

# Computing Reversed Lempel-Ziv Factorization Online

Shiho Sugimoto<sup>1</sup>, Tomohiro I<sup>1,2</sup>, Shunsuke Inenaga<sup>1</sup>, Hideo Bannai<sup>1</sup>, and Masayuki Takeda<sup>1</sup>

<sup>1</sup> Department of Informatics, Kyushu University, Japan

{shiho.sugimoto,tomohiro.i,inenaga,bannai,takeda}@inf.kyushu-u.ac.jp

<sup>2</sup> Japan Society for the Promotion of Science (JSPS)

**Abstract.** Kolpakov and Kucherov proposed a variant of the Lempel-Ziv factorization, called the reversed Lempel-Ziv (RLZ) factorization (Theoretical Computer Science, 410(51):5365–5373, 2009). In this paper, we present an on-line algorithm that computes the RLZ factorization of a given string  $w$  of length  $n$  in  $O(n \log^2 n)$  time using  $O(n \log \sigma)$  bits of space, where  $\sigma \leq n$  is the alphabet size. Also, we introduce a new variant of the RLZ factorization with self-references, and present two on-line algorithms to compute this variant, in  $O(n \log \sigma)$  time using  $O(n \log n)$  bits of space, and in  $O(n \log^2 n)$  time using  $O(n \log \sigma)$  bits of space.

**Keywords:** reversed Lempel-Ziv factorization, on-line algorithms, suffix trees, palindromes

## 1 Introduction

The Lempel-Ziv (LZ) factorization of a string [21] is an important tool of data compression, and is a basis of efficient string processing algorithms [9,4] and compressed full text indices [11]. In the off-line setting where the string is static, there exist efficient algorithms to compute the LZ factorization of a given string  $w$  of length  $n$ , running in  $O(n)$  time and using  $O(n \log n)$  bits of space, assuming an integer alphabet. See [1] for a survey, and [8,5,7,6] for more recent results in this line of research. In the on-line setting where new characters may be appended to the end of the string, Okanohara and Sadakane [16] gave an algorithm that runs in  $O(n \log^3 n)$  time using  $n \log \sigma + o(n \log \sigma) + O(n)$  bits of space, where  $\sigma$  is the size of the alphabet. Later, Starikovskaya [18] proposed an algorithm running in  $O(n \log^2 n)$  time using  $O(n \log \sigma)$  bits of space, assuming  $\frac{\log_\sigma N}{4}$  characters are packed in a machine word. Very recently, Yamamoto et al. [20] developed a new on-line LZ factorization algorithm running in  $O(n \log n)$  time using  $O(n \log \sigma)$  bits of space.

In this paper, we consider the *reversed* Lempel-Ziv factorization (RLZ in short<sup>1</sup>) proposed by Kolpakov and Kucherov [10], which is used as a basis of computing gapped palindromes. In the on-line setting, the RLZ factorization can be computed in  $O(n \log \sigma)$  time using  $O(n \log n)$  bits of space, utilizing the algorithm by Blumer et al. [3]. We present a more space-efficient solution to the same problem, which requires only  $O(n \log \sigma)$  bits of working space with slightly slower  $O(n \log^2 n)$  running time.

We also introduce a new, self-referencing variant of the RLZ factorization, and propose two on-line algorithms; the first one runs in  $O(n \log \sigma)$  time and  $O(n \log n)$  bits of space, and the second one in  $O(n \log^2 n)$  time and  $O(n \log \sigma)$  bits of space. A

<sup>1</sup> Not to be confused with the *relative* Lempel-Ziv factorization proposed in [12].

key to achieve such complexity is efficient on-line computation of the longest suffix palindrome for each prefix of the string  $w$ .

As an independent interest, we consider the relationship between the number of factors in the RLZ factorization of a string  $w$ , and the size of the smallest grammar that generates only  $w$ . It is known that the number of factors in the LZ factorization of  $w$  is a lower bound of the smallest grammar for  $w$  [17]. We show that, unfortunately, this is not the case with the RLZ factorization with or without self-references.

## 2 Preliminaries

### 2.1 Strings and model of computation

Let  $\Sigma$  be the alphabet of size  $\sigma$ . An element of  $\Sigma^*$  is called a string. For string  $w = xyz$ ,  $x$  is called a prefix,  $y$  is called a substring, and  $z$  is called a suffix of  $w$ , respectively. The sets of substrings and suffixes of  $w$  are denoted by  $Substr(w)$  and  $Suffix(w)$ , respectively. The length of string  $w$  is denoted by  $|w|$ . The empty string  $\varepsilon$  is a string of length 0, that is,  $|\varepsilon| = 0$ . For  $1 \leq i \leq |w|$ ,  $w[i]$  denotes the  $i$ -th character of  $w$ . For  $1 \leq i \leq j \leq |w|$ ,  $w[i..j]$  denotes the substring of  $w$  that begins at position  $i$  and ends at position  $j$ . Let  $w^{\text{rev}}$  denote the reversed string of  $s$ , that is,  $w^{\text{rev}} = w[|w|] \cdots w[2]w[1]$ . For any  $1 \leq i \leq j \leq |w|$ , note  $w[i..j]^{\text{rev}} = w[j]w[j-1] \cdots w[i]$ .

A string  $x$  is called a palindrome if  $x = x^{\text{rev}}$ . The *center* of a palindromic substring  $w[i..j]$  of a string  $w$  is  $\frac{i+j}{2}$ . A palindromic substring  $w[i..j]$  is called the *maximal palindrome* at the center  $\frac{i+j}{2}$  if no other palindromes at the center  $\frac{i+j}{2}$  have a larger radius than  $w[i..j]$ , i.e., if  $w[i-1] \neq w[j+1]$ ,  $i = 1$ , or  $j = |w|$ . In particular, a maximal palindrome  $w[i..|w|]$  is called a *suffix palindrome* of  $w$ .

