol{x}$ that can be used either with suffix trees or suffix arrays, depending on circumstances.

The new algorithm is based on the following easily-proved lemma:

Lemma 1 *Let $\widehat{\boldsymbol{x}}$ denote the reverse string $\boldsymbol{x}[n]\boldsymbol{x}[n-1]\cdots\boldsymbol{x}[1]$ of a given string $\boldsymbol{x}$. Then a repeat $M_{\boldsymbol{x},\boldsymbol{u}}$ is LE if and only if $M_{\widehat{\boldsymbol{x}},\widehat{\boldsymbol{u}}}$ is RE.* □

This result suggests a straightforward approach to computing all the NE repeats in $\boldsymbol{x}$:

- compute all the NRE (non-RE) repeats of $\boldsymbol{x}$ and all the NRE (non-RE) repeats of $\widehat{\boldsymbol{x}}$;
- compare the NRE repeats of $\boldsymbol{x}$ with the NRE repeats of $\widehat{\boldsymbol{x}}$ to identify those that are in both lists (the NE repeats).

It turns out that the NRE repeats of a string are identified by its suffix tree (as well as by its suffix array), a fact illustrated by reference to our example (1) in Figure 1. This figure displays a kind of stripped-down suffix tree in which we represent only the distinct prefixes of each suffix in $\boldsymbol{x}$. The internal (circular) nodes are recognizable as the lcp values contained in the array $\boldsymbol{\pi_x}$, while the leaf (square) nodes represent the starting positions i of the suffixes $\boldsymbol{x}[i..n]\$$, $i = 1, 2, \ldots, n$, as specified in the array $\boldsymbol{\lambda_x}$. As in every suffix tree, the lcp value at the root of each subtree applies to all the leaf nodes of that subtree; thus the lcp values also specify the complete repeats in $\boldsymbol{x}$ that are according to Definition 2 NRE. For example, the substrings

$$\boldsymbol{x}[7..10] = \boldsymbol{x}[12..15]$$

defined by the subtree rooted at the rightmost internal node 4 (see Figure 1) constitute a complete NRE repeat for $\boldsymbol{u} = caba$

$$M^*_{\boldsymbol{x},\boldsymbol{u}} = (4;\ 7, 12).$$

In Section 2 we describe an algorithm that computes the NE tree of $\boldsymbol{x}$, hence all the NE repeats in $\boldsymbol{x}$, in linear time from the suffix (NRE) trees of $\boldsymbol{x}$ and $\widehat{\boldsymbol{x}}$. Then Section 3 explains how this algorithm can be implemented to compute the NE array of $\boldsymbol{x}$, also in linear time, based on the suffix (NRE) arrays of $\boldsymbol{x}$ and $\widehat{\boldsymbol{x}}$. Section 4 briefly discusses extensions of these algorithms and open problems.

2. Computing the NE Tree

We suppose that the suffix trees, say $T_{\boldsymbol{x}}$ and $T_{\widehat{\boldsymbol{x}}}$ of $\boldsymbol{x}$ and $\widehat{\boldsymbol{x}}$ respectively, have already been computed, thereby identifying complete NRE repeats in $\boldsymbol{x}$ and complete NRE repeats in $\widehat{\boldsymbol{x}}$. From Lemma 1 we know that the NRE repeats in $\widehat{\boldsymbol{x}}$ identify NLE repeats in $\boldsymbol{x}$. For brevity we refer to the generator $\boldsymbol{u}$ of an NRE (respectively, NLE, NE) repeat $M_{\boldsymbol{x},\boldsymbol{u}}$ as an NRE (respectively, NLE, NE) repeating substring, while asking the reader to bear in mind that the terms NRE, NLE, NE really relate to the repeat that is a collection of repeating substrings.

Suppose that some nonzero lcp node (i.e. an internal node) $\widehat{p}$ in $T_{\widehat{\boldsymbol{x}}}$ has as child a position node (i.e. a leaf) $\widehat{\imath}$. Then in $\widehat{\boldsymbol{x}}$ there exists an NRE repeating substring

$$\widehat{\boldsymbol{u}} = \widehat{\boldsymbol{x}}[\widehat{\imath}..\widehat{\imath}+\widehat{p}-1]$$

that is the reverse of an NLE repeating substring

$$\boldsymbol{u} = \boldsymbol{x}\big[n-(\widehat{\imath}+\widehat{p}-1)+1..n-\widehat{\imath}+1\big]$$

