Various improvements to text fingerprinting*

Djamal Belazzougui\textsuperscript{1}    Roman Kolpakov\textsuperscript{2}    Mathieu Raffinot\textsuperscript{1}

\textsuperscript{1} LIAFA, Univ. Paris Diderot - Paris 7, 75205 Paris Cedex 13, France, 
\{delaz, raffinot\}@liafa.jussieu.fr
\textsuperscript{2} Liapunov French-Russian Institute, Lomonosov Moscow State University, Moscow, 
Russia, foroman@mail.ru

Abstract: Let \( s = s_1..s_n \) be a text (or sequence) on a finite alphabet \( \Sigma \) of size \( \sigma \). A fingerprint in \( s \) is the set of distinct characters appearing in one of its substrings. The problem considered here is to compute the set \( \mathcal{F} \) of all fingerprints of all substrings of \( s \) in order to answer efficiently certain questions on this set. A substring \( s_i..s_j \) is a maximal location for a fingerprint \( f \in \mathcal{F} \) (denoted by \( (i,j) \)) if 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 in \( s \) is \( \mathcal{L} \) (it is easy to see that \( |\mathcal{L}| \leq n\sigma \)). Two maximal locations \( (i,j) \) and \( (k,l) \) such that \( s_i..s_j = s_k..s_l \) are named copies, and the quotient set of \( \mathcal{L} \) according to the copy relation is denoted by \( \mathcal{L}_C \).

We first present new exact efficient algorithms and data structures for the following three problems: (1) to compute \( \mathcal{F} \); (2) given \( f \) as a set of distinct characters in \( \Sigma \), to answer if \( f \) represents a fingerprint in \( \mathcal{F} \); (3) given \( f \), to find all maximal locations of \( f \) in \( s \). As well as in papers concerning succinct data structures, in the paper all space complexities are counted in bits. Problem 1 is solved either in \( O(n + |\mathcal{L}_C| \log \sigma) \) worst-case time (in this paper all logarithms are intended as base two logarithms) using \( O((n + |\mathcal{L}_C| + |\mathcal{F}| \log \sigma) \log n) \) bits of space, or in \( O(n + |\mathcal{L}| \log \sigma) \) randomized expected time using \( O((n + |\mathcal{F}| \log \sigma) \log n) \) bits of space. Problem 2 is solved either in \( O(|f| \log n) \) expected time if only \( O(|f| \log n) \) bits of working space for queries is allowed, or in worst-case \( O(|f|/\epsilon) \) time if a working space of \( O(\sigma' \log n) \) bits is allowed (with \( \epsilon \) a constant satisfying \( 0 < \epsilon < 1 \)). These algorithms use a data structure that occupies \( |\mathcal{F}|(2\log \sigma + \log_2 e)(1 + o(1)) \) bits. Problem 3 is solved with the same time complexity as Problem 2, but with the addition of an \( \text{occ} \) term to each of the complexities, where \( \text{occ} \) is the number of maximal locations corresponding to the given fingerprint. Our solution of this last problem requires a data structure that occupies \( O((n + |\mathcal{L}_C|) \log n) \) bits of memory.

In the second part of our paper we present a novel Monte Carlo approximate construction approach. Problem 1 is thus solved in \( O(n + |\mathcal{L}|) \) expected time using \( O(|\mathcal{F}| \log n) \) bits of space but the algorithm is incorrect with an extremely small probability that can be bounded in advance.

1 Introduction

We consider a finite ordered alphabet \( \Sigma \) with \( \sigma = |\Sigma| \) and \( s = s_1..s_n \) a sequence of \( n \) letters, \( s_i \in \Sigma \). The set of all sequences over \( \Sigma \) is denoted \( \Sigma^* \). The rank of each letter \( \alpha \) in \( \Sigma \) 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 \( s_i..s_j \).

\* 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 work is also supported by the french ANR project MAPPI.
### Definition 1.
Let $C$ be a set of letters of $\Sigma$. A maximal location of $C$ in $s = s_1..s_n$ is an interval $[i, j]$, $1 \leq i \leq j \leq n$, such that

1. $C_s(i, j) = C$;
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 over $\mathcal{L}$, and we denote $\mathcal{L}_C$ the set of equivalence classes. In this paper, given a sequence $s$, we are interested in three main problems:

1. Compute the set $\mathcal{F}$ of all fingerprints in $s$;
2. Given a fingerprint $f$, find whether $f$ is a fingerprint in $\mathcal{F}$;
3. Given a fingerprint $f$, find all the maximal locations of $f$ in $s$.

Efficient answers to these questions have many applications in information retrieval, computational biology and natural language processing. The input alphabet $\Sigma$ is considered to be the alphabet of the input sequence, thus $|\Sigma| \leq n$. The best current algorithms solve Problem 1 in $\Theta(\min\{n + |\mathcal{L}| \log |\Sigma|, n^2\})$ time and space. The bound $\Theta(n + |\mathcal{L}| \log |\Sigma|)$ is that of [15]. The $\Theta(n^2)$ bound is obtained using the algorithm of Didier et al. [16]. Problem 2 is solved in $O(|\mathcal{F}| \log(\sigma/|\mathcal{F}|))$ time and $O(|\mathcal{F}|)$ space ($O(|\mathcal{F}| \log n)$ bits) and Problem 3 in $O(|\mathcal{F}| \log(\sigma/|\mathcal{F}|) + \text{occ})$ time (where $\text{occ}$ is the number of maximal locations that match the given fingerprint) and $O(|\mathcal{F}| + |\mathcal{L}|)$ space ($O(|\mathcal{F}| + |\mathcal{L}|) \log n)$ bits) in [34].

We first present new exact efficient algorithms and data structures for the three problems we considered above.

Problem 1 is solved either in $O(n + |\mathcal{L}_C| \log |\Sigma|)$ worst-case time using $O((n + |\mathcal{L}_C| + |\mathcal{F}| \log |\Sigma|) \log n)$ bits of space, or in $O(n + |\mathcal{L}| \log |\Sigma|)$ randomized expected time using $O((n + |\mathcal{F}|) \log |\Sigma|) \log n)$ bits of space.

Problem 2 is solved either in $O(|\mathcal{F}|)$ expected time and space if only $O(|\mathcal{F}| \log n)$ bits of working space for queries is allowed, or in $O(|\mathcal{F}| / \epsilon)$ worst-case time if a working space of $O(\sigma^2 \log n)$ bits is allowed. This problem uses a data structure which occupies $|\mathcal{F}|(2 \log \sigma + \log_2 \epsilon)(1 + o(1))$ bits. Previous and new exact results are summarized in table 2.

Problem 3 is solved in the same time as Problem 2, with the addition of an $\text{occ}$ term to each of the complexities, where $\text{occ}$ is the number of maximal locations corresponding to the fingerprint searched. Previous and new exact results are summarized in tables 1-3.

| Solution | Build space (bits) | Build time |
|----------|--------------------|------------|
| prev. [15] (worst-case) | $O((n + |\mathcal{L}_C|) \log n)$ | $O(n + |\mathcal{L}| \log |\Sigma|)$ |
| theorems 2,3 (worst-case) | $O((n + |\mathcal{L}_C| + |\mathcal{F}| \log |\Sigma|) \log n)$ | $O(n + |\mathcal{L}_C| \log |\Sigma|)$ |
| theorem 4 (randomized expected) | $O((n + |\mathcal{F}|) \log |\Sigma|) \log n)$ | $O(n + |\mathcal{L}| \log |\Sigma|)$ |
| theorem 7 (Monte-Carlo) | $O((n + |\mathcal{F}|) \log n)$ | $O(n + |\mathcal{L}|)$ |

Table 1. Previous and new solutions to Problem 1 (Determination of $\mathcal{F}$).
Table 2. Previous and new solution for Problem 2 (existential fingerprint queries).

| Solution | Data structure space (bits) | Query time |
|----------|----------------------------|------------|
| prev.    | $O(|F| \log n)$             | $O(|f| \log(\sigma/|f|))$ |
| theorem 4| $O(|F| \log \sigma)$       | $O(|f|)$   |

Table 3. Previous and new solutions to Problem 3 (maximal location report queries)

| Solution | Data structure space (bits) | Query time |
|----------|----------------------------|------------|
| prev.    | $O(|L| \log n)$             | $O(|f| \log(\sigma/|f|) + \text{occ})$ |
| theorem 6| $O(|L| \log n)$             | $O(|f| + \text{occ})$ |
| theorem 7| $O((n + |L_C|) \log n)$      | $O(|f| + \text{occ})$ |

In this article we also propose a novel Monte Carlo approximate query approach. The result of the query may not be exact, but an error occurs at a probability that one can fix a priori as small as required. This approach has the advantage of speeding up the identification of all fingerprints by a $\log \sigma$ factor. Problem 1 is thus solved in $O(n + |L|)$ expected time using $O(|F| \log n)$ bits of space using a Monte Carlo approach, but the algorithm yields incorrect results with an extremely low probability. Table 3 summarizes the complexities of the construction space and time including the Monte-Carlo method.

Our algorithms are based on several tools of four main natures: hash functions, succinct data structures, trees, and naming techniques first introduced in [12], adapted to the fingerprint problem in [11] and then successively improved in [9] and in [15]. These tools are presented in Section 2. In Section 4 we present our $O(n + |L_C| \log \sigma)$ worst-case time construction algorithm. Section 3 presents a more space efficient representation of $F$ in space $O(|F| \log \sigma)$ bits instead of $O(|F| \log n)$ bits. This data structure allows us to solve Problem 2 and 3 in the complexities bounds announced above. Then Section 6 contains the $O(n + |L| \log \sigma)$ expected time algorithm using $O((n + |F| \log \sigma) \log n)$-bit space for solving Problem 1. Finally, in Section 5 we present the Monte Carlo algorithm that allows us to efficiently solve Problem 1 in time $O(|L|)$ and space $O(|F| \log n)$ thus saving a $\log \sigma$ factor in both space and time complexity of the algorithm in section 5.

