

New Algorithms for Text Fingerprinting

- extended abstract -

Roman Kolpakov¹ Mathieu Raffinot²

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 $s = s_1 \dots s_n$ 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 and being able to efficiently answer several questions on this set. A given fingerprint $f \in \mathcal{F}$ is represented by a binary array, F , of size $|\Sigma|$ named a fingerprint table. A fingerprint, $f \in \mathcal{F}$, admits a number of maximal locations (i, j) in S , that is the alphabet of $s_i \dots s_j$ is f and s_{i-1}, s_{j+1} , if defined, are not in f . The total number of maximal locations is $\mathcal{L} \leq n|\Sigma| + 1$. We present new algorithms and a new data structure for the three problems: (1) compute \mathcal{F} ; (2) given F , answer if F represents a fingerprint in \mathcal{F} ; (3) given F , find all maximal locations of F in s . These problems are respectively solved in $O((\mathcal{L} + n) \log |\Sigma|)$, $\Theta(|\Sigma|)$, and $\Theta(|\Sigma| + K)$ time - where K is the number of maximal locations of F .

1 Introduction

We consider a finite ordered alphabet, Σ , and $s = s_1 \dots s_n$ 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 $s_i \dots s_j$. A fingerprint is represented below by a binary table of F of size $|\Sigma|$. If s contains the character α , $F[\alpha] \leftarrow 1$, otherwise $F[\alpha] \leftarrow 0$.

Definition 1. Let \mathcal{C} be a set of letters of Σ . A maximal location of \mathcal{C} in $s = s_1 \dots s_n$ is an interval $[i, j]$, $1 \leq i \leq j \leq n$, such that

- (1) $C_s(i, j) = \mathcal{C}$; (2) if $i > 1, s_{i-1} \notin C_s(i, j)$; (3) if $j < n, s_{j+1} \notin C_s(i, j)$

We denote by \mathcal{F} the number of distinct fingerprints and by \mathcal{L} the number of maximal locations of all fingerprints of \mathcal{F} . In this paper, given a sequence s , we are interested in three main algorithmic problems: 1. Compute the set \mathcal{F} of all fingerprints in s ; 2. Given a fingerprint table F , find if F represents a fingerprint in \mathcal{F} or not; 3. Given a fingerprint table 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 [1]. The input alphabet Σ is considered to be the alphabet of the input sequence, thus $|\Sigma| \leq n$.

The best actual algorithms solve Problem 1 in $\Theta(\min\{n|\Sigma| \log |\Sigma|, n^2\})$ time. The bound $\Theta(n|\Sigma| \log |\Sigma|)$ is that of the algorithm of Tsur in [4] that we present in depth. The $\Theta(n^2)$ bound is obtained using the algorithm of Didier also presented in [4], although this algorithm was first presented 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). Problem 2 is solved in $O(|\Sigma| \log n)$ time and Problem 3 in $O(|\Sigma| \log n + n)$ time in [1, 4]. Surprisingly enough, and this a strong motivational factor for this paper, these complexities are independent of the sizes of \mathcal{F} and \mathcal{L} , although many sequence families have few fingerprints or few maximal locations.

In this paper we present new algorithms and a new tree structure for solving these three problems. Problem 1 is solved in $O((|\mathcal{L}| + n) \log |\Sigma|)$ time. As $|\mathcal{L}| \leq n|\Sigma| + 1$, our algorithm is, at worst, as efficient as that of Tsur, but much more efficient on many sequence families. It can however be slower than that of Didier when $|\Sigma| = \Omega(n/\log n)$. Although, even in this case, the real complexity of our algorithm depends of the number of maximal locations that can be much less than $n|\Sigma| + 1$. Our algorithm also has the advantage of being simple to implement in its real worst case complexity. Problem 2 is optimally solved in $\Theta(|\Sigma|)$ using a new tree structure, improving the fastest algorithm by $\log n$. Finally, Problem 3 can be solved either in $\Theta(|\Sigma| + K)$ time - where K is the number of maximal locations of F - using $\Theta(|\mathcal{F}| \log |\Sigma| + |\mathcal{L}|)$ space; or $\Theta(|\Sigma| + n)$ time using $\Theta(|\mathcal{F}| \log |\Sigma|)$ space. To maintain continuity with the previous approaches, our algorithms improve a naming technique introduced in [1] and [4]. The paper is organized as follows. The original naming technique is presented first in Section 2. In Section 3 we present our new naming algorithm. In the next Section 4 we detail our tree data structure and the algorithmic improvements it permits. We do not provide any proof in this extended abstract. The interested reader should refer to [5] for details.

