

# Computing LZ78-Derivates with Suffix Trees

Dominik Köppl, University of Yamanashi, Kōfu, Japan, [dkppl@yamanashi.ac.jp](mailto:dkppl@yamanashi.ac.jp)

## Abstract

We propose algorithms computing the semi-greedy Lempel–Ziv 78 (LZ78), the Lempel–Ziv Double (LZD), and the Lempel–Ziv–Miller–Wegman (LZMW) factorizations in linear time for integer alphabets. For LZD and LZMW, we additionally propose data structures that can be constructed in linear time, which can solve the substring compression problems for these factorizations in time linear in the output size.

**Keywords:** lossless data compression, factorization algorithms, substring compression

## 1 Introduction

The substring compression problem [5] is to preprocess a given input text  $T$  such that computing a compressed version of a substring of  $T[i..j]$  can be done efficiently. This problem has been originally stated for the Lempel–Ziv-77 (LZ77) factorization, but extensions to the generalized LZ77 factorization [13], the Lempel–Ziv 78 factorization [16], the run-length encoded Burrows–Wheeler transform (RLBWT) [1], and the relative LZ factorization [14, Sect. 7.3] have been studied. Given  $n$  is the length of  $T$ , a trivial solution would be to precompute the compressed output of  $T[\mathcal{I}]$  for all intervals  $\mathcal{I} \subset [1..n]$ . This however gives us already  $\Omega(n^2)$  solutions to compute and store. We therefore strive to find data structures within  $o(n^2)$  words of space, more precisely:  $\mathcal{O}(n \lg n)$  bits of space, that can answer a query in time linear in the output size with a polylogarithmic term on the text length. We investigate variations of the LZ78 factorization, namely LZD [10] and LZMW [20], which have not yet been studied with regard to that aspect. In what follows, we briefly highlight work in the field of substring compression, and then list work related to the LZ78 derivations we study in this paper.

**Substring Compression.** Cormode and Muthukrishnan [5] solved the substring compression problem for LZ77 with a data structure answering the query for  $\mathcal{I}$  in  $\mathcal{O}(z_{\text{SS}[\mathcal{I}]} \lg n \lg \lg n)$  time, where  $z_{\text{SS}[\mathcal{I}]}$  denotes the number of produced LZ77 factors of the queried substring  $T[\mathcal{I}]$ . Their data structure uses  $\mathcal{O}(n \lg^\epsilon n)$  space, and can be constructed in  $\mathcal{O}(n \lg n)$  time. The substring compression problem has also been studied [1] for another compression technique, the RLBWT: Babenko et al. [1] showed how to compute the RLBWT of  $T[\mathcal{I}]$  in  $\mathcal{O}(r_{\text{BWT}[\mathcal{I}]} \lg |\mathcal{I}|)$  time, where  $r_{\text{BWT}[\mathcal{I}]}$  denotes the number of character runs in the BWT of

$T[\mathcal{I}]$ . Recently, Köppl [16] proposed data structures for the substring compression problem with respect to the LZ78 factorization. These data structures can compute the LZ78 factorization of  $T[\mathcal{I}]$  in  $\mathcal{O}(z_{78[\mathcal{I}]})$  time using  $\mathcal{O}(n \lg n)$  bits of space, or in  $\mathcal{O}(z_{78[\mathcal{I}]}(\log_\sigma n + \lg z_{78[\mathcal{I}]}))$  time using  $\mathcal{O}(n \lg \sigma)$  bits of space, where  $z_{78[\mathcal{I}]}$  is the number of computed LZ78 factors, and  $\epsilon \in (0, 1]$  a selectable constant. Finally, Kociumaka et al. [15] studied tools for internal queries that can be constructed in  $\mathcal{O}(n/\log_\sigma n)$  time optimally in the word RAM model, which also led to new complexities for the LZ77 substring compression.

**LZ78 Derivates.** In this paper, we highlight three factorizations deriving from the LZ78 factorization: (a) the flexible parsing variants [12, 18, 19] of LZ78, (b) LZMW [20] and (c) LZD [10]. The first (a) achieves the fewest number of factors among all LZ78 parsings that have additionally the choice to reuse a phrase instead of strictly following the greedy strategy to extend the longest possible factor by one additional character. The last two factorizations (b) and (c) let factors refer to *two* previous factors, and are noteworthy variations of the LZ78 factorization. For instance, LZ78 factorizes  $T = \mathbf{a}^n$  into  $\Theta(\sqrt{n})$  factors, while both variations have  $\Theta(\lg n)$  factors, where the factor lengths scale with a power of two or the Fibonacci sequence for LZD and LZMW, respectively. On the downside, De and Kempa [7, Thm 1.1] have shown that, while random access (i.e., accessing a character of the original input string) is possible in  $\mathcal{O}(\lg \lg n)$  time on LZ78-compressed strings, this is not possible on LZD-compressed strings without increasing the space significantly. With respect to factorization algorithms, Goto et al. [10] can compute LZD in  $\mathcal{O}(n \lg \sigma)$  time with  $\mathcal{O}(n \lg n)$  bits of space, or in  $\mathcal{O}(z \lg n)$  bits of space. LZD and LZMW were studied by Badkobeh et al. [2], who gave a bound of  $\Omega(n^{5/4})$  time for the latter factorization algorithm using only  $\Theta(z \lg n)$  bits of working space, where  $z$  denotes the output size of the respective factorization. Interestingly, the same lower bound holds for the original LZMW algorithm. They also gave Las Vegas algorithms for computing the factorizations in  $\mathcal{O}(n + z \lg^2 n)$  expected time and  $\mathcal{O}(z \lg^2 n)$  space.