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 $\alpha$, of any input sequence can be reduced to a unique occurrence of $\alpha$. The two sequences have the same set $F$ and the same sets $L$ and $L_C$, up to small changes in the bounds (these changes can be simply retrieved in $\Theta(1)$ time per maximal location and produced by trivial algorithm in $\Theta(n)$ time). This technical trick greatly simplifies the algorithms we present by removing many straightforward technical cases.

All the algorithms presented in this paper assume the unit-cost word RAM model with word length $w = \Omega(\log n)$ and with usual arithmetic and logic operations taking constant time (additions, multiplication, bitwise operations etc.).

2 Tools

This section is devoted to the four main tools we use in our algorithms, namely polynomial hash functions, the suffix tree, the participation tree and the naming technique.
2.1 Hash functions

Our constructions are based on the use of polynomial hash functions modulo $P$, where $P$ is a suitably chosen prime. Given a collection $M$ of $m$ sets over a universe $\sigma$, our goal is to find a polynomial hash function so that each set is mapped to a distinct value. The polynomials are evaluated modulo an arbitrary prime $P$ chosen so that $m^2\sigma \leq P \leq 2m^2\sigma$ (we will show later how to efficiently find such a prime). More precisely, we will use a family of hash functions $H_P = \{h_X|X \in [1, P-1]\}$, where each hash function $h_X \in H_P$ in the family is parametrized with an integer $X \in [1, P-1]$.

The functions of the family are defined in the following way: for any set $S = \{e_1, e_2, \ldots, e_t\}$ such that $S \subseteq [0, \sigma - 1]$ we have:

$$h_X(S) = \sum_{i=1}^{t} X_{e_i} \mod P$$

In order to compute a fixed hash function $h_X$ on any set $S$ in $O(|S|)$ time, we can use a precomputed table of size $\sigma$, which stores all the powers of $X$ up to $X^{\sigma-1}$. Alternatively, we could use a two-dimensional precomputed table $T$ of size $c \cdot \lceil \sigma^{1/c} \rceil$ for any integer $c$ ensuring a computation time of $O(c|S|)$. That is, we store in $T[i,j]$ the number $X^{ij}$ where $\gamma = \lceil \sigma^{1/c} \rceil$. Then in order to compute $X^{e_i}$, we can use the property that $e_i$ can be decomposed into a sum of $c$ numbers:

$$e_i = \sum_{j=0}^{c-1} d_{ij} \gamma^j$$

where each $d_{ij}$ can be computed using the formula:

$$d_{ij} = \lfloor e_i/c \rfloor \mod \gamma$$

Thus for computing $X^{e_i}$, it suffices to use the formula:

$$X^{e_i} = \prod_{j=0}^{c-1} X^{d_{ij} \gamma^j} = \prod_{j=0}^{c-1} T[d_{ij}, j]$$

To summarize, given any set $S = \{e_1, e_2, \ldots, e_t\}$ where $S \subseteq [0, \sigma - 1]$, $h_X(S)$ can be computed in $O(c \cdot t)$ time. First, for each $e_i$, compute $X^{e_i}$ in $O(c)$ time: for each $e_i$, compute its decomposition $\sum_{0 \leq j < c} d_{ij} \gamma^j$ in $O(c)$ time where each $d_{ij}$ is computed by $d_{ij} = \lfloor e_i/c \rfloor \mod \gamma$, and then compute $X^{e_i}$ also in $O(c)$ time using the formula $X^{e_i} = \prod_{0 \leq j < c} T[d_{ij}, j]$. Thus, the computations of all $X^{e_i}$ take $O(c \cdot t)$ time in total. The final step is to sum all of the computed $X^{e_i}$ which takes time $O(t)$.

Summarizing, for any set $S$ of $t$ elements the computation of $h_X(S)$ takes $O(c \cdot t)$. The space needed by the precomputed table $T$ is $O(c \cdot \sigma^{1/c})$.

In the following we will need the technical lemma below:

**Lemma 1.** Given a collection $M$ of $m$ integer sets where each set is a subset of $[0, \sigma - 1]$, a randomly chosen hash function $h_X \in H_P$ for $P \geq m^2\sigma$ will injectively map the collection $M$ to the interval $[0, P-1]$ with probability at least $1/2$.

**Proof.** The lemma is easy to prove. Take any pair of sets $(x, y) \in M^2$. The two sets $x$ and $y$ are mapped to the same hash value by a function $h_X \in H_P$ if and only if
\((h_X(x) - h_X(y)) = 0\). Now \(h_X(x) - h_X(y)\) is a polynomial of degree at most \(\sigma - 1\) over the field \(GF[P]\) which consequently can have at most \(\sigma - 1\) roots. Therefore for any pair \((x, y) \in M^2\) we have that \((h_X(x) - h_X(y))\) can possibly be zero for at most \(\sigma - 1\) different values of \(X\). As we have \(m(m - 1)/2\) such pairs, the number of values of \(X\) for which we have a collision for any of the pairs is at most \(t = (\sigma - 1)m(m - 1)/2\).

We have \(P = \sigma m^2\) and therefore \(t \leq P/2\).

We now sketch how to efficiently find one prime number in the interval \([m^2\sigma, 2m^2\sigma]\).

By well known properties of the distribution of prime numbers, we know that the density of primes below a given number \(N\) is roughly logarithmic in \(N\). This suggests the following simple algorithm: randomly pick a number \(P\) in the interval \([m^2\sigma, 2m^2\sigma]\).

The number \(P\) will be prime with probability \(\Omega(1/\log(m^2\sigma)) = \Omega(1/(\log m + \log \sigma))\).

Then test whether \(P\) is a prime using any efficient deterministic primality testing algorithm that takes time polylogarithmic in \(P\). If \(P\) is not a prime, then repeat the same procedure (pick a random \(P\) in the interval and test its primality) until we get a prime \(P\). Because the probability of \(P\) being prime is \(\Omega(1/(\log m + \log \sigma))\), the expected number of repeated procedures will be \(O(\log m + \log \sigma)\). As a primality testing takes time polylogarithmic in \((m^2 \log \sigma)\) and we are doing \(O(\log m + \log \sigma)\) expected primality tests, we deduce that the total time for finding \(P\) is \(O((\log m + \log \sigma)^c)\) for some constant \(c\).

### 2.2 Succinct Data Structures

#### Succinct Static Function Representation

We will make use of the following result described in \([17]\):

**Lemma 2.** \([16]\) Given a set \(S \subseteq U\) where \(|U| \leq 2^w, |S| \geq \log |U|\) and a function \(f\) from \(S\) into \([0, 2^k - 1]\) (with \(k \leq w\)), we can, in \(O(|S|)\) time build a succinct representation of the function \(f\) that uses \(|S|k(1+o(1))\) bits. Given any element \(x \in S\) the representation returns \(f(x)\) in constant time. Given an element \(x \in U\setminus S\), the representation returns an arbitrary value in \([0, 2^k - 1]\) in constant time.

The result stated in the lemma was first described in \([16]\). It combines the use of a set of hash functions with matrix solving on \(GF[2^k]\) (two similar methods are also described in \([50]\) but have slightly worse performance). The lemma says that we can have a representation of a function \(f\) from \(S \subseteq U = [0, 2^w - 1]\) into \([0, 2^k - 1]\) that can successfully return the correct value for \(f(x)\) when queried for an element \(x \in S\), but returns an arbitrary value for any element \(x\) outside \(S\). Therefore, the representation is unable to detect whether a given element \(x\) is in \(S\) or not. This is why the space usage in the lemma has no dependence on \(U\), but instead only depends on \(k\) and on the cardinality of \(S\) (it is easy to see that in order to detect whether \(x \in S\) we need to store \(S\) in one way or another and thus need to use a space of at least \(\Omega(|S|\log |U|)\) bits).

**Succinctly Encoded Tries (Cardinal trees)** A trie (or cardinal tree) is a tree where each edge has a label from the alphabet \(\Sigma\). The maximal degree in a trie is thus \(\sigma = |\Sigma|\). A standard representation of a trie of \(N\) nodes would need \(O(N \log N)\) bits (essentially the \(\log N\) bits are needed to encode pointers in the trie). In our case we need a succinct representation that uses less than \(O(N \log N)\) bits, ideally close to the information theoretic lower bound which is about \(N \log \sigma + O(N)\) bits. We will thus use the following result described in \([17]\):
**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 explicitly write on the edge for clarity.

**Lemma 3.** Given a trie (cardinal tree) having a total of $N$ nodes over an alphabet of size $\sigma \geq 2$, we can build a representation that uses $N(\log \sigma + \log_2 e + o(1))$ bits of space and supports basic navigation operations in constant time. In particular it supports the following operation in constant time: given a node $p$ having identifier $i_p$ and a character $\alpha$, tell whether $p$ has a child $d$ labeled with character $\alpha$ and return its identifier $i_d$.

The operation cited in the lemma is the only one which will be used in this paper.

### 2.3 Trees

**Suffix Tree** The suffix tree $ST(s)$ is a compact representation of all suffixes of a given sequence $s = s_1...s_n$. It is basically a trie of all suffixes of $s$ where all the 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 it can be built in $O(n)$ time even on integer alphabet using the construction algorithm of [10]. An example of such a suffix tree is given in Figure 1.

We assume below that in the suffix tree each transition interval $[i,j]$ of $ST(s)$ corresponds to the leftmost occurrence of the factor $s_i...s_j$ in $s$. For instance, in Figure 1 the transition from 1 to 2 is the pointer $[1,1] = s_1 = a$. This property is ensured by Ukkonen [18] algorithm, but can also be ensured on every suffix tree by a simple additional $O(n)$ steps.

