Faster Text Fingerprinting

Roman Kolpakov[1]* Mathieu Raffinot[2]

[1] Liapunov French-Russian Institute, Lomonosov Moscow State University, Moscow, Russia, foroman@mail.ru
[2] CNRS, Poncelet Laboratory, Independent University of Moscow, 11 street Bolchoï Vlassievski, 119 002 Moscow, Russia, mathieu@raffinot.net

Abstract: Let $s = s_1..s_n$ be a text (or sequence) on a finite alphabet Σ. A fingerprint in s is the set of distinct characters contained in one of its substrings. Fingerprinting a text consists of computing the set $\mathcal{F}$ of all fingerprints of all its substrings. A fingerprint, $f \in \mathcal{F}$, admits a number of maximal locations $\langle i, j\rangle$ in S, that is the alphabet of $s_i..s_j$ is f and s_{i-1}, s_{j+1}, if defined, are not in f. The set of maximal locations is $\mathcal{L}$, $|\mathcal{L}| \le n|\Sigma|$. Two maximal locations $\langle i, j\rangle$ and $\langle k, l\rangle$ such that $s_i..s_j = s_k..s_l$ are named *copies* and the quotient of $\mathcal{L}$ according to the copy relation is named $\mathcal{L}_C$. The faster algorithm to compute all fingerprints in s runs in $O(n + |\mathcal{L}| \log |\Sigma|)$ time. We present a quite always faster $O((n + |\mathcal{L}_C|) \log |\Sigma|)$ algorithm.

1 Introduction

We consider a finite ordered alphabet Σ and $s = s_1..s_n$ a sequence of n letters, $s_i \in \Sigma$. The set of all sequences over Σ is denoted Σ^*. The rank of each letter α in Σ is given by $f_\Sigma(\alpha)$ that ranges between 0 and $|\Sigma| - 1$. A sequence $v \in \Sigma^*$ is a factor or substring of s if $s = uvw$. The fingerprint $C(s)$ of a sequence s is the set of distinct letters in s. By extension, $C_s(i, j)$ is the set of distinct letters in $s_i..s_j$. A fingerprint is represented below by a binary table of F of size $|\Sigma|$. If s contains the character α, $F[f_\Sigma(\alpha)] \leftarrow 1$, otherwise $F[f_\Sigma(\alpha)] \leftarrow 0$.

Definition 1. *Let $\mathcal{C}$ be a set of letters of Σ. A maximal location of $\mathcal{C}$ in $s = s_1..s_n$ is an interval $[i, j]$, $1 \le i \le j \le n$, such that*
(1) $C_s(i, j) = \mathcal{C}$; (2) *if* $i > 1, s_{i-1} \notin C_s(i, j)$; (3) *if* $j < n, s_{j+1} \notin C_s(i, j)$
This maximal location is denoted $\langle i, j\rangle$.

We denote by $\mathcal{F}$ the set of distinct fingerprints and by $\mathcal{L}$ the set of maximal locations of all fingerprints of $\mathcal{F}$.

Definition 2. *Two maximal locations $\langle i, j\rangle$ and $\langle k, l\rangle$ of $s = s_1..s_n$ are* copies *if $s_i..s_j = s_k..s_l$.*

The "copy" relation is obviously an equivalence relation. We denote $\mathcal{L}_C$ the set of equivalence classes. Let $q \in \mathcal{L}_C$ and $\langle i, j\rangle$ a maximal location in q, we denote $\mathrm{st}_s(q)$ the string $s_i..s_j$. Table 1 shows an example of a copy relation.

In this paper, given a sequence s, we are interested in the following algorithmic problem:

+ Compute the set $\mathcal{F}$ of all fingerprints in s

* This work is supported by the Russian Foundation for Fundamental Research (Grant 05-01-00994) and the program of the President of the Russian Federation for supporting of young researchers (Grant MD-3635.2005.1).

This problem has many applications in information retrieval, computational biology and natural language processing [1]. The input alphabet Σ is considered to be the alphabet of the input sequence, thus $|\Sigma| \leq n$.

The problem has first be considered in [1] in which they presented a $O(n|\Sigma| \log n \; \log|\Sigma|)$ algorithm. This complexity has been improved to $\Theta(\min\{n|\Sigma| \log|\Sigma|, n^2\})$ time in [4]. The bound $\Theta(n|\Sigma|\log|\Sigma|)$ is that of the last algorithm in [4]. The $\Theta(n^2)$ bound is obtained using the first algorithm of [4], although this algorithm was first presented by Didier with $O(n^2 \log n)$ and $\Omega(n^2)$ time complexities in [3]. The $\log n$ gain between these two versions has been obtained using a lowest common ancestor algorithm (LCA). Surprisingly enough, these complexities were independent of the sizes of $\mathcal{F}$ and $\mathcal{L}$, although many sequence families have few fingerprints or few maximal locations. We thus proposed in [7] a new algorithm running in $O((n + |\mathcal{L}|)\log|\Sigma|)$ time. We improved it later to $O(n + |\mathcal{L}|\log|\Sigma|)$ time in [8]. As $|\mathcal{L}| \leq n|\Sigma|$, this algorithm is, at worst, as efficient as the last algorithm of [4], but much faster on many sequence families. However, in our will to deeply understand the problem, a question arises: what are the best parameters for this problem ? This paper is a new step toward the answer.