**Our Contribution.** In what follows, we propose construction algorithms for the three aforementioned types of factorizations. For LZD and LZMW, we also study their substring compression problem. Regarding these two factorizations, despite having an  $\Omega(n^{5/4})$  time bound on the running time of the known deterministic algorithms, we leverage the data structure of [16] to answer a substring compression query for each of the two factorizations in  $\mathcal{O}(z)$  time using  $\mathcal{O}(n \lg n)$  bits of space, where  $z$  again denotes the number of factors of the corresponding factorization. Since the used data structure can be computed in linear time, this also leads to the first deterministic linear-time computation of LZD and LZMW; the aforementioned Las Vegas algorithm of Badkobeh et al. [2] is only linear *in expectancy* for  $z \in o(n/\lg^2 n)$ <sup>1</sup>. Additionally, we can compute the flexible parse of LZ78 in the same complexities, or in  $\mathcal{O}(n(\lg z + \log_\sigma n))$  time within  $\mathcal{O}(n \lg \sigma)$  bits of space. This holds if we work with the flexible parsing variant of Horspool [12] or Matias and Sahinalp [18]. Best previous results have linear expected time or are only linear for constant alphabet sizes [19].

---

<sup>1</sup>Assuming  $z \in o(n/\lg^2 n)$  is reasonable for relatively compressible strings.

## 2 Preliminaries

With  $\lg$  we denote the logarithm  $\log_2$  to base two. Our computational model is the word RAM model with machine word size  $\Omega(\lg n)$  bits for a given input size  $n$ . Accessing a word costs  $\mathcal{O}(1)$  time.

Let  $T$  be a text of length  $n$  whose characters are drawn from an integer alphabet  $\Sigma = [1..\sigma]$  with  $\sigma \leq n^{\mathcal{O}(1)}$ . Given  $X, Y, Z \in \Sigma^*$  with  $T = XYZ$ , then  $X$ ,  $Y$  and  $Z$  are called a *prefix*, *substring* and *suffix* of  $T$ , respectively. We call  $T[i..]$  the  $i$ -th suffix of  $T$ , and denote a substring  $T[i] \cdots T[j]$  with  $T[i..j]$ . A *parsing dictionary* is a set of strings. A parsing dictionary  $\mathcal{D}$  is called *prefix-closed* if it contains, for each string  $S \in \mathcal{D}$ , all prefixes of  $S$  as well. A *factorization* of  $T$  of size  $z$  partitions  $T$  into  $z$  substrings  $F_1 \cdots F_z = T$ . Each such substring  $F_x$  is called a *factor*.

Stipulating that  $F_0$  is the empty string, a factorization  $F_1 \cdots F_z = T$  is called the *LZ78 factorization* of  $T$  iff, for all  $x \in [1..z]$ , the factor  $F_x$  is the longest prefix of  $T[|F_1 \cdots F_{x-1}| + 1..]$  such that  $F_x = F_y \cdot c$  for some  $y \in [0..x-1]$  and  $c \in \Sigma$ , that is,  $F_x$  is the longest possible previous factor  $F_y$  appended by the following character in the text. The dictionary for computing  $F_x$  is  $\mathcal{D} = \{F_y \cdot c : y \in [0..x-1], c \in \Sigma\}$ . Formally,  $F_x$  starts at  $\text{dst}_x := |F_1 \cdots F_{x-1}| + 1$  and  $y = \text{argmax}\{|F_{y'}| : F_{y'} = T[\text{dst}_x..\text{dst}_x + |F_{y'}| - 1]\}$ . We say that  $y$  and  $F_y$  are the *referred index* and the *referred factor* of the factor  $F_x$ , respectively.

An *interval*  $\mathcal{I} = [b..e]$  is the set of consecutive integers from  $b$  to  $e$ , for  $b \leq e$ . For an interval  $\mathcal{I}$ , we use the notations  $\mathbf{b}(\mathcal{I})$  and  $\mathbf{e}(\mathcal{I})$  to denote the beginning and the end of  $\mathcal{I}$ , i.e.,  $\mathcal{I} = [\mathbf{b}(\mathcal{I})..\mathbf{e}(\mathcal{I})]$ . We write  $|\mathcal{I}|$  to denote the length of  $\mathcal{I}$ ; i.e.,  $|\mathcal{I}| = \mathbf{e}(\mathcal{I}) - \mathbf{b}(\mathcal{I}) + 1$ , and  $T[\mathcal{I}]$  for the substring  $T[b..e]$ .

**Text Data Structures.** Let  $\text{SA}$  denote the *suffix array* [17] of  $T$ . The entry  $\text{SA}[i]$  is the starting position of the  $i$ -th lexicographically smallest suffix such that  $T[\text{SA}[i]..] \prec T[\text{SA}[i+1]..]$  for all integers  $i \in [1..n-1]$ . Let  $\text{ISA}$  of  $T$  be the inverse of  $\text{SA}$ , i.e.,  $\text{ISA}[\text{SA}[i]] = i$  for every  $i \in [1..n]$ . The *LCP array* is an array with the property that  $\text{LCP}[i]$  is the length of the longest common prefix (LCP) of  $T[\text{SA}[i]..]$  and  $T[\text{SA}[i-1]..]$  for every  $i \in [2..n]$ . For convenience, we stipulate that  $\text{LCP}[1] := 0$ . The array  $\Phi$  is defined as  $\Phi[i] := \text{SA}[\text{ISA}[i] - 1]$ , and  $\Phi[i] := n$  in case that  $\text{ISA}[i] = 1$ . The *permuted LCP (PLCP) array*  $\text{PLCP}$  stores the entries of  $\text{LCP}$  in text order, i.e.,  $\text{PLCP}[\text{SA}[i]] = \text{LCP}[i]$ . Given an integer array  $Z$  of length  $n$  and an interval  $[i..j] \subset [1..n]$ , the range minimum query  $\text{RmQ}_Z(i, j)$  (resp. the range maximum query  $\text{RMQ}_Z(i, j)$ ) asks for the index  $p$  of a minimum element (resp. a maximum element) of the subarray  $Z[i..j]$ , i.e.,  $p \in \text{arg min}_{i \leq k \leq j} Z[k]$ , or respectively  $p \in \text{arg max}_{i \leq k \leq j} Z[k]$ . We use the following data structure to handle these kind of queries:

**Lemma 1** ([6]). Given an integer array  $Z$  of length  $n$ , there is an  $\text{RmQ}$  (resp.  $\text{RMQ}$ ) data structure taking  $2n + o(n)$  bits of space that can answer an  $\text{RmQ}$  (resp.  $\text{RMQ}$ ) query on  $Z$  in constant time. This data structure can be constructed in  $\mathcal{O}(n)$  time with  $o(n)$  bits of additional working space.

An *LCE (longest common extension)* query  $\text{lce}(i, j)$  asks for the longest common prefix of two suffixes  $T[i..]$  and  $T[j..]$ . We can answer LCE queries in  $\mathcal{O}(1)$ -query time with an

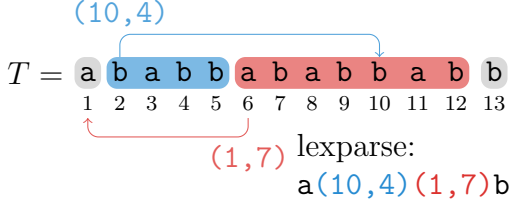


Figure 1: lexparse of our running example  $T = \text{ababbababb}$ , given by  $F_1 = a$ ,  $F_2 = \text{babb} = T[10..13]$ ,  $F_3 = \text{ababbab} = T[1..7]$ , and  $F_4 = b$ . Non-literal factors are encoded by text position and length of the substring in  $T$  they refer to.

RmQ data structure built on the LCP array because  $\text{lce}(i, j)$  is the LCP value of the RmQ in the range  $[\text{ISA}[i] + 1.. \text{ISA}[j]]$ .

As a warm-up we start with the substring compression for lexparse.

### 3 A Warm-Up: Lexparse

The *lex-parse* [22, Def. 11] is a factorization  $T = F_1 \cdots F_v$  such that  $F_x = T[\text{dst}_x.. \text{dst}_x + \ell_x - 1]$  with  $\text{dst}_1 = 1$  and  $\text{dst}_{x+1} = \text{dst}_x + \ell_x$  if  $\ell_x := \text{PLCP}[\text{dst}_x] \geq 1$ , or  $F_x$  is a *literal factor* with  $\ell_x := |F_x| = 1$  otherwise. For  $\text{PLCP}[\text{dst}_x] \geq 1$ , the reference of  $F_x$  is  $\Phi(F_x)$  since by definition of the PLCP array  $T[\text{dst}_x.. \text{dst}_x + \ell_x - 1] = T[\Phi(\text{dst}_x).. \Phi(\text{dst}_x) + \ell_x - 1]$ . See Fig. 1 for an example.

For computation, it actually suffices to have sequential access to  $\Phi$  and random access to the text. That is because we can naively compute  $\text{PLCP}[\text{dst}] = \text{lce}(\text{dst}, \Phi(\text{dst}))$  in  $\mathcal{O}(\text{PLCP}[\text{dst}])$  time by linearly scanning the character pairs in the text. The number of scanned character pairs sum up to at most  $2n$ . This however scales linearly in the interval length. Here, our idea is to use LCE queries to speed up the factor-length computation. To find the reference position, i.e., the position from where we want to compute the LCE queries, we use the following data structure.

**Theorem 2** ([1, Theorem 3.8]). There is a data structure that can give, for an interval  $\mathcal{I}$ , the  $k$ -th smallest suffix of  $T[\mathcal{I}]$  or the rank of a suffix of  $T[\mathcal{I}]$ , each in  $\mathcal{O}(\lg s)$  time, where  $s = |\mathcal{I}|$ . It can be constructed in  $\mathcal{O}(n\sqrt{\lg n})$  time, and takes  $\mathcal{O}(n \lg n)$  bits of space.

Having the data structure of Thm. 2 and  $\mathcal{O}(1)$ -time support for LCE queries, we can compute the substring compression variant of *lex-parse* for a given query interval  $\mathcal{I}$  with the following instructions in  $\mathcal{O}(v_{\mathcal{I}} \lg s)$  time, where  $s = |\mathcal{I}|$  and  $v_{\mathcal{I}}$  is the number of factors of the *lex-parse* factorization of  $T[\mathcal{I}]$ .