The default base of logarithms will be 2. Our model of computation is the unit cost word RAM with the machine word size at least  $\lceil \log n \rceil$  bits. We will evaluate the space complexities in bits (not in words). For an input string  $w$  of length  $n$  over an alphabet of size  $\sigma \leq n$ , let  $r = \frac{\log_{\sigma} n}{4} = \frac{\log n}{4 \log \sigma}$ . For simplicity, assume that  $\log n$  is divisible by  $4 \log \sigma$ , and that  $n$  is divisible by  $r$ . A string of length  $r$ , called a *meta-character*, fits in a single machine word. Thus, a meta-character can also be transparently regarded as an element in the integer alphabet  $\Sigma^r = \{1, \dots, n\}$ . We assume that given  $1 \leq i \leq n - r + 1$ , any meta-character  $A = w[i..i+r-1]$  can be retrieved in constant time. We call a string on the alphabet  $\Sigma^r$  of meta-characters, a *meta-string*. Any string  $w$  whose length is divisible by  $r$  can be viewed as a meta-string  $w$  of length  $m = \frac{n}{r}$ . We write  $\langle w \rangle$  when we explicitly view string  $w$  as a meta-string, where  $\langle w \rangle[j] = w[(j-1)r+1..jr]$  for each  $j \in [1, m]$ . Such range  $[(j-1)r+1, jr]$  of positions will be called *meta-blocks* and the beginning positions  $(j-1)r+1$  of meta-blocks will be called *block borders*. For clarity, the length  $m$  of a meta-string  $\langle w \rangle$  will be denoted by  $\|\langle w \rangle\|$ . Note that  $m \log n = n \log \sigma$ .

### 2.2 Suffix Trees and Generalized Suffix Tries

The suffix tree [19] of string  $s$ , denoted  $STree(s)$ , is a rooted tree such that

1. Each edge is labeled with a non-empty substring of  $s$ , and each path from the root to a node spells out a substring of  $s$ ;
2. Each internal node  $v$  has at least two children, and the labels of distinct out-going edges of  $v$  begin with distinct characters;

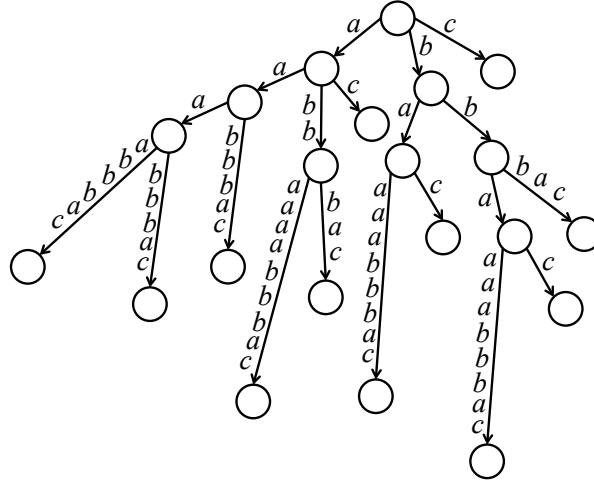


Figure 1.  $STree(w)$  with  $w = abbaaaabbbac$ .

3. For each suffix  $x$  of  $w$ , there is a path from the root that spells out  $x$ .

The number of nodes and edges of  $STree(s)$  is  $O(|s|)$ , and  $STree(s)$  can be represented using  $O(|s| \log |s|)$  bits of space, by implementing each edge label  $y$  as a pair  $(i, j)$  such that  $y = s[i..j]$ .

For a constant alphabet, Weiner’s algorithm [19] constructs  $STree(s^{rev})$  in an on-line manner from left to right, i.e., constructs  $STree(s[1..j]^{rev})$  in increasing order of  $j = 1, 2, \dots, |s|$ , in  $O(|s|)$  time using  $O(|s| \log |s|)$  bits of space. It is known that the tree of the suffix links of the directed acyclic word graph [3] of  $s$  forms  $STree(s^{rev})$ . Hence, for larger alphabets, we have the following:

**Lemma 1 ([3]).** *Given a string  $s$ , we can compute  $STree(s^{rev})$  on-line from left to right, in  $O(|s| \log \sigma)$  time using  $O(|s| \log |s|)$  bits of space.*

In our algorithms, we will also use the generalized suffix *trie* for a set  $W$  of strings, denoted  $STrie(W)$ .  $STrie(W)$  is a rooted tree such that

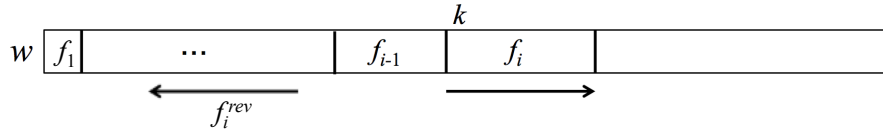
1. Each edge is labeled with a character, and each path from the root to a node spells out a substring of some string  $w \in W$ ;
2. The labels of distinct out-going edges of each node must be different;
3. For each suffix  $s$  of each string  $w \in W$ , there is a path from the root that spells out  $s$ .

### 2.3 Reversed LZ factorization

Kolpakov and Kucherov [10] introduced the following variant of LZ77 factorization.

**Definition 2 (Reversed LZ factorization without self-references).** *The reversed LZ factorization of string  $w$  without self-references, denoted  $RLZ(w)$ , is a sequence  $(f_1, f_2, \dots, f_m)$  of non-empty substrings of  $w$  such that*

1.  $w = f_1 \cdot f_2 \cdots f_m$ , and
2. For any  $1 \leq i \leq m$ ,  $f_i = w[k..k + \ell_{\max} - 1]$ , where  $k = |f_1 \cdots f_{i-1}| + 1$  and  $\ell_{\max} = \max(\{\ell \mid 1 \leq \exists t < k - \ell + 1, (w[t..t + \ell - 1])^{rev} = w[k..k + \ell - 1]\} \cup \{1\})$ .



**Figure 2.** Let  $k = |f_1 \cdots f_{i-1}| + 1$ .  $f_i$  is the longest non-empty prefix of  $w[k..n]$  that is also a substring of  $(w[1..k-1])^{\text{rev}}$  if such exists.

