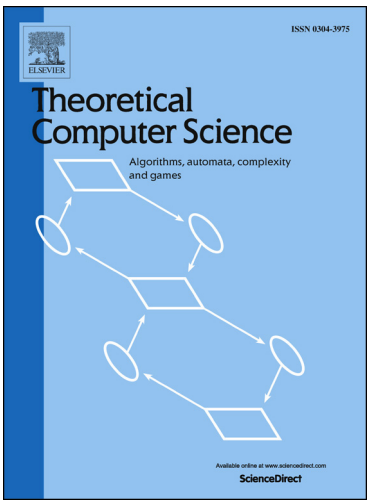
DOI:
10.1016/j.tcs.2018.10.033

Document Version
Peer reviewed version

Accepted Manuscript



Off-line and on-line algorithms for closed string factorization

Mai Alzamel, Costas S. Iliopoulos, W.F. Smyth, Wing-Kin Sung

PII: S0304-3975(18)30654-6
DOI: https://doi.org/10.1016/j.tcs.2018.10.033
Reference: TCS 11791

To appear in: *Theoretical Computer Science*

Received date: 31 December 2017
Revised date: 6 September 2018
Accepted date: 24 October 2018

Please cite this article in press as: M. Alzamel et al., Off-line and on-line algorithms for closed string factorization, *Theoret. Comput. Sci.* (2018), https://doi.org/10.1016/j.tcs.2018.10.033

Off-line and On-Line Algorithms for Closed String Factorization

Mai Alzamel[b,c], Costas S. Iliopoulos[b], W.F. Smyth[d,e], Wing-Kin Sung[a]

[a] *Department of Computer Science, National University of Singapore, Singapore, Singapore*
[b] *Department of Informatics, King's College London, London, UK*
[c] *Department of Computer Science, King Saud University ,Riyadh, SA*
[d] *Department of Computing & Software, McMaster University, Ontario, Canada*
[e] *School of Engineering & Information Technology, Murdoch University, Perth, Australia*

Abstract

A string $X = X[1..n]$, $n > 1$, is said to be closed if it has a nonempty proper prefix that is also a suffix, but that otherwise occurs nowhere else in X; for $n = 1$, every X is closed. Closed strings were introduced by Fici in [1] as objects of combinatorial interest. Recently Badkobeh *et al.* [2] described a variety of algorithms to factor a given string into closed factors. In particular, they studied the Longest Closed Factorization (LCF) problem, which greedily computes the decomposition $X = X_1 X_2 \cdots X_k$, where X_1 is the *longest closed prefix* of X, X_2 the *longest closed prefix* of X with prefix X_1 removed, and so on. In this paper we present an $O(\log n)$ amortized per character algorithm to compute LCF on-line, where n is the length of the string. We also introduce the Minimum Closed Factorization (MCF) problem, which identifies the minimum number of closed factors that cover X. We first describe an off-line $\mathcal{O}(n \log^2 n)$-time algorithm to compute $MCF(X)$, then we present an on-line algorithm for the same problem. In fact, we show that $MCF(X)$ can be computed in $O(L \log n)$ time from $MCF(X')$, where $X' = X[1..n-1]$, and L is the largest integer such that the suffix $X[n-L+1..n]$ is a substring of X'.

Keywords: closed string, on-line, minimum factorization

2010 MSC: 00-01, 99-00

Email addresses: mai.alzamel@kcl.ac.uk (Mai Alzamel), csi@kcl.ac.uk (Costas S. Iliopoulos), smyth@mcmaster.ca (W.F. Smyth), ksung@comp.nus.edu.sg (Wing-Kin Sung)

1. Introduction

Given a nonnegative integer n, a string $X = X[1..n]$ is a sequence of length $n = |X|$, containing letters $X[i]$, $1 \leq i \leq n$, drawn from an alphabet Σ of size $\sigma = |\Sigma|$, which we assume throughout to be constant. If $n = 0$, $X = \varepsilon$ is the empty string. $X = UVW$ is said to have a prefix U, a factor V and a suffix W; they are called proper if $|U| < |X|$, $|V| < |X|$, $|W| < |X|$, respectively. If B is both a proper prefix and a proper suffix of X, then B is said to be a border of X. X is closed if $n = 1$; for $n > 1$, X is closed if and only if its *longest* border occurs exactly twice in X. For example, $X = ababa$ is closed, because the border $B = aba$ occurs in X only as suffix and prefix. Note, however, that X also has a shorter border $B' = a$, which of course must be a border of B; but it occurs three times in X.

Closed strings were first studied by Fici *et al.* [1], the more practical relevance of closed strings was established via their relationship with palindromic strings. The number of closed factors in a string is minimised if these factors are also palindromic. Additionally it was shown that the upper bound on the number of palindromic factors of a string coincides with the lower bound on the number of closed factors (see [3] and references therein). Thus the study of closed strings shows potential applications in connection with applications of palindromes [4]. On the algorithmic side Badkobeh *et al.* in [2] presented (among others) an algorithm for the factorisation of a given string of length n into a sequence of longest closed factors (LCFs) in time and space $\mathcal{O}(n)$ and another algorithm for computing the longest closed factor starting at every position in the string in $\mathcal{O}(n\frac{\log n}{\log\log n})$ time and $\mathcal{O}(n)$ space. Moreover, Iliopoulos *et al.* [5] presented an on-line $\mathcal{O}(n)$-time algorithm to calculate the size of a minimum closed cover for each prefix of a given string X of length n. (A set of closed strings $W = \{w_1, \cdots, w_l\}$ is called a cover of a string X if X can be constructed by concatenations and overlaps of elements of W.)