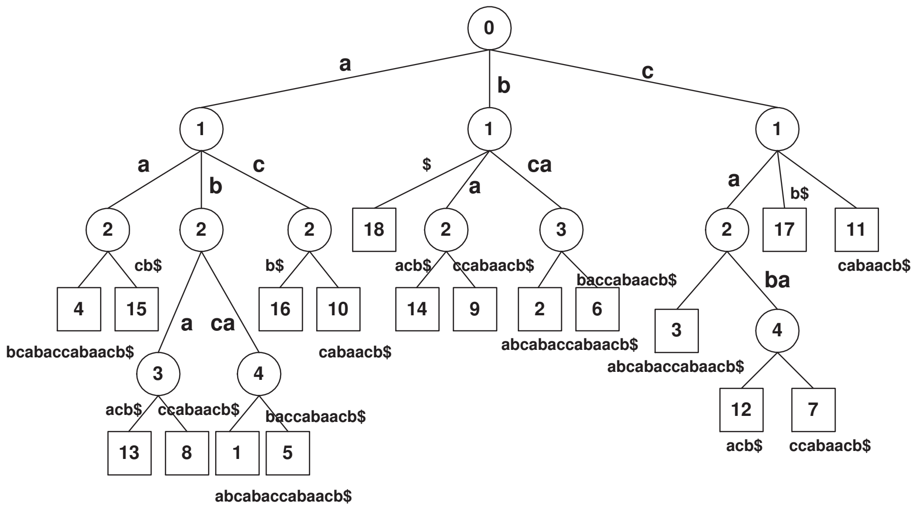

Figure 1: Suffix (NRE) tree $T_{\boldsymbol{x}}$ for $\boldsymbol{x} = \begin{array}{cccccccccccccccccc} {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} & {\scriptstyle 13} & {\scriptstyle 14} & {\scriptstyle 15} & {\scriptstyle 16} & {\scriptstyle 17} & {\scriptstyle 18} \\ a & b & c & a & a & b & c & a & b & a & c & c & a & b & a & a & c & b \end{array}$

in $\boldsymbol{x}$. Thus the assignment

$$i \leftarrow n-(\widehat{\imath}+\widehat{p}-2) \tag{5}$$

identifies one start position i of an NLE repeating substring $\boldsymbol{u}$ in $\boldsymbol{x}$ of length $\widehat{p}$. Observe that in fact i is also a start position of one of a set of NLE repeating substrings of every length $j \in 1..\widehat{p}$; thus if in $T_{\boldsymbol{x}}$ we find for some i the largest value of $\widehat{p}$ such that i is one of a collection of substrings of length $\widehat{p}$ that are both NRE and NLE, then every parent of $\widehat{p}$ in $T_{\boldsymbol{x}}$ will also identify collections of substrings that are both NRE and NLE.

Let us say that a substring $\boldsymbol{u}$ of $\boldsymbol{x}$ is a ***maximal NE repeating substring*** of $\boldsymbol{x}$ if and only if

- $\boldsymbol{u}$ occurs at least twice in $\boldsymbol{x}$;
- $\boldsymbol{u}$ is not a proper substring of any repeating substring of $\boldsymbol{x}$.

The following result, again easily proved, then tells us that a maximal NE repeating substring must be identifiable in both $T_{\boldsymbol{x}}$ and $T_{\widehat{\boldsymbol{x}}}$:

Lemma 2 *If* $\boldsymbol{u} = \boldsymbol{x}[i..i+p-1]$ *is a maximal* NE *repeating substring of* $\boldsymbol{x}$*, then the position node* i *occurs as a child of the lcp node* p *in* $T_{\boldsymbol{x}}$*, and* $n-(i-p-2)$ *occurs as a child of* p *in* $T_{\widehat{\boldsymbol{x}}}$. $\square$

Recall the observation made above that every internal (lcp) node in $T_{\boldsymbol{x}}$ that is an ancestor of an NE repeating substring must itself be the root of a subtree of $T_{\boldsymbol{x}}$ whose leaf (position) nodes are the repeating substrings of a complete NE repeat. This lemma therefore provides us with a simple strategy for identifying all NE repeating

Algorithm 1 (Compute the NE Tree)

— *Given the suffix trees* $T_{\boldsymbol{x}}$ & $T_{\widehat{\boldsymbol{x}}}$, *compute the* NE *tree of* $\boldsymbol{x}[1..n]$

(1) traverse $T_{\boldsymbol{x}}$ to create a table POINTER[i] that for each
position i in $\boldsymbol{x}$ points to the corresponding node in $T_{\boldsymbol{x}}$

(2) **for** every lcp node p in $T_{\boldsymbol{x}}$ **do**
NE[p] $\leftarrow$ `FALSE`