Assume we have  $f_1, \dots, f_{i-1}$ , and let  $k = |f_1 \cdots f_{i-1}| + 1$ . The above definition implies that  $f_i$  is the longest non-empty prefix of  $w[k..n]$  that is also a substring of  $(w[1..k-1])^{\text{rev}}$  if such exists, and  $f_i = w[k]$  otherwise. See also Figure 2.

*Example 3.* For string  $w = abbaaaabbbac$ ,  $RLZ(w)$  consists of the following factors:  $f_1 = a$ ,  $f_2 = b$ ,  $f_3 = ba$ ,  $f_4 = a$ ,  $f_5 = aabb$ ,  $f_6 = ba$ , and  $f_7 = c$ .

We are interested in on-line computation of  $RLZ(w)$ . Using Lemma 1, one can compute  $RLZ(w)$  on-line in  $O(n \log \sigma)$  time using  $O(n \log n)$  bits of space [10], where  $n = |w|$ . The idea is as follows: Assume we have already computed the first  $j$  factors  $f_1, f_2, \dots, f_j$ , and we have constructed  $STree(w[1..l_j]^{\text{rev}})$ , where  $l_j = \sum_{h=1}^j |f_h|$ . Now the next factor  $f_{j+1}$  is the longest prefix of  $w[l_j + 1..n]$  that is represented by a path from the root of  $STree(w[1..l_j]^{\text{rev}})$ . After the computation of  $f_{j+1}$ , we update  $STree(w[1..l_j]^{\text{rev}})$  to  $STree(w[1..l_{j+1}]^{\text{rev}})$ , using Lemma 1. In the next section, we will propose a new space-efficient on-line algorithm which requires  $O(n \log^2 n)$  time using  $O(n \log \sigma)$  bits of space.

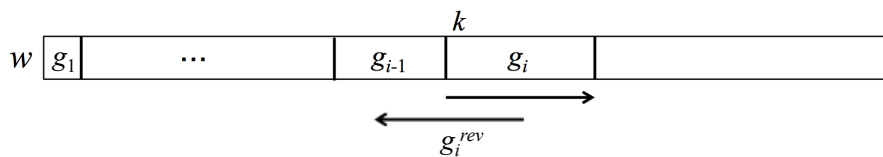
We introduce yet another new variant, the reversed LZ factorization *with self-references*.

**Definition 4 (Reversed LZ factorization with self-references).** *The reversed LZ factorization of string  $w$  with self-references, denoted  $RLZS(w)$ , is a sequence  $(g_1, g_2, \dots, g_p)$  of non-empty substrings of  $w$  such that*

1.  $w = g_1 \cdot g_2 \cdots g_p$ , and
2. For any  $1 \leq i \leq p$ ,  $g_i = w[k..k + \ell_{\max} - 1]$ , where  $k = |g_1 \cdots g_{i-1}| + 1$  and  $\ell_{\max} = \max(\{\ell \mid 1 \leq \exists r < k, (w[r..r + \ell - 1])^{\text{rev}} = w[k..k + \ell - 1]\} \cup \{1\})$ .

Since  $r$  is at most  $k - 1$  in the above definition,  $g_i$  is the longest non-empty prefix of  $w[k..n]$  that is also a substring of  $(w[1..k + |g_i| - 2])^{\text{rev}}$  if such exists, and  $g_i = w[k]$  otherwise. See also Figure 3.

*Example 5.* For string  $w = abbaaaabbbac$ ,  $RLZS(w)$  consists of the following factors:  $g_1 = a$ ,  $g_2 = b$ ,  $g_3 = baaaabb$ ,  $g_4 = ba$ , and  $g_5 = c$ .



**Figure 3.** Let  $k = |g_1 \cdots g_{i-1}| + 1$ .  $g_i$  is the longest prefix of  $w[k..n]$  that is also a substring of  $(w[1..k + |g_i| - 2])^{\text{rev}}$  if such exists.

Note that in Definition 4 the ending position of a previous occurrence of  $g_i^{\text{rev}}$  does not have to be prior to the beginning position  $k$  of  $g_i$ , while in Definition 2 it has to, because of the constraints “ $t < k - \ell + 1$ ”. This is the difference between  $RLZ(w)$  and  $RLZS(w)$ .

In this paper we propose two on-line algorithms to compute  $RLZS(w)$ ; the first one runs in  $O(n \log \sigma)$  time using  $O(n \log n)$  bits of space, and the second one does in  $O(n \log^2 n)$  time using  $O(n \log \sigma)$  bits of space.

### 3 Computing $RLZ(w)$ in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space

The outline of our on-line algorithm to compute  $RLZ(w)$  follows the algorithm of Starikovskaya [18] which computes Lempel-Ziv 77 factorization [21] in an on-line manner and in  $O(n \log^2 n)$  time using  $O(n \log \sigma)$  bits of space. The Starikovskaya algorithm maintains the suffix tree of the meta-string  $\langle w \rangle$  in an on-line manner, i.e., maintains  $STree(\langle w \rangle[1..k])$  in increasing order of  $k = 1, 2, \dots, n/r$ , and maintains a generalized suffix trie for a set of substrings of  $w[1..kr]$  of length  $2r$  that begin at a block border. In contrast to the Starikovskaya algorithm, our algorithm maintains  $STree(\langle w \rangle[1..k]^{\text{rev}})$  in increasing order of  $k = 1, 2, \dots, n/r$ , and maintain a generalized suffix trie for a set of substrings of  $w[1..kr]^{\text{rev}}$  of length  $2r$  that begin at a block border.