In [6] an algorithm for the reverse engineering problem was introduced: given an LCF array that gives the longest closed factor of every prefix of some string, to reconstitute such a string. This algorithm makes use of Weiner's algorithm for right-to-left suffix tree construction: here we apply Ukkonen's suffix tree algorithm for

left-to-right on-line construction. It may be that Weiner's algorithm could also be applied for off-line computation of LCF(X).

Recently, Alamro *et al.* [7] produced an $\mathcal{O}(kn)$-time algorithm for testing whether a string is k-closed allowing Hamming distance errors bounded by the parameter k. The theoretical and practical relevance of closed strings was established via their relationship with palindromic strings. The number of closed factors in a string is minimised, if these factors are also palindromic as shown in [8]. Additionally it was shown that the upper bound on the number of palindromic factors of a string coincides with the lower bound on the number of closed factors [9]. Thus the study of closed strings shows potential applications in connection with applications of palindromes. In molecular biology, for instance, palindromic sequences are extensively studied: they are often distributed around promoters, introns, and untranslated regions, playing important roles in gene regulation and other cell processes (see e.g. [4] [10]).

A direct motivation comes from computational biology: Target Site Duplications (TSDs) are direct repeats that occur at insertion sites of transposable elements. They are thought to occur due to the filling in of the sticky ends (borders) derived from the staggered cut by transposes. They flank transposable elements and can be used to find their loci in the genome. Long Terminal Repeats (LTRs) are direct repeats which flank the transposed coding regions, and which themselves are flanked by TSDs [11][12].

This paper considers two variants of the decomposition of a given string X into closed factors:

- $LCF(X)$, the longest closed factorization of X, is a concatenation of k strings $X_k \cdots X_2 X_1$ such that X_1 is the *longest closed suffix* of X, X_2 is the *longest closed suffix* of X/X_1 (that is, X with suffix X_1 removed), X_3 the *longest closed suffix* of $(X/X_1)/X_2$, and so on. The *longest closed suffix* problem stated here is a variant of the *longest closed prefix* problem described in the abstract, which we have changed for technical reasons (see below). Denoting by X^R the reverse $X[n]X[n-1]\cdots X[1]$ of X, and denoting by LCF_p and LCF_s the *longest closed prefix* and the *longest closed suffix* versions, respectively, of LCF, it is not

difficult to show that

$$LCF_p(X) = X_1X_2\cdots X_k \iff LCF_s(X^R) = X_1^RX_2^R\cdots X_k^R.$$

Thus, for both the *longest closed prefix* and the *longest closed suffix* versions of the problem, we may define $\alpha(X) = k$, to be the number of LCF factors of a specified string X of length n. More formally, $\alpha(i) = \alpha(X[i..n])$ for $1 \leq i \leq n$. When there is no ambiguity, we may use $\alpha(n)$ instead. In [2] an off-line algorithm is described that computes $LCF_p(X)$ in $\mathcal{O}(n)$ time.

- $MCF(X)$, the minimum closed factorization of X, yields a factorization $X = X_1X_2\cdots X_k$, where each X_j, $1 \leq j \leq k$, is closed, and k is a minimum over all such factorizations. Accordingly we define $\gamma(X) = k$ (alternatively, $\gamma(n) = k$) giving the size of the MCF of $X[1..n]$. Currently no algorithm exists to compute γ.

To see that these two problems are indeed distinct, consider the string of length $n = 15$:

$$X = adabvwadvcvwbvw.$$

$LCF_p(X) = (adabvwad)(vcv)(wbvw)$, so that $\alpha(X) = 3$, while $MCF(X) = (ada)(bvwadvcvwbvw)$, so that $\gamma(X) = 2$.

In this paper, we begin with an on-line algorithm to compute $\alpha(X)$ using $LCF_s(X)$ (henceforth just $LCF(X)$). All our algorithms make use of properties of Ukkonen's on-line suffix tree construction algorithm [13], of which we provide critical properties in Section 2. Since suffix tree construction depends upon an ordering of the letters of the alphabet, we therefore assume throughout that Σ is a globally ordered set. In section 3, we describe the on-line algorithm for computing $\alpha(X)$. In section 4, we present an $\mathcal{O}(n\log^2 n)$-time algorithm to compute $\gamma(X)$, followed in Section 5 by an on-line algorithm for the same problem. An interesting result here is that $MCF(X)$ can be computed in $\mathcal{O}(L\log n)$ time from $MCF(X')$, $X' = X[1..n-1]$, where L is the largest integer such that the suffix $X[n-L+1..n]$ is a substring of X'. In section 6, we briefly discuss future work.

2. Properties of Ukkonen's On-Line Suffix Tree Algorithm

Our on-line algorithms make use of Ukkonen's on-line suffix tree construction algorithm UKK [13], whose properties are reviewed in this section. We also provide a bound on the worst-case time required by UKK to compute the suffix tree $T_{X[1..n]}$ from $T_{X[1..n-1]}$.