(3) **for** every parent-child pair $(\widehat{p},\widehat{\imath})$ in $T_{\widehat{\boldsymbol{x}}}$ $(\widehat{p} > 0)$ **do**
— *Here use* POINTER[i]*:*
if $i = n-(\widehat{\imath}+\widehat{p}-2)$ is a child of lcp node $\widehat{p}$ in $T_{\boldsymbol{x}}$ **then**
while not NE[$\widehat{p}$] **do**
NE[$\widehat{p}$] $\leftarrow$ `TRUE`
if $\widehat{p} \neq 0$ **then**
$\widehat{p} \leftarrow$ parent of $\widehat{p}$ in $T_{\boldsymbol{x}}$

(4) traverse $T_{\boldsymbol{x}}$ deleting every subtree rooted
at an lcp node p for which NE[p] = `FALSE`

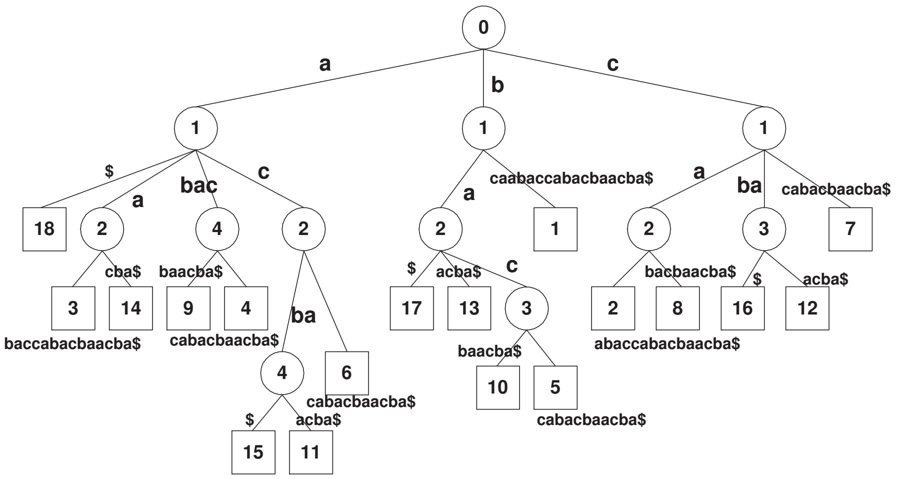

Figure 2: Suffix tree $T_{\widehat{\boldsymbol{x}}}$ for $\widehat{\boldsymbol{x}} = \begin{array}{cccccccccccccccccc} \scriptstyle 1 & \scriptstyle 2 & \scriptstyle 3 & \scriptstyle 4 & \scriptstyle 5 & \scriptstyle 6 & \scriptstyle 7 & \scriptstyle 8 & \scriptstyle 9 & \scriptstyle 10 & \scriptstyle 11 & \scriptstyle 12 & \scriptstyle 13 & \scriptstyle 14 & \scriptstyle 15 & \scriptstyle 16 & \scriptstyle 17 & \scriptstyle 18 \\ b & c & a & a & b & a & c & c & a & b & a & c & b & a & a & c & b & a \end{array}$

substrings: just find the maximal ones (largest value of p), then locate all their ancestors in $T_{\boldsymbol{x}}$. Algorithm 1 provides an outline of how this can be accomplished.

The algorithm is expressed in terms of four steps. Step (1) is a traversal of $T_{\boldsymbol{x}}$ that sets up a table enabling each position node i in $T_{\boldsymbol{x}}$ to be accessed later in constant time. In Step (2) a Boolean variable corresponding to each marked node p in $T_{\boldsymbol{x}}$ is initialized to `FALSE`, indicating that no terminal nodes in the subtree rooted at p have currently been identified as NE. Step (3) processes every parent-child pair $(\widehat{p}, \widehat{\imath})$ in $T_{\widehat{\boldsymbol{x}}}$, testing to determine whether or not the equivalent parent-child pair $\big(\widehat{p}, n-(\widehat{\imath}+\widehat{p}-2)\big)$ exists in $T_{\boldsymbol{x}}$; if so, then the marked node $\widehat{p}$ and all its ancestors in $T_{\boldsymbol{x}}$ must be NE — accordingly, until an ancestor is found that is already NE, $\widehat{p}$ and its ancestors in $T_{\boldsymbol{x}}$ are identified as NE. A final step traverses $T_{\boldsymbol{x}}$ to eliminate all subtrees rooted at any node p for which NE$[p]$ = `FALSE`; the remaining tree, $T_{\boldsymbol{x}}^{\mathrm{NE}}$, is the NE tree of $\boldsymbol{x}$.

