Lempel-Ziv Factorization Powered by Space Efficient Suffix Trees*

Johannes Fischer[1], Tomohiro I[2], Dominik Köppl[1], and Kunihiko Sadakane[3]

[1]Department of Computer Science, TU Dortmund, Germany
[2]Kyushu Institute of Technology, Japan
[3]Graduate School of Information Science and Technology, University of Tokyo, Japan

Abstract

We show that both the Lempel-Ziv-77 and the Lempel-Ziv-78 factorization of a text of length n on an integer alphabet of size σ can be computed in $\mathcal{O}(n)$ time with either $\mathcal{O}(n \lg \sigma)$ bits of working space, or $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits (for a constant $\epsilon > 0$) of working space (including the space for the output, but not the text).

1 Introduction

Central to many compression algorithms are the LZ77 [60] or LZ78 [61] factorizations. Both techniques were invented in the late 70's and set a milestone in the field of data compression. They have also been found to be valuable tools for detecting various kinds of regularities in strings [8, 42, 36, 37, 24, 10, 35], for indexing [30, 29, 18, 19, 46, 13], and for analyzing strings [9, 39, 40].

Although both factorizations are well-studied, there are still improvements being made in computing them with respect to space and time. Computing the factorizations efficiently is of practical interest, since main memory sizes of ordinary computers do not scale as fast as the sizes of commonly maintained datasets. Both huge mainframes with massive datasets and tiny embedded systems are valid examples for which a simple compressor may end up depleting all available RAM. Additionally, compression algorithms with low memory footprints are good candidates for compressing transient data kept in memory. For instance, the zram module of modern Linux kernels [53] compresses blocks of the main memory in order to prevent the system from running out of working memory. Compressing RAM is sometimes preferable to storing transient data on secondary storage (e.g., in a swap file), as the latter poses a more severe performance loss. Another example is transferring websites as "gzipped" data by hosting servers [52]. A server may cache generated web pages in a compressed form in RAM for performance benefits.

To solve this problem, one has to think either about external memory compression algorithms, or about how to slim down memory consumption during computation in RAM. In this article we focus on the latter approach.

*Parts of this work have already been presented at the 26th Annual Symposium on Combinatorial Pattern Matching [16], and at the 2016 Data Compression Conference [38].

Time	Working Space	Ref.
$\mathcal{O}(n)$	$3n\lg n$	[21]
$\mathcal{O}(n)$	$2n\lg n$	[32]
$\mathcal{O}(n)$	$n\lg n+\mathcal{O}(\sigma\lg n)$	[22]
$\mathcal{O}(n)$	$(1+\epsilon)n\lg n+\mathcal{O}(n)$	Cor. 3.7
$\mathcal{O}(n)$	$\mathcal{O}(n\lg\sigma)$	Cor. 3.4
$\mathcal{O}\left(n\frac{\lg n}{\lg\lg n}\right)$	$n\lg n+n+o(n)$	[22, 48]
$\mathcal{O}(n\lg\sigma)$	$(1+\epsilon)n\lg n+n+\mathcal{O}(\sigma\lg n)$	[34]
$\mathcal{O}(n\lg_\sigma n)$	$\mathcal{O}(n\lg\sigma)$	[33]
$\mathcal{O}(n\lg\lg\sigma)$	$\mathcal{O}(n\lg\sigma)$	[4]

(a) LZ77

Time	Working Space	Ref.
$\mathcal{O}(n)$	$\mathcal{O}(n\lg n)$	[45]
$\mathcal{O}(n)$	$(1+\epsilon)n\lg n+\mathcal{O}(n)$	Cor. 4.10
$\mathcal{O}(n)$	$\mathcal{O}(n\lg\sigma)$	Cor. 4.3
$\mathcal{O}(n\lg\sigma)$	$\mathcal{O}(z\lg z)$	folklore
$\mathcal{O}(n(\lg\sigma+\lg\lg n))$	$2z\lg z+z\lg\sigma+\mathcal{O}(z)$	[2]
$\mathcal{O}\left(\frac{n}{\lg_\sigma n}\frac{\lg^2\lg n}{\lg\lg\lg n}\right)$	$\mathcal{O}\left(n\frac{\lg\sigma+\lg\lg_\sigma n}{\lg_\sigma n}\right)$	[28]
$\mathcal{O}\left(n+z\frac{\lg^2\lg\sigma}{\lg\lg\lg\sigma}\right)$	$\mathcal{O}(z\lg z)$	[14]

(b) LZ78

Figure 1: Recent approaches in LZ77 and LZ78 factorization computation. The working spaces are measured in bits. The horizontal line separates algorithms running in linear time (above) from algorithms running in near-linear time (below).

Our Results. To this end, we use two suffix tree representations whose construction algorithms need only limited working space. The representations differ in the fact that one is favorable for small alphabets, and the other is independent of the alphabet size. Powered by two different suffix tree representations, we show that the LZ77 and the LZ78 factorization of a text of length n on an integer alphabet of size σ can be computed in linear time

- with $\mathcal{O}(n\lg\sigma)$ bits of working space[1] (Corollaries 3.4 and 4.3), or
- with $(1+\epsilon)n\lg n+\mathcal{O}(n)$ bits (for a constant $\epsilon>0$) working space (Corollaries 3.7 and 4.10).

The working space bounds of the algorithms using $\mathcal{O}(n\lg\sigma)$ bits are due to the suffix tree: Given that we have constant time access to the Ψ function and to the tree navigation operations of the suffix tree, we can compute both factorizations in linear time using $z\lg n+\mathcal{O}(n)$ additional bits of working space (Theorems 3.3 and 4.2).

Related Work. While there are naive algorithms taking $\mathcal{O}(1)$ working space with quadratic running time (for both LZ77 and LZ78), algorithms with very restricted space and (nearly) linear time have only emerged in recent years (see Figure 1).

In the following section we give an overview of the most recent approaches for algorithms computing the LZ77 and LZ78 factorization.

First, let us look at the LZ77 factorization algorithms that run in linear time. There, Goto and Bannai [21] showed an algorithm using $3n\lg n$ bits. This bound was very soon lowered to $2n\lg n$ by Kärkkäinen et al. [32]. With similar techniques, Goto and Bannai [22] proposed later a solution with $n\lg n+\mathcal{O}(\sigma\lg n)$ bits, which is compelling if the alphabet size σ is small. The common idea of the above papers is the usage of previous- and/or next-smaller-value-queries [51]. While Kärkkäinen et al. [32] need to hold the suffix array and the next smaller value array in two arrays, Goto and Bannai [22] can cope with a single $n\lg n$ bits array and

[1] In the initial submission, the $\mathcal{O}(n)$ deterministic time suffix tree construction algorithm of Munro et al. [44] was not yet published. Our former results were $\mathcal{O}(n)$ randomized or $\mathcal{O}(n\lg\lg\sigma)$ deterministic time based on the suffix tree construction algorithm of Belazzougui [3].

an auxiliary array of size $\mathcal{O}(\sigma \lg n)$ bits. This auxiliary array is used to store the bucket boundaries used by the suffix array construction algorithm of Nong [50]. The bucket boundaries could also be represented by a dynamic bit vector [48] of length n yielding $n + o(n)$ additional bits (instead of the $\mathcal{O}(\sigma \lg n)$ bits used by the auxiliary array) of working space and $\mathcal{O}(n \lg n / \lg \lg n)$ time.

Another algorihm close to linear time is proposed by Kempa and Puglisi [34]. They show a practical variant with $(1+\epsilon)n \lg n + n + \mathcal{O}(\sigma \lg n)$ bits of working space and $\mathcal{O}\big(n \lg \sigma / \epsilon^2\big)$ time. There is also a trade-off algorithm given by Kärkkäinen et al. [33], using $\mathcal{O}(n/d)$ words of working space and $\mathcal{O}(dn)$ time. By setting $d = \log_\sigma n$ we get $\mathcal{O}(n \lg \sigma)$ bits of working space and $\mathcal{O}(n \log_\sigma n)$ time. The algorithm of Belazzougui and Puglisi [4] derives its dominant terms in space and time from the compressed suffix tree construction algorithm of Belazzougui [3]; they showed an algorithm using $\mathcal{O}(n \lg \sigma)$ bits and $\mathcal{O}(n)$ randomized or $\mathcal{O}(n \lg \lg \sigma)$ deterministic time. We expect that their algorithm runs in deterministic linear time by exchanging the suffix tree construction algorithm of [3] with an improved version [44].

The LZ78 computation is done with different techniques. A classic approach is to build a trie maintaining the factors during the computation. Storing z factors in a simple pointer-based trie data structure takes $\mathcal{O}(z \lg z) = \mathcal{O}(n \lg \sigma)$ bits (see Lemma 2.5 why this holds) and $\mathcal{O}(n \lg \sigma)$ time, if the children of a node are maintained in a sorted list. Improving LZ78 computation can be done by using sophisticated trie implementations [14, 28], or by superimposing the suffix tree with the suffix trie [45].

Following the former approach, Jansson et al. [28] proposed a compressed dynamic trie based on word packing, and showed an application computing the LZ78 factorization with $\mathcal{O}(n(\lg \sigma + \lg \lg_\sigma n)/ \lg_\sigma n)$ bits of working space and $\mathcal{O}\big(n \lg^2 \lg n / \left(\lg_\sigma n \lg \lg \lg n\right)\big)$ time. When $\lg \sigma = o\big(\lg n \lg \lg \lg n / \lg^2 \lg n\big)$, their algorithm runs even in sub-linear time, but in the worst case it is super-linear. More sophisticated trie implementations [14] improve this to $\mathcal{O}\big(n + z \lg^2 \lg \sigma / \lg \lg \lg \sigma\big)$ time, while using $\mathcal{O}(z \lg z)$ bits of space like the naive trie implementation. The time bound $\mathcal{O}\big(n + z \lg^2 \lg \sigma / \lg \lg \lg \sigma\big)$ gets linear when $\lg \sigma = o\big(\lg n \lg \lg \lg n / \lg^2 \lg n\big)$. A version with succinct data structures is shown by Arroyuelo and Navarro [2] using $2z \lg z + z \lg \sigma + \mathcal{O}(z)$ bits of working space, but $\mathcal{O}(n(\lg \sigma + \lg \lg n))$ time.

We follow the latter approach that superimposes the suffix tree with the suffix trie. There, Nakashima et al. [45] recently proposed a linear time algorithm. Although their algorithm works with $\mathcal{O}(n \lg n)$ bits of space, they did not care about the constant factor, and the use of the (complicated) dynamic marked ancestor queries [1] seems to prevent them from achieving a small constant factor.

2 Preliminaries

Our computational model is the word RAM model with word size $\Omega(\lg n)$ for some natural number n. Accessing a word costs $\mathcal{O}(1)$ time.

Let Σ denote an integer alphabet of size $\sigma = |\Sigma| = n^{\mathcal{O}(1)}$. We call an element $T \in \Sigma^*$ a ***string*** or ***text***. Its length is denoted by $|T|$. The empty string is λ with $|\lambda| = 0$. We access the j-th character of T with $T[j]$. Given $x, y, z \in \Sigma^*$ with $T = xyz$, then x, y and z are called a ***prefix***, a ***substring***, and a ***suffix*** of T, respectively. In particular, the suffix starting at position j of T is called the ***j-th suffix*** of T. For an integer j with $1 \le j \le |T|$, let $\mathcal{S}_j\left(T\right)$ denote the set of substrings of T that start strictly before j. The length of the ***longest common prefix*** of two strings S and T is the value i such that $T[1..i] = S[1..i]$ and either $T[i+1] \neq S[i+1]$ or $i = \min\left(|T|, |S|\right)$ hold.

For a binary string $T \in \{0,1\}^*$ and a bit $c \in \{0,1\}$, we are interested in computing the following operations:

- $T.\operatorname{rank}_c(j)$ counts the number of 'c's in $T[1..j]$, and
- $T.\operatorname{select}_c(j)$ gives the position of the j-th 'c' in T.

There are data structures [26, 7] that use $o(|T|)$ extra bits of space, and can compute rank and select in constant time, respectively. Each data structure 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 accompanied with data structures providing constant time access to rank and select, respectively.

Given an integer array of length n, a ***range minimum query*** (RMQ) asks for the index of a minimum value in a given range of the array. There is a lightweight data structure that can be built on top of the array such that it can answer RMQs efficiently:

Lemma 2.1 ([15, Thm 3.7])**.** *Let $A[1..n]$ be an integer array, where accessing an element $A[i]$ takes t_A time ($1 \leq i \leq n$). Given a constant $\delta > 0$, there exists a data structure of size $\delta n + o(n)$ bits built on top of A that answers RMQs in $\mathcal{O}(t_A/\delta)$ time. It is constructed in $\mathcal{O}(t_A n)$ time with $o(n)$ additional bits of working space.*

In the rest of this paper, we take a read-only string T of length n, which is subject to the LZ77 or the LZ78 factorization. Let $T[n] = \texttt{\$}$ be a special character appearing nowhere else in T, so that no suffix of T is a prefix of another suffix of T. We assume that `$` is smaller than all other characters. Further, we assume that accessing a character of T costs $\mathcal{O}(1)$ time (e.g., T is stored in RAM using $n \lg \sigma$ bits).