2 Fingerprints and Naming Technique

In this section we recall the naming technique introduced in [1] and then improved by Tsur in [4]. 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 .

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 0. 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. Fig 1 shows a simple example with $|\Sigma| = 8$.

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

[7]			
[5]		[6]	
[2]	[2]	[3]	[4]
[1][0]	[1][0]	[1][1]	[0][0]

Fig. 1. Naming example.

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 Fig. 1, the name [2] is associated to ([1], [0]) the first time this pair is encountered. The second time, this name is directly retrieved.

Naming all Fingerprints. A change in the lowest level of this array, that is changing a [1] to a [0] or a [0] to a [1] causes, at most, $\log |\Sigma| + 1$ changes in the names on the path from the modified cell to the root. This property is used in the original algorithm of [1]. Their idea is to enumerate all fingerprints containing a fixed number k of different characters by shifting two indices $1 \leq i \leq j \leq |s|$ on the sequence. The algorithm first identifies a pair $(i_0 = 1, j_0)$ such that $s_{i_0..s_{j_0}}$ contains exactly k distinct characters. This fingerprint is named using the previous technique. Then the two indices are shifted to (i_1, j_1) that points the beginning and the end of the next substring containing exactly k different characters and with a different fingerprint than the previous one. The key point of the algorithm of [1] is that this new fingerprint only differs from the previous one in two positions in the lowest array. Updating the array of names thus requires, at most, $2 \log |\Sigma| + 2$ changes. *Complexity.* Each change in the name array requires checking whether a pair of names has already received a name or not. At each level there are, at most, n new names, and thus searching for the pair can be done in $\log n$ time using a balanced tree. For each value of k , initializing the array of names requires $O(|\Sigma| \log |\Sigma|)$ time. Then each new pair (at most n when reading the sequence) requires, at most, 2 global changes in the whole array, each requiring $O(\log |\Sigma| \log n)$. Thus, for each value of k , building the names requires $O(n \log |\Sigma| \log n)$ time and as $k = 1..|\Sigma|$, the whole algorithm takes $O(n|\Sigma| \log |\Sigma| \log n)$ time.

In [4] Tsur presented a faster algorithm to build all names. The algorithm still performs $|\Sigma|$ iterations in a similar way to the previous one, but fills the names level by level. The list of changes in each level over the whole sequence is recorded in an ordered list. This list is sorted using an $O(n)$ sort algorithm (for instance radix sort) and new names are given according to this sort. These new names are placed in the original list (the order of this list is important) that is used to build the initial list of the next level. A pseudo-code of the main part of the algorithm (slightly modified) can be found in [5].

We number the level from 1, the lowest, to $\log |\Sigma| + 1$. For each k , $1 \leq k \leq |\Sigma|$, the initialization of the ordered list L_1 at level 1 is performed by reading the sequence. This list records the changes at level 1. In order to build it, we move two pointers on the sequence in exactly the same way as in the previous algorithm. When the first pair $(i_0 = 1, j_0)$ is encountered, the values in the fingerprint table

A (the array of level 1) is registered in L_1 under the form of pairs $\{A[i], i\}$ for $i = 0..|\Sigma| - 1$. The two pointers are then moved and for each new pair of pointers there are only two modifications in the array A . For each such modification, if A changed in position j , $0 \leq j \leq |\Sigma| - 1$, this change is recorded by adding $\{A[j], j\}$ to the end of L_1 . At the end of this process, the ordered list L_1 records all changes to be performed at level 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 in the following way. 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 $\{(FT[2\lfloor j/2 \rfloor], FT[2\lfloor j/2 \rfloor + 1]), j/2\}$ 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 $(FT[2\lfloor j/2 \rfloor], FT[2\lfloor j/2 \rfloor + 1])$.