First, we give some definitions. Given $X = X[1..n]$, the implicit suffix tree T_X of X contains the paths corresponding to $X[j..n]$, for every $j \in 1..n$; while the explicit suffix tree $T_{X\$}$ is just the implicit suffix tree of $X\$$, where the terminal letter $\$$ is less than any other letter in X. For any node u in T_X, $plab_X(u)$ is the label of the path from the root to u, while $Leaf_X(u)$ is the set of leaves in the subtree of T_X rooted at u. For a leaf in T_X representing the suffix $X[i..n]$, the leaf is denoted as i.

The suffix link of a node v with path-label αy is a pointer to the node path-labelled y, where $\alpha \in \Sigma$ is a single letter and y is a string. The suffix link of v exists whenever v is a non-root internal node of T. The suffix links can be computed as follows. The first step is to mark each internal node v of the suffix tree with a pair of leaves (i, j), such that the lowest common ancestor of the two leaves i and j is v. This can be done by a DFS traversal of the tree.

A suffix $X[j..n]$ of $X[1..n]$ is said to be a quasiborder of $X[1..n]$ if $X[j..n]$ is a substring in $X[1..n-1]$. Let $\ell_n = n - j + 1$ if $X[j..n]$ is the longest quasiborder of X; and $\ell_n = 0$, if no quasiborder exists for X. Now let q_n denote the largest position in X such that $X[q_n..n]$ does not occur in $X[1..n-1]$; that is, $X[q_n + i..n]$, $i = 1, 2, \ldots, n - q_n$, *do* occur in X[1..n-1]. Note that $q_n = n - \ell_n$. For example, consider $X = ababacbcabab$. For $X[1..1]$, $q_1 = 1$ and $\ell_1 = 1 - q_1 = 0$. For $X[1..n]$ where $n = 2, \ldots, 5$, $q_n = 2$ and $\ell_n = n - q_n$. (1) gives ℓ_n and q_n for $n = 1, 2, \ldots, 12$.

$$
\begin{array}{lcccccccccccc}
n = & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
X = & a & b & a & b & a & c & b & c & a & b & a & b \\
\ell_n = & 0 & 0 & 1 & 2 & 3 & 0 & 1 & 1 & 1 & 2 & 3 & 4 \\
q_n = & 1 & 2 & 2 & 2 & 2 & 6 & 6 & 7 & 8 & 8 & 8 & 8
\end{array}
\tag{1}
$$

Lemma 1. *The implicit suffix tree for $X[1..n]$ is formed by suffixes $X[i..n]$, $i =$*

$1, 2, \ldots, q_n$.

Given $T_{X[1..n-1]}$, Ukkonen proposed an online algorithm to construct $T_{X[1..n]}$ from $T_{X[1..n-1]}$ by inserting suffixes $X[j..n]$ for $j = q_{n-1}, q_{n-1}+1, \ldots, q_n$. Using this methodology, Ukkonen established the following:

Lemma 2 (Ukkonen [13])**.** *Given $T_{X[1..n-1]}$, Ukkonen's algorithm uses $O(1)$ amortized time to build $T_{X[1..n]}$. As a byproduct, it computes the active point; that is, a node v in $T_{X[1..n]}$ such that $X[q_n + 1..n] = plab(v)$; otherwise, it is an edge (u, v) in $T_{X[1..n]}$ such that $plab(u)$ is a prefix of $X[q_n + 1..n]$ and $X[q_n + 1..n]$ is a prefix of $plab(v)$.*

Proof. Ukkonen's algorithm inserts suffixes $X[j..n], j = q_{n-1}+1, \ldots, q_n$ into $T_{X[1..n-1]}$, thus obtaining $T_{X[1..n]}$. In step j, a node w_j is inserted where w_j is the internal node in $T_{X[1..n]}$ which is the parent of the leaf representing $X[j..n]$. The suffix link speeds up these insertions, as we now describe.

We insert w_j into the tree as follows. First, from node w_{j-1}, we go up one edge to a node v, then follow the suffix link $sl(v)$ of v. Then, it takes us down a number of nodes to identify the edge to insert the node w_j. Ukkonen showed that amoritized $O(1)$ nodes will be traversed. □

The following lemma gives the worst case running time for this algorithm:

Lemma 3. *Given $T_{X[1..n-1]}$, Ukkonen's algorithm uses $O(n - q_{n-1})$ worst case time to build $T_{X[1..n]}$. As a byproduct, it computes the active point.*

Proof. Lemma 2 shows that Ukkonen's algorithm needs to traverse down d nodes the tree to find the edge to insert the node, where d is shown to be amortized $O(1)$. Since d must be short than length of the suffix $X[q_{n-1}..n]$, we know that $d \leq n - q_{n-1}$. The result follows. □

3. Computing $\alpha(X)$ On-Line

3.1. The Algorithm

Here we show how to compute $\alpha(X)$ for a given string $X[1..n]$, with $\alpha(0) = 0$ for $X = \varepsilon$. Recall that q_n is the largest position such that $X[q_n..n]$ does not appear in $X[1..n-1]$. Hence $X[q_n + 1..n]$ is the longest suffix that appears in $X[1..n-1]$.

