Longest Property-Preserved Common Factor

Lorraine a K Ayad, Giulia Bernardini, Roberto P Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P Pissis, Giovanna Rosone

Longest Property-Preserved Common Factor

Lorraine A.K Ayad[1], Giulia Bernardini[2], Roberto Grossi[3], Costas S. Iliopoulos[4], Nadia Pisanti[5], Solon P. Pissis[6], and Giovanna Rosone[7]

[1]Department of Informatics, King's College London, London, UK,
`lorraine.ayad@kcl.ac.uk`
[2]Department of Informatics, Systems and Communication (DISCo), University of
Milan-Bicocca, Italy, `giulia.bernardini@unimib.it`
[3]Department of Computer Science, University of Pisa, Italy and ERABLE Team, INRIA,
France, `grossi@di.unipi.it`
[4]Department of Informatics, King's College London, London, UK, `c.iliopoulos@kcl.ac.uk`
[5]Department of Computer Science, University of Pisa, Italy and ERABLE Team, INRIA,
France, `pisanti@di.unipi.it`
[6]Department of Informatics, King's College London, London, UK, `solon.pissis@kcl.ac.uk`
[7]Department of Computer Science, University of Pisa, Italy, `giovanna.rosone@unipi.it`

Abstract

In this paper we introduce a new family of string processing problems. We are given two or more strings and we are asked to compute a factor common to all strings that preserves a specific property and has maximal length. Here we consider three fundamental string properties: square-free factors, periodic factors, and palindromic factors under three different settings, one per property. In the first setting, we are given a string x and we are asked to construct a data structure over x answering the following type of on-line queries: given string y, find a longest square-free factor common to x and y. In the second setting, we are given k strings and an integer $1 < k' \le k$ and we are asked to find a longest periodic factor common to at least k' strings. In the third setting, we are given two strings and we are asked to find a longest palindromic factor common to the two strings. We present linear-time solutions for all settings. We anticipate that our paradigm can be extended to other string properties or settings.

1 Introduction

In the longest common factor problem, also known as longest common substring problem, we are given two strings x and y, each of length at most n, and we are asked to find a maximal-length string occurring in both x and y. This is a classical and well-studied problem in computer science arising out of different practical scenarios. It can be solved in $\mathcal{O}(n)$ time and space [10, 18] (see also [21, 26]). Recently, the same problem has been extensively studied under distance metrics; that is, the sought factors (one from x and one from y) must be at distance at most k and have maximal length [8, 28, 27, 2, 25, 24] (and references therein).

In this paper we initiate a new related line of research. We are given two or more strings and our goal is to compute a *factor* common to all strings that preserves a specific *property* and has maximal length. An analogous line of research was introduced in [11]. It focuses on computing a *subsequence* (rather than a factor) common to all strings that preserves a specific property and has

1

maximal length. Specifically, in [11, 3, 19], the authors considered computing a longest common palindromic subsequence and in [20] computing a longest common square subsequence.

We consider three fundamental string properties: *square-free* factors, *periodic*, and *palindromic* factors [23] under three different settings, one per property. In the first setting, we are given a string x and we are asked to construct a data structure over x answering the following type of on-line queries: given string y, find a longest square-free factor common to x and y. In the second setting, we are given k strings and an integer $1 < k' \leq k$ and we are asked to find a longest periodic factor common to at least k' strings. In the third setting, we are given two strings and we are asked to find a longest palindromic factor common to the two strings. We present linear-time solutions for all settings. We anticipate that our paradigm can be extended to other string properties or settings.

1.1 Definitions and Notation

An *alphabet* Σ is a non-empty finite ordered set of letters of size $\sigma = |\Sigma|$. In this work we consider that $\sigma = \mathcal{O}(1)$ or that Σ is a linearly-sortable integer alphabet. A *string* x on an alphabet Σ is a sequence of elements of Σ. The set of all strings on an alphabet Σ, including the *empty string* ε of length 0, is denoted by Σ^*. For any string x, we denote by $x[i..j]$ the *substring* (sometimes called *factor*) of x that starts at position i and ends at position j. In particular, $x[0..j]$ is the *prefix* of x that ends at position j, and $x[i..|x| - 1]$ is the *suffix* of x that starts at position i, where $|x|$ denotes the *length* of x. A string uu, $u \in \Sigma^*$, is called a *square*. A *square-free* string is a string that does not contain a square as a factor.

