Substring Compression Variations and LZ78-Derivates*

Dominik Köppl

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. For substring compression, we give the first results for lexparse and closed factorizations.

Keywords: lossless data compression, factorization algorithms, substring compression

1 Introduction

The substring compression problem [26] is to preprocess a given input text T such that computing a compressed version of a substring $T[i..j]$ can be performed efficiently. This problem has originally been stated for the Lempel–Ziv-77 (LZ77) factorization [80], but extensions to the generalized LZ77 factorization [53], the Lempel–Ziv 78 factorization [61], the run-length encoded Burrows–Wheeler transform (RLBWT) [8], and the relative LZ factorization [57, Sect. 7.3] have been studied. Given n is the length of T, a trivial solution is 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 [44] and LZMW [70], which have not yet been studied in this regard.

1.1 Related Work

In what follows, we briefly highlight work in the field of substring compression, and then list work related to the LZ78 derivations that we study in this paper.

Substring Compression. Cormode and Muthukrishnan [26] solved the substring compression problem for LZ77 with a data structure answering the query for $\mathcal{I}$ in $\mathcal{O}(z_{77[\mathcal{I}]} \lg n \lg \lg n)$

*Parts of this work have already been presented at the Data Compression Conference 2024 [63].

Table 1: Results on various types of substring compression problems. For a given query interval $\mathcal{I}$, z denotes the output size (e.g., the number of factors) and $s = |\mathcal{I}|$ the length of the query interval. For generalized Lempel–Ziv (gLZ) we have a second query interval $\mathcal{J}$ with length s'. e denotes the number of edges of the compressed acyclic word graph (CDAWG) [20]. $\epsilon \in (0, 1]$ is a selectable constant. Space is given in words, where we assume the computation model to be a RAM with word size $\Omega(\lg n)$ bits. The complexities for $S_{\text{rsucc}}(n)$, $Q_{\text{rsucc}}(n)$, and $C_{\text{rsucc}}(n)$ are due to the dependency on a range successor data structure, for which some known bounds are given in Table 2.

problem	space	constr. time	query time	ref.
LZ77	$\mathcal{O}(n \lg^{\epsilon} n)$	$\mathcal{O}(n \lg n)$	$\mathcal{O}(z \lg n \lg \lg n)$	[26]
	$\mathcal{O}(n \lg^{\epsilon} n)$	-	$\mathcal{O}(z \lg \lg n)$	[53, Thm. 2]
	$\mathcal{O}(n + S_{\text{rsucc}}(n))$	$\mathcal{O}(n + C_{\text{rsucc}}(n))$	$\mathcal{O}(zQ_{\text{rsucc}}(n))$	[53, Thm. 2]
gLZ	$\mathcal{O}(n \log^{\epsilon} n)$	-	$\mathcal{O}(z \lg \frac{s'}{z} \lg \lg n)$	[53, Thm. 4]
	$\mathcal{O}(n)$	$\mathcal{O}(n \lg n)$	$\mathcal{O}(z \lg \frac{s'}{z} \lg^{\epsilon} n)$	[53, Thm. 4]
	$\mathcal{O}(n)$	-	$\mathcal{O}(z \lg^{\epsilon} n)$	[2, Lemma 4]
	$\mathcal{O}(n + S_{\text{rsucc}}(n))$	$\mathcal{O}(n + C_{\text{rsucc}}(n))$	$\mathcal{O}(z \lg \lg \frac{s'}{z} Q_{\text{rsucc}}(n))$	[60, Theorem 9.10]
RLBWT	$\mathcal{O}(n)$	$\mathcal{O}(n\sqrt{\lg n})$ expected	$\mathcal{O}(r_{\text{BWT}[\mathcal{I}]} \lg \lvert\mathcal{I}\rvert)$	[8]
Lyndon	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(z)$	[56]
LZ78	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(z)$	[61]
	$\mathcal{O}(e)$	$\mathcal{O}(n \lg n)$	$\mathcal{O}(z)$	[79]
LZD	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(z)$	Thm. 16
LZMW	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(z)$	Thm. 17
lexparse	$\mathcal{O}(n)$	$\mathcal{O}(n\sqrt{\lg n})$	$\mathcal{O}(z \lg s)$	Thm. 3
shortest closed fact.	$\mathcal{O}(n)$	$\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$	$\mathcal{O}(z \lg n)$	Thm. 4
longest closed fact.	$\mathcal{O}(n + S_{\text{rsucc}}(n))$	$\mathcal{O}(n + C_{\text{rsucc}}(n))$	$\mathcal{O}(zQ_{\text{rsucc}}(n) \lg \lg n)$	Thm. 8

time, where $z_{77[\mathcal{I}]}$ 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, where $\epsilon \in (0, 1]$ denotes a selectable constant. This result has been improved by Keller et al. [53] to $\mathcal{O}(z_{77[\mathcal{I}]} \lg \lg n)$ query time for the same space, or to $\mathcal{O}(z_{77[\mathcal{I}]} \lg^{\epsilon} n)$ query time for linear space. They also gave other trade-offs regarding query times and the sizes of the used data structure. Keller et al. also introduced the generalized substring compression query problem, a variant of the relative LZ compression [64]. Given two intervals $\mathcal{I}$ and $\mathcal{J}$ as query input, the task is to compute the Lempel–Ziv parsing of $T[\mathcal{I}]$ relative to $T[\mathcal{J}]$ (meaning that we compress $T[\mathcal{I}]$ with references only based on substrings in $T[\mathcal{J}]$). Their results are similar to the LZ77 case: $\mathcal{O}(z_{\text{rel}[\mathcal{I},\mathcal{J}]} \lg(|\mathcal{J}|/z_{\text{rel}[\mathcal{I},\mathcal{J}]}) \lg \lg n)$ query time for $\mathcal{O}(n \log^{\epsilon} n)$ space, and $\mathcal{O}(z_{\text{rel}[\mathcal{I},\mathcal{J}]} \lg(|\mathcal{J}|/z_{\text{rel}[\mathcal{I},\mathcal{J}]}) \lg^{\epsilon} n)$ query time for $\mathcal{O}(n)$ space, where $z_{\text{rel}[\mathcal{I},\mathcal{J}]}$ denotes the number of factors in the Lempel–Ziv factorization of $T[\mathcal{I}]$ relative to $T[\mathcal{J}]$. The generalized LZ77 compression method is also known as relative LZ [64].

The main idea of tackling the problem for LZ77 is to use a data structure answering interval LCP queries, which are usually answered by two-dimensional range successor/predecessor data structures. Much effort has been put in devising such data structures [2, 4, 5, 77]. Most recently, Matsuda et al. [68] proposed a data structure answering an interval LCP query in $\mathcal{O}(n^{\epsilon})$ time while taking $\mathcal{O}(\epsilon^{-1} n(H_0(T) + 1))$ bits of space, where H_0 denotes the zeroth order empirical entropy. Therefore, they can implicitly answer an LZ77 substring compression query in $\mathcal{O}(z_{77[\mathcal{I}]} n^{\epsilon})$ time within compressed space. Recently, Bille et al. [19]

Table 2: Time and space complexities of range successor data structures with known construction times, cf. [60, Proposition 9.7]. $\epsilon \in (0, 1]$ is a selectable constant. Space is given in words, where we assume the computation model to be a RAM with word size $\Omega(\lg n)$ bits.

$S_{\text{rsucc}}(n)$ space in words	$C_{\text{rsucc}}(n)$ construction time	$Q_{\text{rsucc}}(n)$ query time	ref.
$\mathcal{O}(n)$	$\mathcal{O}(n \lg n)$	$\mathcal{O}(\lg^{\epsilon} n)$	[14, 40, 76]
$\mathcal{O}(n \lg \lg n)$	$\mathcal{O}(n\sqrt{\lg n})$	$\mathcal{O}(\lg \lg n)$	[14, 40, 82]
$\mathcal{O}(n^{1+\epsilon})$	$\mathcal{O}(n^{1+\epsilon})$	$\mathcal{O}(1)$	[29]

Table 3: Results for computing the flexible parsings FP78 and FPA78 described in Sect. 4. $\delta = \mathcal{O}(\sqrt{n})$ is the number of memoized values. $t_{\mathcal{D}}$ is the time for answering child queries, which is $\mathcal{O}(\lg \sigma)$ if we implement child traversals with balanced binary search trees. z is the number of factors of the LZ78 or FPA factorization for computing FP78 or FPA78, respectively. t_{SA} is the time for accessing the suffix array, which can be implemented in $\mathcal{O}(\log_{\sigma}^{\epsilon} n)$ without increasing the stated space bounds [46, Thm. 2], for a constant $\epsilon > 0$.

problem	space in bits	computation time	ref.
FPA78	$\mathcal{O}(n \lg n)$	$\mathcal{O}(n^{3/2})$	[49]
FP78/FPA78	$\mathcal{O}(n \lg n)$	$\mathcal{O}(n)$ expected	[67]
FP78	$\mathcal{O}((z + \delta) \lg n)$	$\mathcal{O}(n t_{\mathcal{D}})$	Thm. 12
FP78/FPA78	$\mathcal{O}(n \lg n)$	$\mathcal{O}(n)$	Thm. 15
FP78/FPA78	$\mathcal{O}(n \lg \sigma)$	$\mathcal{O}(n t_{\text{SA}} \lg z)$	Thm. 15

proposed data structures storing the LZ77-compressed suffixes of T for answering a pattern matching query of an LZ77-compressed pattern P without decompressing P. Their proposed data structures seem also to be capable to answer substring compression queries.

The substring compression problem has also been studied [8] for another compression technique, the RLBWT [21]: Babenko et al. [8] 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 $T[\mathcal{I}]$. Another kind of factorization is the Lyndon factorization [23], for which Kociumaka [56] gave an algorithm that can compute the Lyndon factorization of a substring in time linear in the number of factors. Recently, Köppl [61] 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 $\mathcal{O}(z_{78[\mathcal{I}]}(\log_{\sigma}^{\epsilon} n + \lg z_{78[\mathcal{I}]}))$ time using $\mathcal{O}(n \lg \sigma)$ bits of space,

where $z_{78[\mathcal{I}]}$ is the number of computed LZ78 factors, σ is the alphabet size, and $\epsilon \in (0, 1]$ a selectable constant. Finally, Kociumaka et al. [59] 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. We have summarized the addressed solutions in Table 1.

Another related research field covers internal queries such as queries for the longest palindrome [6, 71] or longest common substring, pattern matching [58], counting of distinct patterns [22], quasi-periodicity testing [31], and range shortest unique substring queries [1], all inside a selected substring of the indexed text.

LZ78 Derivates. In this paper, we highlight three factorizations derived from the LZ78 factorization: (a) the flexible parsing variants [49, 66, 67] of LZ78, (b) LZMW [70] and (c) LZD [44]. The first (a) achieves the fewest number of factors among all LZ78 parsings that have an additional choice 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 notable variations of the LZ78 factorization. For instance, LZ78 factorizes $T = \mathtt{a}^n$ into $\Theta(\sqrt{n})$ factors, while both variations have $\Theta(\lg n)$ factors, where the factor lengths scale with a power of two or with the Fibonacci numbers for LZD and LZMW, respectively. That is because LZD allows the selection of two references, making it possible to form factors of the form $(\mathtt{a}^k, \mathtt{a}^k) = \mathtt{a}^{k+1}$, while LZMW requires such a selection to be for subsequent factors. Here, the best is to factorize $\mathtt{a}^n$ in the lengths of the Fibonacci numbers $\mathtt{a}^1, \mathtt{a}^1, \mathtt{a}^2, \mathtt{a}^3, \mathtt{a}^5, \mathtt{a}^8, \mathtt{a}^{13}, \ldots$ since then the length of the two preceding factors is maximized. For Fibonacci numbers it is known that they grow exponentially such that the number of LZMW factors of $\mathtt{a}^n$ is $\Theta(\lg n)$. The best lower bound on the number of LZMW factors with binary alphabets can be achieved with a cousin G_k of the Fibonacci words[1], defined by $G_k = G_{k-2}G_{k-1}$, $G_1 = \mathtt{a}$, and $G_2 = \mathtt{b}$. Then the concatenated text $G_1G_2 \cdots G_k$ has k LZMW factors.

On the downside, De and Kempa [33, 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. [44] can compute LZD in $\mathcal{O}(n \lg \sigma)$ time with $\mathcal{O}(n \lg n)$ bits of space, or in $\mathcal{O}(z(m + \min(z, m) \lg \sigma)$ time with $\mathcal{O}(z \lg n)$ bits of space, where m denotes the length of the longest computed LZD factor. LZD and LZMW were studied by Badkobeh et al. [10], who gave a bound of $\Omega(n^{5/4})$ time for the latter factorization algorithm [44] 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.

1.2 Our Contribution

In what follows, we propose construction algorithms for several text factorizations, including the three aforementioned types of factorizations: flexible parsings, LZD, and LZMW. For

[1]We call G_k a cousin because the Fibonacci words are defined by $F_k = F_{k-1}F_{k-2}$ with $F_i = G_i$ for $i \in \{1, 2\}$.

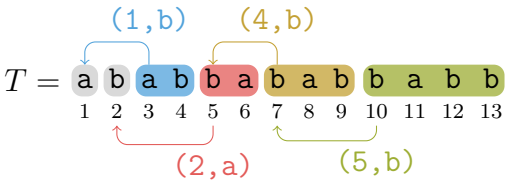
LZD and LZMW, we also study their substring compression problem. Regarding these two factorizations, despite having an $\Omega(n^{5/4})$ time bound on the running time of the known deterministic algorithms, we leverage the data structure of [61] to answer a substring compression query for each of the 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. [10] is only linear *in expectancy* for $z \in o(n/\lg^2 n)$[2]. We further improve the space usage of the linear-time computation of both factorizations to $\mathcal{O}(n \lg \sigma)$ bits of space at the expense of $\mathcal{O}(n \lg^{1+\epsilon} n)$ time, and thus obtain one of the currently fastest algorithms computing LZD and LZMW in compact space deterministically.

Additionally, we can compute the flexible parse of LZ78 in the same complexities, or in $\mathcal{O}(n \log_\sigma^\epsilon n \lg z)$ time within $\mathcal{O}(n \lg \sigma)$ bits of space. This holds if we work with the flexible parsing variant of Horspool [49] or Matias and Sahinalp [66]. The best previous results run in expected linear time or have linear time bounds only for constant alphabet sizes [67]. Here we introduce two flexible parsing variations of LZ78, FP78 and FPA78, that are derived from the flexible parsing variants of LZW. In the experiments, we observe that both flexible parsings always produce fewer factors than the standard LZ78 factorization. Since the factors of both flexible parsings can be represented in the same way as LZ78 factors by pairs of reference and character, these variants are a compelling alternative to standard LZ78 due to the fewer number of factors, which can theoretically be guaranteed for FP78. We summarize previously achieved bounds and our contributions in Table 3.