When $q_n < n$, let e_n be the largest index smaller than $q_n + 1$ such that $X[e_n..e_n + (n - q_n) - 1] = X[q_n + 1..n]$; otherwise, when $q_n = n$, set $e_n = n$.

Lemma 4. *$X[e_n..n]$ is the longest suffix which is a closed factor.*

Proof. For $q_n = n$, $X[n]$ does not appear in $X[1..n-1]$. Hence, $X[n]$ is the longest suffix which is a closed factor.

For $q_n < n$, suppose on the contrary that there exists $a < e_n$ such that $X[a..n]$ is a closed factor. Hence, there exists ℓ such that $X[a..a + \ell] = X[n - \ell..n]$ while $X[j..j + \ell] \neq X[n - \ell..n]$ for $a < j < n - \ell$.

By definition, $n - \ell > q_n$. This means that $X[n - \ell..n] = X[n - \ell + e_n - q_n..e_n + n - q_n]$. Note that $a < e_n \leq e_n + (n - \ell - q_n)$. Hence, $X[a..n]$ is not a closed factor, a contradiction. □

Lemma 5. $\alpha(n) = \alpha(e_n - 1) + 1$

Proof. This lemma follows from Lemma 4. □

Given $X = X[1..n]$, recall that the implicit suffix tree T_X has q_n leaves. Denote $L_X[1..q_n]$ be the list of leaves in T_X ordered from left to right. Precisely, suppose the ith leaf from left to right of T_X is suffix $X[j..n]$, we set $L_X[i] = j$.

Based on the above lemmas, the algorithm ComputeAlpha given below correctly computes $\alpha(n)$. Our algorithm requires a range maximum data structure for finding range maximum. The data structure is described in Lemma 6.

Lemma 6. *We can maintain a list of integers such that the following operations take $O(\log n)$ time where n is the length of the list.*

(1) Insertion of an integer π in position i;

(2) Deletion of an integer from the list;

(3) Find the maximum of the integers in the range $i..j$;

(4) Find the entry in the list corresponding to position i; and

(5) Given an entry in the list, find its position in the list.

(6) Given an entry, find its predecessor or successor in the list.

Proof. We maintain the list of integers using a balanced search tree. Furthermore, each internal node of the tree specifies the size of the subtree and the maximum among all nodes in the subtree. Then, all six operations take $O(\log n)$ time. □

Theorem 1. *Given a string X of length n, the algorithm ComputeAlpha() computes the longest closed factorization in $O(\log n)$ amortized time per character.*

Proof. The algorithm referenced in Lemma 2 builds $T_{X[1..n]}$ from $T_{X[1..n-1]}$. Also, it computes the active point, which is either (1) a node v in $T_{X[1..n]}$ such that $plab(v) = X[q_n+1..n]$ or (2) an edge (u, v) in $T_{X[1..n]}$ such that $plab(u)$ is a prefix of $X[q_n+1..n]$ and $X[q_n+1..n]$ is a prefix of $plab(v)$. The running time is $O(\log \sigma) = O(1)$ amortized time.

During the construction of $T_{X[1..n]}$, new leaves are created. We insert the new leaves into the range maximum data structure using Lemma 6, which takes $O(\log n)$ time per each new leaf. Since we expected to have amortized $O(1)$ additional leaves, the data structure can be updated in $O(\log n)$ amortized time.

Let st and ed be the leftmost and rightmost leaves below v in $T_{X[1..n]}$. Set e_n be the range maximum among the leaves in $st..ed$, which can be found in $O(\log n)$ time (by Lemma 6). Then, $\alpha(n) = \alpha(e_n - 1) + 1$ by Lemma 5. The total running time is $O(\log n)$ amortized time. □

As a matter of fact, the algorithm ComputeAlpha shown in Algorithm 1 is a dynamic programming algorithm that computes $\alpha(i)$ iteratively for $i = 0, 1, \ldots, n$ using Lemma 5. By backtracking, we can retrieve an LCF for $X[1..n]$.

Algorithm 1 Algorithm for computing $\alpha(n)$.

Algorithm ComputeAlpha$(T_{X[1..n-1]}, X[n], \alpha[1..n-1])$

Require: the implicit suffix tree $T_{X[1..n-1]}$ for $X[1..n-1]$, the character $X[n]$, the range maximum data structure for $L_{X[1..n-1]}$ (see Lemma 6); the values $\alpha[i]$ for $1 \leq i \leq n-1$

Ensure: $\alpha[n]$ and the implicit suffix tree T_n

1: Given $T_{X[1..n-1]}$, using Ukkonen's algorithm, we build $T_{X[1..n]}$ and $L_{X[1..n]}$; Update the range maximum data structure if $L_{X[1..n]}$ contains new leaves.