A *period* of $x[0..|x| - 1]$ is a positive integer p such that $x[i] = x[i + p]$ holds for all $0 \leq i < |x| - p$. The smallest period of x is denoted by $\mathsf{per}(x)$. String u is called *periodic* if and only if $\mathsf{per}(u) \leq |u|/2$. A *run* of string x is an interval $[i, j]$ such that for the smallest period $p = \mathsf{per}(x[i..j])$ it holds that $2p \leq j - i + 1$ and the periodicity cannot be extended to the left or right, *i.e.*, $i = 0$ or $x[i - 1] \neq x[i + p - 1]$, and, $j = |x| - 1$ or $x[j - p + 1] \neq x[j + 1]$.

We denote the *reversal* of x by string x^R, i.e. $x^R = x[|x| - 1]x[|x| - 2] \ldots x[0]$. A string p is said to be a *palindrome* if and only if $p = p^R$. If factor $x[i..j]$, $0 \leq i \leq j \leq n - 1$, of string x of length n is a palindrome, then $\frac{i+j}{2}$ is the *center* of $x[i..j]$ in x and $\frac{j-i+1}{2}$ is the *radius* of $x[i..j]$. In other words, a palindrome is a string that reads the same forward and backward, i.e. a string p is a palindrome if $p = yay^R$ where y is a string, y^R is the reversal of y and a is either a single letter or the empty string. Moreover, $x[i..j]$ is called a *palindromic factor* of x. It is said to be a *maximal palindrome* if there is no other palindrome in x with center $\frac{i+j}{2}$ and larger radius. Hence x has exactly $2n - 1$ maximal palindromes. A maximal palindrome p of x can be encoded as a pair (c, r), where c is the center of p in x and r is the radius of p.

1.2 Algorithmic Toolbox

The maximum number of runs in a string of length n is less than n [4], and, moreover, all runs can be computed in $\mathcal{O}(n)$ time [22, 4].

The *suffix tree* $\mathsf{ST}(x)$ of a non-empty string x of length n is a compact trie representing all suffixes of x. $\mathsf{ST}(x)$ can be constructed in $\mathcal{O}(n)$ time [14]. We can analogously define and construct the *generalised suffix tree* $\mathsf{GST}(x_0, x_1, \ldots, x_{k-1})$ for a set of k strings. We assume the reader is familiar with these data structures.

The matching statistics capture all matches between two strings x and y [7]. More formally, the *matching statistics* of a string $y[0..|y| - 1]$ with respect to a string x is an array $\mathsf{MS}_y[0..|y| - 1]$, where $\mathsf{MS}_y[i]$ is a pair (ℓ_i, p_i) such that (i) $y[i..i + \ell_i - 1]$ is the longest prefix of $y[i..|y| - 1]$ that is

2

a factor of x; and (ii) $x[p_i..p_i + \ell_i - 1] = y[i..i + \ell_i - 1]$. Matching statistics can be computed in $\mathcal{O}(|y|)$ time for $\sigma = \mathcal{O}(1)$ by using $\mathsf{ST}(x)$ [18, 6, 16].

Given a rooted tree T with n leaves coloured from 0 to $k - 1$, $1 < k \leq n$, the *colour set size* problem is finding, for each internal node u of T, the number of different leaf colours in the subtree rooted at u. In [10], the authors present an $\mathcal{O}(n)$-time solution to this problem.

In the *weighted ancestor* problem, introduced in [15], we consider a rooted tree T with an integer weight function μ defined on the nodes. We require that the weight of the root is zero and the weight of any other node is strictly larger than the weight of its parent. A weighted ancestor query, given a node v and an integer value $\ell \leq \mu(v)$, asks for the highest ancestor u of v such that $\mu(u) \geq \ell$, *i.e.*, such an ancestor u that $\mu(u) \geq \ell$ and $\mu(u)$ is the smallest possible. When T is the suffix tree of a string x of length n, we can locate the locus of any factor of $x[i..j]$ using a weighted ancestor query. We define the weight of a node of the suffix tree as the length of the string it represents. Thus a weighted ancestor query can be used for the terminal node corresponding to $x[i..n - 1]$ to create (if necessary) and mark the node that corresponds to $x[i..j]$. Given a collection Q of weighted ancestor queries on a weighted tree T on n nodes with integer weights up to $n^{\mathcal{O}(1)}$, all the queries in Q can be answered *off-line* in $\mathcal{O}(n + |Q|)$ time [5].

2 Square-Free-Preserved Matching Statistics

In this section, we introduce the square-free-preserved matching statistics problem and provide a linear-time solution. In the *square-free-preserved matching statistics* problem we are given a string x of length n and we are asked to construct a data structure over x answering the following type of on-line queries: given string y, find the longest square-free prefix of $y[i..|y| - 1]$ that is a factor of x, for all $0 \leq i < |y| - 1$. (For related work see [12].) We represent the answer using an integer array $\mathsf{SQMS}_y[0..|y| - 1]$ of lengths, but we can trivially modify our algorithm to report the actual factors. It should be clear that a maximum element in SQMS gives the length of some longest square-free factor common to x and y.

