Time and Space Efficient Lempel-Ziv Factorization based on Run Length Encoding

Yamamoto, Jun'ichi
Department of Informatics, Kyushu University

Bannai, Hideo
Department of Informatics, Kyushu University

Inenaga, Shunsuke
Department of Informatics, Kyushu University

Takeda, Masayuki
Department of Informatics, Kyushu University

Time and Space Efficient Lempel-Ziv Factorization based on Run Length Encoding

Jun'ichi Yamamoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda

Department of Informatics, Kyushu University
{junichi.yamamoto,bannai,inenaga,takeda}@inf.kyushu-u.ac.jp

Abstract. We propose a new approach for calculating the Lempel-Ziv factorization of a text efficiently, based on run length encoding (RLE). We present a conceptually simple off-line algorithm based on a variant of suffix arrays, as well as an on-line algorithm based on a variant of directed acyclic word graphs (DAWGs). Both algorithms run in $O(N + n \log n)$ time and $O(n)$ extra space, where N is the size of the text, $n \leq N$ is the number of RLE factors. The time dependency on N is only in the conversion of the text to RLE, which can be computed very efficiently in $O(N)$ time and $O(1)$ extra space. When the text is compressible via RLE, i.e., $n = o(N)$, our algorithms are, to the best of our knowledge, the first algorithms which require only $o(N)$ extra space while running in $o(N \log N)$ time.

1 Introduction

The *run-length encoding* (*RLE*) of a string S is a natural encoding of S, where each maximal run of character a of length p in S is encoded as a^p, e.g., the RLE of string aaaabbbaa is $\mathtt{a}^4\mathtt{b}^3\mathtt{a}^2$. Since RLE can be regarded as a compressed representation of strings, processing time and working space can be reduced significantly when RLE strings are not required to be decompressed while being processed. In fact, quite a number of efficient algorithms that deal with RLE strings have been proposed (e.g.:[2, 1, 15, 9, 14, 4]). In this paper, we introduce yet another application of efficient processing via RLE, the *Lempel-Ziv factorization* (*LZ factorization*).

The LZ factorization (and its variants) of a string [22, 21, 5] is a very important concept with applications not only in the field of data compression but also in the field string algorithms [13, 8]. Therefore, there exists a large amount of work devoted to its efficient computation. A naïve algorithm that computes the longest common prefix with each of the $O(N)$ previous positions only requires $O(1)$ space, but can take $\Theta(N^2)$ time. Most recent algorithms [7, 6, 18] basically require the suffix array of the string, consequently taking $\Theta(N)$ extra space and at least $\Theta(N)$ time to construct.

In this paper, we propose a new approach for calculating the Lempel-Ziv factorization of a string, based on RLE. An interesting feature of our algorithms is that the input string is first "compressed" to RLE, and is then further "compressed" to the LZ encoding. We first show that the size of the LZ encoding

with self-references (i.e., allowing overlapping factors) is at most twice as large as the size of RLE, and can be much smaller. This implies that the working space of our approach is dependent only on the size of RLE. We present two efficient algorithms: an off-line algorithm based on suffix arrays for RLE strings, and an on-line algorithm based on directed acyclic word graphs (DAWGs) for RLE strings. Both algorithms run in $O(N + n \log n)$ time and $O(n)$ extra space, where N is the size of the text, $n \leq N$ is the size of RLE. The time dependency on N is only in the conversion of the text to RLE, which can be computed very efficiently in $O(N)$ time and $O(1)$ extra space.

For general alphabets, we achieve the same worst case time complexity as previous algorithms even when $n = N$, since the construction of the suffix array can take $O(N \log N)$ time. For integer alphabets, our algorithms may be slightly slower than previous algorithms. However, the significance of our algorithms is that when the text is compressible via RLE, i.e., when $n = o(N)$, our algorithms are, to the best of our knowledge, the first algorithms which require only $o(N)$ extra space while running in $o(N \log N)$ time.

Related Work. For computing the LZ78 [23] factorization, a sub-linear time and space algorithm was presented in [10]. In this paper, we consider a variant of the more powerful LZ77 [22] factorization. Two on-line algorithms for LZ factorization based on succinct data structures have been proposed. The first runs in $O(N \log^3 N)$ time and $N \log \sigma + o(N \log \sigma) + O(N)$ bits of space [19], the other runs in $O(N \log^2 N)$ time with $O(N \log \sigma)$ bits of space [20], where σ is the size of the alphabet. Succinct data structures basically simulate accesses to their non-succinct counterparts using less space at the expense of speed. A notable distinction of our approach is that we reduce the *problem size* via compression in order to improve both time and space efficiency.

2 Preliminaries

Let $\mathcal{N}$ be the set of non-negative integers. Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $S = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of S, respectively. The set of prefixes of S is denoted by $\mathit{Prefix}(S)$. The *longest common prefix* of strings X, Y, denoted $\mathit{lcp}(X, Y)$, is the longest string in $\mathit{Prefix}(X) \cap \mathit{Prefix}(Y)$.

The i-th character of a string S is denoted by $S[i]$ for $1 \leq i \leq |S|$, and the substring of a string S that begins at position i and ends at position j is denoted by $S[i..j]$ for $1 \leq i \leq j \leq |S|$. For convenience, let $S[i..j] = \varepsilon$ if $j < i$.

For any character $a \in \Sigma$ and $p \in \mathcal{N}$, let a^p denote the concatenation of p a's, e.g., $a^1 = a$, $a^2 = aa$, and so on. p is said to be the exponent of a^p. Let $a^0 = \varepsilon$.

Our model of computation is the word RAM with the computer word size at least $\lceil \log_2 |S| \rceil$, and hence, standard instructions on values representing lengths and positions of string S can be manipulated in constant time. Space complexities will be determined by the number of computer words (not bits).