As results of independent interest, we present approaches for computing the substring compression of two other factorizations, the lexparse [75] and the closed factorizations [9]. Although algorithms computing these factorizations take time linear in the input size [9, 62], we show that we can solve a substring compression query for one of these factorization types in quasi-linear time with respect to the number of computed factors. The final time and space bounds for the proposed solutions to various substring compression problem variants are listed in Table 1.

Structure of this Article After the preliminaries in Sect. 2, we focus on the substring compression for lexparse (Sect. 3.1) and the closed factorizations (Sect. 3.2), which serve as an appetizer for what follows. For the closed factorization, we also make our first acquaintance with a solution using a suffix tree. Subsequently, in Sect. 4, we introduce the flexible parsing in two different flavors for LZ78, and show how to compute both parsings with the AC-automaton and suffix trees in Sect. 4.1 and Sect. 4.2, respectively. The final part of the theoretical analysis is Sect. 5, in which we propose suffix tree-based solutions for computing the LZD and LZMW parsings. The remainder of this article covers two practical benchmarks in Sect. 6. First, we study the number of factors of the two flexible parsing variations compared to standard LZ78 in Sect. 6.1. Second, we evaluate the computation time for LZ78 substring compression when using an already computed suffix tree versus

[2]Assuming $z \in o(n/\lg^2 n)$ is reasonable for relatively compressible strings.



LZ78: ab(1,b)(2,a)(4,b)(5,b)

Figure 1: LZ78 factorization of $T =$ ababbababbabb, given by $F_1 =$ a, $F_2 =$ b, $F_3 =$ ab $= F_1 \cdot$ b, $F_4 =$ ba $= F_2 \cdot$ a, $F_5 =$ bab $= F_4 \cdot$ b, and $F_6 =$ babb $= F_5 \cdot$ b. The figure shows an encoding of the factors, where a number denotes the index of the referred factor.

a standard LZ78 compressor in Sect. 6.2. A conclusion with outlook for future work and open problems concludes the article in Sect. 7. Compared to the conference version [63], we added the closed factorizations, the approach with the AC automaton, and improved the structure by adding more examples and figures. In addition, we empirically evaluate some proposed approaches in Sect. 6.

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 $\sigma \leq n^{\mathcal{O}(1)}$. An element of Σ^* is called a *string*. The alphabet Σ induces the *lexicographic order* $\prec$ on the set of strings Σ^*.

Given $X, Y, Z \in \Sigma^*$ with $T = XYZ$, then X, Y and Z are called a *prefix*, *substring* and *suffix* of T, respectively. A substring Y of T is called *proper* if $Y \neq T$. A *border* of T is a substring of T that is both a prefix and a suffix of T. We call $T[i..]$ the i-th suffix of T, and denote a substring $T[i] \cdots T[j]$ by $T[i..j]$. 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*. Let $\mathsf{dst}_x := |F_1 \cdots F_{x-1}| + 1$ denote the starting position of factor F_x.

Stipulating that F_0 is the empty string, a factorization $F_1 \cdots F_z = T$ is called the *LZ78 factorization* [83] 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,

$$y = \operatorname{argmax}\{|F_{y'}| : F_{y'} = T[\mathsf{dst}_x..(\mathsf{dst}_x + |F_{y'}| - 1)] \ \wedge \ y' \in [0..x-1]\}.$$

We say that y and F_y are the *referred index* and the *referred factor* of the factor F_x, respectively. We call a factor of length one *literal*; such a factor has always the referred index 0. The other factors are called *referencing*. See Fig. 1 for an example of the LZ78 factorization.

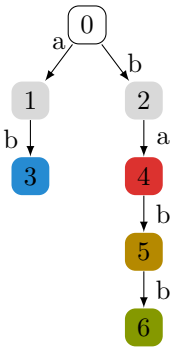


Figure 2: The LZ trie of our running example $T =$ ababbababbabb. The coloring of the LZ trie matches the colors of the LZ78 factors in Fig. 1 visualizing the LZ78 factorization of T.

All factors F_x are distinct, except maybe the last factor F_z, which needs to be treated as a border case. In what follows, we omit this border case analysis for the sake of simplicity (in LZ78 as well as in all later explained variants). If T terminates with a character appearing nowhere else in T, then the last factor is also distinct from the others. 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 $F_x = F_y c$. See Fig. 2 for the LZ trie of our running example.

When computing F_x, the LZ78 parsing dictionary $\mathcal{D}$ contains all previous factors $F_0, \ldots, F_{x-1}$ with all possible character extensions such that determining F_x can be reduced to finding the longest element $F_y \cdot c$ in $\mathcal{D}$ that matches with F_x.

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 $\mathsf{b}(\mathcal{I})$ and $\mathsf{e}(\mathcal{I})$ to denote the beginning and the end of $\mathcal{I}$, i.e., $\mathcal{I} = [\mathsf{b}(\mathcal{I})..\mathsf{e}(\mathcal{I})]$. We write $|\mathcal{I}|$ to denote the length of $\mathcal{I}$; i.e., $|\mathcal{I}| = \mathsf{e}(\mathcal{I}) - \mathsf{b}(\mathcal{I}) + 1$, and $T[\mathcal{I}]$ for the substring $T[b..e]$.

Text Data Structures. Let SA denote the *suffix array* [65] of T. The entry $\mathsf{SA}[i]$ is the starting position of the i-th lexicographically smallest suffix, i.e., $T[\mathsf{SA}[i]..] \prec T[\mathsf{SA}[i+1]..]$ for all integers $i \in [1..n-1]$. Let ISA of T be the inverse of SA, i.e., $\mathsf{ISA}[\mathsf{SA}[i]] = i$ for every $i \in [1..n]$. The *LCP array* is an array with the property that $\mathsf{LCP}[i]$ is the length of the longest common prefix (LCP) of $T[\mathsf{SA}[i]..]$ and $T[\mathsf{SA}[i-1]..]$ for every $i \in [2..n]$. For convenience, we stipulate that $\mathsf{LCP}[1] := 0$. The array Φ is defined as $\Phi[i] := \mathsf{SA}[\mathsf{ISA}[i]-1]$, and $\Phi[i] := \mathsf{SA}[n]$ in case that $\mathsf{ISA}[i] = 1$. By construction, SA, ISA and Φ are permutations of $[1..n]$. The *permuted LCP* (*PLCP*) *array* PLCP stores the entries of LCP in text order, i.e., $\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i]$. See Table 4 for an example of some introduced data structures. The usefulness of the data structures introduced will become clear during our warm-up in Sect. 3.1 for computing lexparse. Informally, SA groups suffixes in lexicographic order such that pairs of suffixes with longest common prefixes are neighboring. Information about the lengths of these longest common prefixes can be retrieved with LCP in lexicographic order or with PLCP in text-order. Lexparse leverages this fact for compression by using the lexicographically preceding suffix retrieved via Φ as a reference.

Given an integer array Z of length n and an interval $[i..j] \subset [1..n]$, the range minimum query $Z.\mathbf{RmQ}(i,j)$ (resp. the range maximum query $Z.\mathbf{RMQ}(i,j)$) asks for the index p

of a minimum element (resp. a maximum element) of the subarray $Z[i..j]$, i.e., $p \in \arg\min_{i \le k \le j} Z[k]$, or respectively $p \in \arg\max_{i \le k \le j} Z[k]$. We use the following data structure to handle this kind of query:

Lemma 1 ([32])**.** Given an integer array Z of length n, there is an RmQ (resp. RMQ) data structure taking $2n + o(n)$ bits of space that can answer the query $Z.\mathsf{RmQ}$ (resp. $Z.\mathsf{RMQ}$) 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 $\mathsf{lce}(i, j)$ asks for the longest common prefix of two suffixes $T[i..]$ and $T[j..]$. We can answer an LCE query in constant time with an RmQ data structure built on LCP because $\mathsf{lce}(i, j) = \mathsf{LCP}[x]$ with $x := \mathsf{LCP}.\mathsf{RmQ}[\ell + 1..r]$ for $\ell := \min(\mathsf{ISA}[i], \mathsf{ISA}[j])$ and $r := \max(\mathsf{ISA}[i], \mathsf{ISA}[j])$, i.e., $\mathsf{lce}(i, j)$ is the smallest LCP value in the range $[\ell + 1..r]$.

Given a character $\mathtt{c} \in \Sigma$, and an integer j, the *rank* query $T.\mathrm{rank}_{\mathtt{c}}(j)$ counts the occurrences of $\mathtt{c}$ in $T[1..j]$, and the *select* query $T.\mathrm{select}_{\mathtt{c}}(j)$ gives the position of the j-th $\mathtt{c}$ in T, if it exists. We stipulate that $\mathrm{rank}_{\mathtt{c}}(0) = \mathrm{select}_{\mathtt{c}}(0) = 0$. If the alphabet is binary, i.e., when T is a bit vector, there are data structures [24, 51] that use $o(|T|)$ extra bits of space, and can compute rank and select in constant time, respectively. There are representations [13] with the same constant-time bounds that can be constructed in time linear in $|T|$. We say that a bit vector has a *rank-support* and a *select-support* if it is endowed with data structures that provide constant time access to rank and select, respectively.

Suffix Tree. From now on, we assume that we have appended, to the input text T, a special character $\mathtt{\$}$ smaller than all other characters appearing in T that is not subject to a query. Since $\mathtt{\$}$ appears only at the end of T, there is no suffix of T having another suffix of T as a prefix. The *suffix trie* of T is the trie of all suffixes of T. There is a one-to-one relationship between the suffix trie leaves and the suffixes of T. The *suffix tree* ST of T is the tree obtained by compacting the suffix trie of T. *Compacting* a trie means that we contract maximal unary paths to a single path by removing all nodes with unary out-degree.[3] Algorithmically speaking, if we have a path $u \to^S v \to^T w$ such that

- v has out-degree 1,
- the labels of the edges (u, v) and (v, w) are the strings S and T (which can be plain characters), respectively,

then we compact this path to $u \to^{S \cdot T} w$, remove v, and recurse until such a path no longer exists. By compacting a trie, the edge labels are no longer single characters in general, but become strings.

Like the suffix trie, the suffix tree has n leaves, but the number of internal nodes of the suffix tree is at most n because every ST node is branching. The string stored in a

[3]For general directed graphs, compacting is defined to contract maximal paths of nodes with *unary in-degree* (excluding the first node) and unary out-degree (excluding the last node). Here, we can neglect the condition for the unary in-degrees because every node in a (directed) tree has at most one parent.

suffix tree edge e is called the *label* of e. The *string label* of a node v is defined as the concatenation of all edge labels on the path from the root to v; its *string depth*, denoted by $\mathsf{str\_depth}(v)$, is the length of its string label. The leaf corresponding to the i-th suffix $T[i..]$ is labeled with the *suffix number* $i \in [1..n]$. The *leaf-rank* is the preorder rank ($\in [1..n]$) of a leaf among the set of all ST leaves. For instance, the leftmost leaf in ST has leaf-rank 1, while the rightmost leaf has leaf-rank n. Reading the suffix numbers stored in the leaves of ST in leaf-rank order gives the suffix array. We say that the *locus* of a substring S in T is the highest ST node whose string label has S as a prefix. Fig. 3 depicts the suffix tree of our running example.

Suffix trees can be computed in linear time [36], take $\mathcal{O}(n \lg n)$ bits of space, and can be augmented to support the following operations in constant time (cf. [39] for details):

- $\mathsf{depth}(v)$ returns the depth of an ST node v.
- $\mathsf{range}_{\mathrm{L}}(v)$ and $\mathsf{range}_{\mathrm{R}}(v)$ return the leaf-rank of the leftmost and the rightmost leaf of the subtree rooted at an ST node v, respectively.
- $\mathsf{level\_anc}(\lambda, d)$ selects the ancestor of the ST leaf λ at depth d.

We also want to answer the following query.

- $\mathsf{str\_depth}(v)$ returns the string depth of an *internal* node.

Within $\mathcal{O}(n \lg n)$ bits of space, we can answer $\mathsf{str\_depth}(v)$ by taking the minimal LCP value in the SA range representing the leaves in v's subtree, i.e., the value of $\mathsf{LCP}.\mathbf{RmQ}[\mathsf{range}_{\mathrm{L}}(v) + 1..\mathsf{range}_{\mathrm{R}}(v)]$. That is because, for each $i \in [\mathsf{range}_{\mathrm{L}}(v)..\mathsf{range}_{\mathrm{R}}(v)]$, the string label of v is a prefix of the suffix $T[i..]$, but not a prefix of any other suffix. When storing LCP is too costly, we can get access to LCP with PLCP and the compressed suffix array [46] within $\mathcal{O}(\log_\sigma^\epsilon n)$ access time for a constant $\epsilon > 0$. Stipulating that t_{SA} denotes the time to access SA, we can answer $\mathsf{str\_depth}(v)$ in $\mathcal{O}(t_{\mathrm{SA}})$ time. Without LCP and the full suffix array, there is an $\mathcal{O}(n \lg \sigma)$-bits representation of the suffix tree [39, 73], which can be computed in linear time. In what follows, we analyze our algorithms using suffix trees in both settings, having $\mathcal{O}(n \lg n)$ bits with the above queries at constant time, or $\mathcal{O}(n \lg \sigma)$ bits with a slower $\mathcal{O}(\log_\sigma^\epsilon n)$ time for $\mathsf{str\_depth}(v)$. Similarly, we can perform the following operation in $\mathcal{O}(t_{\mathrm{SA}})$ time:

- $\mathsf{select\_leaf}(i)$ returns the ST leaf with suffix number i.

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

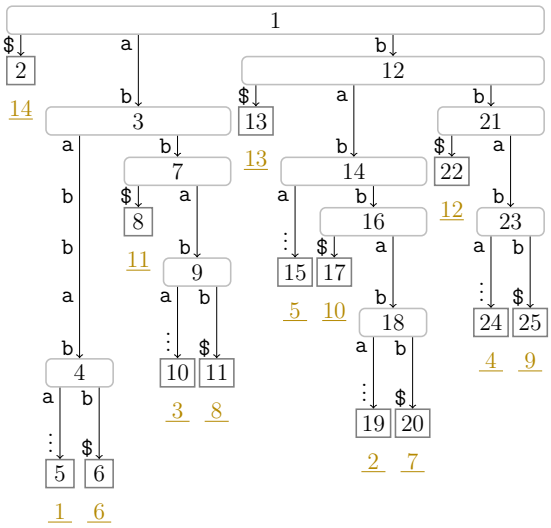


Figure 3: The suffix tree of $T =$ ababbababbabb. Edges to leaves with labels longer than two are trimmed with '⋮' to save space on the paper. Nodes are labeled by their preorder ranks. Each leaf is decorated with its suffix number with a golden (■) underlined number beneath. Listing the suffix numbers in-order gives the suffix array of $T\$$ — here we obtain SA of Table 4 if we omit the first leaf.