We present below an algorithm running in $O((n + |\mathcal{L}_C|)\log|\Sigma|)$ time, quite always faster than the previous algorithm because it depends of $\mathcal{L}_C$ instead of $\mathcal{L}$. Note that the number $|\mathcal{L}_C|$ can be significantly less than $|\mathcal{L}|$. As an example, we can consider the word w_k over the alphabet $\Sigma_k = \{a_1, a_2, \ldots, a_k\}$ which is defined in the following inductive way: $w_1 = a_1$ and $w_k = w_{k-1}(a_1 a_2 \ldots a_k)^k$ for $k > 1$. For this word we have $|w_k| = \frac{1}{6}k(k+1)(2k+1)$, $|\mathcal{L}| = \frac{1}{12}k(3k^3 + 2k^2 - 9k + 16) = \Theta(|w_k|^{4/3})$, and $|\mathcal{L}_C| = \frac{1}{6}k(k^2+5) = \Theta(|w_k|)$. Thus, in this case $|\mathcal{L}_C| = o(|\mathcal{L}|)$ as $k \to \infty$.

Following the previous approaches, our algorithm improve a naming technique introduced in [6], adapted to the fingerprint problem in [1] and then successively improved in [4] and in [7]. The paper is organized as follows. In Section 2 we present a new structure called "participation tree" that is built from the

Class q	Maximal locations	$\mathrm{st}_s(q)$
I	$\emptyset$	ε
1	$a_1 \mid a_3 \mid a_6 \mid a_8$	a
2	$b_2 \mid b_7$	b
3	$c_4 \mid c_9$	c
4	d_{10}	d
5	e_5	e
6	$a_3c_4 \mid a_8c_9$	ac
7	c_9d_{10}	cd
8	c_4e_5	ce
9	e_5a_6	ea
10	$a_1b_2a_3 \mid a_6b_7a_8$	aba
11	$a_1b_2a_3c_4 \mid a_6b_7a_8c_9$	$abac$
12	$a_3c_4e_5a_6$	$acea$
13	$e_5a_6b_7a_8$	$eaba$
14	$a_6b_7a_8c_9d_{10}$	$abacd$
15	$a_1b_2a_3c_4e_5a_6$	$abacea$
16	$a_3c_4e_5a_6b_7a_8$	$aceaba$
17	$a_3c_4e_5a_6b_7a_8c_9$	$aceabac$
18	$a_3c_4e_5a_6b_7a_8c_9d_{10}$	$aceabacd$
19	$a_1b_2a_3c_4e_5a_6b_7a_8c_9d_{10}$	$abaceabcd$

Table 1. Copy relation example for $s = a_1 \; b_2 \; a_3 \; c_4 \; e_5 \; a_6 \; b_7 \; a_8 \; c_9 \; d_{10}$.

suffix tree. This structure contains all the fingerprints that need to be coded using the new naming algorithm presented in Section 3.

We assume below without loss of generality that the input sequence does not contain two consecutive repeating characters. Such a sequence is named *simple*. The segments of repeating characters, say α, of any input sequence can be reduced to a unique occurrence of α. The two sequences have the same sets, $\mathcal{F}$, and the same sets, $\mathcal{L}$ and $\mathcal{L}_C$, up to small changes in the bounds. These changes can, however, be simply retrieved in $\Theta(1)$ per maximal location and the reducing algorithm is $\Theta(n)$. This technical trick really simplifies the algorithms we present by removing many straightforward technical cases.

2 Participation Tree

Let $s = s_1..s_n$ be a simple sequence of characters over Σ. In this first phase, for reasons that will appear clearly below, we add to the sequence a last character $s_{n+1} = \#$ that does not appear in the sequence. Thus $s = s_1..s_n\#_{n+1}$. Let i, j be a position in s, $1 \leq i \leq j \leq n+1$. We define $\text{fo}_s(i,j)$ as the string formed by concatenating the first occurrences of each distinct character touched when reading s from position i (included) to position j (included). For instance, if $s = a_1b_2a_3c_4e_5a_6b_7a_8c_9d_{10}\#$, $\text{fo}_s(3,9) = ace$ and $\text{fo}_s(5,10) = eabcd$.

Definition 3. *Let $s = s_1..s_ns_{n+1}$ with $s_{n+1} = \#$ and $1 \leq i \leq n$ a position in s. Let $j > i$ the minimum position such that $s_j = s_i$ if it exists, $j = n+2$ otherwise. We define $lfo_s(i) = fo_s(i, j-1)$.*

For instance, if $s = a_1b_2c_3a_4d_5a_6b_7a_8c_9b_{10}e_{11}\#_{12}$, $\text{lfo}_s(1) = abc$ and $\text{lfo}_s(5) = dabce\#$.

The participation tree ressembles a tree of all $\text{lfo}_s(i)$ in which we removed each last character (the need of this removal will appear clearly below). It contains the same paths labels. However, building this exact tree is too time consuming and the participation tree allows some redundancy in the path labels, the same path label might correspond to several paths from the root. Our tree is thus not always "deterministic" in the sense that a node can have several transitions by the same character. We define it and build it from the suffix tree by cutting and shrinking edges. We first succintly recall the properties of the suffix tree.