2.1 LZ Encodings

LZ encodings are dynamic dictionary based encodings with many variants. As in most recent work, we describe our algorithms with respect to a well known variant called s-factorization [5] in order to simplify the presentation.

Definition 1 (s-factorization [5]). *The s-factorization of a string S is the factorization $S = f_1 \cdots f_n$ where each s-factor $f_i \in \Sigma^+$ $(i = 1, \ldots, n)$ is defined inductively as follows: $f_1 = S[1]$. For $i \geq 2$: if $S[|f_1 \cdots f_{i-1}| + 1] = c \in \Sigma$ does not occur in $f_1 \cdots f_{i-1}$, then $f_i = c$. Otherwise, f_i is the longest prefix of $f_i \cdots f_n$ that occurs at least twice in $f_1 \cdots f_i$.*

Note that each s-factor can be represented in constant size, i.e., either as a single character or a pair of integers representing the position of a previous occurrence of the factor and its length. For example the s-factorization of the string $S =$ `abaababababaaaaabbabab` is `a`, `b`, `a`, `aba`, `baba`, `aaaa`, `b`, `babab`. This can be represented as `a`, `b`, $(1,1)$, $(1,3)$, $(5,4)$, $(10,4)$, $(2,1)$, $(5,5)$.

2.2 Run Length Encoding

Definition 2. *The Run-Length (RL) factorization of a string S is the factorization $f_1, \ldots, f_n$ of S such that for every $i = 1, \ldots, n$, factor f_i is the longest prefix of $f_i \cdots f_n$ with $f_i \in \{a^p \mid a \in \Sigma, p > 0\}$.*

Note that each factor f_i can be written as $f_i = a_i^{p_i}$ for some character $a_i \in \Sigma$ and some integer $p_i > 0$ and for any consecutive factors $f_i = a_i^{p_i}$ and $f_{i+1} = a_{i+1}^{p_{i+1}}$, we have that $a_i \neq a_{i+1}$. The *run length encoding* (RLE) of a string S, denoted RLE_S, is a sequence of pairs consisting of a character a_i and an integer p_i, representing the RL factorization. The *size* of RLE_S is the number of RL factors in RLE_S and is denoted by $size(RLE_S)$, i.e., if $RLE_S = a_1^{p_1} \cdots a_n^{p_n}$, then $size(RLE_S) = n$. RLE_S can be computed in $O(N)$ time and $O(1)$ extra space (excluding the $O(n)$ space for output), simply by scanning S from beginning to end, counting the exponent of each RL factor.

Let *val* be the function that "decompresses" RLE_S, i.e., $val(RLE_S) = S$. For any $1 \leq i \leq j \leq n$, let $RLE_S[i..j] = a_i^{p_i} a_{i+1}^{p_{i+1}} \cdots a_j^{p_j}$. For convenience, let $RLE_S[i..j] = \varepsilon$ if $i > j$. Let

$$RLE\_Substr(S) = \{RLE_S[i..j] \mid 1 \leq i, j \leq n\} \text{ and}$$
$$RLE\_Suffix(S) = \{RLE_S[i..n] \mid 1 \leq i \leq n\}.$$

The following simple but nice observation allows us to represent the complexity of our algorithms in terms of $size(RLE_S)$.

Lemma 1. *For a given string S, let n_{RL} and n_{LZ} respectively be the number of factors in its RL factorization and s-factorization. Then, $n_{LZ} \leq 2n_{RL}$.*

Proof. Consider an s-factor that starts at the jth position in some RL-factor $a_i^{p_i}$ where $1 < j \leq p_i$. Since $a_i^{p_i - j + 1}$ is both a suffix and a prefix of $a_i^{p_i}$, we have that the s-factor extends at least to the end of $a_i^{p_i}$. This implies that a single RL-factor is always covered by at most 2 s-factors, thus proving the lemma. □

Note that for LZ factorization variants without self-references, the size of the output LZ encoding may come into play, when it is larger than $O(size(RLE_S))$.

2.3 Priority Search Trees

We will use the following data structure in our LZ factorization algorithms.

Theorem 1 (McCreight [17]). *For a dynamic set D which contains n ordered pairs of integers, the priority search tree data structure supports all the following operations and queries in $O(\log n)$ time, using $O(n)$ space:*

- *Insert(x, y): Insert a pair (x, y) into D;*
- *Delete(x, y): Delete a pair (x, y) from D;*
- *MinXInRectangle(L, R, B): Given three integers $L \leq R$ and B, return the pair $(x, y) \in D$ with minimum x satisfying $L \leq x \leq R$ and $y \geq B$;*
- *MaxXInRectangle(L, R, B): Given three integers $L \leq R$ and B, return the pair $(x, y) \in D$ with maximum x satisfying $L \leq x \leq R$ and $y \geq B$;*
- *MinYInRange(L, R): Given two integers $L \leq R$, return the pair $(x, y) \in D$ with minimum y.*

3 Off-line LZ Factorization based on RLE

In this section we present our off-line algorithm for s-factorization. The term off-line here implies that the input string S of length N is first converted to a sequence of RL factors, $RLE_S = a_1^{p_1} a_2^{p_2} \cdots a_n^{p_n}$. In the algorithm which follows, we use several new data structures for RLE_S.