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 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 fingerprints containing exactly k distinct characters in the original sequence. *Complexity.* The initialization of L_1 is $\Theta(n)$ time. Then a linear sort of at most $\Theta(n)$ elements is performed for every level. As there are $\log |\Sigma| + 1$ levels, the process is $\Theta(n \log |\Sigma|)$ time. As $k = 1..|\Sigma|$, the whole complexity is $\Theta(n|\Sigma| \log |\Sigma|)$ time. This saves $\log n$ over the previous algorithm.

3 New Algorithm to Compute All Fingerprints

The faster algorithm of Section 2 is $\Theta(n|\Sigma| \log |\Sigma|)$ time. This complexity is independent of the number of maximal locations, \mathcal{L} , although this is one of the main parameters of the fingerprinting problem. The naming algorithm we present depends on \mathcal{L} and its complexity is $O((\mathcal{L} + n) \log |\Sigma|)$ time. As \mathcal{L} is, at most, $n|\Sigma| + 1$, but is much less on many sequence families, our algorithm is faster than or as efficient as previous ones.

Moreover, the fingerprint tree we present in the next section permits efficient searching for a given fingerprint to appear in the sequence. However, it requires all fingerprint names to be globally built on the same name subsets, which is not the case of the names generated by the two algorithms of Section 2. It also requires these names to be sorted in the lexicographical order of their fingerprint tables. We present in this section a new naming algorithm that fulfills these requirements.

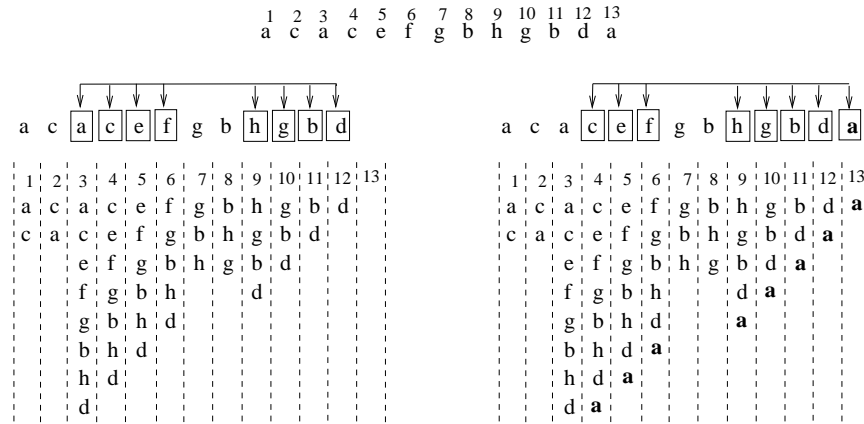


Fig. 2. A (schematic) step of algorithm FINGERPRINT_CHANGES that is the first phase for computing all fingerprints. We add the character **a** in table TN_1 .

The main idea of the second algorithm of Section 2 is to record the changes in the fingerprint tables before computing names. We reuse this approach but we (a) process the whole sequence once before computing all names and (b) record only changes corresponding to maximal locations. Point (a) is achieved by keeping for each position i in the sequence a virtual list of all fingerprint tables of substrings $s_i \dots s_j$, $i \leq j \leq n$, beginning in i . This list is virtual in the sense that instead of keeping a list of fingerprint tables we only record all changes in the fingerprint table in i . At the beginning all fingerprint tables are considered *empty*, that is full of $|\Sigma|$ zeros. Point (b) consists in considering only positions that correspond to maximal locations.

Our algorithm runs in two phases. The first phase identifies on the sequence the fingerprints that must be encoded. The second phase builds names for all these fingerprints. The sequence is thus read once.

We assume that without loss of generality below 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 set, \mathcal{L} , up to small changes in the bounds. These changes can, however, be simply retrieved in $\Theta(1)$ per maximal location. The reducing algorithm is $\Theta(n)$. This technical trick really simplifies the algorithms we present by removing many straightforward technical cases.

