International Journal of Foundations of Computer Science
Vol. 27, No. 5 (2016) 529–543
© World Scientific Publishing Company
DOI: 10.1142/S0129054116500143

Algorithms for Longest Common Abelian Factors

Ali Alatabbi* and Costas S. Iliopoulos†
King's College London, London, UK
**ali.alatabbi@kcl.ac.uk*
†costas.iliopoulos@kcl.ac.uk

Alessio Langiu
IAMC-CNR - National Research Council, Unit of Capo Granitola, TP, Italy
alessio.langiu@cnr.it

M. Sohel Rahman‡
AℓEDA Group, Department of CSE, BUET, Dhaka-1000, Bangladesh
msrahman@cse.buet.ac.bd

Received 29 April 2015
Accepted 23 July 2015
Communicated by Alberto Apostolico

In this paper[a] we consider the problem of computing the longest common abelian factor (LCAF) between two given strings. We present a simple $\mathcal{O}(\sigma\ n^2)$ time algorithm, where n is the length of the strings and σ is the alphabet size, and a sub-quadratic running time solution for the binary string case, both having linear space requirement. Furthermore, we present a modified algorithm applying some interesting tricks and experimentally show that the resulting algorithm runs faster.

Keywords: Longest common factor; linear; algorithm; abelian properties; abelian match; Parikh vector.

1. Introduction

Abelian properties concerning words have been investigated since the very beginning of the study of Formal Languages and Combinatorics on Words. Abelian powers were first considered in 1961 by Erdös [6] as a natural generalization of usual powers. In 1966, Parikh [10] defined a vector having length equal to the alphabet cardinality, which reports the number of occurrences of each alphabet symbol inside a given string. Later on, the scientific community started referring to such a vector as the *Parikh vector*. Clearly, two strings having the same Parikh vector are permutations of one another and there is an *abelian match* between them.

‡Currently on a sabbatical leave from BUET.
[a]Preliminary version appeared in [1].

Abelian properties of strings have recently grown tremendous interest among the Stringology researchers and have become an involving topic of discussion in the recent issues of the StringMasters meetings. Despite the fact that there are not so many real life applications where comparing commutative sequence of objects is relevant, abelian combinatorics has a potential role in filtering the data in order to find potential occurrences of some approximate matches. For instance, when one is looking for typing errors in a natural language, it can be useful to select the abelian matches first and then look for swap of adjacent or even near appearing letters. The swap errors and the inversion errors are also very common in the evolutionary process of the genetic code of a living organism and hence is often interesting from Bioinformatics perspective. Similar applications can also be found in the context of network communications.

In this paper, we focus on the problem of finding the Longest Common Abelian Factor of two given strings. The problem is combinatorially interesting and analogous to the Longest Common Substring (LCStr) problem for the usual strings. The LCStr problem is a Historical problem and Dan Gusfield reported the following in his book [7] regarding the belief of Don Knuth about the complexity of the problem:

> ...in 1970 Don Knuth conjectured a linear time algorithm for this problem would be impossible.

However, contrary to the above conjecture, decades later, a linear time solution for the LCStr problem was in fact obtained by using the linear construction of the suffix tree. For Stringology researchers this alone could be the motivation for considering LCAF from both algorithmic and combinatorics point of view. However, despite a number of works on abelian matching, to the best of our knowledge, this problem has never been considered until very recently when it was posed in the latest issue of the StringMasters, i.e., StringMasters 2013. To this end, this research work can be seen as a first attempt to solve this problem with the hope of many more to follow.

In this paper, we first present a simple solution to the problem running in $\mathcal{O}(\sigma\, n^2)$ time, where σ is the alphabet size (Sec. 3). Then we present a sub-quadratic algorithm for the binary string case (Sec. 4). Both the algorithms have linear space requirement. Furthermore, we present a modified algorithm applying some interesting tricks (Sec. 5) and experimentally show that the resulting algorithm runs in $\mathcal{O}(n \log n)$ time (Sec. 6).

2. Preliminaries

An alphabet Σ of size $\sigma > 0$ is a finite set whose elements are called letters. A string on an alphabet Σ is a finite, possibly empty, sequence of elements of Σ. The zero-letter sequence is called the empty string, and is denoted by ε. The length of a string S is defined as the length of the sequence associated with the string S, and is denoted by $|S|$. We denote by $S[i]$ the i-th letter of S, for all $1 \le i \le |S|$ and

$S = S[1\,..\,|S|]$. A string w is a factor of a string S if there exist two strings u and v, possibly empty, such that $S = uwv$. A factor w of a string S is proper if $w \neq S$. If $u = \varepsilon$ ($v = \varepsilon$), then w is a prefix (suffix) of S.

Given a string S over the alphabet $\Sigma = \{a_1, \ldots a_\sigma\}$, we denote by $|S|_{a_j}$ the number of a_j's in S, for $1 \leq j \leq \sigma$. We define the Parikh vector of S as $\mathcal{P}_S = (|S|_{a_1}, \ldots |S|_{a_\sigma})$.

In the binary case, we denote $\Sigma = \{0, 1\}$, the number of 0's in S by $|S|_0$, the number of 1's in S by $|S|_1$ and the Parikh vector of S as $\mathcal{P}_S = (|S|_0, |S|_1)$. We now focus on binary strings. The general alphabet case will be considered later.