3.1 RLE Suffix Arrays

Let $\Sigma_{RLE_S} = \{RLE_S[i] \mid i = 1, \ldots, n\}$. For instance, if $RLE_S = \mathtt{a^3b^5a^3b^5a^1b^5a^4}$, then $\Sigma_{RLE_S} = \{\mathtt{a^1}, \mathtt{a^3}, \mathtt{a^4}, \mathtt{b^5}\}$. For any $a_i^{p_i}, a_j^{p_j} \in \Sigma_{RLE_S}$, let the order $\prec$ on Σ_{RLE_S} be defined as

$$a_i^{p_i} \prec a_j^{p_j} \iff a_i < a_j, \text{ or } a_i = a_j \text{ and } p_i < p_j.$$

The lexicographic ordering on $RLE\_Suffix(S)$ is defined over the order on Σ_{RLE_S}, and our RLE version of suffix arrays [16] is defined based on this order:

Definition 3 (RLE suffix arrays). *For any string S, its* run length encoded suffix array*, denoted RLE_SA_S, is an array of length $n = size(RLE_S)$ such that for any $1 \leq i \leq n$, $RLE\_SA_S[i] = j$ when $RLE_S[j..n]$ is the lexicographically i-th element of $RLE\_Suffix(S)$.*

Let $SparseSuffix(S) = \{val(s) \mid s \in RLE\_Suffix(S)\}$, namely, $SparseSuffix(S)$ is the set of "uncompressed" RLE suffixes of string S. Note that the lexicographic order of $RLE\_Suffix(S)$ represented by RLE_SA_S does not necessarily correspond to the lexicographic order of $SparseSuffix(S)$.

In the running example, $RLE\_SA_S = [5, 3, 1, 7, 4, 2, 6]$. However, the lexicographical order for the elements in $SparseSuffix(S)$ is actually $(7, 1, 3, 5, 6, 2, 4)$.

Lemma 2. *Given RLE_S for any string $S \in \Sigma^*$, RLE_SA_S can be constructed in $O(n \log n)$ time, where $n = size(RLE_S)$.*

Proof. In Appendix.

Let RLE_RANK_S be an array of length $n = size(RLE_S)$ such that

$$RLE\_RANK_S[j] = i \iff RLE\_SA_S[i] = j.$$

Clearly RLE_RANK_S can be computed in $O(n)$ time provided that RLE_SA_S is already computed. To make the notations simpler, in what follows we will denote $rs(h) = RLE\_SA_S[h]$ and $rr(h) = RLE\_RANK_S[h]$ for any $1 \leq h \leq n$.

In our algorithm we will also use an RLE version of *LCP arrays*. For any RLE strings RLE_X and RLE_Y with $val(RLE_X) = X$ and $val(RLE_Y) = Y$, let $lcp(RLE_X, RLE_Y) = lcp(X, Y)$, i.e., $lcp(RLE_X, RLE_Y)$ is the longest prefix of the "uncompressed" strings X and Y.

Definition 4 (RLE LCP array). *For any string S, its* run length encoded longest common prefix array*, denoted RLE_LCP_S, is an array of length $n = size(RLE_S)$ such that*

$$RLE\_LCP_S[i] = \begin{cases} 0 & \text{if } i = 1, \\ |lcp(RLE_S[rs(i-1)..n], RLE_S[rs(i)..n])| & \text{if } 2 \leq i \leq n, \end{cases}$$

In the running example where $RLE\_SA_S = [5, 3, 1, 7, 4, 2, 6]$, $RLE\_LCP_S = [0, 1, 9, 3, 0, 6, 8]$.

Lemma 3. *For any string $S \in \Sigma^*$, given RLE_S and its RLE suffix array RLE_SA_S, RLE_LCP_S can be computed in $O(n)$ time, where $n = size(RLE_S)$.*

Proof. In Appendix.

The following two lemmas imply an interesting and useful property of our data structure; although RLE_SA_S does not necessarily correspond to the lexicographical order of the uncompressed RLE suffixes, adjacent RLE suffixes in RLE_SA_S still share the longest common prefix among all the RLE suffixes.

Lemma 4. *Let i, j be any integers such that $1 \leq i < j \leq n$. For any $j' > j$, $|lcp(RLE_S[rs(i)..n], RLE_S[rs(j)..n])| \geq |lcp(RLE_S[rs(i)..n], RLE_S[rs(j')..n])|$.*

Proof. Let

$$k = \min\{t \mid RLE_S[rs(i)..rs(i)+t-1] \neq RLE_S[rs(j)..rs(j)+t-1]\} \text{ and}$$
$$k' = \min\{t' \mid RLE_S[rs(i)..rs(i)+t'-1] \neq RLE_S[rs(j')..rs(j')+t'-1]\}.$$

Namely, the first $(k-1)$ RL factors of $RLE_S[rs(i)..n]$ and $RLE_S[rs(j)..n]$ coincide and the kth RL factors differ. The same goes for k', $RLE_S[rs(i)..n]$, and $RLE_S[rs(j')..n]$. Since $j' > j$, $k \geq k'$. If $k > k'$, then clearly the lemma holds. If $k = k'$, then $RLE_S[rs(i)+k] \prec RLE_S[rs(j)+k] \preceq RLE_S[rs(j')+k]$. This implies that $|lcp(RLE_S[rs(i)+k], RLE_S[rs(j)+k])| \geq |lcp(RLE_S[rs(i)+k], RLE_S[rs(j')+k])|$. The lemma holds since for these pairs of suffixes, the RL factors after the kth do not contribute to their *lcp*s. □

Lemma 5. *Let i, j be any integers such that $1 \leq i < j \leq n$. For any $i' < i$, $|lcp(RLE_S[rs(i)..n], RLE_S[rs(j)..n])| \geq |lcp(RLE_S[rs(i')..n], RLE_S[rs(j)..n])|$.*

Proof. By a similar argument to Lemma 4. □

3.2 LZ factorization using *RLE_SA*

In what follows we describe our algorithm that computes the s-factorization using RLE_SA_S and RLE_LCP_S. Assume that we have already computed the first $(j-1)$ s-factors $f_1, f_2, \ldots, f_{j-1}$ of string S. Let $\sum_{h=1}^{j-1} |f_h| = \ell - 1$, i.e., the next s-factor f_j begins at position ℓ of S. Let $d = \min\{k \mid \sum_{i=1}^{k}(p_i) \geq \ell\} + 1$, i.e., the $(d-1)$-th RL factor $a_{d-1}^{p_{d-1}}$ contains the occurrence of the ℓ-th character $S[\ell] = a_{d-1}$ of S. Let $q = \sum_{i=1}^{d-1}(p_i) - \ell + 1$, i.e., $S[\ell..N] = a_{d-1}^{q} a_d^{p_d} \cdots a_n^{p_n}$. Note that $1 \leq q \leq p_{d-1}$.