2.1 Lempel-Ziv Factorization

A ***factorization*** of T of size z partitions T into z substrings $f_1 \cdots f_z = T$. These substrings are called ***factors***. In particular, we have:

Definition 2.2. *A factorization $f_1 \cdots f_z = T$ is called the* **LZ77 factorization** *of T iff $f_x = \operatorname{argmax}_{S \in \mathcal{S}_j(T) \cup \Sigma} |S|$ for all $1 \leq x \leq z$ with $j = |f_1 \cdots f_{x-1}| + 1$.*

This factorization is also called LZSS factorization [58]. An example of the LZ77 factorization of the text `aaababaaabaaba$` is given in Figure 2a.

A variant of the LZ77 factorization is the classic-LZ77 factorization. The difference is that each factor of the classic-LZ77 factorization [60] introduces an additional character at its ends:

Definition 2.3. *A factorization $f_1 \cdots f_z = T$ is called the* **classic-LZ77 factorization** *of T iff f_x is the shortest prefix of $f_x \cdots f_z$ that occurs exactly once in $f_1 \cdots f_x$.*

We refer to Figure 2b for an example of the classic-LZ77 factorization.

Definition 2.4. *A factorization $f_1 \cdots f_z = T$ is called the* **LZ78 factorization** *of T iff $f_x = f'_x c$ with $f'_x = \operatorname{argmax}_{S \in \{f_y : y < x\} \cup \{\lambda\}} |S|$ and $c \in \Sigma$ for all $1 \leq x \leq z$.*

See Figure 2c for an example of the LZ78 factorization.

Although the above defined factorizations do not share the same value of z in general, there is a well-known upper bound of z for all these factorizations:

	1	2	3	4	5	6	7
Factor	a	aa	b	aba	aaba	aba	$
Coding	a	1,2	b	3,3	2,4	3,3	$

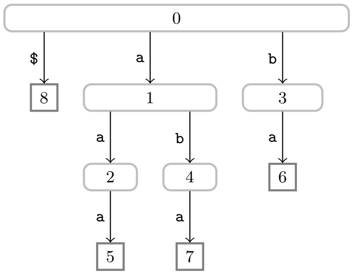
(a) LZ77 Factorization $\overset{1}{\texttt{a}}|\overset{2}{\texttt{aa}}|\overset{3}{\texttt{b}}|\overset{4}{\texttt{aba}}|\overset{5}{\texttt{aaba}}|\overset{6}{\texttt{aba}}|\overset{7}{\texttt{\$}}$

	1	2	3	4	5
Factor	a	aab	abaa	abaab	a$
Coding	a	1,2,b	3,3,a	5,4,b	1,1,$

(b) Classic-LZ77 Factorization $\overset{1}{\texttt{a}}|\overset{2}{\texttt{aab}}|\overset{3}{\texttt{abaa}}|\overset{4}{\texttt{abaab}}|\overset{5}{\texttt{a\$}}$

LZ78	1	2	3	4	5	6	7	8
Factor	a	aa	b	ab	aaa	ba	aba	$
Coding	0,a	1,a	0,b	1,b	2,a	3,a	4,a	0,$

(c) LZ78 Factorization $\overset{1}{\texttt{a}}|\overset{2}{\texttt{aa}}|\overset{3}{\texttt{b}}|\overset{4}{\texttt{ab}}|\overset{5}{\texttt{aaa}}|\overset{6}{\texttt{ba}}|\overset{7}{\texttt{aba}}|\overset{8}{\texttt{\$}}$



(d) LZ Trie

Figure 2: Three kinds of Lempel-Ziv factorizations of the text aaababaaabaaba$. The coding represents a fresh factor by a single character, and a referencing factor by a tuple with two or three entries. For LZ77 (a), this tuple consists of the referred position and the number of characters to copy. For classic-LZ77 (b), the tuple additionally contains the character at the end of the factor. For LZ78 (c), it consists of the referred index and the character at the end of the factor. The LZ trie (d) is another representation of the LZ78 factorization.

Lemma 2.5 ([61])**.** *Given a text of length n on an alphabet of size σ, the LZ77, LZ77-classic, or LZ78 factorization divides the text into z factors with $z \le \mathcal{O}(n/\lg_\sigma n)$. We yield that $\mathcal{O}(z \lg z) = \mathcal{O}(z \lg n) = \mathcal{O}(n \lg \sigma)$.*

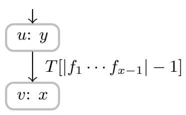
We call a text position j ($1 \le j \le n$) the ***factor position*** of the factor f_x ($1 \le x \le z$) iff f_x starts at position j. A factor f_x may refer to either

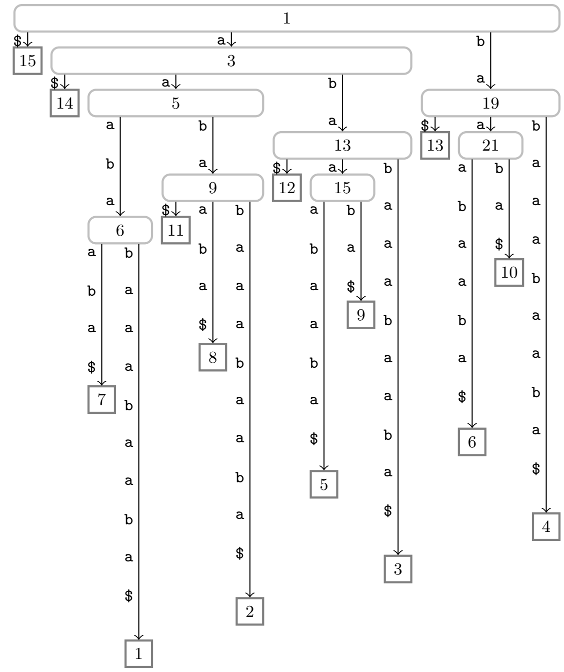
(LZ77) a previous text position j (called f_x's ***referred position***), or

(LZ78) to a previous factor f_y (called f_x's ***referred factor***—in this case y is also called the ***referred index*** of f_x).

If there is no suitable reference found for a given factor f_x with factor position j, then f_x consists of just the single letter $T[j]$. We call such a factor a ***fresh factor***. Fresh LZ78 factors refer to $f_0 := \lambda$. The other factors are called ***referencing factors***; let z_{R} denote the number of referencing factors.

A natural representation of the LZ78 factors is a trie, the so-called ***LZ trie***. Each node in the trie represents a factor and is labeled with its factor index (see Figure 2d). The trie has the following properties: If the x-th factor refers to the y-th factor, then there is a node u having a child v such that u and v have the unique labels y and x, respectively. The edge (u, v) is labeled with the last character of the x-th factor. A node with label x is the child of the root iff the x-th factor is a fresh factor. Hence, each node (except the root) represents a factor.





(a) Suffix Tree

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	a	a	b	a	b	a	a	a	b	a	a	b	a	$
SA	15	14	7	1	11	8	2	12	5	9	3	13	6	10	4
ISA	4	7	11	15	9	13	3	6	10	14	5	8	12	2	1
Ψ	4	1	6	7	8	10	11	12	13	14	15	2	3	5	9

(b) Suffix Array, its inverse and the Ψ-function

Figure 3: The suffix tree (a) of T = aaababaaabaaba\$. The internal nodes are labeled by their preorder numbers, leaves by the text position where their respective suffix starts. The number of letters on an edge e is $c(e)$. Reading the labels of the leaves from left to right gives SA (b).

2.2 Suffix Tree

The ***suffix trie*** of T is the trie of all suffixes of T. The ***suffix tree*** of T is the tree obtained by compacting the suffix trie of T; the suffix tree has n leaves and at most n internal nodes. The string stored in a suffix tree edge e is called the ***label*** of e. We define the function $c(e)$ returning, for each edge e, the length of e's label. The ***string depth*** of a node v is defined as the length of the concatenation of all edge labels on the path from the root to v. The leaf corresponding to the i-th suffix is labeled with i (see Figure 3). We write $\text{label}(\ell)$ for the label of a leaf ℓ. Reading the leaf labels in depth first order gives the suffix array [43]. We denote the suffix array and the inverse suffix array of T by SA and ISA, respectively. The array ISA is defined such that $\mathsf{ISA}[\mathsf{SA}[i]] = i$ for every $i = 1, \ldots, n$. $\mathsf{LCP}[2..n]$ is an array such that $\mathsf{LCP}[i]$ is the length of the longest common prefix of the *lexicographically* i-th smallest suffix with its lexicographic predecessor (i.e., the $(i-1)$-th smallest suffix) for $i = 2, \ldots, n$.

2.2.1 Operations on the Suffix Tree

We need the following navigation methods on the suffix tree (v is a suffix tree node, and ℓ, ℓ' are suffix tree leaves; v, ℓ and ℓ' are represented by their preorder numbers):

- $\text{parent}(v)$ returns the parent of the node v,
- $\text{depth}(v)$ returns the tree depth of the node v,

- level_anc(ℓ, d) returns the ancestor at depth d of the leaf ℓ,
- leaf_select(i) returns the i-th leaf (in lexicographic order) for an integer i with $1 \leq i \leq n$,
- leaf_rank(ℓ) returns the number of preceding leaves (in lexicographic order) of the leaf ℓ,
- lca(ℓ, ℓ') returns the lowest common ancestor of the leaves ℓ and ℓ',
- lmost_leaf(v) and rmost_leaf(v) return the leftmost and rightmost leaf of the node v, respectively,
- child_rank(v) returns the number of preceding siblings of the node v,
- $v.$child(i) returns the i-th child of the node v (we only need $i = 1, 2$), and
- next_sibling(v) returns the next sibling of v.

Note that leaf_rank and label are related, but different (we require access only to the former operation from a suffix tree). The difference is that, given a leaf ℓ representing the i-th suffix, label$(\ell) = i$, whereas leaf_rank$(\ell) = \mathsf{ISA}[i]$, i.e., label$(\ell) = \mathsf{SA}[$leaf_rank$(\ell)]$.

For the algorithms in this article we are interested in the following specific methods:

- head(ℓ) returns the first character of the suffix whose starting position is label(ℓ),
- smallest_leaf returns the leaf with label 1,
- next_leaf(ℓ) returns the leaf labeled with label$(\ell) + 1$, and
- str_depth(v) returns the string depth of an *internal* node.

In the following, we show how to represent the suffix tree and how the above methods can be computed. There are plenty of suffix tree representations to choose from. Although we can build a pointer based suffix tree in linear time [12] the tree itself takes $\mathcal{O}(n \lg n)$ bits. There are variants that can be stored in $nH_k + o(n \lg \sigma)$ bits [54], but they still need $\Omega(n \lg \sigma)$ working space during their computation. Here, we focus on two representations of the suffix tree that are especially trimmed on a small memory footprint during their construction. The first one, called a ***compressed suffix tree***, uses $\mathcal{O}(n \lg \sigma)$ bits of total space that is smaller than the uncompressed version for $\sigma = o(n)$, but does not have any advantages over the non-succinct version when $\sigma = \Omega(n)$. In case of a large alphabet, we choose an alphabet-independent solution, which we call a ***succinct suffix tree***. It uses $(1 + \epsilon)n \lg n$ bits of total space; its memory footprint is independent of the size of the alphabet.

2.2.2 Compressed Suffix Tree

The compressed suffix tree (CST) consists of two components. The first data structure is the Ψ-function [23] defined by $\mathsf{SA}[i] = \mathsf{SA}[\Psi(i)] - 1$ for $1 \leq i \leq n$ with $\mathsf{SA}[i] \neq n$ (and $\Psi(i) = \mathsf{ISA}[1]$ for $\mathsf{SA}[i] = n$). It can be stored in $\mathcal{O}(n \lg \sigma)$ bits while allowing constant access time [25]. The second one is a $4n + o(n)$-bit balanced parenthesis representation [26] of the tree topology [56]. Munro et al. [44] show the following result:

Theorem 2.6 ([44])**.** *We can build the compressed suffix tree consuming $\mathcal{O}(n \lg \sigma)$ bits of space in $\mathcal{O}(n)$ time.*

The balanced parentheses sequence allows us to answer the tree navigation queries of Section 2.2.1 in constant time [49, 27]. We can implement the other methods efficiently as follows:

head(ℓ) Given that Σ is the *effective* alphabet of T, i.e., each character of Σ appears in T at least once, the root of the suffix tree has σ children, each corresponding to a different character. The order of the children of the root and of the characters of Σ is the same. Hence, child_rank(level_anc($\ell, 1$)) = head(ℓ) holds, and the left hand side can be computed in constant time.

If Σ is not effective, then we copy the text T to an array A of size $n \lg \sigma$ bits, and sort its characters with a linear time integer sorting algorithm using $\mathcal{O}(\lg n)$ bits of working space [17]. After sorting A, we remove all adjacent duplicate characters such that $A[i] \in \Sigma$ stores the i-th lexicographically sorted character contained in T, for every integer i from one up to the size of the *effective* alphabet. We compute head(ℓ) with A[child_rank(level_anc($\ell, 1$))] in constant time.

smallest_leaf Since $ is the smallest character in the text T appearing exactly once at $T[n]$, it holds that $\mathsf{SA}[1] = n$; hence $\Psi(1) = \mathsf{ISA}[1] =$ smallest_leaf.

next_leaf(ℓ) We can compute next_leaf(ℓ) = leaf_select(Ψ(leaf_rank(ℓ))) in constant time.

str_depth(v) We use the Ψ-function and the head-function to compute str_depth(v) in time proportional to the string depth. To this end, we take two different children of v (they exist since v is an internal node), and choose an arbitrary leaf in the subtree of each child. By doing so, we have selected two leaves representing two suffixes whose longest common prefix is the string read from the edge labels on the path from the root to the lowest common ancestor of both leaves. Our task is to compute the length of this prefix. To this end, we match the first characters of both suffixes by the head-function. If they match, we use Ψ to move to the next pair of suffixes (i.e., suffixes), and apply the head-function again. Informally, applying Ψ strips the first character of both suffixes (like taking a suffix link, i.e., an edge from a node whose path from the root reads cS to a node whose path from the root reads S, with $c \in \Sigma, S \in \Sigma^*$). We repeat this as long as the first characters of both suffixes match. On a mismatch of the first characters, each character finally represents an edge from v to a different child, i.e., we have found the string depth of v that is equal to the number of matched characters.