For a given binary string S of length n, we define an $n \times n$ matrix M_S as follows. Each row of M_S is dedicated to a particular length of factors of S. So, Row ℓ of M_S is dedicated to ℓ-length factors of S. Each column of M_S is dedicated to a particular starting position of factors of S. So, Column i of M_S is dedicated to the position i of S. Hence, $M_S[\ell][i]$ is dedicated to the ℓ-length factor that starts at position i of S and it reports the number of 1's of that factor. Now, $M_S[\ell][i] = m$ if and only if the ℓ-length factor that starts at position i of S has a total of m 1's, that is, $|S[i\,..\,i+\ell-1]|_1 = m$. We formally define the matrix M_S as follows.

Definition 1. *Given a binary string S of length n, M_S is an $n \times n$ matrix such that $M_S[\ell][i] = |S[i\,..\,i+\ell-1]|_1$, for $1 \leq \ell \leq n$ and $1 \leq i \leq (n - \ell + 1)$, and $M_S[\ell][i] = 0$, otherwise.*

In what follows, we will use $M_S[\ell]$ to refer to Row ℓ of M_S. Assume that we are given two strings A and B on an alphabet Σ. For the sake of ease, we assume that $|A| = |B| = n$. We want to find the length of a longest common abelian factor between A and B.

Definition 2. *Given two strings A and B over the alphabet Σ, we say that w is a common abelian factor for A and B if there exist a factor (or substring) u in A and a factor v in B such that $\mathcal{P}_w = \mathcal{P}_u = \mathcal{P}_v$. A common abelian factor of the highest length is called the* Longest Common Abelian Factor (LCAF) *between A and B. The length of LCAF is referred to as the* LCAF length.

In this paper we study the following problem.

Problem 2.1 (LCAF Problem). *Given two strings A and B over the alphabet Σ, compute the length of an LCAF and identify some occurrences of an LCAF between A and B.*

Assume that the strings A and B of length n are given.

Now, suppose that the matrices M_A and M_B for the binary strings A and B have been computed. Now we have the following easy lemma that will be useful for us later.

Lemma 3. *There is a common abelian factor of length ℓ between A and B if and only if there exists p, q such that $1 \leq p, q \leq n - \ell + 1$ and $M_A[\ell][p] = M_B[\ell][q]$.*

Proof. Suppose there exists p, q such that $1 \leq p, q \leq n - \ell + 1$ and $M_A[\ell][p] = M_B[\ell][q]$. By definition this means $|A[p\,..\,p+\ell-1]|_1 = |B[q\,..\,q+\ell-1]|_1$. So there is a common abelian factor of length ℓ between A and B. The other way is also obvious by definition. □

Clearly, if we have M_A and M_B we can compute the LCAF by identifying the highest ℓ such that there exists p, q having $1 \leq p, q \leq n - \ell + 1$ and $M_A[\ell][p] = M_B[\ell][q]$. Then we can say that the LCAF between A and B is either $A[p\,..\,p+\ell-1]$ or $B[q\,..\,q+\ell-1]$ having length ℓ.

We now generalize the definition of the matrix M_S for strings over a fixed size alphabet $\Sigma = \{a_1, \ldots a_\sigma\}$ by defining an $n \times n$ matrix M_S of $(\sigma-1)$-length vectors. $M_S[\ell][i] = V_{\ell,i}$, where $V_{\ell,i}[j] = |S[i\,..\,i+\ell-1]|_{a_j}$, for $1 \leq \ell \leq n$, $1 \leq i \leq (n-\ell+1)$ and $1 \leq j < \sigma$, and $V_{\ell,i}[j] = 0$, otherwise. We will refer to the j-th element of the array $V_{\ell,i}$ of the matrix M_S by using the notation $M_S[\ell][i][j]$. Notice that the last component of a Parikh vector is determined by using the length of the string and all the other components of the Parikh vector. Now, $M_S[\ell][i][j] = m$ if and only if the ℓ-length factor that starts at position i of S has a total of m a_j's, that is $|S[i\,..\,i+\ell-1]|_{a_j} = m$. Clearly, we can compute $M_S[\ell]$ using the following steps.

Similar to the binary case, the above computation runs in linear time because we can compute $|S[i+1\,..\,i+1+\ell-1]|_{a_j}$ from $|S[i\,..\,i+\ell-1]|_{a_j}$ in constant time by simply decrementing the $S[i]$ component and incrementing the $S[i+\ell]$ one.

3. A Quadratic Algorithm

A simple approach for finding the LCAF length considers computing, for $1 \leq \ell \leq n$, the Parikh vectors of all the factors of length ℓ in both A and B, i.e., $M_A[\ell]$ and $M_B[\ell]$. Then, we check whether $M_A[\ell]$ and $M_B[\ell]$ have non-empty intersection. If yes, then ℓ could be the LCAF length. So, we return the highest of such ℓ.

Moreover, if one knows a Parikh vector having the LCAF length belonging to such intersection, a linear scan of A and B produces one occurrence of such a factor. The asymptotic time complexity of this approach is $\mathcal{O}(\sigma\ n^2)$ and it requires $\mathcal{O}(\sigma\ n \log n)$ bits of extra space. The basic steps are outlined as follows:

```
(1) For $\ell = 1$ to $n$ do the following
(2)     For $i = 1$ to $n - \ell + 1$ do the following
(3)         compute $M_A[\ell][i]$ and $M_B[\ell][i]$
(4)     If $M_A[\ell] \bigcap M_B[\ell] \neq \emptyset$ then
(5)         LCAF = $\ell$.
```

It is easy to establish that, for fixed length ℓ, one can compute all the Parikh vectors in linear time and store them in $\mathcal{O}(\sigma\ n \log n)$ bits. Now once M_A and M_B are computed, we simply need to apply the idea of Lemma 3. The idea is to check for all values of ℓ whether there exists a pair p, q such that $1 \leq p, q \leq n - \ell + 1$ and $M_A[\ell][p] = M_B[\ell][q]$. Then return the highest value of ℓ and corresponding values of p, q.

