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 $O(n \log n)$ 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 [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 [4] for a survey, and [3,5,6,7] 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 [14] 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 [13] proposed an algorithm running in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space, assuming $\frac{\log N}{4}$ 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) 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

1 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 \( \Sigma^n \) 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 \( \text{Substr}(w) \) and \( \text{Suffix}(w) \), 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 palindromic 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 \( r = \frac{\log \sigma s}{4} = \frac{\log n}{4 \log \sigma} \). 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 \( \Sigma^r = \{1, \ldots, n\} \). 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 [14] 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;
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 \cite{19} constructs $STree(s^{rev})$ in an online manner from left to right, i.e., constructs $STree(s[1..j]^{rev})$ in increasing order of $j = 1, 2, \ldots, |s|$, 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 \cite{3} of $s$ forms $STree(s^{rev})$.

Hence, for larger alphabets, we have the following:

**Lemma 1** (\cite{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 \cite{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 $(f_1, f_2, \ldots, f_m)$ of non-empty substrings of $w$ such that

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

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{STree.png}
\caption{STree($w$) with $w = abaaabbbab$.}
\end{figure}
Assume we have $f_1, \ldots, f_{i-1}$, 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 $(w[1..k-1])^{rev}$ if such exists. See also Figure 2.

**Example 3.** For string $w = abaaaaabbac$, $RLZ(w)$ 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 $f_1, f_2, \ldots, f_j$, and we have constructed $STree(w[1..l_j])^{rev}$, 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 $STree(w[1..l_j])^{rev}$. After the computation of $f_{j+1}$, we update $STree(w[1..l_j])^{rev}$ to $STree(w[1..l_{j+1}])^{rev}$, 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 $(g_1, g_2, \ldots, g_p)$ of non-empty substrings of $w$ such that

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

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 $(w[1..k + |g_i| - 2])^{rev}$ if such exists, and $g_i = w[k]$ otherwise. See also Figure 3.

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

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 $(w[1..k - 1])^{rev}$ if such exists.

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 $(w[1..k + |g_i| - 2])^{rev}$ if such exists.
Note that in Definition 1 the ending position of a previous occurrence of \( q_t \) does not have to be prior to the beginning position \( k \) of \( q_t \), 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 \( w \) in an on-line manner, i.e., maintains \( STree(w[1..k]) \) in increasing order of \( k = 1, 2, \ldots, n/r \), 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 \( STree((w)[1..k])^{rev} \) in increasing order of \( k = 1, 2, \ldots, n/r \), and maintain a generalized suffix trie for a set of substrings of \( w[1..kr]^{rev} \) of length \( 2r \) that begin at a block border.

Assume we have already computed the first \( i-1 \) factors \( f_1, \ldots, f_{i-1} \) 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 \( (w[1..k])^{rev} \) where \( k = [l_i/r] \), 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 \( W_k^{rev} \) denote the set of substrings of \( w[1..kr]^{rev} \) of length \( 2r \) that begin at a block border, i.e., \( W_k^{rev} = \{ w[t+1..(t+2)r]^{rev} \mid 1 \leq t \leq (k-2) \} \). We maintain \( STrie(W_k^{rev}) \) in an on-line manner, for \( k = 1, 2, \ldots, n/r \). Note that \( STrie(W_k^{rev}) \) represents all substrings of \( w[1..kr]^{rev} \) of length \( r \) which do not necessarily begin at a block border. Therefore, we can use \( STrie(W_k^{rev}) \) to determine if \( |f_i| < r \), and if so, compute \( f_i \). An example for \( STrie(W_k^{rev}) \) is shown in Figure 3.

A minor issue is that \( STrie(W_k^{rev}) \) may contain “unwanted” substrings that do not correspond to a previous occurrence of \( f_i^{rev} \) in \( w[l_i..l_i] \), since substrings \( w[(k-2)r+1..y]^{rev} \) for any \( l_i < y \leq kr \) are represented by \( STrie(W_k^{rev}) \). In order to avoid finding such unwanted occurrences of \( f_i^{rev} \), we associate to each node \( v \) representing a reversed substring \( x^{rev} \), the leftmost ending position of \( x \) in \( w[1..kr]^{rev} \). 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 \( w[(k-1)r+1..(k+1)r]^{rev} \) of length \( 2r \) into the trie, and obtain the updated trie \( STrie(W_{k+1}^{rev}) \). Since there exist \( \sigma^{2r} = \sigma^{\log n} = \sqrt{n} \),
Figure 4. Let $r = 3$ and consider string $w = bba|aaa|bbabac$, where $|$ represents a block border. The figure shows $STrie(W_{3}^{rev})$ where $W_{3}^{rev} = \{aaaabb, abbaaa\}.

distinct strings of length $2r$, the number of nodes in the trie is bounded by $O(\sqrt{n}r^{2}) = O(\sqrt{n}(\log_{n} n)^{2})$. 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 $f_{i}^{rev}$ 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 $STrie(W_{k}^{rev})$ 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. $\beta_{m}(f) \in Substr(\langle w \rangle))$, 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, $STree((\langle w \rangle[1..k - 1])^{rev})$ and $STree((\langle w \rangle[1..k])^{rev})$. Depending on the value of $m$, we use either $STree((\langle w \rangle[1..k - 1])^{rev})$ and $STree((\langle w \rangle[1..k])^{rev})$.