2: Let (u, v) be the edge in $T_{X[1..n]}$ such that $plab(u)$ is a prefix of $X[q_n + 1..n]$ and $X[1_n + 1..n]$ is a prefix of $plab(v)$;

3: Let st and ed be the leftmost and rightmost leaves below v in $T_{X[1..n]}$; Set e_n be the range maximum among $L[st], \ldots, L[ed]$;

4: Report $\alpha(n) = \alpha(e_n - 1) + 1$;

3.2. Example

Consider the string $X = adabvwadvevwbvw$. Set $\alpha(0) = 0$. The following table shows an example run for the sequence X.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
q_n	1	2	2	4	5	6	6	6	8	10	10	10	12	12	12
e_n	1	2	1	4	5	6	3	1	5	10	9	5	4	4	4
$\alpha(n)$	1	2	1	2	3	4	3	1	3	4	2	3	2	2	2

4. Off-Line Computation of The Minimum Closed Factorisation $\gamma(X)$

For every position $i < j$ and an integer d, (i, j, d) is called a valid tuple if $X[i..j + d - 1]$ is a closed factor with a maximum border of length d that occurs exactly twice. (Recall, as noted in the Introduction, that any other borders of length less than d must occur at least three times in the closed factor.)

Let $T = T_{X\$}$ be the explicit suffix tree of X. Let $lca(i, j)$ be the lowest common ancestor of two leaves i and j in T. For any node u in T, let T^u be the subtree of T

rooted at u and $depth(u)$ be the length of the path label of u in T. Let $v(i,j)$ be the lowest common ancestor $lca(i,j)$ of T if $T^{v(i,j)}$ contains some leaf z where $i < z < j$; otherwise, let $v(i,j)$ be the root of T.

The next lemma gives a necessary and sufficient condition for (i,j,d) to be a valid tuple.

Lemma 7. *For $1 \leq i < j \leq n$, (i,j,d) is a valid tuple if and only if $depth(lca(i,j)) \geq d > depth(v(i,j))$.*

Proof. ($\rightarrow$) We show that if $d > depth(lca(i,j))$ or $d \leq depth(v(i,j))$, then (i,j,d) is not a valid tuple.

As i and j are in $T^{lca(i,j)}$, $X[i..i+d-1] \neq X[j..j+d-1]$ when $d > depth(lca(i,j))$. Hence, when $d > depth(lca(i,j))$, (i,j,d) is not a valid tuple.

By the definition of $v(i,j)$, there exists some leaf z in $T^{v(i,j)}$ such that $i < z < j$. Also, $v(i,j)$ is an ancestor of $lca(i,j)$. Hence, when $d \leq depth(v(i,j))$, $X[i..i+d-1] = X[z..z+d-1] = X[j..j+d-1]$. This implies that $X[j..j+d-1]$ occurs at least three times in $X[i..j+d-1]$; hence, (i,j,d) is not a valid tuple when $d \leq depth(v(i,j))$.

($\leftarrow$) Now, we consider the case where $depth(lca(i,j)) \geq d > depth(v(i,j))$. As i and j are in $T^{lca(i,j)}$, we have $X[i..i+d-1] = X[j..j+d-1]$. When $d > depth(v(i,j))$, As there is no leaf z in $T^{v(i,j)}$ such that $i < z < j$, we have $X[i..i+d-1] \neq X[z..z+d-1]$ for $d > depth(v(i,j))$. Hence, when $depth(lca(i,j)) \geq d > depth(v(i,j))$, (i,j,d) is a valid tuple. □

Lemma 8. *For any $i < j$, (i,j,d) is a valid tuple for some integer d if and only if $T^{lca(i,j)}$ does not contain any leaf z such that $i < z < j$.*

Proof. By Lemma 7, (i,j,d) is valid if and only if $depth(lca(i,j)) > depth(v(i,j))$. This means that $v(i,j) \neq lca(i,j)$. In other words, $T^{lca(i,j)}$ does not contain any leaf z such that $i < z < j$. □

For every node u in T, denote by $hvy(u)$ the child of u with the most number of leaves — called the "heavy" child. Denote $I_u = \{(i,j) \mid lca(i,j) = u$ and (i,j,d) is a valid tuple for some $d\}$. Let $I_T = \bigcup_{u \in T} I_u$. The following lemma states the size of I_u.

Corollary 9. $|I_u| \leq |T^u| - |T^{hvy(u)}|$.

Proof. By Lemma 8, for every $(i, j) \in I_u$, we have $lca(i, j) = u$ and there is no integer z in the $T^{lca(i,j)}$ such that $i < z < j$. Hence, i and j must be adjacent in the sorted list of the leaves in T^u. Since $lca(i, j) = u$, i and j must be in different subtrees attached to u. So, for each $(i, j) \in I_u$, we cannot have both i and j in $T^{hvy(u)}$. Hence, the size of I_u is bounded above by the number of leaves in $T^u - T^{hvy(u)}$. The lemma follows. □

There is a known fact related to heavy children introduced in [14]:

$$\sum_{u \in T} (|T^u| - |T^{hvy(u)}|) = O(n \log n). \tag{2}$$

Thus:

Lemma 10. $|I_T| = \sum_{u \in T} |I_u| = O(n \log n)$.

Proof. An immediate consequence of Lemma 9 and (2). □

Lemma 11. *We can compute I_T in $O(n \log^2 n)$ time.*

Proof. We compute I_u for all $u \in T$ in bottom-up order. Before we process u, we maintain the invariant that, for every child v of u, the balanced binary search tree data structure B_v for all leaves in T^v is available (see Lemma 6). Then, we create B_u by inserting all leaves in $T^u - T^{hvy(u)}$ into $B_{hvy(u)}$, which requires $O((|T^u| - |T^{hvy(u)}|) \log n)$ time. Using B_u, for every leaf z in $T^u - T^{hvy(u)}$, we can compute the predecessor z' and the successor z'' of z in B_u in $O(\log n)$ time (see Lemma 6). We insert (z', z) into I_u if $lca(z', z) = u$, and (z, z'') into I_u if $lca(z, z'') = u$. Hence, the processing time for the node u is $O((|T^u| - |T^{hvy(u)}|) \log n)$.