Construction. Our data structure over string x consists of the following:

- An integer array $L_x[0..n - 1]$, where $L_x[i]$ stores the length of the longest square-free factor starting at position i of string x.

- The suffix tree $\mathsf{ST}(x)$ of string x.

The idea for constructing array L_x efficiently is based on the following crucial observation.

Observation 1. *If $x[i..n - 1]$ contains a square then $L_x[i] + 1$, for all $0 \leq i < n$, is the length of the* shortest *prefix of $x[i..n - 1]$ (factor f) containing a square. In fact, the square is a suffix of f, otherwise f would not have been the shortest. If $x[i..n - 1]$ does not contain a square then $L_x[i] = n - i$.*

We thus shift our focus to computing the shortest such prefixes. We start by considering the runs of x. Specifically, we consider squares in x observing that a run $[\ell, r]$ with period p contains $r - \ell - 2p + 2$ squares of length $2p$ with the leftmost one starting at position ℓ. Let $r' = \ell + 2p - 1$ denote the ending position of the leftmost such square of the run. In order to find, for all i's, the shortest prefix of $x[i..n - 1]$ containing a square s, and thus compute $L_x[i]$, we have two cases:

1. s is part of a run $[\ell, r]$ in x that starts *after* i. In particular, $s = x[\ell..r']$ such that $r' \leq r$, $\ell > i$, and r' is minimal. In this case the shortest factor has length $\ell + 2p - i$; we store this value in an integer array $C[0..n - 1]$. If no run starts after position i we set $C[i] = \infty$. To compute C,

after computing in $\mathcal{O}(n)$ time all the runs of x with their p and r' [22, 4], we sort them by r'. A right-to-left scan after this sorting associates to i the closest r' with $\ell > i$.

2. s is part of a run $[\ell, r]$ in x and $i \in [\ell, r]$. This implies that if $i \leq r - 2p + 1$ then a square *starts at* i and we store the length of the shortest such square in an integer array $S[0..n-1]$. If no square starts at position i we set $S[i] = \infty$. Array S can be constructed in $\mathcal{O}(n)$ time by applying the algorithm of [13].

Since we do not know which of the two cases holds, we compute both C and S. By Observation 1, if $C[i] = S[i] = \infty$ ($x[i..n-1]$ does not contain a square) we set $L_x[i] = n - i$; otherwise ($x[i..n-1]$ contains a square) we set $L_x[i] = \min\{C[i], S[i]\} - 1$.

Finally, we build the suffix tree $\mathsf{ST}(x)$ of string x in $\mathcal{O}(n)$ time [14]. This completes our construction.

Querying. We rely on the following fact for answering the queries efficiently.

Fact 1. *Every factor of a square-free string is square-free.*

Let string y be an on-line query. Using $\mathsf{ST}(x)$, we compute the matching statistics MS_y of y with respect to x. For each $j \in [0, |y| - 1]$, $\mathsf{MS}_y[j] = (\ell_i, i)$ indicates that $x[i..i + \ell_i - 1] = y[j..j + \ell_i - 1]$. This computation can be done in $\mathcal{O}(|y|)$ time [18, 6]. By applying Fact 1, we can answer any query y in $\mathcal{O}(|y|)$ time for $\sigma = \mathcal{O}(1)$ by setting $\mathsf{SQMS}_y[j] = \min\{\ell_i, L_x[i]\}$, for all $0 \leq j \leq |y| - 1$.

We arrive at the following result.

Theorem 1. *Given a string x of length n over an alphabet of size $\sigma = \mathcal{O}(1)$, we can construct a data structure of size $\mathcal{O}(n)$ in time $\mathcal{O}(n)$, answering SQMS_y on-line queries in $\mathcal{O}(|y|)$ time.*

Proof. The time complexity of our algorithm follows from the above discussion.

We next show the correctness of our algorithm. Let us first show the correctness of computing array L_x. The square contained in the shortest prefix of $x[i..n-1]$ (containing a square) starts by definition either at i or after i. If it starts at i this is correctly computed by the algorithm of [13] which assigns the length of the shortest such square in $S[i]$. If it starts after i it must be the leftmost square of another run by the runs definition. $C[i]$ stores the length of the shortest prefix containing such a square. Then by Observation 1, $L_x[i]$ is computed correctly.