A key idea of our algorithm is that we first search the arrays for the longest previously occurring prefix of $RLE_S[d..n] = a_d^{p_d} \cdots a_n^{p_n}$, rather than for $a_{d-1}^{q} a_d^{p_d} \cdots a_n^{p_n}$ This is because, in the RLE_SA_S, there always exists an entry corresponding to $a_d^{p_d} \cdots a_n^{p_n}$, but there does not necessarily exist one corresponding to $a_{d-1}^{q} a_d^{p_d} \cdots a_n^{p_n}$ (this can happen when $q < p_{d-1}$). To compute f_j, we use the following lemma:

Lemma 6. *If $q = p_{d-1}$ and $a_{d-1} \neq a_i$ for all $1 \leq i \leq d-2$, then $f_j = S[\ell] = a_{d-1}$. Otherwise,*

$$|f_j| = q + \max\{|lcp(RLE_S[x_1..n], RLE_S[d..n])|, |lcp(RLE_S[d..n], RLE_S[x_2..n])|\}$$

where

$$x_1 = \max\{u \mid 1 \leq u < d, rr(u) < rr(d), a_{u-1} = a_{d-1}, p_{u-1} \geq q\} \text{ and}$$
$$x_2 = \min\{v \mid 1 \leq v < d, rr(d) < rr(v), a_{v-1} = a_{d-1}, p_{v-1} \geq q\}.$$

Proof. The case where $q = p_{d-1}$ and $a_{d-1} \neq a_i$ for all $1 \leq i \leq d-2$ is trivial. Otherwise, $|f_j|$ is at least q since $a_{d-1}^{q} = S[\ell..\ell+q-1]$ is a prefix of f_j due to the self-referencing nature of s-factorization. Let $f_j = a_{d-1}^{q} X$. Then X is the longest common prefix of $S[\ell+q..N]$ and $S[h..N]$ for all $1 \leq h \leq \ell+q-1$ *that are immediately preceded by* a_{d-1}^{q}. Consider the sparse RLE suffix array for RLE_SA_S which consists only of the entries t that satisfy $1 < t \leq d$, $a_{t-1} = a_{d-1}$ and $p_{t-1} \geq q$. In this sparse array, x_1 and x_2 are, respectively, the left and right neighbor of the entry corresponding to $RLE_S[d..n]$. Note that $RLE_{S[\ell+q..N]} = a_d^{p_d} \cdots a_n^{p_n} = RLE_S[d..n]$. It follows from Lemmas 4 and 5 that, by computing the (uncompressed) length of the lcp of $RLE_S[d..n]$ and $RLE_S[x_1..n]$, and that of $RLE_S[d..n]$ and $RLE_S[x_2..n]$, we obtain the length of X. □

The main result of this section follows:

Theorem 2. *Given a string S of length N, we can compute the s-factorization of string S in $O(N + n \log n)$ time and extra $O(n)$ space, where $n = size(RLE_S)$.*

Proof. First, compute RLE_S from S in $O(N)$ time, and RLE_SA_S, RLE_RANK_S, and RLE_LCP_S in $O(n \log n)$ time. In the sequel, we show how each s-factor f_j can be computed in $O(\log n)$ time. We compute x_1 and x_2 of Lemma 6 as follows. Recall that we are processing the d-th RL factor. For each character $a \in \Sigma$, we maintain a priority search tree T_a^{d-1} of Theorem 1 for the dynamic set U_a^{d-1} of pairs (x, y) such that $x = rr(e)$ with $1 \leq e < d$ and $a_{e-1} = a$, and $y = p_{e-1}$. We can compute x_1 and x_2 using the priority search tree $T_{a_{d-1}}^{d-1}$ in $O(\log n)$ time as

$$x_1 = \mathit{MaxXInRectangle}(1, rr(d) - 1, q) \text{ and}$$
$$x_2 = \mathit{MinXInRectangle}(rr(d) + 1, n, q).$$

To compute the length of the lcp's, we use another priority search tree L for a static set of pairs $(x, RLE\_LCP_S[x])$ for all $1 < x \leq n$. These values can be computed in $O(n)$ time by Lemma 3, and then L can be constructed in $O(n \log n)$ time by Theorem 1. Then we have

$$|lcp(RLE_S[x_1..n], RLE_S[d..n])| = \mathit{MinYInRange}(rr(x_1) + 1, rr(d)) \text{ and}$$
$$|lcp(RLE_S[d..n], RLE_S[x_2..n])| = \mathit{MinYInRange}(rr(d) + 1, rr(x_2)),$$

and these values can be retrieved in $O(\log n)$ time from L by Theorem 1.

After computing the s-factor f_j, we update the dynamic priority search trees. Namely, if f_j overlaps with RL factors $a_d^{p_d} \cdots a_{d+g}^{p_{d+g}}$, then we insert pair $(rr(d), p_{d-1})$ into $T_{a_{d-1}}^{d-1}$, pair $(rr(d+1), p_d)$ into $T_{a_d}^{d}$, and so on. These insertion operations take $O(g \log n)$ time by Theorem 1, which takes a total of $O(n \log n)$ time for computing all f_j. Hence the total time complexity is $O(N + n \log n)$.