2.1 Suffix Tree

The suffix tree $\text{ST}(s)$ is a compact representation of all suffixes of a given sequence $s = s_1 \dots s_n$. It is basically a trie of all suffixes of s where all nodes with a single child are merged with their parents. Each transition of the tree is then coded as an interval $[i,j]$ corresponding to $s_i..s_j$. Its size is $O(n)$ and there exists many $O(n \log |\Sigma|)$ time contruction algorithm based on different paradigms. The three most important are chronically that of Weiner [11], McCreight[9] and Ukkonen [10]. An example of such a suffix tree is given in Figure 1.

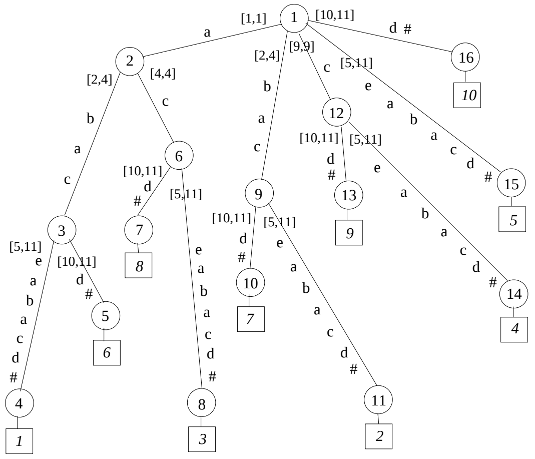

Fig. 1. Suffix tree of $s = a_1\ b_2\ a_3\ c_4\ e_5\ a_6\ b_7\ a_8\ c_9\ d_{10}\#_{11}$. Square boxes contain the initial position of the suffix. Each edge is labeled by a pair $[k, l]$ pointing to $s_k..s_l$ that we explicity write on the edge for clarity.

We assume below that in the suffix tree each transition interval $[i, j]$ of $\mathrm{ST}(s)$ corresponds to the leftmost occurrence of the factor $s_i \dots s_j$ in s. For instance, in Figure 1, the transition from 1 to 2 is the pointer $[1, 1] = s_1 = a$. This property is insured by Ukkonen [10] algorithm, but can also be insured on every suffix tree by a simple additional $O(n)$ step.

2.2 Participation Tree

Let $s = s_1..s_n s_{n+1}$ where $s_{n+1} = \#$. The participation tree $PT(s)$ is built from the suffix tree $ST(s)$ the following way. Imagine the suffix tree in an "expanded" version, that is each edge $[i, j]$ explicitely written by the corresponding factor $s_i..s_j$ (see Figure 1). Let us consider the sequence of characters on each path from the root and let α be the first character on this path. Let o be the second occurrence of α on this path if it exists. We perform the following steps:

1. we first reduce all characters on this path after o (included) to the empty string ε;
2. then, on the section from the root to the character before o we only keep the first occurrence of each appearing character, i.e. the others are reduced to ε;
3. we then replace the last character of each path from the root to a leaf by ε;
4. we replace all multi-characters edges by an equivalent serie of a single character and a node. An example of such a resulting tree is shown in Figure 2 (left);
5. as a last step, all ε edges (p, ε, q) are removed by merging p and q. The resulting tree is the participation tree. An example of this last tree is shown in Figure 2 (right).

For each node q of $ST(s)$ and $PT(s)$ we denote $\mathrm{Suff}(q)$ the set of suffixes of s that appear as leaves of the subtree rooted in q. We consider below that the

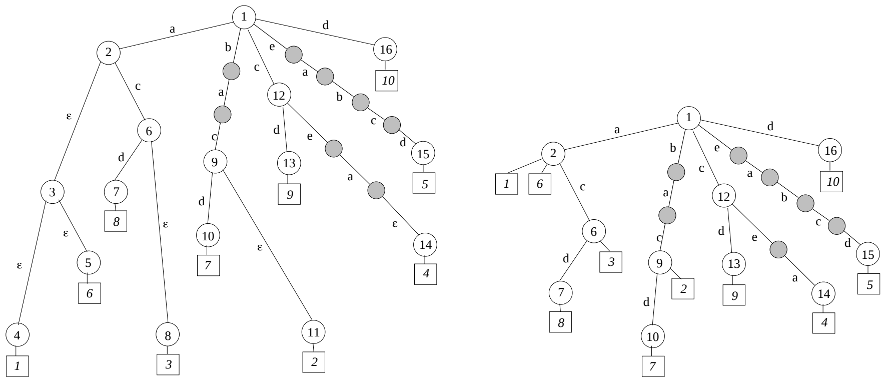

Fig. 2. From suffix tree to the participation tree (right picture) of $s = a_1b_2a_3c_4e_5a_6b_7a_8c_9d_{10}\#_{11}$. New nodes are in gray. The ε transitions are removed in the last step. Attached suffixes are shown in square boxes.

suffixes associated to a node in $ST(s)$ remains associated to the node in $PT(s)$, even after the merging. This is shown in Figure 2: the suffixes in the square boxes associated to nodes 4 and 5 in the left picture are associated to node 2 in the participation tree (right picture).