It suffices to show that, if w is the longest square-free substring common to x and y occurring at position i_x in x and at position i_y in y, then (i) $\mathsf{MS}_y[i_y] = (\ell, i_x)$ with $\ell \geq |w|$ and $x[i_x..i_x + \ell - 1] = y[i_y..i_y + \ell - 1]$; (ii) w is a prefix of $x[i_x..i_x + L_x[i_x] - 1]$; and (iii) $\mathsf{SQMS}_y[i_y] = |w|$. Case (i) directly follows from the correctness of the matching statistics algorithm. For Case (ii), since w occurs at i_x and w is square-free, $L_x[i_x] \geq |w|$. For Case (iii), since w is square-free we have to show that $|w| = \min\{\ell_i, L_x[i]\}$. We know from (i) that $\ell \geq |w|$ and from (ii) that $L_x[i_x] \geq |w|$. If $\min\{\ell_i, L_x[i]\} = \ell$, then w cannot be extended because the possibly longer than $|w|$ square-free string occurring at i_x does not occur in y, and in this case $|w| = \ell$. Otherwise, if $\min\{\ell_i, L_x[i]\} = L_x[i_x]$ then w cannot be extended because it is no longer square-free, and in this case $|w| = L_x[i_x]$. Hence we conclude that $\mathsf{SQMS}_y[i_y] = |w|$. The statement follows. $\square$

The following example provides a complete overview of the workings of our algorithm.

Example 1. Let $x = \mathtt{aababaababb}$ and $y = \mathtt{babababbaaab}$. The length of a longest common square-free factor is 3, and the factors are $\mathtt{bab}$ and $\mathtt{aba}$.

i	0	1	2	3	4	5	6	7	8	9	10	
$x[i]$	a	a	b	a	b	a	a	b	a	b	b	
$C[i]$	5	6	5	4	3	5	5	4	3	∞	∞	
$S[i]$	2	4	4	6	∞	2	4	∞	∞	2	∞	
$L_x[i]$	1	3	3	3	2	1	3	3	2	1	1	
j	0	1	2	3	4	5	6	7	8	9	10	11
$y[j]$	b	a	b	a	b	a	b	b	a	a	a	b
$\mathsf{MS}_y[j]$	(4,2)	(5,1)	(4,2)	(5,6)	(4,7)	(3,8)	(2,9)	(3,4)	(2,0)	(3,0)	(2,1)	(1,2)
$\mathsf{SQMS}_y[j]$	3	3	3	3	3	2	1	2	1	1	2	1

3 Longest Periodic-Preserved Common Factor

In this section, we introduce the longest periodic-preserved common factor problem and provide a linear-time solution. In the *longest periodic-preserved common factor* problem, we are given $k \geq 2$ strings $x_0, x_1, \ldots, x_{k-1}$ of total length N and an integer $1 < k' \leq k$, and we are asked to find a longest periodic factor common to at least k' strings. In what follows we present two different algorithms to solve this problem. We represent the answer $\mathsf{LPCF}_{k'}$ by the length of a longest factor, but we can trivially modify our algorithms to report an actual factor. Our first algorithm, denoted by LPCF, works as follows.

1. Compute the runs of string x_j, for all $0 \leq j < k$.

2. Construct the generalised suffix tree $\mathsf{GST}(x_0, x_1, \ldots, x_{k-1})$ of $x_0, x_1, \ldots, x_{k-1}$.

3. For each string x_j and for each run $[\ell, r]$ with period p_ℓ of x_j, augment GST with the explicit node spelling $x_j[\ell..r]$, decorate it with p_ℓ, and mark it as a *candidate* node. This can be done as follows: for each run $[\ell, r]$ of x_j, for all $0 \leq j < k$, find the leaf corresponding to $x_j[\ell..|x_j|{-}1]$ and answer the weighted ancestor query in GST with weight $r - \ell + 1$. Moreover, mark as candidates all *explicit* nodes spelling a prefix of length d of any run $[\ell, r]$ with $2p_\ell \leq d$.

4. Mark as *good* the nodes of the tree having at least k' different colours on the leaves of the subtree rooted there. Let aGST be this augmented tree.

5. Return as $\mathsf{LPCF}_{k'}$ the string depth of a candidate node in aGST which is also a good node, and that has maximal string depth (if any, otherwise return 0).

Theorem 2. *Given k strings of total length N on alphabet $\Sigma = \{1, \ldots, N^{\mathcal{O}(1)}\}$, and an integer $1 < k' \leq k$, algorithm LPCF returns $\mathsf{LPCF}_{k'}$ in time $\mathcal{O}(N)$.*

Proof. Let us assume wlog that $k' = k$, and let w with period p be the longest periodic factor common to all strings. By the construction of aGST (Steps 1-4), the path spelling w leads to a good node n_w as w occurs in all the strings. We make the following observation.

