LZ78 Substring Compression in Compressed Space

Hiroki Shibata[1*] and Dominik Köppl[2]

[1*]Joint Graduate School of Mathematics for Innovation, Kyushu University, 744, Motooka, Nishi-ku, 819-0395, Fukuoka, Japan.
[2]Department of Computer Science and Engineering, University of Yamanashi, 4-4-37 Takeda, Kofu, 400-8510, Yamanashi, Japan.

*Corresponding author(s). E-mail(s): shibata.hiroki.753@s.kyushu-u.ac.jp;
Contributing authors: dkppl@yamanashi.ac.jp;

Abstract

The Lempel–Ziv 78 (LZ78) factorization is a well-studied technique for data compression. It and its derivates are used in compression formats such as `compress` or `gif`. Although most research focuses on the factorization of plain data, not much research has been conducted on indexing the data for fast LZ78 factorization. Here, we study the LZ78 factorization and its derivates in the substring compression model, where we are allowed to index the data and return the factorization of a substring specified at query time. In that model, we propose an algorithm that works in compressed space, computing the factorization with a logarithmic slowdown compared to the optimal time complexity.

Keywords: lossless data compression, LZ78 factorization, substring compression, CDAWG

1 Introduction

The substring compression problem [1] is to preprocess a given input text T such that computing a compressed version of a substring of $T[i..j]$ can be performed efficiently. This problem has been originally stated for the Lempel–Ziv-77 (LZ77) factorization [2], but extensions to the generalized LZ77 factorization [3], the Lempel–Ziv 78 (LZ78) factorization [4] as well as two of its derivates [5], the run-length encoded Burrows–Wheeler transform (RLBWT) [6], and the relative LZ factorization [7, Sect. 7.3] have been studied. Given that n is the length of T, a trivial solution would be to precompute the compressed output of $T[\mathcal{I}]$ for all intervals $\mathcal{I} \subseteq [1..n]$. This, however, gives us already $\Omega(n^2)$ solutions to store.

For an appealing solution, we want to be able to index a large amount of data efficiently within a fraction of space; two criteria (speed and space) that are likely to be anti-correlated. However, as far as we are aware, the substring compression problem has not yet been studied with compressed space bounds that can be sublinear for compressible input data. Our main target is therefore a solution that works in compressed space and can answer a query in time linear in the output size with a polylogarithmic term on the text length.

In this paper, we build upon the line of research on LZ78 factorization algorithms that superimpose the LZ trie on the suffix tree [4, 5, 8–10], which all use the $\Omega(n)$ space (in words) to store the suffix tree. Here, we make the algorithmic idea of the superimposition compatible with the compact directed acyclic word graph (CDAWG) [11], trading a tiny time penalty with a large space

Table 1 Solutions for computing the LZ78 (or alternatively LZD, or LZMW) substring compression for a string of length n over an alphabet of size σ, with z LZ78 (LZD, or LZMW) factors, e CDAWG edges, r BWT runs, and having substring complexity δ [14]. Extra space means the additional working space required for processing queries in addition to the index. All space complexities are measured in words.

method	space		time
	index	extra space	query
naive	$\Omega(n^2)$	$\mathcal{O}(z)$	$\mathcal{O}(z)$
Köppl [4]	$\mathcal{O}(n)$	$\mathcal{O}(z)$	$\mathcal{O}(z)$
Theorem 2	$\mathcal{O}(e)$	$\mathcal{O}(z)$	$\mathcal{O}(z \lg n)$
Theorem 3	$\mathcal{O}\left(r \lg \frac{n}{r}\right)$	$\mathcal{O}(z)$	$\mathcal{O}\left(z \left(\lg \lg \frac{r}{\lg n} + \lg \frac{n}{r} + \lg z\right)\right)$
Theorem 4	$\mathcal{O}\left(\delta \lg \frac{n \lg \sigma}{\delta \lg n}\right)$	$\mathcal{O}(z)$	$\mathcal{O}\left(z \lg^{4+\varepsilon} n\right)$

improvement for compressible texts. Furthermore, we generalize the LZ78 substring compression algorithm to work with any data structure that can perform a limited set of suffix tree operations. As a result, we obtain alternative solutions for computing the LZ78 substring compression. We also apply the proposed method for LZ78 substring compression to LZD [12] and LZMW [13] compression, which are derivates of the LZ78 factorization. Table 1 summarizes both known and newly proposed solutions for the substring compression problem of LZ78, LZD, and LZMW.

Our contribution fits into the line of research focused on data compression with the CDAWG. Given e and z are the number of edges of the CDAWG and the number of LZ78 factors, respectively, in that line, [15] proposed a straight-line program (SLP), which can be computed in $\mathcal{O}(e)$ time taking $\mathcal{O}(e)$ words of space. Given an SLP of size $\mathcal{O}(g)$, [16] showed how to compute LZ78 from that SLP in $\mathcal{O}(g + z \lg z)$ time and space. Combining both solutions, we can compute LZ78 from the CDAWG in $\mathcal{O}(e + z \lg z)$ time and space. Recently, [17] showed how to compute, among others, the RLBWT and LZ77 in $\mathcal{O}(e)$ time and space.

Beyond LZ78, variants such as LZD [12] and LZMW [13] have been introduced to improve compression performance. Both factorizations represent each factor as the concatenation of two previous factors. Since the length of each factor can grow exponentially in the best case, the lower bound on the number of factors becomes $\Omega(\log n)$. This is significantly smaller than the lower bound for standard LZ78 factorization, which is $\Omega(\sqrt{n})$. However, while the standard LZ78 factorization can be computed in an online manner in $O(n)$ time using $O(z)$ additional words of space where z is the number of LZ78 factors, computing these two variants within the same additional space is more difficult. [18] showed that the greedy algorithm for these variants takes $\Omega(n^{5/4})$ time in the worst case. The fastest known solution for this problem runs in expected $O(n + z' \log^2 n)$ time, where z' is the size of the corresponding parsing [18]. This gap in the computational complexity motivates the investigation into the possibility of computing these factorizations in compressed space.

We also note that our research can be viewed as a practical application of compressed suffix trees. The majority of the queries needed to compute substring compressions correspond to standard suffix tree operations. Indeed, the compressed indexes we employ can be seen as specific instances or variants of compressed suffix trees. Our approach shows how to maintain the LZ78, LZD, and LZMW tries within this framework, leveraging the capabilities of these compressed structures. This perspective not only unifies different parsing algorithms but also broadens the applicability of compressed suffix tree-based methods.

This article is an extended version of our contribution [19] to the SPIRE'24 conference. Our extension is as follows. First, we have adapted our techniques for substring compression to LZD and LZMW. Second, we formulate an abstract data type for the data structure we need for our algorithm computing the substring compression for any of the addressed compression types (LZ78/LZMW/LZD). We subsequently provide examples of implementations other than CDAWGs for this abstract data type. Finally, we provide additional experimental results.

2 Preliminaries

With lg we denote the logarithm $\log_2$ to base two. Our computational model is the word RAM model with machine word size $\Omega(\lg n)$ bits for a given input size n. Accessing a word costs $\mathcal{O}(1)$ time. Unless stated otherwise, we measure space complexity in words rather than bits.

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

LZ78 Factorization.

Stipulating that F_0 is the empty string, a factorization $F_1 \cdots F_z = T$ is called the *LZ78 factorization* [20] of T if and only if, for all $x \in [1..z]$, the factor F_x is the longest prefix of $T[|F_1 \cdots F_{x-1}|+1..]$ such that $F_x = F_y \cdot c$ for some $y \in [0..x-1]$ and $c \in \Sigma$, that is, F_x is the longest possible previous factor F_y appended by the following character in the text. The dictionary for computing F_x is $\mathcal{D}_x := \{F_y \cdot c : y \in [0..x-1], c \in \Sigma\}$, which is prefix-closed. Formally, F_x starts at $\mathsf{dst}_x := |F_1..F_{x-1}|+1$ and $y = \operatorname{argmax}\{|F_{y'}| : F_{y'} = T[\mathsf{dst}_x..\mathsf{dst}_x + |F_{y'}| - 1]\}$. We say that y and F_y are the *referred index* and the *referred factor* of the factor F_x, respectively. The LZ78 factorization of $T = \mathtt{babac}$ is $F_0, F_1, \ldots, F_4 = \epsilon, \mathtt{b}, \mathtt{a}, \mathtt{ba}, \mathtt{c}$. The referred factor of $F_3 = F_1\mathtt{a}$ is F_1; F_3's referred index is 1.

$\mathcal{D}_x$ is often implemented by the *LZ trie*, which represents each LZ factor as a node; the root represents the factor F_0. The node representing the factor F_y has a child representing the factor F_x connected with an edge labeled by a character $c \in \Sigma$ if and only if $F_x = F_y c$. To see the connection of the LZ trie and $\mathcal{D}_x$, we observe that adding any new leaf to the LZ trie storing $\{F_1, \ldots, F_{x-1}\}$ gives an element of $\mathcal{D}_x$, and vice versa we can obtain any element of $\mathcal{D}_x$ by doing so. A crucial observation is that every path from the LZ trie root downwards visits nodes in increasing LZ factor index order.