**Algorithm.** Start at text position  $\mathbf{b}(\mathcal{I})$ , and query the rank of the substring suffix  $T[\mathbf{b}(\mathcal{I}).. \mathbf{e}(\mathcal{I})]$ . Given this rank is  $k$ , select the  $(k - 1)$ -st substring suffix of  $T[\mathcal{I}]$ . Say this suffix starts at position  $j$ , then  $j$  is the reference of the first factor. Next, compute  $\ell \leftarrow \text{lce}(\mathbf{b}(\mathcal{I}), j)$  to determine the factor length  $\ell$ . On the one hand, if the computed factor protrudes the substring  $T[\mathcal{I}]$  with  $\mathbf{b}(\mathcal{I}) + \ell - 1 > \mathbf{e}(\mathcal{I})$ , trim  $\ell \leftarrow \mathbf{e}(\mathcal{I}) - \mathbf{b}(\mathcal{I}) + 1$ . On the other hand, if  $k = 1$  or  $\ell = 0$ , then the factor is literal. Anyway, we have computed the first factor, and recurse on the interval  $[\mathbf{b}(\mathcal{I}) + \ell.. \mathbf{e}(\mathcal{I})]$  to compute the next factor while keeping the query interval  $\mathcal{I}$  fixed for the data structure of Thm. 2. Each recursion step in

the algorithm uses a select and a rank query on the data structure of Thm. 2, which takes  $\mathcal{O}(\lg s)$  time each.

## 4 Semi-Greedy Parsing

The semi-greedy parsing [11] is a variation of the LZ77 parsing in that a factor  $F_x$  of the factorization  $F_1 \cdots F_z$  starting at text position  $\mathbf{dst}_x := |F_1 \cdots F_{x-1}|$  does not necessarily have to be the longest prefix of  $T[\mathbf{dst}_x..]$  that has an occurrence starting before  $\mathbf{dst}_x$ . Given that longest prefix has length  $\ell$ , the semi-greedy parsing instead selects the length  $\ell' \in [1..\ell]$  that maximizes the next factor length  $|F_{x+1}|$  like it would have been selected by the standard greedy parsing continuing at  $T[\mathbf{dst}_x + \ell'..]$ .

**Semi-Greedy LZW.** Horspool [12] proposed adaptations of the semi-greedy parsing for Lempel–Ziv–Welch (LZW), a variation of the LZ78 factorization. These LZ78-based semi-greedy parsings have been studied by Matias and Sahinalp [18], who further generalized the semi-greedy parsing to other parsings, and coined the name *flexible parsing* for this parsing strategy. They also showed that the flexible parsing variant of parsings using prefix-closed dictionaries is optimal with respect to the minimal number of factors. A set of strings  $\mathcal{S} \subset \Sigma^+$  is called *prefix-closed* if every non-empty prefix of every element of  $\mathcal{S}$  is itself an element of  $\mathcal{S}$ . A follow-up [19] presents practical compression improvements as well as algorithmic aspects in how to compute the flexible parsing of LZW, called LZW-FP, or the variant of Horspool [12], which they call FPA. The main difference between FPA and LZW-FP is that LZW-FP uses the parsing dictionary of the standard LZW factorization, while the factors in FPA refer to already computed FPA factors. This makes the computation of LZW-FP easier since we can statically build the LZW parsing dictionary with a linear-time LZW construction algorithm like [8].

**LZ78 version of LZW-FP.** We first describe the task in LZ78-factorization terminology as follows (the adaptation to LZW is straightforward). Briefly speaking, we apply the flexible parsing on the LZ78 factorization without changing the LZ78 dictionary at any time. Thus, for computing a factor  $F_x$ , we restrict us to refer to any LZ78 factor that ends prior to the start of  $F_x$ . The rationale of this factorization is that the decompression works by building the LZ78 dictionary while decoding the factors of the flexible parsing. For a formal definition, assume that the LZ78 factorization of  $T$  is  $T = G_1 \cdots G_{z_{78}}$ . Given  $G_0$  denotes the empty string, we write  $\mathbf{e}(G_0) = 0$  and  $\mathbf{e}(G_x) := \sum_{y=1}^{y=x} |G_y|$  for the ending position of  $G_x$  in  $T$ . Now suppose we have parsed a prefix  $T[1..i-1] = F_1 \cdots F_{x-1}$  with the LZ78 variant of LZW-FP and want to compute  $F_x$ . At that time point, the dictionary  $\mathcal{D}$  consists of all LZ78 factors that end before  $\mathbf{dst}_x$ , appended by any character in  $\Sigma$ . Instead of greedily selecting the longest match in  $\mathcal{D}$  for  $F_x$  as LZ78 would do, we select the length  $|F_x|$  for which  $F_{x+1}$  would be longest (ties are broken in favor of a longer  $F_x$ ). Note that at the time for selecting  $F_{x+1}$ , we can choose among all LZ78 factors that end before  $\mathbf{dst}_{x+1}$ , which depends on  $|F_x|$ . Formally, let  $w$  be the index of the longest LZ78 factor  $G_w$  that is a prefix of the starting position  $T[\mathbf{dst}_x..]$  of  $F_x$ , i.e.,  $w = \operatorname{argmax}\{|G_{y'}| : G_{y'} = T[\mathbf{dst}_x..\mathbf{dst}_x + |G_{y'}| - 1] \wedge \mathbf{e}(G_{y'}) < \mathbf{dst}_x\}$ . Then we determine in the interval  $\mathcal{J} := [\mathbf{dst}_x + 1..\mathbf{dst}_x + |G_w|]$  the starting position

of  $F_{x+1}$  by the flexible parsing, i.e., we select the position  $\ell \in \mathcal{J}$  for which the value  $\max\{|G_{y'}| : G_{y'} = T[\ell..\ell + |G_{y'}| - 1] \wedge e(G_{y'}) < \ell\}$  is maximized. Then  $F_x$  has length  $\ell - \text{dst}_x$ ; if  $|F_x| \geq 2$ , then its reference exists since LZ78 is prefix-closed. By doing so, we obtain the LZ78 version of LZW-FP.