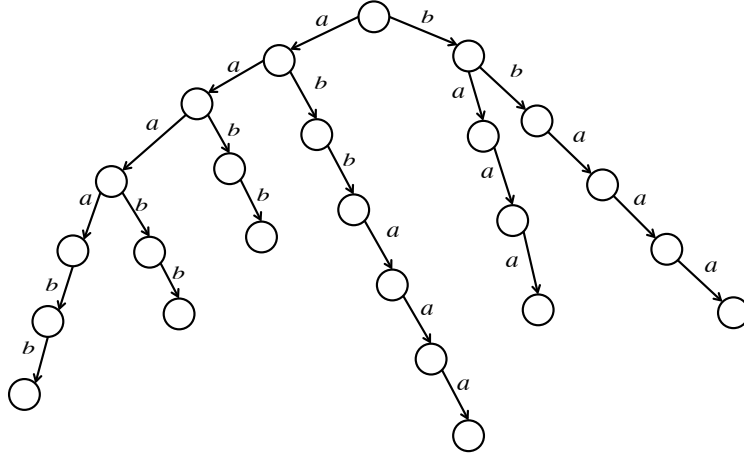
Assume we have already computed the first  $i - 1$  factors  $f_1, \dots, f_{i-1}$  of  $RLZ(w)$  and are computing the  $i$ th factor  $f_i$ . Let  $l_i = \sum_{j=1}^{i-1} |f_j|$ . This implies that we have processed  $(\langle w \rangle[1..k])^{\text{rev}}$  where  $k = \lceil l_i/r \rceil$ , i.e., the  $k$ th meta block contains position  $l_i$ . As is the case with the Starikovskaya algorithm, our algorithm consists of two main phrases, depending on whether  $|f_i| < r$  or  $|f_i| \geq r$ .

#### 3.1 Algorithm for $|f_i| < r$

For any  $k$  ( $1 \leq k \leq n/r$ ), let  $W_k^{\text{rev}}$  denote the set of substrings of  $w[1..kr]^{\text{rev}}$  of length  $2r$  that begin at a block border, i.e.,  $W_k^{\text{rev}} = \{w[tr + 1..(t+2)r]^{\text{rev}} \mid 1 \leq t \leq (k-2)\}$ . We maintain  $STrie(W_k^{\text{rev}})$  in an on-line manner, for  $k = 1, 2, \dots, n/r$ . Note that  $STrie(W_k^{\text{rev}})$  represents all substrings of  $w[1..kr]^{\text{rev}}$  of length  $r$  which do not necessarily begin at a block border. Therefore, we can use  $STrie(W_k^{\text{rev}})$  to determine if  $|f_i| < r$ , and if so, compute  $f_i$ . An example for  $STrie(W_k^{\text{rev}})$  is shown in Figure 4.

A minor issue is that  $STrie(W_k^{\text{rev}})$  may contain “unwanted” substrings that do not correspond to a previous occurrence of  $f_i^{\text{rev}}$  in  $w[1..l_i]$ , since substrings  $w[(k-2)r + 1..y]^{\text{rev}}$  for any  $l_i < y \leq kr$  are represented by  $STrie(W_k^{\text{rev}})$ . In order to avoid finding such unwanted occurrences of  $f_i^{\text{rev}}$ , we associate to each node  $v$  representing a reversed substring  $x^{\text{rev}}$ , the leftmost ending position of  $x$  in  $w[1..kr]$ . Assume we have traversed the prefix of length  $p \geq 0$  of  $w[l_i + 1..n]$  in the trie, and all the nodes involved in the traversal have positions smaller than  $l_i + 1$ . If either the node representing  $w[l_i + 1..l_i + p + 1]$  stores a position larger than  $l_i$  or there is no node representing  $w[l_i + 1..l_i + p + 1]$ , then  $f_i = w[l_i + 1..l_i + p]$  if  $p \geq 1$ , and  $f_i = w[l_i + 1]$  if  $p = 0$ .

As is described above,  $f_i$  can be computed in  $O(|f_i| \log \sigma)$  time. When  $l_i + p > kr$ , we insert the suffixes of a new substring  $w[(k-1)r + 1..(k+1)r]^{\text{rev}}$  of length  $2r$  into the trie, and obtain the updated trie  $STrie(W_{k+1}^{\text{rev}})$ . Since there exist  $\sigma^{2r} = \sigma^{\frac{\log n}{2}} = \sqrt{n}$



**Figure 4.** Let  $r = 3$  and consider string  $w = bba|aaa|bba|bac$ , where  $|$  represents a block border. The figure shows  $STrie(W_3^{\text{rev}})$  where  $W_3^{\text{rev}} = \{aaaabb, abbaaa\}$ .

distinct strings of length  $2r$ , the number of nodes in the trie is bounded by  $O(\sqrt{nr^2}) = O(\sqrt{n}(\log_\sigma n)^2)$ . Hence the trie requires  $o(n)$  bits of space. Each update adds  $O(r^2)$  new nodes and edges into the trie, taking  $O(r^2 \log \sigma)$  time. Since there are  $n/r$  blocks, the total time complexity to maintain the trie is  $O(nr \log \sigma) = O(n \log n)$ .

The above discussion leads to the following lemma:

**Lemma 6.** *We can maintain in  $O(n \log n)$  total time, a dynamic data structure occupying  $o(n)$  bits of space that allows whether or not  $|f_i| < r$  to be determined in  $O(|f_i| \log \sigma)$  time, and if so, computes  $f_i$  and a previous occurrence of  $f_i^{\text{rev}}$  in  $O(|f_i| \log \sigma)$  time.*

### 3.2 Algorithm for $|f_i| \geq r$

Assume we have found that the length of the longest prefix of  $w[l_i + 1..n]$  that is represented by  $STrie(W_k^{\text{rev}})$  is at least  $r$ , which implies that  $|f_i| \geq r$ .