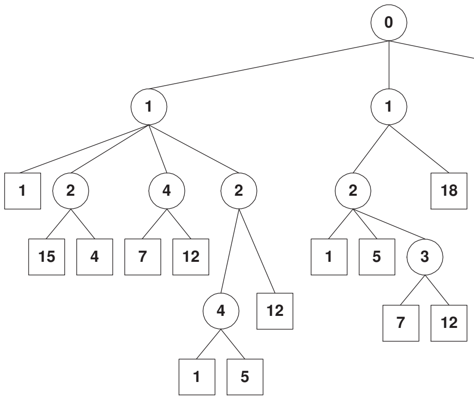

Figure 3: $T_{\widehat{\boldsymbol{x}}}$ with leaves labelled by transformation $i = n-(\widehat{\imath}+\widehat{p}-2)$ (cf. Figure 2)

Each of the steps (1), (2), (4) of Algorithm 1 is a traversal of an NRE tree that requires $O(1)$ time at each node, hence $O(n)$ time in total. Step (3) is another tree traversal, but with slightly more complex processing: since the **if** statement makes use of the POINTER array, it too requires only $O(1)$ time at each execution, hence also $O(n)$ time overall. The **while** loop in Step (3) causes the NE variable for at most n marked nodes to be set `TRUE`, and tests the current setting of at most n marked nodes for which NE is already `TRUE` — thus the total time consumed in the while loop is $O(n)$. We have established:

Theorem 1 *Given the suffix trees $T_{\boldsymbol{x}}$ and $T_{\widehat{\boldsymbol{x}}}$ for $\boldsymbol{x}[1..n]$, Algorithm 1 correctly computes the* NE *tree $T_{\boldsymbol{x}}^{\mathrm{NE}}$ using $O(n)$ time and $\Theta(n)$ additional space.* $\Box$

Figure 2 displays the suffix tree $T_{\widehat{\boldsymbol{x}}}$ for our example string (1), and Figure 3 displays the tree modified so that position nodes $\widehat{\imath}$ are replaced by $i = n-(\widehat{\imath}+\widehat{p}-2)$. Note that only positions $1, 4, 5, 7, 10, 12, 15, 16, 18$ give rise to NLE repeats; thus the NE tree reduces to the subtree of $T_{\boldsymbol{x}}$ depicted in Figure 4. The complete NE repeats of (1) are thus seen from Figure 4 to be all the occurrences of $a, aa, ab, abca, ac, b, c, ca, caba$. It is easy to check that these substrings repeat in (1): a, aa, ab, ac, abc (but always as a substring of $abca$), $abca, b, ba$ (but always as a substring of $caba$), bc (but always as a substring of $abca$), bca (but always as a substring of $abca$), c, ca, cab (but always as a substring of $caba$), and $caba$. Hence we got them all.

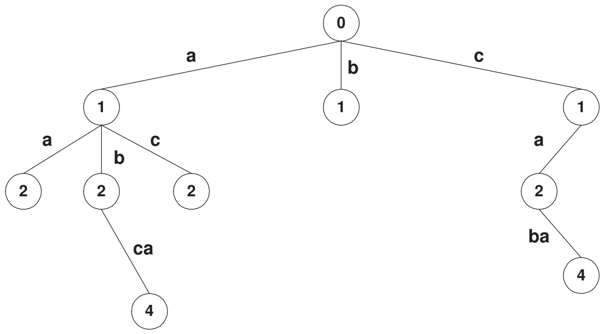

Figure 4: The NE tree for $\boldsymbol{x}$ (cf. Figure 1)

In general, the occurrences of NE repeats, whether all of them or selected according to various criteria, can be located and output in time proportional to their number by traversal of the NE tree. Such problems are also discussed in [2]. Another quite different approach to the computation of nonextendible repeats (though still making use of suffix trees) is described in [8].

3. Computing the NE Array

We provide here an outline of a version of the algorithm that avoids the use of suffix trees completely: the entire calculation is carried out based on suffix arrays (both $\boldsymbol{\lambda_x}$ and $\boldsymbol{\pi_x}$), and its end-product is not an NE tree, but rather an NE array. Since the suffix array of $\boldsymbol{x}[1..n]$ requires only $2n$ computer words for its storage, this revision yields significant saving in space.

For every $j \in 1..n$, let $p_j = \boldsymbol{\pi}[j]$. It then follows from (4) that for every $j \in 2..n$,

if $p_j > p_{j-1}$ **then**
  the internal node identified by p_j is a descendant of the
  internal node identified by p_{j-1} in the suffix tree
elsif $p_j < p_{j-1}$ **then**
  internal node labelled by p_j is an ancestor of the internal
  node labelled by p_{j-1} in the suffix tree
else
  p_j and p_{j-1} identify the same internal node in the suffix
  tree

Since the suffix array contains the same lcp information that is provided by the suffix tree, these relationships imply that the suffix array must be decomposable into subarrays, each of which specifies nodes that lie on a single path from the root to a terminal node in the suffix tree. For example, the lcp values in the array (3) can be separated into seven subarrays

$$0,2,\underline{1} \mid \underline{1},3,\underline{2} \mid \underline{2},4,\underline{1} \mid \underline{1},2,\underline{0} \mid \underline{0},1,2,\underline{1} \mid \underline{1},3,\underline{0} \mid \underline{0},2,4,1,1 \qquad (6)$$

corresponding to the seven paths in the suffix tree of Figure 1. Here the repeated lcp values $\underline{0}$, $\underline{1}$, and $\underline{2}$ identify internal nodes at which new paths begin. These nodes are called ***branch nodes***, marking the point at which a new path diverges from the preceding one. (Note that the paths correspond to the depth first traversal of the suffix tree that produces the suffix array).