**Fingerprint Trie** We now present the fingerprint trie (this is called backtracking tree in [3,4]). The fingerprint trie is a tree representation of the fingerprints. The trie representation exploits the property that for every $f \in \mathcal{F}$ such that $|f| \geq 2$ there exists necessarily at least one other fingerprint $g \in \mathcal{F}$ and some letter $\alpha$ such that $g \cup \{\alpha\} = f$. In other words, for every $f \in \mathcal{F}$ there exists some $g \in \mathcal{F}$ such that $f$ can be written as a sequence $\beta_0..\beta_j, \alpha$ (of distinct characters) and $g \in \mathcal{F}$ written as a sequence $\beta_0..\beta_j$.

This property means that the set of fingerprints can be represented as a trie. More
precisely, let $F_i \subseteq \mathcal{F}$ be the subset of the fingerprints of $\mathcal{F}$ where each $f \in F_i$ is of size $i$. At the beginning, we start with a trie which contains only a root. Then we take the subset $F_1$ of all fingerprints in $\mathcal{F}$ consisting of one character. Then for each fingerprint $f \in F_i$ consisting of a character $\alpha$, we create a new node corresponding to $f$ and attach it as a child of the root with a link labeled with the character $\alpha$. Then the remainder of the trie can be built level-by-level: for building level $i \geq 2$, we consider the set $\mathcal{F}_i$ and for each $f \in \mathcal{F}_i$ do the following:

1. First consider a fingerprint $g \in \mathcal{F}_{i-1}$ (represented by a node $q_g$) and a character $\alpha$ such that $g \cup \{\alpha\} = f$ (by the property above there exists at least one such pair $(g, \alpha)$). If there exist several such pairs choose one arbitrarily.
2. Then create a new node $q_f$ and attach it as a child of the node $q_g$ (which corresponds to $g$) with a link labeled with character $\alpha$.

### 2.4 Naming Technique

The naming technique is used to give a unique name to each fingerprint from $\mathcal{F}$. 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 are [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 2 shows a simple example with $\sigma = 8$.

|    | [7] |
|----|-----|
| [5] | 2   |
| [6] | 2   |
| [1] | 0   |
| [1] | 0   |
| [1] | 1   |
| [0] | 0   |

Fig. 2. Naming example.

The names in the fingerprint table are only [0] or [1] and are given as input. 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 2, the name [2] is associated to ([1], [0]) the first time this pair is encountered. The second time, this name is directly retrieved.

**Naming a List of Fingerprint Changes.** Assume that a specific set $S$ of fingerprints can be represented as a list $L = (\alpha_1, \alpha_2, \ldots, \alpha_p)$ of distinct characters such that $S = \{f_1, f_2, \ldots, f_p\}$ where $f_i = \cup_{1 \leq j \leq i} \{\alpha_j\}$.

The core idea of the algorithm of [6] 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 3. We explain it below.

We number the levels from 1, the lowest, to $\log \sigma + 1$. The original list $L$ is first transformed into a list $L_1$ of changes on level 1 by replacing each character $\alpha_i$ by the pair $\{(1), f_{\alpha}(\alpha_i)\}$. To initialize the process we add a list of $\sigma$ pairs $\{(0), i\}, i = 0..\sigma - 1$ at the beginning of $L_1$. 
NAME LISTS(L = (α1, α2, . . . , αp) initial list of changes)
1. L1 ← ([{0}, 0], . . . , [{0}, σ − 1])
2. add ({[1], fS(α1)}, . . . , {[1], fS(αp)}) to end of L1
3. For r = 1..log σ Do
4. FTr ← name table of size σ/2r−1
5. Etp ← first element of Lr
6. For l = 0..σ/2r−1 − 1 Do /* initialization of table FT */
7. {[a], j} ← Etp
8. FTr[j] ← [a]
9. Etp ← next element in Lr
10. End of for
11. Let L′ r be an empty list
12. Etp ← first element of Lr
13. While Etp exists Do
14. {[a], j} ← Etp
15. FTr[j] ← [a]
16. add {((FTr[2[j/2]], FTr[2[j/2] + 1]), [j/2])} to end of L′ r
17. Etp ← next element in Lr
18. End of while
19. sort the pair of names in L′ r in lexicographical order
20. give new names in each unique pair in L′ r
21. build Lr+1 by copying L′ r but replacing each pair by its new name
22. End of for

Fig. 3. Naming a list L = (α1, α2, . . . , αp) of fingerprint changes.

This initial list is then used to compute all names of the cells in the second level. A table FT of σ names temporary records the pair of names to be coded. A list L′ 1 of pairs of names is built as follows. The first σ elements of L1 are read to initialize FT. The list L′ 1 is initialized with σ/2 pairs built by reading FT. Then, the remainder of the list L1 is read and for each new element {[a], j} (1) the table FT is changed in position j by FT[j] ← [a] and (2) the pair {[FT[2[j/2]], FT[2[j/2] + 1]], [j/2]} is 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[j/2]], FT[2[j/2] + 1]).

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 lexicographical order (through a radix sort) and a new name is assigned to each distinct pair of names (n1, n2). A new list L2 is built from L′ 1 (keeping the initial order of L′ 1 and thus of L1) 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 L2 now contains {[2], 1}.

The list L2 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 Llog σ + 1 contains the names of all the fingerprints of S.

Complexity. The sum σ + σ/2 + σ/4 + . . . (lines 1 and 6-10 of pseudo-code in Fig. 3) for all cell initializations is bounded by 2σ. The remaining construction of L1 (line 2) requires Θ(|L|) time. Then a linear sort of Θ(|L|) elements is performed for every level. As there are log σ + 1 levels, naming the list takes Θ(σ + |L| log σ) time.
3 Faster Fingerprint Computation

Let \( q \in \mathcal{L}_C \) and \((i,j)\) be a maximal location of \( q \), then we denote \( st_s(q) \) as the string \( s_i \ldots s_j \). Table 1 shows an example of a copy relation. 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 \( |w_k| = \frac{1}{3} k(k+1)(2k+1) \), \( |\mathcal{L}| = \frac{1}{13} k(3k^3 + 2k^2 - 9k + 16) = \Theta(|w_k|^{4/3}) \), 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 \( k \to \infty \).

Participation Tree Let \( s = s_1 \ldots s_n \) be a simple sequence of characters over \( \Sigma \). In this first phase, for reasons that will become clear below, we add to the sequence a last character \( s_{n+1} = \# \) that does not appear in the sequence. Thus \( s = s_1 \ldots s_n \#_{n+1} \). Let \( i \) and \( j \) be positions in \( s, 1 \leq i \leq j \leq n + 1 \). We define \( f_0(i,j) \) 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 \( s = a_1 b_2 c_3 a_4 d_5 a_6 b_7 a_8 c_9 b_{10} \#_{11} \#_{12} \), \( f_0(3, 9) = abc \) and \( f_0(5, 10) = cabc \).

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

For instance, if \( s = a_1 b_2 c_3 a_4 d_5 a_6 b_7 a_8 c_9 b_{10} \#_{11} \#_{12} \), \( lfo_s(1) = abc \) and \( lfo_s(5) = dabc \).

The participation tree resembles a tree of all \( lfo_s(i) \) in which we removed terminal characters (the need of this removal will appear clearly below). It contains the same path labels. The participation tree allows some redundancy in the path labels, i.e. the same path label might correspond to several paths from the root. Thus, our tree is 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.

Let \( s = s_1 \ldots s_n \#_{n+1} \) where \( s_{n+1} = \# \). The participation tree \( PT(s) \) is built from the suffix tree \( ST(s) \) in the following way. Imagine the suffix tree in an “expanded” version, that is, each edge \([i,j]\) is explicitly written by the corresponding factor \( s_i \ldots s_j \) (see Figure 1). Let us consider the sequence of characters on some path from the root

| Class | Maximal locations | \( st_s(q) \) |
|-------|------------------|--------------|
| 1     | \( a_1 \) | \( a_2 \) |
| 2     | \( a_3 \) | \( a_4 \) |
| 3     | \( a_5 \) | \( a_6 \) |
| 4     | \( a_7 \) | \( a_8 \) |
| 5     | \( a_9 \) | \( a_{10} \) |
| 6     | \( a_{11} \) | \( a_{12} \) |
| 7     | \( a_{13} \) | \( a_{14} \) |
| 8     | \( a_{15} \) | \( a_{16} \) |

| Class | Maximal locations | \( st_s(q) \) |
|-------|------------------|--------------|
| 9     | \( b_1 \) | \( b_2 \) |
| 10    | \( b_3 \) | \( b_4 \) |
| 11    | \( b_5 \) | \( b_6 \) |
| 12    | \( b_7 \) | \( b_8 \) |
| 13    | \( b_9 \) | \( b_{10} \) |
| 14    | \( b_{11} \) | \( b_{12} \) |
| 15    | \( b_{13} \) | \( b_{14} \) |
| 16    | \( b_{15} \) | \( b_{16} \) |

Table 4. Copy relation example for \( s = a_1 b_2 c_3 a_4 b_5 c_6 d_7 a_8 b_{10} \#_{11} \#_{12} \).
and let $\alpha$ be the first character on this path. Let $o$ be the second occurrence of $\alpha$ 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 $\varepsilon$;
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 $\varepsilon$;
3. We then replace the terminal character of each path from the root to a leaf by $\varepsilon$;
4. We replace all multi-character edges by an equivalent series of a single character and a node. An example of such a resulting tree is shown in Figure 4 (left);
5. As a last step, all $\varepsilon$ edges $(p,\varepsilon,q)$ are removed by merging $p$ and $q$. The resulting tree is the participation tree. An example of this last tree is shown in Figure 4 (right).

For each node $q$ of $ST(s)$ and $PT(s)$ we denote by $Suff(q)$ the set of suffixes of $s$ that appear as leaves of the subtree rooted in $q$. We consider below that the suffixes associated to a node in $ST(s)$ remain associated to the node in $PT(s)$, even after the merging. This is shown in Figure 4: 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 4.** Let $s = s_1..s_n$. For all $i = 1,\ldots,n$, each proper prefix of $ifo_s(i)$ labels a path from the root in $PT(s)$.