2.2.3 Succinct Suffix Tree

The arrays SA and ISA already need $2n \lg n$ bits of space, which is too much for getting our aimed space bounds of $(1 + \epsilon)n \lg n$ bits. We cannot store both SA and ISA as plain arrays. Instead, we employ the following data structure:

Lemma 2.7 ([47, page 106]). *Given a permutation A of n integers, there is a data structure that gives access to A's inverse in $\mathcal{O}(1/\epsilon)$ time. The data structure uses $\epsilon n \lg n + n$ bits, provided that all integers of A can be stored in $\lg n$ bits. It can be constructed with additional n bits in $\mathcal{O}(n)$ time.*

The succinct suffix tree (SST) consists of

- ISA with $n \lg n$ bits,
- SA with $\epsilon n \lg n$ bits (using Lemma 2.7),

- LCP with $2n + o(n)$ bits [55],[2]
- an RMQ data structure on LCP with $2n + o(n)$ bits (using Lemma 2.1 with $t_{\mathsf{SA}} = 1/\epsilon$),
- and a $4n + o(n)$-bit representation of the suffix tree topology in DFUDS [6].

We construct the succinct suffix tree in the following way while making use of several algorithms from the literature:

1. SA can be constructed in $\mathcal{O}(n/\epsilon^2)$ time and $(1+\epsilon)n \lg n$ bits of space, including the space for SA itself [31].[3]
2. We invert SA to ISA with $n + \mathcal{O}(\lg n)$ bits of working space [11].
3. We construct the data structure of Lemma 2.7 to regain access to SA with $\mathcal{O}(1/\epsilon)$ access time.
4. Given SA and ISA, LCP can be computed in $\mathcal{O}(n/\epsilon)$ time with no extra space [59].
5. The RMQ data structure on LCP can be constructed in $\mathcal{O}(n/\epsilon)$ time, and answers queries in $\mathcal{O}(1/\epsilon)$ time (see Lemma 2.1).
6. Given both SA and LCP with $\mathcal{O}(1/\epsilon)$ access time, a space economical construction of the tree topology was discussed in [51, Alg. 1]. The authors showed that the DFUDS representation of the suffix tree topology can be built in $\mathcal{O}(n/\epsilon)$ time with $n + o(n)$ bits of working space.

Putting together all ingredients of the above list yields

Theorem 2.8. *The succinct suffix tree takes $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits of space. We can construct it in-place in $\mathcal{O}(n/\epsilon^2)$ time.*

The navigation methods of the suffix tree described in Section 2.2.1 can be answered in constant time due to the chosen suffix tree topology [27]. We can support the other methods easily with

- $\mathrm{head}(\ell) := T[\mathrm{label}(\ell)]$,
- $\mathrm{str\_depth}(v) := \mathsf{LCP.RMQ}\left[\mathrm{leaf\_rank}(\mathrm{lmost\_leaf}(v) + 1), \mathrm{leaf\_rank}(\mathrm{rmost\_leaf}(v))\right]$,
- $\mathrm{smallest\_leaf} := \mathrm{leaf\_select}(\mathsf{ISA}[1])$, and
- $\mathrm{next\_leaf}(\ell) := \mathrm{leaf\_select}(\mathrm{label}(\ell) + 1)$.

All methods using the label-function take $\mathcal{O}(1/\epsilon)$ time due to the way in which the suffix array is stored. These times differ from the query times for the compressed suffix tree – see Figure 4 for a juxtaposition.

For our algorithms to work space efficiently, it is important that the space taken by the succinct suffix tree is *rewritable*. Let us call X the $n \lg n$ bits of space occupied by ISA, and Y the $\epsilon n \lg n$ bits space taken

[2] More precisely, we use the permuted longest common prefix array that can access LCP only in conjunction with SA.

[3] The time bound for computing the suffix array has recently been improved to $\mathcal{O}(n)$ by two in-place suffix sorting algorithms [41, 20]. Our succinct suffix tree is composed of both SA and ISA, yielding $(1+\epsilon)n \lg n$ bits and $\mathcal{O}(n/\epsilon)$ construction time. This construction time is the bottleneck of the succinct suffix tree construction and the later described algorithms. Hence, we can lower the time $\mathcal{O}(n/\epsilon^2)$ to $\mathcal{O}(n/\epsilon)$ in Theorem 2.8 and Corollaries 3.2, 3.7 and 4.10.

	SST	CST
Construction Time	$\mathcal{O}(n/\epsilon^2)$	$\mathcal{O}(n)$
Space in Bits	$(1+\epsilon)n\lg n+\mathcal{O}(n)$	$\mathcal{O}(n\lg\sigma)$

Operation	SST time	CST time
label(ℓ)	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(n)$
str_depth(v)	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(\text{str\_depth}(v))$
$c(e)$	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(\text{str\_depth}(v))$
head(ℓ)	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(1)$
smallest_leaf	$\mathcal{O}(1)$	$\mathcal{O}(1)$
next_leaf	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(1)$

Figure 4: Left: Construction time and needed space for the succinct suffix tree (SST) and compressed suffix tree (CST) representations. Right: Time bounds for certain operations needed by our LZ factorization algorithms. We circumvent the need for the function label by the operations in the lower part. Having the label of a leaf ℓ, next_leaf(ℓ) can be computed in constant time by the succinct suffix tree representation without access to SA.

by SA. We chose such generic names since the contents of these arrays will change several times during the LZ-computation.

Overwriting X has some undesirable side effects, besides removing the access to ISA: That is because we can access LCP[i] and the RMQ data structure on LCP *only* if SA[i] is also available. Similarly, we can access SA only when ISA is available.

2.3 Framework of the Algorithms and Model of Computation

Common to our LZ77- and LZ78-factorization algorithms is the traversal of the suffix tree during which we mark nodes. To mark nodes efficiently we uniquely identify each node of the suffix tree with its preorder number. We can address a node by its preorder number and vice versa in constant time by adding a rank- and a select-support to the bit vector representing the suffix tree topology (i.e., either in DFUDS or in the balanced parenthesis representation). If the context is clear, we implicitly convert a suffix tree node to its preorder number, and vice versa.

Refining this idea, we distinguish between leaves and internal nodes since we will often mark either leaves or internal nodes. Then we only have to allocate a bit vector of length n for marking either leaves or internal nodes of the suffix tree. To this end, we enumerate the leaves by their leaf_rank-value, and the internal nodes by their preorder numbers when omitting the leaves in the suffix tree. The latter number can be computed in constant time since we can convert the preorder number v of an internal node by $v - \text{leaf\_rank}(\text{lmost\_leaf}(v)) + 1$.

Next, we explain the common framework shared by our algorithms. We will do this by introducing some new keywords:

Witnesses. Witnesses are *internal* nodes that act as signposts for finding (LZ77) the referred position or (LZ78) the referred index of a factor. These nodes are necessary in our approach due to the space restrictions. To compute the referred position or referred index of a factor f, we mark a certain node as the witness of f such that we can determine the referred position or referred index of f by examining its witness at a later stage. The *number of witnesses* z_W is at most the number of *referencing* factors z_R. (Since a node can be the witness of more than one factor, we could have $z_{\mathrm{W}} < z_{\mathrm{R}}$.) We enumerate the witnesses from 1 to z_{W} using a bit vector B_{W} of length n marking internal suffix tree nodes that are witnesses. We add a rank_1-support to B_{W} such that we can map the preorder numbers of the witnesses to the B_{W}-ranks. We call $B_{\mathrm{W}}.\mathrm{rank}_1(w)$ the ***witness rank*** of a witness w.

Passes. We divide our algorithms into several passes. In a pass, we visit the leaves of the suffix tree in text order. This is done by using smallest_leaf and then calling next_leaf successively. The passes differ in how a leaf is processed. While processing a leaf ℓ, we want to access label(ℓ). We can track the label of the current leaf with a counter variable, since we start at the leaf with the label 1.

Corresponding Leaves. We say that a leaf ℓ ***corresponds to*** the factor f if label(ℓ) is the factor position of f. During a pass, we keep track of whether a visited leaf corresponds to a factor. To this end, for each leaf ℓ corresponding to a factor f, we compute the length of f while processing ℓ. This length tells us the number of leaves after ℓ (in *text order*) that do not correspond to a factor. By remembering the next corresponding leaf, we know whether the current leaf corresponds to a factor — remember that a pass selects leaves successively in *text order*, and smallest_leaf always corresponds to the first factor.

Our Setting. The algorithms have to factorize (i.e., determine the factor lengths and the factor positions) and compute the referred positions (LZ77) or the referred indices (LZ78). We consider two scenarios: In the **first scenario**, the output has to be stored *explicitly* in RAM.

For LZ77, we store the referred positions in an array with $z \lg n$ bits such that the x-th entry stores the referred position of the x-th factor. The factor lengths can be represented by an additional array with $z \lg n$ bits or a bit vector of length n marking the factor positions. In the latter case, we enhance the bit vector with a select-support such that we get the length of the x-th factor with $\mathrm{select}_1(x+1) - \mathrm{select}_1(x)$ (or $n - \mathrm{select}_1(z)$ if $x = z$).

For LZ78, we store the factor indices in an array with $z \lg z$ bits. The characters at the endings of the factors can be stored either explicitly in a $z \lg \sigma$ bits array or implicitly by a bit vector B_{T} with a select-support marking the factor positions. The bit vector B_{T} takes $n + o(n)$ bits and can look up the character at the end of the x-th factor ($1 \le x \le z$) in $T[B_{\mathrm{T}}.\mathrm{select}_1(x+1) - 1]$.

In the **second scenario (output-streaming)**, we want the output to be streamed sequentially. An output-streaming algorithm has to output a factor as a pair of values, i.e., (LZ77) its referred position and length, or (LZ78) its referred index and the character at its end; we call the set of these value-pairs the ***coding*** of the respective factorization (see Figure 2). The algorithm has to output the coding sequentially *in text order*. We do not count the output in the final working space.

We can alter an output-streaming algorithm to store the factorization explicitly by adding (LZ77) $z \lg n + \min(n + o(n), z \lg n)$ bits or (LZ78) $z \lg z + \min(n + o(n), z \lg \sigma)$ bits of additional working space.

3 LZ77

LZ77 Passes. Common to all passes is the following procedure: For each visited leaf ℓ, we perform a leaf-to-root traversal, i.e., we visit every node on the path from ℓ to the root. But we stop the leaf-to-root traversal on visiting a node visited in an earlier traversal. The idea is the following: Before starting the leaf-to-root traversal of the leaf with label j ($1 \le j \le n$), each string in $\mathcal{S}_j(T)$ can be obtained by reading the edge labels on the paths from the root to every already visited node. Given that the leaf with label j

corresponds to a factor f, the factor can be determined by accessing an already visited node in a leaf-to-root traversal from the leaf with label j. We only have to access the *lowest* already visited node, since the LZ77 factorization chooses the *longest* preceding substring matching f. For this purpose we create a bit vector B_{V} of length n with which we mark the visited internal nodes (we represent internal nodes by a number from 1 up to n as described in Section 2.3). This bit vector is cleared before a pass starts. Since the suffix tree of T contains at most $n-1$ internal nodes, a pass can be conducted in linear time.

LZ77 Witnesses. Determining the witnesses is done in the first pass in the following way: Reaching the root from a leaf corresponding to a factor (while visiting only nodes that are not marked in B_{V}) means that we found a fresh factor. Otherwise, assume that we visit an already visited node u from a leaf ℓ, and that u is not the root. If ℓ corresponds to a factor f, u *witnesses* the referred position of f. This means that there is a suffix starting before text position label(ℓ) having a prefix equal to the string read from the edge labels on the path from the root to u. Moreover, u is the lowest node in the set $\{\mathrm{lca}(\ell, \ell') : \mathrm{label}(\ell') < \mathrm{label}(\ell)\}$ comprising the lowest common ancestors of ℓ with all already visited leaves ℓ'; the factor corresponding to ℓ has to refer to a text position coinciding with the label of a leaf belonging to u's subtree. Additionally, we can determine the factor position of the next factor by computing the length of f with str_depth(u).

We computed the witnesses for our running example in Figure 5. For LZ77 (left side), the witness nodes have preorder numbers 5, 10, and 14, and the leaves corresponding to factors have labels $1, 2, 4, 5, 8, 12$, and 15. For instance, the leaf corresponding to the 4-th factor has label 5. Its witness has preorder number 13. Among all descendant leaves of this witness our algorithms choose the leaf with the lowest label. In this example the referred position of the 4-th factor is the text position 3 (that is equal to the label of the chosen leaf). The length of the 4-th factor is the string depth of its witness. We have $z_{\mathrm{W}} = 3$ and $z_{\mathrm{R}} = 4$.