In the binary case, a Parikh vector is fully represented by just one arbitrary chosen component. Hence, the set of Parikh vectors of binary factors is just a one dimension list of integers that can be stored in $\mathcal{O}(n \log n)$ bits, since we have n values in the range $[0 \mathinner{..} n]$. The intersection can be accomplished in two steps. First, we sort the $M_A[\ell]$ and $M_B[\ell]$ rows in $\mathcal{O}(n)$ time by putting them in two lists and using the classic Counting Sort algorithm [4]. Then, we check for a non empty intersection with a simple linear scan of the two lists in linear time by starting in parallel from the beginning of the two lists and moving forward element by element on the list having the smallest value among the two examined elements. A further linear scan of $M_A[\ell]$ and $M_B[\ell]$ will find the indexes p, q of an element of the not empty intersection. This gives us an $\mathcal{O}(n^2)$ time algorithm requiring $\mathcal{O}(n \log n)$ bits of space for computing an LCAF of two given binary strings.

In the more general case of alphabet greater than two, comparing two Parikh vectors is no more a constant time operation and checking for empty intersections is not a trivial task. In fact, sorting the set of vectors requires a full order to be defined. We can define an order component by component giving more value to the first component, then to the second one and so on. More formally, we define $x < y$, with $x, y \in \mathbb{N}^\sigma$, if there exist $1 \leq k \leq \sigma$ such that $x[k] < y[k]$ and, for any i with $1 \leq i < k$, $x[i] = y[i]$. Notice that comparing two vectors will take $\mathcal{O}(\sigma)$ time.

Now, one can sort two lists of n vectors of dimension $\sigma - 1$, i.e., $M_A[\ell]$ and $M_B[\ell]$, in $\mathcal{O}(\sigma\ n)$ by using n comparisons taking $\mathcal{O}(\sigma)$ each. Therefore, now the algorithm runs in $\mathcal{O}(\sigma\ n^2)$ time using $\mathcal{O}(\sigma\ n \log \sigma)$ bits of extra space.

4. A Sub-Quadratic Algorithm for the Binary Case

In Sec. 3, we have presented an $\mathcal{O}(n^2)$ algorithm to compute the LCAF between two binary strings and two occurrences of common abelian factors, one in each string, having LCAF length. In this section, we show how we can achieve a better running time for the LCAF problem. We will make use of the recent data structure of Moosa and Rahman [8] for indexing an abelian pattern. The results of Moosa and Rahman [8] is presented in the form of following lemmas with appropriate rephrasing to facilitate our description.

Lemma 4. (*Interpolation lemma*). *If S_1 and S_2 are two substrings of a string S on a binary alphabet such that $\ell = |S_1| = |S_2|$, $i = |S_1|_1$, $j = |S_2|_1$, $j > i + 1$, then, there exists another substring S_3 such that $\ell = |S_3|$ and $i < |S_3|_1 < j$.*

Lemma 5. *Suppose we are given a string S of length n on a binary alphabet. Suppose that $maxOne(S, \ell)$ and $minOne(S, \ell)$ denote, respectively, the maximum and minimum number of 1's in any substring of S having length ℓ. Then, for all*

$1 \leq \ell \leq n$, $maxOne(S, \ell)$ and $minOne(S, \ell)$ can be computed in $\mathcal{O}(n^2/\log n)$ time and linear space.

A result similar to Lemma 4 is contained in the paper of Cicalese *et al.* [3], while the result of Lemma 5 has been discovered simultaneously and independently by Moosa and Rahman [8] and by Burcsi *et al.* [2]. Notably, in [9], a slightly better data structure has been presented which assumes word RAM operations. In addition to the above results we further use the following lemma.

Lemma 6. *Suppose we are given two binary strings A, B of length n each. There is a common abelian factor of A and B having length ℓ if and only if $maxOne(B, \ell) \geq minOne(A, \ell)$ and $maxOne(A, \ell) \geq minOne(B, \ell)$.*

Proof. Assume that $min_A = minOne(A, \ell)$, $max_A = maxOne(A, \ell)$, $min_B = minOne(B, \ell)$, $max_B = maxOne(B, \ell)$. Now by Lemma 4, for all $min_A \leq k_A \leq max_A$, we have some ℓ-length substrings $A(k_A)$ of A such that $|A(k_A)|_1 = k_A$. Similarly, for all $min_B \leq k_B \leq max_B$, we have some ℓ-length factors $B(k)$ of B such that $|B(k_B)|_1 = k_B$. Now, consider the range $[min_A \mathinner{..} max_A]$ and $[min_B \mathinner{..} max_B]$. Clearly, these two ranges overlap if and only if $max_B \not< min_A$ and $max_A \not< min_B$. If these two ranges overlap then there exists some k such that $min_A \leq k \leq max_A$ and $min_B \leq k \leq max_B$. Then we must have some substring ℓ-length factors $A(k)$ and $B(k)$. Hence the result follows. □