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 $STree((\langle w \rangle[1..k])^{rev})$ to find $f_{i}^{m}$. We associate to each internal node $v$ of $STree((\langle w \rangle[1..k])^{rev})$ 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 $STree((\langle w \rangle[1..k])^{rev})$ correspond to the block borders $1, r + 1, \ldots, (k - 1)r + 1$. Hence, $\alpha_{m}(f_{i}^{m})\beta_{m}(f_{i}^{m})$ occurs in $w[1..l_{i}]^{rev}$ 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 $\text{STree}((\langle w \rangle[1..k])^{rev})$ 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 $w[1..l]^{rev}$ iff there is a node $u$ of $T_v$ representing $\gamma_m(f_i^m)$ and the interval $[\text{left}(u_1), \text{right}(u_2)]$ 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 the distance between the $(k - 1)$th block border and position $l_i$ is less than $m$, then we use $\text{STree}((\langle w \rangle[1..k - 1])^{rev})$ to find $f_i^m$. This allows us to find only previous occurrences of $f_i^{rev}$ 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 $O(n \log \sigma)$ 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 $f_i^{rev}$ in $O(|f_i| \log^2 n)$ time.

**Proof.** Traversing the suffix tree for $\beta_m(f_i^m)$ takes $O(|f_i^m| \log n) = O(|f_i^m| \log \sigma)$ time since $||\beta_m(f_i^m)|| \leq |f_i^m|$. 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 [13] which is based on the dynamic wavelet trees [13]. At each node $v$, the data structure allows us to check if the interval $[\text{left}(v), \text{right}(v)]$ 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 $O(\sum_{0 \leq n \leq r - 1}(|f_i^m| \log \sigma + r \log \sigma + |f_i^m| \log^2 n)) = O(|f_i^m| \log^2 n)$. The position of a previous occurrence of $f_i^{rev}$ 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 $\text{STree}((\langle w \rangle[1..k])^{rev})$ to $\text{STree}((\langle w \rangle[1..k'])^{rev})$, such that the $k'$th block border contains position $l_{i+1}$ in $w$. Using Lemma 4, 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 3 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 [13] takes $O(n \log \sigma)$ bits of space, the total space complexity of our algorithm is $O(n \log \sigma)$ bits.

The main result of this section follows from Lemma 3 and Lemma 4.

**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 3, which illustrates on-line computation of RLZS($w$) with $w = \text{abbaaababbac}$. Focus on the factorization of abbaaab. Although there is a factor starting at position 5 in RLZS(abbaaab), there is no factor starting at position 5 in RLZS(abbaaab). Below, we will characterize this with its close relationship to palindromes.
\[ \text{RLZS}(w) \text{ of } w = ababaaabbabc. \]

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

4.1 Computing \text{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 \( w[j..j'] + |p| - 1]^{rev} = w[j..j + |p| - 1] \) and \( j \leq j' + |p| - 1 < j + |p| - 1 \).

For any \( 1 \leq k \leq n \), let \( Lpal_w(k) = \max\{k - j + 1 \mid w[j..k] = w[j..k]^{rev}, 1 \leq j \leq k\} \). 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 \( \text{RLZS}(w[1..k-1]) = g_1, \ldots, g_p \). Let \( \ell_q = \sum_{h=1}^{q} |g_h| \) for any \( 1 \leq q \leq p \). Then

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

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 \( \text{RLZS}(w[1..\ell_j]) = g_1, \ldots, g_j \), the third case follows.

See Figure 5 and focus on \( \text{RLZS}(ababaaab) \), where \( g_1 = a, g_2 = b, g_3 = ba \), and \( g_4 = aaab \). Consider to compute \( \text{RLZS}(ababaaabb) \). Since the longest suffix palindrome \( bbabaaabb \) intersects the boundary between \( g_3 \) and \( g_4 \) of \( \text{RLZS}(ababaaab) \), the third case of Lemma 5 applies. Consequently, the new factorization \( \text{RLZS}(ababaaabb) \)
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 $g_p w[k] \in Substr(w[1..\ell_{p-1}]^{rev})$ or not using $STree(w[1..\ell_{p-1}]^{rev})$. 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. \hfill \Box

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.
Lemma 12 to compute \( L_{\text{pal}} \) the next factor. Overall, the algorithm takes \( O(n) \) time and \( O(n \log \sigma) \) bits of space.

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 \( f^{rev} \) 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 \( L_{\text{pal}}(k) \) 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.

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 $w = (abc)^2$. Then, $RLZ(w) = RLZS(w) = a, b, c, a, b, c, \ldots, a, b, c$, 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.  

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**