3.1 Output-Streaming with the Succinct Suffix Tree

We build an RMQ data structure on SA to be able to find the leaf with the smallest label in the subtree rooted at a given witness in $\mathcal{O}(1/\epsilon)$ time. This data structure allows us to output the factorization with a single pass in linear time: On visiting an already visited node v during a leaf-to-root traversal from a corresponding leaf ℓ, we find the leaf ℓ' with the smallest label having v as its ancestor in $\mathcal{O}(1/\epsilon)$ time by an RMQ on SA. The label of ℓ' is the referred position of the factor corresponding to the leaf ℓ, and str_depth(v) is its factor length. Altogether we process n leaves. The access to SA is the only operation that costs us $\mathcal{O}(1/\epsilon)$ time; the other operations can be executed in constant time. In total, the algorithm runs in $\mathcal{O}(n/\epsilon)$ time. We get

Theorem 3.1. *Given the succinct suffix tree of T and a constant $\delta > 0$, we can compute the LZ77 factorization in $\mathcal{O}(z/(\delta\epsilon) + n/\delta) = \mathcal{O}(n/(\delta\epsilon))$ time using $(1+\delta)n + o(n)$ bits of working space when output-streaming.*

Proof. We build the RMQ data structure on SA before inverting it to ISA during the construction of the succinct suffix tree. By Lemma 2.1, the construction of the RMQ data structure takes $\mathcal{O}(n)$ time and uses $o(n)$ additional bits. The RMQ data structure uses δn bits and answers RMQs in $\mathcal{O}(1/\delta)$ time. After storing SA in Y, the access time to the RMQ data structure worsens to $\mathcal{O}(1/(\delta\epsilon))$.

Finally, we add B_{V} using n bits to our final working space. Since we do everything in a single pass, we do not need to maintain B_{W}. □

Corollary 3.2. *We can stream the LZ77 factorization of a text of length n in $\mathcal{O}(n/\epsilon^2)$ time using $(1+\epsilon)n\lg n + \mathcal{O}(n)$ bits of space.*

Proof. We use the succinct suffix tree to compute the LZ77 factorization. Its space requirement given in Theorem 2.8 dominates the space needed for the factorization algorithm given in Theorem 3.1. □

This is already an improvement over the previously best linear-time algorithm using $2n\lg n$ bits [32] for general integer alphabets.

Unfortunately, combining this algorithm with the compressed suffix tree inherently causes the need to simulate SA. Simulating SA with the compressed suffix tree still takes $\omega(1)$ time per SA access (see [5] for some known bounds). Hence, we cannot hope for a linear time algorithm with an approach that uses $\mathcal{O}(n\lg\sigma)$ bits and RMQs on the suffix array (which is not stored explicitly). Luckily, with a tiny trick, we can avoid this problem by making two passes as shown in the next section.

3.2 Alphabet-Sensitive LZ77 Factorization

We study only the output-streaming variant, for which we claim the following result:

Theorem 3.3. *Given the compressed suffix tree of T, we can compute the LZ77 factorization in $\mathcal{O}(n)$ time using $2n + z\lg n + o(n)$ additional bits of working space when output-streaming. The factorization could also be stored in RAM with additional $z\lg n$ bits.*

In order to obtain the results of Theorem 3.3, we perform two passes:

(a) create B_{W} in order to determine the witnesses, and

(b) stream the output by using an array mapping witness ranks to text positions.

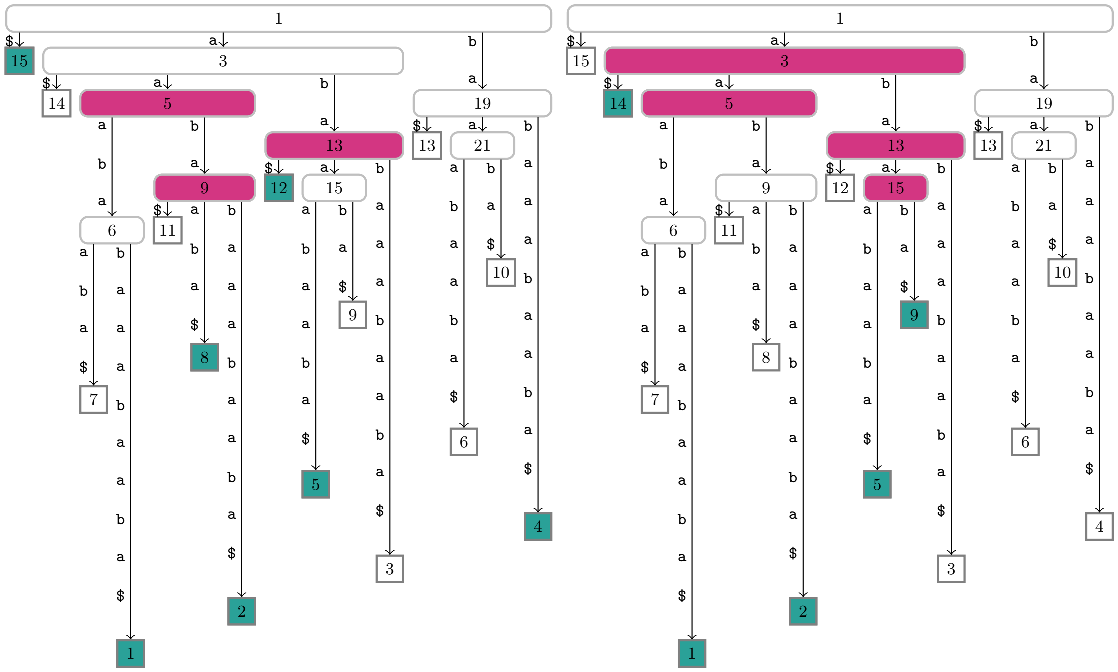
Pass (a). We find the witnesses with the leaf-to-top traversals as described in the beginning of this section. On finding a witness we mark it in B_{W} in order to determine the referred position in the next pass. After this pass, we have determined the z_{W} witnesses by the '1's stored in B_{W}. We use the witnesses in the next pass to compute the referred positions (see Figure 5).

Pass (b). We clear B_{V}, create a rank-support on B_{W} and allocate an array W consuming $z_{\mathrm{W}}\lg n$ bits. We use W to map a witness rank to a text position (a referred position in particular, see Figure 6). We set $W[w]$ to the label of the leaf from which we visited the witness w for the first time. By doing so, we find the referred position of a referencing factor f in $W[w]$ when visiting w again from the leaf corresponding to f. The length of f is the string depth of w. Since fresh factors consist of single characters, we can output a fresh factor by applying the head-function to its corresponding leaf.

The following corollary sums up our obtained result:

Corollary 3.4. *We can compute the LZ77 factorization of a text of length n in $\mathcal{O}(n)$ time using $\mathcal{O}(n\lg\sigma)$ bits of space.*

Proof. We use the compressed suffix tree to compute the LZ77 factorization. Its space requirement given in Theorem 2.6 dominates the space needed for the factorization algorithm given in Theorem 3.3. □



Preorder	1	3	5	6	9	13	15	19	21
B_{W}	0	0	1	0	1	1	0	0	0

(a) LZ77 Factorization

Preorder	1	3	5	6	9	13	15	19	21
B_{W}	0	1	1	0	0	1	1	0	0

(b) Classic-LZ77 Factorization

Figure 5: Suffix tree with the witness nodes and the corresponding leaves highlighted. The witness nodes and the corresponding leaves are determined during Pass (a) in Section 3.2 and Section 3.3. The witnesses are colored in magenta, the leaves corresponding to factors are colored in dark cyan.

Storing the Output. Instead of streaming the output we allocate an additional $z \lg n$ bits array to store the factor indices. Next, we clear the bit vector B_{W} to use it for storing the factor positions. We can compute the new contents of B_{W} in an additional pass.

Classic-LZ77 factorization. The LZ77 and the classic-LZ77 factorizations differ in the fact that the latter always introduces a new character at the end of each factor. We can easily adapt our algorithm to the classic-LZ77 factorization. To this end, we prolong each factor during the first pass where we determine the witnesses and factor lengths: On processing a corresponding leaf ℓ during this pass, we set the length of the factor f corresponding to ℓ to the string depth of ℓ's witness *incremented by one*. We can get the new character of f with factor index x when accessing the leaf whose label is one text position smaller than the label of the leaf corresponding to the $(x+1)$-th factor (we then apply head on that leaf to obtain the new character).

We conducted the classic LZ77 factorization on our running example in Figure 5. There, the witness nodes have the preorder numbers $3, 5, 13,$ and 15, and the leaves corresponding to factors have the labels $1, 2, 5, 9,$ and 14. We have $z_{\mathrm{W}} = z_{\mathrm{R}} = 4$.

witness rank	1	2	3
W	1	2	3

(a) LZ77

witness rank	1	2	3	4
W	1	1	3	5

(b) Classic-LZ77

Figure 6: The array W storing a referred position for each witness rank in Section 3.2.

	Initial	(a)	(b)	(c)	(M)
X	ISA	ISA	ISA	D	Referred Positions
Y	SA	SA	-	-	Helper Array

Figure 7: Chronological table of the contents of the arrays X and Y modified in Section 3.3. The algorithms working on the succinct suffix tree overwrite the contents of the arrays X and Y, initially storing ISA and SA. The columns represent the different phases that are chronologically sorted. An entry of one of both tables gives the content stored in X or Y at a certain phase.

3.3 Computing the LZ77 Factorization In-Place Alphabet-Independently

Allowing only $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits for the entire working space plus output, it is no longer possible to store both SA and ISA at the same time.

Our idea is to gradually compute the necessary information about the factors (the factor position and the lengths) such that we can overwrite X and Y when we no longer need SA and/or ISA (see Figure 7 for an overview). As a final result, we will store in $X[x]$ the referred position of the x-th factor. To this end, we divide our algorithm into three passes and a final matching phase, all of which will be discussed in detail in the following:

(a) Construct a bit vector $B_{\mathrm{T}}[1..n]$ marking all factor positions in T. Determine the witnesses, and mark them in B_{W}.

(b) Construct a bit vector B_{D} counting (in unary) the number of witnesses visited during each leaf-to-root traversal.

(c) Construct an array D storing the witness ranks of all witnesses visited during each leaf-to-root traversal (as counted in the second round).

(M) Convert the witness ranks in D to the referred positions.

Pass (a). This pass works exactly as Pass (a) in Section 3.2. After Pass (a), SA is no longer needed.

Before starting with the next pass, let us assume *conceptually* that each leaf ℓ maintains a list $\mathcal{L}_\ell$ storing the visited witnesses during the leaf-to-root traversal from the leaf ℓ in chronological order. If the last visited node w during the leaf-to-root traversal from the leaf ℓ has been visited during a former traversal, we add w to the list $\mathcal{L}_\ell$ *only if* the leaf ℓ corresponds to a factor f; in this case the node w is the witness of the factor f. We call the last entry in $\mathcal{L}_\ell$ of a corresponding leaf ℓ the ***referred entry*** of ℓ. Given that the list $\mathcal{L}_\ell$ of a leaf ℓ stores the witness w in its referred entry, there is exactly one other leaf ℓ' maintaining a list $\mathcal{L}_{\ell'}$ that contains w not as a referred entry, and it holds that label(ℓ') < label(ℓ). The leaf ℓ' has also the smallest label among all leaves that contain w in their lists. This means that w is first found on the

	111 111
text positions	1 2 3 45 678 9012 345
B_{D}	1010010110111011110111

(a) Bit Vector B_{D}

text position	1	2		3	5	8	12
D	1	2	1	3	3	2	3

(b) Array D

Figure 8: The list of lists $\mathcal{L}$ represented by B_{D} (a) and D (b), computed on our running example. The number of zeros between two ones ($B_{\mathrm{D}}.\mathrm{select}_1(i)$ and $B_{\mathrm{D}}.\mathrm{select}_1(i+1)$ for a text position i with $1 \leq i \leq n$) in B_{D} equals the number of entries in D for the text position i. We shaded the referred entries in D. Non-shaded numbers in D are unique, and each non-shaded number appears later in a referred entry at least once. The LZ77-witnesses are depicted in the left side of Figure 5.

leaf-to-root traversal starting from ℓ', and is later determined as the witness of ℓ. Consequently, the factor corresponding to ℓ refers to the text position coinciding with the label of ℓ'. To sum up, finding the leaf whose list, excluding its referred entry, contains the referred entry of a corresponding leaf ℓ is the crucial step in finding the referred position of the referencing factor corresponding to ℓ.

In our running example, the lists have the contents $\mathcal{L}_1 = (1)$, $\mathcal{L}_2 = (2, 1)$, $\mathcal{L}_3 = (3)$, $\mathcal{L}_4 = ()$, $\mathcal{L}_5 = (3)$, $\mathcal{L}_6 = ()$, and so on (we wrote $\mathcal{L}_{\mathrm{label}(\ell)}$ instead of $\mathcal{L}_\ell$). Due to Pass (a) we know that the leaf with label 2 is a corresponding leaf. The list $\mathcal{L}_2$ stores at its referred entry the witness with witness rank 1. This witness is found during the leaf-to-root traversal from the leaf with label 1, since this leaf is the first whose list contains this witness rank.