Using (2), the total running time is $O(n \log^2 n)$. □

For every $(i, j) \in I_T$, recall that $v(i, j)$ is the lowest common ancestor $lca(i, j)$ such that $T^{v(i,j)}$ contains some leaf z where $i < z < j$. The next result shows how to compute $v(i, j)$ for $(i, j) \in I_T$.

Lemma 12. *For any $(i, j) \in I_T$, let $prev_i(j) = \max\{j' \mid (i, j') \in I_T, j' < j\}$. Then $v(i, j) = lca(i, prev_i(j))$ if $prev_i(j)$ exists; otherwise, $v(i, j)$ is the root of T.*

Proof. Let $j' = prev_i(j)$, i.e., j' is the largest integer in $\{j' \mid (i,j') \in I_T\}$ such that $i < j' < j$. We claim that $v(i,j) = lca(i,j')$. Suppose on the contrary that $v(i,j)$ is either a descendant or an ancestor of $lca(i,j')$. But $v(i,j)$ cannot be an ancestor of $lca(i,j')$, since$v(i,j)$ is not the lowest ancestor of $lca(i,j)$ such that $T^{v(i,j)}$ contains some leaf z where $i < z < j$. If $v(i,j)$ is a descendant of $lca(i,j')$, this means that there exists a leaf z where $i < z < j$ and $lca(i,z) = v(i,j)$. Let z be the smallest such integer. Then, $(i, z, depth(lca(i,z))$ is a valid tuple, which contradicts the fact that $(i,z) \notin I_T$. Hence $v(i,j) = lca(i,j')$. □

Lemma 13. *$v(i,j)$ for all $(i,j) \in I_T$ can be computed in $O(n \log^2 n)$ time.*

Proof. For every i, we can sort all integers in $\{j' \mid (i,j') \in I_T\}$. Since I_T is of size $O(n \log n)$ (see Lemma 10), the sort takes $O(n \log^2 n)$ time. Then, for every $(i,j) \in I_T$, $lca(i, prev_i(j))$ can be computed in constant time. The lemma follows. □

Lemma 14.

$$\gamma(X[i..n]) = 1 + \min\left\{\gamma(X[i+1..n]), \min_{j:(i,j)\in I_T} \min_{k=j+depth(v(i,j))}^{j+depth(lca(i,j))-1} \gamma(X[k+1..n])\right\}$$

.

Proof. $\gamma(X[i..n]) = 1 + \min_{k=i}^{n}\{\gamma(X[k+1..n]) \mid X[i..k]\}$ is by definition a closed factor.

$X[i..k]$ is a closed factor if $k = i$ or there exists a valid tuple (i,j,d) such that $k = j + d - 1$. By Lemma 7 for every $(i,j) \in I_T$, (i,j,d) is a valid tuple for $depth(v(i,j)) < d \leq depth(lca(i,j))$. Hence, $\{k \mid X[i..k]$ is a closed factor$\} = \{i\} \cup \{k \mid j + depth(v(i,j)) - 1 < k \leq j + depth(lca(i,j)) - 1\}$. The lemma follows. □

Based on this result, Algorithm 2 gives a dynamic programming algorithm to compute $\gamma(X[1..n])$. Thus:

Theorem 2. *Given a string X of length n the minimum closed factorisation for string X can be computed off-line in $O(n \log^2 n)$ time by using algorithm ComputeGammaOffline.*

Proof. By Lemma 11, I_u for all u in T can be computed in $O(n \log^2 n)$ time. By Lemma 13, $v(i,j)$ for all $(i,j) \in I_T$ can be found in $O(n \log^2 n)$ time.

To compute $\min_{k=x}^{y} \gamma(X[k..n])$ for $i < x \leq y \leq n$, we maintain a balanced search tree of all values in $\gamma(X[j..n])$ for $i < j \leq n$. Then, $\min_{k=x}^{y} \gamma(X[k..n])$ can be found in $O(\log n)$ time. Using the recursive formula in Lemma 14, $\gamma(X[i..n])$ can be found in $O(|\{j : (i,j) \in I_T\}| \log n)$ time.

To process $\gamma(X[i..n])$ for all i, the time complexity is $O(|I_T| \log n) = O(n \log^2 n)$.

□

Algorithm 2 The offline algorithm for computing $\gamma(n)$.

Algorithm ComputeGammaOffline$(X[1..n])$

Require: the string $X[1..n]$

Ensure: $\gamma(X[1..n])$

1: Construct the suffix tree T for $X[1..n]$. Also, construct the lca data structure.
2: Compute I_u for all u in T using Lemma 11.
3: For every $(i,j) \in \bigcup_u I_u$, compute $v(i,j)$ using Lemma 12.
4: Set $\gamma(X[n..n]) = 1$ and $\gamma(X[n+1..n]) = 0$;
5: Construct the balanced search tree B with one element $\gamma(X[n])$.
6: **for** $i = n-1$ downto 1 **do**
7: Compute $\gamma(X[i..n])$ using Lemma 14
8: Insert $\gamma(X[i..n])$ into the balanced search tree B
9: **end for**
10: Report $\gamma(X[1..n])$

5. On-Line Computation of the Minimum Closed Factorisation $\gamma(X)$

5.1. The Algorithm

Let $\gamma(0) = 0$ and $\gamma(n) = \gamma(X[1..n])$. Let $b_n(q) = p$ if p is the maximum index smaller than q such that $X[q..n] = X[p..p+n-q]$; otherwise, $b_n(q) = n$.