Let us now focus on devising an algorithm for computing the LCAF given two binary strings A and B of length n. For all $1 \leq \ell \leq n$, we compute $maxOne(A, \ell)$, $minOne(A, \ell)$, $maxOne(B, \ell)$ and $minOne(B, \ell)$ in $\mathcal{O}(n^2/\log n)$ time (Lemma 5). Now we try to check the necessary and sufficient condition of Lemma 6 for all $1 \leq \ell \leq n$ starting from n down to 1. We compute the highest ℓ such that

$$[minOne(A, \ell) \mathinner{..} maxOne(A, \ell)] \text{ and } [minOne(B, \ell) \mathinner{..} maxOne(B, \ell)] \text{ overlap.}$$

Suppose that $\mathcal{K}$ is the set of values that is contained in the above overlap, that is

$$\mathcal{K} = \{\ k \mid k \in [minOne(A, \ell) \mathinner{..} maxOne(A, \ell)] \text{ and } k \in [minOne(B, \ell) \mathinner{..} maxOne(B, \ell)]\ \}.$$

Then by Lemma 6, we must have a set $\mathcal{S}$ of common abelian factors of A, B such that for all $S \in \mathcal{S}$, $|S| = \ell$. Since we identify the highest ℓ, the length of a longest common factor must be ℓ, i.e., LCAF length is ℓ. Additionally, we have further identified the number of 1's in such longest factors in the form of the set $\mathcal{K}$. Also, note that for a $k \in \mathcal{K}$ we must have a factor $S \in \mathcal{S}$ such that $|S|_1 = k$.

Now let us focus on identifying an occurrence of the LCAF. There are a number of ways to do that. But a straightforward and conceptually easy way is to run the folklore ℓ-window based algorithm in [8] on the strings A and B to find the ℓ-length factor with number of 1's equal to a particular value $k \in \mathcal{K}$.

The overall running time of the algorithm is deduced as follows. By Lemma 5, the computation of $maxOne(A, \ell)$, $minOne(A, \ell)$, $maxOne(B, \ell)$ and $minOne(B, \ell)$ can be done in $\mathcal{O}(n^2/\log n)$ time and linear space. The checking of the condition of Lemma 6 can be done in constant time for a particular value of ℓ. Therefore, in total, it can be done in $\mathcal{O}(n)$ time. Finally, the folklore algorithm requires $\mathcal{O}(n)$ time to identify an occurrence (or all of them) of the factors. In total the running time is $\mathcal{O}(n^2/\log n)$ and linear space.

5. Towards a Better Time Complexity

In this section we discuss a simple variant of the quadratic algorithm presented in Sec. 3. We recall that the main idea of the quadratic solution is to find the greatest ℓ with $M_A[\ell] \bigcap M_B[\ell] \neq \emptyset$. The variant we present here is based on the following two simple observations:

(1) One can start considering sets of factors of decreasing lengths;
(2) When an empty intersection is found between $M_A[\ell]$ and $M_B[\ell]$, some rows can possibly be skipped based on the evaluation of the *gap* between $M_A[\ell]$ and $M_B[\ell]$.

The first observation is trivial. The second observation is what we call the *skip trick*. Assume that $M_A[\ell]$ and $M_B[\ell]$ have been computed and $M_A[\ell] \bigcap M_B[\ell] = \emptyset$ have been found. It is easy to see that, for any starting position i and for any component j (i.e., a letter a_j), we have

$$M_A[\ell][i][j] - 1 \leq M_A[\ell-1][i][j] \leq M_A[\ell][i][j] + 1.$$

Exploiting this property, we keep track, along the computation of $M_A[\ell]$ and $M_B[\ell]$, of the minimum and maximum values that appear in Parikh vectors of factors of length ℓ. We use four arrays indexed by σ, namely min_A, max_A, min_B, max_B. Notice that such arrays do not represent Parikh vectors as they just contain min and max values component by component. Formally, $min_A[j] = \min\{M_A[\ell][i][j]\}$, for any $i = 1, \ldots \ell + 1$. The others have similar definitions.

We compare, component by component, the range of a_j in A and B and we skip as many Rows as $\max_{j=1}^{\sigma-1}(min_B[j] - max_A[j])$, assuming $min_B[j] \geq max_A[j]$ (swap A and B, otherwise). The modified algorithm is reported in Algorithm 1.

Note that the tricks employed in our skip trick algorithm are motivated by the fact that the expected value of the LCAF length of an independent and identically distributed (i.i.d.) source is exponentially close to n according to classic Large Deviation results [5]. The same result is classically extended to an ergodic source and it is meant to be a good approximation for real life data when the two strings follow the same probability distribution. Based on this, we have the following conjecture.