Table 4: Some introduced text data structures on the string $T =$ ababbababbabb.

i	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	b	a	b	a	b	b	a	b	b
ISA	1	9	4	12	7	2	10	5	13	8	3	11	6
Φ	9	10	11	12	13	1	2	3	4	5	6	7	8
PLCP	0	4	3	2	1	7	6	5	4	3	2	1	0
SA	1	6	11	3	8	13	5	10	2	7	12	4	9
LCP	0	7	2	3	5	0	1	3	4	6	1	2	4

3 Warm-Ups: Lexparse and Closed Factorization

The *lex-parse* [75, Def. 11] is a factorization $T = F_1 \cdots F_v$ such that $F_x = T[\mathsf{dst}_x..\mathsf{dst}_x+\ell_x-1]$ with $\mathsf{dst}_1 = 1$ and $\mathsf{dst}_{x+1} = \mathsf{dst}_x + \ell_x$ if $\ell_x := \mathsf{PLCP}[\mathsf{dst}_x] \geq 1$, or F_x is a *literal factor* with $\ell_x := |F_x| = 1$ otherwise. For $\mathsf{PLCP}[\mathsf{dst}_x] \geq 1$, the reference of F_x is $\Phi(F_x)$ since by definition of the PLCP array $T[\mathsf{dst}_x..\mathsf{dst}_x + \ell_x - 1] = T[\Phi(\mathsf{dst}_x)..\Phi(\mathsf{dst}_x) + \ell_x - 1]$. See Fig. 4 for an example. Among all factorizations based on the selection of a lexicographically preceding position as factor reference, lex-parse produces the least number of factors [75] and is therefore a lower bound for other factorizations based on the lexicographic order

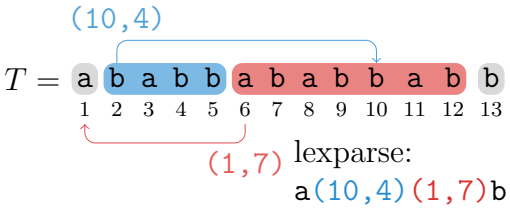


Figure 4: lexparse of our running example $T =$ ababbababbabb, given by $F_1 =$ a, $F_2 =$ babb $= T[10..13]$, $F_3 =$ ababbab $= T[1..7]$, and $F_4 =$ b. Non-literal factors are encoded by text position and length of the substring in T they refer to.

such as lcpcomp [34] or plcpcomp [35].

Closed factorizations [9, Def. 1 and 4] create factors that are *closed*. The definition of *closed strings* is based on their lengths:

- All strings of length 1 are closed [9, Sect. 2.1].
- A string S with $|S| \geq 2$ is closed if S has a proper substring that is both a prefix and a suffix of S but nowhere else appears in S.

Badkobeh et al. studied two kinds of factorizations whose factors are closed. First, the *longest closed factorization* factorizes T into $T = F_1 \cdots F_z$ such that F_x is the *longest* prefix of $T[|F_1 \cdots F_{x-1}|+1..n]$ that is closed. The longest closed factorization of $T = \texttt{ababbababbabb}$ is $T = F_1 \cdot F_2 = \texttt{ababbababbab} \cdot \texttt{b}$, where the border $\texttt{ababbab}$ of F_1 nowhere else appears in F_1. Second, the *shortest closed factorization* factorizes T into $T = F_1 \cdots F_z$ such that F_x is the *shortest* prefix of $T[|F_1 \cdots F_{x-1}| + 1..n]$ that is closed and is of length at least 2. The shortest closed factorization may not exist, for instance if T starts with a unique character. The shortest closed factorization of $T = \texttt{ababbababbabba}$ is $T = \texttt{aba} \cdot \texttt{bb} \cdot \texttt{aba} \cdot \texttt{bb} \cdot \texttt{abba}$.

3.1 Lexparse

Despite being perceived that both Φ and PLCP seem necessary for a linear-time computation [62], it actually suffices to have sequential access to Φ and random access to the text. That is because we can naively compute $\mathsf{PLCP}[\mathsf{dst}] = \mathsf{lce}(\mathsf{dst}, \Phi(\mathsf{dst}))$ in $\mathcal{O}(\mathsf{PLCP}[\mathsf{dst}])$ time by linearly scanning the character pairs in the text. For determining the factor F_x we compare $|F_x|$ matching character pairs and at most one mismatching character pair. Hence, the total number of character pairs scanned is up to at most $\sum_{x=1}^{z} |F_x| + 1 = n + z \leq 2n$. Linearly scanning the character pairs in the text, 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.

Lemma 2 ([8, 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 Lemma 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 $\mathsf{b}(\mathcal{I})$, and query the rank of the substring suffix $T[\mathsf{b}(\mathcal{I})..\mathsf{e}(\mathcal{I})]$. 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 $\ell \leftarrow \mathsf{lce}(\mathsf{b}(\mathcal{I}), j)$ to determine the factor length ℓ. On the one hand, if the computed factor protrudes the substring $T[\mathcal{I}]$ with $\mathsf{b}(\mathcal{I}) + \ell - 1 > \mathsf{e}(\mathcal{I})$, trim $\ell \leftarrow \mathsf{e}(\mathcal{I}) - \mathsf{b}(\mathcal{I}) + 1$. 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 $[\mathsf{b}(\mathcal{I}) + \ell..\mathsf{e}(\mathcal{I})]$ to compute the next factor while keeping the query interval $\mathcal{I}$ fixed for the data structure of Lemma 2. Each recursion step in the algorithm uses a select and a rank query on the data structure of Lemma 2, which takes $\mathcal{O}(\lg s)$ time each.

Theorem 3. There is a data structure that, given a query interval $\mathcal{I}$, can compute lexparse of $T[\mathcal{I}]$ in $\mathcal{O}(z \lg s)$ time, where z is the number of lexparse factors and $s = |\mathcal{I}|$ the length of the query interval. This data structure can be constructed in $\mathcal{O}(n\sqrt{\lg n})$ time, using $\mathcal{O}(n \lg n)$ bits of working space.

Algorithm 1 Longest closed factorization with the bounds stated in Thm. 8. Here, and in all further pseudocode listings, the symbol $\leftarrow$ denotes the operation to set a variable. We use $\leftarrow$ instead of the symbol $=$ to distinguish from the check for equality.

Require: query interval $\mathcal{I}$
if $\mathcal{I} = \emptyset$ **then return** ▷ *factorization of empty interval is empty*
$L \leftarrow \mathsf{blcp}(\mathsf{b}(\mathcal{I}), [\mathsf{b}(\mathcal{I}) + 1..\mathsf{e}(\mathcal{I})])$
if $|L| = 0$ **then**
    output factor $T[\mathsf{b}(\mathcal{I})]$ ▷ *$b(\mathcal{I})$ is rightmost occurrence of character $T[b(\mathcal{I})]$*
    return by recursing on $[\mathsf{b}(\mathcal{I}) + 1..\mathsf{e}(\mathcal{I})]$
$\lambda \leftarrow \mathsf{select\_leaf}(\mathsf{b}(\mathcal{I}))$ ▷ *find u via binary search on $d \mapsto \mathsf{level\_anc}(\lambda, d)$*
$u \leftarrow \operatorname{argmin}\{\mathsf{str\_depth}(u) \geq |L| \mid u \text{ ancestor of } \lambda\}$
$j \leftarrow \mathsf{SA.RNV}(\mathsf{b}(\mathcal{I}), [\mathsf{range}_{\mathrm{L}}(u)..\mathsf{range}_{\mathrm{R}}(u)])$ ▷ *invariant: $j + |L| \leq \mathsf{e}(\mathcal{I})$*
output factor $T[\mathsf{b}(\mathcal{I})..j + |L| - 1]$
return by recursing on $[j + |L|..\mathsf{e}(\mathcal{I})]$

3.2 Closed Factorizations

For the shortest closed factorization, Badkobeh et al. [9, Lemma 3] showed that a shortest closed factor of length at least two has a unique border of length one. Given that we have factorized a prefix of T by $F_1 \cdots F_{x-1}$ and our task is to compute F_x starting at $\mathsf{src}_x := 1 + \sum_{y=1}^{x-1} |F_y|$, we scan for the succeeding occurrence of $T[\mathsf{src}_x]$ in $T[\mathsf{src}_x + 1..]$, which determines the end of F_x because then F_x is the shortest substring starting at src_x having $T[\mathsf{src}_x]$ as a border appearing nowhere else in F_x. By preprocessing T with a wavelet tree, we can *rank* the current occurrence of $T[\mathsf{src}_x]$ and *select* the subsequent occurrence of this character. There are implementations [12, Theorem 2][43, Theorem 4] achieving time $\mathcal{O}(1 + \min\left(\frac{\lg \sigma}{\lg \lg n}, \lg \lg \sigma\right))$ per query if we reduce the alphabet of T such that each character of the alphabet appears in T. Now, given a queried interval $\mathcal{I} \subset [1..n]$, we can factorize $T[\mathcal{I}]$ with $\Theta(z)$ rank/select queries, where z is the number of computed factors. It is also possible to store for each character $c \in \Sigma$ a bit vector $B_c[1..n]$ marking all occurrences of c in T, and augment B_c with rank/select support data structures that answer each query in constant time, but totaling up to $n\sigma + o(n\sigma)$ bits of space, which is better than a naive solution storing all answers when $\sigma \in o(n)$.

Theorem 4. There are data structures that, given a query interval $\mathcal{I}$, can compute the shortest closed factorization of $T[\mathcal{I}]$ in (a) $\mathcal{O}(z)$ time, (b) $\mathcal{O}(z(1 + \min\left(\frac{\lg\sigma}{\lg\lg n}, \lg\lg\sigma\right)))$ time, or (c) $\mathcal{O}(z \lg n)$ time, where z is the number of factors. These data structures take (a) $n\sigma + o(n\sigma)$ bits of space or (b)/(c) $\mathcal{O}(n \lg\sigma)$ bits of space. These data structures can be constructed in (a) $\mathcal{O}(n\sigma)$ time or (c) $\mathcal{O}(n \lg\sigma/\sqrt{\lg n})$ time.

Proof. The first data structure (a) consists of the σ bit vectors supporting constant query time. The second data structure (b) is due to [12, Theorem 2][43, Theorem 4], whose construction time does not seem to have yet been addressed. The last data structure (c) is the wavelet tree [46], whose construction time is due to Munro et al. [72]. □

For computing the *longest* closed factorization, we modify the algorithm presented in [9, Section 3]. The idea there is to use the suffix tree (as we will also later do for the LZ78-related factorizations). In detail, the authors precompute an array $P[1..n]$ such that $P[i]$ stores the highest ancestor of the leaf λ with suffix number i among all ancestors v of λ with the property that the largest suffix number of all leaves in the subtree rooted in v is i, i.e.,

$$P[i] = \text{argmin}_{v \text{ ancestor of } \lambda}\{\mathsf{depth}(v) \mid \mathsf{SA.RMQ}[\mathsf{range}_{\mathrm{L}}(v)..\mathsf{range}_{\mathrm{R}}(v)] = i\} \text{ for each } i \in [1..n].$$

Like the authors explained, computing P can be done by a preorder traversal of the suffix tree. As a starter, suppose that we are given a query interval $\mathcal{I} \subset [1..n]$ with $\mathsf{e}(\mathcal{I}) = n$, i.e., we want to factorize a suffix of T. To this end, we traverse the suffix tree. At the beginning, we take the node $P[\mathsf{b}(\mathcal{I})]$ corresponding to the leaf λ with suffix number $\mathsf{b}(\mathcal{I})$. Let $P[\mathsf{b}(\mathcal{I})]$'s parent be u. By the property of P, u is the *lowest* ancestor of λ whose subtree has a leaf with a suffix number j greater than $\mathsf{b}(\mathcal{I})$, or the root if no such j exists. If u is the root, then $\mathsf{b}(\mathcal{I})$ is the rightmost occurrence of character $T[\mathsf{b}(\mathcal{I})]$ in T. Consequently, the longest closed factor starting at text position $\mathsf{b}(\mathcal{I})$ is just $T[\mathsf{b}(\mathcal{I})]$.

Otherwise, the string label L of u is not empty. L occurs at least as two substrings, two of them starting at text positions $\mathsf{b}(\mathcal{I})$ and j. The substring $T[\mathsf{b}(\mathcal{I})..j + |L| - 1]$ is therefore bordered with border L, but may not be necessarily closed because L may appear inside somewhere else. For that property to hold, we need to select the *smallest* such j, i.e., the range successor of $\mathsf{b}(\mathcal{I})$ among the suffix numbers in the subtree of u.

Example 5. For our running example $T = \texttt{ababbababbabb}$, $P[1]$ is the leaf with suffix number 1 itself. Its parent has preorder number 4 with string length 7; the succeeding suffix starts at position 6. Hence, the first factor is $T[1..6 + 7 - 1] = \texttt{ababbababbab}$.

We now generalize our approach to also accommodate arbitrary query intervals $\mathcal{I} \subset [1..n]$. The main obstacle is to find the last factor, for which we need to seek an ancestor that can be higher than the one stored in the array P. More precisely, the array P becomes futile for substring compression when $j + \mathsf{str\_depth}[u] > \mathsf{e}(\mathcal{I})$, i.e., when the computed factor protrudes the query interval at its end. In what follows, we replace P with a data structure for *bounded longest common prefix* (BLCP) queries [53]. A BLCP query $\mathsf{blcp}(i, \mathcal{J})$ asks for,

given a position i and a query range $\mathcal{J}$, the longest common prefix of $T[i..]$ and a substring of $T[\mathcal{J}]$, i.e., any substring $T[j..k]$ with $j, k \in \mathcal{J}$.

Suppose that we want to compute the longest closed factorization of $T[\mathcal{I}]$, and we are currently processing the factor F_x, which starts at dst_x. First, we obtain with $L = \mathsf{blcp}(\mathsf{dst}_x, [\mathsf{dst}_x + 1..\mathsf{e}(\mathcal{I})])$ the border of F_x. However, if $|L| = 0$, then F_x is a single character and we recurse on the query interval $[\mathsf{b}(\mathcal{I}) + 1..\mathsf{e}(\mathcal{I})]$. Otherwise, we visit the locus u of L in ST, i.e., u is the highest node whose string label has L as a prefix. By definition of the BLCP query,

- there exists a leaf in u's subtree that has a suffix number j such that $j + |L| \leq \mathsf{e}(\mathcal{I})$, and
- there is no descendant node of u on the path downwards to the leaf with suffix number dst_x whose subtree has a leaf with suffix number j' such that $j' + |L| + 1 \leq \mathsf{e}(\mathcal{I})$.