**Proof.** When nodes are ignored, the reduction of the path of a suffix $i$ in the suffix tree corresponds to $ifo_s(i)$ without its terminal character. \(\square\)

Note that a proper prefix of $ifo_s(i)$ might label several paths from the root in $PT(s)$.

Let $[i, j]$ be an interval on $s = s_1..s_n$ and let $Support([i, j])$ be the minimal position $p, i \leq p \leq j$, of the rightmost occurrences of each letter in $s_i..s_j$. We define $O^{[i,j]}$ as $ifo_s(Support([i, j]), j)$. For instance, if $s = a_1b_2a_3c_4a_6b_7a_8c_9d_{10}#_{11}$, $Support([1,3]) = 2$, $Support([4,10]) = 5$, $O^{[1,3]} = ba$ and $O^{[4,10]} = cabcd$. 

---

**Fig. 4.** From suffix tree to participation tree (right picture) of $s = a_1b_2a_3c_4a_6b_7a_8c_9d_{10}#_{11}$. New nodes are in gray. The $\varepsilon$ transitions are removed in the last step. Attached suffixes are shown in square boxes.
**Definition 4.** Let \( s = s_1 \ldots s_n \) and \( 1 \leq i \leq j \leq n \). We define \( \text{Extend}_s(i, j) \) 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 \( s = a_1 b_2 a_3 c_4 \varepsilon_5 a_6 b_7 a_8 c_9 d_{10} \#_{11}, \) \( \{1, 4\} = \text{Extend}_s(2, 4) \) and \( \{1, 9\} = \text{Extend}_s(2, 7) \)

**Lemma 5.** Let \((i, j)\) be a maximal location of \( s = s_1 \ldots s_n \). There exists a permutation of all characters of \( C_s(i, j) \) that labels a path from the root in \( PT(s) \).

**Proof.** \( O^{(i, j)}_s \) is obviously a permutation of \( C_s(i, j) \) and a proper prefix of \( \text{lfo}_s(\text{Support}((i, j))) \), which, by lemma 4, labels a path from the root in \( PT(s) \).

**Corollary 1.** Let \( s = s_1 \ldots s_n \). 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 \( s_i \ldots s_j \) to \((k, l) = \text{Extend}_s(i, j)\) in which it is contained. Then \( C_s(i, j) = C_s(k, l) \) and lemma 5 applies. \( \Box \)

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

\[
\Phi : \text{SPref}(s) \longrightarrow \mathcal{P}(\mathcal{L}) \quad z \quad \mapsto \quad \{ (k, l) \in \mathcal{L} \mid O^{(k, l)}_s = w(z) \text{ and } \text{Support}((k, l)) \in \text{Suff}(z) \}
\]

**Lemma 6.** Let \( z = ((r, \alpha_1, p_1), \ldots, (p_{i-1}, \alpha_i, p_i)) \) be a non-empty path in \( \text{SPref}(s) \). Then \( \Phi(z) \neq \emptyset \).

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

We prove now that \( \text{Support}((k, l)) = m \). As \( \alpha_1 \ldots \alpha_i \) is a proper prefix of \( \text{lfo}(m) \), there exists an \( \alpha = \text{lfo}(m)_{i+1} \) such that there is no occurrence of \( \alpha \) in the interval \([m, p]\), and thus after the extension of \([m, p]\) to a maximal location \((k, l)\), the indice \( l \) is strictly less than the indice of the first occurrence of \( \alpha \) after \( m \). As, by definition of \( \text{lfo}(m) \), there is no occurrence of \( s_m \) before the indice of \( \alpha \) after \( m \) in \( s \), there is no other occurrence of \( s_m \) at the right of \( s_m \) in the interval \([m, l]\). Moreover, since all characters in \( \alpha_1 \ldots \alpha_i \) and only them appear after \( m \) in \([m, l]\) in the order of \( \alpha_1 \ldots \alpha_i \) and the extension procedure ensures that all characters in \([k, m]\) are characters from \( \alpha_1 \ldots \alpha_i \), we have \( \text{Support}((k, l)) = m \).

Finally, it is obvious that \( O^{(k, l)}_s = O^{[m, p]}_s = \alpha_1 \ldots \alpha_i = w(z) \), and thus \((k, l) \in \Phi(z)\). \( \Box \)

**Lemma 7.** Let \( z_1, z_2 \in \text{SPref}(s) \) be two distinct non-empty paths. Then \( \Phi(z_1) \cap \Phi(z_2) = \emptyset \).

**Proof.** Assume a contrario that there exists \((k, l) \in \Phi(z_1) \cap \Phi(z_2)\). Let \( m = \text{Support}((k, l)) \), \( m \in \text{Suff}(z_1) \) and \( m \in \text{Suff}(z_2) \). Thus one of the paths is a prefix of the other. As \( O^{(k, l)}_s = w(z_1) = w(z_2) \), the two paths must be equal, which contradicts the hypothesis. \( \Box \)
Lemma 8. Let \((i, j)\) and \((k, l)\) be two distinct maximal locations of \(s = s_1 \ldots s_n\) in the same equivalence class of \(L_C\). Then there exists \(z \in \text{SPref}(s)\) such that both \((i, j)\) and \((k, l)\) are contained in \(\Phi(z)\).

Proof. Let \(m_1 = \text{Support}((i, j))\) and \(m_2 = \text{Support}((k, l))\). As \(s_i \ldots s_j = s_k \ldots s_l\), \(u = s_{m_1} \ldots s_j = s_{m_2} \ldots s_l\) 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 \(w(z) = O^{(i, j)} = O^{(k, l)}\), so \(m_1, m_2 \in \text{Suff}(z)\). Thus \((i, j), (k, l) \in \Phi(z)\). \(\Box\)

Theorem 1. Any maximal location is contained in the image \(\Phi(z)\) of some path \(z\) in \(PT(s = s_1 \ldots s_n)\), and the size of \(PT(s)\) (without the initial positions of suffixes) is \(O(|L_C|)\).

Proof. Lemma 8 directly implies that all maximal locations are in the image \(\Phi(z)\) of a path \(z\) in \(PT(s)\). As by lemma 7 the images \(\Phi(z)\) are non-overlapping, they form a partition of \(L\). Lemma 8 ensures that \(L_C\) partition is a subpartition of the partition formed by the images of \(\Phi\). As by lemma 6 there is no empty image, the number of such images is smaller than or equal to \(|L_C|\). \(\Box\)

Note that we considered the size of \(PT(s = s_1 \ldots s_n)\) without the initial positions of suffixes (square boxes in Figure 4). With these positions, the size of \(PT(s)\) is \(O(n + |L_C|)\).

We explain below how to compute the participation tree from the suffix tree in linear time.

**From Suffix Tree to Participation Tree** We extend the notion of \(\text{efo}(i, j)\) keeping the positions of the characters in \(s = s_1 \ldots s_n\). We define \(\text{efo}(i)\) 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 \(s = a_1b_2a_3c_4e_5a_6b_7c_8d_9d_{10}\#_{11}\), \(\text{efo}(3) = a_3c_4e_5b_7d_{10}\#_{11}\) and \(\text{efo}(5) = e_5a_6b_7c_8d_{10}\#_{11}\).

The idea of the algorithm is the following. For each transition \((i, j)\) on the path of a longest suffix \(v = s_k \ldots s_n\), we compute the “participation” of the edge to \(\text{efo}(k)\) that is, the new characters the edge brings in \(\text{efo}(k)\). For instance, in Figure 3 the participation of edge \((6, 8) = [5, 11]\) is \(\epsilon\), since it is on the path of the suffix \(s_3 \ldots s_n\) and \(\text{efo}(3) = ace\). The participation of edge \((12, 14) = [5, 11]\) is \(\text{cabc}\) since \(\text{efo}(4) = \text{ceab}\).

To compute the participation of interval \([i, j]\) on the path of a suffix \(v = s_k \ldots s_n\), we use \(\text{efo}(k)\) and also the next position of \(s_k\) after \(k\) in \(s\), if it exists. Assume it is the case and let \(p\) be this position. Thus \(s_p = s_k\). Let \(\text{efo}(k) = s_{k1} s_{k2} \ldots s_{k_s}\) and \(l_k \leq p \leq l_{k,s+1}\). If \(i \geq p\), the participation of \([i, j]\) is the empty word \(\epsilon\). Otherwise, if \(i < p\) then the participation of \([i, j]\) is the string (potentially empty) \(s_{i1} \ldots s_{il}\) with

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

[Note that this computation requires that the interval \([i, j]\) which annotates a transition in the suffix tree corresponds to the suffix \(v\) used as reference. In order to ensure this, below we "shift" each interval \([i, j]\) according to the suffix we are currently reading before computing its participation.]

For instance, in Figure 4 \(\text{efo}(2) = b_2a_3c_4e_5d_{10}\#_{11}\) and \(p = 7\) since 7 is the next position of \(b\) after position 2. Thus, participation of edge \((1, 9) = [2, 4]\) is \(b_{2}a_3c_4 = \text{bac}\).
Fig. 5. Building the participation tree from the suffix tree.

participation of (9, 11) = [5, 11] = e5 = e (since \( p = 7 \)). For each suffix \([k, n]\), given \( efo_s(k) \) and \( p \), a bottom-up process from leaf \( k \) to the root of the suffix tree allows us to:

(a) shift the pointed positions to positions corresponding to the suffix considered. The bottom-up approach allows to read the suffix from its end, and thus the sizes of the encountered transitions are enough to know which segment of the suffix the edge represents;
(b) compute the participation of each (not previously touched) edge on this path.

Also, the bottom-up approach allows us to avoid unnecessary computation, since the participation of an upper edge ends in \( efo_s(k) \) where the participation of the lower begins.

We modify the suffix tree using successive \( efo_s(k) \), for \( k = n..1 \). A sketch of this algorithm is given in Figure 5. At the end of this process, we first replace the terminal character of all paths from the root by \( \varepsilon \). We finally remove all \((u, \varepsilon, v)\) edges by merging \( u \) and \( v \).