Algorithm 1 Compute LCAF of $\boldsymbol{x}$ and $\boldsymbol{y}$ using the ***skip trick***.

```
 1: function ComputeLCAF($\boldsymbol{x}, \boldsymbol{y}$)
 2:     set $\ell = n = |\boldsymbol{x}|$
 3:     while ($\ell \geq 0$) do
 4:         $par_{\boldsymbol{x}} = \mathcal{P}_{x[1..\ell]}$
 5:         $par_{\boldsymbol{y}} = \mathcal{P}_{y[1..\ell]}$
 6:         for ($i = 1$; $i \leq (n - \ell + 1)$; $i$++) do
 7:             if $i == 1$ then
 8:                 SetMinMax($par_{\boldsymbol{x}}, par_{\boldsymbol{y}}$)
 9:             else
10:                 $par_{\boldsymbol{x}}$ = Slide($par_{\boldsymbol{x}}, \boldsymbol{x}, \ell, i$)
11:                 $par_{\boldsymbol{y}}$ = Slide($par_{\boldsymbol{y}}, \boldsymbol{y}, \ell, i$)
12:             end if
13:             push($par_{\boldsymbol{x}}, list_{\boldsymbol{x}}$)
14:             push($par_{\boldsymbol{y}}, list_{\boldsymbol{y}}$)
15:             UpdateMinMax($par_{\boldsymbol{x}}, par_{\boldsymbol{y}}$)
16:         end for
17:         sort list$_{\boldsymbol{x}}$, list$_{\boldsymbol{y}}$
18:         if list$_{\boldsymbol{x}} \bigcap$ list$_{\boldsymbol{y}} \neq \emptyset$ then
19:             return $\ell$
20:         end if
21:         $\ell = \ell -$ Skip($min_{\boldsymbol{x}}, max_{\boldsymbol{x}}, min_{\boldsymbol{y}}, max_{\boldsymbol{y}}$)
22:     end while
23:     return 0
24: end function
25: function Skip($min_{\boldsymbol{x}}, max_{\boldsymbol{x}}, min_{\boldsymbol{y}}, max_{\boldsymbol{y}}$)
26:     $gap = 1$
27:     for ($j = 1$; $j \leq \sigma$; $j$++) do
28:         if $min_{\boldsymbol{x}}[j] > max_{\boldsymbol{y}}[j]$ then
29:             $tmp = min_{\boldsymbol{x}}[j] - max_{\boldsymbol{y}}[j]|$
30:         else
31:             $tmp = min_{\boldsymbol{y}}[j] - max_{\boldsymbol{x}}[j]|$
32:         end if
33:         if $tmp > gap$ then
34:             $gap = tmp$
35:         end if
36:     end for
37:     return $gap$
38: end function
39: function Slide($par, \boldsymbol{s}, \ell, i$)
40:     $par[\boldsymbol{s}[i]]$- -
41:     $par[\boldsymbol{s}[i + l - 1]]$++
42:     return $par$
43: end function
```

```
1: function UpdateMinMax($par_{\boldsymbol{x}}, par_{\boldsymbol{y}}$)
2:     for ($c = 1$; $c \le \sigma$; $c$++) do
3:         if $par_{\boldsymbol{x}}[c] < min_{\boldsymbol{x}}[c]$ then
4:             $min_{\boldsymbol{x}}[c] = par_{\boldsymbol{x}}[c]$
5:         end if
6:         if $par_{\boldsymbol{y}}[c] < min_{\boldsymbol{y}}[c]$ then
7:             $min_{\boldsymbol{y}}[c] = par_{\boldsymbol{y}}[c]$
8:         end if
9:         if $par_{\boldsymbol{x}}[c] > max_{\boldsymbol{x}}[c]$ then
10:            $min_{\boldsymbol{x}}[c] = par_{\boldsymbol{x}}[c]$
11:        end if
12:        if $par_{\boldsymbol{y}}[c] > max_{\boldsymbol{y}}[c]$ then
13:            $min_{\boldsymbol{y}}[c] = par_{\boldsymbol{y}}[c]$
14:        end if
15:    end for
16: end function
17: function SetMinMax($par_{\boldsymbol{x}}, par_{\boldsymbol{y}}$)
18:    $min_{\boldsymbol{x}} = par_{\boldsymbol{x}}$
19:    $min_{\boldsymbol{y}} = par_{\boldsymbol{y}}$
20:    $max_{\boldsymbol{x}} = par_{\boldsymbol{x}}$
21:    $max_{\boldsymbol{y}} = par_{\boldsymbol{y}}$
22: end function
```

Conjecture 5.1. *The expected length of LCAF between two strings A, B drawn from an i.i.d. source is $LCAF_{avg} = n - \mathcal{O}(\log n)$, where $|A| = |B| = n$, and the number of computed rows in Algorithm* 1 *is $\mathcal{O}(\log n)$ in average.*

Finally, we will make use of one more trick to speed up the computation of M rows, except the first one, in Algorithm 2. When our algorithm moves from row $M[\ell{+}1]$ to row of $M[\ell]$, instead of computing the new row from scratch in $O(n)$ time, we compute the first vector $M[\ell][1]$ of the new row in $O(1)$ time by using the first vector $M[\ell{+}1][1]$ of the previous computed row. Subsequently, we slide a window of length ℓ through the row in $n-\ell$ constant time steps while we compute $M[\ell][j+1]$, $1 \le j < n - \ell$. Function *StepDown* in Algorithm 2 is in charge of computing the first Parikh vector of a new row. To compute $M[\ell][1]$ using $M[\ell+1][1]$, we have to subtract 1 from the vector $M[\ell{+}1][1]$ at index $c = \boldsymbol{s}[\ell{+}1]$, that is the last character of the factor $w = \boldsymbol{s}[1..\ell+1]$ of length $\ell + 1$ starting at position 1 in $\boldsymbol{s}$.