Consequently, it must hold that $F_x = T[\mathsf{dst}_x..j + |L| - 1]$ for the smallest such j, i.e., the range next value of dst_x among the suffix numbers in the subtree of u. The *range next value* (RNV) [28, Sect. 5] of an integer x in a range $\mathcal{J}$ in SA is defined by

$$\mathsf{SA.RNV}(x, \mathcal{J}) = \min\{\mathsf{SA}[y] \mid y \in \mathcal{J} \wedge \mathsf{SA}[y] > x\}.$$

The pseudocode in Algorithm 1 summarizes our algorithmic steps. There, an invariant is that $\mathsf{dst}_x = \mathsf{b}(\mathcal{I})$ due to how we perform the recursion. We start with computing L in Line 2. We subsequently process the case where we cannot find a border for any prefix of $T[\mathsf{b}(\mathcal{I})..]$ in Lines 3–5. In Line 6, we compute the locus u of L. The RNV query of dst_x among the suffix numbers in the subtree of u determines the subsequent starting position j of L in Line 8. Finally, we output the computed factor and recurse.

Data Structures We need to address how to perform (a) a BLCP query, (b) an RNV query, or (c) how to find the locus of a string in ST efficiently. For BLCP queries (a), we use the following data structure.

Lemma 6 ([60, Theorem 1.7])**.** There exists a data structure of size $\mathcal{O}(n + S_{\text{rsucc}}(n))$ words that answers a BLCP query in $\mathcal{O}(Q_{\text{rsucc}}(n) \lg \lg \ell)$ time, where ℓ is the length of the returned prefix. The data structure can be constructed in $\mathcal{O}(n + C_{\text{rsucc}}(n))$ time. The complexities for $S_{\text{rsucc}}(n)$, $Q_{\text{rsucc}}(n)$, and $C_{\text{rsucc}}(n)$ are due to the used range successor data structure, for which known bounds are given in Table 2.

Similarly, we can answer (b) directly with a range successor data structure. For (c), to find the locus of a string L in ST, we perform a binary search on the path from the root to the leaf λ with suffix number dst_x, i.e., the starting position of the factor we currently process. For that, we perform a binary search with $d \mapsto \mathsf{level\_anc}(\lambda, d)$ and query the string depth of each returned node. This gives $\mathcal{O}(\lg n)$ time. We can improve this to constant time at the expense of $\mathcal{O}(n)$ construction time and $\mathcal{O}(n)$ words of space with the following data structure.

Lemma 7 ([15, 41])**.** There exists a weighted ancestor data structure for ST, which supports, given a leaf λ and integer d, constant-time access to the ancestor of λ with string depth d. It can be constructed in linear time using $\mathcal{O}(n \lg n)$ bits of space.

Putting everything together gives us the following main result.

Theorem 8. There is a data structure that, given a query interval $\mathcal{I}$, can compute the longest closed factorization of $T[\mathcal{I}]$ in $\mathcal{O}(zQ_{\text{rsucc}}(n))$ time, where z is the number of factors. This data structure can be constructed in $\mathcal{O}(n+C_{\text{rsucc}}(n))$ time and uses $\mathcal{O}(S_{\text{rsucc}}(n))$ words of space.

Proof. The construction time bounds are usually dominated by the range successor data structure. All other data structures need $\mathcal{O}(n)$ construction time. Within $\mathcal{O}(n)$ words of space and $\mathcal{O}(n)$ time [55], we can construct and store the suffix array with $\mathcal{O}(1)$ access time. For each factor we compute, we issue one BLCP query, one RNV query, and one weighted ancestor query. Thus, we need $\mathcal{O}(Q_{\text{rsucc}}(n) \lg \lg n + Q_{\text{rsucc}}(n) + 1)$ time per factor. □

In what follows, we apply the suffix tree traversal techniques to other factorizations. Two of them belong to the family of semi-greedy parsings, which we introduce next.

4 Semi-Greedy Parsing

The semi-greedy parsing [47] is a variation of the LZ77 parsing where a factor F_x of the factorization $F_1 \cdots F_z$ starting at text position $\mathsf{dst}_x := |F_1 \cdots F_{x-1}|$ does not necessarily have to be the longest prefix of $T[\mathsf{dst}_x..]$ that has an occurrence starting before dst_x. Given that the longest prefix has length ℓ, the semi-greedy parsing instead selects the length $\ell' \in [1..\ell]$ that maximizes the sum of the lengths of the current and the next factor $|F_x| + |F_{x+1}|$ like F_x would have length ℓ' and F_{x+1} would have been selected by the standard greedy parsing continuing at $T[\mathsf{dst}_x + \ell'..]$.

Semi-Greedy LZ77. Crochemore et al. [30, Algorithm 10] proposed an algorithm working with the array LPF to compute the semi-greedy parsing in $\mathcal{O}(n)$ time. LPF, called longest previous factor table [27], is an array of length n with $\mathsf{LPF}[j] = \max\{\ell \mid \text{there exists an } i \in [1..j-1] \text{ such that } T[i..i+\ell-1] = T[j..j+\ell-1]\}$ for every $j \in [1..n]$. The algorithm of Crochemore et al. [30] can be modified so that a substring query for $\mathcal{I}$ with $1 \in \mathcal{I}$ can be answered in $\mathcal{O}(z)$ time if $\mathcal{O}(n)$ preprocessing is allowed. For that, build an RMQ data structure on $f : i \mapsto \mathsf{LPF}[i] + i$ for all $i \in [1..n]$ such that the query for the range $R := [i..i+\mathsf{LPF}[i]]$ leads to the position $j := i+\ell'-1 \in R$ for which $f(j)$ is maximal. Thus, we can spend constant time per factor; we stop when the last computed factor F_z ends or goes beyond the end of $\mathcal{I}$, where we trim F_z in the latter case. For supporting intervals that do not start with the first text position, we need to recompute LPF whose entries depend on previous substring occurrences.

Semi-Greedy LZW. Horspool [49] proposed adaptations of the semi-greedy parsing for Lempel–Ziv–Welch (LZW) [81], a variation of the LZ78 factorization. These LZ78-based semi-greedy parsings have been studied by Matias and Sahinalp [66], 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 [67] 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 [49], 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 substrings that would have been computed greedily at the starting positions of the FPA factors. On the one hand, the selection of reference makes the computation of LZW-FP (theoretically) easier since we can statically build the LZW parsing dictionary with a linear-time LZW construction algorithm like [39]. In doing so, we also upper bound the number of LZW-FP factors by the number of LZW factors because we do not change the original LZW dictionary while making a wiser choice for the factors. On the other hand, FPA may create more factors than LZW, which has, however, not yet been observed in practice. The major conceptual difference of LZW-FP to LZW is that a *reference* is no longer a factor of the computed factorization itself. Instead, a reference is a string of the dictionary of the conceptionally already computed LZW factorization, and has a *reference index* mimicking the definition of a *factor index*. References are assigned their indices based on the time of creation. Because the dictionary is prefix-free, it exhibits the same characteristics as the LZ trie, which now represents the references instead of the factors. For LZW-FP, the LZ trie is actually the LZ trie of LZW. In what follows, we adapt both flexible parsing variants to LZ78 as a means of didactic reduction (the adaptation from LZ78 to LZW is straightforward). We first formally define both factorizations in the LZ78 terminology and subsequently give algorithms for the factorizations.

FP78: LZ78 version of LZW-FP. Briefly speaking, we apply the flexible parsing to 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 $T = R_1 \cdots R_{z_{78}}$. Given R_0 denotes the empty string, we write $\mathsf{e}(R_0) = 0$ and $\mathsf{e}(R_x) := \sum_{y=1}^{x} |R_y|$ for the ending position of R_x in T. Now suppose that we have parsed a prefix $T[1..i-1] = F_1 \cdots F_{x-1}$ with FP78 and want to compute F_x. At that time point, the dictionary $\mathcal{D}$ consists of all LZ78 factors that end before dst_x, appended by any character in Σ. 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 $F_x \cdot F_{x+1}$ is longest (ties are broken in favor of a longer F_x in case we have two candidates $F_x \cdot F_{x+1}$ and $F'_x \cdot F'_{x+1}$ with the same combined lengths). Note that at the time for selecting F_{x+1}, we can choose among all LZ78 factors that end before dst_{x+1}, which depends on $|F_x|$. Formally, let y be the index of the longest LZ78 factor R_y that is a prefix of $T[\mathsf{dst}_x..]$, i.e.,

$$y = \text{argmax}_{y' \in [1..z_{78}]}\{|R_{y'}| : R_{y'} = T[\mathsf{dst}_x..\mathsf{dst}_x + |R_{y'}| - 1] \wedge \mathsf{e}(R_{y'}) < \mathsf{dst}_x\}.$$

Then the maximal length of F_x is $|R_y| + 1$. For the flexible parsing, we do not greedily

select the maximal length of F_x, but instead determine the starting position of F_{x+1} in the interval $\mathcal{J} := [\mathsf{dst}_x + 1..\mathsf{dst}_x + |R_y| + 1]$. In detail, we select $\mathsf{dst}_{x+1} \in \mathcal{J}$ by

$$\mathsf{dst}_{x+1} := \text{argmax}_{p\in\mathcal{J}}\{p + |R_{y'}| : \exists y' \text{ with } R_{y'} = T[p..p + |R_{y'}| - 1] \wedge \mathsf{e}(R_{y'}) < p\}$$

such that the sum of the lengths $|F_x|+|F_{x+1}|$ is maximized. Then F_x has length $\mathsf{dst}_{x+1}-\mathsf{dst}_x$; if $|F_x| \geq 2$, then its reference exists since LZ78 is prefix-closed. By doing so, we obtain the factorization of FP78.

FPA78: LZ78 version of FPA. The other semi-greedy variant we study in its LZ78 version is the one of FPA, which we call FPA78. The only difference to FP78 is that the dictionary $\mathcal{D}$ is based on the LZ78-type factors $R'_1, \ldots, R'_{z'_{78}}$ starting at the FPA78 factor starting positions dst_x for all x. Formally, R'_x is the longest prefix of $T[\mathsf{dst}_x..]$ such that $R'_x = R'_y \cdot c$ for some $y \in [0..x-1]$ and $c \in \Sigma$, where dst_x is the starting position of the x-th factor of the FPA78 factorization. As a side note, the concatenation $R'_1 \cdots R'_{z'_{78}}$ is not the input text, in general, i.e., $R'_1, \ldots, R'_{z'_{78}}$ may not be a factorization of T.

The definition of $\mathcal{J}$ and p is identical to the above if we replace R_y with R'_y. Similarly, our task is to select the length $|F_x| \in [1..|R'_y| + 1]$ such that we advance the farthest in the text with a factor starting at $\mathsf{dst}_x + |F_x|$, for all such possible lengths $|F_x|$.

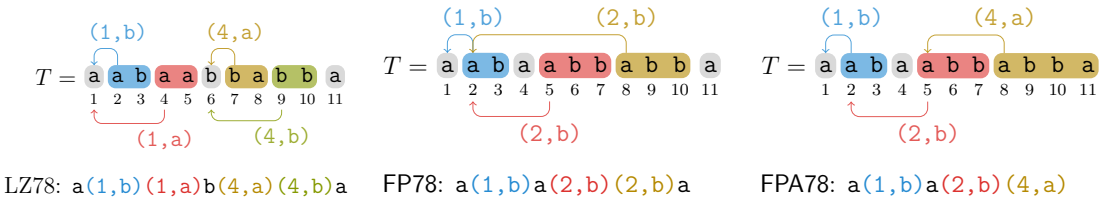


Figure 5: Visualization of the factorizations in Example 9.

Example 9. Let $T = \mathtt{aabaabbabba}$. The LZ78 factorization of T is $R_1 = \mathtt{a}, R_2 = \mathtt{ab} = R_1\mathtt{b}, R_3 = \mathtt{aa} = R_1\mathtt{a}, R_4 = \mathtt{b}, R_5 = \mathtt{ba} = R_4\mathtt{a}, R_6 = \mathtt{bb} = R_4\mathtt{b}, R_7 = \mathtt{a}$. FP78 builds upon $R_1 \cdots R_7$ to produce the factorization $F_1 = \mathtt{a}, F_2 = \mathtt{ab} = R_1\mathtt{b}, F_3 = \mathtt{a}, F_4 = \mathtt{abb} = R_2\mathtt{b}, F_5 = \mathtt{abb} = R_2b, F_6 = \mathtt{a}$. Noteworthy is the choice of F_3 being shorter than R_3 in the favor for creating a longer factor F_4 contrary to the 4-th factor R_4 of LZ78.

Finally, the FPA78 factorization is $F'_1 = \mathtt{a}, F'_2 = \mathtt{ab} = R'_1\mathtt{b}, F'_3 = \mathtt{a}, F'_4 = \mathtt{abb} = R'_2\mathtt{b}, F'_5 = \mathtt{abba} = R'_4\mathtt{a}$, where $R'_1 = \mathtt{a}$ starts at $T[1..]$, $R'_2 = \mathtt{ab}$ at $T[2..]$, $R'_3 = \mathtt{aa}$ at $T[4..]$, $R'_4 = \mathtt{abb}$ at $T[5..]$. Notable is the overlap of R'_3 and R'_4, which is because R'_j starts with F'_j for every j by definition. This choice allows F'_5 to refer to the preferably long reference R'_4. We could thus shorten the factorization size from 7 (LZ78) to 6 (FP78) and 5 (FPA78) with the flexible parsing variants. A visualization of all three factorizations is given in Fig. 5.

The difference in the number of factors can be greater than the constant experienced in Example 9. In fact, the following example gives a difference in $\Theta(n/\lg n)$.