Lemma 1. *Let $s = s_1..s_n$. For all $i = 1..n$, each proper prefix of $lfo_s(i)$ labels a path from the root in $PT(s)$.*

Proof. The reduction of the path of suffix i in the suffix tree corresponds, when nodes are ignored, to $\mathrm{lfo}_s(i)$ without its last character. □

Note that a proper prefix of $\mathrm{lfo}_s(i)$ might label several paths from the root in $PT(s)$.

Let $[i, j]$ be an interval on $s = s_1..s_n$ and let $\mathrm{Support}([i, j])$ be the minimum of the indices of the rightmost occurrences of $\alpha = s_p$, $i \leq p \leq j$, in the interval $[i, j]$. We define $O_s^{[i,j]}$ as $\mathrm{fo}_s(\mathrm{Support}([i, j]), j)$. For instance, if $s = a_1b_2a_3c_4e_5a_6b_7a_8c_9d_{10}\#_{11}$, $\mathrm{Support}(\langle 1, 3\rangle) = 2$, $\mathrm{Support}([4, 10]) = 5$, $O_s^{\langle 1,3\rangle} = ba$ and $O_s^{[4,10]} = eabcd$.

Definition 4. *Let $s = s_1..s_n$ and $1 \leq i \leq j \leq n$. We define $Extend_s(i, j)$ as the maximal location reached when extending the interval $[i, j]$ to the left and to the right while the closest external characters s_{i-1} or s_{j+1} (if they exist) belong to $C_s(i, j)$.*

For instance, if $s = a_1\ b_2\ a_3\ c_4\ e_5\ a_6\ b_7\ a_8\ c_9\ d_{10}\#_{11}$, $\langle 1, 4\rangle = \mathrm{Extend}_s(2, 4)$ and $\langle 1, 9\rangle = \mathrm{Extend}_s(2, 7)$

Lemma 2. *Let $\langle i, j\rangle$ be a maximal location of $s = s_1..s_n$. There exists a permutation of all characters of $C_s(i, j)$ that labels a path from the root in $PT(s)$.*

Proof. $O_s^{\langle i,j\rangle}$ is obviously a proper prefix of $\mathrm{lfo}_s(\mathrm{Support}(\langle i,j\rangle))$, which, by lemma 1, labels a path from the root in $PT(s)$. □

Corollary 1. *Let $s = s_1..s_n$. For all $i, j, 1 \leq i \leq j \leq n$, there exists a permutation of all characters of $C_s(i,j)$ that labels a path from the root in $PT(s)$.*

Proof. It suffices to extend the segment $s_i..s_j$ to $\langle k,l\rangle = \mathrm{Extend}_s(i,j)$ in which it is contained. Then $C_s(i,j) = C_s(k,l)$ and lemma 1 applies. □

Let $z = ((r,\alpha_1,p_1),\ldots,(p_{i-1},\alpha_i,p_i))$ be a path in $PT(s = s_1..s_n)$ from its root r. By notation extension, we denote $\mathrm{Suff}(z) = \mathrm{Suff}(p_i)$. Let $\mathrm{SPref}(s)$ be the set of all such paths and $w(z) = \alpha_1\alpha_2..\alpha_i$. Let $\mathcal{P}(\mathcal{L})$ be the set of all sets of maximal locations.We consider the function Φ formally defined as:

$$\Phi : \left| \begin{array}{lcl} \mathrm{SPref}(s) & \longrightarrow & \mathcal{P}(\mathcal{L}) \\ \quad z & \longmapsto & \{\langle k,l\rangle \in \mathcal{L} \mid O_s^{\langle k,l\rangle} = w(z) \text{ and } \mathrm{Support}(\langle k,l\rangle) \in \mathrm{Suff}(z)\} \end{array} \right.$$

Lemma 3. *Let $z = ((r,\alpha_1,p_1),\ldots,(p_{i-1},\alpha_i,p_i))$ be a non empty path in SPref(s). Then $\Phi(z) \neq \emptyset$.*

Proof. By construction of the participation tree, there exits $m \in \mathrm{Suff}(z)$ such that $\alpha_1\ldots\alpha_i$ is a proper prefix of $\mathrm{lfo}(m)$. Let p be the first position of α_i in s following m. Then $\cup_{1\leq f\leq i}\{\alpha_f\} = C_s(m,p)$. Let $\langle k,l\rangle = \mathrm{Extend}_s(m,p)$.

We prove now that $\mathrm{Support}(\langle k,l\rangle) = m$. As $\alpha_1\ldots\alpha_i$ is a proper prefix of $\mathrm{lfo}(m)$, there exist $\alpha = \mathrm{lfo}(m)_{i+1}$ such that there is no occurrence of α in the interval $[m,p]$, and thus after the extension of $[m,p]$ to a maximal location $\langle k,l\rangle$, the indice l is strictly less than the indice of the first occurrence of α after m. As, by definition of lfo(m), there is no occurrence of s_m before the indice of α after m in s, there is no other occurrence of s_m at the right of s_m in the interval $[m,l]$. Moreover, as all characters in $\alpha_1\ldots\alpha_i$ and only them appears after m in $[m,l]$ in the order of $\alpha_1\ldots\alpha_i$, and that the extension procedure insures that all characters in $[k,m]$ are characters of $\alpha_1\ldots\alpha_i$, $\mathrm{Support}(\langle k,l\rangle) = m$.