Lempel-Ziv Double (LZD) [12] and Lempel-Ziv-Miller-Wegman (LZMW) [13] factorization are non-prefix-closed variants of LZ78 factorization. A factorization $F_1 \cdots F_z$ of T is *LZD* if F_x is represented by $F_x = G_1 \cdot G_2$ with $G_1 \in \{F_0, F_1, \ldots, F_{x-1}\}, G_2 \in \{F_1, \ldots, F_{x-1}\} \cup \Sigma$ such that G_1 and G_2 are, respectively, the longest possible prefixes of $T[\mathsf{dst}_x..]$ and of $T[\mathsf{dst}_x + |G_1|..]$. A factorization $F_1 \cdots F_z$ of T is *LZMW* if $F_x = F_y \cdot F_{y+1}$ with $1 \leq y < x-1$ or $F_x \in \Sigma$ and all factors are greedily selected by length. Examples of the LZ78, LZD, and LZMW factorizations are shown in Fig. 1. Because non-prefix closed parsings may introduce unary paths in the parse trees, we need to study LZ tries having unary paths.

Suffix Tree.

Given a tree, with an *s-t path* we denote the path from a node s to a node t. All trees in this paper are considered non-empty with a root node, which we denote by root. The *suffix trie* of T is the trie of all suffixes of T. There is a one-to-one relationship between the suffix trie leaves and the suffixes of T. The *suffix tree* [21] ST of T is the tree obtained by compacting the suffix trie of T. The string stored in an ST edge g is called the *label* of g. The *string label* of a node v is defined as the concatenation of all edge labels on the root-v path; its *string depth* is the length of its string label. The leaf corresponding to the i-th suffix $T[i..]$ is labeled with the *suffix number* $i \in [1..n]$. The *locus* of a substring S of T is the place we end up when reading S from ST starting at root. The locus of S is either an ST node, or on an ST edge (called an *implicit node* because it is represented by a suffix trie node). The left of Fig. 2 gives an example of ST.

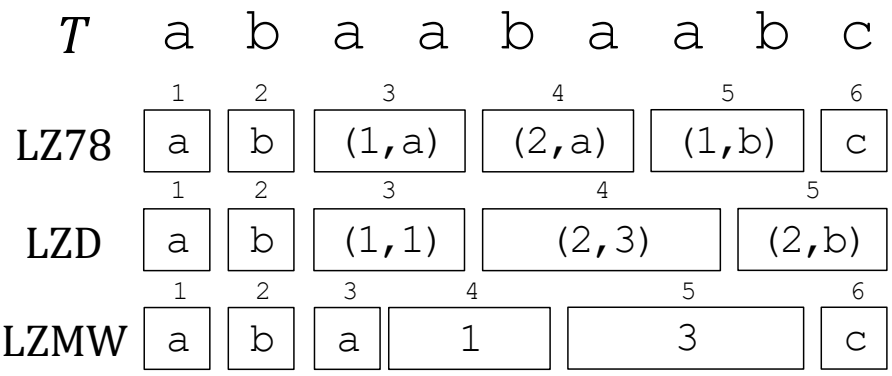

Fig. 1 The LZ78, LZD, and LZMW factorizations for $T = \texttt{abaabaabc}$. Each block corresponds to a factor, and the number above each block indicates the factor's index. Blocks labeled with a single character represent that character as a factor. In LZ78, a factor F_i represented by a tuple (j, c) means $F_i = F_j c$. In LZD, a factor F_i represented by a tuple (j, k) means $F_i = F_j F_k$. In LZMW, a factor F_i represented by a single integer j means $F_i = F_j F_{j+1}$.

Reading the suffix numbers stored in the leaves of ST in leaf-rank order gives the suffix array [22]. We denote the suffix array of T by SA. The array ISA is defined such that $\mathsf{ISA}[\mathsf{SA}[i]] = i$ for every $i = 1, \ldots, n$. We also denote the *LCP array* LCP of T by $\mathsf{LCP}[i] = |\mathsf{lcp}(T[\mathsf{SA}[i]], T[\mathsf{SA}[i+1]])|$. Since the ST leaves are sorted in SA order, the value $\mathsf{LCP}[i]$ is the string depth of the lowest common ancestor of the i-th leaf and $(i+1)$-st leaf in leaf-rank order. An ST node v can be uniquely represented by an SA range $[i..j]$ such that the k-th leaf is in the subtree of v for all $k \in [i..j]$.

Centroid Path Decomposition.

The centroid path decomposition [23] of a tree is defined as follows. For each internal node, we call its child whose subtree is the largest among all its siblings (ties are broken arbitrarily if there are multiple such children) a *heavy* node, while we call all other children *light* nodes. Additionally, we make root a light node. A *heavy path* is a maximal-length path from a light node u to the parent of a leaf containing, except for u, only heavy nodes. Since heavy paths do not overlap, we can contract all heavy paths to single nodes and thus form the centroid-path decomposed tree whose nodes are the heavy paths that are connected by the light edges. The centroid-path decomposed tree is helpful because the number of light nodes on a path from root to a leaf is $\mathcal{O}(\lg n)$, which means that a path from root to a leaf contains only $\mathcal{O}(\lg n)$ nodes. This can be seen from the fact that the subtree size of a light node is at most half of the subtree size of its parent; thus when visiting a light node during a top-down traversal in the tree, at least half the number of nodes we can visit from then on. Consequently, a root-leaf path in a centroid-path decomposed tree has $\mathcal{O}(\lg n)$ nodes.

CDAWG.

In what follows, we adapt LZ78-substring-compression techniques to work with the CDAWG instead of ST. The CDAWG of T, denoted by CDAWG, is the minimal compact automaton that recognizes all suffixes of T [11, 24]. The CDAWG of T is the minimization of ST, in which (a) all leaves are merged to a single node, called sink, and (b) all nodes, except sink, are in one-to-one correspondence with the maximal repeats of T [25], where a maximal repeat S is a substring of T having two occurrences $a_1 S b_1$ and $a_2 S b_2$ in T with $a_1 \neq a_2$ and $b_1 \neq b_2$. When transforming ST into CDAWG, multiple ST nodes can collapse into a single CDAWG node, and we say that such a CDAWG node *corresponds* to these collapsed ST nodes. We denote the number of CDAWG edges by e. With $\bar{e}$, we denote the number of edges of the CDAWG of the reverse of T. The number of CDAWG edges e can be regarded as a compression measure. For highly repetitive text, e can become asymptotically smaller than the text length n. In general, we can bound e with $e \in \mathcal{O}(n)$ and $e \in \Omega(\lg n)$. The upper bound is obtained from the fact that the number of ST edges is at most $2n - 1$; the lower bound is obtained from the fact that $g \in \mathcal{O}(e)$ and $g \in \Omega(\lg n)$, where g is the size of smallest grammar that produces T [15, Lemma 1]. Furthermore, there is a string family that achieves $e \in \Theta(\lg n)$ [26]. The middle of Fig. 2 gives an example of CDAWG.

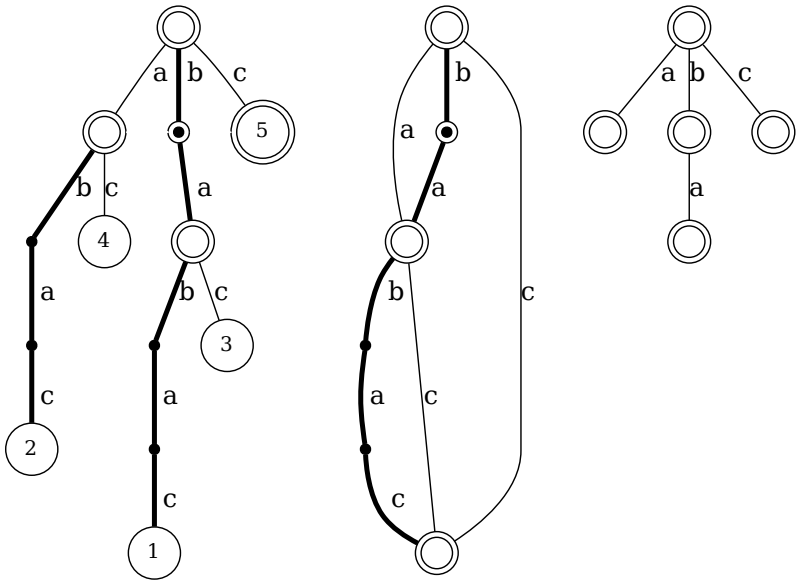

Fig. 2 ST (left), CDAWG (center), and the LZ trie (right) for $T = \texttt{babac}$. The LZ78 factorization of T is $F_0, F_1, \ldots, F_4 = \epsilon, \texttt{b}, \texttt{a}, \texttt{ba}, \texttt{c}$. We superimpose the suffix trie and the DAWG on ST and CDAWG, respectively, by drawing implicit nodes with black dots on the edges. We additionally encircle vertices corresponding to LZ78 factors (thus showing explicit nodes as double circles). Bold and thin lines represent, respectively, the heavy and light edges of the centroid path decomposition. The CDAWG sink represents the set of strings $\{\texttt{c}, \texttt{ac}, \texttt{bac}, \texttt{abac}, \texttt{babac}\}$, which can be read on the root-sink paths. Only a part of these strings are LZ78 factors.

RLBWT.