**Theorem 2.** The participation tree of \( s = s_1..s_n \) can be built in \( O(n + |C|) \) time and \( O((n + |C|) \log n) \) bits of space.

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

For each suffix \([k, n]\), given \( efo_s(k) \) and \( p \), the bottom-up process from leaf \( k \) to the root of the suffix tree can be done in \( O(1) \) time for each unmarked node.

We maintain each \( efo_s(i) \) as a combination of a doubly linked list and an array of size \( \Sigma \) in which each cell \( j \) points to the position of character \( f^i_{\Sigma}(j) \) in the doubly linked list. Thus, adding a character \( c \) to the head of the doubly linked list while recording its position in the corresponding cell of the array is \( O(1) \). Removing a character out of the list is also \( O(1) \) since it suffices to find its position in the list using the array and remove the character using the pointer to the previous and next character in the list.
Initializing the structure is $O(\sigma)$ but it has only to be done once. In addition to the array and the doubly linked list, a pointer $tp$ points to the character in the list whose position is just before $p$ (the next position of $s_i$ in $s$) if such character exists or to the end of the list otherwise. An instance of this structure is given in Figure 6.

Fig. 6. Data structure for maintaining $efo(i)$ shown on $efo_s(2) = b_2a_3c_4e_5d_{10}\#_{11}$. The pointer $tp$ points to the character in the list whose position is the largest smaller position in the list compared to the next position $p$ of $b$ in $s$, which is 7.

Assume that $efo_s(i)$ is represented in this way, with knowing $tp_i$, the next position in the doubly linked list of the first character $s_{i-1}$ in $efo_s(i)$. To compute $efo_s(i - 1)$ and $tp_{i-1}$, it suffices to test in the array if $\alpha = s_{i-1}$ already appears in the list. If yes, $tp_{i-1}$ points to the character just before $\alpha$ in the list, if not $tp_{i-1}$ is set to the end of the list. Then $\alpha$ is removed out of the list and inserted at its head. The first $efo_s(n)$ is simply $s_n$, and $tp$ points to the end of the list.

Computing the participation of each non-touched edge on a path from the root to a leaf corresponding to suffix $i$ in a bottom-up manner is not expensive since it suffices to “consume” $efo_s(i)$ backward from $tp_i$ edge after edge as soon as an edge $[k, l]$ (shifted to correspond to suffix $i$) is such that $k$ is less than the position of the element pointed by $tp_i$. Thus, calculating the participation of each edge in the suffix tree can be done in a time proportional to the participation of the edge in $PT(s)$ tree plus the total number of edges in the tree.

Replacing the terminal character of each path from the root by $\varepsilon$ is $O(n)$. Merging each of the $\varepsilon$ edges can also be performed in $O(n)$ since each such $\varepsilon$ edge is either a previous edge of the suffix tree or was labeled by a single terminal character of a path from the root. The whole construction of $PT(s)$ is thus $O(n + |LC|)$ time.

The space required is the size of the suffix tree plus the size of the participation tree plus the size of the data structure representing $efo_s(i)$, thus $O(n + |LC|)$ space.

We now explain how to name all fingerprints from the participation tree.

**Naming a Participation Tree** The naming approach of the previous section has been modified in [14] 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 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, except that the initialization of the naming list is done once for all paths. Corollary 1 guarantees our approach. The NAME_FINGERPRINT algorithm names all fingerprints. Its pseudo-code is given in Figure 7.
**NAME_FINGERPRINT**(PT(s))

1. \( n_{init} \leftarrow 0 \)
2. \( \text{for} \ k = 1..\log \sigma \ \text{do} \)
3. \( FT_k \leftarrow \text{name table of size } \sigma/2^{k-1} \text{ all initialized to } n_{init_k} \)
4. \( \text{DEPTH_FIRST_SEARCH}(FT_k, \text{Root}(PT(s))) \)
5. \( SL \leftarrow \emptyset \) /* empty stack */
6. \( \text{for all edges } e = (p, \alpha, q) \text{ in } PT(s) \ \text{do} \)
7. \( \{(n_1, n_2), j\} \leftarrow \Delta(p, \alpha, q) \)
8. \( \text{Add } (n_1, n_2) \text{ to } SL \)
9. \( \text{end of for} \)
10. \( \text{add the couple } (n_{init_k}, n_{init_k}) \text{ to } SL \)
11. \( \text{sort } SL \text{ in lexicographical order} \)
12. \( \text{give new names for each different couple in } SL \)
13. \( \text{replacing each pair in } \Delta(p, \alpha, q) \text{ by its new name} \)
14. \( \text{end of for} \)

**Fig. 7.** Naming all fingerprints in a participation tree \( PT(s) \).

As in the list naming of section 2.4 \( \log \sigma \) iterations are performed, one by fingerprint array level (loop 11-24), the lowest one excepted. With each edge \((p, \alpha, q)\) of \( PT(s) \) a value \( \Delta(p, \alpha, q) \) is associated. 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 \( \{[1], \Sigma(\alpha)\} \) 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 13) 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 2.4. Note that in **DEPTH_FIRST_SEARCH** a special \( FT \) table is modified (line 5) before the recursive call but reinitialized to the previous value after the call (line 8). This permits to initialize the table \( FT \) only once before the first call to **DEPTH_FIRST_SEARCH** (line 12) and thus the initialization costs are the same for all paths as for a single list, and thus are bounded by \( 2\sigma \).

After the depth first search the values \( \Delta(p, \alpha, q) \) are collected on all the edges \((p, \alpha, q)\) of the participation tree (lines 14-18) in a list \( SL \). This list is lexicographically sorted and a new name is given to each unique pair (line 20), similarly to the naming of a single list in section 2.4. The initial 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 \((n_{init_k}, n_{init_k})\) is added to the list of names (line 19) and its new name is retrieved after the sorting and the renaming (line 22).

**Theorem 3.** The Name\_fingerprint algorithm applied on \(PT(s)\) names all fingerprints of \(s\) in \(\Theta(\sigma + |L_C| \log \sigma)\) time using \(O(|L_C| + |F| \log \sigma)\log n\) bits of working space.

## 4 A Space Efficient Fingerprint Representation

### 4.1 Overview

In this section we show how the fingerprint set can be represented in just \(|F| (2 \log \sigma + \log_2 e)(1 + o(1))\) bits of space instead of \(O(|F| \log n)\) bits. Our solution is particularly attractive whenever \(\sigma\) is sufficiently small (e.g. \(\log \sigma = o(\log n)\)) as it saves a factor \(\Theta(\frac{\log_2 e}{\log \sigma})\) compared with a standard non-succinct representation that uses at least \(\Theta(|F|)\) words of space, which translates into \(\Theta(|F| \log n)\) bits.

Our representation relies on the fingerprint trie as described in section 2.3.

Before describing our solution, we first recall some basic facts on the fingerprint trie that will be needed to understand our solution. First, recall the following two facts:

1. Each node in the trie corresponds to a unique set and each set corresponds to a unique node.
2. Each prefix of a fingerprint is also a fingerprint.

Note also that the fingerprint trie implies an ordering on the characters of any given fingerprint represented in the trie. More precisely for a given node \(q\), the characters of the corresponding fingerprint \(f_q\) are ordered according to the order in which they appear as labels of the nodes in the path from the root to the node \(q\).

In our representation, the fingerprint trie will be represented in two different ways. This is why the space usage will be at least \(2|F| \log \sigma\) bits. The first representation will permit a traversal of the fingerprint trie bottom-up (climb the trie) and the second one will permit a traversal of the fingerprint trie top-down (descend the trie). If the fingerprint is represented in the trie, then a bottom-up traversal will permit one to get the proper ordering on the fingerprint characters. Then, the presence of the fingerprint can be confirmed by a top-down traversal. Note that this second traversal can only return true if the fingerprint exists and is in the correct order represented in the trie. Therefore a top-down traversal will never return a false positive answer (it will never return true for a fingerprint not represented in the trie or for fingerprint represented in the trie but with a different ordering). Likewise, this top-down traversal will never return a false negative (it will always give a positive answer for an existing fingerprint) as it will be proven later that a bottom-up traversal will always return the correct ordering of the characters of an existing fingerprint and this correct ordering will thus be used to do a successful top-down traversal of the trie.

We now give more details on our representation. First, notice that each set (fingerprint) uniquely corresponds to a distinct node of the fingerprint trie. Let \(f_q\) denote the fingerprint associated with the node \(q\). Let \(\alpha(q_1, q_2)\) denote the characters that label the edge which connects a node \(q_1\) to its child \(q_2\). Notice that by definition of the fingerprint trie for any node \(q_2\) having a parent \(q_1\), we have \(f_{q_2} = f_{q_1} \cup \{\alpha(q_1, q_2)\}\). That is, the fingerprint of the node \(q_2\) is obtained by adding one character \(\alpha(q_1, q_2)\) to
the fingerprint of its parent node $q_1$.

The solutions we propose are able to find whether a given query fingerprint $f$ is in the set $\mathcal{F}$ in $O(|f|)$ time. A query for a fingerprint $f$ represented by a string which contains all the characters of $f$ in an arbitrary order will work in three steps:

1. We query the bottom-up representation of the trie, which, when given the fingerprint $f$, returns a string $s$ of length $|f|$. This bottom-up representation relies on the use of succinct function representation of lemma 2. A detailed description of the step is in section 4.2.

2. We check whether the string $s$ is a permutation of the set $f$. That is, we check whether $s[i] \in f$ for each $i \in [0, |f| - 1]$ and check also that all characters of $s$ are distinct. This step is done in time $O(|f|)$ with high probability using $O(|f| \log \sigma)$ bits working space or in deterministic time $O(\epsilon |f|)$ using working space $O(\sigma^{1/4} \log \sigma)$ bits for any positive integer $\epsilon$. A detailed description of the step is in section 4.4.