In the following, we aim at representing the list of lists $\mathcal{L}$, sorted by the labels of the leaves (such that $\mathcal{L}[1]$ start with the list of the leaf with label 1). Despite the fact that $\mathcal{L}$ contains $z_{\mathrm{W}} + z_{\mathrm{R}}$ witness ranks in total, we want that it takes at most $n \lg n$ bits of space so that we can store it in X. This is possible due to

Lemma 3.5. *The number of witnesses z_{W} plus the number of referencing factors z_{R} is at most n.*

Proof. Let z_{R}^1 (resp. z_{R}^+) denote the number of referencing factors of length 1 (resp. longer than 1), and let z_{W}^1 (resp. z_{W}^+) denote the number of witnesses whose string depth is 1 (resp. longer than 1). Also, z_{F} denotes the number of fresh factors. Clearly, $z_{\mathrm{W}} = z_{\mathrm{W}}^1 + z_{\mathrm{W}}^+$, $z_{\mathrm{R}} = z_{\mathrm{R}}^1 + z_{\mathrm{R}}^+$, $z_{\mathrm{R}}^1 \leq z_{\mathrm{F}}$, and $z_{\mathrm{W}}^+ \leq z_{\mathrm{R}}^+$. Hence $z_{\mathrm{R}} + z_{\mathrm{W}} = z_{\mathrm{W}}^1 + z_{\mathrm{W}}^+ + z_{\mathrm{R}}^1 + z_{\mathrm{R}}^+ \leq z_{\mathrm{F}} + z_{\mathrm{R}}^1 + 2z_{\mathrm{R}}^+ \leq n$. The last inequality follows from the fact that the factors are counted disjointly by z_{F}, z_{R}^1 and z_{R}^+, and the sum over the lengths of all factors is bounded by n, and every factor counted by z_{R}^+ has length at least 2. □

The data structure we use to store $\mathcal{L}$ consists of an integer array D storing the contents of $\mathcal{L}$, and a bit vector B_{D} partitioning D into the n lists of $\mathcal{L}$. The array D can be built sequentially by appending the witness ranks whenever they are marked or referred to during the leaf-to-root traversals. The bit vector B_{D} stores a '1' for each text position $1 \leq j \leq n$, and intersperses these '1's with '0's counting the number of witnesses written to D during the j-th traversal. In total, the bit vector B_{D} contains n ones and $z_{\mathrm{W}} + z_{\mathrm{R}}$ zeros. The '1's in B_{D} implicitly divide D into n partitions (the j-th partition corresponds to the list of the leaf with label j). The size of the j-th partition ($1 \leq j \leq n$) is determined by the number of witnesses accessed during the j-th traversal. Hence the number of '0's between the $(j-1)$-th and j-th '1' represents the number of entries in $\mathcal{L}_\ell$ for the leaf ℓ with label j. Conceptually, we can access the list $\mathcal{L}_\ell$ by $D[B_{\mathrm{D}}.\mathrm{rank}_0(B_{\mathrm{D}}.\mathrm{select}_1(j-1)) + 1..B_{\mathrm{D}}.\mathrm{rank}_0(B_{\mathrm{D}}.\mathrm{select}_1(j))]$, where $j = \mathrm{label}(\ell)$. Since we will perform only sequential scans over B_{D}, there is no need for a rank_0 or a select_1 support on B_{D}. We depicted the bit vector B_{D} and the array D of our running example in Figure 8. Note that D only stores witness ranks. Nevertheless, it uses $\lg n$ bits (instead of $\lg z_{\mathrm{W}}$ bits) per entry because we overwrite the referred entries with the respective referred positions in the end.

Finally, we show the actual computation of D and B_{D}. We want to store D in X in order to get the claimed space bounds. Unfortunately, up to now, we store ISA in X, which is necessary for traversing all leaves in text order, so that overwriting X naively with D would result in losing this functionality. This problem can be solved by performing *two* more passes, as already outlined above.

Pass (b). With the aid of B_{W}, B_{D} is generated by counting the lengths of all lists in $\mathcal{L}$. Next, according to B_{D}, we sparsify ISA by discarding values related to suffixes that will not contribute to the construction of D, i.e., those values i for which there is no '0' between the $(i-1)$-th and the i-th '1' in B_{D}. We align the resulting sparse ISA to the right of X.

Pass (c). We overwrite X with D from left to right using the sparse ISA. Since each suffix having an entry in the sparse ISA has at least one entry in D, overwriting the remaining ISA values before using them cannot happen.

Matching. Once we have D in X, we start matching referencing factors with their referred positions. Recall that each referencing factor has one referred entry, and its referred position is obtained by matching the leftmost occurrence of its witness in D.

Let us first consider the easy case with $z_{\mathrm{W}} \leq \lfloor n\epsilon \rfloor$ such that all referred positions fit into Y (the helper array of size $\epsilon n \lg n$ bits). For each m with $1 \leq m \leq z_{\mathrm{W}}$, the zero-initialized $Y[m]$ will be used to store the label of the smallest leaf among all leaves ℓ having the m-th witness in their list $\mathcal{L}_{\ell}$.

Let us consider that we have set $Y[m] = k$, i.e., the m-th witness was discovered for the first time by the traversal from the leaf labeled with k. Whenever we read the referred entry $D[i]$ of a factor f with factor position larger than k and $B_{\mathrm{W}}.\operatorname{rank}_1(D[i]) = m$, we know by $Y[m] = k$ that the referred position of f is k. Both the filling of Y and the matching are done in one single, sequential scan over D (stored in X) from left to right: While tracking the label of the currently processed leaf with a counter k $(1 \leq k \leq n)$, we look at $t := B_{\mathrm{W}}.\operatorname{rank}_1(D[i])$ and $Y[t]$ for each array position i $(1 \leq i \leq |D|)$: if $Y[t] = 0$, we set $Y[t]$ to k. Otherwise, $D[i]$ is the referred entry of the factor f with factor position k (stored in $Y[t]$). We set $X[i]$ equal to $Y[t]$. By doing this, we overwrite the referred entry of every referencing factor f in D with the referred position of f.

If $z_{\mathrm{W}} > \lfloor n\epsilon \rfloor$, we run the same scan multiple times, i.e., we partition $\{1, \ldots, z_{\mathrm{W}}\}$ into $\lceil z_{\mathrm{W}}/(n\epsilon) \rceil$ equidistant intervals (pad the size of the last one) of size $\lfloor n\epsilon \rfloor$, and perform $\lceil z_{\mathrm{W}}/(n\epsilon) \rceil$ scans. Since each scan takes $\mathcal{O}(n)$ time, the whole computation takes $\mathcal{O}(z_{\mathrm{W}}/\epsilon) = \mathcal{O}(z/\epsilon)$ time. One problem remains: Since the domain of the witness ranks and referred positions can collide (both are integer values), we have to track which referred entries in D are already converted to referred positions. To this end, we use a bit vector marking all processed referred entries such that we can skip the already processed referred entries.

Finally we have the complete information of the factorization: The length of the factors can be obtained by a select-query on B_{T}, and X contains the referred positions of all referencing factors. By a left shift we can restructure X such that $X[x]$ gives us the referred position (if it exists) for each factor $1 \leq x \leq z$. Hence, looking up a factor can be done in $\mathcal{O}(1)$ time.

The following theorem sums up the results of this section:

Theorem 3.6. *Allowing the succinct suffix tree of T to be* rewritable*, we can overwrite it with the LZ77* *factorization in $\mathcal{O}(n)$ time using $4n + 2z_{\mathrm{W}} + 2z_{\mathrm{R}} + o(n) \leq 6n + o(n)$ bits of working space on top of the space used by the suffix tree.*

Proof. Besides B_{V} and B_{W} (defined in Section 3, using $2n + o(n)$ bits together) we need to allocate extra space for the bit vectors B_{D} ($n + z_{\mathrm{W}} + z_{\mathrm{R}}$ bits), B_{T} (n bits) and a bit vector ($z_{\mathrm{W}} + z_{\mathrm{R}}$ bits) used for marking already processed values in D in the matching phase. □

Corollary 3.7. *We can compute the LZ77 factorization of a text of length n in $\mathcal{O}\big(n/\epsilon^2\big)$ time using $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits of space. The factors are stored in-place.*

Proof. We create the succinct suffix tree of Theorem 2.8 to compute the LZ77 factorization. Subsequently, we overwrite the $(1 + \epsilon)n \lg n$ bits of space used by the succinct suffix tree (for representing the suffix array and its inverse) in order to compute the LZ77 factorization in-place according to Theorem 3.6. □

Classic LZ77 factorization. During the leaf-to-root traversals in Section 3.2, we have to account for the fact that the length of each referencing factor has to be enlarged (due to the fresh character). It suffices to mark the factors in B_{T} appropriately to the possibly modified lengths (B_{T} is used to retrieve the position and the length of a factor); the new shape of B_{T} induces implicitly a modification of B_{D}. By doing so, the fresh character that ends a referencing factor will never be considered to be the beginning of a factor. Finally, the fresh character of each referencing factor can be looked up with B_{T} and T. Lemma 3.5 still holds for this variant of the factorization; in fact, since $z^1_{\mathrm{R}} = z^1_{\mathrm{W}} = 0$, the proof gets easier.

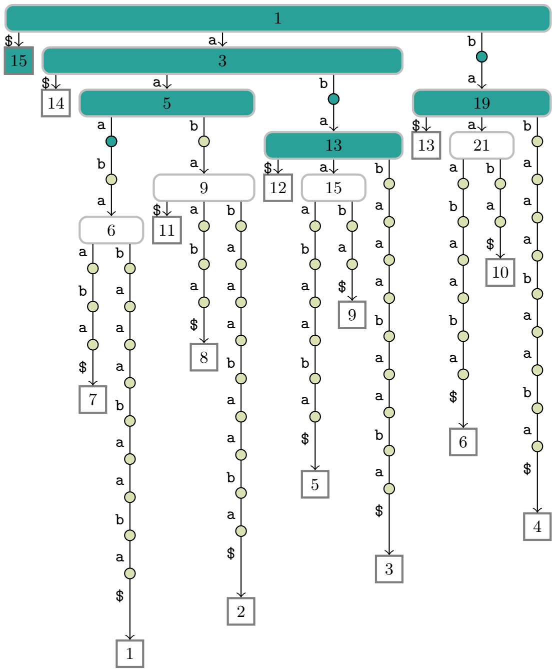
4 LZ78

A natural way to compute the LZ78 factors is to build the LZ trie (see Section 2.1). We can maintain the LZ trie by a dynamic trie implementation. Recall that all dynamic trie implementations have a (log-)logarithmic dependence on σ for top-down-traversals (see Fig. 1 in the introduction). One of our tricks is using level ancestor queries starting from the suffix tree leaves in order to get rid of this dependence. To this end, we superimpose the LZ trie on the suffix trie, which is explained in the next section.

4.1 Storing the LZ Trie Topology

The main idea is the superimposition of the suffix trie on the suffix tree, borrowed from Nakashima et al. [45]: In this context, we think about the LZ trie as a connected subgraph of the suffix trie containing its root (see Figure 9). In the suffix tree, the LZ nodes are either already represented by a suffix tree node (explicit), or lie on a suffix tree edge (implicit). To ease the explanation, we associate each suffix tree edge $e = (u, v)$ from a node u to its child v uniquely with v (each suffix tree node except the root is associated with an edge). In order to address all LZ nodes, we keep track of how far an edge on the suffix tree got explored so far. To this end, for an edge $e = (u, v)$, we define the ***exploration counter*** $0 \leq n_v \leq c(e)$ storing how far e is explored. Adding a factor to the LZ trie results in incrementing one exploration counter. If $n_v = 0$, then the factorization has not (yet) explored e, whereas $n_v = c(e)$ tells us that we have already reached v.

For the LZ78 factorization we mark again certain nodes as witnesses. Here, a witness w will be used for storing the referred indices of factors whose corresponding LZ nodes are on the incoming edge of w. In order to save space, we are interested in a certain type of suffix tree nodes: A ***witness*** is a suffix tree node whose



SA	15	14	7	1	11	8	2	12	5	9	3	13	6	10	4
B_{C}	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Preorder	1	3	5	6	9	13	15	19	21
B_{V}	0	1	1	0	0	1	0	1	0

Figure 9: The suffix tree of `aaababaaabaaba$` superimposed by the suffix trie and the LZ trie. The suffix trie is obtained (conceptually) by exchanging every suffix tree edge e with $c(e) - 1$ new suffix trie nodes superimposing e. These new suffix trie nodes are represented by the small rounded nodes. They represent the implicit suffix trie nodes, while the remaining suffix tree nodes represent the explicit suffix trie nodes. Dark (cyan) colored nodes represent the nodes of the LZ trie. The right side shows B_{C} and B_{V} as explained at the end of Section 4.1. The entries in B_{C} depend on how the binary values are represented (here the least-significant bit is the rightmost bit). The horizontal bars groups the entries of B_{C} into intervals; each interval stores the exploration counter of its respective partially explored node. The edge witness with preorder number 6 is not a witness, since neither its exploration counter was incremented twice nor were any exploration counters of its children incremented.

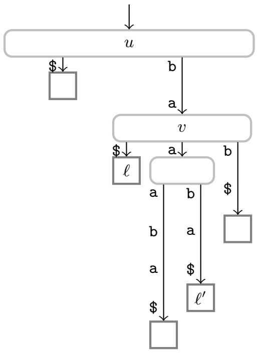
exploration counter gets larger than one while building the LZ trie, or is a node (whose incoming edge e can have a length $c(e) = 1$) having a child whose exploration counter got incremented during the parsing.