We also adapt the algorithm to work in RLBWT-compressed space. The *Burrows–Wheeler transform (BWT)* [27] of T is the concatenation of the last characters of all rotations of T sorted in lexicographic order. Since a BWT puts together all the suffixes starting with the same context, it typically has many runs. Therefore, the *run-length encoded Burrows–Wheeler transform (RLBWT)* [6], denoted by RLBWT, compresses the text effectively. It is known that $r \in \mathcal{O}(e)$ holds for all strings [28], and a simple string $\texttt{a}^{n-1}\$$ achieves $e \in \Theta(n)$ and $r \in \mathcal{O}(1)$. RLBWT can also be used for compressed indexes. There is a data structure that efficiently simulates typical suffix tree operations in $\mathcal{O}\left(r \lg \frac{n}{r}\right)$ space [29]. Note that the two compression measures e and $r \lg \frac{n}{r}$ are incomparable. Therefore, an RLBWT-based $\mathcal{O}\left(r \lg \frac{n}{r}\right)$ space index can be considered as an alternative to a CDAWG-based $\mathcal{O}(e)$ space index.

Substring Complexity.

The *substring complexity* [14] is a repetitiveness measure that can be efficiently computed. For a given text string T of length n, the substring complexity δ is defined as $\delta = \max\{d_k(T)/k \mid k \in [1, n]\}$, where $d_k(T)$ is the number of distinct substrings in T of length k. δ can be computed in linear time and is a lower bound for many repetitiveness measures such as the number of edges e of the CDAWG or the number of runs r of the BWT. A string of length n with substring complexity δ can be represented using $\mathcal{O}\left(\delta \lg \frac{n \lg \sigma}{\delta \lg n}\right)$ space [30]. Recently, a data structure has been proposed that supports various queries by providing random access on both SA and ISA and supporting longest common extension queries, all within the same space complexity [31].

3 Factorization Algorithm

The aim of this paper is to compute, after indexing the input text T in a preprocessing step, upon request for a provided interval $[i..j] \subseteq [1..n]$, the LZ78 factorization of $T[i..j]$, in compressed space with time bounded linearly in the output size and logarithmic in the text length. For that, we propose two algorithms, where the first one simulates ST with CDAWG, and thus directly applies the techniques of our pre-cursors working on ST. For the last algorithm, we show how to eliminate the need for the ST functionality to improve the time bounds.

3.1 Superimposing CDAWG

In the following, we show how to adapt the ST superimposition by the LZ trie from [8] to CDAWG. The main observation of [8, Sect. 3] is that the LZ trie is a connected subgraph of the suffix trie containing its root because LZ78 is prefix-closed. However, the compacted suffix trie, i.e., ST, does not contain all suffix trie nodes. In fact, the locus of each factor F_x is either an ST node v or lies on an ST edge g. In the former case, v corresponds to the LZ node v' representing F_x in the sense that both have the same string depth. In the latter case, the locus of F_x can be witnessed by the lower node the edge g connects to (by storing information on the length of F_x at that node). Thus, we can represent the LZ trie with a marking of ST nodes. The marking is done dynamically while computing the factorization as we mark the locus of each factor after having it processed. By marking the ST root node, we identify the LZ trie root with the ST root. To find the factor lengths, we perform a traversal from the leaf λ to its lowest marked ancestor, where λ is the leaf whose suffix number corresponds to the starting position of the factor we want to compute. Thus, we process the leaves in the order of their suffix numbers while computing the factorization.

To translate this technique to CDAWG, we no longer move to different leaves since all leaves are contracted to sink. This is no problem if we keep track of the starting position of the factor we want to compute. However, an obstacle is that a CDAWG node can have multiple parents. Given we superimpose the LZ trie on CDAWG such that an explicit LZ (trie) node v is stored in its corresponding CDAWG node v'. Unlike the case for ST, we generally have no information on the actual length of the string of v because v' can have multiple paths leading to root. Fig. 2 presents an example for which we cannot superimpose the LZ trie on CDAWG. In what follows, we propose two different solutions.

3.2 First Approach: Plug&Play Solution

Our solutions make the idea of the superimposition more implicit by modeling the LZ trie with a weighted segment tree data structure whose intervals correspond to ST nodes. In detail, we augment each LZ trie node with an SA range. For explicit LZ trie nodes having a corresponding ST node v, its SA range is the SA range of v. Otherwise, its range is the SA range of the ST node directly below. SA ranges of LZ trie nodes can be nested but do not overlap due to the tree topology of ST. This makes it feasible to model the lowest marked ancestor data structure used in the precursor algorithms with a weighted segment tree data structure that represents each LZ trie by its SA range and its LZ index as weight. For the example in Fig. 2, we end by storing the weighted intervals $(0, [1..5]), (1, [3..4]), (2, [1..2]), (3, [3..4]), (4, [5..5])$, where the first components denote the weights (i.e., the factor indices). In particular, we can make use of the following data structure for a *stabbing-max query*, i.e., for a given query point q, to find the interval with the highest weight containing q in a set of weighted intervals.

Lemma 1 ([32])**.** *Given a set of z intervals in the range $[1..n]$ with weights in $[1..n]$, there exists a linear-space data structure that answers stabbing-max queries and supports the insertion of a weighted interval in $\mathcal{O}(\lg z)$ time.*

With the data structure of Lemma 1, it is already possible to compute the LZ78 factorization without constructing the LZ trie in $\mathcal{O}(z \lg z)$ time with ST. For that we maintain the intervals of all computed LZ factors in an instance Stab of this data structure such that we can identify the factor index by the returned interval. We additionally index F_0 with interval $[1..n]$ and weight 0 for determining non-referencing factors. By doing so, given we want to compute a factor

$F_x = T[\mathsf{dst}_x..\mathsf{dst}_x + |F_x| - 1]$, we can determine its reference y by querying Stab. If $y > 0$, then $F_x = F_y \cdot T[\mathsf{dst}_x + |F_y|]$. It is left to determine the interval of F_x, which we need to add to Stab. For that, we find the locus of F_x in the suffix tree, which can be done with a weighted level ancestor data structure in constant time [33, 34].

This approach can be directly rewritten for CDAWG. To this end, we make use of the $\mathcal{O}(e + \bar{e})$-words representation of [28] and [15], which represents an ST node v with $\mathcal{O}(\lg n)$ bits of information, namely: (a) v's corresponding CDAWG node, (b) the string length of v, and (c) v's SA range. Their representation supports the following ST operations: (a) $\mathsf{suffixlink}(v, i)$ returns the ST node after taking i suffix links starting from v, in $\mathcal{O}(\lg n)$ time; (b) $\mathsf{strAncestor}(v, d)$ returns the highest ancestor of an ST node v with string depth of at least d, in $\mathcal{O}(\lg n)$ time.

Furthermore, it is known that the number of runs r in the BWT is upper-bounded by e [28]. Hence, in $\mathcal{O}(e)$ space, we can store the run-length compressed FM-index (RLFM)-index [35]. Given $\mathsf{SA}[i]$, RLFM can recover $T[i-1]$ in $\mathcal{O}(\lg n)$ time. By storing RLFM-index in both directions, we can sequentially extract characters in $\mathcal{O}(\lg n)$ time, which we use to match the next factor in CDAWG — remembering that each ST node representation also stores the corresponding SA range.

Let us recall that for computing a factor $F_x = T[\mathsf{dst}_x..\mathsf{dst}_x + |F_x| - 1]$, the only thing left undone is to find its Stab interval. For that, we stipulate the invariant that when computing F_x, we have selected the SA leaf λ whose suffix number is dst_x. To ensure this invariant for F_{x+1}, we call $\mathsf{suffixlink}(\lambda, |F_x|)$ to obtain the needed SA leaf. Finally, we find the locus of F_x by $\mathsf{strAncestor}(\lambda, |F_x|)$. Since each ST node stores its SA range, we have all the information to add the interval of F_x to Stab, and we are done. The time complexity is dominated by the ST simulation of CDAWG.

Theorem 1. *For a text T of length n, there exists a data structure of size $\mathcal{O}(e + \bar{e})$, which can, given an interval $\mathcal{I} \subseteq [1..n]$, compute the LZ78 factorization of $T[\mathcal{I}]$ in $\mathcal{O}(z \lg n)$ time with $\mathcal{O}(z)$ extra space, where z is the number of computed factors.*

3.3 Second Approach: Climbing Up

In what follows, we show how to get rid of the dependency on the ST simulation, which costs us $\mathcal{O}(\lg n)$ time per query and makes it necessary to also store the CDAWG of the inverted text. Instead of simulating the ST leaf with suffix number dst_x for computing factor F_x, we select sink and search for a path to root of length $\ell := n - \mathsf{dst}_x + 1$. This also means that instead of the top-down traversals as in the previous subsection, we climb up CDAWG from sink. To this end, we use the centroid path decomposition and some definitions.

Centroid Path Decomposition.

By applying the centroid path decomposition on ST, we obtain a centroid-path decomposed tree whose nodes are the heavy paths of ST and its edges the remaining light ST edges. Each root-leaf path in the centroid-path decomposed tree has a length of $\mathcal{O}(\lg n)$. [15] observed that the CDAWG edges corresponding to the ST heavy edges form a spanning tree of CDAWG. We apply the centroid path decomposition to the spanning tree of heavy edges again. We denote the heavy edges obtained by the second centroid path decomposition as the heavy edges of the CDAWG, and all other edges of the CDAWG as the light edges. After the second centroid path decomposition, the heavy edges form a set of disjoint paths, and each root-sink path in CDAWG visits at most $2 \lg n \in \mathcal{O}(\lg n)$ light edges. Fig. 2 gives an example of the centroid path decomposition and the correspondence between ST and CDAWG.