3. The final step is using the succinct top-down representation of the trie to do a top-down traversal for the string $s$. This step permits checking whether the string $s$ exists in the trie representation in $O(|s|) = O(|f|)$ time. Notice that this is equivalent to checking that $f \in \mathcal{F}$. This is the case as by previous step we have checked that $s$ is a permutation of $f$ and we know that the trie stores a unique string corresponding to each fingerprint in $\mathcal{F}$. A detailed description of the step is in section 4.3.

In the following three subsections we describe in more detail the data structures used for each of the three steps. In subsection 4.5 we give the full picture of the query and prove its correctness.

### 4.2 Backtracking Function (bottom-up trie representation)

The first step is achieved through a data structure we call the backtracking function, which is in fact a bottom-up representation of the trie. This function associates to each fingerprint $f_i$ the last character in its string representation $s_i$. We will simply use a static function that maps each set (fingerprint) to the last character in the character ordering. In other words whenever we have a fingerprint $f$ corresponding to a node $q$ in the fingerprint trie, we associate with $f$ the character which labels the edge which connects $p$ to $q$. That is, for each set we have a string representation that contains exactly the same characters as the set in a certain order. With each set we associate the last character in its string representation.

It turns out that representing this backtracking function can be done using just $(|\mathcal{F}| \log \sigma)(1 + o(1))$ bits of space which is optimal. The generation of the backtracking function from the set $\mathcal{F}$ can be done in optimal $O(|\mathcal{F}|)$ time. The generation is based on the use of a polynomial hash function (the same used in the so-called Rabin-Karp fingerprints [13]). The first step consists in a top-down traversal of the fingerprint trie. Recall that each node represents a distinct fingerprint. Given a node $q$ with a parent $p$, we note the fingerprint associated with $p$ by $f_p$ and the fingerprint associated with $q$ by $f_q$. Then, if the edge which connects $p$ to $q$ is labeled by character $\alpha$, we will have $f_q = f_p \cup \{\alpha\}$. So, during the top-down traversal of the trie we will compute a hash value associated with each fingerprint. For that we will make use of the polynomial hash functions family as described in section 2.1. More precisely, the hash functions we will use are polynomials modulo a prime $P$ chosen such that $P \in [\mathcal{F}^2 \sigma, 2|\mathcal{F}|^2 \sigma]$. Finding $P$ takes time $O((\log(|\mathcal{F}|^2 \sigma))^c) = O((\log(|\mathcal{F}| + \log \sigma))^c)$ for some constant $c$. (see 2.1 for details on the algorithm used to find $P$.)
Before beginning the top-down traversal of the trie, we will randomly choose a number \( r \) from the interval \([0, P - 1]\). For any fingerprint \( f_i \) having elements \( \alpha_1, \alpha_2, \ldots, \alpha_{|f_i|} \), we will associate the hash value computed using the formula 
\[
H(f_i) = r \cdot \Sigma(\alpha_1) + r \cdot \Sigma(\alpha_2) + \ldots + r \cdot \Sigma(\alpha_{|f_i|})
\]
where multiplications and additions are all done modulo \( P \).

Now the generation of the hash values for all fingerprints is done in the following way:
we first associate the hash value 0 with the root node which does not represent any fingerprint. We note by \( H_q \) we first associate the hash value 0 with the root node which does not represent any fingerprint. We note by \( H_q \) the hash value associated with the node \( q \) and by \( H_p \) the hash value associated with node \( p \). From the definition it is evident that \( H_q = H_p + r \cdot \Sigma(\alpha) \) where \( \alpha \) is the character which labels the edge connecting node \( p \) to node \( q \). Therefore, during a top-down traversal of the trie, we can compute the hash value for each fingerprint in constant time given the fingerprint of its parent node. Once we have generated the \(|\mathcal{F}| \) hash values corresponding to the \(|\mathcal{F}| \) fingerprints, we will check whether all fingerprints are distinct. According to lemma \[1\] we deduce that this is the case with probability of at least \( 1/2 \). If this is not the case, we will choose a new value \( r \) and recompute the hash values in the same way during a top-down traversal of the trie.

As on expectation we will do \( O(1) \) trials and each trial taking time \( O(|\mathcal{F}|) \), we deduce that the total expected time is \( O(|\mathcal{F}|) \).

Once we have successfully mapped all the keys to distinct hash values in range \([0, P - 1]\), we will store a static function using lemma \[2\] which for each fingerprint \( f_i \) will associate the character \( f \cdot \Sigma(\alpha_i) \) (where \( \alpha_i \) is the last character in \( f_i \)) to the hash value \( H(f_i) \). The space used by the static function will clearly be \(|\mathcal{F}|(\log \sigma)(1 + o(1)) \) bits.

### 4.3 Deterministic and Probabilistic Set Equality Testing

We now describe a method to test for set equality. This is step 2 in our query algorithm. Given two strings \( s_1 \) and \( s_2 \) where \(|s_1| = |s_2|\), we would wish to test whether the two strings are permutations of the same set. That is, we are asking if we can obtain the string \( s_1 \) by doing a permutation on the characters of the string \( s_2 \). We propose two solutions for this problem. The first one is randomized while the second one is deterministic. The two solutions are folklore, but we describe them here for completeness.

**Randomized Method** The randomized method works in the following way: we use a dynamic perfect hash table \[3\] (or any other efficient hash table implementation) in which we insert all the characters of the string \( s_1 \). This takes time \( O(|s_1|) \) with high probability and uses space \( O(|s_1| \log \sigma) \) bits \[4\].

During the insertion, we can easily check that the characters of \(|s_1|\) are all distinct by checking that every character of \( s_1 \) is not present in the table at the time of its insertion. In the hash table, we associate a bit with each key and we initialize the bit to zero. Now, we process the string \( s_2 \). For each character \( \alpha \) of \( s_2 \) we query the perfect hash table for the character \( \alpha \). In case we do find it, we mark the bit associated with it. After we have processed all characters of \( s_2 \), we check if all the bits associated with characters of \( s_1 \) are now set to one. If this is the case, we conclude that \( s_2 \) and \( s_1 \) are permutations of the same set.

---

\[3\] To declare that two strings are equal we require that the two strings are permutations. That is, the characters of each string are all distinct.

\[4\] A linear-space hash table needs \( O(\log |U|) \) per element where \( U \) is the universe. In our case \( U = \Sigma \) and thus \(|U| = \sigma|\).
Clearly this randomized method uses $O(|s_1|)$ words of space that is $O(|s_1| \log \sigma)$ bits of space, which is optimal up to a constant-factor, as we also need $|s_1| \log \sigma$ bits to represent $|s_1|$.

**Deterministic Method** We now describe a deterministic method which can be used to do equality testing. The basic method needs $\sigma$ bits of working space for queries and checks set equality in optimal time $O(|s_1|)$. A more sophisticated method could use space $O(\sigma^{1/k} \log \sigma + |s_1| \log \sigma)$ bits and answers set equality in time $O(k|s_1|)$ for any integer $k$ such that $k > 1$. In the basic method, we will simply use a bitvector $B$ of $\sigma$ bits. At the beginning all the bits in $B$ are set to zero, and we require that they are reset to zero after each equality test.

The equality test works in the following way: we first process the string $s_1$. For each $i$ in $[0, |s| - 1]$, we set $c = f_\Sigma(s_1[i])$ and then set $B[c] = 1$. Before setting $B[c] = 1$, we check that $B[c] \neq 0$ and thus that the character $s_1[i]$ does not occur twice in $s_1$.

We now traverse the string $s_2$. For each $i$ in $[0, |s| - 1]$, we set $c = f_\Sigma(s_1[i])$ and check that $B[c] = 1$. If this was the case, then we set $B[c] = 0$, otherwise, we declare that $s_1$ and $s_2$ are two distinct strings. Setting $B[c]$ to zero is necessary to ensure that all the characters of $s_2$ are all distinct.

It is easy to see that the above procedure correctly computes the equality of $s_1$ and $s_2$. In the first phase we have set all the $|s_1|$ distinct bits corresponding to characters of $s_1$. In the second phase, we check that the bits corresponding to characters of $s_2$ are all distinct and all set which can only be the case if those bits are precisely the $|s_1|$ bits corresponding to character of $|s_1|$.

At the end of checking, if the two strings are equal, then all the bits of $B$ are set to zero, so that $B$ is ready for the next query. If the two strings are not equal, then we need to traverse the string $s_1$ and clear the bits of $B$ which were set to one when $s_1$ was first traversed (we set $B[c] = 0$ for every $c = f_\Sigma(s_1[i])$)

**Lemma 9.** We can do equality testing between two strings $s_1$ and $s_2$ over an alphabet of size $\sigma$ in time $O(|s_1|)$ using $\sigma$ bits of working space.

We now describe the more sophisticated method. We only describe how to achieve $O(\sqrt{\sigma})$ space. The generalization to $O(\sigma^{1/k})$ space for $k > 2$ can easily be deduced from the case $k = 2$.

The method works in the following way: we first partition the characters of $s_1$ according to their $\lceil \log \sigma/2 \rceil$ most significant bits. We also do the same partitioning for the characters of $s_2$. Finally, we compare all the pairs of partitions (one from $s_1$ and one from $s_2$) in which the characters share the same $\lceil \log \sigma/2 \rceil$ most significant bits.