Example 10. Select an LZ78-incompressible string S of length n on the alphabet $\{\mathtt{b}, \mathtt{c}\}$ such that the number of LZ78 factors of S is $\Theta(n/\lg n)$. Examples are binary de Bruijn sequences

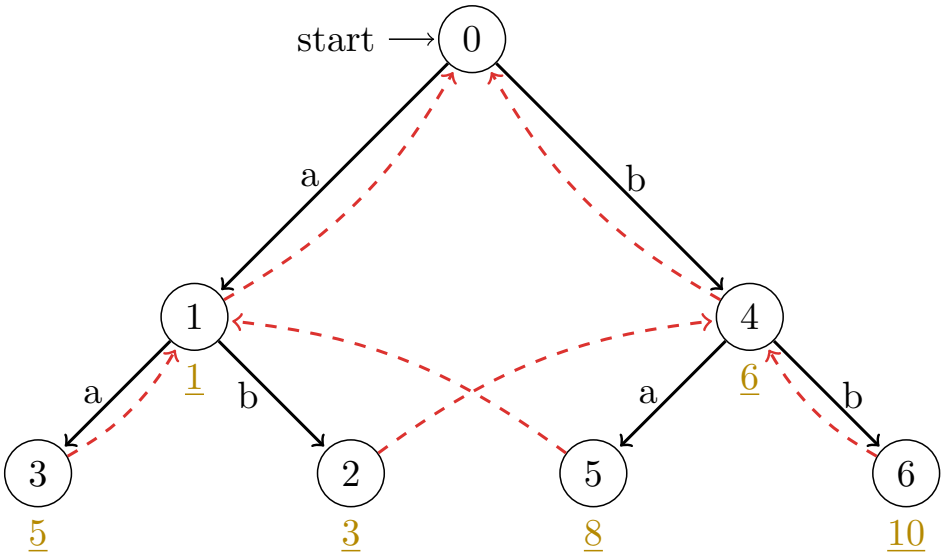


Figure 6: AC automaton of the LZ78 factors of $T = \mathtt{aabaabbabba}$ described in Example 9. A node is labeled by x if its string label is R_x. Except for the root node, each node is augmented by $\mathsf{e}(R_x)$, i.e., the factor ending position in the LZ78 factorization, which is drawn under the node in golden (■) color and underlined. Red (■) dashed arrows are suffix links.

or [83, Eq. (8)]. Suppose that we parse the string $T = \mathtt{a} \cdot \mathtt{a}S[1] \cdot \mathtt{a}S[1..2] \cdot \mathtt{a}S[1..3] \cdots \mathtt{a}S[1..n] \cdot \mathtt{a}S[1..n] \cdot \mathtt{a} \cdot S[1..n]$. On the one hand, the LZ78 factorization would create the factors $R_1 = \mathtt{a}$, $R_2 = \mathtt{a}S[1]$, $R_3 = \mathtt{a}S[1..2]$, …, $R_{n+1} = \mathtt{a}S[1..n]$, $R_{n+2} = \mathtt{a}S[1..n]\mathtt{a}$, and finally create the $\Theta(n/\lg n)$ factors for factorizing S. On the other hand, both flexible parsing variants have $F_x = R_x$ for all $x \in [1..n+1]$, but select $F_{n+2} = R_{n+1} = F_n S[n]$ because doing so maximizes the length of $F_{n+3} = \mathtt{a}S[1..n] = R_{n+1} = F_n S[n]$. While the flexible parses have thus $n+3$ factors, the LZ78 factorization has $n + 2 + \Theta(n/\lg n)$ factors.

Decompression. By storing the FP78 factors of T as a list of pairs like in LZ78 allows for reconstructing T. While literal factors can be restored by reading the stored characters, a referencing factor refers to a reference that is an LZ78 factor. We therefore need to compute the LZ78 factorization on the decoded output on-the-fly. For FPA78, for each referencing factor we read we need to restore an LZ78-type factor starting at the same position, which becomes a reference. This is not immediately possible if the read FP78 or FPA78 factor is shorter, so we trigger this LZ78 factor (resp. LZ78-type factor) computation when enough text has been decompressed. This is also the reason why, at the computation of an FP78 or FPA78 factor F_x, we can only choose references representing factors that *end before* the starting position dst_x of F_x.

4.1 Computation with AC automaton

For the computation of any of the two LZ78 flexible parsings, 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. Matias et al. [67, Section 3] achieved $t_{\mathcal{D}} = \mathcal{O}(1)$ expected time per text position by storing Karp–Rabin fingerprints [52] in a hash table to match the substrings considered 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 Matias and Sahinalp [66] claimed that LZW-FP, the flexible parsing of LZW, can be computed in linear time, no algorithmic details have been given.

However, it is also possible to set $t_{\mathcal{D}}$ to the time for a node-to-child traversal time in a trie implementation. For FP78, we build an Aho–Corasick (AC) automaton [3] on $\mathcal{D}$ just after we have computed the LZ78 factorization, so $\mathcal{D}$ stores all (classic) LZ78 factors $R_1, \ldots, R_{z_{78}}$. Roughly speaking, this AC automaton is the LZ trie augmented with suffix links. An example is given in Fig. 6. Like the LZ trie, the AC-automaton allows us to search for the longest matching prefix of a suffix $T[p..]$ in $\mathcal{O}(\ell t_{\mathcal{D}})$ time if ℓ is the length of this prefix, where $t_{\mathcal{D}}$ is the time for a node-to-child traversal. To this end, we traverse the LZ trie downward, matching characters of $T[p..]$ with edge labels. Given that we read ℓ characters and end up at a node v such that none of its outgoing edges match $T[p+\ell]$, then $T[p..p+\ell-1]$ is the longest LZ78 factor that is a prefix of $T[p..]$. Given that this factor is R_w, we say that v is the *locus* of R_w, or more generally: the locus of an LZ78 factor F is the LZ trie node whose string label is F.

With suffix links, the AC automaton allows us to select, from the locus of R_w in the trie, the longest proper suffix $T[p'..p+\ell-1]$ with $p' > p$ of the matched LZ78 factor R_w that is again an LZ78 factor R_u. Since $\mathcal{D}$ is prefix-closed, for all other LZ78 factors that would match with prefixes P of $T[k..]$ with $k \in [p+1..p'-1]$, P ends before R_u, and can be omitted for the computation of F_x. Nevertheless, we need to traverse downward from the locus of R_u to find the longest possible LZ78 factor that is a prefix of $T[p'..]$. The explained algorithm uses the following two queries on the AC automaton:

- $\mathsf{suffixlink}(w)$: returns the node whose string label is longest among all proper suffixes of w that are string labels of nodes of the LZ trie.
- $\mathsf{child}(v, c)$: returns the child of the LZ trie node v connected by an edge with label $c \in \Sigma$.

Our algorithm computes the first search window $T[p..p+\ell-1]$ by $\ell+1$ child queries, but then subsequently uses suffixlink to shrink the window from the left side or extend the window from the right side by child. The total number of suffixlink and child queries is thus bounded by the number of characters that we want to factorize.

To correctly compute F_x, we need to omit the nodes that correspond to factors R_y with $\mathsf{e}(R_y) \geq \mathsf{dst}_x$. To this end, we augment the node corresponding to R_y with $\mathsf{e}(R_y)$, for each y. When traversing the automaton for pattern matching to find the longest match of $T[p..]$

in $\mathcal{D}$, we stop the traversal as soon as we try to visit a node with reference index y and $\mathsf{e}(R_y) \geq \mathsf{dst}_x$. That is because the reference index of a child is larger than that of its parent, and thus the augmented values (i.e., the LZ78 factor indices) strictly grow when we descend in the AC automaton. Furthermore, while matching $T[\mathsf{dst}_x + \ell..]$, we may take a suffix link to a node v with reference index y and $\mathsf{e}(R_y) \geq \mathsf{dst}_x$; in this case we skip v and continue to take the suffix link of v (which involves increasing ℓ).

Finally, to perform a query in the AC automaton only once per text position, we keep some computed values in memory. In detail, after we have computed the factor F_x and want to compute F_{x+1}, we might already have computed the longest reference starting at a position of at least dst_{x+1} during the computation of F_x if $\mathsf{dst}_x + |R_w| > \mathsf{dst}_{x+1}$, where R_w is the longest LZ78 factor with $\mathsf{e}(R_w) < \mathsf{dst}_x$ that is also a prefix of $T[\mathsf{dst}_x..]$. Hence, we temporarily memoize the computed references for the range $[\mathsf{dst}_{x+1}..\mathsf{dst}_x + |R_w|] \subset \mathcal{J}$ so that we can skip the recomputation of the longest match when searching for the length of F_{x+1}. In detail, when computing F_x, we search, for each $s \in T[\mathsf{dst}_x + 1...\mathsf{dst}_x + |R_w|]$, the longest reference $R_{\pi(s)}$ starting at $T[s..]$ for $\pi(s) \in [1..z_{78}]$. After computing F_x, we keep the values $\pi(s)$ for $s > \mathsf{e}(F_x)$ in memory — by doing so we do not need to recompute these again when computing the reference candidates for F_{x+1}.

With memoization, we compute the same factorization as long as creating factors does *not* change the parsing dictionary. The memoized values are discarded after having factorized the respective text positions. The number of memoized positions is upper-bounded by the longest LZ78 factor length, which we can bound by $\mathcal{O}(\sqrt{n})$ — a factor length is $\mathcal{O}(\sqrt{n})$ because the fewest number of factors is $\Omega(\sqrt{n})$ achieved by factorizing unary strings, cf. [11, Sect. II B.].

Example 11. Reusing the string of Example 9, we build the AC automaton depicted in Fig. 6. The first two traversals are straightforward, but are described for completeness. With child, we traverse to the locus v of R_1 while reading from T the first character $\mathtt{a}$. However, the augmented value of node v is 1, which means that we can use R_1 only after having already parsed the first character of T. We conclude that F_1 is a literal factor.

For computing F_2, we traverse to the leaf λ with label 2 and string label $\mathtt{ab}$. Again, its augmented value is too high to serve as a reference for F_2. However, we can now make use of λ's parent node v, the locus of R_1, as a candidate for the reference of F_2. Alternatively, we can take the suffix link of λ to find a shorter reference in favor of the next factor to process. However, here we end up at the node with label 4 whose augmented value is too high to be a reference for F_2. We conclude that $F_2 = R_1 \cdot \mathtt{b}$.

We now parse the remaining text $\mathtt{aabbabba}$ from $\mathsf{dst}_3 = 4$. By traversing the AC automaton downward, we end at the locus $\bar{\lambda}$ of R_3, whose augmented value is again too high. We can again use $\bar{\lambda}$'s parent v as the reference for F_3. However, this time the suffix link also leads to v, and from there on we can match the next character $\mathtt{b}$ of the input (recall that we have already read $\mathtt{aa}$ to traverse to $\bar{\lambda}$) to visit the locus of R_2. Luckily, the augmented value of R_3 is $3 < \mathsf{dst}_3 + 1 = 5$, where $+1$ corresponds to the fact that we will refer to R_3 for the next factor F_4 in case we set $|F_3| = 1$. Since it is not possible to descend further from R_3, we conclude that selecting $F_3 = R_1$ and $F_4 = R_2\mathtt{b}$ gives the

combined length $|F_3F_4| = 1 + 3 = 4$. This choice is maximal, since for the alternative to select $F_3 = R_1\mathsf{a}$ we only obtain $F_4 = \mathsf{b}$, for which we need an extra traversal from the root — this is only needed for the first factor candidate; for all subsequent candidates we obtain the combined length via a combination of suffix links and downward traversals. The subsequent traversals work similarly, so that we obtain the factorization as depicted in Fig. 5.

Theorem 12. We can compute FP78 in $\mathcal{O}(nt_{\mathcal{D}})$ time with $\mathcal{O}((z + \delta) \lg n)$ bits of space, where $\delta = \mathcal{O}(\sqrt{n})$ is the number of memoized values. If we answer child queries with balanced binary search trees, then $t_{\mathcal{D}} = \mathcal{O}(\lg \sigma)$.

To adapt this approach to FPA78, we need a dynamic dictionary, which gets filled with the LZ78-type factors $R'_1, R'_2, \ldots$ starting with the FPA78 factors. It is possible to make the AC-automaton semi-dynamic in the sense that it supports adding new strings to $\mathcal{D}$, for which we are aware of two approaches [48, 69] whose running times depend on the actual strings indexed by the AC-automaton.

4.2 Computation with Suffix Trees

The algorithms we propose in the following are based on the superposition of the suffix tree with the LZ trie of [39], which we briefly review.

Suffix Tree Superimposition. An observation of Nakashima et al. [74, Sect. 3] is that the LZ trie is a connected subgraph of the suffix trie containing its root. That 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. However, in ST, 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 starts at the first character of the edge. To represent all implicit LZ trie nodes, it suffices to augment each ST edge e with a counter that counts the number of e's implicit LZ trie nodes. We call this counter an *exploration counter*, and we write $n_v \in [0..|e|]$ for the exploration counter of an edge $e = (u, v)$, which is stored in the lower node v that e connects to. Here, $|e|$ denotes the length of e's label. The *locus* of an LZ factor F is a node v whose string label is F (then F has an explicit LZ trie node represented by v) or contains F as a proper prefix (then F is represented implicitly by the exploration counter n_v). Furthermore, we call an ST node v an *edge witness* if n_v increases 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, which means that v is an explicit LZ trie node. We also stipulate that the root of ST is an edge witness whose exploration counter is always full. Then all edge witnesses form a connected sub-graph of ST sharing the root node. See Fig. 8 for an example. Finally, Fischer et al. [39, 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 $\mathcal{O}(nt_{\mathrm{SA}})$-time deterministic solution for FP78 and FPA78, we make use of the superimposition of ST. The marking is done with the following data structure.

Lemma 13 ([25])**.** There is a semi-dynamic lowest marked 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.

Algorithm 2 Computing FPA78.

Require: query interval $\mathcal{I} = [1..n]$, root is marked
1: **if** $\mathcal{I} = \emptyset$ **then return**
2: $\lambda_1 \leftarrow$ select_leaf($\mathsf{b}(\mathcal{I})$) ▷ $\mathcal{O}(t_{SA})$ *time*
3: $w_1 \leftarrow$ lowest marked ancestor of λ_1
4: **if** $w_1 =$ root **then** ▷ *factor is literal*
5: **output** factor $T[\mathsf{b}(\mathcal{I})]$
6: **return** by recursing on $[\mathsf{b}(\mathcal{I}) + 1..\mathsf{e}(\mathcal{I})]$
7: $\ell_0 \leftarrow$ str_depth(parent(w_1)) $+ n_{w_1} + 1, p \leftarrow 0, \ell_{\max} \leftarrow \ell_0$
8: **for** $\ell_1 = 1$ to ℓ_0 **do** ▷ $\mathcal{O}(\ell_0 t_{SA})$ *time*
9: $\lambda_2 \leftarrow$ select_leaf($\mathsf{b}(\mathcal{I}) + \ell_1$)
10: $w_2 \leftarrow$ lowest marked ancestor of λ_2
11: $\ell_2 =$ str_depth(parent(w_2)) $+ n_{w_2} + 1$
12: **if** $\ell_1 + \ell_2 > \ell_{\max}$ **then**
13: $\ell_{\max} \leftarrow \ell_1 + \ell_2$
14: $p \leftarrow \ell_1$
15: create reference $T[\mathsf{b}(\mathcal{I})..\mathsf{b}(\mathcal{I}) + \ell_0]$ with reference ID stored in w_1 after having factorized up to position $\mathsf{b}(\mathcal{I}) + \ell_0$ ▷ *store the reference at its* ST *locus* v *and increment* n_v
16: **output** factor $T[\mathsf{b}(\mathcal{I})..\mathsf{b}(\mathcal{I}) + p - 1]$
17: **return** by recursing on the interval $[\mathsf{b}(\mathcal{I}) + p..]$

We use the data structure of Lemma 13 to issue the following query for both flexible parsing variants. For each suffix $T[i..]$, we query the lowest marked ancestor w of the leaf with suffix number i. Subsequently, we obtain the length of the longest reference that is a prefix of $T[i..]$ by taking w's parent u and adding u's exploration counter to u's string depth, which we obtain in $\mathcal{O}(t_{\mathrm{SA}})$ time. For FP78, we simultaneously compute the LZ78 factorization as described in [39, Sect. 4] such that all LZ78 edge witnesses of the parsed text $T[1..\mathsf{dst}_x - 1]$ are marked when computing the factor F_x. For FPA, we create an LZ trie node for the LZ78-type factor R' starting at dst_x after having processed text position $\mathsf{e}(R')$.