To speed up the CDAWG traversal for the factorization computation, we want to skip heavy edges. For that, we accumulate the information about LZ nodes of all heavy nodes in a heavy path P and store this information directly in P so that we only need to query a heavy path instead of all its heavy nodes. A linear sink-root traversal in CDAWG thus visits $\mathcal{O}(\lg n)$ light nodes and heavy paths. We can perform this traversal efficiently with some preprocessing:

Node Lengths.

Let $\mathsf{len}(u)$ for a CDAWG node u denote the set of the string lengths of all root-u paths in CDAWG. Actually, the set $\mathsf{len}(u)$ is an interval. This can be seen as follows: if there are root-u paths with

labels X and Y for $X \in \Sigma^*$ and Y being a suffix of X, then any suffix Z of X longer than Y has the same occurrences as X and Y in T, implying that these occurrences all follow the same characters, and therefore we can also reach u from root by reading Z. As a consequence, we can represent $\mathsf{len}(u)$ in $\mathcal{O}(1)$ words by using both interval ends, and augment each CDAWG node u with $\mathsf{len}(u)$ without violating our space budget.

Node Distances.

For two CDAWG nodes u and v on the same heavy path, let $\mathsf{dist}(u, v)$ be their string depth distance, which is well-defined because either u is the parent of v or vice versa (otherwise they cannot belong to the same heavy path).

Upward Navigation.

Recall that our aim is, after determining a factor $F_x = T[\mathsf{dst}_x..\mathsf{dst}_x + |F_x| - 1]$ with Stab, to find its interval for indexing F_x with Stab. For that, we climb up CDAWG from sink and search a root-sink path P of length $\ell := n - \mathsf{dst}_x + 1$, which is the string depth of the ST leaf having suffix number dst_x. Such a path P is uniquely defined since the ST nodes collapsed to a CDAWG node have all distinct string depths. In particular, ST nodes with the same string depth cannot have isomorphic subtrees, and therefore no two root-v paths can share the same length (substituting v with non-root ST nodes).

For upward navigation, we augment each node v with a binary search tree B_v. For each parent u of v connected by a light edge (u, v), we store (u, v) with key $\min(\mathsf{len}(u)) + c(u, v)$ in B_v, where $c(u, v)$ is the number of characters on the edge (u, v). With B_v, we can find the last edge (u, v) of the root-v path P of string length ℓ in $\mathcal{O}(\lg e)$ time. After climbing up to u, the remaining prefix of P is a root-u path P' of string length $\ell - c(u, v)$.

Now, a CDAWG ancestor u of v in the same heavy path can be a node in P if and only if $\ell - \mathsf{dist}(u, v) \in \mathsf{len}(u)$. Finding the highest possible such ancestor can be done with an exponential search in $\mathcal{O}(\lg e)$ time. We end up with a CDAWG ancestor u of v in P that is connected to its parent node w in P via a light edge (or $u =$ root, and we terminate the traversal). We can find w with B_u, and recurse on w belonging to another heavy path closer to the root node. In total, we visit $\mathcal{O}(\min(\lg n, e)) = \mathcal{O}(\lg n)$ heavy paths and light nodes. On each heavy path or light node that we process, we spend $\mathcal{O}(\lg e)$ time. Thus, the total time per factor is $\mathcal{O}(\lg n \lg e)$.

Finding the SA Range.

Given we process factor F_x, we use the above procedure to find the ST locus of F_x represented by CDAWG. For that, we stop climbing when we reach the shortest path P with a string length of at least $|F_x|$. However, unlike the previous approach, we do not have the SA ranges at hand. To compute them, we perform the following pre-computation step: We let each CDAWG node store (a) the number of ST leaves in the subtree rooted at one of its collapsed ST nodes (this is well defined because all these collapsed ST nodes have the same tree topology) and (b) the number of ST leaves of its lexicographically preceding sibling nodes, which we call the *aggregated* CDAWG *value*. Additionally, each heavy path stores from bottom up the prefix-sums of the aggregated CDAWG values of the nodes such that we can get for the i-th node on a heavy path the number of all leaves of all lexicographically preceding siblings of the descendant nodes of the i-th node belonging to the same heavy path. This whole pre-preprocessing helps us find the SA range of F_x as follows: We use a counter c that accumulates the leftmost border of the SA range we want to compute. For that, we increment c when climbing up to a light node by its aggregated CDAWG value. Additionally, when we leave a heavy node, we use the prefix-sum stored in its respective heavy path to perform the computation in constant time per light node or heavy path. When we reach the CDAWG node v representing the locus of F_x, c gives us the left border of the SA range we want to compute. However, the length of this SA range is given by the subtree size stored in v. This concludes our algorithm.

Speeding Up by Interval-Biased Search Trees.

The above time can be improved from $\mathcal{O}(\lg n \lg e)$ to $\mathcal{O}(\lg n)$ by implementing (a) B_v and (b) the exponential search in each heavy path with *interval-biased search trees*.

Lemma 2 ([36, Lemma 3.1])**.** *Given a sequence of integers $\ell_1 \leq \cdots \leq \ell_m$ from a universe $[0..u]$, the interval-biased search tree is a data structure of $\mathcal{O}(m)$ space that can compute, for an integer p given a query time, the predecessor of p in $\mathcal{O}(\lg(u/x))$ time, where $x = \mathsf{successor}(p) - \mathsf{predecessor}(p)$ is the difference between the predecessor $\mathsf{predecessor}(p)$ and successor $\mathsf{successor}(p)$ of p in $\{\ell_1, \ldots, \ell_m\}$.*

We note that there are faster predecessor data structures with time related to the distance of the query element to the predecessor such as [37, 38], which however do not improve the total running time, which is dominated by the number of nodes $\mathcal{O}(\lg n)$ we visit.

For the former (a), denoting B_v as $B_\cdot$ for any node v, during a $\mathsf{sink\text{-}root}$ traversal, a query of $B_\cdot$ always leads us to a higher node v such that the next search in $B_\cdot$ is bounded by $\max(\mathsf{len}(v))$, and therefore the query times in Lemma 2 lead to a telescoping sum of $\mathcal{O}(\lg n)$ total time.

For the latter (i.e, (b) the heavy paths), we let each heavy path maintain an interval-biased search tree storing its CDAWG nodes. A node u is stored with the key $\mathsf{dist}(u, v')$, where v' is the deepest node in the heavy path. At query time, we have the desired path-length ℓ and $\mathsf{len}(u) = [\min(\mathsf{len}(u))..\max(\mathsf{len}(u))]$ available such that we can query for the highest node u_1 with $\mathsf{dist}(u_1, v) = \mathsf{dist}(u_1, v') - \mathsf{dist}(v, v') \leq \ell - \min(\mathsf{len}(u_1))$, i.e., $\mathsf{dist}(u_1, v') \leq \ell - \min(\mathsf{len}(u_1)) + \mathsf{dist}(v, v')$ and the highest node u_2 with $\mathsf{dist}(u_2, v') \geq \ell - \max(\mathsf{len}(u_2)) + \mathsf{dist}(v, v')$. Then the deepest node among u_1 and u_2 is the highest ancestor of v that is still in P and is a member of the same heavy edge. The time complexity forms like for (a) a similar telescoping sum if we add to each key $\mathsf{dist}(u, v')$ the maximum depth of a heavy path such that each heavy path visit shrinks the search domain to be upper bounded by the last obtained key.

Theorem 2. *For a text T of length n, there exists a data structure of size $\mathcal{O}(e)$, which can, given an interval $\mathcal{I} \subseteq [1..n]$, compute the LZ78 factorization of $T[\mathcal{I}]$ in $\mathcal{O}(z \lg n + z \lg z) \subseteq \mathcal{O}(z \lg n)$ time and $\mathcal{O}(z)$ extra space, where z is the number of computed factors.*

3.4 Replacing CDAWG by Other Data Structures

Not only the CDAWG-based index mentioned above, but other compressed indexes can also be used for substring compression. To make this possible, we revisit the algorithm described in Section 3.2 and identify the core operations it requires from the underlying data structure. To compute a new factor $F_x = T[\mathsf{dst}_x..\mathsf{dst}_x + |F_x| - 1]$, we first find the ST leaf λ that represents $T[\mathsf{dst}_x..]$. Note that λ is the lexicographically $\mathsf{ISA}[\mathsf{dst}_x]$-th smallest suffix of T. Second, we compute the lowest LZ78 node u on the root-λ path. We find u by actually performing a stabbing-max query on Stab with value $\mathsf{ISA}[\mathsf{dst}_x]$. Finally, we compute the SA range R corresponding to the locus of F_x and insert R into Stab. We observe that the SA range R computed by the above procedure is the same as the SA range corresponding to $T[\mathsf{dst}_x..\mathsf{dst}_x + |F_x| - 1]$. In general, our algorithm requires the following operations:

(a) compute $\mathsf{ISA}[i]$,
(b) access $T[i]$,
(c) compute the SA range corresponds to a substring $T[\ell..r]$ for $1 \leq \ell \leq r \leq n$, and
(d) perform a stabbing-max query or update on a dynamic set of intervals.

Therefore, any data structure supporting (a)-(c) can take over the role of suffix trees or CDAWGs for substring compression with our algorithm.