LZ78 Passes. An LZ78 pass builds the LZ trie topology implicitly by incrementing the exploration counters and marking (in a bit vector B_{V}) which edges were fully explored. As in Section 2.3, a pass processes all leaves of the suffix tree in text order. For the LZ78 factorization, we only care about the corresponding leaves (see Section 2.3). Starting with the leaf labeled '1', which corresponds to the first factor, we compute the length of the factor corresponding to the currently processed leaf so that we know the distance (in text positions) to the next corresponding leaf. Given a corresponding leaf ℓ, we want to find the first edge (u, v) from a node u to its child v on the path from the root to ℓ that is not yet fully explored. The node u is the lowest node on the path that has been marked in B_{V}. To find u, we traverse the tree downwards along the path from the root to ℓ by level ancestor queries. After having found u at depth d, we know that its child $v = \mathrm{level\_anc}(\ell, d + 1)$ is not yet fully explored, i.e., $n_v < c(u, v)$. We add a new factor by incrementing n_v by one. If the edge e is fully explored (checking this condition is the topic of the next paragraph), we additionally mark v in B_{V}.

Whether the edge e was fully explored or not can be checked as follows: If we use the succinct suffix tree, then we have access to $c(e)$, giving us the maximum value of an exploration counter. Otherwise (when using the compressed suffix tree), we first check if v is a leaf, because then the edge (u, v) can be explored at most once due to

Lemma 4.1. *Let $e = (u, v)$ be a suffix tree edge. Then $n_v \leq \min(c(e), s)$, where s is the number of leaves of the subtree rooted at v.*

Proof. Let S be the string of the edge labels on the path from the root to v. Then S occurs exactly s times in T. The exploration counter n_v can only be incremented for a factor having S as a prefix. Therefore, $n_v \leq s$. □



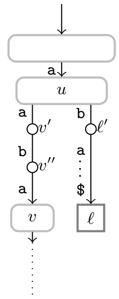
Otherwise (if v is an internal node), we check the condition $c(u, v) = n_v$ similarly to the computation of str_depth in Section 2.2.2 as follows: We choose a leaf ℓ' such that the lowest common ancestor of ℓ and ℓ' is v. The idea is that str_depth(v) is the length of the longest common prefix of two suffixes corresponding to two leaves having v as their lowest common ancestor (e.g., ℓ and ℓ'). We can compare the m-th character of both suffixes by applying next_leaf m-times on both leaves before using the head-function. With $m :=$ str_depth(u) $+ n_v + 1$ we can check whether the edge (u, v) is fully explored. This process also determines the length of the current corresponding factor, which is m. Although we apply next_leaf as many times as the factor length, we still get linear time overall, because the lengths of all factors add up to n.

Overall, an LZ78 pass is conducted in $\mathcal{O}(n)$ time, since we query for a level ancestor at most n times. That is because the depth of a suffix tree node is at most its string depth. The number of level ancestor queries is bounded by the sum of the lengths of all factors, which is n.

The suffix tree nodes that we explore during an LZ78 pass are the important nodes for the factorization. The suffix tree node whose exploration counter gets incremented during a traversal from the root to a leaf ℓ is called the ***edge witness*** of ℓ. A witness is an edge witness, but not necessarily vice versa. Also, while witnesses are always internal nodes, edge witnesses can also be leaves (cf. Figure 10).

Bookkeeping the Exploration Counters. Unfortunately, storing the exploration counters in an integer array for all edges costs $2n \lg n$ bits. Our idea is to choose different representations of the exploration counters depending on the state (*not*, *fully*, or *partially explored*). First, there is no need to represent n_v for a node v with parent u until u's incoming edge is fully explored. Second, as the fully explored edges are marked in a bit vector B_{V}, we do not need to store their exploration counters. The third and last type of nodes are those nodes whose parents are fully explored but they are not. In the following, we call them partially explored nodes, and show how to maintain their exploration counters in n bits.

The subtrees of all partially explored nodes are disjoint, i.e., each leaf has at most one partially explored ancestor. We associate each partially explored node v with an interval $I_v =$ [leaf_rank(lmost_leaf(v)), leaf_rank(rmost_leaf(v))]. These intervals are pairwise disjoint, and the sum of their lengths is at most n. The idea is now to partition a bit vector B_{C} of length n into these intervals such that we store in each interval the exploration counter of a partially explored node, i.e., we assign the exploration counter of v to



	u	v'	ℓ'	v''	v
u	e	w	w	w	w
v		e	e	w	w
ℓ			e	e	e

Figure 10: Comparison of witnesses and edge witnesses. Assume that the LZ78 factorization explores the LZ trie nodes u, v', ℓ', v'', and v belonging to the subtree (left) of the suffix trie in this order (we implicitly map suffix tree nodes to LZ trie nodes). The right table classifies chronologically the nodes u, v, and ℓ in witnesses ('w'), edge witnesses ('e'), or plain suffix tree nodes (empty). In each column we create a new LZ78 factor by exploring an LZ trie node (top row).

the space of $B_{\mathrm{C}}[I_v]$. This space is sufficient since $n_v \leq |I_v|$ due to Lemma 4.1. By storing n_v in binary using the first $\lg |I_v|$ bits of $B_{\mathrm{C}}[I_v]$, we can lookup and increment n_v in constant time. After fully exploring the edge of v ($n_v = c(u,v)$, where u is v's parent), we clear (the first $\lg |I_v|$ bits of) $B_{\mathrm{C}}[I_v]$. As a side effect, this approach resets the counter n_u of every child u of v (the children of v become partially explored on having v fully explored).

Applying this procedure during a pass, we can determine the fully explored edges and maintain n_v of each partially explored node v. The right side of Figure 9 shows B_{C} after building the LZ trie on our running example, where we stored the exploration counter of the node with preorder number 6 at the positions 3 and 4 in B_{C}.

4.2 Alphabet-Sensitive LZ78 Factorization

We present two LZ78 factorization variants. The first variant is an output-streaming algorithm outputting the coding in text order. For our running example, the output is the list of pairs (`0,a`), (`1,a`), (`0,b`), (`1,b`), (`2,a`), (`3,a`), (`4,a`), (`0,$`), see also Figure 2c.

The other variant builds the LZ trie explicitly in such a way that we can return a factor (its referred index and the character at its end), and perform navigations in the LZ trie in constant time. As an application, we will show that our representation can be enhanced with a data structure (see Lemma 4.6) to allow lookups of small substrings of T (which we consider when T is unavailable) in constant time.

4.2.1 Output-Streaming Variant

Equipped with the compressed suffix tree of T we do two passes:

(a) create B_{W} to mark the witnesses, and

(b) stream the output by using a helper array mapping witness ranks to factor indices.

Pass (a). The goal of this pass is to determine the witnesses. To this end, we alter the LZ78 pass described in Section 4.1 in the following way on accessing an edge witness v: If v is an internal suffix tree node whose exploration counter was already incremented, we make v a witness (if it is not yet a witness) by marking v in B_{W}. (The idea is that v's incoming edge is *superimposed* by some LZ nodes whose witness is v.) If the parent u of v is not the root, we make u a witness if it has not yet been one (this can only happen if $c(\mathrm{parent}(u), u) = 1$).

Pass (b). In this pass we compute the referred indices and the characters at the factor endings in text order. We first focus on the referred indices. We compute them indirectly by assigning each factor to a witness found during the former pass. To this end, we create an array W with $z_{\mathrm{W}} \lg z$ bits to store a factor index for each witness rank. We fill W in such a way that the referred index of the currently processed factor can be looked up in the entry in W belonging to its witness. Initially all entries of W are set to $\perp$ (a fixed chosen invalid value).

Before conducting the pass, we reset the exploration counters and B_{V}. We keep B_{W} and add a rank-support to B_{W} in order to get the witness ranks.

Assume that we visit the leaf ℓ corresponding to the x-th factor during the pass, i.e., ℓ is the x-th visited corresponding leaf. As in Section 4.1, we first determine the edge witness v of ℓ. Then we determine the referred index of the x-th factor by distinguishing two cases:

- If v is a witness and $y := W[B_{\mathrm{W}}.\mathrm{rank}_1(v)] \neq \perp$, then the x-th factor refers to the y-th factor (the y-th factor is represented by an LZ node that is the parent of the LZ node representing the x-th factor).
- Otherwise (there is no entry in W for v), we have two cases regarding the parent of v:
  - If v is a child of the root, then the x-th factor is a fresh factor (otherwise v would be a witness with an entry in W equal to the referred index of the x-th factor).
  - Otherwise (v's parent is a witness), the x-th factor refers to $W[B_{\mathrm{W}}.\mathrm{rank}_1(\mathrm{parent}(v))]$ (the x-th factor is represented by an LZ node that is the first node on the edge from $\mathrm{parent}(v)$ to v).

Afterwards, if v is a witness, we update W by setting $W[B_{\mathrm{w}}.\mathrm{rank}_1(v)]$ to x.

Up to now, we can output the referred index of the x-th factor during this pass in text order. Finally, the last character of the x-th factor can be obtained by $\mathrm{head}(\ell')$, where ℓ' is the leaf that occurs in text order *before* the leaf corresponding to the $(x+1)$-th factor.

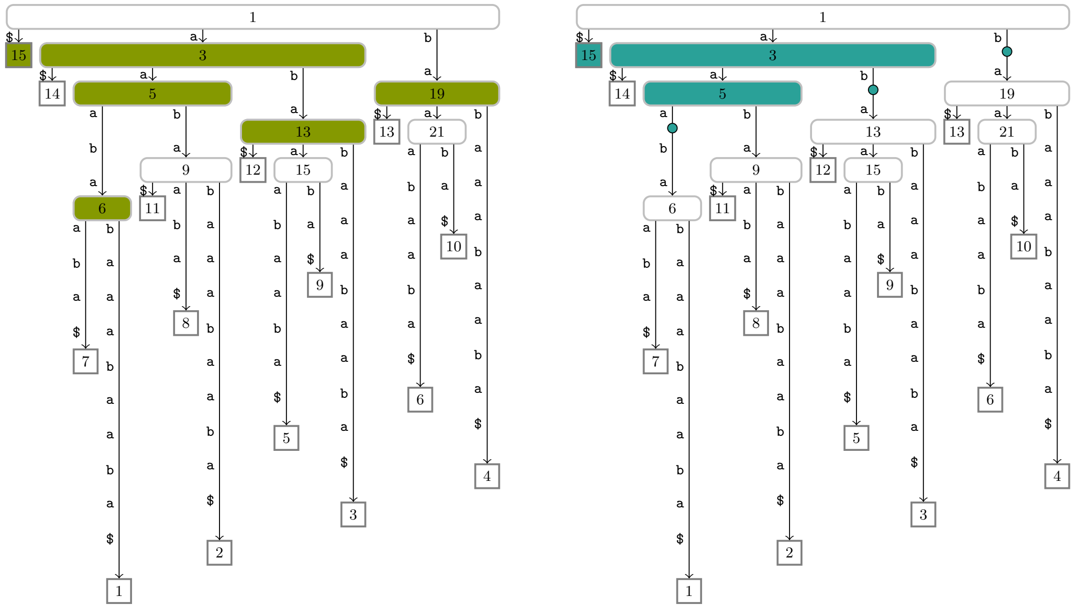
We summarize the result of this algorithm in the following theorem:

Theorem 4.2. *Given the compressed suffix tree of T, we can compute the LZ78 factorization in $\mathcal{O}(n)$ time using $3n + z \lg z + o(n)$ bits of working space when streaming the output.*

Proof. Maintaining the exploration counters as described in Section 4.1 takes n bits. With the space of B_{V} and B_{W} this sums up to $3n + o(n)$ space. The array W uses $z \lg z$ bits. □

Corollary 4.3. *We can compute and stream the LZ78 factorization of a text of length n in $\mathcal{O}(n)$ time using $\mathcal{O}(n \lg \sigma)$ bits of space.*

Proof. Analogous to Corollary 3.4. □



(a) Suffix Tree with Marked Edge Witnesses

(b) Superimposed LZ Trie with Nodes Marked in B_{LZ}

Preorder number	111111111122222 123456789012345678901234
B_{E}	011011000000100000100000

(c) Bit Vector B_{E}

Preorder Enumeration	01 234 56 78
Balanced Parentheses	(()((())(()))(()))
B_{LZ}	1 111 10 10

(d) Bit Vector B_{LZ}

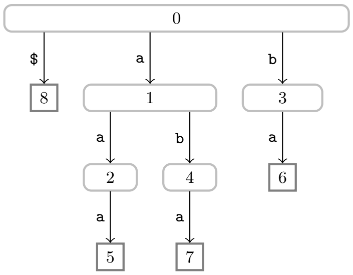
Figure 11: The suffix tree (a) and the LZ trie superimposed on the suffix tree (b). The shaded suffix tree nodes in the left image and the shaded LZ nodes in the right image are the ones marked in B_{E} and B_{LZ}, respectively, as explained in Section 4.2.2. The bit vectors B_{E} and B_{LZ} mark the same number of nodes. In particular, there is a bijection between the suffix tree nodes marked in B_{E} and the LZ nodes marked in B_{LZ}. The bijection is induced by adding rank- and select-support data structures to B_{E} and B_{LZ}.

4.2.2 Explicitly Storing the LZ Trie

In some applications we are interested in the LZ trie instead of the LZ78 factorization. We provide such an application in Lemma 4.6 that shows a data structure built on top of the LZ trie retrieving small substrings of T efficiently. It is easy to compute the LZ trie after having computed the LZ78 factorization with the approach in Section 4.2.1. We can modify it to build a pointer based tree data structure storing the LZ trie in $\mathcal{O}(z \lg z)$ bits. Here, we present a succinct variant of the LZ trie. It is composed of three data structures:

- a balanced parentheses sequence storing the LZ trie topology,
- an array W' with $z \lg z$ bits storing the factor indices, and
- an array with $z \lg \sigma$ bits storing the last character of each factor, i.e., the labels of the LZ trie edges (see Figure 12).