We now give the details of the implementation. We use a table $T_1$ with $2^{\lceil \log \sigma/2 \rceil} \leq 2\sqrt{\sigma}$ cells where each cell $T_1[i]$ contains a pointer (denoted by $T_1[i].P$) to a list of characters. At the beginning we suppose that every $T_1[i].P$ is initialized to null meaning that all the lists are empty. We also use a list $L_1$ which stores a list of non-empty cells (cells with non null pointers) of $T_1$. At the beginning we process the characters of $s_1$ one by one and for each character $\alpha_i$ do the following steps:

1. Compute $j = MSB(f_\Sigma(\alpha_i))$, the $\lfloor \log \sigma/2 \rfloor$ most significant bits of $f_\Sigma(\alpha_i)$.
2. Save in variable $oldP$ the old value of $T_1[j].P$.
3. Add $\alpha_i$ to the list $T_1[j].P$.
4. If $oldP$ equals null, add $j$ to the list $L_1$. That is, the list $T_1[j].P$ which was previously empty is added to $L_1$ as now it is non-empty.
At the end of the third step we are left with two lists $L_1$ and $L_2$ according to the list $L_j$. In this step we process the characters of $s$ one by one and for each character $\alpha_j$, we add $\alpha_j$ to the list $T_j.P[j]$. In the third step, we use two lists $L_1'$ and $L_2'$ initially empty. We take the list $L_1$ and for each element $j$ in the list do the following:

1. Add all elements of the list $T_1[j].P$ at the end of the list $L_1'$.
2. Add all elements of the list $T_2[j].P$ at the end of the list $L_2'$.

At the end of the third step we are left with two lists $L_1'$ and $L_2'$ which are sorted according to the list $L_1$. That is, in each of the two lists we have first all characters whose $\lceil \log \sigma/2 \rceil$ most significant bits are equal to $L_1[0]$ followed by all characters whose most significant are equal to $L_1[1]$ etc. Thus, to finish the equality testing it suffices for every $j$ in the list $L_1$ to do the following:

1. First advance in $L_1'$ in order to find $R_{j1}$ the longest run of $t_1$ characters in $L_1'$ whose $\lceil \log \sigma/2 \rceil$ most significant bits are equal to $j$.
2. Similarly, advance in $L_2'$ to identify $R_{j2}$ the longest run of $t_2$ characters in $L_2'$ whose $\lceil \log \sigma/2 \rceil$ most significant bits are equal to $j$.
3. Check that $t_1 = t_2$. If this is not the case, immediately declare that $s_1$ is distinct from $s_2$.
4. Otherwise we check for the equality of the characters in $R_{j1}$ and $R_{j2}$. To this end we already know that they have the same $\lceil \log \sigma/2 \rceil$ most significant bits, so that we only need to do equality testing for the $\lceil \log \sigma/2 \rceil$ least significant bits between characters of $R_{j1}$ and $R_{j2}$, which can be done using the procedure of lemma 9. This will take time $O(t_1)$ and needs to use just a bitvector of size $2^{\lceil \log \sigma/2 \rceil} \leq \sqrt{\sigma}$ bits.

If all the iterations are completed, we immediately deduce that the two sets $s_1$ and $s_2$ are equal. Concerning the running time, it is clear that the above procedure runs in time $O(|s_1| \log \sigma)$. Every element of $L_1'$ and $L_2'$ is only traversed twice, the first time for determining the length of the runs and the second time for determining the equality between elements of two runs. Each time an element is traversed, only a constant number of operations are carried on.

We now analyze the space usage. The total space needed to store the different lists will be upper bounded by $O(|s_1| \log \sigma)$. The table $T_1$ will use space $O(\sqrt{\sigma} \log \sigma)$ bits, while the bitvector $B$ will use space $O(\sqrt{\sigma})$ bits.

The above algorithm can be easily generalized to use space $(\sigma^{1/k} \log \sigma)$. For that it suffices to do the partitioning of the characters of $s_1$ and $s_2$ in $k - 1$ phases. The $\log \sigma$ bits of the characters are divided in slices of size about $\log \sigma/k$ bits each. Then in each phase we partition the keys according to a one of the slices starting from the most significant slice to the least significant. After $k - 1$ partitioning we will be left with partitions which only differ in their (at most) $\log \sigma/k$ least significant bits. In the final phase, pairs of partitions (one from $s_1$ and one from $s_2$) can easily be matched as was done above using lemma 9.

**Lemma 10.** Given any two strings $s_1$ and $s_2$ of equal length, testing for the equality of the multisets induced by $s_1$ and $s_2$ can be done:

1. In expected $O(|s_1|)$ time with high probability using only $O(|s_1| \log \sigma)$ bits of space.
2. In worst case $O(k|s_1|)$ time using $(\sigma^{1/k} \log \sigma)$ bits of space.
4.4 Succinct Trie Representation (top-down trie representation)

The third step of a query uses a top-down trie representation which we describe in this section. First of all, a trie $Tr$ of size $N$ over an alphabet $\sigma$ can be represented compactly to use optimal space $N(\log \sigma + \log_2 e + o(1))$ using the representation described in [17] permitting many navigation operations on the trie in constant time. In particular, a top-down traversal of the trie for a string $s$ can be done in time $O(|s|)$ by using $O(1)$ time at each step $i$ of the traversal which consists in finding the child labeled with character $s[i]$. Given a string $s$, we can determine whether $s \in S$ in time $O(|s|)$, by doing a top-down traversal of the trie. Thus, given the set $F$ of fingerprints in a trie of size $|F|$, we can succinctly encode the trie representing the set $F$ in time $O(|F|)$ so that the trie uses space of $|F|(\log \sigma + \log_2 e + o(1))$ bits. A top-down traversal of the trie will take time $O(1)$ time per traversed node. Thus given a fingerprint $f$ in the correct order, we can check whether it is presented in the set $F$ by doing a top-down traversal of the succinctly encoded trie representing the set $F$.

4.5 Putting Things Together

We are now ready to describe the full details of the queries on our data structures described in the previous subsections. A query for a fingerprint $f = \{\alpha_1, \alpha_2, \ldots, \alpha_{|f|}\}$ is given as a string $s_f$ of characters consisting in the concatenation of the characters $\alpha_1, \alpha_2, \ldots, \alpha_{|f|}$. The characters are not necessarily lexicographically sorted. The query involves the following steps:

1. Compute the hash value:

$$H(f) = \sum_{1 \leq i \leq |f|} r^{f_\Sigma(\alpha_i)}$$

This operation takes time $O(|f|)$, as it involves only $O(|f|)$ arithmetic operations. In the following we note $f$ by $f[1]$ and note $H(f)$ by $H_j$.

2. Probe the backtracking function using the hash value $H[|f|] = H(f)$, retrieving a character $\beta_j$ (actually retrieving $f_\Sigma(\beta_j)$ then use the reverse mapping $f_\Sigma^{-1}$ to get $\beta_j$). Then we do $|f| - 1$ steps, computing for each $j \in [1, |f| - 1]$ the hash value $H_{j-1} = H_j - r^{f_\Sigma(\beta_j)}$ and probe the backtracking function using the hash value $H_{j-1}$ retrieving the character $\beta_{j-1}$. At the end of the $|f| - 1$ steps we will have obtained a sequence $s'_f = \beta_{|f|}, \beta_{|f|-1}, \ldots, \beta_1$ of characters. Suppose that $f \in F$. When queried with the hash value $H_j$, the backtracking function would return in this case the last character of the fingerprint representation of $f$. Then $f_{j-1} = f_j/\{\beta_j\}$ would also represent another fingerprint from $F$. More generally we will have $f_j \in F$ for every $j \in [1, |f|]$ with $f_j = \{\beta_1, \beta_2, \ldots, \beta_j\}$

3. The third step is to apply the method described in section 4.3 in order to determine whether the set of characters in $s'_f$ equals the set of characters in $f$. If the two sets differ, we immediately conclude that $f \notin F$.

4. Finally we do a top-down traversal of the succinctly encoded trie described in section 4.4 for the string $s'_f$. Here if the traversal fails before attaining a leaf, we immediately conclude that $f \notin F$, otherwise conclude that $f \in F$.

Now we can more precisely describe what is happening inside the data structure. We have to analyze two cases, the case $f \in F$ and the case $f \notin F$. For that we first prove the following lemmata:
Lemma 11. Let $f \in F$. Then

1. for each $j \in [1, |f|]$, $f_j \in F$;
2. the string $s'_f$ is stored in the fingerprint trie.

Proof. The proof of fact 1 is by induction: $f$ is a valid fingerprint (by assumption) which means that the backtracking function returns the last character $\beta_j$ in the trie representation of $f$. Then we know that there exists some $f_{j-1} \in F$ such that $f_{j-1} \cup \{\beta_j\} \in F$. The base case of the induction is for $j = 1$ (fingerprint consists of a single character $\beta_1$) in which case we clearly have a child of the fingerprint root labeled with character $\beta_1$.

The proof of fact 2 can also be obtained by induction. Assume that the assertion is true for a fingerprint $f_{j-1}$ of length $j-1$. Then it can be proved for a fingerprint $f_j$ of length $j$, i.e. the assumption says that the sequence $s'_{f_{j-1}} = \beta_1, \beta_2, \ldots, \beta_{j-1}$ is a permutation of $f_{j-1}$. We know that the backtracking function returns a character $\beta_j$ which is the last character of the representation of $f_j$ in the fingerprint trie and that there exists a fingerprint $f_{j-1}$ of size $j-1$ such that $f_{j-1} \cup \{\beta_j\} \in F$. As we know that fact 2 is true for $f_{j-1}$, it means that the sequence $s'_{f_{j-1}} = \beta_1, \beta_2, \ldots, \beta_{j-1}$ forms a permutation of $f_{j-1}$. Hence, by adding the character $\beta_j \notin f_{j-1}$ to the sequence we obtain a permutation of $f_j$.

From there we can get the following lemma:

Lemma 12. If $f \in F$ then the query successfully detects that $f \in F$ and returns a positive answer.

Proof. By assumption $f \in F$, which means by fact 2 of lemma 11 that step 2 returns a sequence $s'_f$ which is a permutation of the set $f$. That means that step 3 will return a positive answer. It remains to be proven that step 4 is also successful. Moreover by fact 1 of lemma 11 step 4 will also be successful as step 4 traverses the fingerprint trie top-down where at each step it reaches a valid fingerprint $f_j$.

Lemma 13. Assuming that $f \notin F$, either step 3 or step 4 will successfully detect that $f \notin F$ and the query returns a negative answer.