For example, consider $\boldsymbol{s} = aacgcctaatcg$, we have $M[12][1] = (4a, 4c, 2g, 2t)$ and $M[11][1] = (4a, 4c, 1g, 2t)$, i.e., $(4a, 4c, 2g, 2t)$ minus $1g$. Then, Function *Slide* in Algorithm 2 computes $M[\ell][j + 1]$ from $M[\ell][j]$, for $1 \le j < n - \ell$, in order to compute the whole row $M[\ell]$. Since we now use the first vector of the previous computed row to compute a new row in M, we have to compute first vector even

Algorithm 2 Compute LCAF of $\boldsymbol{x}$ and $\boldsymbol{y}$ using the ***first vector trick***.

```
function ComputeLCAF($\boldsymbol{x}, \boldsymbol{y}$)
    set $\ell = n = |\boldsymbol{x}|$
    $first_{\boldsymbol{x}} = \mathcal{P}_{x[1..\ell]}$
    $first_{\boldsymbol{y}} = \mathcal{P}_{y[1..\ell]}$
    while ($\ell \geq 0$) do
        $par_{\boldsymbol{x}} = first_{\boldsymbol{x}} =$ Stepdown($first_{\boldsymbol{x}}, \boldsymbol{x}, \ell$)
        $par_{\boldsymbol{y}} = first_{\boldsymbol{y}} =$ Stepdown($first_{\boldsymbol{y}}, \boldsymbol{y}, \ell$)
        for ($i = 1$; $i \leq (n - \ell + 1)$; $i$++) do
            if $i == 1$ then
                SetMinMax($par_{\boldsymbol{x}}, par_{\boldsymbol{y}}$)
            else
                $par_{\boldsymbol{x}} =$ Slide($par_{\boldsymbol{x}}, \boldsymbol{x}, \ell, i$)
                $par_{\boldsymbol{y}} =$ Slide($par_{\boldsymbol{y}}, \boldsymbol{y}, \ell, i$)
            end if
            push($par_{\boldsymbol{x}}, list_{\boldsymbol{x}}$)
            push($par_{\boldsymbol{y}}, list_{\boldsymbol{y}}$)
            UpdateMinMax($par_{\boldsymbol{x}}, par_{\boldsymbol{y}}$)
        end for
        sort list$_{\boldsymbol{x}}$, list$_{\boldsymbol{y}}$
        if list$_{\boldsymbol{x}} \bigcap$ list$_{\boldsymbol{y}} \neq \emptyset$ then
            return $\ell$
        end if
        $skip =$ Skip($min_{\boldsymbol{x}}, max_{\boldsymbol{x}}, min_{\boldsymbol{y}}, max_{\boldsymbol{y}}$)
        while $skip > 1$ do
            $\ell$ - -
            $skip$ - -
            $first_{\boldsymbol{x}} =$ Stepdown($first_{\boldsymbol{x}}, \boldsymbol{x}, \ell$)
            $first_{\boldsymbol{y}} =$ Stepdown($first_{\boldsymbol{y}}, \boldsymbol{y}, \ell$)
        end while
        $\ell$ - -
    end while
    return 0
end function
function StepDown($par, \boldsymbol{s}, \ell$)
    ▷ We assume $\boldsymbol{x}$ and $\boldsymbol{y}$ have a terminal symbol ($).
    if $\boldsymbol{s}[\ell] \neq \$$ then
        $par[\boldsymbol{s}[\ell]]$ - -
    end if
    return $par$
end function
```

when we skipping some rows. Hence, lines $24 - 29$ of Algorithm 2 compute the first vector $M[\ell][1]$ of row $M[\ell]$ when ℓ is a row that is to be skipped.

6. Experiments

We have conducted some experiments to analyze the behaviour and running time of our skip trick algorithm in practice. The experiments have been run on a Windows Server 2008 R2 64-bit Operating System, with Intel(R) Core(TM) i7 2600 processor @ 3.40GHz having an installed memory (RAM) of 8.00 GB. Codes were implemented in $C\#$ language using Visual Studio 2010.

Our first experiment has been carried out principally to verify our rationale behind using the skip trick. We experimentally evaluated the expected number of rows computed in average by using the skip trick of Algorithm 1.

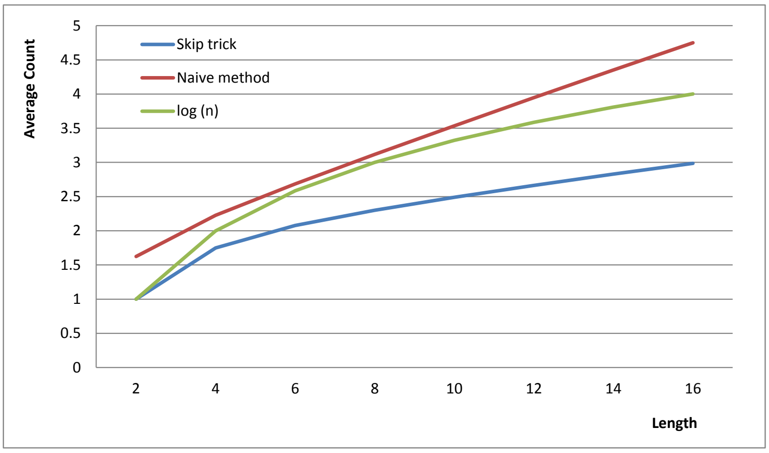

Fig. 1. Plot of the average number of rows computed executing Algorithm 1 on all the strings of length $2, 3, \dots 16$ over the binary alphabet.