Observation 2. *Each periodic factor with period p of string x is a factor of $x[i..j]$, where $[i, j]$ is a run with period p.*

By Observation 2, in all strings, w is included in a run having the same period. Observe that for at least one of the strings, there is a run ending with w, otherwise we could extend w obtaining a longer periodic common factor (similarly, for at least one of the strings, there is a run starting with w). Therefore n_w is *both* a good and a candidate node. By definition, n_w is at string depth at
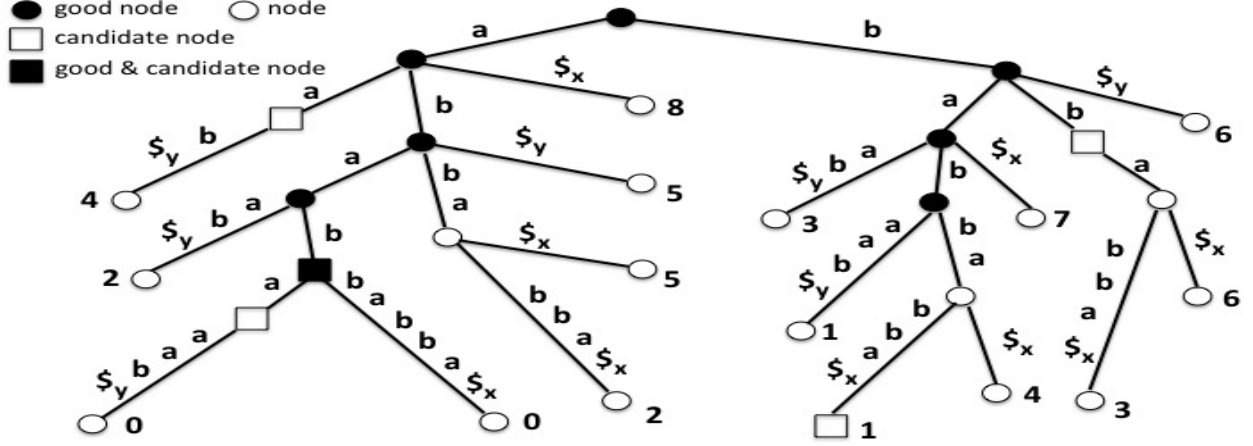
Figure 1: aGST for $x =$ ababbabba, $y =$ ababaab, and $k = k' = 2$.

least $2p$ and, by construction, $\mathsf{LPCF}_{k'}$ is the string depth of a deepest such node; thus $|w|$ will be returned by Step 5.

As for the time complexity, Step 1 [22, 4] and Step 2 [14] can be done in $\mathcal{O}(N)$ time. Since the total number of runs is less than N [4], Step 3 can be done in $\mathcal{O}(N)$ time using off-line weighted ancestor queries [5] to mark the runs as candidate nodes; and then a post-order traversal to mark their ancestor explicit nodes as candidates, if their string-depth is at least $2p_\ell$ for any run $[\ell, r]$ with period p_ℓ. The size of the aGST is still in $\mathcal{O}(N)$. Step 4 can be done in $\mathcal{O}(N)$ time [10]. Step 5 can be done in $\mathcal{O}(N)$ by a post-order traversal of aGST. $\qquad\square$

The following example provides a complete overview of the workings of our algorithm.

Example 2. Consider $x =$ ababbabba, $y =$ ababaab, and $k = k' = 2$. The runs of x are: $r_0 = [0, 3]$, $\mathsf{per}(\mathsf{abab}) = 2$, $r_1 = [1, 8]$, $\mathsf{per}(\mathsf{babbabba}) = 3$, $r_2 = [3, 4]$, $\mathsf{per}(\mathsf{bb}) = 1$, and $r_3 = [6, 7]$, $\mathsf{per}(\mathsf{bb}) = 1$; those of y are $r_4 = [0, 4]$, $\mathsf{per}(\mathsf{ababa}) = 2$ and $r_5 = [4, 5]$, $\mathsf{per}(\mathsf{aa}) = 1$. Fig 1 shows aGST for x, y, and $k = k' = 2$. Algorithm LPCF outputs $4 = |\mathsf{abab}|$, with $\mathsf{per}(\mathsf{abab}) = 2$, as the node spelling abab is the deepest good one that is also a candidate.

We next present a second algorithm to solve this problem with the same time complexity but without the use of off-line weighted ancestor queries. The algorithm works as follows.