We analyze the space complexity of our data structure. Notice that a collection of sets U_a^{d-1} for all characters $a \in \Sigma$ are pairwise disjoint, and hence $\sum_{a \in \Sigma} |U_a^{d-1}| = d - 1$. By Theorem 1, the overall size of the dynamic priority search trees T_a^{d-1} is $O(n)$ at any stage of $d = 1, 2, \ldots, n$. The size of the static priority search tree L is clearly $O(n)$. Since RLE_SA_S, RLE_RANK_S, and RLE_LCP_S occupy $O(n)$ space each, we conclude that the overall space requirement of our data structure is $O(n)$. □

4 On-line LZ Factorization based on RLE

In this section we present an on-line algorithm that computes s-factorization based on RLE. The term on-line here implies that the RL factors of the input string S of length N are computed from left to right, while computing the s-factors of S simultaneously. Note that transforming the off-line algorithm described in the previous section to an efficient on-line algorithm is not immediately apparent, even if we simulate the suffix array using suffix trees which can be constructed online. This is because the elements inserted into the priority search tree depended on the lexicographic rank of each suffix, which can change dynamically in the on-line setting. To overcome this problem, we consider a different approach based on *directed acyclic word graphs* (*DAWGs*) [3].

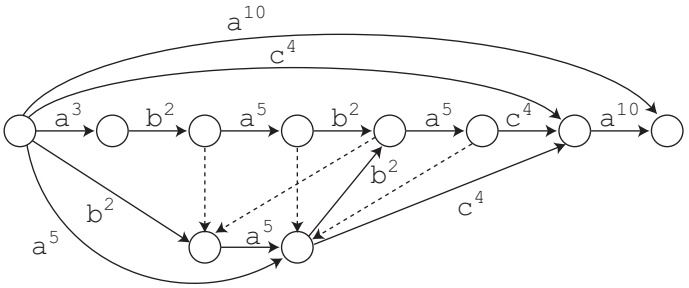

Fig. 1. Illustration for the RLE DAWG of $\mathtt{a}^3\mathtt{b}^2\mathtt{a}^5\mathtt{b}^2\mathtt{a}^5\mathtt{c}^4\mathtt{a}^{10}$. The edges in E are represented by the solid arcs, while the suffix links of some nodes are represented by broken arcs (but their labels are omitted). For simplicity the suffix links of the other nodes are omitted in this figure.

The DAWG of a string S is the smallest automaton that accepts all suffixes of S. Below we introduce an RLE version of DAWGs: We regard RLE_S as a string of length n over alphabet $\Sigma_{RLE_S} = \{RLE_S[i] \mid i = 1, \ldots, n\}$. For any $u \in RLE\_Substr(S)$, let $EndPos_{RLE_S}(u)$ denote the set of positions where an occurrence of u ends in RLE_S, i.e.,

$$EndPos_{RLE_S}(u) = \{j \mid u = RLE_S[i..j], 1 \leq i \leq j \leq n\}.$$

Define an equivalence relation for any $u, w \in RLE\_Substr(S)$ by

$$u \equiv_{RLE_S} w \iff EndPos_{RLE_S}(u) = EndPos_{RLE_S}(w).$$

The equivalence class of $u \in RLE\_Substr(S)$ w.r.t. $\equiv_{RLE_S}$ is denoted by $[u]_{RLE_S}$. When clear from context, we abbreviate the above notations as $EndPos$, $\equiv$ and $[u]$, respectively.

Definition 5. *The* run length encoded DAWG *of a string $S \in \Sigma^*$, denoted by RLE_DAWG_S, is the DAWG of RLE_S over alphabet $\Sigma_{RLE_S} = \{RLE_S[i] \mid i = 1, \ldots, n\}$. Namely, $RLE\_DAWG_S = (V, E)$ where*

$$V = \{[u] \mid u \in RLE\_Substr(S)\},$$
$$E = \{([u], a^p, [ua^p]) \mid u, ua^p \in RLE\_Substr(S), u \not\equiv ua^p\}.$$

We also define the set F of labeled reversed edges, called suffix links, by

$$F = \{([a^p u], a^p, [u]) \mid u \text{ is the longest member of } [u]\}.$$

See also Fig. 1 that illustrates RLE_DAWG_S for $RLE_S = \mathtt{a}^3\mathtt{b}^2\mathtt{a}^5\mathtt{b}^2\mathtt{a}^5\mathtt{c}^4\mathtt{a}^{10}$. Since $EndPos(\mathtt{b}^2\mathtt{a}^5) = EndPos(\mathtt{a}^5) = \{3, 5\}$, $\mathtt{b}^2\mathtt{a}^5$ and $\mathtt{a}^5$ are represented by the same node. On the other hand, $EndPos(\mathtt{a}^3\mathtt{b}^2\mathtt{a}^5) = \{3\}$ and hence $\mathtt{a}^3\mathtt{b}^2\mathtt{a}^5$ is represented by a different node.

Lemma 7. *For any string S of length N, let $size(RLE_S) = n$. RLE_DAWG_S has $O(n)$ nodes and edges, and can be constructed in $O(N + n \log n)$ time and $O(n)$ extra space in an on-line manner, together with the suffix link set F.*

Proof. In Appendix.