These ideas can be made more precise as follows:

Definition 3 *An **I-run** $I_{j,h}$ in a suffix array is a sequence of $h \geq 2$ lcp values $p_j, p_{j+1}, \ldots, p_{j+h-1}$, not all equal, such that*

(a) *either $j = 1$ or else $p_{j-1} > p_j$;*

(b) $p_j \leq p_{j+1} \leq \cdots \leq p_{j+h-1}$*;*

(c) *either $j+h-1 = n$ or else $p_{j+h-1} > p_{j+h}$.*

Thus an I-run is a maximum-length sequence of nondecreasing lcp values in the suffix array; similarly, a ***D-run*** $D_{j,h}$ is a maximum-length sequence of nonincreasing lcp values. Using these definitions, we can now characterize paths in the suffix tree in terms of runs in the suffix array:

Lemma 3 *Every sequence $I_{j,h}D_{j+h-1,h'}$ or $I_{j,h}$, where $j+h-1 = n$, in the suffix array identifies nodes that lie on a single path in the suffix tree.*

Proof. Since $p_1 = 0$, it follows from Definition 3(a)-(b) that either there are no runs in the suffix array ($p_j = 0$ for every $j \in 1..n$) or else the sequence of runs begins with an I-run $I_{1,h}$. Observe that Definition 3(c) ensures that every I-run is followed by a D-run or else by the end of the array; similarly, every D-run must be followed by an I-run or else by the end of the array.

Now consider an I-run $I_{j,h}$. Since $p_{j+(t-1)} \leq p_{j+t}$ for every $t \in 1..h-1$, it follows that $I_{j,h}$ determines a sequence of descendants of p_j that lie on a single path in the suffix tree. If in fact $i+h-1 = n$, then $I_{j,h}$ is in itself the final path identified in the suffix array. If not, however, then $I_{j,h}$ is followed by a D-run $D_{j+h-1,h'}$, where $p_{(j+h-1)+(t-1)} \leq p_{(j+h-1)+t}$ for every $t \in 1..h'-1$. These lcp values determine a sequence of ancestors of p_{j+h-1}, all on the same path from the root already identified by $I_{j,h}$. The subsequent I-run $I_{j+h+h'-2,h''}$, if it exists, of course specifies descendants of the branch node $p_{j+h+h'-2}$ that lie on a path distinct from the path specified by $I_{j,h}$. □ □

We see then that the ID-runs in the suffix array essentially decompose the internal (lcp) nodes of the suffix array into paths. Adjacent pairs of runs in the array identify two paths in the tree that have exactly one node (a branch node) in common. Observe also that for each ID-run, the minimum node in the corresponding path must be either the first node of the I-run or the last node of the D-run.

Since a branch node is *both* the last node of a D-run *and* the first node of an I-run, the values of pairs of adjacent branch nodes in the suffix array therefore determine the way in which two paths in the suffix tree are related. If we include the first lcp value ($p_1 = 0$) in the suffix array as a branch node, and denote consecutive branch nodes by b_1 and b_2, respectively, we see that there are only three possible relationships between the corresponding paths, as shown in Figure 5. Notice in particular that in each of these three cases, future paths can be added only as descendants of b_2 on the current path or as descendants of ancestors of b_2 in the tree. No further change can be made to the path rooted at b_2 that was specified by the previous path containing b_1.

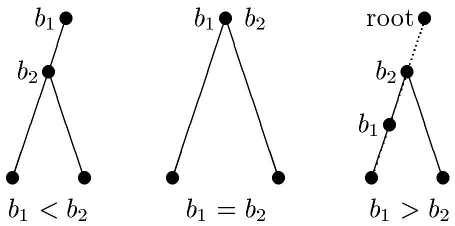

Figure 5: Branch nodes b_1 and b_2 of adjacent paths (ID-runs)