1. Compute the runs of string x_j, for all $0 \leq j < k$.

2. Construct the generalised suffix tree $\mathsf{GST}(x_0, x_1, \ldots, x_{k-1})$ of $x_0, x_1, \ldots, x_{k-1}$.

3. Mark as *good* the nodes of GST having at least k' different colours on the leaves of the subtree rooted there.

4. Compute and store, for every leaf node, the *nearest* ancestor that is good.

5. For each string x_j and for each run $[\ell, r]$ with period p_ℓ of x_j, check the nearest good ancestor for the leaf corresponding to $x_j[\ell..|x_j| - 1]$. Let d be the string-depth of the nearest good ancestor. Then:

   (a) If $r - \ell + 1 \leq d$, the entire run is also good.
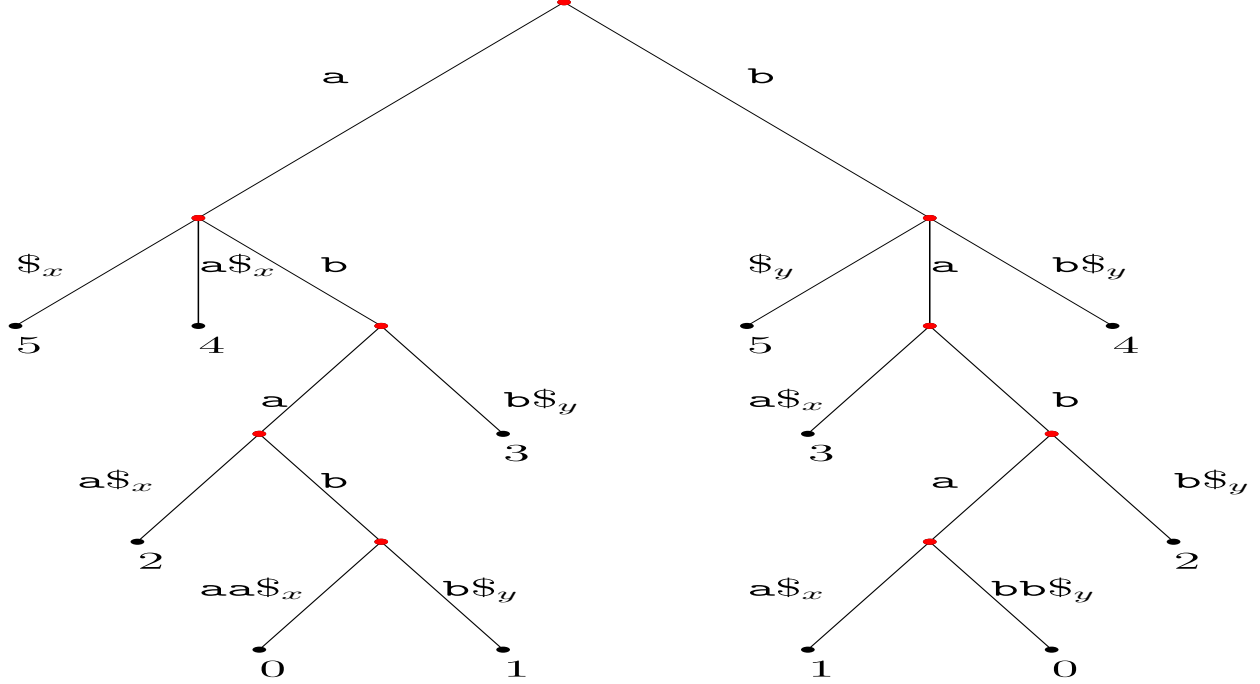   (b) If $r - \ell + 1 > d$, check if $2p_\ell \leq d$, and if so the string for the good ancestor is periodic.

6

Figure 2: GST for $x = $ ababaa, $y = $ bababb, and $k = k' = 2$. Good nodes are marked red.

6. Return as $\mathsf{LPCF}_{k'}$ the maximal string depth found in Step 5 (if any, otherwise return 0).

Let us analyse this algorithm. Let us assume wlog that $k' = k$, and let w with period p be the longest periodic factor common to all strings. By the construction of GST (Steps 1-3), the path spelling w leads to a good node n_w as w occurs in all the strings.

By Observation 2, in all strings, w is included in a run having the same period. Observe that for at least one of the strings, there is a run starting with w, otherwise we could extend w obtaining a longer periodic common factor. So the algorithm should check, for each run, if there is a periodic-preserved common prefix of the run and take the longest such prefix. $\mathsf{LPCF}_{k'}$ is the string depth of a deepest good node spelling a periodic factor; thus $|w|$ will be returned by Step 6.

As for the time complexity, Step 1 [22, 4] and Step 2 [14] can be done in $\mathcal{O}(N)$ time. Step 3 can be done in $\mathcal{O}(N)$ time [10] and Step 4 can be done in $\mathcal{O}(N)$ time by using a tree traversal. Since the total number of runs is less than N [4], Step 5 can be done in $\mathcal{O}(N)$ time. We thus arrive at Theorem 2 with a different algorithm.