Proof. The proof is by contradiction. Suppose that step 4 has concluded that $f \in F$. Then steps 3 tells us that we have a sequence of $j$ characters $s'_f = \beta_1, \beta_2, \ldots, \beta_{|f|-1}$ which is a permutation of $f$ and that moreover by successfully traversing the trie in step 4 we deduce that $f \in F$ which contradicts the premise that $f \notin F$.

Thus, we get the following theorem:

Theorem 4. The set of $F$ of fingerprints of a sequence $s = s_1, s_n$ can be represented using a data structure that occupies $|F|((2 \log \sigma + \log_2 e)(1 + o(1)))$ bits. Given a set of characters $f$ the data structure is able to determine whether $f \in F$ (existential queries) in time $O(|f|)$.

We can also use the data structure to answer to report queries. However, in this case, because of the need to store pointers to occurrences, the representation will no longer be succinct (a pointer needs $\Omega(\log n)$ bits to be represented). We note that for each fingerprint, we can just store the list of maximal locations in the sequence using $2 \log n$ bits for each element giving a total of $O(|L| \log n)$ bits. However, a more space efficient approach is to use the suffix tree and for each fingerprint store a list of pointers to
named copies in the suffix tree. This reduces the space to \(O((n + |L|) \log n)\) bits. Moreover, reporting the locations of the \(occ\) named copies from the suffix tree takes optimal \(O(occ)\) time as it consists in traversing a subtree with at most \(occ\) leaves and \(occ - 1\) internal nodes.

**Theorem 5.** Given a sequence \(s = s_1..s_n\) of characters we can in time \(O(n + |L|) \log \sigma\) build a data structure that occupies \(O((n + |L|) \log n)\) bits of space such that given a fingerprint \(f \in F\) the data structure is able to report all the \(occ\) maximal locations in \(s\) corresponding to \(f\) in time \(O(|f| + occ)\).

## 5 Identifying Fingerprints in Less Space

The result of theorem 3 names all fingerprints of \(s\) in time \(\Theta(2\sigma + |L| \log \sigma)\) while using \(O((|L| + |F|) \log \sigma) \log n)\) bits of working space during the building. The value \(|L|\) in the working space can dominate the value \(|F| \log \sigma\) when \(|F| \ll |L|\). When we need to build a data structure for report queries, then the value \(|L|\) is also present in the final size of required space and hence this presence in building space is unavoidable. However, when we only need to answer to existential queries, then the final data structure will use space of \(O(|F| \log \sigma)\) bits only. In this case it would be desirable to reduce the construction time as well. In this section, we show how to compute the set \(F\) in time \(O(|L| \log \sigma)\), but using space of \(O(|F| \log \sigma \log n)\) bits only.

The original naming algorithm of [1] is convenient for our purpose as it does the naming online without the need to carry the list of fingerprint changes (which is essentially equivalent to \(L\)) until the end of the construction. The complexity of the algorithm of [1] is \(O(n \sigma \log n \log \sigma)\). The \(\log n\) factor comes from the complexity of the use of binary search tree which is responsible for the following task: given a pair of names \((\text{sub}_0, \text{sub}_1)\) at level \(i\), find whether there is a unique name \(\text{up}\) at level \(i + 1\) associated with the pair and if not add a new unique name \(\text{up}\), associate it with the pair \((\text{sub}_0, \text{sub}_1)\) and add it to the binary search tree. This complexity of the naming algorithm was improved in [14,15] from \(O(n \sigma \log n \log \sigma)\) to just \(O(|L| \log \sigma)\) by the following way.

1. Notice that the naming has to deal only with \(|L|\) fingerprint changes instead of \(n\sigma\). This reduces the factor \(n\sigma\) to \(|L|\).
2. Deferring the naming process until all the fingerprint changes have been recorded. Then using radix sort, the process time of giving unique names at level \(i + 1\) to pairs of names from level \(i\) is reduced to constant time per pair. This dispenses from the use of the binary search tree and reduces the factor \(\log n\) to just 1.

This is the approach used in theorem 3 and described in section 3.

Our approach to improve [1] is to notice that the binary search tree can be replaced with any hash table implementation which will change the time per operation from worst-case \(O(\log n)\) to randomized expected \(O(1)\). By this change the query time reduces to expected \(O(|L| \log \sigma)\), but contrary to theorem 3 the building space remains as small as in [1], as we do not need to record the fingerprint changes during the building process. More precisely during the naming process we need only to maintain at most \(|F| \log \sigma\) names (each fingerprint might incur at most \(\log \sigma\) names, one name at each level), which have been attributed so far. These names are recorded in a hash table which will use \(O(|F| \log \sigma \log n)\) bits of space.

Thus, we have proven the following theorem:
Theorem 6. The set $F$ of fingerprints of a sequence $s = s_1..s_n$ can be computed in expected time $O(n + |L| \log \sigma)$ time using $O((n + |F| \log \sigma) \log n)$ bits of working space.

6 Randomized Identification Using a Monte Carlo Algorithm

We now briefly sketch our construction algorithm that constructs the set of fingerprints $F$ of the sequence $s$, using only $O(|F| \log n)$ bits ($O(|F|)$ words) of temporary space and running in time $O(|L|)$. While this approach might fail with an extremely small probability (the approach is said to be Monte Carlo or MC for short), it might still be useful in case one wishes to get approximate statistics on fingerprints: counting the total number of distinct fingerprints, or counting the total number of strings having a given fingerprint, etc.

To name the fingerprints we use use hash values of size $\Theta(\log n)$ bits. The hash values are computed using polynomial hash functions as described in section 2.1. Like in the previous section, the naming will be done online: we do not need to store the fingerprint changes during the naming process. Unlike the method described in the previous section, the fingerprint names will not be assigned deterministically, but will instead be assigned using hash values which could collide with extremely small probability. More specifically, in order to identify the existence of a fingerprint we will use the polynomial hash functions as described in section 2.1 on the whole fingerprint. The polynomial hash function will be computed modulo $P$, where $P$ is a prime selected such that $P > n^{c n^2 \sigma^3}$). The chosen value of $P$ will ensure that each fingerprint will be mapped to a distinct value with probability at least $n - c$. This can easily be seen: we have $|F| < n \sigma$ which implies that $|F|^2 < n^2 \sigma^2$. Given that the polynomials are of degree at most $\sigma$, we can deduce that the probability of collision is at most $\frac{|F|^2 \sigma}{n^2 \sigma^2} = n - c$.

We now describe our algorithm in more detail. We assume that a set $S$ of fingerprints can be represented as a list $L = (\alpha_1, \alpha_2, \ldots, \alpha_p)$ of distinct characters such that $S = \{ f_1, f_2, \ldots, f_p \}$ where $f_i = \cup_{1 \leq j \leq i} \{ \alpha_j \}$. We randomly choose a number $r \in [0, P]$ and the random hash function $H_r$ will be such that:

$$H_r(f_i) = \sum_{1 \leq j \leq i} (r^{f_j \Sigma(\alpha_j)})$$

The number $H_r(f_i)$ will be the unique name associated with the fingerprint $f_i$. Now observe that $H_r(f_i) = H_r(f_{i-1}) + r^{f_i \Sigma(\alpha_i)}$. Thus computing the label of $f_i$ can be done online using constant number of arithmetic operations based on $\alpha_i$ and $H_r(f_{i-1})$. In order to maintain the set of already processed fingerprints, we use a dynamic hash table (for example using the MC real time dynamic hashing method described in [7]) that records the names of already processed fingerprints. Each time we generate the name of the fingerprint associated with a given maximal location we probe the dynamic hash table to see if that name already exists and if not add it to the hash table. If we also need to maintain the set of maximal locations along with the set of fingerprints, we just associate a list of maximal locations to each fingerprint and store that list as satellite data associated to the fingerprint name stored in the hash table. When the name of the fingerprint associated to a maximal location already exists in the hash table, this maximal location is added to the list of maximal locations associated with the fingerprint name in the hash table. If the fingerprint name did not already exist
in the hash table, we add the name to hash table and associate a list of maximal locations which contains only the maximal location corresponding to the newly added fingerprint.

In conclusion, we have proven the following theorem:

**Theorem 7.** The set \( F \) of fingerprints of a sequence \( s = s_1..s_n \) can be probabilistically computed in time \( O(n + |L|) \) using \( O((n + |F|) \log n) \) bits of working space. Moreover the set of maximal locations \( L \) can be probabilistically determined in time \( O(n + |L|) \) using \( O((n + |L|) \log n) \) bits of working space. The error rate probability can be made to \( O(n^{-c}) \) for any constant \( c \).

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. C.-Y. Chan, H.-I Yu, W.-K. Hon, and B.-F. Wang. A faster query algorithm for the text fingerprinting problem. In *ESA*, pages 123–135, 2007.
4. Chi-Yuan Chan, Hung-I Yu, Wing-Kai Hon, and Biing-Feng Wang. Faster query algorithms for the text fingerprinting problem. *Inf. Comput.*, 209(7):1057–1069, 2011.
5. D. Xavier Charles and K. Chellapilla. Bloomier filters: A second look. In *ESA*, pages 259–270, 2008.
6. G. Didier, T. Schmidt, J. Stoye, and D. Tsur. Character sets of strings. 2004. Submitted.
7. M. Dietzfelbinger and F. M. auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *ICALP*, pages 6–19, 1990.
8. M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *FOCS*, pages 524–531, 1988.
9. M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In *ICALP (1)*, pages 385–396, 2008.
10. M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science*, pages 137–143, Miami Beach, FL, 1997.
11. 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.
12. 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.
13. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
14. 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.
15. R. Kolpakov and M. Raffinot. New algorithms for text fingerprinting. *J. Discrete Algorithms*, 6(2):243–255, 2008.

16. E. Porat. An optimal bloom filter replacement based on matrix solving. In *CSR*, pages 263–273, 2009.

17. R. Raman, V. Raman, and S. Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

18. 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.