For any string  $f$  and integer  $0 \leq m \leq \min(|f|, r - 1)$ , let strings  $\alpha_m(f)$ ,  $\beta_m(f)$ ,  $\gamma_m(f)$  satisfy  $f = \alpha_m(f)\beta_m(f)\gamma_m(f)$ ,  $|\alpha_m(f)| = m$ , and  $|\beta_m(f)| = j'r$  where  $j' = \max\{j \geq 0 \mid m + jr \leq |f|\}$ . We say that an occurrence of  $f$  in  $w$  has offset  $m$  ( $0 \leq m \leq r - 1$ ), if, in the occurrence,  $\alpha_m(f)$  corresponds to a suffix of a meta-block,  $\beta_m(f)$  corresponds to a sequence of meta-blocks (i.e.  $\beta_m(f) \in \text{Substr}(\langle w \rangle)$ ), and  $\gamma_m(f)$  corresponds to a prefix of a meta-block. Let  $f_i^m$  denote the longest prefix of  $w[l_i + 1..n]$  which has a previous occurrence in  $w[1..l_i]$  with offset  $m$ . Thus,  $|f_i| = \max_{0 \leq m < r} |f_i^m|$ .

Our algorithm maintains two suffix trees on meta-strings,  $STree(\langle w[1..k-1] \rangle^{\text{rev}})$  and  $STree(\langle w[1..k] \rangle^{\text{rev}})$ . Depending on the value of  $m$ , we use either  $STree(\langle w[1..k-1] \rangle^{\text{rev}})$  and  $STree(\langle w[1..k] \rangle^{\text{rev}})$ .

If  $l_i - (k-1)r \geq m$ , i.e. the distance between the  $(k-1)$ th block border and position  $l_i$  is not less than  $m$ , then we use  $STree(\langle w[1..k] \rangle^{\text{rev}})$  to find  $f_i^m$ . We associate to each internal node  $v$  of  $STree(\langle w[1..k] \rangle^{\text{rev}})$  the lexicographical ranks of the leftmost and rightmost leaves in the subtree rooted at  $v$ , denoted  $left(v)$  and  $right(v)$ , respectively. Recall that the leaves of  $STree(\langle w[1..k] \rangle^{\text{rev}})$  correspond to the block borders  $1, r + 1, \dots, (k-1)r + 1$ . Hence,  $\alpha_m(f_i^m)\beta_m(f_i^m)$  occurs in  $w[1..l_i]^{\text{rev}}$  iff there is a node  $v$  representing  $\beta_m(f_i^m)$  and the interval  $[left(v), right(v)]$  contains at least one block



border  $b$  such that  $w[b-m..b-1] = \alpha_m(f_i^m)$ . To determine  $\gamma_m(f_i^m)$ , at each node  $v$  of  $STree(\langle w \rangle[1..k]^{\text{rev}})$  we maintain a trie  $T_v$  that stores the first meta-characters of the outgoing edge labels of  $v$ . Then,  $\alpha_m(f_i^m)\beta_m(f_i^m)\gamma_m(f_i^m)$  occurs in  $w[1..l_i]^{\text{rev}}$  iff there is a node  $u$  of  $T_v$  representing  $\gamma_m(f_i^m)$  and the interval  $[left(u_1), right(u_2)]$  contains at least one block border  $b$  such that  $w[b-m..b-1] = \alpha_m(f_i^m)$ , where  $u_1$  and  $u_2$  are respectively the leftmost and rightmost children of  $u$  in  $T_v$ .

If  $l_i - (k-1)r < m$ , i.e. if the distance between the  $(k-1)$ th block border and position  $l_i$  is less than  $m$ , then we use  $STree(\langle w \rangle[1..k-1]^{\text{rev}})$  to find  $f_i^m$ . This allows us to find only previous occurrences of  $f_i^{\text{rev}}$  that end before  $l_i + 1$ . All the other procedures follow the case where  $l_i - (k-1)r \geq m$ , mentioned above.

**Lemma 7.** *We can maintain in  $O(n \log^2 n)$  total time, a dynamic data structure occupying  $O(n \log \sigma)$  bits of space that allows to compute  $f_i$  with  $|f_i| \geq r$  and a previous occurrence of  $f_i^{\text{rev}}$  in  $O(|f_i| \log^2 n)$  time.*

*Proof.* Traversing the suffix tree for  $\beta_m(f_i^m)$  takes  $O(\frac{|f_i^m|}{r} \log n) = O(|f_i^m| \log \sigma)$  time since  $\|\langle \beta_m(f_i^m) \rangle\| \leq \frac{|f_i^m|}{r}$ . Also, traversing the trie for  $\gamma_m(f_i^m)$  takes  $O(r \log \sigma)$  time, since  $|\gamma_m(f_i^m)| < r$ . To assure  $\beta_m(f_i^m)\gamma_m(f_i^m)$  is immediately preceded by  $\alpha_m(f_i^m)$ , we use the dynamic data structure proposed by Starikovskaya [18] which is based on the dynamic wavelet trees [13]. At each node  $v$ , the data structure allows us to check if the interval  $[left(v), right(v)]$  contains a block border of interest in  $O(\log^2 n)$  time, and to insert a new element to the data structure in  $O(\log^2 n)$  time. Thus,  $f_i$  can be computed in  $O(\sum_{0 \leq m \leq r-1} (|f_i^m| \log \sigma + r \log \sigma + \frac{|f_i^m|}{r} \log^2 n)) = O(|f_i| \log^2 n)$ . The position of a previous occurrence of  $f_i^{\text{rev}}$  can be retrieved in constant time, since each leaf of the suffix tree corresponds to a block border. Once  $f_i$  is computed, we update  $STree(\langle w \rangle[1..k]^{\text{rev}})$  to  $STree(\langle w \rangle[1..k']^{\text{rev}})$ , such that the  $k'$ th block border contains position  $l_{i+1}$  in  $w$ . Using Lemma 1, the suffix tree can be maintained in a total of  $O(\frac{n}{r} \log \sigma) = O(n \log n)$  time.