The following example provides a complete overview of the workings of our algorithm.

Example 3. Consider $x = $ ababaa, $y = $ bababb, and $k = k' = 2$. The runs of x are: $r_0 = [0, 4]$, $\mathsf{per}(\mathsf{ababa}) = 2$, $r_1 = [4, 5]$, $\mathsf{per}(\mathsf{aa}) = 1$; those of y are $r_2 = [0, 4]$, $\mathsf{per}(\mathsf{babab}) = 2$ and $r_3 = [4, 5]$, $\mathsf{per}(\mathsf{bb}) = 1$. Fig 2 shows GST for x, y, and $k = k' = 2$. Consider the run $r_0 = [0, 4]$. The nearest good node of leaf spelling $x[0..|x| - 1]$ is the node spelling abab. We have that $r - \ell + 1 = 5 > d = 4$, and $2p = 4 \leq d = 4$. The algorithm outputs $4 = |\mathsf{abab}|$ as abab is a longest periodic-preserved common factor. Another longest periodic-preserved common factor is baba.

4 Longest Palindromic-Preserved Common Factor

In this section, we introduce the longest palindromic-preserved common factor problem and provide a linear-time solution. In the *longest palindromic-preserved common factor* problem, we are given two strings x and y, and we are asked to find a longest palindromic factor common to the two strings. (For related work in a dynamic setting see [17, 1].) We represent the answer LPALCF by the length of a longest factor, but we can trivially modify our algorithm to report an actual factor. Our algorithm is denoted by LPALCF. In the description below, for clarity, we consider odd-length palindromes only. (Even-length palindromes can be handled in an analogous manner.)

1. Compute the maximal odd-length palindromes of x and the maximal odd-length palindromes of y.

2. Collect the factors $x[i..i']$ of x (resp. the factors $y[j..j']$ of y) such that i (j) is the center of an odd-length maximal palindrome of x (y) and i' (j') is the ending position of the odd-length maximal palindrome centered at i (j).

3. Create a lexicographically sorted list L of these strings from x and y.

4. Compute the longest common prefix of consecutive entries (strings) in L.

5. Let ℓ be the maximal length of longest common prefixes between any string from x and any string from y. For odd lengths, return LPALCF$= 2\ell - 1$.

Theorem 3. *Given two strings x and y on alphabet $\Sigma = \{1, \ldots, (|x| + |y|)^{\mathcal{O}(1)}\}$, algorithm* LPALCF *returns* LPALCF *in time* $\mathcal{O}(|x| + |y|)$.

Proof. The correctness of our algorithm follows directly from the following observation.

Observation 3. *Any longest palindromic-preserved common factor is a factor of a maximal palindrome of x with the same center and a factor of a maximal palindrome of y with the same center.*

Step 1 can be done in $\mathcal{O}(|x| + |y|)$ time [18]. Step 2 can be done in $\mathcal{O}(|x| + |y|)$ time by going through the set of maximal palindromes computed in Step 1. Step 3 and Step 4 can be done in $\mathcal{O}(|x| + |y|)$ time by constructing the data structure of [9]. Step 5 can be done in $\mathcal{O}(|x| + |y|)$ time by going through the list of computed longest common prefixes. ∎

The following example provides a complete overview of the workings of our algorithm.

Example 4. Consider $x =$ ababaa and $y =$ bababb. In Step 1 we compute all maximal palindromes of x and y. Considering odd-length palindromes gives the following factors (Step 2) from x: $x[0..0] =$ a, $x[1..2] =$ ba, $x[2..4] =$ aba, $x[3..4] =$ ba, $x[4..4] =$ a, and $x[5..5] =$ a. The analogous factors from y are: $y[0..0] =$ b, $y[1..2] =$ ab, $y[2..4] =$ bab, $y[3..4] =$ ab, $y[4..4] =$ b, and $y[5..5] =$ b. We sort these strings lexicographically and compute the longest common prefix information (Steps 3-4). We find that $\ell = 2$: the maximal longest common prefixes are ba and ab, denoting that aba and bab are the longest palindromic-preserved common factors of odd length. In fact, algorithm LPALCF outputs $2\ell - 1 = 3$ as aba and bab are the longest palindromic-preserved common factors of any length.

5 Final Remarks

In this paper, we introduced a new family of string processing problems. The goal is to compute factors common to a set of strings preserving a specific property and having maximal length. We showed linear-time algorithms for square-free, periodic, and palindromic factors under three different settings. We anticipate that our paradigm can be extended to other string properties or settings.