First Phase. Let $s = s_1 \dots s_n$ be a sequence of characters over Σ . For each character $\alpha \in \Sigma$ we define R_s^α as the indice in s of the rightmost occurrence of α in s , and we fix $R_s^\alpha = 0$ if there is no such occurrence. We define the last occurrence list L_s as being the sorted list (in increasing order) of all indices in s of character last occurrences. We add an arbitrary 0 (if not already there)

```

FINGERPRINT_CHANGES( $s = s_1 \dots s_n$ )
1. Let  $TN_1[1..n]$  a table of  $n$  lists
2.    $TN_1[i]$  are all initialized to an empty list
3.  $L \leftarrow (0)$ 
4. For  $i=1..n$  Do
5.    $\alpha \leftarrow s_i$ 
6.   add  $\{[1], f_\Sigma(\alpha)\}$  on top of  $TN_1[i]$ 
7.    $j \leftarrow$  top element of  $L$ 
8.   While  $j > 0$  AND  $s_j \neq \alpha$  Do
9.     add  $\{[1], f_\Sigma(\alpha)\}$  on top of  $TN_1[j]$ 
10.     $j \leftarrow$  previous element in  $L$ 
11.   End of while
12.   If  $j > 0$  Then /* there is a indice of an  $\alpha$  in  $L$  */
13.     remove  $j$  from  $L$ 
14.   End of if
15.   add  $i$  on top of  $L$ 
16. End of for

```

Fig. 3. Computing all fingerprint changes as a first phase of the new naming procedure.

before all indices; Notice that the last indice of L_s must be n . Thus, formally, $L_s = (0, R_s^{\alpha_1}, R_s^{\alpha_2}, \dots, R_s^{\alpha_k} = n)$.

Suppose now that $C_s(i, j)$ is known for all pairs $1 \leq i \leq j \leq |s|$. When concatenating a letter α to s , we aim to compute all $C_{s\alpha}(i, j)$ for $1 \leq i \leq j \leq |s\alpha|$.

Lemma 1. *Let $s = s_1..s_n$, $s_i, \alpha \in \Sigma$ and $L_s = (0, R_s^{\alpha_1}, R_s^{\alpha_2}, \dots, R_s^{\alpha_k} = n)$. The following properties hold:*

1. For all pairs $1 \leq i \leq j \leq n$, $C_{s\alpha}(i, j) = C_s(i, j)$.
2. Let z such that $L_s[l - 1] < z \leq L_s[l]$, $0 < l \leq k$. Then $C_{s\alpha}(z, |s\alpha|) = C_s(L_s[l], |s|) \cup \{\alpha\}$.
3. If $R_s^\alpha > 0$, let $0 < z \leq R_s^\alpha$. Then $C_{s\alpha}(z, |s\alpha|) = C_s(z, |s|)$.

The first phase of the algorithm reads the sequence s one character after the other. Assume that we have already read and processed the characters up to position j . For each position $k = 1..j$ a list encodes the series of fingerprint changes to code for $C(k, j)$. We read the character $\alpha = s_{j+1}$. According to lemma 1 the algorithm goes down the indice list of last character occurrences until either (a) the same character α is encountered (points 2 and 3 of lemma 1), or (b) the beginning of the list is reached (point 2 of lemma 1). For each indice i touched in this list, we add the character α to the list representing $C(i, j)$ in order to obtain $C(i, j + 1)$. The list of last occurrences is then updated by removing the indice of the previous occurrence of α and adding $j + 1$ as the new indice of α . The first phase of the algorithm is called FINGERPRINT_CHANGES and its pseudo-code is given in Figure 3. A step of the algorithm is shown in Figure 2.

Second Phase. The second phase is based on the second algorithm of section 2. It remains to name all fingerprints appearing as a prefix of each list in TN_1 .

This is done with the algorithm NAME_ALL_LISTS for which the pseudo-code is given in Figure 4. In a similar way to the second algorithm of Section 2, $\log |\Sigma|$ iterations are performed for each fingerprint array level.

In each iteration, each list in TN_k is read to build a corresponding list of cell changes in level $k + 1$ (lines 5-19). A new list table TN'_k is thus built and records all these new lists. The pair of names in TN'_k are sorted altogether in lexicographic order through a radix sort (line 23). A new name is then given to each different pair (line 24). A new list table TN_{k+1} is then built by copying TN'_k , but replacing each name pair with its new name (line 25). This list is the input list of the next iteration of the general loop (lines 2-27).