It follows from Lemma 1 that the suffix tree on meta-strings requires  $O(\frac{n}{r} \log n) = O(n \log \sigma)$  bits of space. Since the dynamic data structure of Starikovskaya [18] takes  $O(n \log \sigma)$  bits of space, the total space complexity of our algorithm is  $O(n \log \sigma)$  bits.  $\square$

The main result of this section follows from Lemma 6 and Lemma 7:

**Theorem 8.** *Given a string  $w$  of length  $n$ , we can compute  $RLZ(w)$  in an on-line manner, in  $O(n \log^2 n)$  time and  $O(n \log \sigma)$  bits of space.*

## 4 On-line computation of reversed LZ factorization with self-references

In this section, we consider to compute  $RLZS(w)$  for a given string  $w$  in an on-line manner. An interesting property of the reversed LZ factorization with self-references is that, the factorization can significantly change when a new character is appended to the end of the string. A concrete example is shown in Figure 5, which illustrates on-line computation of  $RLZS(w)$  with  $w = abbaaaabbbac$ . Focus on the factorization of  $abbaaaab$ . Although there is a factor starting at position 5 in  $RLZS(abbaaaab)$ , there is no factor starting at position 5 in  $RLZS(abbaaaabb)$ . Below, we will characterize this with its close relationship to palindromes.

$a b b a a a b b b a c$
-------------------------

$a|$   
 $a|b|$   
 $a|b|b|$   
 $a|b|b a|$   
 $a|b|b a|a|$   
 $a|b|b a|a a|$   
 $a|b|b a|a a a|$   
 $a|b|b a a a a b|$   
 $a|b|b a a a a b b|$   
 $a|b|b a a a a b b|b a|$   
 $a|b|b a a a a b b|b a|c|$

**Figure 5.** A snapshot of on-line computation of  $RLZS(w)$  with  $w = abbaaaabbbac$ . For each non-empty prefix  $w[1..k]$  of  $w$ ,  $|$  denotes the boundary of factors in  $RLZS(w[1..k])$ .

#### 4.1 Computing $RLZS(w)$ in $O(n \log \sigma)$ time and $O(n \log n)$ bits of space

Let  $w$  be any string of length  $n$ . For any  $1 \leq j \leq n$ , the occurrence of substring  $p$  starting at position  $j$  is called self-referencing, if there exists  $j'$  such that  $w[j'..j' + |p| - 1]^{\text{rev}} = w[j..j + |p| - 1]$  and  $j \leq j' + |p| - 1 < j + |p| - 1$ .

For any  $1 \leq k \leq n$ , let  $Lpal_w(k) = \max\{k - j + 1 \mid w[j..k] = w[j..k]^{\text{rev}}, 1 \leq j \leq k\}$ . That is,  $Lpal_w(k)$  is the length of the longest palindrome that ends at position  $k$  in  $w$ .

**Lemma 9.** For any string  $w$  of length  $n$  and  $1 \leq k \leq n$ , let  $RLZS(w[1..k - 1]) = g_1, \dots, g_p$ . Let  $\ell_q = \sum_{h=1}^q |g_h|$  for any  $1 \leq q \leq p$ . Then

$$RLZS(w[1..k]) = \begin{cases} g_1, \dots, g_p w[k] & \text{if } g_p w[k] \in \text{Substr}(w[1..\ell_{p-1}]^{\text{rev}}) \text{ and } \ell_{p-1} + 1 \leq d_k, \\ g_1, \dots, g_p, w[k] & \text{if } g_p w[k] \notin \text{Substr}(w[1..\ell_{p-1}]^{\text{rev}}) \text{ and } \ell_{p-1} + 1 \leq d_k, \\ g_1, \dots, g_j, w[\ell_j + 1..k] & \text{otherwise,} \end{cases}$$

where  $d_k = k - Lpal_w(k) + 1$  and  $j$  is the minimum integer such that  $\ell_j \geq d_k$ .

*Proof.* By definition of  $Lpal_w(k)$  and  $d_k$ ,  $w[d_k..k]$  is the longest suffix palindrome of  $w[1..k]$ . If  $\ell_{p-1} + 1 \leq d_k$ ,  $w[\ell_{p-1} + 1..k]$  cannot be self-referencing. Hence the first and the second cases of the lemma follow. Consider the third case. Since  $\ell_j \geq d_k$ ,  $w[\ell_j + 1..k]$  is self-referencing. Since  $RLZS(w[1..\ell_j]) = g_1, \dots, g_j$ , the third case follows.  $\square$

See Figure 5 and focus on  $RLZS(abbaaaab)$ , where  $g_1 = a$ ,  $g_2 = b$ ,  $g_3 = ba$ , and  $g_4 = aaab$ . Consider to compute  $RLZS(abbaaaabb)$ . Since the longest suffix palindrome  $bbbaaaabb$  intersects the boundary between  $g_3$  and  $g_4$  of  $RLZS(abbaaaab)$ , the third case of Lemma 9 applies. Consequently, the new factorization  $RLZS(abbaaaabb)$



consists of  $g_1 = a$  and  $g_2 = b$  of  $RLZS(abbaaaab)$ , and a new self-referencing factor  $g_3 = baaaaabb$ .

**Theorem 10.** *Given a string  $w$  of length  $n$ , we can compute  $RLZS(w)$  in an on-line manner, in  $O(n \log \sigma)$  time and  $O(n \log n)$  bits of space.*

*Proof.* Suppose we have already computed  $RLZS(w[1..k-1])$ , and we are computing  $RLZS(w[1..k])$  for  $1 \leq k \leq n$ .

Assume  $\ell_{p-1} + 1 \leq d_k$ . We check whether  $g_p w[k] \in \text{Substr}(w[1..\ell_{p-1}]^{\text{rev}})$  or not using  $STree(w[1..\ell_{p-1}]^{\text{rev}})$ . If the first case of Lemma 9 applies, then we proceed to the next position  $k+1$  and continue to traverse the suffix tree. If the second case of Lemma 9 applies, then we update the suffix tree for the reversed string, and proceed to computing  $RLZS(w[1..k+1])$ .

Assume  $\ell_{p-1} + 1 > d_k$ , i.e., the third case of Lemma 9 holds. For every  $j < e \leq p$ , we remove  $g_e$  of  $RLZS(w[1..k-1])$ , and the last factor of  $RLZS(w[1..k])$  is  $w[\ell_j+1..k]$ . We then proceed to computing  $RLZS(w[1..k+1])$ .