For any $u \in RLE\_Substr(S)$, and each character $a \in \Sigma$, let $\mathrm{maxe}_u(a) = \max\{p \mid a^p u \in RLE\_Substr(S)\}$. If there is no such p, let $\mathrm{maxe}_u(a) = 0$. In our on-line algorithm which will follow, we will need to compute $\mathrm{maxe}_u(a)$ efficiently. This can be achieved by the next lemma:

Lemma 8. *$RLE\_DAWG_S = (V, E)$ can be dynamically augmented in a total of $O(n \log n)$ time with $O(n)$ space so that $\mathrm{maxe}_u(a)$ can be computed in $O(\log |\Sigma|)$ time, given any $u \in RLE\_Substr(S)$, its node $[u] \in V$, and any character $a \in \Sigma$.*

Proof. For any $u \in RLE\_Substr(S)$ and character $a \in \Sigma$, consider the following cases: (case 1) u is not the longest member of $[u]$. For any $j \in EndPos(u)$ let $j' = j - |u|$. We have that $RLE_S[j'] = a_{j'}^{p_{j'}}$ where $a_{j'}^{p_{j'}} u \equiv u$, i.e., u is always immediately preceded by $a_{j'}^{p_{j'}}$ in RLE_S. Therefore, $\mathrm{maxe}_u(a) = p_{j'}$ if $a_{j'} = a$ and 0 otherwise. An arbitrary j can be easily determined in $O(1)$ time for each node when the node is first constructed during the on-line construction of RLE_DAWG_S, and does not need to be updated. (case 2) u is the longest member of $[u]$. If there exists $a^p u \in RLE\_Substr(S)$, then there must exist a suffix link $([a^p u], a^p, [u]) \in F$. Therefore $\mathrm{maxe}_u(a)$ is the maximum of the exponent in the labels of all such incoming suffix links, or 0 if there are none. By maintaining a balanced binary search tree at every node $[u]$, we can retrieve this value for any $a \in \Sigma$ in $O(\log |\Sigma|)$ time. It also follows from the on-line construction algorithm of RLE_DAWG_S that the set of labels of incoming suffix links to a node only increases, and we can update this value in $O(\log |\Sigma|)$ time for each new suffix link. Since $|F| = O(n)$, constructing the balanced binary search trees take a total of $O(n \log |\Sigma|)$ time, and the total space requirement is $O(n)$.

It is easy to check whether u is the longest element of $[u]$ in $O(1)$ time by maintaining the length of the longest path to any given node during the on-line construction of RLE_DAWG_S. This completes the proof. □

The next lemma shows how the augmented RLE_DAWG_S can be used to efficiently compute the longest prefix of a given pattern string that appears in string S, which will be a core of our algorithm.

Lemma 9. *For any pattern string $P \in \Sigma^*$, let $RLE_P = b_1^{q_1} b_2^{q_2} \cdots b_m^{q_m}$. We can compute the length of the longest prefix P' of P which occurs in a text string S in $O(size(RLE_{P'}) \log n)$ time, using a data structure of $O(n)$ space, where $n = size(RLE_S)$.*

Proof. We use RLE_DAWG_S. If $m = 0$ (i.e., $P = \varepsilon$), we simply output 0. Let h be the maximum exponent of the labels of the out-going edges of the source

node that are associated with b_1. h can be computed in $O(\log n)$ time using $O(n)$ space. If $m = 1$ or $h < q_1$, then we output $\min\{h, q_1\}$.

Now suppose $m \geq 2$ and $h \geq q_1$. RLE_DAWG_S is traversed with $RLE_P[2..m]$, i.e., starting from the second RL-factor for the same reasons as in Section 3. The occurrences of the traversed factor are checked if it is preceded by $b_1^{q_1}$ using $\mathrm{maxe}_u(b_1)$. A more detailed procedure is shown below, starting from $i = 2$:

1. Let $z = RLE_P[2..i-1]$ and $z' = RLE_P[2..i]$. Check if there is an out-going edge from $[z]$ labeled with $b_i^{q_i}$ to $[z']$, and if so, check if $\mathrm{maxe}_{z'}(b_1) \geq q_1$.
   - If there is no such edge or $\mathrm{maxe}_{z'}(b_1) < q_1$, then go to Line 3.
   - Otherwise, traverse the edge to node $[z']$, and go to Line 2.
2. Check if we have reached the end of RLE_P.
   - If $i < m$, then increment i and go to Line 1.
   - If $i = m$, then P itself occurs in S, and hence output $|P|$.
3. Let $E_v(b_i)$ be the set of out-going edges of node $v = [z]$ labeled by b_i^q with some integer q (note that q is not necessarily equal to q_i). Let $k = \max\{q \mid \mathrm{maxe}_w(b_1) \geq q_1, ([z], b_i^q, [w]) \in E_v(b_i)\}$, that is, k is the maximum exponent of b_i such that $b_1^{q_1} b_2^{q_2} \cdots b_i^k \in RLE\_Substr(S)$. Output $|val(b_1^{q_1} b_2^{q_2} \cdots b_i^{\min\{q_i,k\}})|$ as the result.

We analyze the complexities of the above algorithm. We can find in $O(\log n)$ time the out-going edge that is labeled with $b_i^{q_i}$ from a node. We can retrieve the value of $\mathrm{maxe}_{z'}(b_1)$ in $O(\log |\Sigma|)$ time by Lemma 8. The value of k of Line 3 can be computed in $O(\log n)$ time by maintaining a priority search tree at node $[z]$ for each character $b \in \Sigma$, for the set of pairs (x, y) where the x-coordinate corresponds to the exponent q of the label of edge $([z], b_i^q, [w])$ and the y-coordinate is $\mathrm{maxe}_w(b_1)$. Then $k = MaxXInRectangle(1, |S|, q_1)$. Thus the length of the longest prefix P' of P that occurs in S can be computed in $O(size(RLE_{P'})(\log n + \log |\Sigma|)) = O(size(RLE_{P'}) \log n)$ time. By Theorem 1, and Lemmas 7 and 8, our data structure requires $O(n)$ space. □