This discussion puts us in a position to describe an approach to the calculation of NE repeats that is based on suffix arrays only. As before we denote the position and lcp arrays for $\boldsymbol{x}$ by $\boldsymbol{\lambda_x}$ and $\boldsymbol{\pi_x}$, respectively; for $\widehat{\boldsymbol{x}}$ the corresponding arrays are $\widehat{\boldsymbol{\lambda_x}}$ and $\widehat{\boldsymbol{\pi_x}}$. We also make use of a LOC array that gives for each position node $\widehat{\imath}$ in $T_{\widehat{\boldsymbol{x}}}$ its location in $\widehat{\boldsymbol{\lambda_x}}$; this enables us to find any position in $\widehat{\boldsymbol{\lambda_x}}$ in constant time. A bit vector NE$[1..n]$ specifies for each $j \in 1..n$ whether (`TRUE`) or not (`FALSE`) the lcp node $\boldsymbol{\pi}[j]$ is also NLE. Finally, in order to identify the beginning and end of ID paths in $\boldsymbol{\pi_x}$, we use the ID patterns (Lemma 3) to compute a BRANCH array that includes, in addition to the branch points defined above, also the first and last positions in $\boldsymbol{\pi_x}$. An outline of the processing, more complex than the tree-based calculation but not necessarily more time-consuming, is displayed as Algorithm 2.

Step (1) of the algorithm is a straightforward **for** loop that initializes the LOC array and the bit vector NE. Step (2) computes the BRANCH array based on Lemma 3 and a linear scan of $\boldsymbol{\pi}[1..n]$ (the size of BRANCH is determined during processing by the number of branch nodes – for an illustration see (6)). Step (3) is another **for** loop that first identifies all the matches of (p,i) pairs in the NRE arrays for $\boldsymbol{x}$ and $\widehat{\boldsymbol{x}}$ (the parent of the leaf labelled by $\boldsymbol{\lambda}[j]$ is the node labelled by the bigger value of $\{\boldsymbol{\pi}[j], \boldsymbol{\pi}[j{+}1]\}$), then begins to deal with the more complicated problem of ensuring that every location in $\boldsymbol{\pi}$ is marked `TRUE` if it corresponds to a node in $T_{\boldsymbol{x}}$ that is an ancestor of some NE node. The simple **while** loop of Algorithm 1 is replaced by three distinct processing threads:

- A record is kept of the lcp value p_0 of the node whose NE value was most recently set `TRUE`. Whenever a node of smaller lcp value is found, its NE value is also set `TRUE`.
- For every path (identified by adjacent branch nodes), the node of maximum lcp for which NE has been set `TRUE` is determined, and then NE is set `TRUE` for every node in the path whose lcp value is less than this maximum.

Algorithm 2 (Compute the NE Array)

— *Given the suffix (NRE) arrays of* $\boldsymbol{x} = \boldsymbol{x}[1..n]$ *and* $\widehat{\boldsymbol{x}}$,
compute the NE *array of* $\boldsymbol{x}$

(1) **for** $j \leftarrow 1$ **to** n **do**
    $\mathrm{LOC}\big[\widehat{\boldsymbol{\lambda}}[j]\big] \leftarrow j$; $\mathrm{NE}[j] \leftarrow$ `FALSE`

(2) compute $\mathrm{BRANCH}[1..b^*]$, where for $b \in 1..b^*$, $\mathrm{BRANCH}[b]$
  is the position of the b^{th} branch node in $\boldsymbol{\pi_x}$
  (include positions $j = 1$ and $j = n$ as branch nodes)

(3) $b \leftarrow 1$; $p_0 \leftarrow 0$
  for $j \leftarrow 1$ **to** n **do**
    determine j' such that $(p, i) = \big(\boldsymbol{\pi}[j'], \boldsymbol{\lambda}[j]\big)$ is a
      parent-child pair in the suffix tree of $\boldsymbol{x}$
    if $p > 0$ **then**
      if $p < p_0$ **then**
        $\mathrm{NE}[j'] \leftarrow$ `TRUE`; $p_0 \leftarrow p$
      else
        $\widehat{\imath} \leftarrow n-(i+p-2)$; $\widehat{\jmath} \leftarrow \mathrm{LOC}[\widehat{\imath}]$
        determine $\widehat{j'}$ such that $(\widehat{p}, \widehat{\imath}) = \big(\widehat{\boldsymbol{\pi}}[\widehat{j'}], \widehat{\boldsymbol{\lambda}}[\widehat{\jmath}]\big)$ is a
          parent-child pair in the suffix tree of $\widehat{\boldsymbol{x}}$
        if $p = \widehat{p}$ **then**
          $\mathrm{NE}[j'] \leftarrow$ `TRUE`; $p_0 \leftarrow p$

      if $j = \mathrm{BRANCH}[b+1]$ **then**
        determine $j^* \in \mathrm{BRANCH}[b]..\mathrm{BRANCH}[b+1]$ such that
          $p^* = \boldsymbol{\pi}[j^*]$ is the maximum lcp
          for which $\mathrm{NE}[j^*] =$ `TRUE`
        for every $j' \in \mathrm{BRANCH}[b]..\mathrm{BRANCH}[b+1]$ **do**
          if $\boldsymbol{\pi}[j'] \leq p^*$ **then**
            $\mathrm{NE}[j'] \leftarrow$ `TRUE`
        $b \leftarrow b+1$