If we allow $\mathcal{O}(t_{\mathrm{SA}} \lg z)$ time per text position, we can omit the lowest marked ancestor data structure and use the exponential search [18] in ST, as described in [61, Section 4]. Unlike binary search, which takes $\mathcal{O}(\lg n)$ node visits along a root-to-leaf path on ST, which can be of length n, we query by doubling the path length, initially at 1, at each iteration

step until we find a not yet marked ancestor. Then we continue like in binary search to half the search space until we find the lowest marked ancestor on the root-to-leaf path. This search visits $\mathcal{O}(\lg z)$ nodes because the depth of the lowest marked node is at most z.

Since $|R_y| \leq y$, finding the reference R_y involves $\mathcal{O}(\lg y)$ node visits, and for each of them we pay $\mathcal{O}(t_{\mathrm{SA}})$ time for computing its string length. The total time is $\mathcal{O}(nt_{\mathrm{SA}} \lg z)$. Since the LZ78-dictionary and 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.

A pseudocode is given in Algorithm 2. We select the leaf with suffix number $\mathsf{b}(\mathcal{I})$ (Line 2) and retrieve its lowest marked ancestor w_1 (Line 3) with a string depth of $\ell_0 - 1$ (Line 7). Subsequently, we check for each possible factor length $\ell_1 = |F_x| \in [1..\ell_0]$ via a loop (Line 8) whether a subsequent factor F_{x+1} starting at $\mathsf{b}(\mathcal{I}) + \ell_1$ gives the largest advance $|F_x| + |F_{x+1}|$ from $\mathsf{b}(\mathcal{I})$ in the text. For that, select the leaf with suffix number $\mathsf{b}(\mathcal{I}) + |F_x|$ (Line 9) and retrieve its lowest marked ancestor w_2 (Line 10) with a string depth of $\ell_2 - 1$ (Line 11). Let us focus on the loop of Line 8 that takes $\mathcal{O}(\ell_0 t_{\mathrm{SA}})$ time per factor. Naively, this time bound can get asymptotically larger than $\mathcal{O}(|\mathcal{I}|t_{\mathrm{SA}})$ if we select only short factors $p \ll \ell_0$, where p is the computed factor length. However, we can memoize the results for the lowest marked ancestor for the queried range $T[\mathsf{b}(\mathcal{I})..\mathsf{b}(\mathcal{I}) + \ell_0]$ such that subsequent queries for the same positions can be performed in constant time until the respective text position has been factorized (the same argument as for the AC automaton approach in Sect. 4.1). In doing so, we process each position once. Another matter that needs to be taken care of is the dictionary of references, which we update in Line 15. The idea is to postpone this command for inserting the reference $T[\mathsf{b}(\mathcal{I})..\mathsf{b}(\mathcal{I}) + \ell_0]$ until text position $\mathsf{b}(\mathcal{I}) + \ell_0$ has been factorized (otherwise we might create a factor using this reference that cannot be decompressed). To insert the reference $R = T[\mathsf{b}(\mathcal{I})..\mathsf{b}(\mathcal{I}) + \ell_0]$, we proceed as follows.

1. Find R's locus w in ST, which is the lowest marked ancestor of the leaf λ_1 (the leaf with suffix number $\mathsf{b}(\mathcal{I})$).
2. If w is full:
   (a) select w's child u on the path downwards to λ_1 via $u \leftarrow \mathsf{level\_anc}(\lambda_1, \mathsf{depth}(w)+1)$,
   (b) mark u with the lowest marked ancestor data structure, and
   (c) exchange w with u for the next instruction.
3. Finally, store the index of R in w and increment n_w by one.

The difference from FP78 is what references the parsing dictionary stores. For FP78 it suffices to do the same steps but computing the LZ78 factors in the suffix tree simultaneously to FP78, so only Line 15 needs to be changed. We use an LZ78 algorithm (like [39]) that parses the text from left to right and can be paused at any step to obtain the first factors of the LZ78 factorization $T = R_1 \cdots R_{z_{78}}$.

Example 14. Continuing with the example of Example 9, to compute FP78, we build the suffix tree of $T = \texttt{aabaabbabba}$ and superimpose the LZ trie of the LZ78 factorization on the suffix tree while computing FP78. Assume that we have processed $T[1..3]$, and we want to compute F_3 with $\mathsf{dst}_3 = 4$ like in Example 9. The current state of the superimposition is shown in Fig. 7, where we have already computed the LZ78 factors R_1 and R_2. The next LZ78 factor R_3 will become available after processing position 5.

Now, we follow the steps of Algorithm 2: we select the leaf λ_1 with suffix number 4 (Line 2), which has label 7, and select its lowest marked ancestor w_1 corresponding to R_1 (Line 3). The string depth of w_1 is one, so we can potentially create a factor $F_3 = R_1 \cdot a$ of length two. We consider two cases (Line 8 onwards):

- If we let F_3 be of length $\ell_1 = 1$, then the next factor starts at position $\mathsf{b}(F_3) + \ell_1 = 5$. The leaf λ_2 with suffix number 5 has the lowest marked ancestor w_2 with label 8 and string label ab. Thus, the next factor can have length 3, and we have a combined length of 4.

- If we let F_3 be of length $\ell_1 = 2$, then the next factor starts at position $\mathsf{b}(F_3) + \ell_1 = 6$. We now mark the locus of $R_3 = \texttt{aa}$ in ST since we can make a reference to R_3 from now on. However, the leaf λ_2 with suffix number 6 has the root as the lowest marked ancestor. Thus, the next factor is a literal, and we have a combined length of 3.

We conclude that $F_3 = \texttt{a}$ and recurse. As a remark, without memoizing the computed references, we need to revert the marking of the locus of the references we computed while processing F_x but end after F_x.

While the AC automaton solution for FP78 represents the parsing dictionary with a static data structure, here we need updates due to the use of the lowest marked ancestor data structure for finding a selectable reference. It is possible to make the dictionary for the suffix tree-based approach also static by first computing the LZ78 factorization but augment the edge witnesses with the first ending positions of the LZ78 factors they present. We then use these augmented values as the weights of the nodes indexed by a data structure for weighted ancestor queries. A *weighted ancestor query* allows us to jump from a leaf λ to its lowest edge witness ancestor that represents a reference ending before the suffix number of λ, and it can be answered with the data structure of Lemma 7. However, the time and space complexities do not change for this variation.

Theorem 15. We can compute FP78 or FPA78 of T in $\mathcal{O}(n)$ time with $\mathcal{O}(n \lg n)$ bits of space, or in $\mathcal{O}(nt_{\mathrm{SA}} \lg z)$ 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 FP78 or FPA78, respectively. (For FP78, z here includes the LZ78 factors while FPA78 creates as many references as factors.)

5 LZ78 Factorization Variants

In what follows, we show applications of the technique of Sect. 4.2 for variants of the LZ78 factorization, and also give solutions to their substring compression variants.

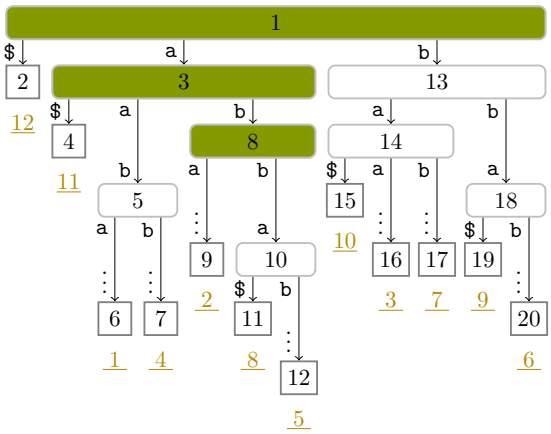


Figure 7: The suffix tree of $T =$ aabaabbabba described in Example 9 superimposed by the LZ trie of the LZ78 factorization of $T[1..3]$. This illustration is used in Example 14.

5.1 Lempel–Ziv Double (LZD)

LZD [44] is a variation of the LZ78 factorization. A factorization $F_1 \cdots F_z$ of T is LZD if $F_x = R_1 \cdot R_2$ with $R_1, R_2 \in \{F_1, \ldots, F_{x-1}\} \cup \Sigma$ such that R_1 and R_2 are, respectively, the longest possible prefixes of $T[\mathsf{dst}_x..]$ and of $T[\mathsf{dst}_x + |R_1|..]$, where 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. 9 for an example of the LZD factorization. Like in LZ78, references are factors. We postpone the exact definition of LZMW to the beginning of Sect. 5.2, but remark here that an LZMW factor is a variation of the LZD factorization with the restriction that a factor needs to refer to two consecutive factor indices.

A computational problem for LZD and LZMW is that their dictionaries are not prefix-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. 8). 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 [39], 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. 4.2 because some ST nodes on a path from the ST root to a locus of an LZD factor may not be edge witnesses. Thus, it seems that using the dynamic marked ancestor data structure of Lemma 13 is the only efficient way to accomplish this 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 for the locus of F_x, which is either represented by v, or is on the edge to v. To find this locus, it suffices to traverse the path from the root to the leaf with suffix number dst_x. The number of visited nodes is upper bounded by the factor length $|F_x|$, so 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 weighted ancestor data structure introduced in Lemma 7. Our result is as follows.

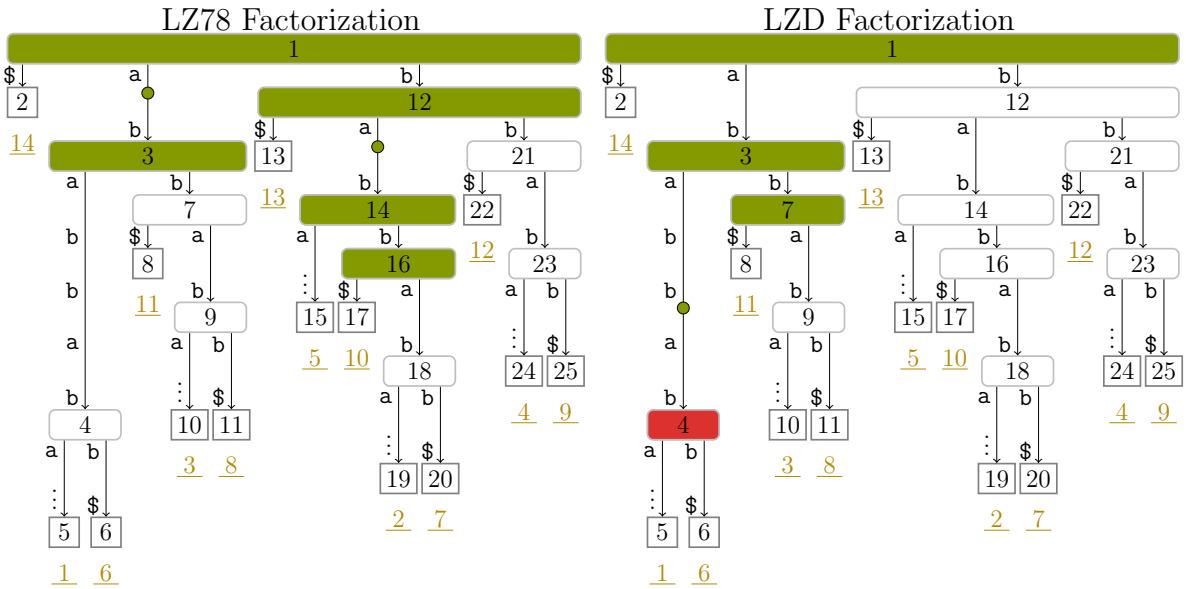


Figure 8: Suffix tree superimposition by the LZ trie of the LZ78 (left) and the LZD (right) factorization on our running example string $T = \texttt{ababbababbabb}$. Small circles in green (■) on edges are implicit LZ trie nodes. The ST nodes colored in green are full. In the LZD factorization, the edge witness with label 4 is colored in red (■); while not representing directly an LZ trie node, it witnesses factor ababb represented by a green circle on its incoming edge. The LZD factorization is not prefix-closed, for instance the nodes marked in green are not connected in the suffix *trie*.

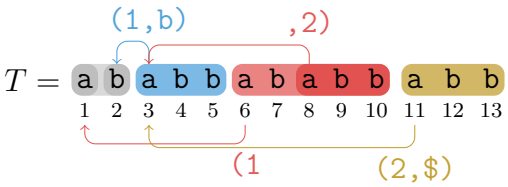
Theorem 16. There is a data structure that, given a query interval $\mathcal{I}$, can compute the LZD factorization in $\mathcal{O}(z_{\mathrm{D}[\mathcal{I}]})$ time, where $z_{\mathrm{D}[\mathcal{I}]}$ is the number of LZD factors. This data structure can be constructed in $\mathcal{O}(n)$ time, using $\mathcal{O}(n \lg n)$ bits of working space.

Proof. To obtain the claimed space and time bounds, we use the suffix tree augmented by

- a weighted ancestor query data structure (Lemma 7) and
- a lowest marked ancestor data structure (Lemma 13) to implement the superimposition of the LZ trie with the suffix tree storing the exploration counter.

On this augmented suffix tree, we perform the following steps, which we list in Algorithm 3. For each factor F_x we want to determine, compute first the lowest marked ancestor w_1 of the ST leaf λ_1 with suffix number dst_x (Line 3). Say the exploration counter of w_1 plus the string depth of w_1's parent is ℓ_1 (Line 8), compute the lowest marked ancestor w_2 of the leaf with suffix number $\mathsf{dst}_x + \ell_1$ (Line 11). Given the exploration counter of w_2 combined with the string depth of w_2's parent is ℓ_2 (Line 16), F_x has length $\ell_1 + \ell_2$, and its locus is on the path between w_1 and λ_1 on the string depth $\ell_1 + \ell_2$ (Line 18). After finding the locus u of F_x, we mark u, increase its exploration counter, and continue processing F_{x+1}.