Below we give an example for Lemma 9. See Fig. 1 that illustrates RLE_DAWG_S for $RLE_S = \mathtt{a}^3\mathtt{b}^2\mathtt{a}^5\mathtt{b}^2\mathtt{a}^5\mathtt{c}^4\mathtt{a}^{10}$, and consider searching text S for pattern P with $RLE_P = \mathtt{a}^5\mathtt{b}^2\mathtt{a}^7$. We start traversing RLE_DAWG_S with the second RL factor $\mathtt{b}^2$ of P. Since there is an out-going edge of the source node labeled with $\mathtt{b}^2$, we reach node $v = [\mathtt{b}^2]$. There are two suffix links that point to node v, $([\mathtt{a}^3\mathtt{b}^2], \mathtt{a}^3, [\mathtt{b}^2])$ and $([\mathtt{a}^5\mathtt{b}^2], \mathtt{a}^5, [\mathtt{b}^2])$. Hence $\mathrm{maxe}_{\mathtt{b}^2}(\mathtt{a}) = \max\{3, 5\} = 5$, and thus the prefix $\mathtt{a}^5\mathtt{b}^2$ of P occurs in S. We examine whether a longer prefix of P occurs in S by considering the third RL factor $\mathtt{a}^7$. There is no out-going edge from v that is labeled with $\mathtt{a}^7$, hence the longest prefix of P that occurs in S is of the form $\mathtt{a}^5\mathtt{b}^2\mathtt{a}^\ell$ with some $\ell \geq 0$. We consider the set $E_v(\mathtt{a})$ of out-going edges of v that are labeled by $\mathtt{a}^q$ with some q, and obtain $E_v(\mathtt{a}) = \{([\mathtt{b}^2], \mathtt{a}^5, [\mathtt{b}^2\mathtt{a}^5])\}$. We have $\mathrm{maxe}_{\mathtt{b}^2\mathtt{a}^5}(\mathtt{a}) = \max\{3, 5\} = 5$ due to the two suffix links pointing to u. Thus, the longest prefix of P that occurs in S is $\mathtt{a}^5\mathtt{b}^2\mathtt{a}^{\min\{7,5\}} = \mathtt{a}^5\mathtt{b}^2\mathtt{a}^5$.

Theorem 3. *For any string S of length N, there exists an on-line algorithm that computes the s-factorization of S in $O(N + n \log n)$ time and $O(n)$ extra space, where $size(RLE_S) = n$.*

Proof. To describe our s-factorization algorithm, we use a similar assumption as in Section 3: Assume that we have already computed the first $(j-1)$ s-factors $f_1, f_2, \ldots, f_{j-1}$ of string S. Let $\sum_{h=1}^{j-1} |f_h| = \ell - 1$, i.e., the next s-factor f_j begins at position ℓ of S. Let $d = \min\{k \mid \sum_{i=1}^{k}(p_i) \geq \ell\} + 1$, i.e., the $(d-1)$-th RL factor $a_{d-1}^{p_{d-1}}$ contains the occurrence of the ℓ-th character $S[\ell] = a_{d-1}$ of S. Let $q = \sum_{i=1}^{d-1}(p_i) - \ell + 1$, i.e., $S[\ell..N] = a_{d-1}^{q} a_d^{p_d} \cdots a_n^{p_n}$. Note that $1 \leq q \leq p_{d-1}$. In addition, we assume that we have constructed $RLE\_DAWG_S^{d-1}$, where $RLE\_DAWG_S^{d-1}$ denotes the DAWG for $RLE_S[1..d-1] = a_1^{p_1} a_2^{p_2} \cdots a_{d-1}^{p_{d-1}}$.

By definition, the longest prefix Z of $S[\ell..N]$ that occurs in $S[1..\ell-1]$ is a prefix of f_j. By Lemma 9, we can compute Z in $O(size(RLE_Z) \times \log d)$ time. Let v be the node that corresponds to Z in $RLE\_DAWG_S^{d-1}$. We retrieve the ending position e of Z that is stored in v, and then we compute the longest common prefix W of $S[e+1..N]$ and $S[\ell + |Z|..N]$. Then we have $f_j = ZW$. It is clear that W can be computed in $O(|W|)$ time. If f_j overlaps a sequence of g maximal runs of characters in S, then we compute RL factors $a_d^{p_d} \cdots a_{d+g}^{p_{d+g}}$ in $O(|f_j|)$ time, and update $RLE\_DAWG_S^{d-1}$ to $RLE\_DAWG_S^{d+g}$ in amortized $O(g \log n)$ time. The balanced binary search trees of Lemma 8 and the priority search trees of Lemma 9 are updated in $O(\log |\Sigma|)$ time and in $O(\log n)$ time, respectively. Thus we can compute the s-factorization in $O(N + n \log n)$ time, in an on-line manner. $O(n)$ space complexity follows from Lemmas 7, 8 and 9. □

5 Discussion

We proposed off-line and on-line algorithms that compute a well-known variant of LZ factorization, called s-factorization, of a given string S in $O(N + n \log n)$ time using only $O(n)$ extra space, where $N = |S|$ and $n = size(RLE_S)$. The algorithms are more efficient than the previous LZ factorization algorithms when input strings are compressible by RLE.

Our algorithms can be easily extended to other variants of LZ factorization: Let m be the size of the s-factorization *without* self-references of a given string. We modify the on-line algorithm described in Theorem 3, in a way that the longest prefix Z of $S[\ell..N]$ that occurs in $S[1..\ell-1]$ is output as the s-factor f_j. Since Lemma 1 does not hold for s-factorization without self-references, the time complexity of the algorithm is $O(N + (n+m) \log n)$. The working space remains $O(n)$ (excluding the output size). It is trivial that both of our off-line and on-line algorithms can be extended for the *LZ77 factorization* [22] with/without self-references, in the same time and space complexities as the s-factorization.

Our algorithms are based on RLE variants of classical string data structures. It would be interesting to explore whether succinct data structures can be used in combination with our approach to further improve the space efficiency.