Figure 1 shows the average number of rows computed executing Algorithm 1 on all the strings of length $2, 3, \dots 16$ over the binary alphabet. Naive method line refers to the number of rows used without the skip trick, starting from $\ell = n$ and decreasing ℓ by one at each step. Notice that the skip trick line is always below the $\log n$ line. Figure 1 also illustrates that the computed rows, starting from $\ell = n$ to $\ell = n - \log n$, sum up to $\mathcal{O}(\log n)$.

On the other hand, to reach a conclusion in this aspect we would have to increase the value of n in our experiment to substantially more than 64; for $n = 64$, $\sqrt{n}$ is just above $\log n$. Regrettably, limitation of computing power prevents us from doing such an experiment. So, we resort to two more (non-exhaustive) experimental setup as follows to check the practical running time of the skip trick algorithm.

To this end, we conduct our experiments on two datasets, real genomic data and random data. We have taken a sequence ($\mathcal{S}$) from the Homo sapiens genome (250MB) for the former dataset. The latter dataset is generated randomly on the DNA alphabet (i.e., $\Sigma = \{a, c, g, t\}$). In particular, here we have run the skip trick algorithm on 2 sets of pairs of strings of lengths $10, 20, .., 1000$. For the genomic dataset, these pairs of strings have been created as follows. For each length $\ell, \ell \in \{10, 20, .., 1000\}$ two indexes $i, j \in [1..|\boldsymbol{x}| - \ell]$ have been randomly selected to get a pair of strings $\mathcal{S}[i..i + \ell - 1], \mathcal{S}[j..j + \ell - 1]$, each of length ℓ. A total of 1000 pairs of strings have been generated in this way for each length ℓ and the skip trick algorithm has been run on these pairs to get the average results. On the other hand for random dataset, we simply generate the same number of strings pairs randomly and run the skip trick algorithm on each pair of strings and get the average results for each length group. In both cases, we basically count the numbers of computed rows.

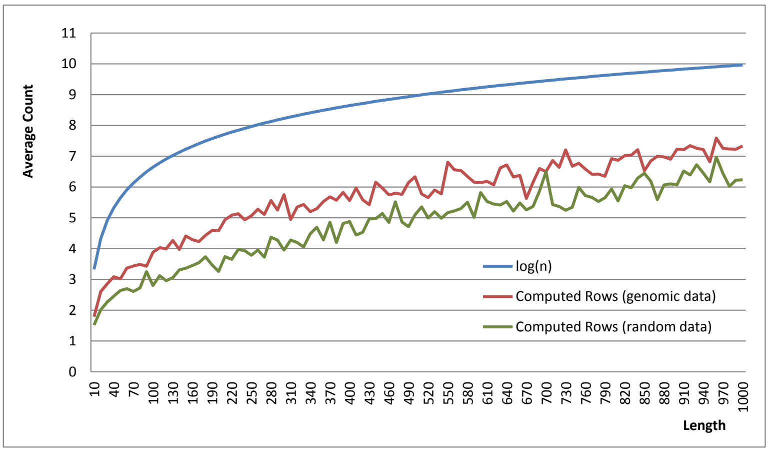

Fig. 2. Plot of the average number of rows computed executing Algorithm 1 on both genomic and random datasets over the DNA alphabet.

Figure 2 shows the average number of rows computed executing Algorithm 1 on both genomic and random datasets over the DNA alphabet (i.e., $\Sigma = \{a, c, g, t\}$). Notice that the skip trick line is always below the $\log n$ line. Figure 2 shows that the computed rows of $\boldsymbol{x}, \boldsymbol{y}$, starting from $\ell = n$ to $\ell = n - \log n$, sum up to $\mathcal{O}(\log n)$.

We have further experimentally evaluated the computation of the first vector and the expected number of rows computed in average by employing the first vector trick (Algorithm 2). We have used the same experimental setup as the above. The average number of rows and of the first vector are counted by executing Algorithm 2 on both genomic and random datasets over the DNA alphabet (i.e., $\Sigma = \{a, c, g, t\}$).

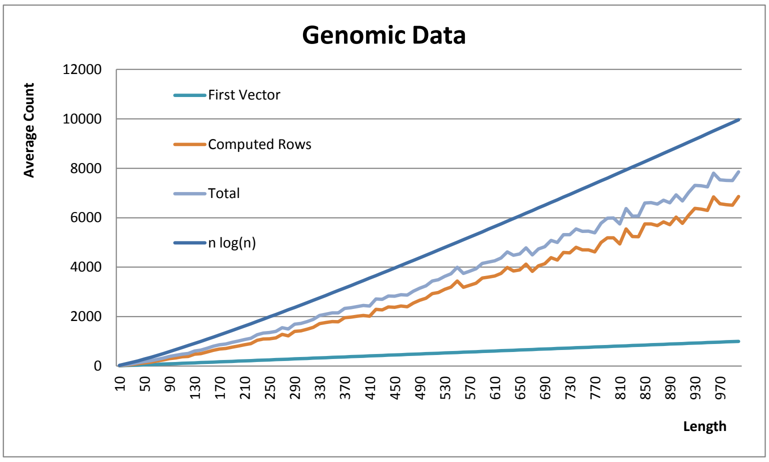

Fig. 3. Plot of the average number of rows computed executing Algorithm 2 on sequences taken from the Homo sapiens genome.