Here, we show that an RLBWT-based compressed index satisfies the above properties. There is a compressed data structure that supports typical suffix tree operations in $\mathcal{O}\left(r \lg \frac{n}{r}\right)$ space [29]. It supports

- random access on the text and on the inverse suffix array in $\mathcal{O}\left(\lg \frac{n}{r}\right)$ time, and
- computing $\mathsf{PSV}(x, d) = \max(\{0\} \cup \{1 \leq y < x \mid \mathsf{LCP}[y] < d\})$ and $\mathsf{NSV}(x, d) = \min(\{n\} \cup \{x \leq y < n \mid \mathsf{LCP}[y] < d\})$ in $\mathcal{O}\left(\frac{\lg n}{\lg \lg n} + \lg \frac{n}{r}\right)$ time.

It thus supports operations (a) and (b). We can support (c) by computing $[\mathsf{PSV}(\mathsf{ISA}[\ell], r-\ell+1), \mathsf{NSV}(\mathsf{ISA}[\ell], r-\ell+1)-1]$. Therefore, by replacing CDAWG with this data structure, we can obtain the following result.

Theorem 3. *For a text T of length n, there exists a data structure of size $\mathcal{O}\left(r \lg \frac{n}{r}\right)$, which can, given an interval $\mathcal{I} \subseteq [1..n]$, compute the LZ78 factorization of $T[\mathcal{I}]$ in $\mathcal{O}\left(z\left(\lg\lg \frac{r}{\lg n} + \lg \frac{n}{r} + \lg z\right)\right)$ time with $\mathcal{O}(z)$ extra space, where z is the number of computed factors and r is the number of BWT-runs of T.*

Furthermore, we also show that the *δ-index* [31] also satisfies the above requirements. The δ-index can be stored in $\mathcal{O}\left(\delta \lg \frac{n \lg \sigma}{\delta \lg n}\right)$ space, and supports random access for the text and ISA in $\mathcal{O}\left(\log^{4+\varepsilon} n\right)$ time, where $\varepsilon > 0$ is a given constant. Furthermore, it also allows computing the length of the longest common prefix between any two suffixes of the text in $\mathcal{O}(\lg n)$ time. Since it supports LCE operations, we can simulate $\mathsf{PSV}(x, d)$ and $\mathsf{NSV}(x, d)$ using the δ-index by combining binary search and LCE queries. Therefore, we can use it instead of CDAWG and the RLBWT-based compressed index for the LZ78 substring compression problem. With the δ-index, we obtain the following result.

Theorem 4. *For a text T of length n, there exists a data structure of size $\mathcal{O}\left(\delta \lg \frac{n \lg \sigma}{\delta \lg n}\right)$, which, given an interval $\mathcal{I} \subseteq [1..n]$, can compute the LZ78 factorization of $T[\mathcal{I}]$ in $\mathcal{O}\left(z \lg^{4+\varepsilon} n\right)$ time with $\mathcal{O}(z)$ extra space, where z is the number of factors computed, and δ is the substring complexity of T, and $\varepsilon > 0$ is a constant.*

Note that we focus solely on the theoretical result because there is no practical implementation of this data structure, and it is not designed with practical performance in mind.

3.5 Extension for LZMW and LZD

In this section, we show the compressed-space substring compression algorithms for LZMW and LZD factorization. The main difference between these factorizations and LZ78 lies in the structure of their corresponding LZ tries. Fig. 3 illustrates the LZ tries considered in this work. While the standard LZ78 trie contains only nodes that correspond to factors, the LZD and LZMW tries also include intermediate nodes that do not represent any factor. As a result, the total number of nodes in these tries may exceed the number of factors. Therefore, maintaining the LZ tries in a straightforward manner does not guarantee good space complexity. However, our approach avoids this issue by not maintaining the trie topology explicitly. Instead, we associate each factor with an interval in the suffix tree, which allows us to sidestep the overhead caused by the structural differences in the LZ tries.

LZD Factorization.

We first review the original substring compression algorithm for LZD proposed in [10]. Suppose we want to compute a new factor $F_i = G_1 G_2$. In the algorithm, the first half G_1 is computed using the LZ trie consisting of $\{F_0, \ldots, F_{i-1}\}$, which is superimposed on the suffix tree. The second half G_2 is then computed in the same manner as the first. Finally, a new node corresponding to the new factor F_i is inserted into the LZ trie.

Since the procedures for computing both G_1 and G_2 are essentially the same as in the LZ78 algorithm, we can directly apply the compressed-space algorithm developed for LZ78. In addition, the method for computing the new SA range described in Section 3.3 can also be reused. Therefore, by using compressed-space data structures for LZ78 substring compression, we obtain the same performance guarantees for LZD.

LZMW Factorization.

We next consider the LZMW substring compression problem. Unlike the LZ78 and LZD tries, the LZMW trie consists of all concatenations of two consecutive factors. In the LZMW substring compression algorithm, we superimpose the LZMW trie onto the suffix tree, in the same way as done for LZ78. Using this superimposed structure, we find the longest matching factor starting at position

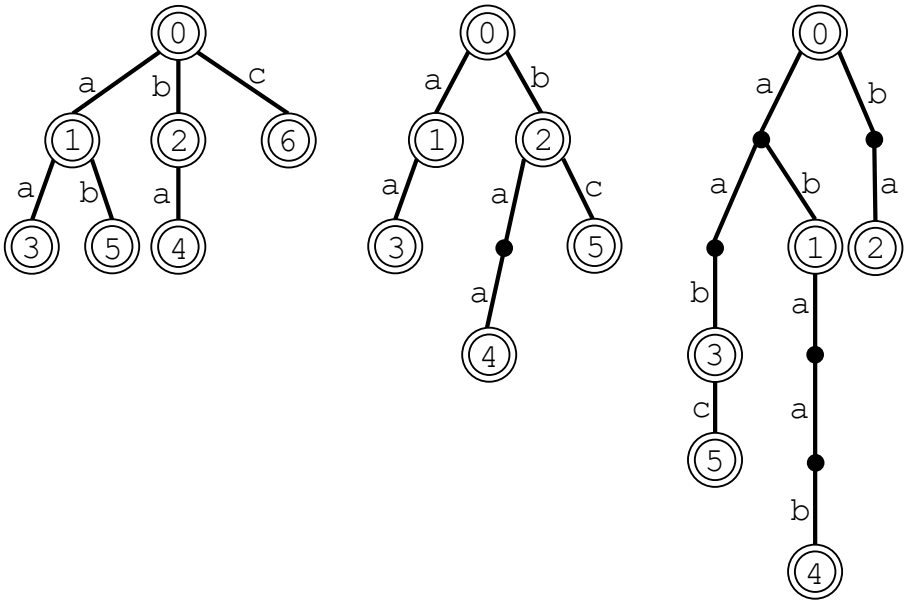

Fig. 3 The LZ78 (left), LZD (center), and LZMW (right) tries for $T = \texttt{abaabaabc}$, the same string as in Fig. 1. Vertices shown as double circles represent nodes corresponding to factors in each LZ trie, while black dots mark intermediate nodes that do not correspond to any factor. The integer inside each node indicates the index of the corresponding factor. In the LZ78 and LZD tries, a node labeled i represents the factor F_i. In the LZMW trie, a node labeled i represents the concatenation of two consecutive factors, namely the string F_iF_{i+1}, except for the special case when $i = 0$.

p in T by identifying the deepest mark corresponding to an LZMW factor along the path representing $T[p..n]$. Since this procedure is nearly identical to that of the LZ78 substring compression, our compressed-space method can be naturally extended to support LZMW factorization as well. Thus, we can compute the LZMW substring compression using the same data structures.

Therefore, using the methods of Theorem 2, Theorem 3, and Theorem 4, we can obtain these results.

Theorem 5. *For a text T of length n over an alphabet of size σ, there exist data structures that, given an interval $\mathcal{I} \subseteq [1..n]$, can compute the LZMW or LZD factorization of $T[\mathcal{I}]$ under the following space and time complexities:*

- *A structure of size $\mathcal{O}(e)$ takes $\mathcal{O}(z \lg n + z \lg z) = \mathcal{O}(z \lg n)$ time with $\mathcal{O}(z)$ extra space.*
- *A structure of size $\mathcal{O}\left(r \lg \frac{n}{r}\right)$ takes $\mathcal{O}\left(z \left(\lg \lg \frac{r}{\lg n} + \lg \frac{n}{r} + \lg z\right)\right) \subseteq \mathcal{O}(z \lg n)$ time with $\mathcal{O}(z)$ extra space.*
- *A structure of size $\mathcal{O}\left(\delta \lg \frac{n \lg \sigma}{\delta \lg n}\right)$ takes $\mathcal{O}\left(z \lg^{4+\varepsilon} n\right)$ time with $\mathcal{O}(z)$ extra space.*

Here, z is the number of computed factors, and r is the number of BWT-runs of T, δ is the substring complexity of T, and $\varepsilon > 0$ is a constant.

4 Experiments

In what follows, we empirically evaluate CDAWG-based and RLBWT-based indexes on real-world text strings for computing LZ78 substring compression. To do this, we first highlight the details of our implementation in Section 4.1. Subsequently, we describe our experimental settings in Section 4.2. Finally, we report the memory consumption, running time, and the distribution of the number of edges on paths between the CDAWG root and its sink in Section 4.3.