(4) process the paths identified by BRANCH in reverse order,
  setting NE true for ancestors of p^* as in Step (3)

- The same processing is repeated in Step (4) for all paths, but in reverse order, to ensure that `TRUE` values of NE are propagated to all paths further up the tree.

Each of the two sections of Step (3) requires $\Theta(n)$ time, as does Step (4). Finally the NE array is determined by all those positions j in $\boldsymbol{\pi_x}$ for which $\mathrm{NE}[j]=$ `TRUE`. Hence

Theorem 2 *Given the suffix (NRE) arrays corresponding to* $\boldsymbol{x}[1..n]$ *and* $\widehat{\boldsymbol{x}}$*, Algorithm 2 correctly computes the* NE *array using* $\Theta(n)$ *time and* $\Theta(n)$ *additional space.* □

We remark that the NE array can if desired be used to compute the corresponding NE tree in linear time.

Let us illustrate the ideas of Algorithm 2 on our example string $\boldsymbol{x}$ given in (1). The suffix array for $\boldsymbol{x}$ is given in (3). The paths separations of this suffix array is given in (6). The reverse string $\widehat{\boldsymbol{x}}$ and its suffix array are:

$$
\begin{array}{rccccccccccccccccccc}
 & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} & {\scriptstyle 13} & {\scriptstyle 14} & {\scriptstyle 15} & {\scriptstyle 16} & {\scriptstyle 17} & {\scriptstyle 18} & \\
\widehat{\boldsymbol{x}} = & b & c & a & a & b & a & c & c & a & b & a & c & b & a & a & c & b & a & \$ \\
\boldsymbol{\lambda} = & 18 & 3 & 14 & 9 & 4 & 15 & 11 & 6 & 17 & 13 & 10 & 5 & 1 & 2 & 8 & 16 & 12 & 7 & \\
\boldsymbol{\pi} = & . & 1 & 2 & 1 & 4 & 1 & 4 & 2 & 0 & 2 & 2 & 3 & 1 & 0 & 2 & 1 & 3 & 1 &
\end{array} \tag{7}
$$

First we identify the pairs (parent,leaf) for every leaf in (7). The parent of $\boldsymbol{\lambda}[i]$ is $\boldsymbol{\pi}[j]$ for $j \in \{i, i+1\}$ so that $\boldsymbol{\pi}[j] = \max\{\boldsymbol{\pi}[i], \boldsymbol{\pi}[i+1]\}$. Hence we get (1,18), (2,3), (2,14), (4,9), (4,4), (4,15), (4,11), (2,6), (2,17), (2,13), (3,10), (3,5), (1,5), (2,2), (2,8), (3,16), (3,12), and (1,7). Now we can perform the transformation $i = n - (\widehat{p} + \widehat{\imath} - 2)$ obtaining $18 \to 1$, $3 \to 15$, $14 \to 4$, $9 \to 7$, $4 \to 12$, $15 \to 1$, $11 \to 5$, $6 \to 12$, $17 \to 1$, $13 \to 5$, $10 \to 7$, $5 \to 12$, $1 \to 18$, $2 \to 16$, $8 \to 10$, $16 \to 1$, $12 \to 5$, $7 \to 12$. Therefore we conclude that we need only the "paths" from the suffix array (3) that lead to the leaves 1, 4, 5, 7, 10, 12, 15, 16, and 18. By "tracing" the "path" in (6) we get

$$
\begin{array}{rcccccccccccccccccc}
 & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} & {\scriptstyle 13} & {\scriptstyle 14} & {\scriptstyle 15} & {\scriptstyle 16} & {\scriptstyle 17} & {\scriptstyle 18} \\
\mathrm{NE} = & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T}
\end{array}
$$

We see that NE together with the suffix array (3) determine the same "tree" as in Figure 4. How can we "read" the result?

$$
\begin{array}{rcccccccccccccccccc}
 & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} & {\scriptstyle 13} & {\scriptstyle 14} & {\scriptstyle 15} & {\scriptstyle 16} & {\scriptstyle 17} & {\scriptstyle 18} \\
\mathrm{NE} = & \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} \\
\boldsymbol{\lambda_x} = & 4 & 15 & 13 & 8 & 1 & 5 & 16 & 10 & 18 & 14 & 9 & 2 & 6 & 3 & 12 & 7 & 17 & 11 \\
\boldsymbol{\pi_x} = & . & 2 & 1 & 3 & 2 & 4 & 1 & 2 & 0 & 1 & 2 & 1 & 3 & 0 & 2 & 4 & 1 & 1
\end{array}
$$