Special care is required for the initialization process of the temporary table FT of size $|\Sigma|/2^{k-1}$. The table is initialized once (line 3) and reinitialized after coding each list of cell changes in TN'_k . However, for complexity issues, this re-initialization is performed in an amount of time proportional to the size of this list by simply erasing the changes that have been made (lines 14-19).

```

NAME_ALL_LISTS( $TN_1[1..n]$  initial table of fingerprint changes)
1.  $ninit_1 \leftarrow [0]$ 
2. For  $k = 1.. \log |\Sigma|$  Do
3.    $FT_k \leftarrow$  name table of size  $|\Sigma|/2^{k-1}$  all initialized to  $ninit_k$ 
4.   Let  $TN'_k[1..n]$  be a table of  $n$  lists.
5.   For  $i = 1..n$  Do
6.     initialize  $TN'_k[i]$  to the empty list
7.      $L \leftarrow$  first element of  $TN_k[i]$ 
8.     While  $L$  exists Do
9.        $\{[a], j\} \leftarrow L$ 
10.       $FT_k[j] \leftarrow [a]$ 
11.      add  $\{(FT_k[2\lfloor j/2 \rfloor], FT_k[2\lfloor j/2 \rfloor + 1]), j/2\}$  to end of  $TN'_k[i]$ 
12.       $L \leftarrow$  next element in  $TN_k[i]$ 
13.     End of while
14.      $L \leftarrow$  first element of  $TN_k[i]$ 
15.     While  $L$  exists Do
16.        $\{[a], j\} \leftarrow L$ 
17.        $FT_k[j] \leftarrow ninit_k$ 
18.        $L \leftarrow$  next element in  $TN_k[i]$ 
19.     End of while
20.   End of for
21.    $Sl \leftarrow$  list of all cell pairs in  $TN'_k$ 
22.   add the pair  $(ninit_k, ninit_k)$  to  $Sl$ 
23.   sort  $Sl$  in lexicographical order
24.   give new names for each different pair in  $Sl$ 
25.   build  $TN_{k+1}$  by copying  $TN'_k$  but replacing each pair by its new name
26.    $ninit_{k+1} \leftarrow$  name of the pair  $(ninit_k, ninit_k)$ 
27. End of for

```

Fig. 4. Naming all fingerprints in all lists of TN_1 .

The result of the first iteration of the NAME_ALL_LISTS algorithm on the TN_1 table is given in Figure 5.

1	2	3	4	5	6	7	8	->
{(1,1), (0), 0}	{(1,1), (0), 1}	{(1,1), (0), 0}	{(1,1), (0), 1}	{(1,1), (0), 2}	{(0), (1), 2}	{(1,1), (0), 3}	{(0), (1), 0}	->
{(1,1), (0), 1}	{(1,1), (0), 0}	{(1,1), (0), 1}	{(1,1), (0), 2}	{(1,1), (1), 2}	{(1,1), (0), 3}	{(0), (1), 0}	{(0), (1), 3}	->
		{(1,1), (0), 2}	{(1,1), (1), 2}	{(1,1), (0), 3}	{(0), (1), 0}	{(1,1), (1), 3}	{(1,1), (1), 3}	->
		{(1,1), (1), 2}	{(1,1), (0), 3}	{(0), (1), 0}	{(1,1), (1), 3}			->
		{(1,1), (0), 3}	{(0), (1), 0}	{(1,1), (1), 3}	{(0), (1), 1}			->
		{(1,1), (1), 0}	{(1,1), (1), 3}	{(0), (1), 1}	{(1,1), (1), 0}			->
		{(1,1), (1), 3}	{(1,1), (1), 1}	{(1,1), (1), 0}				->
		{(1,1), (1), 1}	{(1,1), (1), 0}					->

Fig. 5. First columns of the table TN_1 of lists of cell changes.

The last sort of the table $TN_{\log|\Sigma|}$ records the fingerprint names. The NAME_ALL_LISTS algorithm obviously insures that these names are sorted in the lexical order of their fingerprint tables.