For actually computing this factorization, let us stipulate that we can compute the longest possible factor starting in  $\mathcal{J}$  for each text position by a lookup in  $\mathcal{D}$  within  $t_{\mathcal{D}}$  time.<sup>2</sup> Matias et al. [19, Section 3] achieved  $t_{\mathcal{D}} = \mathcal{O}(1)$  expected time per text position by storing Karp–Rabin fingerprints in a hash table to match considered substrings with the parsing dictionary that represents its elements via fingerprints. Alternatively, they get constant time for constant-sized alphabets with two tries, where one trie stores the dictionary, and the other the reversed strings of the dictionary. Despite claimed [18] that the flexible parsing of LZW-FP can be computed in linear time, no algorithmic details have been given.

**LZ78 version of FPA.** The other semi-greedy variant we study in its LZ78 version is the one of FPA, which we call LZ78-FP. Compared to LZW-FP the dictionary  $\mathcal{D}$  is now based on the computed LZ78-FP factors instead of the LZ78 factors. Given  $\text{dst}_x$  denotes the starting position of the factor  $F_x$ , which we want to compute, let  $y := \text{argmax}\{|F_{y'}| : F_{y'} = T[\text{dst}_x..\text{dst}_x + |F_{y'}| - 1]\}$ . Then our task is to select the length  $|F_x| \in [1..|F_y| + 1]$  such that a factor starting at  $\text{dst}_x + |F_x|$  is longest among all such possible lengths  $|F_x|$ . Given that  $\mathcal{D}$  stores all selectable candidate LZ78-FP factors (concatenated by any character), finding the longest match with each suffix  $T[p..]$  for  $p \in [i..i + |F_y| - 1]$  can be achieved by querying  $\mathcal{D}$ . Assuming this takes  $t_{\mathcal{D}}$  time per suffix, we need  $\mathcal{O}(nt_{\mathcal{D}})$  time since we query  $\mathcal{D}$  for each text position  $p$ .

**Example 3.** Select an LZ78-incompressible string  $S$  of length  $n$  on the alphabet  $\{\mathbf{b}, \mathbf{c}\}$  such that the number of LZ78 factors of  $S$  is  $\Theta(n/\lg n)$ . Suppose we parse the string  $T = \mathbf{a} \cdot \mathbf{aS}[1] \cdot \mathbf{aS}[1..2] \cdot \mathbf{aS}[1..3] \cdots \mathbf{aS}[1..n] \cdot \mathbf{aS}[1..n] \cdot \mathbf{a} \cdot S[1..n]$ . On the one hand, the LZ78 factorization would create the factors  $G_1 = \mathbf{a}$ ,  $G_2 = \mathbf{aS}[1]$ ,  $G_3 = \mathbf{aS}[1..2]$ ,  $\dots$ ,  $G_{n+1} = \mathbf{aS}[1..n]$ ,  $G_{n+2} = \mathbf{aS}[1..n]\mathbf{a}$ , and finally create the  $\Theta(n/\lg n)$  factors for factorizing  $S$ . On the other hand, both flexible parsing variants have  $F_x = G_x$  for all  $x \in [1..n+1]$ , but select  $F_{n+2} = G_{n+1} = F_n S[n]$  because doing so maximizes the length of  $F_{n+3} = \mathbf{aS}[1..n] = G_{n+1} = F_n S[n]$ . While the flexible parses have thus  $n+3$  factors, the LZ78 factorization has  $n+2 + \Theta(n/\lg n)$  many, and thus can produce considerably more factors.

**Suffix Tree Superimposition.** The algorithms we propose in the following are based on the suffix tree superimposition with the LZ trie of [8], which we briefly review. The LZ trie represents each LZ factor as a node (the root represents the factor  $F_0$ ). The node representing the factor  $F_y$  has a child representing the factor  $F_x$  connected with an edge labeled by a character  $c \in \Sigma$  if and only if  $F_x = F_y c$ . An observation of Nakashima et al. [21, Sect. 3] is that the LZ trie is a connected subgraph of the suffix trie containing its root. That

---

<sup>2</sup>To perform that computation for each text position only once, we keep the information gathered for the range  $[\text{dst}_{x+1}..\text{dst}_x + |G_w|] \subset \mathcal{J}$  in memory such that we can skip the recomputation of the longest match when searching for the length of  $F_{x+1}$ .

is because the LZ78 parsing dictionary (or the dictionary of any variant of our semi-greedy parsings) is prefix-closed. We can therefore simulate the LZ trie by marking nodes in the suffix trie. Since the suffix trie has  $\mathcal{O}(n^2)$  nodes, we use the suffix tree **ST** instead of the suffix trie to save space. In **ST**, however, not every LZ trie node is represented; these implicit LZ trie nodes are on the **ST** edges between two **ST** nodes. Since the LZ trie is a connected subgraph of the suffix trie sharing the root node, implicit LZ trie nodes on the same **ST** edge have the property that they are all consecutive and that the first one starts at the first character of the edge. To represent all implicit LZ trie nodes, it suffices to augment each **ST** edge with a counter counting the number of its implicit LZ trie nodes. We call this counter an *exploration counter*, and we write  $n_v \in [0..c(e)]$  for the exploration counter of an edge  $e = (u, v)$ , which is stored in the lower node  $v$  that  $e$  connects to. Additionally, we call an **ST** node  $v$  an *edge witness* if  $n_v$  becomes incremented during the factorization. We say that  $n_v$  is *full* if  $n_v$  is the length of the string label of the edge connecting to  $v$ , meaning that  $v$  is an explicit LZ trie node. The *string label* of  $v$  is the concatenation of the edge labels read when traversing from the root to  $v$ . The *string depth* of  $v$  is the length of its string label. Our **ST** representations can compute the string depth in  $\mathcal{O}(t_{\text{SA}})$  time, where  $t_{\text{SA}}$  is  $\mathcal{O}(1)$  or  $\mathcal{O}(\log_{\sigma} n)$  if we allow **ST** to take  $\mathcal{O}(n \lg n)$  bits or  $\mathcal{O}(n \lg \sigma)$  bits of space, respectively. We additionally stipulate that the root of **ST** is an edge witness, whose exploration counter is always full. Then all edge witnesses form a sub-graph of **ST** sharing the root node. See Fig. 2 for an example. Finally, Fischer et al. [8, Sect. 4.1] gave an  $\mathcal{O}(n)$ -bits representation of all exploration counters, which however builds on the fact that the parse dictionary is prefix-closed.