In total, for each factor F_x, we use



LZD: (a,b)(1,b)(1,2)(2,$)

Figure 9: LZD factorization of our running example $T = \texttt{ababbababbabb}$, given by $F_1 = \texttt{ab}$, $F_2 = \texttt{abb} = F_1 \cdot \texttt{b}$, $F_3 = \texttt{ababb} = F_1 \cdot F_2$, and $F_4 = \texttt{abb\$} = F_2 \cdot \$$. 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.

- two lowest marked ancestor queries,
- a weighted ancestor query, and
- mark a node or increase its exploration counter.

Each step takes constant time. □

Algorithm 3 Computing LZD within the bounds claimed in Thm. 16.

Require: query interval $\mathcal{I}$, root is marked

if $\mathcal{I} = \emptyset$ **then return**
$\lambda_1 \leftarrow$ select_leaf(b($\mathcal{I}$)) ▷ $\mathcal{O}(t_{SA})$ *time*
$w_1 \leftarrow$ lowest marked ancestor of λ_1
if $w_1 =$ root **then** ▷ *left-hand side of factor is literal*
    $\ell_1 \leftarrow 1$
    output $T[\mathsf{b}(\mathcal{I})]$ as left-hand side of factor
else
    $\ell_1 \leftarrow$ str_depth(parent(w_1)) $+ n_{w_1}$
    output reference stored in w_1 as left-hand side of factor
$\lambda_2 \leftarrow$ select_leaf(b($\mathcal{I}$) $+ \ell_1$) ▷ $\mathcal{O}(t_{SA})$ *time*
$w_2 \leftarrow$ lowest marked ancestor of λ_2
if $w_2 =$ root **then** ▷ *right-hand side of factor is literal*
    $\ell_2 \leftarrow 1$
    output $T[\mathsf{b}(\mathcal{I}) + \ell_1]$ as right-hand side of factor
else
    $\ell_2 \leftarrow$ str_depth(parent(w_2)) $+ n_{w_2}$
    output reference stored in w_2 as right-hand side of factor
create reference for factor $T[\mathsf{b}(\mathcal{I})..\mathsf{b}(\mathcal{I}) + \ell_1 + \ell_2 - 1]$ ▷ *access the locus with weighted ancestor query*
return by recursing on the interval $[\mathsf{b}(\mathcal{I}) + \ell_1 + \ell_2..]$

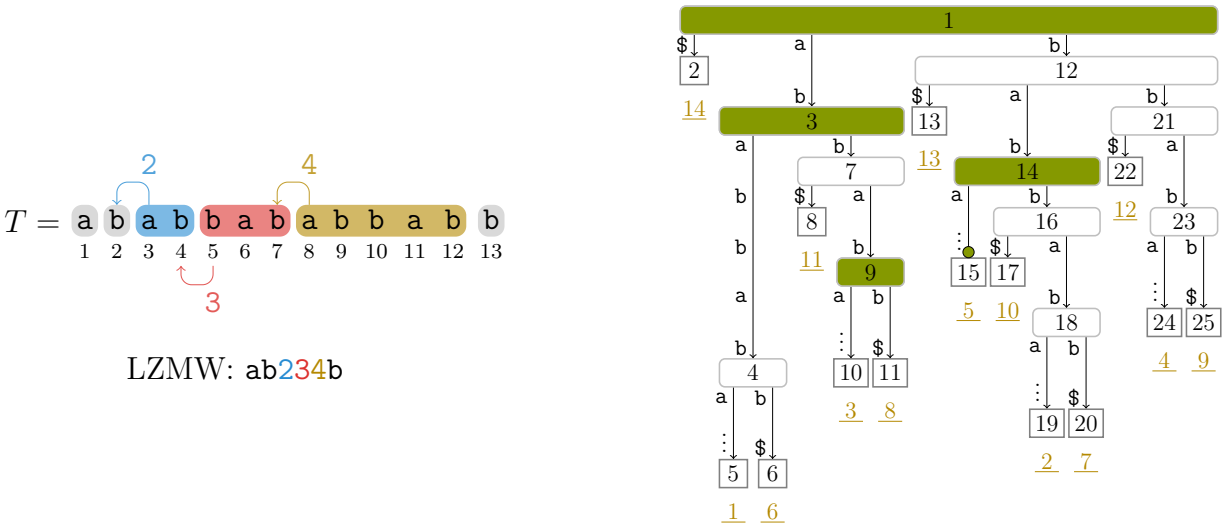


Figure 10: LZMW factorization of our running example $T = \mathtt{ababbababbabb}$, given by $F_1 = \mathtt{a}$, $F_2 = \mathtt{b}$, $F_3 = \mathtt{ab} = F_1F_2$, $F_4 = \mathtt{bab} = F_2F_3$, $F_5 = \mathtt{abbab} = F_3F_4$, and $F_6 = \mathtt{b}$. **Left:** We encode the factors such that a number denotes the index of a referred factor. Writing x means that we refer to the string $F_{x-1}F_x$. **Right:** ST superimposed by the LZ trie for the LZMW factorization decorating $F_x \cdot F_{x+1}$, where F_4F_5 is an implicit LZ trie node on the edge leading to the leaf with suffix number 5.

5.2 Lempel–Ziv–Miller–Wegman (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 $T[\mathsf{dst}_x..]$ with $F_x \in \{F_{y-1}F_y : y \in [2..x-1]\} \cup \Sigma$ where $\mathsf{dst}_x = 1 + \sum_{y=1}^{x-1} |F_y|$. 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. 10 for an example of the LZMW factorization.

Theorem 17. There is a data structure that, given a query interval $\mathcal{I}$, can compute LZMW in $\mathcal{O}(z_{\mathrm{MW}[\mathcal{I}]})$ time, where $z_{\mathrm{MW}[\mathcal{I}]}$ 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.

Proof. For each factor F_x, compute the lowest marked ancestor v of the ST leaf with suffix number 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. □

To change Algorithm 3 to compute the LZMW factorization, we remove all commands concerning the leaf λ_2 starting with Line 10, and store the reference $F_{x-1} \cdot F_x$ instead of just F_x at Line 18.

5.3 $\mathcal{O}(n \lg \sigma)$-Bits Solution

We can get down to $\mathcal{O}(n \lg \sigma)$ bits of space if we allow the time to linearly depend on n. Then we can afford to mark all ST ancestors of the loci of the computed LZD factors with an extra bit vector B of length n. By doing so, the set of nodes marked by B forms a sub-graph of the suffix tree, whose lowest nodes leading to a suffix tree leaf we can query by exponential search using level ancestor queries. We therefore can find w_1 in Line 3 of Algorithm 3 in $\mathcal{O}(\lg z)$ time. On the downside, we mark $\mathcal{O}(n)$ nodes in B during the computation of LZD.

Next, we need to get rid of (1) the lowest marked ancestor data structure and (2) the $\mathcal{O}(n \lg n)$-bit representation of the exploration counters. First, we overcome the computation bottleneck to find the locus u of the new factor we want to insert into the LZ trie. This locus u is on the path on the first selected leaf λ_1 and its lowest marked ancestor w_1. We can find u with a binary search in depth $d \mapsto \textsf{level\_anc}(\lambda_1, d)$ with str_depth being the key to evaluate. Finding u takes $\mathcal{O}(t_{\text{SA}} \lg n)$ additional time per factor.

Second, we store the exploration values of each edge witness in a dynamic balanced binary search tree using $\mathcal{O}(z \lg n) = \mathcal{O}(n \lg \sigma)$ bits of space, where $z = \mathcal{O}(n / \log_\sigma n)$ denotes the number of factors of LZD or LZMW. Lookups and updates of any exploration value thus cost $\mathcal{O}(\lg z) = \mathcal{O}(\lg n)$ time.

Theorem 18. We can compute the LZD or LZMW factorization in $\mathcal{O}(n t_{\text{SA}} \lg n) = \mathcal{O}(n \lg^{1+\epsilon} n)$ time, using $\mathcal{O}(n \lg \sigma)$ bits of working space with $t_{SA} = \mathcal{O}(\log_\sigma^\epsilon n)$ for a selectable constant $\epsilon \in (0, 1]$.

6 Evaluation

For the following evaluations, we used a machine with an Intel Xeon Gold 6330 CPU, Ubuntu 22.04, and gcc version 11.4. The evaluations were performed on datasets from the Pizza&Chili [37] corpus, the Canterbury corpus [7], and the Calgary corpus [16].

6.1 Flexible Parsing

As far as we are aware of, only compiled binaries for computing a binary-encoded LZW-FP or FPA factorization on Solaris and Irix platforms are available [4]. Here we provide source code for both flexible parsings written in Python at `https://github.com/koeppl/lz78flex`. The code is not algorithmically engineered, but suitable as a reference implementation. We conducted experiments to empirically assess the change in the number of factors compared to (classic) LZ78. In Table 5 we observe that there is no instance where any of the flexible parsing variants produces more factors than LZ78. Instead, the factorizations have up to 13% fewer factors than LZ78. Despite that, we have no guarantee about an upper bound on the number of FPA78 factors in contrast to FP78, FPA78 almost always has fewer factors than FP78; an exception is the dataset E.COLI.

[4] `https://www.dcs.warwick.ac.uk/~nasir/work/fp/`, accessed 12th of September 2024.

text	n [K]	z_{78} [K]	z_{FP78} [K]	$\frac{z_{\mathsf{FP78}}}{z_{78}}$	z_{FPA78} [K]	$\frac{z_{\mathsf{FPA78}}}{z_{78}}$
E.COLI	4638.69	491.11	488.36	99.44%	490.19	99.81%
ALICE29.TXT	148.48	28.73	27.87	97.03%	27.50	95.72%
ASYOULIK.TXT	125.18	25.59	24.82	97.00%	24.50	95.73%
BIB	111.26	21.46	20.40	95.05%	19.49	90.80%
BIBLE.TXT	4047.39	490.81	472.51	96.27%	453.89	92.48%
BOOK1	768.77	131.07	128.07	97.71%	126.94	96.85%
BOOK2	610.86	102.51	98.76	96.34%	96.11	93.75%
FIELDS.C	11.15	2.79	2.66	95.40%	2.58	92.46%
GRAMMAR.LSP	3.72	1.07	1.03	95.89%	0.98	91.13%
LCET10.TXT	419.24	71.12	68.78	96.71%	67.37	94.72%
PAPER1	53.16	12.17	11.74	96.52%	11.49	94.44%
PAPER2	82.20	17.34	16.81	96.97%	16.60	95.75%
PAPER3	46.53	10.91	10.60	97.21%	10.49	96.16%
PAPER4	13.29	3.65	3.53	96.74%	3.51	96.30%
PAPER5	11.95	3.41	3.30	96.74%	3.29	96.36%
PAPER6	38.11	9.15	8.82	96.41%	8.66	94.68%
PLRABN12.TXT	471.16	84.11	82.25	97.80%	81.54	96.95%
PROGC	39.61	9.46	9.09	96.13%	8.87	93.72%
PROGL	71.65	13.62	12.95	95.05%	12.43	91.24%
PROGP	49.38	9.81	9.32	94.94%	8.99	91.58%
WORLD192.TXT	2408.28	309.45	290.48	93.87%	269.63	87.13%
XARGS.1	4.23	1.34	1.30	97.02%	1.28	95.46%

Table 5: Flexible parsing variants of LZ78 compared with the LZ78 factorization on the number of factors of the given datasets. The text length and the number of factors are given in thousands (divided by 10^3, marked by [K]).

More research is necessary to determine the potential for compression, which involves the encoding of the factors. For instance, we should see similar relative numbers when we encode the factors with a fixed length code. However, with a variable length coding such as Huffman code, factors may need to be weighted appropriately to gain better compression.

6.2 Substring Compression

Curious about the practicality of our proposed solution with the suffix tree, we prepared a preliminary implementation based on the LZ78 factorization algorithm with suffix trees [39]. We call this implementation cics for *computation in compressed space*. For comparison, we took the LZ78 implementation in tudocomp [34] using a ternary trie [17] as the LZ78 trie implementation. We call this implementation ternary in the following. See [38] for implementation details of ternary. Both cics and ternary are implemented in C++ in the

text	n [M]	z_{78} [M]	ternary	cics
E.COLI	4.64	0.49	0.92	28.50
BIBLE.TXT	4.05	0.49	0.95	21.81
DBLP.XML.00001.1	104.86	3.94	1.40	35.23
DBLP.XML.00001.2	104.86	3.97	1.41	37.08
DBLP.XML.0001.1	104.86	3.95	1.43	40.22
DBLP.XML.0001.2	104.86	4.25	1.36	41.18
FIB41	267.91	0.42	13.56	289.62
FIB46	1836.31	1.52	33.73	1143.04
RS.13	216.75	0.44	10.60	223.19
TM29	268.44	0.62	9.61	230.44
WORLD192.TXT	2.47	0.31	0.92	15.20

Table 6: LZ78 factorization speed benchmark. File sizes in the second column are given in megabytes. The third column z_{78} denotes the number of computed factors divided by 10^6. The two last columns measure the average delay in microseconds per factor for each respective factorization algorithm ($\mu s/z$).

tudocomp framework available in the `cics` branch at `https://github.com/tudocomp/tudocomp`. As ST implementation, we use the class `cst_sada` of the SDSL library [42], a C++ implementation of the compressed suffix tree of [78].

In what follows, we benchmark the time needed for the substring compression problem of LZ78 when the entire text should be compressed. This is the best case scenario for an index for substring compression with respect to the comparison with a direct factorization algorithm.[5] That is because the direct factorization algorithm has to read the entire text, spending at least $\Omega(n)$ time, while the index should work in sublinear time for highly-compressible texts. In this setting, cics first builds ST at the precomputation time, while starting the factorization at query time. We therefore omit the construction of ST in the time benchmark. Nevertheless, we observe in Table 6 that ternary processes a factor faster on average on all datasets.

The reason for the bad performance of cics is the slow performance of queries such as parent, str_depth, and select_leaf. All these methods require in the `cst_sada` implementations unpredictable long jumps in memory because the underlying implementation heavily depends on rank/select support data structures. Unfortunately, memoizing the results of these calls in dynamic lookup tables did not give expected speedups. Despite poor performance, we are confident that ST implementations specifically addressing the experienced bottlenecks in answering these queries will improve the performance of ST-based factorization algorithms.

To draw a conclusion, we observed that current suffix tree implementations seem yet not suitable as index data structures for substring compression. The proposed algorithm

[5]With *direct* we emphasize that any precomputation is part of the query.

working on suffix trees was always slower than a straightforward factorization algorithm that does not take advantage of any precomputation. In that light, we did not proceed further with implementations of suffix tree-based approaches for the other factorizations addressed in this article.

7 Outlook

An open problem is whether we can compute LZD or LZMW in $\mathcal{O}(z)$ space within $\mathcal{O}(n)$ time, which would be optimal in the general case. We also wonder about the minimum space required to represent the tables LPF of all suffixes of T.