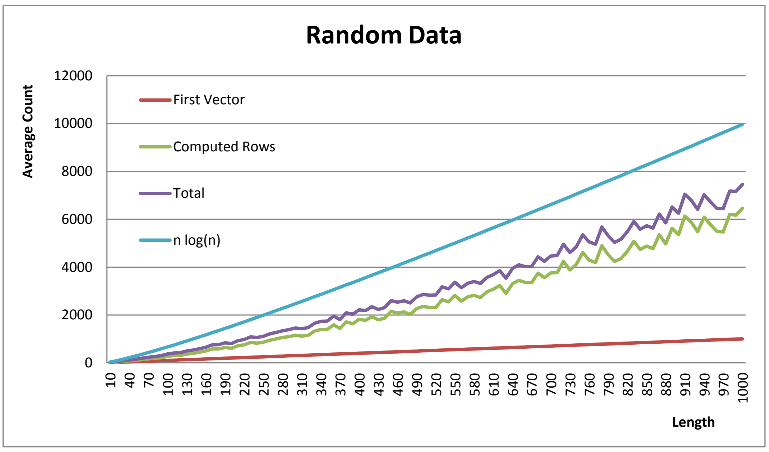

Fig. 4. Plot of the average number of rows computed executing Algorithm 2 on randomly generated sequences over the alphabet $\Sigma = \{a, c, g, t\}$.

The results are illustrated in Figs. 3 and 4. In both cases, the figures report the average count of computed rows (Number of Rows), the average count of the first vector (First Vector) and the summation of these two counts (Total). It also shows the $n \log n$ curve. Both of the figures suggest that the algorithm computed the first vector of the visited rows in $\mathcal{O}(n)$ time and the total running time for Algorithm 2 would be $\mathcal{O}(n \log n)$ in practice. Since any row computation takes $\mathcal{O}(\sigma\ n)$, this suggests an average time complexity of $\mathcal{O}(\sigma\ n\ \log n)$, i.e., $\mathcal{O}(n\ \log n)$ for a constant alphabet.

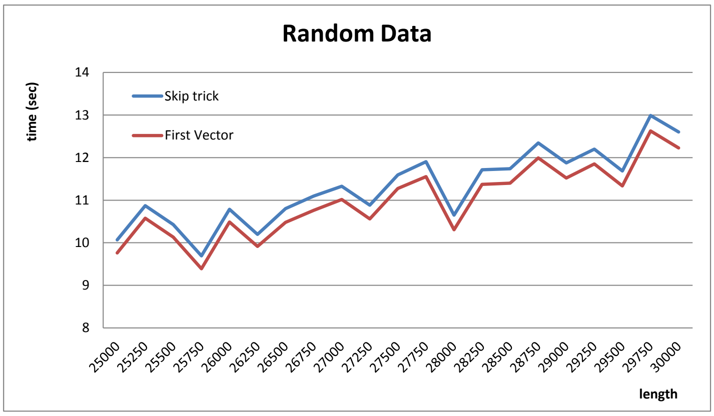

Fig. 5. Plot of the running times executing Algorithms 2 and 1 on randomly generated sequences over the DNA alphabet.

Finally, we experimentally evaluated the two algorithms for their running times. The *LCAF length* is computed by executing Algorithms 2 and 1 on random datasets over the DNA alphabet (i.e., $\Sigma = \{a, c, g, t\}$). The results are reported in Fig. 5.

7. Conclusion

In this paper we present a simple quadratic running time algorithm for the LCAF problem and a sub-quadratic running time solution for the binary string case, both having linear space requirement. Furthermore, we present a variant of the quadratic solution that is experimentally shown to achieve a better time complexity of $\mathcal{O}(n \log n)$.

Acknowledgments

We thank Thierry Lecroq and Arnaud Lefebvre for proposing the LCAF problem and the participants of Stringmasters 2013 meetings for helping us to get more involved with this topic.

References

[1] Ali Alatabbi, Costas S. Iliopoulos, Alessio Langiu, and M. Sohel Rahman. Computing the longest common abelian factor. In *Proceedings of the* 14*th Italian Conference on Theoretical Computer Science* (*ICTCS'*2013), pages 215–221, 2013.

[2] Peter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. On table arrangements, scrabble freaks, and jumbled pattern matching. volume 6099 of *Lecture Notes in Computer Science*, pages 89–101. Springer, 2010.

[3] Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Searching for jumbled patterns in strings. In Jan Holub and Jan Zd'árek, editor, *Stringology*, pages 105–117. Prague Stringology Club, Department of Computer Scienceand Engineering, Faculty of Electrical Engineering, CzechTechnical University in Prague, 2009.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.

[5] R. S. Ellis. *Entropy, Large Deviations, and Statistical Mechanics*. Springer, 1985.

[6] P. Erdös. Some unsolved problems. *Magyar Tud. Akad. Mat. Kutato. Int. Kozl.*, 6:221–254, 1961.

[7] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Cambridge University Press, 1997.

[8] Tanaeem M. Moosa and M. Sohel Rahman. Indexing permutations for binary strings. *Inform. Process. Lett.*, 110(18–19):795–798, September 2010.

[9] Tanaeem M. Moosa and M. Sohel Rahman. Sub-quadratic time and linear space data structures for permutation matching in binary strings. *J. Discrete Algorithms*, 10:5–9, 2012.

[10] Rohit J. Parikh. On context–free languages. *J. Assoc. Comput. Mach.*, 13(4):570–581, 1966.