Lemma 15. *$X[j..n]$ is a closed factor if and only if $j = n$ or $j = b_n(q)$ for some $q \in \{q_n + 1, \ldots, n\}$.*

Proof. ($\rightarrow$) If $X[j..n]$ is a closed factor, there are two cases: (1) $j = n$ or (2) $j < n$. For (2), $X[j..n]$ is a closed factor implies $X[j..j+n-q] = X[q_n\dot{n}]$ for some $q \in \{j+1, \ldots, n\}$ such that $X[q..n]$ is not a length-$(n-q+1)$ substring appear at positions p for $j < p < q$. This implies that $b_n(q) = j$.

($\leftarrow$) If $j = n$, $X[j..n]$ is a closed factor by definition.

Suppose $j = b_n(q)$ for some $j < q \leq n$. This means that $X[j..j+n-q] = X[q..n]$ and $X[p..p+n-q] = X[q..n]$ for $j < p < q$. This implies that $X[j..n]$ is a closed factor. □

Recall that q_n is the smallest index such that $X[q_n..n]$ does not appear in $X[1..n-1]$.

Lemma 16. *Every border b of a closed factor $X[j..n]$ satisfies $|b| \leq n - q_n$.*

Proof. By definition, q_n is the biggest index such that $X[q_n..n]$ does not appears in $T_{[1..n-1]}$. This means that $X[q..n]$ is not a substring in $X[1..n-1]$ for $q \leq q_n$. Hence, the lemma follows. □

Lemma 17. $\gamma(n) = 1 + \min\{\gamma(n-1), \min_{q=q_n+1}^{n} \gamma(b_n(q)-1)\}$.

Proof. By definition, we have: $\gamma(n) = \min_{j=1}^{n}\{\gamma(j-1)+1 \mid X[j..n]$ is a closed factor $\}$.

By Lemmas 15 and 16, $X[j..n]$ is a closed factor if $j = n$ or $j = b_n(q)$ where $n - q + 1 \leq n - q_n$. This means that $X[j..n]$ is a closed factor if $j = n$ or $j = b_n(q)$ where $q \geq q_n + 1$. We have $\gamma(n) = \min\{\gamma(n-1)+1, \min_{q=q_n+1}^{n}\{\gamma(b_n(q)-1)+1\}\}$. The lemma follows. □

To compute $\gamma(n)$ using Lemma 17, we need to compute $b_n(q)$ for $q_n < q \leq n$. Next we describe how to do this using a suffix tree and Ukkonen's algorithm.

Lemma 18. *For $q \in \{q_n + 1, \ldots, n\}$, let v_q be the node in $T_{X[1..n]\$}$ such that $plab(v_q) = X[q..n]$. We have $b_n(q) = j$, where j is the largest leaf index smaller than q in $Leaf_{X[1..n]}(v_q)$.*

Here we maintain a range maximum data structure for all leaves in the suffix tree of T_X (which is actually the suffix array of X). Note that this data structure allows insert, delete and range maximum query in $O(\log n)$ time.

Based on the above lemma and Lemma 17, we obtain Algorithm 3 to compute $\gamma(n)$.

Algorithm 3 Algorithm for computing $\gamma(n)$.

Algorithm ComputeGamma$(T_{X[1..n-1]}, X[n], \gamma[1..n-1])$

Require: the implicit suffix tree $T_{X[1..n-1]}$ for $X[1..n-1]$, range maximum data structure for the leaves in $T_{X[1..n-1]}$, the character $X[n]$, the values $\gamma[i]$ for $1 \leq i \leq n-1$

Ensure: $\gamma[n]$ and the implicit suffix tree T_n

1: Given $T_{X[1..n-1]}$, using Ukkonen's algorithm, we build $T_{X[1..n]}$; update the range maximum data structure if there are new leaves.
2: Given $T_{X[1..n]}$, using Ukkonen's algorithm, we build $T_{X[1..n]\$}$; update the range maximum data structure if there are new leaves.
3: $\gamma_0 = \infty$;
4: Let u be the node in $T_{X[1..n]\$}$ with $plab(u) = X[q_n + 1..n]$;
5: **for** $q = q_n + 1$ to n **do**
6: Let st and ed be the leftmost and rightmost leaves below u in $T_{X[1..n]\$}$; Set $b_n(q)$ to be the range maximum among the leaves in $st..ed$;
7: $\gamma_0 = \min\{\gamma_0, \gamma(b_n(q) - 1) + 1\}$;
8: Set u to be the suffix link of u;
9: **end for**
10: Back-track and revert the suffix tree from $T_{X[1..n]\$}$ back to $T_{X[1..n]}$;
11: Report $\gamma(n) = \gamma_0$;

Theorem 3. *Given the implicit suffix tree for $T_{X[1..n-1]}$, the character $X[n]$ and the values $\gamma[i]$ for $1 \leq i \leq n-1$, $\gamma(n)$, the length of the minimum closed factorization of $X[1..n]$, can be computed in $O(L \log n)$ time, where L is the largest value such that $X[n-L..n-1]$ appears in $X[1..n-2]$.*

Proof. By definition of q_{n-1}, we have $L = n - 1 - q_{n-1}$.