Finally, it is obvious that $O_s^{\langle k,l\rangle} = O_s^{[m,p]} = \alpha_1..\alpha_i = w(z)$, and thus $\langle k,l\rangle \in \Phi(z)$. □

Lemma 4. *Let z_1, $z_2 \in$ SPref(s) be two distinct non empty paths, then $\Phi(z_1) \cap \Phi(z_2) = \emptyset$.*

Proof. Assume *a contrario* that there exists $\langle k,l\rangle \in \Phi(z_1) \cap \Phi(z_2)$. Let $m = \mathrm{Support}(\langle k,l\rangle)$, $m \in \mathrm{Suff}(z_1)$ and $m \in \mathrm{Suff}(z_2)$, thus one of the path is a prefix of the other. As $O_s^{[k,l]} = w(z_1) = w(z_2)$, the two paths must be equal, which contradicts the hypothesis. □

Lemma 5. *Let $\langle i,j\rangle$ and $\langle k,l\rangle$ be two distinct maximal locations of $s = s_1..s_n$ in the same equivalence class of $\mathcal{L}_c$. There exits $z \in$ SPref(s) such that $\langle i,j\rangle \in \Phi(z)$ and $\langle k,l\rangle \in \Phi(z)$.*

Proof. Let $m_1 = \text{Support}(\langle i,j\rangle)$ and $m_2 = \text{Support}(\langle k,l\rangle)$. As $s_i..s_j = s_k..s_l$, $u = s_{m_1}..s_j = s_{m_2}..s_l$ and m_1 and m_2 are thus in the subtree of the path h labeled by u in $ST(s)$. After reduction of this path in $PT(s)$, the resulting path z is such that $w(z) = O_s^{\langle i,j\rangle} = O_s^{\langle k,l\rangle}$ and $m_1, m_2 \in \text{Suff}(z)$. Thus $\langle i,j\rangle, \langle k,l\rangle \in \Phi(z)$. □

Theorem 1. *All maximal locations are in the image $\Phi(z)$ of a path z in $PT(s = s_1..s_n)$ and the size of $PT(s)$ is $O(|\mathcal{L}_C|)$.*

Proof. Lemma 2 directly implies that all maximal locations are in the image $\Phi(z)$ of a path z in $PT(s)$. As by lemma 4 the images $\Phi(z)$ are non overlapping, they form a partition of $\mathcal{L}$. Lemma 5 insures that $\mathcal{L}_C$ partition is a subpartition of the partition formed by the images of Φ. As by lemma 3 there is no empty image, the number of such images is smaller than or equal to $|\mathcal{L}_C|$. □

Note that we considered the size of $PT(s = s_1..s_n)$ without the initial positions of suffixes (square boxes in Figure 2). With these positions, its size is $O(n + |\mathcal{L}_C|)$.

2.3 From Suffix Tree to Participation Tree

We extend the notion of $\text{fo}_s(i,j)$ keeping the positions of the characters in $s = s_1..s_n$. We define $\text{efo}_s(i)$ as the string formed by concatenating the first occurrences of each distinct character touched when reading s from position i (included) to position n (included) but indexed by the position of this character in the sequence. For instance, if $s = a_1b_2a_3c_4e_5a_6b_7a_8c_9d_{10}\#_{11}$, $\text{efo}_s(3) = a_3c_4e_5b_7d_{10}\#_{11}$ and $\text{efo}_s(5) = e_5a_6b_7c_0d_{10}\#_{11}$.

The idea of the algorithm is the following. For each transition (i,j) on the path of a longest suffix $v = s_f \dots s_n$ we compute the "participation" of the edge to $\text{lfo}_s(f)$, that is, the number of new characters the edges brings in $\text{lfo}_s(f)$. For instance, in Figure 1 the participation of edge $(6,8) = [5,11]$ is e, since it is on the path of the longest suffix $s_3 \dots s_n$ and $\text{lfo}_s(3) = ace$. The participation of edge $(12,14) = [5,11]$ is eab since $\text{lfo}_s(4) = ceab$. To compute the participation of interval $[i,j]$ on the path of a longest suffix $v = s_f \dots s_n$, we use $\text{efo}_s(f)$ and also the next position of s_f after f in s, if it exists. Assume it is the case and let p be this position. Thus $s_p = s_f$. Let $\text{efo}_s(f) = s_f s_{l_1} s_{l_2} \dots s_{l_z}$and $l_h \leq p \leq l_{h+1}$. If $i \geq p$, the participation of $[i,j]$ is the empty word ε. Otherwise, $i < p$ and its participation is the string (potentially empty) $s_{l_a} \dots s_{l_b}$ with

– $i \leq l_a$ and l_a is the smallest such indice;
– $l_b \leq min(j, p-1)$ and l_b is the greatest such indice.