As is mentioned in Section 2.3, in a total of  $O(n \log \sigma)$  time and  $O(n \log n)$  bits of space, we can check whether the first or the second case of Lemma 9 holds, as well as maintain the suffix tree for the reversed string on-line. In order to compute  $Lpal_w(k)$  in an on-line manner, we can use Manacher's algorithm [14] which computes the maximal palindromes for all centers in  $w$  in  $O(n)$  time and in an on-line manner. Since Manacher's algorithm actually maintains the center of the longest suffix palindrome of  $w[1..k]$  when processing  $w[1..k]$ , we can easily modify the algorithm to also compute  $Lpal_w(k)$  on-line. Since Manacher's algorithm needs to store the length of maximal palindromes for every center in  $w$ , it takes  $O(n \log n)$  bits of space.

Finally, we show the total number of factors that are removed in the third case of Lemma 9. Once a factor that begins at position  $j$  is removed after computing  $RLZS(w[1..k])$  for some  $k$ , for any  $k \leq k' \leq n$ ,  $RLZS(w[1..k'])$  never contains a factor starting at position  $j$ . Hence, the total number of factors that are removed in the third case is at most  $n$ . This completes the proof.  $\square$

## 4.2 Computing $RLZS(w)$ in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space

In this subsection, we present a space efficient algorithm that computes  $RLZS(w)$  on-line, using only  $O(n \log \sigma)$  bits of space. Note that we cannot use the method mentioned in the proof of Theorem 10, as it requires  $O(n \log n)$  bits of space. Instead, we maintain a compact representation of all suffix palindromes of each prefix  $w[1..k]$  of  $w$ , as follows.

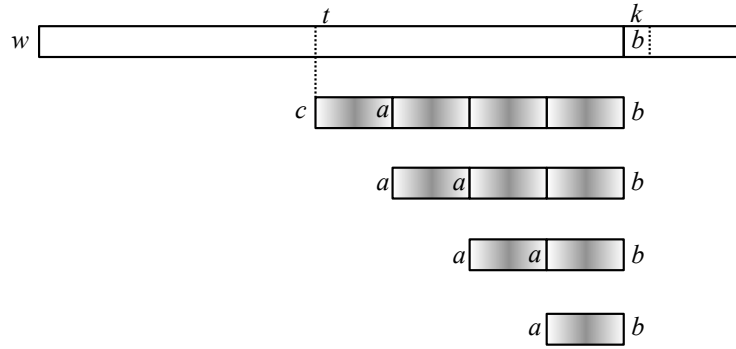
For any string  $w$  of length  $n \geq 1$ , let  $Spals(w)$  denote the set of the beginning positions of the palindromic suffixes of  $w$ , i.e.,

$$Spals(w) = \{n - |s| + 1 \mid s \in \text{Suffix}(w), s \text{ is a palindrome}\}.$$

**Lemma 11 ([2,15]).** *For any string  $w$  of length  $n$ ,  $Spals(w)$  can be represented by  $O(\log n)$  arithmetic progressions.*

The above lemma implies that  $Spals(w)$  can be represented by  $O(\log^2 n)$  bits of space.

**Lemma 12.** *We can maintain  $O(\log^2 n)$ -bit representation of  $Spals(w[1..k])$  on-line for every  $1 \leq k \leq n$  in a total of  $O(n \log n)$  time.*



**Figure 6.** Illustration of Lemma 12. Let  $w[t-1] = c$ ,  $w[t+q-1] = a$ , and  $w[k] = b$ .  $w[t-1..k]$  is a suffix palindrome of  $w[1..k]$  iff  $c = b$ , and  $w[t+iq-1..k]$  is a suffix palindrome of  $w[1..k]$  for any  $1 \leq i < m$  iff  $a = b$ .

*Proof.* We show how to efficiently update  $Spals(w[1..k-1])$  to  $Spals(w[1..k])$ . Let  $S$  be any subset of  $Spals(w[1..k-1])$  which is represented by a single arithmetic progression  $\langle t, q, m \rangle$ , where  $t$  is the first (minimum) element,  $q$  is the step, and  $m$  is the number of elements of the progression. Let  $s_j$  be the  $j$ th smallest element of  $S$ , with  $1 \leq j \leq m$ . By definition,  $s_j$  is a suffix palindrome of  $w[1..k-1]$  for any  $j$ . In addition, if  $m \geq 3$ , then it appears that, for any  $1 \leq j < m$ ,  $s_j$  has a period  $q$ . Therefore, we can test whether the elements of  $S$  correspond to the suffix palindromes of  $w[1..k]$ , by two character comparisons:  $w[t-1] = w[k]$  iff  $t-1 \in Spals(w[1..k])$ , and  $w[t+iq-1] = w[k]$  iff  $t+iq-1 \notin Spals(w[1..k])$  for any  $1 \leq i < m$ . (See also Figure 6.) If the extension of only one element of  $S$  becomes an element of  $Spals(w[1..k])$ , then we check if it can be merged to the adjacent arithmetic progression that contains closest smaller positions. As above, we can process each arithmetic progression in  $O(1)$  time. By Lemma 11, there are  $O(\log n)$  arithmetic progressions in  $Spals(w[1..k])$  for each prefix of  $w[1..k]$  of  $w$ . Consequently, for each  $1 \leq k \leq n$  we can maintain  $O(\log^2 n)$ -bit representation of  $Spals(w[1..k])$  in a total of  $O(n \log n)$  time.  $\square$

The main result of this subsection follows:

**Theorem 13.** *Given a string  $w$  of length  $n$ , we can compute  $RLZS(w)$  in an on-line manner, in  $O(n \log^2 n)$  time and  $O(n \log \sigma)$  bits of space.*