References

1. Amir, A., Landau, G.M., Sokol, D.: Inplace run-length 2d compressed search. Theoretical Computer Science 290(3), 1361–1383 (2003)

2. Apostolico, A., Landau, G.M., Skiena, S.: Matching for run-length encoded strings. Journal of Complexity 15(1), 4–16 (1999)
3. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.: The smallest automaton recognizing the subwords of a text. Theoretical Computer Science 40, 31–55 (1985)
4. Chen, K.Y., Chao, K.M.: A fully compressed algorithm for computing the edit distance of run-length encoded strings. Algorithmica (2011)
5. Crochemore, M.: Linear searching for a square in a word. Bulletin of the European Association of Theoretical Computer Science 24, 66–72 (1984)
6. Crochemore, M., Ilie, L., Iliopoulos, C.S., Kubica, M., Rytter, W., Waleń, T.: LPF computation revisited. In: Proc. IWOCA 2009. pp. 158–169 (2009)
7. Crochemore, M., Ilie, L., Smyth, W.F.: A simple algorithm for computing the Lempel Ziv factorization. In: Proc. DCC 2008. pp. 482–488 (2008)
8. Duval, J.P., Kolpakov, R., Kucherov, G., Lecroq, T., Lefebvre, A.: Linear-time computation of local periods. Theoretical Computer Science 326(1-3), 229–240 (2004)
9. Freschi, V., Bogliolo, A.: Longest common subsequence between run-length-encoded strings: a new algorithm with improved parallelism. Information Processing Letters 90(4), 167–173 (2004)
10. Jansson, J., Sadakane, K., Sung, W.K.: Compressed dynamic tries with applications to LZ-compression in sublinear time and space. In: Proc. FSTTCS 2007. pp. 424–435 (2007)
11. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proc. ICALP 2003. pp. 943–955 (2003)
12. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Proc. CPM 2001. pp. 181–192 (2001)
13. Kolpakov, R., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: Proc. FOCS 1999. pp. 596–604 (1999)
14. Liu, J., Wang, Y., Lee, R.: Finding a longest common subsequence between a run-length-encoded string and an uncompressed string. Journal of Complexity 24(2), 173–184 (2008)
15. Mäkinen, V., Ukkonen, E., Navarro, G.: Approximate matching of run-length compressed strings. Algorithmica 35(4), 347–369 (2003)
16. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Computing 22(5), 935–948 (1993)
17. McCreight, E.M.: Priority search trees. SIAM J. Comput. 14(2), 257–276 (1985)
18. Ohlebusch, E., Gog, S.: Lempel-Ziv factorization revisited. In: Proc. CPM'11. pp. 15–26 (2011)
19. Okanohara, D., Sadakane, K.: An online algorithm for finding the longest previous factors. In: Proc. ESA 2008. pp. 696–707 (2008)
20. Starikovskaya, T.: Computing Lempel-Ziv factorization online (2012), arXiv:1202.5233v1
21. Storer, J., Szymanski, T.: Data compression via textual substitution. Journal of the ACM 29(4), 928–951 (1982)
22. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory IT-23(3), 337–343 (1977)
23. Ziv, J., Lempel, A.: Compression of individual sequences via variable-length coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)

Appendix

This appendix provides complete proofs that were omitted due to lack of space.

Lemma 2. *Given RLE_S for any string $S \in \Sigma^*$, RLE_SA_S can be constructed in $O(n \log n)$ time, where $n = size(RLE_S)$.*

Proof. Σ_{RLE_S} consists of $O(n)$ elements, and can be sorted in $O(n \log n)$ time. By identifying each element of Σ_{RLE_S} with its lexicographic rank, we can consider RLE_S as a string of length n over an integer alphabet $\{1, \ldots, n\}$. Then, any linear time suffix array construction algorithm (e.g.: [11]) can be used to construct RLE_SA_S in $O(n)$ time. □

Lemma 3. *For any string $S \in \Sigma^*$, given RLE_S and its RLE suffix array RLE_SA_S, RLE_LCP_S can be computed in $O(n)$ time, where $n = size(RLE_S)$.*

Proof. Our algorithm to construct RLE_LCP_S is analogous to the algorithm of Kasai et al. [12] that computes the LCP array from a given suffix array.

Assume that we have already computed $RLE\_LCP_S[i]$ for some $2 \leq i \leq n$, and that the first h RL factors of $RLE_S[rs(i-1)..n]$ and $RLE_S[rs(i)..n]$ coincide, and the $(h+1)$-th RL factors differ. If $h \geq 2$, then $RLE_S[rs(i-1)+1..n] \prec RLE_S[rs(i)+1..n]$ holds. Therefore, if t is such that $rs(t) = rs(i)+1$, then since the first $(h-1)$ RL factors of $RLE_S[rs(t-1)..n]$ and $RLE_S[rs(t)..n]$ coincide, we can start by comparing their h-th RL factors. If $h \leq 1$, then the relation $RLE_S[rs(i-1)+1..n] \prec RLE_S[rs(i)+1..n]$ may or may not hold. In this case we compute $RLE\_LCP_S[t]$ in a naïve way of comparing the RL factors of $RLE_S[rs(t-1)..n]$ and $RLE_S[rs(t)..n]$ from left to right. By a similar argument to [12], the total number of comparisons is $O(n)$. □

Lemma 7. *For any string S of length N, let $size(RLE_S) = n$. RLE_DAWG_S has $O(n)$ nodes and edges, and can be constructed in $O(N + n \log n)$ time and $O(n)$ extra space in an on-line manner, together with the suffix link set F.*

Proof. The proof is a simple adaptation of the results from [3]. The DAWG of a string of length m has $O(m)$ nodes and edges. Since RLE_DAWG_S is the DAWG of RLE_S of length n, RLE_DAWG_S clearly has $O(n)$ nodes and edges. If σ is the number of distinct characters appearing in S, then the DAWG of a string of length m can be constructed in $O(m \log \sigma)$ time and $O(m)$ space, in an on-line manner, using suffix links. Since $|\Sigma_{RLE_S}| \leq n$ and RLE_S can be computed from S in $O(N)$ time on-line, RLE_DAWG_S with F can be constructed in $O(N + n \log n)$ time and extra $O(n)$ space, on-line. □