4.1 Deviation from Theory

We implement a simplification of our CDAWG-based index proposed in Section 3.3. In particular, we omit the centroid path decomposition because we empirically observed that the average number of edges on the root-sink paths is small in our datasets. We will discuss this observed phenomenon in detail in Section 4.3. To reduce complexity, we implemented the branches of each internal node, instead of an interval-biased search tree, by a sorted list on which we do a binary search to find the edge with the right label. We also deviate from theory in the implementation of the stabbing-max data structure, for which we use splay trees [39]. With a splay tree built on z intervals, the times to answer a query or add an interval are $\mathcal{O}(\lg z)$ amortized each, and the space is $\mathcal{O}(z)$ words. Therefore, replacing the original data structure with splay trees does not worsen the space of our index and keeps the time within $\mathcal{O}(z \lg z)$. Splay trees provide fast access to frequent elements by rearranging their structure adaptively on each query, and thus can exploit skewed distributions unlike common balanced trees such as AVL trees. The reason for using splay trees is that vertices of the splay tree are sequentially inserted at positions adjacent to the vertex that becomes the root of the splay tree by the previous query. By doing so, chances are high that a splay tree query only involves the very upper part of the tree, making the implementation practically fast in most cases.

For comparison, we also implement a simplification of the ST-based index. Our implementation differs from the method proposed in [4] in the following two points: (i) we omit the weighted-ancestor data structure, and, (ii) we use a stabbing-max data structure instead of a lowest marked ancestor data structure. The first change is because the average number of edges on the root-leaf paths of ST is small on our datasets, similar to the root-sink paths of CDAWG. The second change aims to reduce memory consumption. The stabbing-max data structure requires only $\mathcal{O}(z)$ space, whereas the lowest marked ancestor data structure requires $\mathcal{O}(n)$ space in addition to ST.

We also implement a RLBWT-based index. This implementation is almost the same as the method proposed in [40], except for the implementation of the predecessor structure used for computing $\mathsf{PSV}(x, d)$ and $\mathsf{NSV}(x, d)$. In the original method, they split an LCP array of length n into $\mathcal{O}(r)$ blocks and maintain each block in grammar-compressed form. Finally, they use a predecessor structure to determine the block containing the position x. Since each block has a tree structure of height $\mathcal{O}\left(\lg \frac{n}{r}\right)$ and the used predecessor structure needs $\mathcal{O}\left(\lg \lg \frac{r}{\lg n}\right)$ query time, the time complexity for computing PSV and NSV is $\mathcal{O}\left(\lg \lg \frac{r}{\lg n} + \lg \frac{n}{r}\right)$. However, the predecessor structure mentioned by the authors focus only on the theoretical result and is not going to be implemented. For simplification, in our implementation, we concatenate all blocks and construct a grammar tree of height $\mathcal{O}(\lg n)$. Therefore, our solution answers these queries in $\mathcal{O}(z \lg n)$ time.

Implementation Details.

We maintain the nodes and the edges of CDAWG separately in two arrays A_{V} and A_{E}. We store nodes in A_{V} in an arbitrary order, while we store edges in A_{E} in a sorted order based on two criteria. First, we partially sort the edges in groups sorted by the A_{V} index of the connecting child node. Second, for a fixed child node v, an edge (u, v) is sorted by the key $\min(\mathsf{len}(u)) + c(u, v)$ within its groups of edges sharing the same child node v. This arrangement makes it possible to perform binary search on the edge array for simulating the binary search trees B_v, which we here no longer need. Given a node v, to jump into the range $[\ell..r]$ of the edge array of edges connecting to v for querying B_v, we let v store ℓ. We can do so by letting node $A_{\mathrm{V}}[i]$ store (V1) the sum of the number of children over all preceding nodes (summing up the number of children of node $A_{\mathrm{V}}[j]$ for each $j \in [1..i-1]$). We then also know the right end of the interval $[\ell..r]$ by querying the subsequent node in the A_{V}. Additionally, each node stores (V2) $\max(\mathsf{len}(v))$ and (V3) the number of paths from v to the sink. An edge (u, v) from a node u to its child v is represented as a tuple of three integers: (E1) the index of u's entry in A_{V}, (E2) the string length of (u, v), and (E3) the prefix sum of u's aggregated CDAWG values (defined in Section 3.3). Therefore, both a node and an edge store three integers each. Following Section 3.3, we use these integers as follows: (E1) to select the parent node of v returned by B_v, (E2) to simulate a query on B_v via binary search with (V2), and to compute the string depth of the updated path when moving upward to the returned parent,

Table 2 Sizes and memory usage of CDAWG, ST and RLBWT of each dataset. Memory is measured in mebibytes (MiB). ST has approximately $2n$ vertices and edges regardless of the dataset.

	CDAWG size		RLBWT size		memory usage		
dataset	e	e/n	$r \lg \frac{n}{r}$	$r \lg \frac{n}{r}/n$	ST	CDAWG	RLBWT
SOURCES	66.33e6	0.494	65.74e6	0.489	3970.6	984.5	1764.9
DNA	178.91e6	1.333	56.74e6	0.422	4069.3	2830.2	1258.1
ENGLISH	102.21e6	0.761	71.21e6	0.531	3920.0	1518.6	1809.5
FIB	74	5.513e-7	453.6	3.38e-6	4736.0	1.28e-3	2.76e-2

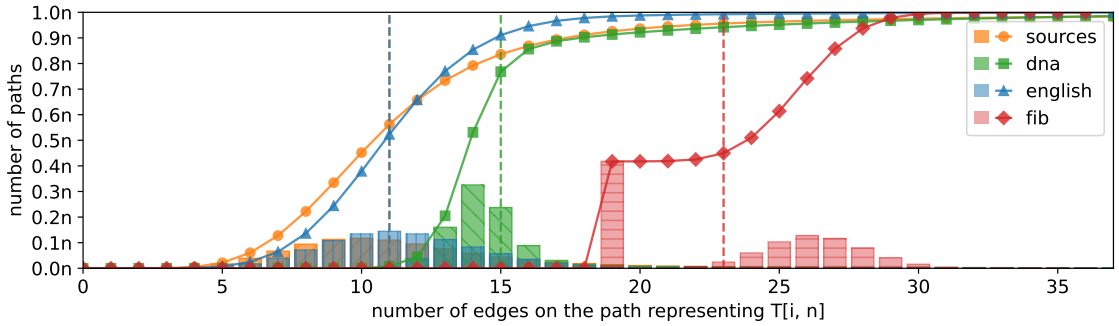

Fig. 4 The histogram of the number of edges on all paths between root and sink. The dashed vertical line for each dataset represents the average number of edges on all root-sink paths. The curve represents the cumulative sum of the histogram. Note that the number of all root-sink paths is $n = 134,217,728$ for all datasets.

and (E3) with (V3) to determine the SA range of the factor we want to compute. In addition, we store the first character of each edge label for each edge incident to root to provide efficient random access to T. To restore $T[i]$ from CDAWG, we first compute the path $(e_1, e_2, \ldots, e_k)$ representing $T[i..n]$. Then, we obtain $T[i]$ by taking the first character of the edge label of e_1. Storing these labels takes $\sigma \lg \sigma$ bits in total. Therefore, the overall memory consumption is $3p(|V| + |E|) + \sigma \lg \sigma$ bits, where $p \in \Omega(\lg n)$ is the size of an integer in bits.

Our suffix tree consists of three parts: an array of nodes, an array of pointers to all leaves sorted by their suffix numbers, and the raw input text. Each node v stores its string depth, the index of v's parent node in the node array, and v's SA range. With the node array and the pointers to the leaves, we can determine the SA range for Stab. We do not need SA because for computing the LZ78 substring compression, we only need to compute the SA range corresponding to an LZ78 factor, not the actual SA values. The total memory consumption is $4p|V| + n \lg \sigma$ bits.

Our index based on RLBWT consists of three data structures: a block tree to access $T[i]$, a block tree for accessing $\mathsf{ISA}[i]$, and a grammar-based index to compute $\mathsf{PSV}(x, d)$ and $\mathsf{NSV}(x, d)$. We omit the details of the implementation, but the overall data structure is almost the same as the structure of [40], except for the predecessor structure. Each of the tree data structures consumes $\mathcal{O}\left(rp \lg \frac{n}{r}\right)$ bits of space, where $p \in \Omega(\lg n)$ is the size of an integer in bits.

4.2 Experimental Settings

We have implemented our LZ78 substring compression algorithms in C++. The source code is available at https://github.com/shibh308/CDAWG-LZ78. For simplification, we assume that the input is interpreted in the byte alphabet ($\lg \sigma = 8$) and $n \leq 2^{32}$ (thus $p = 32$). Table 3 gives characteristics of the input texts used.