For instance, on Figure 1, $\text{efo}_s(2) = b_2a_3c_4e_5d_{10}\#_{11}$ and $p = 7$ since 7 is the next position of b after position 2. Thus, participation of edge $(1,9) = [2,4] = b_2a_3c_4 = bac$, participation of $(9,11) = [5,11] = e_5 = e$ (since $p = 7$). For each suffix $[k,n]$, given $\text{efo}_s(k)$ and p, a bottom-up process from leaf k to the root of the suffix tree allows us to calculate the participation of each (not previously touched) edge on this path. We modify the suffix tree using successively $\text{efo}_s(k)$ for $k = 1..n$. A sketch of this algorithm is given in Figure 3.

```
BUILD_PART_TREE($ST(s = s_1..s_n s_{n+1}$ with $s_{n+1} = \#$))
Compute $\mathrm{efo}_s(1)$ and $p_1$
For $i = 1..n$ Do
    Current $\leftarrow$ Leaf($i$) in $ST(s)$.
    While Current not marked AND Current $\neq$ Root Do
        Prec $\leftarrow$ Parent(Current) in $ST(s)$.
        Compute the participation of edge (Parent, Current) in $\mathrm{efo}_s(i)$
        Mark Current
        $\mathrm{efo}_s(i+1) \leftarrow$ Update $\mathrm{efo}_s(i)$
        $p_{i+1} \leftarrow$ next position of $s_{i+1}$ after $i+1$ in $s$
    End of while
End of for
Replace each last character of all paths from the root by $\varepsilon$.
Remove $\varepsilon$ edges by node merging.
```

Fig. 3. Building the participation tree from the suffix tree.

At the end of this process, we first replace the last character of all paths from the root by ε. we finally remove all (p, ε, q) edges by merging p and q.

Theorem 2. *The participation tree of $s = s_1..s_n$ can be built in $O(n \log |\Sigma| + |\mathcal{L}_C|)$ time and $O(n + |\mathcal{L}_C|)$ space.*

Proof. The algorithm is correct since it consists of directly compute the participation of each edge one after the other. We now study its complexity.

For each suffix $[k, n]$, given $\mathrm{efo}_s(k)$ in an AVL tree and p, the bottom-up process from leaf k to the root of the suffix tree can be done in $O(\log |\Sigma|)$ time for each unmarked node. If the first $\mathrm{efo}_s(1)$ is given as an AVL tree, initially built in $O(|\Sigma| \log |\Sigma|)$, $\mathrm{efo}_s(2)$ can be obtained in $O(\log |\Sigma|)$, and so on for $k = 3..n$, assuming that for each k we know the next position of s_k in $s_{k+1} \ldots s_n$ if it exits. To know these positions, $|\Sigma|$ lists, one for each character α, of positions of α in s can be initialy computed in $O(|\Sigma| + n)$ time and consumed character after character. Thus, calculating the participation of each edge in the suffix tree can be done in $O(\log |\Sigma|)$ writing each time a unique part of the $PT(s)$ tree.

Replacing the last character of each path from the root by ε is $O(n)$. Merging each ε edges can also be performed in $O(n)$ since each such edge is either a previous edge of the suffix tree or was labeled by a single last character of a path from the root. The whole construction of $PT(s)$ is thus $O(n \log |\Sigma| + |\mathcal{L}_C|)$ time.

The size of the AVL tree is bounded by $|\Sigma|$, thus by n. The space required is the size of the suffix tree plus the size of the participation tree, thus $O(n + |\mathcal{L}_C|)$ space. □

3 Naming All Fingerprints

In this section we explain how to name all fingerpints from the participation tree. The naming technique itself is originally based on that of [6] that has been adapted for the fingerprint problem in [1]. The naming technique is used to give a unique name to each fingerprint of a substring of s. We first describe the naming technique and then we explain how to use it to name all fingerprints of s.

3.1 Naming Technique

We assume for simplicity, but without loss of generality, that $|\Sigma|$ is a power of two. We consider a stack of $\log |\Sigma| + 1$ arrays on top of each other. Each level is numbered from 1. The lowest, called the fingerprint table, contains $|\Sigma|$ names that might be only [0] or [1]. Each other array contains half the number of names that the array it is placed on. The highest array only contains a single name that will be the name of the whole array. Such a name is called a fingerprint name. Figure 4 shows a simple example with $|\Sigma| = 8$.

<table>
<tr><td colspan="8">[7]</td></tr>
<tr><td colspan="4">[5]</td><td colspan="4">[6]</td></tr>
<tr><td colspan="2">[2]</td><td colspan="2">[2]</td><td colspan="2">[3]</td><td colspan="2">[4]</td></tr>
<tr><td>[1]</td><td>[0]</td><td>[1]</td><td>[0]</td><td>[1]</td><td>[1]</td><td>[0]</td><td>[0]</td></tr>
</table>

Fig. 4. Naming example.

The names in the fingerprint table are only [0] or [1] and are given. Each cell, c, of an upper array represents two cells of the array it is placed on, and thus a pair of two names. The naming is done in the following way: for each level going from the lowest to the highest, if the cell represents a new pair of names, give this pair a new name and assign it to the cell. If the pair has already been named, place this name into the cell. In the example in Figure 4, the name [2] is associated to $([1], [0])$ the first time this pair is encountered. The second time, this name is directly retrieved.

3.2 Naming a List of Fingerprint Changes

Assume that a specific set $\mathcal{S}$ of fingerprints can be represented as a list $L = (\alpha_1, \alpha_2, \dots \alpha_p)$ of distinct characters such that $S = \{f_1, f_2, \dots, f_p\}$ where $f_i = \cup_{1 \leq j \leq i} \{\alpha_j\}$. The core idea of the algorithm of [4] is to fill a fingerprint table bottom-up by building for each level an ordered list of new names that corresponds to the fingerprint changes induced at the previous level. A pseudo-code of this naming algorithm is given in Figure 5. We explain it below.