In this section, we show that the succinct LZ trie can be computed in $\mathcal{O}(n)$ time by adapting the approach of Section 4.2.1. The difference is that we exchange Pass (b) with a pass that computes W' instead of the



(a) LZ Trie

Preorder Enumeration	01 234 56 78
Balanced Parentheses	(()((())(()))(()))
W'	8 125 47 36
Array of Characters	$ aaa ba ba

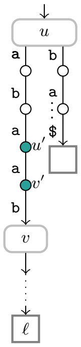
(b) Representation of the LZ Trie in Section 4.2.2

Figure 12: The LZ trie (a) represented by three data structures (b). The balanced parentheses sequence represents its topology, the array of factor indices W' stores the labels of the nodes, and the array of characters stores the edge labels (we identify an edge with its incoming node).

LZ78 factorization, and that we perform an Euler tour on the suffix tree to compute the other two data structures.

In more detail, we build the balanced parentheses sequence directly after Pass (a) of Section 4.2.1. We keep the exploration counters computed during Pass (a) in memory, i.e., we do not clear/erase the exploration counters after the phase has finished. With the exploration counters we can construct the LZ trie since there are exactly n_v LZ trie nodes on the suffix tree edge of a node v to its parent u. The exploration counter n_v is either

- equal to $c(u, v)$ if v is fully explored,
- stored in B_{C} in case that v is a partially explored node (see Section 4.1), or
- 0 if v's parent u is not fully explored.



To build the balanced parentheses sequence, we perform an Euler tour on the suffix tree, i.e., we traverse the suffix tree with a depth first search starting at the root. We write n_v open (resp. close) parenthesis when visiting a suffix tree node v on walking down (resp. climbing up) during the Euler tour. Although looking up $c(u, v)$ for a fully explored node v takes $\mathcal{O}(c(u, v))$ time, this time is bounded by the output of the balanced parentheses sequence. An Euler tour can be performed in $\mathcal{O}(n)$ time, since the suffix tree can compute child(1), next_sibling(), and parent() in constant time (see Section 2.2.1).

During the Euler tour we build the $z \lg \sigma$-bits array storing the LZ trie edge labels: Assume that we create the LZ nodes u' and v' on the suffix tree edge (u, v), where u' is the parent of v'. We can obtain the character of the edge (u', v') by invoking next_leaf and head: Given that s is the string depth of u' in the LZ trie, applying next_leaf s times to a leaf ℓ in the subtree rooted at v returns a leaf ℓ' whose head(ℓ')-value is the label in question. Overall, we can compute the balanced parenthesis representation and the edge labels in $\mathcal{O}(n)$ time taking $2z + z \lg \sigma + o(z)$ bits ($o(z)$ bits for the navigation component [49]).

In order to compute the array W' storing the factor indices we will exchange Pass (b) with an alternative LZ78 pass: Assume that we increment the exploration counter of a node v while processing the x-th factor during this pass. This means that the x-th factor is represented by the n_v-th LZ node on the edge of v to v's parent. Our task is to store the factor index x in W' at the position equal to the preorder number of this LZ node. To compute this preorder number in constant time, we do the following precomputation step, which is the topic of this paragraph: We compute an isomorphic mapping between the edge witnesses and a subset of LZ nodes such that the mapping maps an edge witness v to an LZ node v' with the following properties: First, the LZ trie parent of v' is an explicit LZ node that is represented by the suffix tree parent of v (v has a parent since the root is not an edge witness). Second, there are $n_v - 1$ LZ nodes below v' forming a path.

To compute this mapping in constant time, we create a bit vector B_{E} of length $2n$ marking suffix tree nodes and a bit vector B_{LZ} of length z marking LZ nodes, see Figure 11. The former marks the edge witnesses, the latter marks all LZ trie children of every explicit LZ node. The last thing we do is adding a rank-support to B_{E}, and a select-support to B_{LZ}.

Now we can map an edge witness v to the LZ node $v' = B_{\mathrm{LZ}}.\operatorname{select}_1(B_{\mathrm{E}}.\operatorname{rank}_1(v))$ (conceptually $B_{\mathrm{LZ}}.\operatorname{rank}_1(v') = B_{\mathrm{E}}.\operatorname{rank}_1(v)$) such that v' satisfies the above described properties of the mapping.

Finally, we explain the last pass.

Pass (b'). We recompute B_{V} and the exploration counters. We count the current factor index with a variable x. Assume that we visit the leaf ℓ corresponding to the x-th factor. Let v be the edge witness of ℓ. We assign the label x to the LZ node corresponding to the x-th factor by setting $W'[B_{\mathrm{LZ}}.\operatorname{select}_1(B_{\mathrm{E}}.\operatorname{rank}_1(v)) + n_v - 1]$ to x (after incrementing n_v by one like in a standard LZ78 pass). In this operation, we select the LZ trie node $v' := B_{\mathrm{LZ}}.\operatorname{select}_1(B_{\mathrm{E}}.\operatorname{rank}_1(v))$ corresponding to the edge witness v, and select the LZ trie descendant of v' at depth $\operatorname{depth}(v') + n_v - 1$ by adding $n_v - 1$ to the (LZ trie) preorder number of v' (the next $n_v - 1$ descendants of v' are on a unary path below v', thus the preorder numbers of all those descendants are direct successors of the preorder number of v').

Since W' is the last component of our data structure, we get the following

Theorem 4.4. *Given the compressed suffix tree of T, we can build the LZ trie and store it in RAM taking $z(\lg \sigma + \lg z + 2) + o(z)$ bits. The construction algorithm takes $5n + z + o(n)$ additional bits of working space and runs in $\mathcal{O}(n)$ time.*

Proof. Maintaining the exploration counters as described in Section 4.1 takes n bits. With the space of B_{V} (n bits), B_{W} (n bits) and B_{E} ($2n + o(n)$ bits), this sums up to $5n + o(n)$ space. The bit vector B_{LZ} takes $z + o(z)$ bits. The balanced parentheses sequence needs $2z + o(z)$ bits. Each of the z LZ nodes stores a factor index using $\lg z$ bits and an ending character using $\lg \sigma$ bits. □

Corollary 4.5. *We can compute the LZ trie of a text of length n in $\mathcal{O}(n)$ time using $\mathcal{O}(n \lg \sigma)$ bits of space.*

Proof. Analogous to Corollary 4.3. □

Lastly, we introduce an application of our constructed LZ trie sketched in the beginning of this section in more detail:

Lemma 4.6 ([57, Lemma 2.6]). *Given the LZ trie of T, but not necessarily T, there is a data structure built on top of the LZ trie that can recover a substring of T of length $\mathcal{O}(\lg_\sigma n)$ in constant time. It takes $z \lg z + 3z \lg \sigma + 5z + \mathcal{O}\big(n^{3/4} \lg^2 n\big) + o(z)$ additional bits of space, and can be constructed in-place in linear time.*

	(a)	(b)	Interim	(M)
X	ISA	L	$L \mid W$	Referred Indices
Y	SA	-	-	-

Figure 13: Chronological table of the LZ78 factorization in Section 4.3, structure equivalent to Figure 7. Having $|X| = n \lg n$ bits and $|Y| = \epsilon n \lg n$ bits, the suffix array stored initially in Y can only be used in conjunction with ISA stored initially in X.

The data structure of Sadakane and Grossi [57] needs an LZ trie representation that slightly differs from the one that we have computed: It needs an array that maps a factor index to the preorder number of its corresponding LZ node. This array is the inverse of W' (W' is a permutation of integers). In addition, it needs a bit vector B_{T} marking the factor positions. This bit vector can be computed during Pass (b') by setting $B_{\mathrm{T}}[\mathrm{label}(\ell)]$ to 1 for each corresponding leaf.

Corollary 4.7. *Given a text T, the data structure of Lemma 4.6 can be computed in linear time with $\mathcal{O}(n \lg \sigma)$ bits of working space. It takes $2z \lg z + 4z \lg \sigma + 7z + \mathcal{O}\big(n^{3/4} \lg^2 n\big) + o(z)$ bits of space in total (including the LZ trie).*

4.3 Alphabet-Independent LZ78 Factorization

In this variant we store the referred indices in the array X and mark the factor positions in a bit vector B_{T} of length n such that the referred index and the last character of the x-th factor are $X[x]$ and $T[B_{\mathrm{T}}.\mathrm{select}_1(x+1) - 1]$, respectively.

Having $\mathcal{O}(n)$ bits of working space on top of the space used by the succinct suffix tree, our idea is to overwrite the array X to free up space. We will overwrite X multiple times during the factorization (see Figure 13). After overwriting X (storing ISA initially), we will no longer have access to SA and therefore cannot evaluate $c(\cdot)$ needed to decide whether an edge is fully explored. Fortunately, it is sufficient to know the maximum exploration counter of every witness — a node that is not a witness has an exploration counter of either zero (not touched during an LZ78 pass) or one (represented as an LZ78 trie leaf). We therefore aim at storing the maximum value of the exploration counter of every witness before overwriting X. The maximum values can be determined in one pass. We can store them in a bit vector B_{L} of length at most $z + z_{\mathrm{W}}$, since we increment the exploration counters exactly z times. To this end, we store the exploration value n_w unary with $\mathtt{0}^{n_w}\mathtt{1}$ in B_{L} for every witness w; hence B_{L} has z_{W}-many '1's. We store the values sorted by the preorder numbers of the witnesses such that $B_{\mathrm{L}}.\mathrm{rank}_1(B_{\mathrm{W}}.\mathrm{rank}_1(w) + 1) - B_{\mathrm{L}}.\mathrm{rank}_1(B_{\mathrm{W}}.\mathrm{rank}_1(w)) - 1$ returns n_w.

The algorithm is divided into two passes and a matching phase:

(a) Determine the witnesses and mark them in B_{W}. Write the maximum exploration counters of each *witness* in B_{L} unary.

(b) Mark the factor positions in a bit vector B_{T}, and create an array $L[1..z]$ storing the preorder number of the *edge witness* of each factor.

(M) Use the witness ranks (a subset of the ranks of the edge witnesses) stored in the array $L[1..z]$ to identify the referred indices.

z	#internal-nodes	#leaves
z_{F}	α	β
z_{R}	γ	δ

Table 1: Contingency table depicting the partitioning of all factors in the proof of Lemma 4.8. The columns #internal-nodes and #leaves are defined as the number of internal nodes and the number of leaves in the LZ trie, respectively.

Pass (a). After performing a single LZ78 pass we have determined the witnesses and the edge witnesses. Now we aim at creating the bit vector B_{L}. To this end, we have to determine the exploration counters of every witness. By our bookkeeping method of the exploration counters in Section 4.1, we know the exploration counter of all partially explored edges. Further, the maximum exploration counter of a fully explored node v is equal to the $c(u, v)$-value of the edge (u, v) of the node v to its parent u. Altogether we have the information needed to create the bit vector B_{L}. The exploration values stored in B_{L} are computed by an Euler tour on the suffix tree in depth first order: If a witness w is marked in B_{V}, then n_w is equal to $c(u, w)$, where u is the parent of w. Otherwise (w is not marked in B_{V}), n_w is stored in the bit vector B_{C} ($n_w > 0$ since w is a witness).

With B_{L} we no longer need access to SA (e.g., needed for accessing $c(\cdot)$). This is crucial, since we can access SA only when the arrays X and Y are left untouched (X stores ISA and Y stores the data structure of Lemma 2.7). In the next pass, we will overwrite the ISA entries stored in X after they are no longer used.

Pass (b). The main goal of this pass is the creation of the array L. Its contents are created sequentially during the pass: When performing a traversal from a leaf ℓ corresponding to the x-th factor, we write v to $L[x]$, where v is the *edge* witness of ℓ. With L it will be easy to find the referred index y of a referencing factor f_x. That is because f_y will either share the edge witness with f_x, or $L[y]$ is the parent node of $L[x]$. The latter condition always holds if $n_{L[x]} = 1$, in particular if $L[x]$ is a leaf because we can create at most one LZ78 node on the edge to a leaf.

Since L takes $z \lg n$ bits, we store L in the first z positions of the array X. Although X stores ISA, necessary for next_leaf, we visit the same leaf never again such that we can sequentially overwrite $X[x]$ with $L[x]$ for all $1 \leq x \leq z$ in increasing order.

We can compute B_{T} simultaneously: Given the leaf ℓ with label x and its edge witness v, the length of a factor f_x is computed by summing up the $c(\cdot)$-values along the traversed path, plus n_v's value.

Matching. Matching the factors with their references can now be done with L in a straightforward manner. Let us consider a referencing factor f_x having the referred factor f_y. We have two cases: Whenever f_y is explicitly represented by a node v (i.e., by f_y's witness), v is the parent of f_x's witness. Otherwise, f_y has an implicit representation and hence has the same witness as f_x: If $L[x]$ does not occur in $L[1..x-1]$, then f_y is determined by the largest position $y < x$ for which $L[y] = \mathrm{parent}(L[x])$; otherwise ($L[x]$ is *not* the first occurrence of $L[x]$ in L), the referred factor of f_x is determined by the largest $y < x$ with $L[x] = L[y]$.

Remember that we store L in $X[1..z]$, leaving us $X[z + 1..n]$ as free working space that will be occupied by a new array W, storing for each witness w the index of the most recently processed factor whose witness is w. Fortunately, this space is sufficient due to

Lemma 4.8. *$z + z_{\mathrm{W}} \leq n$.*

Proof. Let α and β be the number of fresh factors that are internal LZ nodes and LZ trie leaves, respectively.