In one experiment instance, we construct the CDAWG, the ST, or the RLBWT-based index of an input text and answer some LZ78 substring compression queries. As input texts, we used SOURCES, DNA, and ENGLISH from the Pizza&Chilli Corpus [42], and the length-n prefix of the (infinite) Fibonacci string FIB. We note that the construction time of these indexes is $\mathcal{O}(n)$ in

Table 3 The alphabet size and repetitiveness measures on the first 128MiB of each dataset ($n = 134,217,728$). σ, e, r, z_{77} and z_{78} represent the alphabet size, the number of edges in the CDAWG, the number of runs of Burrows–Wheeler transform, and the number of factors of LZ77 and LZ78 factorization, respectively. Note that $e \in \Omega(\max\{r, z_{77}\})$ holds for any text [41]

dataset	σ	e	r	z_{77}	z_{78}
SOURCES	227	663278	31303555	7816156	13811755
DNA	16	178908741	84071820	9284690	10825116
ENGLISH	218	102211137	48367053	9639620	14219552
FIB	2	74	20	41	267813

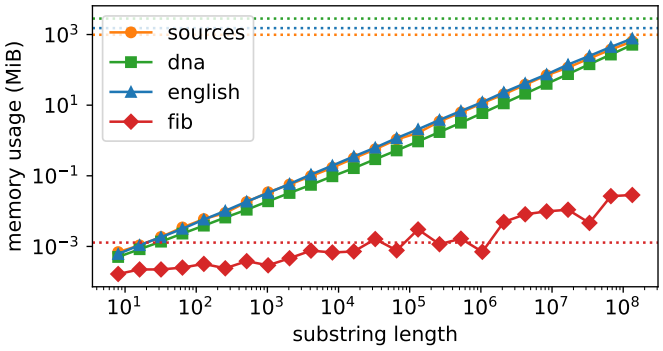

Fig. 5 Average memory usages of the stabbing-max data structure depicted by solid lines. The dotted line with the same respective color represents the memory usage of the CDAWG of the respective dataset.

theory. However, since our implementation is not optimized for construction speed, we omit experiments evaluating construction time. It is known that the CDAWG of the length-n prefix of the Fibonacci string has only $\mathcal{O}(\lg n)$ edges [26], and $r \in \mathcal{O}(\lg n)$ also holds from $r \in \mathcal{O}(e)$. We fixed $n = 2^{27} = 134,217,728$, and generated our input texts by extracting the first 2^{27} bytes (=128MiB) from each dataset from the text collection.

We compiled our source code with GCC 12.2.0 using the -O3 option, and ran all experiments on a machine with Debian 12, Intel(R) Xeon(R) Platinum 8481C processor, and 64GiB of memory.

We first construct the indexes for each text string and compute its memory consumption. We also measure the distribution of the number of edges on the root-sink paths of CDAWG. Note that we did not measure the construction time because we did not focus on efficient construction. After construction, we let them answer LZ78 substring compression queries. For each $\alpha \in \{2^3, 2^4, \ldots, 2^{27}\}$, we choose ten substrings of length α from the text uniformly at random and compute the LZ78 compression of these substrings. We calculate the average memory consumption of the stabbing-max data structure and the elapsed time excluding the maximum and minimum values.

4.3 Experimental Results

Table 2 indicates the size and memory consumption of each approach. For all datasets, CDAWG and RLBWT consume less memory than ST. Except for DNA, CDAWG consumes less space than RLBWT. However, it consumes more than twice the space of RLBWT in the case of DNA. Furthermore, CDAWG takes more space than the input itself because each edge and vertex consists of multiple integers. Even more severe, the number of CDAWG edges alone is higher than n for DNA. For FIB, both CDAWG and RLBWT compress the input text exponentially. Especially, the memory usage of CDAWG is about 10^5 and 3.7×10^6 times less than the raw text and ST, respectively.

Fig. 4 shows the distribution of the number of edges on all root-sink paths. We observe that the average number of edges on a root-sink path is about 10–15, and almost all paths have at most 20 edges in real-world datasets. Therefore, we can regard the number of paths as almost $O(\lg n)$. For plain CDAWGs without centroid path decomposition, path extraction can take $\mathcal{O}(n)$ time at worst. However, from this result, we empirically constitute that such cases are rare in practice, and both with and without centroid path decomposition, the running times are almost the same. Note

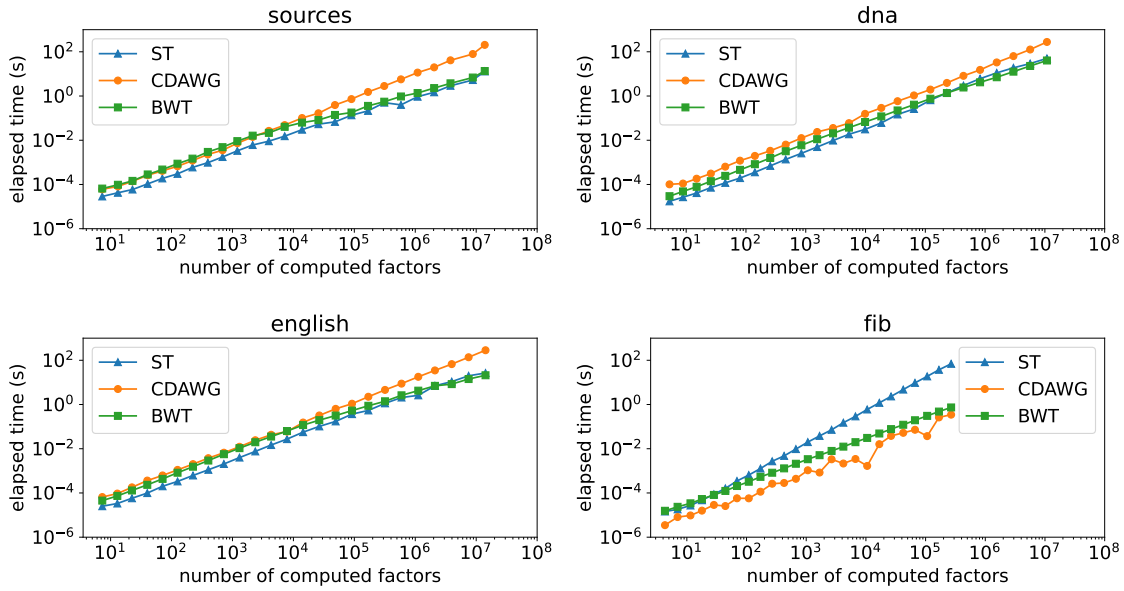

Fig. 6 Average elapsed time for LZ78 substring compression with ST, CDAWG, and RLBWT.

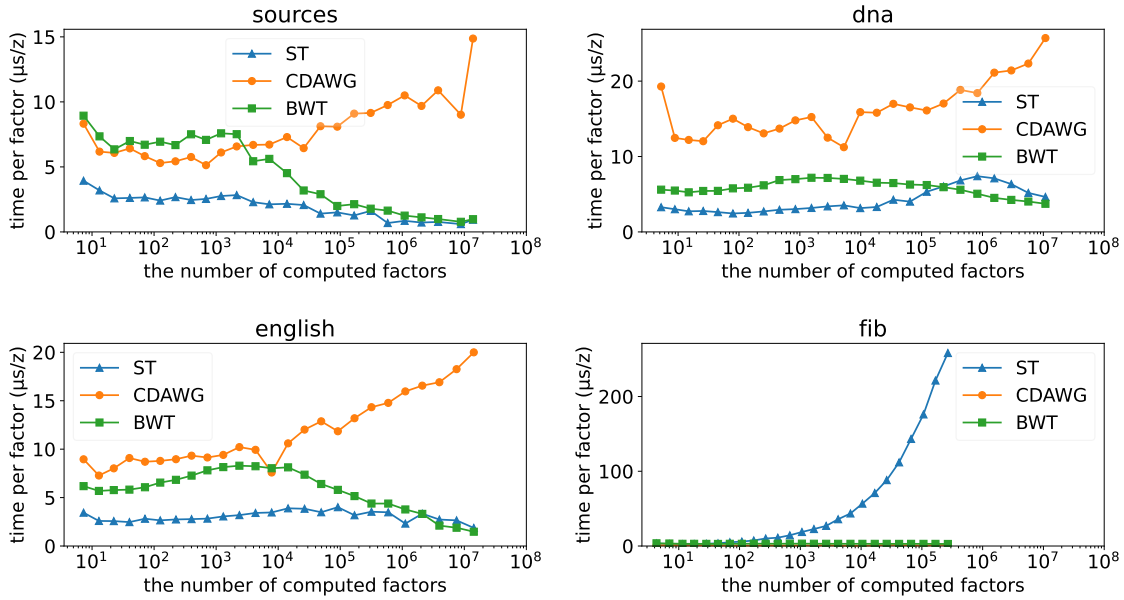

Fig. 7 Average elapsed time divided by the number of computed factors for LZ78 substring compression with ST, CDAWG, and RLBWT.

that the average number of edges on the root-sink paths of CDAWG is the same as the number of root-leaf paths of ST, so this fact also applies to ST.

Fig. 5 plots the memory usage of the stabbing-max data structures used. Memory consumption increases almost linearly with the increase in the length α of the queried substring. The memory usage of the stabbing-max data structure is about 50–80% lower than the memory consumption of CDAWG except for FIB even if $n = \alpha$ (i.e., the substring to compress in the whole text). Therefore, for real-world datasets, the main memory bottleneck is the CDAWG.

Fig. 6 shows the elapsed time for LZ78 substring compression using ST, CDAWG, and RLBWT. Fig. 7 shows the elapsed time divided by the number of factors computed. The elapsed time

increases almost linearly with the increase in the length of the queried substring. On real-world datasets, CDAWG is the slowest and ST is the fastest for most cases. The difference in computation time between CDAWG and ST is about 2-20 times. This is because the traversal of the ST is simpler than that of the others. In contrast, CDAWG and RLBWT computes the LZ78 substring compression about 5–300 times faster than ST in FIB, and CDAWG is the fastest in this case. We speculate that effective CDAWG-based compression has a positive impact on cache-friendly memory access.