Lemma 2. *Let $s = s_1..s_n$, $s_i, \alpha \in \Sigma$ and $L_s = (0, R_s^{\alpha_1}, R_s^{\alpha_2}, \dots, R_s^{\alpha_k} = n)$. Let z be the indice of R_s^α in L_s . Then for all indices l in L_s such that $l > z$, interval $[l + 1 \dots |s|]$ is a maximal location. If $z = 0$, $[1 \dots |s|]$ is also a maximal location.*

Theorem 1. *Our algorithm names all distinct fingerprints of s in $O((\mathcal{L} + n) \log |\Sigma|)$ time.*

It remains to prove that \mathcal{L} is bounded by $n(|\Sigma| + 1)$.

Proposition 1. *The number \mathcal{L} of maximal locations is bounded by $n(|\Sigma| + 1)$.*

Corollary 1. *Our naming algorithm is $\Theta(n|\Sigma| \log |\Sigma|)$ worst case time.*

Computing all Maximal Locations. In order to efficiently solve problem 3, we associate each fingerprint name with its maximal locations. Our approach is to compute the maximal locations during the first phase of the previous algorithm and maintain them through the second phase.

Proposition 2. *Let $s = s_1..s_n$ and $[i, j]$, $0 < i \leq j \leq n$, be a maximal location in s of a fingerprint f ; let $w = s_1 \dots s_j$ and $L_w = (0, R_w^{\alpha_1}, R_w^{\alpha_2}, \dots, R_w^{\alpha_k} = j)$. There exists a unique p in L_w such that $f = C(p + 1, j) = \cup_{R_w^{\alpha_h} > p} \{\alpha_h\}$.*

We modify the algorithm FINGERPRINT_CHANGES to associate each new maximal location with its alphabet that is to be coded in the second phase of the naming. Each time a new j (line 7) except the first one is encountered at iteration i (lines 4-16), the maximal location (insured by lemma 2) $[j + 1, i - 1]$ is associated with its alphabet, which is, according to proposition 2, the last but one element in the column corresponding to the indice after j in $L_{s_1..s_{i-1}}$.

Notice that according to proposition 2 a final iteration of the loop i (lines 4-16) is required to compute the maximal locations appearing at the end of the sequence. In this last iteration, the maximal location $[j + 1, n]$ is associated with the *last* element in the column corresponding to the indice after j in $L_{s_1 \dots s_n}$. This technical add-on could also be fixed by adding a last virtual character to the input sequence. After the first phase, names are built using the NAME_ALL_LISTS algorithm, slightly modified for keeping track in all cell change lists of the associated maximal locations. A last phase is necessary to group the set of maximal locations of each fingerprint name.

4 Fingerprint Tree

Once the fingerprints have been named by one of the two algorithms of Section 2, searching for a given fingerprint to appear in the sequence can be carried out in $O(|\Sigma| \log n)$ time by a similar process to that in the first algorithm, that is, filling the level from bottom to top and checking for each cell if a name has already been given to a pair of cells [1].

This searching could be made $O(|\Sigma|)$ expected time using perfect hashing on name pairs [2] of each level. The hash tables would require $O(\mathcal{F} \log |\Sigma|)$ expected memory space and could be built in $O(\mathcal{F} \log |\Sigma|)$ expected time. However, in the worst case, these hash tables could require $O(\mathcal{F}^2 \log |\Sigma|)$ memory space and be built in $O(\mathcal{F}^2 \log |\Sigma|)$ time.

We propose another approach for solving this problem in $O(|\Sigma|)$ worst case time, requiring in the worst case $O(\mathcal{F})$ additional memory space and $O(\mathcal{F} \log |\Sigma|)$ additional preprocessing time. For these purposes we present a new tree structure of size $O(\mathcal{F})$ that permits searching for a given fingerprint in $O(|\Sigma|)$ time. This tree can be built in $O(\mathcal{F} \log |\Sigma|)$ time. The searching phase requires the names to be given from the same set of names for all the $k = 1..|\Sigma|$ iterations. Therefore, the names in this tree cannot be generated using one of the first algorithms of Section 2. The construction itself requires the fingerprint names to be sorted in the lexicographical order of their corresponding fingerprint tables, which is a property of the naming algorithm we presented. The fingerprint tree is a binary tree in which each fingerprint name is a leaf. Edges are labeled with a triplet $\{DT, l, r\}$ where DT is either (i) a single name [1] or [0] if one of l or r is equal to 1 and the other to 0; or (ii) a pair of names (n_l, n_r) and $1 \leq l \leq |\Sigma|$ and $1 \leq r \leq |\Sigma|$ two lengths. The pair of names (n_l, n_r) is related to l and r by being the lowest pair of consecutive names at the same level that cover the segment from the beginning of l to the end of r in the name array of one of the leaves in the subtree. Figure 6 shows an edge label.