**Linear-Time Computation.** To obtain an  $\mathcal{O}(nt_{\text{SA}})$ -time deterministic solution for both variants, we make use of the superimposition of **ST**. For **LZW-FP**, compute the LZ78 factorization as described in [8, Sect. 4] such that all edge witnesses are marked. The marking is done with the following data structure.

**Lemma 4** ([4]). There is a semi-dynamic lowest common ancestor data structure that can (a) find the lowest marked node of a leaf or (b) mark a specific node, both in constant time. We can augment **ST** with this data structure in  $\mathcal{O}(n)$  time using  $\mathcal{O}(n \lg n)$  bits of space.

For **FPA**, we mark the nodes dynamically. Then we can query, for each suffix  $T[p..]$  the lowest marked ancestor  $w$  of the leaf with suffix number  $p$ . The string depth of  $w$ , obtained in  $\mathcal{O}(t_{\text{SA}})$  time, is the length of the longest factor being a prefix of  $T[p..]$ .

If we allow  $\mathcal{O}(t_{\text{SA}} \lg z)$  time per text position, we can omit the lowest marked ancestor data structure and make use of exponential search on **ST**, as described in [16, Section 4]. Since  $|F_x| \leq x$ , finding  $F_x$  involves  $\mathcal{O}(\lg x)$  node visits, and for each of them we pay  $\mathcal{O}(t_{\text{SA}})$  time for computing its string length. The total time is  $\mathcal{O}(nt_{\text{SA}} \lg z)$ . Since the LZ78-dictionary as well as the **FPA** dictionary are prefix-closed, the **ST** superimposition by the LZ trie can be represented in  $\mathcal{O}(n)$  bits, and thus we need only  $\mathcal{O}(n \lg \sigma)$  bits overall.

**Theorem 5.** We can compute **LZW-FP** or **FPA** in  $\mathcal{O}(n)$  time with  $\mathcal{O}(n \lg n)$  bits of space, or in  $\mathcal{O}(nt_{\text{SA}} \lg z)$  time with  $\mathcal{O}(n \lg \sigma)$  bits of space, where  $z$  is the number of factors of the LZ78 or **FPA** factorization for computing **LZW-FP** or **FPA**, respectively.

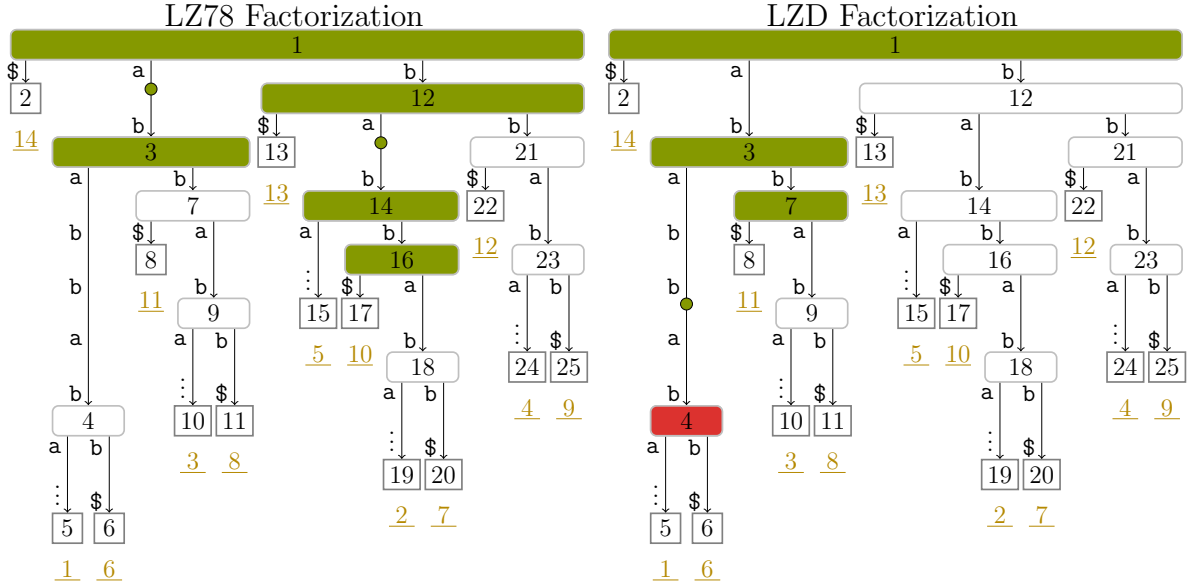


Figure 2: Suffix tree superimposition by the LZ trie of the LZ78 (left) and the LZD (right) factorization on our running example. Small circles in green (■) on edges are implicit LZ trie nodes. The ST nodes colored in green are full. There is only one edge witness, which has label 4, in the LZD factorization, and it is colored in red (■). The LZD factorization is not prefix-closed, and hence the nodes marked in green are not connected in the suffix trie.