LZD Derivates. The idea of LZD spawns new research directions.

- The first question would be whether it makes sense to generalize LZD to refer to at most k factors instead of at most two. By doing so, the string lengths in the parse dictionary can grow much faster. It seems that the computational time complexity also linearly depends on k.
- A problem with the optimality of LZD is that it is not prefix-free, which is a property that can be exploited to find strings for which LZD gives a grammar that is larger than the size of the smallest grammar by a factor of $\Omega(n^3)$ [10]. Also note that the flexible parsing applied to LZ78 or LZW makes both parsings optimal with respect to the minimum number of factors, which, however, does not apply to LZD or LZMW, since both parsings are not prefix-closed in general.
- It is possible to extend LZD to a collage system [54], where we are also allowed to select prefixes of any element in the dictionary as a factor. Such an extension, paired with the flexible parsing, may give better lower bounds on the sizes (now with respect to the smallest collage system), and might also be worth to try in practice.

Other Factorizations. It is interesting to study substring compression queries or internal queries for other factorizations such as block palindrome factorizations [45] or repetition factorizations [50].

Acknowledgements This research was supported by JSPS KAKENHI with grant numbers `JP23H04378` and `JP25K21150`. We thank the anonymous reviewers of DCC'24 and Information Systems for their careful reading of this manuscript and their insightful comments and suggestions.

References

[1] P. Abedin, A. Ganguly, S. P. Pissis, and S. V. Thankachan. Range shortest unique substring queries. In *Proc. SPIRE*, volume 11811 of *LNCS*, pages 258–266, 2019. doi: 10.1007/978-3-030-32686-9_18.

[2] P. Abedin, A. Ganguly, W. Hon, K. Matsuda, Y. Nekrich, K. Sadakane, R. Shah, and S. V. Thankachan. A linear-space data structure for range-LCP queries in poly-logarithmic time. *Theor. Comput. Sci.*, 822:15–22, 2020. doi: 10.1016/j.tcs.2020.04.009.

[3] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi: 10.1145/360825.360855.

[4] A. Amir, A. Apostolico, G. M. Landau, A. Levy, M. Lewenstein, and E. Porat. Range LCP. *J. Comput. Syst. Sci.*, 80(7):1245–1253, 2014. doi: 10.1016/j.jcss.2014.02.010.

[5] A. Amir, M. Lewenstein, and S. V. Thankachan. Range LCP queries revisited. In *Proc. SPIRE*, volume 9309 of *LNCS*, pages 350–361, 2015. doi: 10.1007/978-3-319-23826-5_33.

[6] A. Amir, P. Charalampopoulos, S. P. Pissis, and J. Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi: 10.1007/s00453-020-00744-0.

[7] R. Arnold and T. C. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proc. DCC*, pages 201–210, 1997. doi: 10.1109/DCC.1997.582019.

[8] M. A. Babenko, P. Gawrychowski, T. Kociumaka, and T. Starikovskaya. Wavelet trees meet suffix trees. In *Proc. SODA*, pages 572–591, 2015. doi: 10.1137/1.9781611973730.39.

[9] G. Badkobeh, H. Bannai, K. Goto, T. I, C. S. Iliopoulos, S. Inenaga, S. J. Puglisi, and S. Sugimoto. Closed factorization. *Discret. Appl. Math.*, 212:23–29, 2016. doi: 10.1016/j.dam.2016.04.009.

[10] 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. doi: 10.1007/978-3-319-67428-5_5.

[11] H. Bannai, M. Hirayama, D. Hucke, S. Inenaga, A. Jez, M. Lohrey, and C. P. Reh. The smallest grammar problem revisited. *IEEE Trans. Inf. Theory*, 67(1):317–328, 2021. doi: 10.1109/TIT.2020.3038147.

[12] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014. doi: 10.1007/s00453-012-9726-3.

[13] T. Baumann and T. Hagerup. Rank-select indices without tears. In *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, volume 11646 of *LNCS*, pages 85–98, 2019. doi: 10.1007/978-3-030-24766-9_7.

[14] D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel–Ziv parsing. In *Proc. SODA*, pages 2053–2071, 2016. doi: 10.1137/1.9781611974331.ch143.

[15] 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. doi: 10.4230/LIPIcs.CPM.2021.8.

[16] T. C. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Comput. Surv.*, 21(4):557–591, 1989. doi: 10.1145/76894.76896.

[17] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. SODA*, pages 360–369, 1997.

[18] J. L. Bentley and A. C. Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976. doi: 10.1016/0020-0190(76)90071-5.

[19] P. Bille, I. L. Gørtz, and T. A. Steiner. String indexing with compressed patterns. In *Proc. STACS*, volume 154 of *LIPIcs*, pages 10:1–10:13, 2020. doi: 10.4230/LIPIcs.STACS.2020.10.

[20] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985. doi: 10.1016/0304-3975(85)90157-4.

[21] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[22] P. Charalampopoulos, T. Kociumaka, M. Mohamed, J. Radoszewski, W. Rytter, J. Straszynski, T. Walen, and W. Zuba. Counting distinct patterns in internal dictionary matching. In *Proc. CPM*, volume 161 of *LIPIcs*, pages 8:1–8:15, 2020. doi: 10.4230/LIPIcs.CPM.2020.8.

[23] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958.

[24] D. R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.

[25] R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi: 10.1137/S0097539700370539.

[26] G. Cormode and S. Muthukrishnan. Substring compression problems. In *Proc. SODA*, pages 321–330, 2005.

[27] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008. doi: 10.1016/j.ipl.2007.10.006.

[28] M. Crochemore, M. Kubica, T. Walen, C. S. Iliopoulos, and M. S. Rahman. Finding patterns in given intervals. *Fundam. Informaticae*, 101(3):173–186, 2010. doi: 10.3233/FI-2010-283.

[29] M. Crochemore, C. S. Iliopoulos, M. Kubica, M. S. Rahman, G. Tischler, and T. Walen. Improved algorithms for the range next value problem and applications. *Theor. Comput. Sci.*, 434:23–34, 2012. doi: 10.1016/J.TCS.2012.02.015.

[30] M. Crochemore, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Walen. Efficient algorithms for three variants of the LPF table. *J. Discrete Algorithms*, 11:51–61, 2012. doi: 10.1016/j.jda.2011.02.002.

[31] M. Crochemore, C. S. Iliopoulos, J. Radoszewski, W. Rytter, J. Straszynski, T. Walen, and W. Zuba. Internal quasiperiod queries. In *Proc. SPIRE*, volume 12303 of *LNCS*, pages 60–75, 2020. doi: 10.1007/978-3-030-59212-7_5.

[32] 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. doi: 10.1007/978-3-642-32241-9_34.

[33] 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. doi: 10.1137/1.9781611977912.121.

[34] P. Dinklage, J. Fischer, D. Köppl, M. Löbel, and K. Sadakane. Compression with the tudocomp framework. In *Proc. SEA*, volume 75 of *LIPIcs*, pages 13:1–13:22, 2017. doi: 10.4230/LIPIcs.SEA.2017.13.

[35] P. Dinklage, J. Ellert, J. Fischer, D. Köppl, and M. Penschuck. Bidirectional text compression in external memory. In *Proc. ESA*, pages 41:1–41:16, 2019. doi: 10.4230/LIPIcs.ESA.2019.41.

[36] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi: 10.1145/355541.355547.

[37] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13:1.12:1 – 1.12:31, 2008. doi: 10.1145/1412228.1455268.

[38] J. Fischer and D. Köppl. Practical evaluation of Lempel–Ziv-78 and Lempel–Ziv–Welch tries. In *Proc. SPIRE*, volume 9472 of *LNCS*, pages 191–207, 2017. doi: 10.1007/978-3-319-67428-5_16.

[39] J. Fischer, T. I, D. Köppl, and K. Sadakane. Lempel–Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018. doi: 10.1007/s00453-017-0333-1.

[40] Y. Gao, M. He, and Y. Nekrich. Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 54:1–54:18, 2020. doi: 10.4230/LIPICS.ESA.2020.54.

[41] P. Gawrychowski, M. Lewenstein, and P. K. Nicholson. Weighted ancestors in suffix trees. In *Proc. ESA*, volume 8737 of *LNCS*, pages 455–466, 2014. doi: 10.1007/978-3-662-44777-2_38.

[42] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, volume 8504 of *LNCS*, pages 326–337, 2014. doi: 10.1007/978-3-319-07959-2_28.

[43] A. Golynski, R. Raman, and S. S. Rao. On the redundancy of succinct data structures. In *Proc. SWAT*, volume 5124 of *LNCS*, pages 148–159, 2008. doi: 10.1007/978-3-540-69903-3_15.

[44] 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. doi: 10.1007/978-3-319-19929-0_19.

[45] K. Goto, T. I, H. Bannai, and S. Inenaga. Block palindromes: A new generalization of palindromes. In *Proc. SPIRE*, volume 11147 of *LNCS*, pages 183–190, 2018. doi: 10.1007/978-3-030-00479-8_15.

[46] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi: 10.1137/S0097539702402354.

[47] A. Hartman and M. Rodeh. Optimal parsing of strings. In *Combinatorial Algorithms on Words*, pages 155–167, Berlin, Heidelberg, 1985.

[48] D. Hendrian, S. Inenaga, R. Yoshinaka, and A. Shinohara. Efficient dynamic dictionary matching with DAWGs and AC-automata. *Theor. Comput. Sci.*, 792:161–172, 2019. doi: 10.1016/j.tcs.2018.04.016.

[49] R. N. Horspool. The effect of non-greedy parsing in Ziv–Lempel compression methods. In *Proc. DCC*, pages 302–311, 1995. doi: 10.1109/DCC.1995.515520.

[50] H. Inoue, Y. Matsuoka, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Computing smallest and largest repetition factorizations in $o(n \log n)$ time. In *Proc. PSC*, pages 135–145, 2016.

[51] G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989. doi: 10.1109/SFCS.1989.63533.

[52] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi: 10.1147/rd.312.0249.

[53] O. Keller, T. Kopelowitz, S. L. Feibish, and M. Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:42–54, 2014. doi: 10.1016/j.tcs.2013.10.010.

[54] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theor. Comput. Sci.*, 298(1):253–272, 2003. doi: 10.1016/S0304-3975(02)00426-7.

[55] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005. doi: 10.1016/j.jda.2004.08.002.

[56] T. Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *Proc. CPM*, volume 54 of *LIPIcs*, pages 28:1–28:12, 2016. doi: 10.4230/LIPIcs.CPM.2016.28.

[57] T. Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, 2018.

[58] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. Internal pattern matching queries in a text and applications. In *Proc. SODA*, pages 532–551, 2015. doi: 10.1137/1.9781611973730.36.

[59] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. Internal pattern matching queries in a text and applications. *CoRR*, abs/1311.6235v5, 2023.

[60] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. Internal pattern matching queries in a text and applications. *SIAM J. Comput.*, 53(5):1524–1577, 2024. doi: 10.1137/23M1567618.

[61] D. Köppl. Non-overlapping LZ77 factorization and LZ78 substring compression queries with suffix trees. *Algorithms*, 14(2)(44):1–21, 2021. doi: 10.3390/a14020044.

[62] D. Köppl. Computing lexicographic parsings. In *Proc. DCC*, pages 232–241, 2022. doi: 10.1109/DCC52660.2022.00031.

[63] D. Köppl. Computing LZ78-derivates with suffix trees. In *Proc. DCC*, pages 133–142, 2024. doi: 10.1109/DCC58796.2024.00021.

[64] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel–Ziv compression of genomes for large-scale storage and retrieval. In *Proc. SPIRE*, volume 6393 of *LNCS*, pages 201–206, 2010. doi: 10.1007/978-3-642-16321-0_20.

[65] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi: 10.1137/0222058.

[66] Y. Matias and S. C. Sahinalp. On the optimality of parsing in dynamic dictionary based data compression. In *Proc. SODA*, pages 943–944, 1999.

[67] 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. doi: 10.1145/945394.945404.

[68] K. Matsuda, K. Sadakane, T. Starikovskaya, and M. Tateshita. Compressed orthogonal search on suffix arrays with applications to range LCP. In *Proc. CPM*, volume 161 of *LIPIcs*, pages 23:1–23:13, 2020. doi: 10.4230/LIPIcs.CPM.2020.23.

[69] B. Meyer. Incremental string matching. *Inf. Process. Lett.*, 21(5):219–227, 1985. doi: 10.1016/0020-0190(85)90088-2.

[70] V. S. Miller and M. N. Wegman. Variations on a theme by Ziv and Lempel. In *Combinatorial Algorithms on Words*, pages 131–140, Berlin, Heidelberg, 1985.

[71] K. Mitani, T. Mieno, K. Seto, and T. Horiyama. Internal longest palindrome queries in optimal time. In *Proc. WALCOM*, volume 13973 of *LNCS*, pages 127–138, 2023. doi: 10.1007/978-3-031-27051-2_12.

[72] J. I. Munro, Y. Nekrich, and J. S. Vitter. Fast construction of wavelet trees. *Theor. Comput. Sci.*, 638:91–97, 2016. doi: 10.1016/j.tcs.2015.11.011.

[73] J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. SODA*, pages 408–424, 2017. doi: 10.1137/1.9781611974782.26.

[74] 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. doi: 10.1016/j.ipl.2015.04.002.

[75] G. Navarro, C. Ochoa, and N. Prezza. On the approximation ratio of ordered parsings. *IEEE Trans. Inf. Theory*, 67(2):1008–1026, 2021. doi: 10.1109/TIT.2020.3042746.

[76] Y. Nekrich and G. Navarro. Sorted range reporting. In *Proc. SWAT*, volume 7357 of *LNCS*, pages 271–282, 2012. doi: 10.1007/978-3-642-31155-0_24.

[77] M. Patil, R. Shah, and S. V. Thankachan. Faster range LCP queries. In *Proc. SPIRE*, volume 8214 of *LNCS*, pages 263–270, 2013. doi: 10.1007/978-3-319-02432-5_29.

[78] K. Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

[79] H. Shibata and D. Köppl. LZ78 substring compression with CDAWGs. In *Proc. SPIRE*, volume 14899 of *LNCS*, pages 289–305, 2024. doi: 10.1007/978-3-031-72200-4_22.

[80] J. A. Storer and T. G. Szymanski. The macro model for data compression (extended abstract). In *Proc. STOC*, pages 30–39, 1978. doi: 10.1145/800133.804329.

[81] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17 (6):8–19, 1984. doi: 10.1109/MC.1984.1659158.

[82] G. Zhou. Two-dimensional range successor in optimal time and almost linear space. *Inf. Process. Lett.*, 116(2):171–174, 2016. doi: 10.1016/J.IPL.2015.09.002.

[83] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978. doi: 10.1109/TIT.1978.1055934.