Building the Tree. We denote by *name tree* the tree formed by recursively developing all name pairs deriving from a given name.

Definition 2. Let F_1 and F_2 be two fingerprint tables. The Longest Common Prefix (lcp) of F_1 and F_2 is the longest equal sub-table beginning F_1 and F_2 .

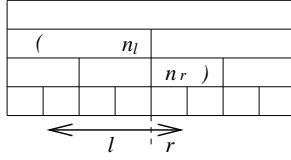


Fig. 6. Label of a transition. The pair (n_l, n_r) is the lowest consecutive pair of names covering the segments l and r .

By extension, we denote the lcp of two fingerprint names the lcp of the fingerprint tables they encode. Let $n_1 \dots n_{\mathcal{F}}$ be the fingerprint names whose fingerprint tables are sorted in lexicographic order (requirement). The construction of the tree is done in $O(\mathcal{F} \log |\Sigma|)$ time in three phases. (i) Compute for every two consecutive names n_i, n_{i+1} their lcp in $LCP[i, i + 1]$. (ii) Build a skeleton of the tree containing all necessary nodes but in which each edge is labeled by an interval $[k..l]$. This interval denotes that the label of the edge must code for the interval $[k..l]$ in any fingerprint table of the leaves in the subtree starting at this edge. (iii) Build the label of each edge. We now detail these three steps.

Computing the Lcp We begin with the two given names and a current lcp fixed to $|\Sigma|$. The lcp of two fingerprint names can be computed by simultaneously going down each name tree. At each step of the algorithm we compare two names. If these names are equal, the resulting lcp is the current lcp. Otherwise, each name corresponds to two other names (unless they are $[0]$ or $[1]$). If their first name (the left part) is equal, then the lcp is the size of this left part (current lcp / 2) plus the lcp of the second (right part). Otherwise the lcp is the current lcp plus the lcp of the first part. A pseudo-code of a recursive version of the algorithm can be found in [5].

Lemma 3. *Let n_1 and n_2 be two fingerprint names. The lcp of n_1 and n_2 can be computed in $O(\log |\Sigma|)$ time.*

The table LCP is built by $\mathcal{F} - 1$ iterations of algorithm lcprec. The whole complexity of this first phase is thus $O(\mathcal{F} \log |\Sigma|)$ time.

Building a Skeleton Tree We build a skeleton of the fingerprint tree by adding a branch and a leaf for each name n_i . This construction is illustrated in Fig. 7.

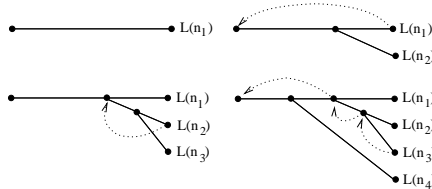


Fig. 7. Building the skeleton of the fingerprint tree. Backward dashed edges illustrate searching for the position of the new branch.

The initial tree is a single root. This branch is plugged at depth $LCP[i - 1, i]$ to the branch previously built for the name n_{i-1} . This is done by going up in

```

BUILD_SKELETON_TREE( $n_1..n_{|\mathcal{F}|}$ ,  $LCP$ )
1.  $depth(root) \leftarrow 0$ 
2. branch  $L(n_1)$  on  $root$ 
3.  $current \leftarrow root$ 
4.  $prev \leftarrow L(n_1)$ 
5. For  $i = 2..|\mathcal{F}|$  Do
6.   While  $depth(current) > LCP[i - 1, i]$  Do
7.      $prev \leftarrow current$ 
8.      $current \leftarrow father(current)$ 
9.   End of while
10.  If  $depth(current) = LCP[i - 1, i]$  Then
11.    branch  $L(n_i)$  on  $current$ 
12.  Else
13.    cut the edge  $(current, prev)$  with  $newnode$ 
14.     $depth(newnode) \leftarrow LCP[i - 1, i]$ 
15.    branch  $L(n_i)$  on  $newnode$ 
16.     $current \leftarrow newnode$ 
17.  End of if
18.   $prev \leftarrow L(n_i)$ 
19. End of for