The complete NE repeats are aa (the first 2 characters of $\boldsymbol{\lambda}[15]$), a (the first character if $\boldsymbol{\lambda}[13]$), ab (the first 2 characters of $\boldsymbol{\lambda}[1]$), $abca$ (the first 4 characters of $\boldsymbol{\lambda}[5]$), a (the first character if $\boldsymbol{\lambda}[16]$), ac (the first 2 characters of $\boldsymbol{\lambda}[10]$), b (the first character if $\boldsymbol{\lambda}[14]$), ca (the first 2 characters of $\boldsymbol{\lambda}[12]$), $caba$ (the first 4 characters of $\boldsymbol{\lambda}[7]$), c (the first character if $\boldsymbol{\lambda}[17]$), and c (the first character if $\boldsymbol{\lambda}[11]$). Therefore, we identified all complete NE repeats (cf. the example after Theorem 1).

4. Extensions & Open Problems

In this paper we have described an algorithm — implemented in two quite different ways — that, given suffix trees or suffix arrays of the strings $\boldsymbol{x}$ and $\widehat{\boldsymbol{x}}$, computes all the complete NE repeats in linear time while representing them in linear space. To restrict the output to repeats that lie within a specified substring of $\boldsymbol{x}$, it is necessary only to ignore values of $\boldsymbol{\lambda}[i]$ (position values) that lie outside the confines of that substring; similarly, in order to consider only repeats above a certain length ℓ, it suffices to consider only subtrees whose roots contain lcp values $\boldsymbol{\pi}[i]$ greater than ℓ.

We remark also that since the order of output is determined by the structure of the suffix tree/array, it is therefore not straightforward to output repeating substrings in the natural order of their left-to-right occurrence in $\boldsymbol{x}$. We leave this as an open problem.

5. Acknowledgement

The authors would like to express their gratitude to the referees of the Journal of Automata, Computing, and Languages for their knowledgeable and constructive comments that much improved the paper.

References

[1] A. Apostolico, F.P. Preparata, Optimal off-line detection of repetitions in a string, *TCS* **22** (1983), 297–315.

[2] G.S. Brodal, R.B. Lyngso, C.N.S. Pedersen, J. Stoye, Finding maximal pairs with bounded gap, *J. Discrete Algs.* **1** (2000), 77–103.

[3] M.T. Chen, J. Seiferas, Efficient and elegant subword-tree construction. In: A. Apostolico, Z. Galil (eds.), *Combinatorial Algorithms on Words (NATO ASI Series F12)*, Springer-Verlag, 1985, 97–107.

[4] M. Crochemore, An optimal algorithm for computing the repetitions in a word, *IPL* **12–5** (1981), 244–250.

[5] M. Crochemore, J. Holub, On the implementation of compact DAWGs, *Proc. Seventh Internat. Conf. Implementation and Application of Automata*, University of Tours, Tours, France, July 2002, 293–298.

[6] M. Crochemore, R. Vérin, Direct construction of compact directed acyclic word graphs. In: *Proc. Eighth Annual Symp. Combinatorial Pattern Matching*, LNCS **1264**, Springer-Verlag, 1997, 116–129.

[7] M. Farach, Optimal suffix tree construction with large alphabets, *Proc.* 38^{th} *Annual IEEE Symp. Foundations of Computer Science*, 1997, 137–143.

[8] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.

[9] R. Kolpakov, G. Kucherov, On maximal repetitions in words, *J. Discrete Algs.* **1** (2000), 159–186.

[10] S. Kurtz, Reducing the space requirement of suffix trees, *Software — Practice and Experience 29–13*, 1999, 1149–1171.

[11] M.G. Main, R.J. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, *J. Algs.* **5** (1984), 422–432.

[12] U. Manber, G.W. Myers, Suffix arrays: a new method for on-line string searches, *SIAM J. Comput.* **22–5** (1993), 935–948.

[13] E.M. McCreight, A space-economical suffix tree construction algorithm, *JACM* **32–2** (1976), 262–272.

[14] K. Sadakane, A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation, *Proc. IEEE Data Compression Conference*, 1998, 129–138.

[15] E. Ukkonen, Constructing suffix trees on-line in linear time, Vol. I of *Proc. IFIP 92*, 1992, 484–492.

[16] P. Weiner, Linear pattern matching algorithms, *Proc. 14th Annual IEEE Symp. Switching and Automata Theory*, 1973, 1–11.