We number the level from 1, the lowest, to $\log |\Sigma| + 1$. The original list L is first tranformed into a list L_1 of changes on level 1 by replacing each character α_i by the pair $\{[1], f_\Sigma(\alpha_i)\}$. To initialize the process we add a list of $|\Sigma|$ pairs $\{[0], i\}$, $i = 1..|\Sigma|$ at the beginning of L_1.

This initial list is then used to compute all names of the cells in the second level. A table, FT, of $|\Sigma|$ names temporary records the pair of names to be coded. A list L'_1 of pairs of names is built as follows. The first $|\Sigma|$ elements of L_1 are read to initialize FT. The list L'_1 is initialized with $|\Sigma|/2$ pairs built by reading FT. Then, the remaining of the list L_1 is read and for each new element $\{[a], j\}$ *(1)* the table FT is changed in position j by $FT \leftarrow [a]$ and *(2)* the pair $\{(FT[2\lfloor j/2 \rfloor], FT[2\lfloor j/2 \rfloor + 1]), j/2\}$ if added to the end of L'_1. This means that in cell $j/2$ of the second level a name has to be given to the name pair $(FT[2\lfloor j/2 \rfloor], FT[2\lfloor j/2 \rfloor + 1])$.

```
NAME_LISTS($L = (\alpha_1, \alpha_2, \dots \alpha_p)$ initial list of changes)
$L_1 \leftarrow (\{[0], 0\}, \dots, \{[0], |\Sigma| - 1\})$
add $(\{[1], f_\Sigma(\alpha_1)\}, \dots, \{[1], f_\Sigma(\alpha_p)\})$ to end of $L_1$
For $r = 1..\log|\Sigma|$ Do
    $FT_r \leftarrow$ name table of size $|\Sigma|/2^{r-1}$
    $E_{tp} \leftarrow$ first element of $L_r$
    For $l = 0..|\Sigma|/2^{r-1} - 1$ Do     /* initialization of table $FT$ */
        $\{[a], j\} \leftarrow E_{tp}$
        $FT_r[j] \leftarrow [a]$
        $E_{tp} \leftarrow$ next element in $L_r$
    End of for
    Let $L'_r$ be an empty list
    For $l = 0..|\Sigma|/2^r - 1$ Do      /* initialization of $L'_r$ list */
        add $\{(FT[2l], FT[2l+1]), l\}$ to end of $L'_r$
    End of for
    $E_{tp} \leftarrow$ first element of $L_r$
    While $E_{tp}$ exists Do
        $\{[a], j\} \leftarrow E_{tp}$
        $FT_r[j] \leftarrow [a]$
        add $\{(FT_r[2\lfloor j/2 \rfloor], FT_r[2\lfloor j/2 \rfloor + 1]), j/2\}$ to end of $L'_r$
        $E_{tp} \leftarrow$ next element in $L_r$
    End of while
    sort the pair of names in $L'_r$ in lexicographical order
    give new names in each unique pair in $L'_r$
    build $L_{r+1}$ by copying $L'_r$ but replacing each pair by its new name
End of for
```

Fig. 5. Naming a list $L = (\alpha_1, \alpha_2, \dots \alpha_p)$ of fingerprint changes.

At this point L'_1 records the list of changes to be made in the cells at level 2 and the pairs of names that must receive a name. The pairs in this list are then sorted in lexicological order (through a radix sort) and a new name is assigned to each distinct pair of names (n_1, n_2). A new list L_2 is built from L'_1 (keeping the initial order of L'_1 and thus of L_1) by replacing each pair with its new name. For instance, if $\{([1], [0]), 1\}$ was in the list L'_1 and if the pair $([1], [0])$ received the new name $[2]$, then L_2 now contains $\{[2], 1\}$.

The list L_2 is the input at level 2 and the same process is repeated to obtain the names in the third level, and so on. The last list $L_{\log|\Sigma|+1}$ contains the names of all the fingerprints of $\mathcal{S}$.
Complexity The initialization of L_1 is $\Theta(|L|)$ time. Then a linear sort of $\Theta(|L|)$ elements is performed for every level. As there are $\log|\Sigma| + 1$ levels, naming the list is $\Theta(|L| \log|\Sigma|)$ time.