5 Conclusion

We propose a technique that allows us to compute the substring compression of LZ78, LZD, and LZMW in compressed space. For that we introduce an abstract data type, whose implementations give various trade-offs, cf. Table 1. We conducted experiments on different types of data, and empirically evaluated that our method performs LZ78 substring compression efficiently with space improvements compared to methods based on ST. For that, we slightly deviated from the theory by omitting the centroid path decomposition and sophisticated data structures.

There are several directions for future work. Reducing the number of required operations for substring compression and exploring smaller indexes that support substring compression is an interesting line of research. For instance, grammar-based compressed indexes are theoretically smaller than CDAWG-based ones, and support not only random access but also pattern matching queries. However, they do not support suffix array queries, which are essential in our current method for LZ78 substring compression. Developing a new substring compression technique that avoids relying on suffix array queries could allow the use of grammar-based indexes, resulting in a more space-efficient solution.

Substring compression for dynamic texts is another promising direction. When the text is modified, the compressed index must be updated accordingly. A straightforward approach would be to use known compressed indexes that support edit operations [43, 44], but most of these incur high computational costs. However, in our case, we only require the support for a limited set of operations. Thus, by focusing on this restricted set, it may be possible to significantly speed up the update process.

Acknowledgments.

This work was supported by the JSPS KAKENHI Grant Numbers JP23H04378 and JP25K21150.

References

[1] Cormode, G., Muthukrishnan, S.: Substring compression problems. In: Proc. SODA, pp. 321–330 (2005)

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

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

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

[5] Köppl, D.: Computing LZ78-derivates with suffix trees. In: Bilgin, A., Fowler, J.E., Serra-Sagristà, J., Ye, Y., Storer, J.A. (eds.) Data Compression Conference, DCC 2024, Snowbird, UT, USA, March 19-22, 2024, pp. 133–142. IEEE, Snowbird, UT, USA (2024). https://doi.org/10.1109/DCC58796.2024.00021 . https://doi.org/10.1109/DCC58796.2024.00021

[6] Babenko, M.A., Gawrychowski, P., Kociumaka, T., Starikovskaya, T.: Wavelet trees meet suffix trees. In: Proc. SODA, pp. 572–591 (2015). https://doi.org/10.1137/1.9781611973730.39

[7] Kociumaka, T.: Efficient data structures for internal queries in texts. PhD thesis, University of Warsaw (2018)

[8] Nakashima, Y., I, T., Inenaga, S., Bannai, H., Takeda, M.: Constructing LZ78 tries and position heaps in linear time for large alphabets. Inf. Process. Lett. **115**(9), 655–659 (2015) https://doi.org/10.1016/j.ipl.2015.04.002

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

[10] Köppl, D.: Substring compression variations and LZ78-derivates. CoRR **abs/2409.14649** (2024) https://doi.org/10.48550/ARXIV.2409.14649 2409.14649

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

[12] Goto, K., Bannai, H., Inenaga, S., Takeda, M.: LZD factorization: Simple and practical online grammar compression with variable-to-fixed encoding. In: Proc. CPM. LNCS, vol. 9133, pp. 219–230 (2015). https://doi.org/10.1007/978-3-319-19929-0_19

[13] Miller, V.S., Wegman, M.N.: Variations on a theme by Ziv and Lempel. In: Combinatorial Algorithms on Words, Berlin, Heidelberg, pp. 131–140 (1985)

[14] Christiansen, A.R., Ettienne, M.B., Kociumaka, T., Navarro, G., Prezza, N.: Optimal-time dictionary-compressed indexes. ACM Trans. Algorithms **17**(1), 8–1839 (2021) https://doi.org/10.1145/3426473

[15] Belazzougui, D., Cunial, F.: Representing the suffix tree with the CDAWG. In: Proc. CPM. LIPIcs, vol. 78, pp. 7–1713 (2017). https://doi.org/10.4230/LIPIcs.CPM.2017.7

[16] Bannai, H., Gawrychowski, P., Inenaga, S., Takeda, M.: Converting SLP to LZ78 in almost linear time. In: Proc. CPM. LNCS, vol. 7922, pp. 38–49 (2013). https://doi.org/10.1007/978-3-642-38905-4_6

[17] Arimura, H., Inenaga, S., Kobayashi, Y., Nakashima, Y., Sue, M.: Optimally computing compressed indexing arrays based on the compact directed acyclic word graph. In: Proc. SPIRE. LNCS, vol. 14240, pp. 28–34 (2023). https://doi.org/10.1007/978-3-031-43980-3_3

[18] Badkobeh, G., Gagie, T., Inenaga, S., Kociumaka, T., Kosolobov, D., Puglisi, S.J.: On two LZ78-style grammars: Compression bounds and compressed-space computation. In: Proc. SPIRE. LNCS, vol. 10508, pp. 51–67 (2017). https://doi.org/10.1007/978-3-319-67428-5_5

[19] Shibata, H., Köppl, D.: LZ78 substring compression with CDAWGs. In: Proc. SPIRE. Lecture Notes in Computer Science, vol. 14899, pp. 289–305. Springer, Puerto Vallarta, Mexico (2024). https://doi.org/10.1007/978-3-031-72200-4_22

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

[21] Weiner, P.: Linear pattern matching algorithms. In: Proc. SWAT, pp. 1–11 (1973). https://doi.org/10.1109/SWAT.1973.13

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

[23] Ferragina, P., Grossi, R., Gupta, A., Shah, R., Vitter, J.S.: On searching compressed string collections cache-obliviously. In: Proc. PODS, pp. 181–190 (2008). https://doi.org/10.1145/1376916.1376943

[24] Crochemore, M., Vérin, R.: Direct construction of compact directed acyclic word graphs. In: Proc. CPM. LNCS, vol. 1264, pp. 116–129 (1997). https://doi.org/10.1007/3-540-63220-4_55

[25] Raffinot, M.: On maximal repeats in strings. Inf. Process. Lett. **80**(3), 165–169 (2001) https://doi.org/10.1016/S0020-0190(01)00152-1

[26] Rytter, W.: The structure of subword graphs and suffix trees of Fibonacci words. Theor. Comput. Sci. **363**(2), 211–223 (2006) https://doi.org/10.1016/j.tcs.2006.07.025

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

[28] Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., Raffinot, M.: Composite repetition-aware data structures. In: Proc. CPM. LNCS, vol. 9133, pp. 26–39 (2015). https://doi.org/10.1007/978-3-319-19929-0_3

[29] Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in BWT-runs bounded space. J. ACM **67**(1), 2–1254 (2020) https://doi.org/10.1145/3375890

[30] Kociumaka, T., Navarro, G., Prezza, N.: Toward a definitive compressibility measure for repetitive sequences. IEEE Trans. Inf. Theory **69**(4), 2074–2092 (2023) https://doi.org/10.1109/TIT.2022.3224382

[31] Kempa, D., Kociumaka, T.: Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In: Proc. FOCS, pp. 1877–1886 (2023). https://doi.org/10.1109/FOCS57990.2023.00114 . https://doi.org/10.1109/FOCS57990.2023.00114

[32] Yang, J., Widom, J.: Incremental computation and maintenance of temporal aggregates. VLDB J. **12**(3), 262–283 (2003) https://doi.org/10.1007/S00778-003-0107-Z

[33] Gawrychowski, P., Lewenstein, M., Nicholson, P.K.: Weighted ancestors in suffix trees. In: Proc. ESA. LNCS, vol. 8737, pp. 455–466 (2014). https://doi.org/10.1007/978-3-662-44777-2_38

[34] Belazzougui, D., Kosolobov, D., Puglisi, S.J., Raman, R.: Weighted ancestors in suffix trees revisited. In: Proc. CPM. LIPIcs, vol. 191, pp. 8–1815 (2021). https://doi.org/10.4230/LIPIcs.CPM.2021.8

[35] Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nord. J. Comput. **12**(1), 40–66 (2005)

[36] Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings and trees. SIAM J. Comput. **44**(3), 513–539 (2015) https://doi.org/10.1137/130936889

[37] Ehrhardt, M., Mulzer, W.: Delta-fast tries: Local searches in bounded universes with linear space. In: Proc. WADS, pp. 361–372 (2017). https://doi.org/10.1007/978-3-319-62127-2_31

[38] Belazzougui, D., Boldi, P., Vigna, S.: Predecessor search with distance-sensitive query time. ArXiv CoRR **abs/1209.5441** (2012) 1209.5441

[39] Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM **32**(3), 652–686 (1985) https://doi.org/10.1145/3828.3835

[40] Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in BWT-runs bounded space. In: Proc. SODA, pp. 1459–1477 (2018). https://doi.org/10.1137/1.9781611975031.96

[41] Navarro, G.: Indexing highly repetitive string collections, part I: repetitiveness measures. ACM Comput. Surv. **54**(2), 29–12931 (2021) https://doi.org/10.1145/3434399

[42] Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM Journal of Experimental Algorithmics **13**, 1–12111231 (2008) https://doi.org/10.1145/1412228.1455268

[43] Nishimoto, T., Tabei, Y.: Dynamic suffix array in optimal compressed space. CoRR **abs/2404.07510** (2024) https://doi.org/10.48550/ARXIV.2404.07510 2404.07510

[44] Nishimoto, T., Tabei, Y.: Dynamic r-index: An updatable self-index for highly repetitive strings. CoRR **abs/2504.19482** (2025) https://doi.org/10.48550/ARXIV.2504.19482 2504.19482