*Proof.* Assume that we are computing a new factor that begins at position  $\ell$  of  $w$ . First, we use the algorithm of Theorem 8 and obtain the longest prefix  $f$  of  $w[\ell..n]$  such that  $f^{\text{rev}}$  has an occurrence in  $w[1..\ell-1]$ . Then we apply Lemma 9 for  $w[1..\ell+|f|-1]$ , and if the third case holds, then we compute the self-reference factor. We use Lemma 12 to compute  $Lpal_w(k)$  for any given position  $k$ . After computing the new factor, then we update the suffix tree of the meta-string, and proceed to computing the next factor. Overall, the algorithm takes  $O(n \log^2 n)$  time and  $O(n \log \sigma + \log^2 n) = O(n \log \sigma)$  bits of space.  $\square$

## 5 Reversed LZ factorization and smallest grammar

For any string  $w$ , the number of the LZ77 factors [21] (with/without self-references) of  $w$  is known to be a lower bound of the smallest grammar that derives only  $w$  [17].

Here we briefly show that this is not the case with the reversed LZ factorization (for either with or without self-references).

**Theorem 14.** *For  $\sigma = 3$ , there is an infinite series of strings for which the smallest grammar has size  $O(\log n)$  while the size of the reversed LZ factorization is  $O(n)$ .*

*Proof.* Let  $w = (abc)^{\frac{n}{3}}$ . Then,  $RLZ(w) = RLZS(w) = a, b, c, a, b, c, \dots, a, b, c$ , consisting of exactly  $n$  factors. On the other hand, it is easy to see that there exists a grammar of size  $O(\log n)$  that generates only  $w$ . This completes the proof.  $\square$

The above theorem applies to any constant alphabet of size at least 3. When  $\sigma = 1$ , the size of the smallest grammar and the number of factors in  $RLZ(w)$  are both  $O(\log n)$ , while the number of factors in  $RLZS(w)$  is  $O(1)$ . The binary case where  $\sigma = 2$  is open.

## References

1. A. AL-HAFEEDH, M. CROCHEMORE, L. ILIE, J. KOPYLOV, W. SMYTH, G. TISCHLER, AND M. YUSUFU: *A comparison of index-based Lempel-Ziv LZ77 factorization algorithms*. ACM Computing Surveys, 45(1) 2012, p. Article 5.
2. A. APOSTOLICO, D. BRESLAUER, AND Z. GALIL: *Parallel detection of all palindromes in a string*. Theoretical Computer Science, 141(1&2) 1995, pp. 163–173.
3. A. BLUMER, J. BLUMER, D. HAUSSLER, A. EHRENFUCHT, M. T. CHEN, AND J. I. SEIFERAS: *The smallest automaton recognizing the subwords of a text*. Theoretical Computer Science, 40 1985, pp. 31–55.
4. J.-P. DUVAL, R. KOLPAKOV, G. KUCHEROV, T. LECROQ, AND A. LEFEBVRE: *Linear-time computation of local periods*. Theoretical Computer Science, 326(1-3) 2004, pp. 229–240.
5. K. GOTO AND H. BANNAI: *Simpler and faster Lempel Ziv factorization*, in Proc. DCC 2013, 2013, pp. 133–142.
6. J. KÄRKKÄINEN, D. KEMPA, AND S. J. PUGLISI: *Lightweight Lempel-Ziv parsing*, in Proc. SEA 2013, 2013, pp. 139–150.
7. J. KÄRKKÄINEN, D. KEMPA, AND S. J. PUGLISI: *Linear time Lempel-Ziv factorization: Simple, fast, small*, in Proc. CPM 2013, 2013, pp. 189–200.
8. D. KEMPA AND S. J. PUGLISI: *Lempel-Ziv factorization: Simple, fast, practical*, in Proc. ALENEX 2013, 2013, pp. 103–112.
9. R. KOLPAKOV AND G. KUCHEROV: *Finding maximal repetitions in a word in linear time*, in Proc. FOCS 1999, 1999, pp. 596–604.
10. R. KOLPAKOV AND G. KUCHEROV: *Searching for gapped palindromes*. Theoretical Computer Science, 410(51) 2009, pp. 5365–5373.
11. S. KREFT AND G. NAVARRO: *Self-indexing based on LZ77*, in Proc. CPM 2011, 2011, pp. 41–54.
12. S. KURUPPU, S. J. PUGLISI, AND J. ZOBEL: *Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval*, in Proc. SPIRE 2010, 2010, pp. 201–206.
13. V. MÄKINEN AND G. NAVARRO: *Dynamic entropy-compressed sequences and full-text indexes*. ACM Transactions on Algorithms, 4(3) 2008.
14. G. K. MANACHER: *A new linear-time “on-line” algorithm for finding the smallest initial palindrome of a string*. J. ACM, 22(3) 1975, pp. 346–351.
15. W. MATSUBARA, S. INENAGA, A. ISHINO, A. SHINOHARA, T. NAKAMURA, AND K. HASHIMOTO: *Efficient algorithms to compute compressed longest common substrings and compressed palindromes*. Theoretical Computer Science, 410(8–10) 2009, pp. 900–913.
16. D. OKANOHARA AND K. SADAKANE: *An online algorithm for finding the longest previous factors*, in Proc. ESA 2008, 2008, pp. 696–707.
17. W. RYTTER: *Application of Lempel-Ziv factorization to the approximation of grammar-based compression*. Theoretical Computer Science, 302(1-3) 2003, pp. 211–222.
18. T. A. STARIKOVSKAYA: *Computing Lempel-Ziv factorization online.*, in Proc. MFCS 2012, 2012, pp. 789–799.

19. P. WEINER: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, 1973, pp. 1–11.
20. J. YAMAMOTO, T. I, H. BANNAI, S. INENAGA, AND M. TAKEDA: *Faster compact on-line Lempel-Ziv factorization*. CoRR, abs/1305.6095 2013.
21. J. ZIV AND A. LEMPEL: *A universal algorithm for sequential data compression*. IEEE Transactions on Information Theory, IT-23(3) 1977, pp. 337–349.