Acknowledgements

References

[1] Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor made fully dynamic. *CoRR*, abs/1804.08731, 2018.

[2] Lorraine A. K. Ayad, Carl Barton, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. Longest common prefixes with k-errors and applications. In *SPIRE*, volume 11147 of *LNCS*, pages 27–41. Springer, 2018.

[3] Sang Won Bae and Inbok Lee. On finding a longest common palindromic subsequence. *Theoretical Computer Science*, 710:29–34, 2018. Advances in Algorithms & Combinatorics on Strings (Honoring 60th birthday for Prof. Costas S. Iliopoulos).

[4] Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017.

[5] Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, and Jakub Radoszewski. Indexing weighted sequences: Neat and efficient. *CoRR*, abs/1704.07625, 2017.

[6] Djamal Belazzougui and Fabio Cunial. Indexed matching statistics and shortest unique substrings. In Edleno Silva de Moura and Maxime Crochemore, editors, *21st International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 8799 of *LNCS*, pages 179–190, 2014.

[7] W. I. Chang and E. L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4):327–344, 1994.

[8] Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Linear-time algorithm for long LCF with k mismatches. In *CPM*, volume 105 of *LIPIcs*, pages 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[9] Panagiotis Charalampopoulos, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Property suffix array with applications. In Michael A. Bender, Martin Farach-Colton, and Miguel A.

Mosteiro, editors, *LATIN 2018: Theoretical Informatics - 13th Latin American Symposium, Buenos Aires, Argentina, April 16-19, 2018, Proceedings*, volume 10807 of *Lecture Notes in Computer Science*, pages 290–302. Springer, 2018.

[10] Lucas Chi and Kwong Hui. Color set size problem with applications to string matching. In *Combinatorial Pattern Matching*, pages 230–243. Springer Berlin Heidelberg, 1992.

[11] Shihabur Rahman Chowdhury, Md. Mahbubul Hasan, Sumaiya Iqbal, and M. Sohel Rahman. Computing a longest common palindromic subsequence. *Fundam. Inf.*, 129(4):329–340, 2014.

[12] Marius Dumitran, Florin Manea, and Dirk Nowotka. On prefix/suffix-square free words. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *22nd International Symposium, on String Processing and Information Retrieval (SPIRE)*, volume 9309 of *LNCS*, pages 54–66, 2015.

[13] Jean-Pierre Duval, Roman Kolpakov, Gregory Kucherov, Thierry Lecroq, and Arnaud Lefebvre. Linear-time computation of local periods. *Theoretical Computer Science*, 326(1):229–240, 2004.

[14] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.

[15] Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *7th Symposium on Combinatorial Pattern Matching (CPM)*, pages 130–140. 1996.

[16] Maria Federico and Nadia Pisanti. Suffix tree characterization of maximal motifs in biological sequences. *Theor. Comput. Sci.*, 410(43):4391–4401, 2009.

[17] Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest substring palindrome after edit. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, volume 105 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[19] Shunsuke Inenaga and Heikki Hyyrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Information Processing Letters*, 129:11–15, 2018.

[20] Takafumi Inoue, Shunsuke Inenaga, Heikki Hyyrö, Hideo Bannai, and Masayuki Takeda. Computing longest common square subsequences. In *29th Symposium on Combinatorial Pattern Matching (CPM)*, volume 105 of *LIPIcs*, pages 15:1–15:13, 2018.

[21] Tomasz Kociumaka, Tatiana A. Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 605–617, 2014.

[22] Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Symposium on Foundations of Comp Science*, pages 596–604, 1999.

[23] M. Lothaire. *Applied Combinatorics on Words*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2005.

[24] Pierre Peterlongo, Nadia Pisanti, Frédéric Boyer, Alair Pereira do Lago, and Marie-France Sagot. Lossless filter for multiple repetitions with hamming distance. *J. Discr. Alg.*, 6(3):497–509, 2008.

[25] Pierre Peterlongo, Nadia Pisanti, Frédéric Boyer, and Marie-France Sagot. Lossless filter for finding long multiple approximate repetitions using a new data structure, the bi-factor array. In *12th International Symposium String Processing and Information Retrieval, 12th International Conference (SPIRE)*, pages 179–190, 2005.

[26] Tatiana A. Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In *24th Symposium on Combinatorial Pattern Matching (CPM)*, pages 223–234, 2013.

[27] Sharma V. Thankachan, Chaitanya Aluru, Sriram P. Chockalingam, and Srinivas Aluru. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In *RECOMB*, volume 10812 of *LNCS*, pages 211–224, 2018.

[28] Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k-mismatch average common substring problem. *Journal of Computational Biology*, 23(6):472–482, 2016.