```

Fig. 8. Computing the skeleton tree.

the tree from the last leaf created for n_{i-1} (denoted $L(n_{i-1})$) to the root. On this path, we isolate the first node q with depth d less than or equal to $LCP[i - 1, i]$. If d is exactly $LCP[i - 1, i]$, then we just plug a new branch to q . Otherwise we create a new node p with depth $LCP[i - 1, i]$ that becomes a new child of q . This node p now has two children, one corresponding to the previous subtree of q . The other is the new branch that is terminated with the new leaf $L(n_i)$ from where the next step begins. A pseudo-code of this construction is given in Fig. 8. In this code, the depth of any $L(n_i)$ is fixed to $|\Sigma|$.

The complexity of building the skeleton tree is $O(\mathcal{F})$ time since each node is at most visited twice, once when created and at most once when going up the nodes to search for the position of the new branch.

Building Edge Labels Once the skeleton tree has been built, each node has a depth associated. Each edge from node q to node p corresponds to the segment $[depth(q) + 1..depth(p)]$ in each fingerprint array of each leaf in the subtree. This segment permits efficient coding of each edge in $O(\log |\Sigma|)$ time. Coding all edges of the fingerprint tree thus requires $O(\mathcal{F} \log |\Sigma|)$ time.

Searching for a Fingerprint in the Tree. The coding of each edge of the fingerprint tree has an important property for the time complexity of the search.

Lemma 4. *Let $\{(n_l, n_r), l, r\}$ be the label of an edge in a fingerprint tree. The level of the pair of consecutive names (n_l, n_r) is $O(\log(l + r))$.*

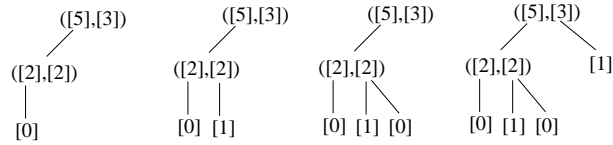


Fig. 9. Decoding the edge label $\{(5, 3), 3, 1\}$.

This property permits the decoding of an edge of the fingerprint tree in a time proportional to the length of this edge. The idea is to traverse the tree formed by the pair of names in a prefix order. Figure 9 illustrates this traversal. As the height of this tree is logarithmic in the length of the segment, the total complexity of decoding an edge $\{(n_l, n_r), l, r\}$ is $O(l + r)$ time. Therefore, searching in the fingerprint tree for a fingerprint given in the form of a fingerprint table of length $|\Sigma|$ is $O(|\Sigma|)$ time. The algorithmic results of this section can be stated in the following theorem.

Theorem 2. *Building a fingerprint tree takes $\Theta(|\mathcal{F}| \log |\Sigma|)$ time and space. Searching for a given fingerprint, F , in it takes $\Theta(|\Sigma|)$ time.*

Considering the maximal locations of a given fingerprint F , two main options exist. The first is to search for the maximal locations once F is known to appear in the sequence. This can be performed by reading the sequence in $\Theta(n)$ time [1]. No extra memory requirement is necessary. The second is to attach to each leaf in the tree the set of its maximal locations computed using the modified naming algorithm of Section 3. This allows searching for all maximal locations of F in $\Theta(|\Sigma| + K)$ time, where K is the number of maximal locations of F . An extra $\Theta(\mathcal{L})$ memory is, however, necessary to record all the maximal locations.

Acknowledgment We thank Guillaume Blin for a complete proofreading of a draft version of this article. We also thank two anonymous referees for pointing us the perfect hashing approach.

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. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms, 2nd Edition*. MIT Press, Cambridge, MA, USA, 2001.
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. R. Kolpakov and M. Raffinot. New Algorithms for Text Fingerprinting. *unpublished*, 2006. Submitted. <http://www-igm.univ-mlv.fr/~raffinot/ftp/fingerprint.pdf>.