By Lemma 3, we can build $T_{X[1..n]}$ from $T_{X[1..n-1]}$ using $O(n-1-q_{n-1}) = O(L)$ time. We may insert at most $O(n-1-q_{n-1}) = O(L)$ leaves. It takes at most $O(L \log n)$ time to maintain the range maximum data structure.

Then, we compute $b_n(q)$ for $q = q_n + 1, \ldots, n$. In each step, we compute suffix link and range maximum, which requires $O(\log n)$ time. Hence, $\gamma(n)$ can be computed in $O(L \log n)$ time. □

5.2. Example

Consider the string $X = adabvwadvevwbvw$. Initially, $\gamma(0) = 0$. The following table shows an example run for the sequence X. where $\gamma(n) = 1 + \min\{\gamma(n-1), \min_{q=q_n+1}^{n} \gamma(b_n(q) - 1)\}$

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
q_n	1	2	2	4	5	6	6	6	8	10	10	10	12	12	12
$b_n(q_n+1)$			1				3	1	5		9	5	4	4	4
$b_n(q_n+2)$								2				6		11	11
$b_n(q_n+3)$															12
$\gamma(n)$	1	2	1	2	3	4	3	1	3	4	2	3	2	2	2

6. Conclusion

The computations of $\alpha(X)$ and $\gamma(X)$ described in this paper, as well as that of $\alpha(X)$ given in [2], depend upon suffix tree construction, hence on an ordered alphabet. However, there is nothing in the statements of these problems that requires such an ordering. Can efficient algorithms for α and γ be found that do not require an ordering of Σ?

Acknowledgements

The authors are grateful to two anonymous referees whose comments and suggestions have greatly improved this paper.

7. References

[1] G. Fici, A classification of trapezoidal words, in: Proceedings 8th International Conference Words 2011, Prague, Vol. 63 of Electronic Proceedings in Theoretical Computer Science, 2011, pp. 129–137.

[2] G. Badkobeh, H. Bannai, K. Goto, T. I, C. S. Iliopoulos, S. Inenaga, S. J. Puglisi, S. Sugimoto, Closed factorization, Discrete Applied Mathematics 212 (2016) 23–29, stringology Algorithms.

[3] G. Badkobeh, G. Fici, Z. Lipták, On the number of closed factors in a word, in: Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings, 2015, pp. 381–390.

[4] M. Adamczyk, M. Alzamel, P. Charalampopoulos, C. S. Iliopoulos, J. Radoszewski, Palindromic Decompositions with Gaps and Errors, Springer International Publishing, Cham, 2017, pp. 48–61.

[5] C. S. Iliopoulos, M. Mohamed, On-line Minimum Closed Covers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 106–115.

[6] H. Bannai, S. Inenaga, T. Kociumaka, A. Lefebvre, J. Radoszewski, W. Rytter, S. Sugimoto, T. Walen, Efficient algorithms for longest closed factor array, in: Proc. Symp. on String Processing and Information Retrieval (SPIRE 2015), 2015, pp. 95–102.

[7] H. Alamro, M. Alzamel, C. S. Iliopoulos, S. P. Pissis, S. Watts, W.-K. Sung, Efficient identification of k-closed strings, in: G. Boracchi, L. Iliadis, C. Jayne, A. Likas (Eds.), Engineering Applications of Neural Networks: 18th International Conference, EANN 2017, Athens, Greece, August 25–27, 2017, Proceedings, Springer International Publishing, Cham, 2017, pp. 583–595.

[8] M. Bucci, A. de Luca, A. De Luca, Rich and Periodic-Like Words, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 145–155.

[9] G. Badkobeh, G. Fici, Z. Lipták, A note on words with the smallest number of closed factors (2013), CoRR abs/1305.6395.

[10] B. M. Muhire, M. Golden, B. Murrell, P. Lefeuvre, J.-M. Lett, A. Gray, A. Y. Poon, N. K. Ngandu, Y. Semegni, E. P. Tanov, et al., Evidence of pervasive biologically functional secondary structures within the genomes of eukaryotic single-stranded DNA viruses, Journal of virology 88 (4) (2014) 1972–1989.

[11] R. S. Linheiro, C. M. Bergman, Whole genome resequencing reveals natural target site preferences of transposable elements in drosophila melanogaster, PLOS ONE 7 (2) (2012) 1–12.

[12] E. A. Gladyshev, I. R. Arkhipova, Rotifer rdna-specific r9 retrotransposable elements generate an exceptionally long target site duplication upon insertion, Gene 448 (2) (2009) 145 – 150, genomic Impact of Eukaryotic Transposable Elements.

[13] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.

[14] R. Cole, L. Gottlieb, M. Lewenstein, Dictionary matching and indexing with errors and don't cares, in: Proceedings of the 36th Annual ACM Symposium on Theory of Computing, 2004.