3.3 Naming a Participation Tree

The naming approach of the previous section has been modified in [7] to name on the same set of names a table of lists of fingerprint changes. The main modification is that the linear sorting is done for each level on all the pairs of all the

```
Depth_first_search($FT_k$,$Current$)
  1. For all $\alpha$ such that $\delta(Current, \alpha) \neq \Theta$ Do
  2.     $q \leftarrow \delta(Current, \alpha)$
  3.     $\{[a], j\} \leftarrow \Delta(Current, \alpha, q)$
  4.     $prec \leftarrow FT_k[j]$
  5.     $FT_k[j] \leftarrow [a]$
  6.     $\Delta(Current, \alpha, q) \leftarrow \{(FT_k[2\lfloor j/2 \rfloor], FT_k[2\lfloor j/2 \rfloor + 1]), j/2\}$
  7.     Depth_first_search($FT_k$,$q$)
  8.     $FT_k[j] \leftarrow prec$
  9. End of for

Name_fingerprint($PT(s)$)
 10. $ninit_1 \leftarrow [0]$
 11. For $k = 1..\log |\Sigma|$ Do
 12.     $FT_k \leftarrow$ name table of size $|\Sigma|/2^{k-1}$ all initialized to $ninit_k$
 13.     Depth_first_search($FT_k$,Root($PT(s)$))
 14.     $Sl \leftarrow \Theta$     /* empty stack */
 15.     For  all edges $e = (p, \alpha, q)$ in $PT(s)$ Do
 16.         $\{(n_1, n_2), j\} \leftarrow \Delta(p, \alpha, q)$
 17.         Add $(n_1, n_2)$ to $Sl$.
 18.     End of for
 19.     add the couple $(ninit_k, ninit_k)$ to $Sl$
 20.     sort $Sl$ in lexicographical order
 21.     give new names for each different couple in $Sl$
 22.     replacing each pair in $\Delta(p, \alpha, q)$ by its new name
 23.     $ninit_{k+1} \leftarrow$ name of the pair $(ninit_k, ninit_k)$
 24. End of for
```

Fig. 6. Naming all fingerprints in a participation tree $PT(s)$.

lists of the table. We use a similar approach, but instead of a table of lists we consider the set of all paths from the root in the participation tree $PT(s)$. Each such path is considered as a list of fingerprint changes. The corollary 1 guaranty our approach. The Name_fingerprint algorithm names all fingerprints. Its pseudo-code is given in Figure 6.

As in the list naming of section 3.2, $\log |\Sigma|$ iterations are performed, one by fingerprint array level (loop 14-27), the lowest one excepted. With each edge (p, α, q) of $PT(s)$ is associated a value $\Delta(p, \alpha, q)$. At the end of iteration k, this value records the change corresponding to the edge in the fingerprint array of level $k + 1$. The value $\Delta(p, \alpha, q)$ is assumed to be initialized with $\{[1], f_\Sigma(\alpha)\}$ corresponding to the change induced by the edge at the lowest level 1.

In each iteration k, the recursive algorithm Depth_first_search is called (line 16) on the participation tree to update all values $\Delta(p, \alpha, q)$ during a depth first search. The update operation on each such value is similar to the pair update in the naming of a simple list of fingerprint changes in section 3.2. Note that in Depth_first_search a special FT table is modified (line 6) before the recursive call but reinitialized to the previous value after the call (line 11). This permits to initialize the table FT only once before the first call to DEPTH_FIRST_SEARCH (line 15).

After the depth first search the values $\Delta(p, \alpha, q)$ are collected on all the edges (p, α, q) of the participation tree (lines 18-22) in a list Sl. This list is lexicographically sorted and a new name is given to each unique pair (line 25), similarly to the naming of a single list in section 3.2. The first pair of names of each $\Delta(p, \alpha, q)$ is then replaced by its new name.

To initialize the fingerprint array at the next level, the couple $(ninit_k, ninit_k)$ is added to the list of names (line 23) and its new name is retrieved after the sorting and the renaming (line 27).

At the end of the last iteration of the main loop (line 14-28), the last naming (line 25) returns the list of all the fingerprint names.

Theorem 3. *The* NAME_FINGERPRINT *algorithm applied on $PT(s)$ names all fingerprints of s in $\Theta(|\mathcal{L}_C| \log |\Sigma|)$ time.*

References

1. A. Amir, A. Apostolico, G. M. Landau, and G. Satta. Efficient text fingerprinting via parikh mapping. *J. Discrete Algorithms*, 1(5-6):409–421, 2003.
2. A. Bergeron, C. Chauve, F. de Montgolfier, and M. Raffinot. Computing common intervals of k permutations, with applications to modular decomposition of graphs. In *European Symposium on Algorithms (ESA)*, number 3669 in LNCS, pages 779–790. Springer-Verlag, 2005.
3. G. Didier. Common intervals of two sequences. In *WABI*, number 2812 in Lecture Notes in Computer Science, pages 17–24. Springer-Verlag, Berlin, 2003.
4. G. Didier, T. Schmidt, J. Stoye, and D. Tsur. Character sets of strings. 2004. Submitted.
5. S. Heber and J. Stoye. Finding all common intervals of k permutations. In *Combinatorial Pattern Matching (CPM)*, number 2089 in Lecture Notes in Computer Science, pages 207–218. Springer-Verlag, Berlin, 2001.
6. R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th ACM Symposium on the Theory of Computing*, pages 125–136, Denver, CO, 1972. ACM Press.
7. R. Kolpakov and M. Raffinot. New algorithms for text fingerprinting. In *Annual Symposium on Combinatorial Pattern Matching*, number 4009 in Lecture Notes in Computer Science, pages 342–353. Springer-Verlag, Berlin, 2006.
8. R. Kolpakov and M. Raffinot. New algorithms for text fingerprinting. *Unpublished*, 2006. Submitted to Journal of Discrete Algorithms. `http://www-igm.univ-mlv.fr/~raffinot/ftp/fingerprint.pdf`.
9. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of Algorithms*, 23(2):262–272, 1976.
10. E. Ukkonen. Constructing suffix trees on-line in linear time. In J. van Leeuwen, editor, *Proceedings of the 12th IFIP World Computer Congress*, pages 484–492, Madrid, Spain, 1992. North-Holland.
11. P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.