## 5 LZ78 Factorization Variants

In what follows, we show applications of the technique of Thm. 5 for variants of the LZ78 factorization, and also give solutions to their substring compression variants.

**LZD.** LZD [10] is a variation of the LZ78 factorization. A factorization  $F_1 \cdots F_z$  of  $T$  is LZD if  $F_x = G_1 \cdot G_2$  with  $G_1, G_2 \in \{F_1, \dots, F_{x-1}\} \cup \Sigma$  such that  $G_1$  and  $G_2$  are respectively the longest possible prefixes of  $T[\text{dst}_x..]$  and of  $T[\text{dst}_x + |G_1|..]$ , where  $\text{dst}_x$  denotes the starting position of  $F_x$ . When computing  $F_x$ , the LZD dictionary stores the phrases  $\{F_y \cdot F_{y'} : y, y' \in [1..x-1]\} \cup \{F_y c : y \in [0..x-1], c \in \Sigma\}$ . See Fig. 3 for an example of the LZD factorization.

A computational problem for LZD and LZMW is that their dictionaries are not prefix-

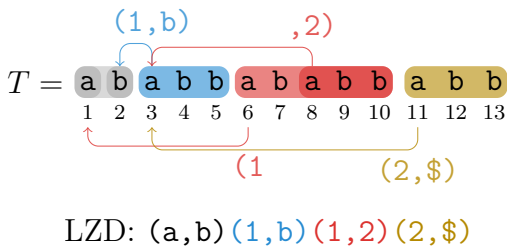


Figure 3: LZD factorization of our running example, given by  $F_1 = ab$ ,  $F_2 = abb = F_1 \cdot b$ ,  $F_3 = ababb = F_1 \cdot F_2$ , and  $F_4 = abb\$ = F_2 \cdot \$$ . The factor  $F_3$  has two referred indexes, which are visualized by two arrows pointing to the referred factors. We encode  $F_4$  with the artificial character  $\$$  to denote that we ran out of characters.



closed in general. Consequently, their LZ tries are not necessarily connected subgraphs of the suffix trie: Indeed, some nodes in the suffix trie can be "jumped over" (cf. Fig. 2). Unfortunately, a requirement of the  $\mathcal{O}(n)$ -bits representation of the LZ trie built upon ST is that the LZ trie is prefix-closed [8], which is not the case in the following two variations. In that light, we give up compact data structures and let each node store its exploration counter explicitly. Another negative consequence is that the search for the longest reference cannot be performed by the exponential search used in Sect. 4. It thus looks like that using the dynamic marked ancestor data structure of Lemma 4 is the only efficient way for that task.

Finally, given we just have computed the factor  $F_x = F_w \cdot F_y$ , we need to find the edge witness  $v$  of  $F_x$  and mark it. For that, we search the locus of  $F_x$ , which is either represented by  $v$ , or is on an edge to  $v$ . To find this locus, it suffices to traverse the path from the root to the leaf with suffix number  $\text{dst}_x$ . The number of visited nodes is upper bounded by the factor length  $|F_x|$ , and thus we traverse  $\mathcal{O}(n)$  nodes in total. For the substring compression problem, we want to get rid of the linear dependency in the text length. For that, we jump to the locus with the following data structure:

**Lemma 6** ([3, 9]). There exists a weighted ancestor data structure for ST, which supports, given a leaf  $\ell$  and integer  $d$ , constant-time access to the ancestor of  $\ell$  with string depth  $d$ . It can be constructed in linear time using  $\mathcal{O}(n \lg n)$  bits of space.

**Theorem 7.** There is a data structure that, given a query interval  $\mathcal{I}$ , can compute LZD in  $\mathcal{O}(z_{\text{D}[\mathcal{I}]})$  time, where  $z_{\text{D}[\mathcal{I}]}$  is the number of LZD factors. This data structure can be constructed in  $\mathcal{O}(n)$  time, using  $\mathcal{O}(n \lg n)$  bits of working space.

*Proof.* For each factor  $F_x$ , compute first the lowest marked ancestor  $v$  of the ST leaf  $\lambda$  with suffix number  $\text{dst}_x$ . Say the exploration counter of  $v$  plus the string depth of  $v$ 's parent is  $\ell_v$ , compute the lowest marked ancestor  $w$  of the leaf with suffix number  $\text{dst}_x + \ell_v$ . Given the exploration counter of  $w$  combined with the string depth of  $w$ 's parent is  $\ell_w$ ,  $F_x$  has length  $\ell_v + \ell_w$ , and its locus is on the path between  $v$  and  $\lambda$  on the string depth  $\ell_v + \ell_w$ . After finding the highest node  $u$  that is this locus or below of it, we mark  $u$ , increase its exploration counter, and continue with processing  $F_{x+1}$ .

In total, for each factor  $F_x$ , we use two lowest marked ancestor queries, a weighted ancestor query, and possibly mark a node or increase its exploration counter. Each step takes constant time.  $\square$

**LZMW.** The factorization  $T = F_1 \cdots F_z$  is the LZMW parsing of  $T$  if, for every  $x \in [1..z]$ ,  $F_x$  is the longest prefix of  $T[\text{dst}_x..]$  with  $F_x \in \{F_{y-1}F_y : y \in [2..\text{dst}_x - 1]\} \cup \Sigma$  where  $\text{dst}_x = 1 + \sum_{y=1}^{x-1} |F_y|$ . The LZMW dictionary for computing  $F_x$  is the set of strings  $\bigcup_{y \in [2..x-1]} (F_{y-1} \cdot F_y) \cup \Sigma$ . See Fig. 4 for an example of the LZMW factorization.