Also, let γ and δ be the number of referencing factors that are internal LZ nodes and LZ trie leaves, respectively. Obviously, $\alpha + \beta + \gamma + \delta = z$ (see Table 1). With respect to the factor length, each referencing factor has a length at least 2, while each fresh factor is exactly one character long. Hence $n \geq 2(\gamma+\delta)+\alpha+\beta = z + \gamma + \delta$. Since each LZ78 leaf that is counted by δ has an LZ78 internal node of depth one as ancestor (counted by α), $\alpha \leq \delta$ holds. Since the number of witnesses is bounded by the number of internal LZ nodes, we yield $z + z_{\mathrm{W}} \leq z + \alpha + \gamma \leq z + \gamma + \delta \leq n$. □

Finally, we describe how to convert L (stored in $X[1..z]$) into the referred indices, such that in the end $X[x]$ contains the referred index of f_x for $1 \leq x \leq z$. We scan $L = X[1..z]$ from left to right. During the scan, for each witness v, we keep track of the index of the most recently visited factor f whose witness is v by storing f's index in $W[B_{\mathrm{W}}.\operatorname{rank}_1(v)]$. Suppose that we are now processing f_x with $v := L[x]$.

- If v is not a witness or $W[B_{\mathrm{W}}.\operatorname{rank}_1(v)]$ is empty, we check the parent of v. If v is a child of the root, then f_x is a fresh factor. Otherwise, its referred index is $W[B_{\mathrm{W}}.\operatorname{rank}_1(\operatorname{parent}(v))]$.
- Otherwise (v is a witness and $W[B_{\mathrm{W}}.\operatorname{rank}_1(v)]$ has a value), $W[B_{\mathrm{W}}.\operatorname{rank}_1(v)]$ is the referred index of f_x.

In either case, if v is a witness, we update W by writing the current factor index x to $W[B_{\mathrm{W}}.\operatorname{rank}_1(v)]$. After processing f_x, we no longer need the value $X[x]$. Hence, we can write the referred index of f_x to $X[x]$ (if it is a referencing factor) or set $X[x]$ to 0 (if it is a fresh factor). In the end, $X[1..z]$ stores the referred indices of every referencing factor.

Overall we obtain the following result:

Theorem 4.9. *Allowing the succinct suffix tree of T to be* rewritable*, we can overwrite it with the LZ78 factorization in $\mathcal{O}(n)$ time using $3n + o(n)$ bits of working space.*

Proof. Maintaining the exploration counters as described in Section 4.1 takes n bits. With the space of B_{T} marking the factor positions and B_{L} storing the maximum exploration values of the witnesses, this sums up to $3n + o(n)$ space. □

Corollary 4.10. *We can compute the LZ78 factorization of a text of length n in $\mathcal{O}(n/\epsilon^2)$ time using $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits of space. The factors are stored in-place.*

Proof. We create the succinct suffix tree of Theorem 2.8 to compute the LZ78 factorization. During the factorization, we overwrite the $n \lg n$ bits of space used for representing the inverse suffix array in order to compute the LZ78 factorization in-place according to Theorem 4.9. □

5 Conclusions

We presented novel approaches for the LZ77 and the LZ78 factorization. Our main idea was based on two different representations of the suffix tree that are especially trimmed on a small memory footprint during their construction. The algorithms for small alphabets take $\mathcal{O}(n \lg \sigma)$ bits of total space, the alphabet independent algorithms take $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits of total space. All algorithms run in linear time. Although we were careful about the choice of the suffix tree representations, they are still the bottleneck in terms of the working space. We therefore hope that our algorithms could work with less space in the light of future achievements in suffix tree construction algorithms.

Aside from this improvement we would like to see whether it is possible to compute LZ77 or LZ78 with a small memory footprint around $\mathcal{O}(z \lg z + n)$ bits in nearly linear time, or with $n \lg n + \mathcal{O}(n)$ bits in linear time. Sampled data structures could further drop the space requirements of suffix trees. A sampling could provide a trade-off between the space and time bounds.

Acknowledgements

We thank the anonymous reviewers for their careful reading of our manuscript and their insightful comments and suggestions. We are especially grateful for the reviewer pointing out a simplification of our original solution on how to store the exploration counters for the LZ78 factorizations (Section 4.1). Further, we are grateful to Florian Kurpicz for proof-reading the paper, and to Sean Tohidi, who spell-checked the initial submission of this paper during his DAAD RISE internship at TU Dortmund.

This research was supported by CREST, JST.

References

[1] A. Amir, M. Farach, R. M. Idury, J. A. L. Poutré, and A. A. Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.

[2] D. Arroyuelo and G. Navarro. Space-efficient construction of lempel-ziv compressed text indexes. *Inf. Comput.*, 209(7):1070–1102, 2011.

[3] D. Belazzougui. Linear time construction of compressed text indices in compact space. In *Proc. STOC*, pages 148–193. ACM, 2014.

[4] D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. SODA*, pages 2053–2071. ACM/SIAM, 2016.

[5] D. Belazzougui, V. Mäkinen, and D. Valenzuela. Compressed suffix array. In *Encyclopedia of Algorithms*, pages 386–390. Springer, 2016.

[6] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

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

[8] M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.

[9] M. Crochemore, G. M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput.*, 32(6):1654–1673, 2003.

[10] J. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre. Linear-time computation of local periods. *Theor. Comput. Sci.*, 326(1-3):229–240, 2004.

[11] H. El-Zein, J. I. Munro, and M. Robertson. Raising permutations to powers in place. In *Proc. ISAAC*, volume 64 of *LIPIcs*, pages 29:1–29:12. Schloss Dagstuhl, 2016.

[12] M. Farach. Optimal suffix tree construction with large alphabets. In *Foundations of Computer Science*, pages 137–143. IEEE Computer Society, 1997.

[13] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[14] J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. CPM*, volume 9133 of *LNCS*, pages 160–171. Springer, 2015.

[15] J. Fischer and V. Heun. Space efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

[16] J. Fischer, T. I, and D. Köppl. Lempel-Ziv computation in small space (LZ-CISS). In *Proc. CPM*, volume 9133 of *LNCS*, pages 172–184. Springer, 2015.

[17] G. Franceschini, S. Muthukrishnan, and M. Pătraşcu. Radix sorting with no extra space. In *Proc. ESA*, volume 4698 of *LNCS*, pages 194–205. Springer, 2007.

[18] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. LATA*, volume 7183 of *LNCS*, pages 240–251. Springer, 2012.

[19] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. LATIN*, volume 8392 of *LNCS*, pages 731–742. Springer, 2014.

[20] K. Goto. Optimal time and space construction of suffix arrays and lcp arrays for integer alphabets. *ArXiv CoRR*, abs/1703.01009, 2017.

[21] K. Goto and H. Bannai. Simpler and faster Lempel Ziv factorization. In *Proc. DCC*, pages 133–142. IEEE Computer Society, 2013.

[22] K. Goto and H. Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Proc. DCC*, pages 163–172. IEEE Computer Society, 2014.

[23] 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.

[24] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.

[25] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. FOCS*, pages 251–260. IEEE Computer Society, 2003.

[26] G. J. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.

[27] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees with applications. *Journal of Computer and System Sciences*, 78(2):619 – 631, 2012.

[28] J. Jansson, K. Sadakane, and W.-K. Sung. Linked dynamic tries with applications to LZ-compression in sublinear time and space. *Algorithmica*, 71(4):969–988, 2015.

[29] J. Kärkkäinen and E. Sutinen. Lempel-Ziv index for q-grams. *Algorithmica*, 21(1):137–154, 1998.

[30] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *South American Workshop on String Processing (WSP)*, pages 141–155. Carleton University Press, 1996.

[31] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6): 1–19, 2006.

[32] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. CPM*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013.

[33] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proc. SEA*, volume 7933 of *LNCS*, pages 139–150. Springer, 2013.

[34] D. Kempa and S. J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proc. ALENEX*, pages 103–112. SIAM, 2013.

[35] T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. A linear time algorithm for seeds computation. In *Proc. SODA*, pages 1095–1112. ACM/SIAM, 2012.

[36] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. FOCS*, pages 596–604, 1999.

[37] R. M. Kolpakov and G. Kucherov. Finding repeats with fixed gap. In *Proc. SPIRE*, pages 162–168. IEEE Computer Society, 2000.

[38] D. Köppl and K. Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. DCC*, pages 3–12. IEEE Computer Society, 2016.

[39] M. Li and R. Sleep. An LZ78 based string kernel. In *Proc. ADMA*, volume 3584 of *LNCS*, pages 678–689. Springer, 2005.

[40] M. Li and Y. Zhu. Image classification via LZ78 based string kernel: A comparative study. In *Proc. PAKDD*, volume 3918 of *LNCS*, pages 704–712. Springer, 2006.

[41] Z. Li, J. Li, and H. Huo. Optimal in-place suffix sorting. *ArXiv CoRR*, abs/1610.08305, 2016.

[42] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1-2):145–153, 1989.

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

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

[45] Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda. Constructing LZ78 tries and position heaps in linear time for large alphabets. *Inform. Process. Lett.*, 115(9):655 – 659, 2015.

[46] G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discrete Algorithms*, 2(1):87–114, 2004.

[47] G. Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016.

[48] G. Navarro and Y. Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43(5): 1781–1806, 2014.

[49] G. Navarro and K. Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):Article No. 16, 2014.

[50] G. Nong. Practical linear-time $\mathcal{O}(1)$-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):15, 2013.

[51] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Proc. SPIRE*, volume 6393 of *LNCS*, pages 322–333. Springer, 2010.

[52] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng. FPGA implementation of GZIP compression and decompression for IDC services. In *Proc. FPT*, pages 265–268. IEEE Computer Society, 2010.

[53] G. G. Richard and A. Case. In lieu of swap: Analyzing compressed RAM in Mac OS X and Linux. *Digital Investigation*, 11, Supplement 2(0):3–12, 2014.

[54] L. M. S. Russo, G. Navarro, and A. L. Oliveira. Fully-compressed suffix trees. In *Proc. LATIN*, volume 4957 of *LNCS*, pages 362–373. Springer, 2008.

[55] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. SODA*, pages 225–237. ACM/SIAM, 2002.

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

[57] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. SODA*, pages 1230–1239. ACM/SIAM, 2006.

[58] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

[59] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit. Engineering a compressed suffix tree implementation. *ACM J. Experimental Algorithmics*, 14:Article No. 2, 2009.

[60] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, 23(3):337–343, 1977.

[61] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.

A Appendix

A.1 List of Identifiers

While describing both factorization algorithms, we used several data structures, among others bit vectors, some with rank or select-support, to achieve the small space bounds. We denote bit vectors with B_α for some letter α.

For all types of LZ-factorizations we use

- B_{W} marking all witness nodes,
- the array W mapping witness ranks to
  - (LZ77) text positions, or
  - (LZ78) factor indices.

In LZ77 we use

- B_{V} marking visited nodes, and

In LZ78 we use

- B_{C} counts n_v of each partially explored node v,
- B_{V} marking suffix tree nodes represented in the LZ trie (their ingoing edges are fully explored),
- B_{LZ} marking explicit LZ nodes, and
- B_{E} marking the edge witnesses.
- the array W' mapping LZ nodes to factor indices

The algorithms based on the SST additionally use

- B_{T} marking the factor positions, used also for representing the length of a factor.

We count the number of

- factors by z
- witnesses by z_{W}
- referencing factors by z_{R}
- fresh factors by z_{F}

Section	CST	SST
LZ77 (Section 3)		
Section 3.1		○
Section 3.2	○	
Section 3.3		○
LZ78 (Section 4)		
Section 4.2.1	○	
Section 4.2.2	○	
Section 4.3		○

Figure 14: Connection between the introduced algorithms and the used suffix tree representations. The figure shows (by marking with a circle) which suffix tree representation is used by an algorithm (introduced in the respective section).

LZ77 Output-Streaming with SST, Section 3.1

name	bits	rank	select	in-place
RMQ.SA	$2n + o(n)$			
B_{V}				

Common after Section 3.1

name	bits	rank	select	in-place
B_{W}	$n + o(n)$	○		
B_{V}				

LZ77 CST, Section 3.2

name	bits	(a)	(b)	rank	select	in-place
W	$z \lg n$	○	○			

LZ77 SST In-Place, Section 3.3

name	bits	(a)	(b)	(c)	(M)	rank	select	in-place
D	$(z_{\mathrm{W}} + z_{\mathrm{R}}) \lg n$			○				○
B_{D}	$n + z_{\mathrm{W}} + z_{\mathrm{R}}$		○	○	○			
B_{T}	$n + o(n)$					○	○	

Common for LZ78, Section 4.1

name	bits	rank	select	in-place
B_{C}				
B_{V}				

LZ78 CST, Section 4.2

name	bits	(a)	(b)	(b')	rank	select	in-place
B_{LZ}	$z + o(z)$			○		○	
B_{E}	$2n$			○			
W/W'	$z \lg z$	○	○	○			

LZ78 SST, Section 4.3

name	bits	(b)	(M)	rank	select	in-place
L	$z \lg n$	○	○			○
W	$z_{\mathrm{W}} \lg n$		○			○
B_{L}	$z + z_{\mathrm{W}}$			○		
B_{T}	$n + o(n)$			○	○	

Table 2: List of data structures with names. The list comprises additional data structures used while computing the LZ77/78 factorization of a text of length n. The letters written in brackets represent a pass (e.g., (a) refers to Pass (a)). The number of bits is omitted if it is exactly n. Circles symbolize that the data structure is used during a pass, or that it is used with a rank or select structure, or that the data structure is stored in the available working space.