**Theorem 8.** There is a data structure that, given a query interval  $\mathcal{I}$ , can compute LZMW in  $\mathcal{O}(z_{\text{MW}[\mathcal{I}]})$  time, where  $z_{\text{MW}[\mathcal{I}]}$  is the number of LZMW factors. This data structure can be constructed in  $\mathcal{O}(n)$  time, using  $\mathcal{O}(n \lg n)$  bits of working space.

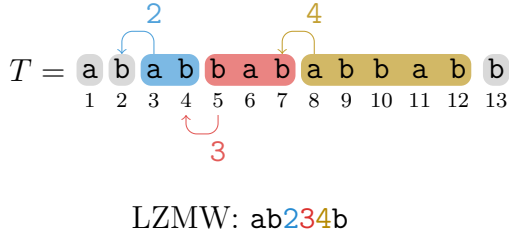


Figure 4: LZMW factorization of our running example  $T = ababbababbabb$ , given by  $F_1 = a$ ,  $F_2 = b$ ,  $F_3 = ab = F_1F_2$ ,  $F_4 = bab = F_2F_3$ ,  $F_5 = abbab = F_3F_4$ , and  $F_6 = b$ . We encode the factors such that a number denotes the index of a referred factor. Writing  $x$  means that we refer to the string  $F_{x-1}F_x$ .

*Proof.* For each factor  $F_x$ , compute the lowest marked ancestor  $v$  of the ST leaf with suffix number  $\text{dst}_x$ . The sum of the exploration counter of  $v$  with the string depth of its parent is the length of  $F_x$ . Finally, we mark the locus of  $F_{x-1}F_x$  in ST. The difference to LZD is that (a) we mark the locus of  $F_{x-1} \cdot F_x$  instead of  $F_x$  and that (b) we have one lowest marked ancestor query instead of two per factor.  $\square$

**Acknowledgements** This research was supported by JSPS KAKENHI with grant numbers JP21K17701 and JP23H04378.

## References

- [1] M. A. Babenko, P. Gawrychowski, T. Kociumaka, and T. Starikovskaya. Wavelet trees meet suffix trees. In *Proc. SODA*, pages 572–591, 2015.
- [2] G. Badkobeh, T. Gagie, S. Inenaga, T. Kociumaka, D. Kosolobov, and S. J. Puglisi. On two LZ78-style grammars: Compression bounds and compressed-space computation. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 51–67, 2017.
- [3] D. Belazzougui, D. Kosolobov, S. J. Puglisi, and R. Raman. Weighted ancestors in suffix trees revisited. In *Proc. CPM*, volume 191 of *LIPICs*, pages 8:1–8:15, 2021.
- [4] R. Cole and R. Hariharan. Dynamic LCA queries on trees. *J. Comput.*, 34(4):894–923, 2005.
- [5] G. Cormode and S. Muthukrishnan. Substring compression problems. In *Proc. SODA*, pages 321–330, 2005.
- [6] P. Davoodi, R. Raman, and S. R. Satti. Succinct representations of binary trees for range minimum queries. In *Proc. COCOON*, volume 7434 of *LNCS*, pages 396–407, 2012.
- [7] R. De and D. Kempa. Grammar boosting: A new technique for proving lower bounds for computation over compressed data. In *Proc. SODA*, pages 3376–3392, 2024.
- [8] J. Fischer, T. I. D. Köppl, and K. Sadakane. Lempel–Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018.
- [9] P. Gawrychowski, M. Lewenstein, and P. K. Nicholson. Weighted ancestors in suffix trees. In *Proc. ESA*, volume 8737 of *LNCS*, pages 455–466, 2014.
- [10] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. LZD factorization: Simple and practical online grammar compression with variable-to-fixed encoding. In *Proc. CPM*, volume 9133 of *LNCS*, pages 219–230, 2015.
- [11] A. Hartman and M. Rodeh. Optimal parsing of strings. In *Combinatorial Algorithms on Words*, pages 155–167, 1985.
- [12] R. N. Horspool. The effect of non-greedy parsing in Ziv–Lempel compression methods. In *Proc. DCC*, pages 302–311, 1995.
- [13] O. Keller, T. Kopelowitz, S. L. Feibish, and M. Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:42–54, 2014.

- [14] T. Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, 2018.
- [15] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. Internal pattern matching queries in a text and applications. *CoRR*, abs/1311.6235v5, 2023.
- [16] D. Köppl. Non-overlapping LZ77 factorization and LZ78 substring compression queries with suffix trees. *Algorithms*, 14(2)(44):1–21, 2021.
- [17] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
- [18] Y. Matias and S. C. Sahinalp. On the optimality of parsing in dynamic dictionary based data compression. In *Proc. SODA*, pages 943–944, 1999.
- [19] Y. Matias, N. M. Rajpoot, and S. C. Sahinalp. The effect of flexible parsing for dynamic dictionary-based data compression. *ACM J. Exp. Algorithmics*, 6:10, 2001.
- [20] V. S. Miller and M. N. Wegman. Variations on a theme by Ziv and Lempel. In *Combinatorial Algorithms on Words*, pages 131–140, 1985.
- [21] Y. Nakashima, T. I. S. Inenaga, H. Bannai, and M. Takeda. Constructing LZ78 tries and position heaps in linear time for large alphabets. *Inf. Process. Lett.*, 115(9):655–659, 2015.
- [22] G. Navarro, C. Ochoa, and N. Prezza. On the approximation ratio of ordered parsings. *IEEE Trans. Inf. Theory*, 67(2):1008–1026, 2021.