LZ78 Substring Compression with CDAWGs

Hiroki Shibata$^{1[0009-0006-6502-7476]}$ and Dominik Köppl$^{2[0000-0002-8721-4444]}$

^{1} Department of Informatics, Kyushu University, Japan
^{2} University of Yamanashi, Kofu, Japan

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`. While 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 in the substring compression model, where we are allowed to index the data and have to return the factorization of a substring specified at query time. In that model, we propose an algorithm that works in CDAWG-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 [10] is to preprocess a given input text T such that computing a compressed version of a substring of $T[i..j]$ can be done efficiently. This problem has been originally stated for the Lempel–Ziv-77 (LZ77) factorization [31], but extensions to the generalized LZ77 factorization [18], the Lempel–Ziv 78 (LZ78) factorization [21] as well as two of its derivates [22], the run-length encoded Burrows–Wheeler transform (RLBWT) [2], and the relative LZ factorization [20, Sect. 7.3] have been studied. Given n is the length of T, a trivial solution would be to precompute the compressed output of $T[\mathcal{I}]$ for all intervals $\mathcal{I} \subset [1..n]$. This however gives us already $\Omega(n^2)$ solutions to 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 of, the substring compression problem has not yet been studied with compressed space bounds that can be sub-linear 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 [25,15,21,22], which all use $\Omega(n)$ space for storing the suffix tree. Here, we make the algorithmic idea of the superimposition compatible with the compact directed acyclic word graph (CDAWG) [9], trading a tiny

Table 1. Solutions for computing the LZ78 substring compression for a substring with z LZ78 facors of a string of length n with e CDAWG edges. Extra space means the additional working space required for processing queries in addition to the index.

method	space		time
	index	extra space	query
naive	$\mathcal{O}(n^2)$	$\mathcal{O}(z)$	$\mathcal{O}(z)$
[21]	$\mathcal{O}(n)$	$\mathcal{O}(z)$	$\mathcal{O}(z)$
Theorem 2	$\mathcal{O}(e)$	$\mathcal{O}(z)$	$\mathcal{O}(z \lg n)$

time-penalty with a large space improvement for compressible texts. Table 1 gives an overview of known solutions for the problem we tackle.

Our contribution fits into the line of research focussed on data compression with the CDAWG. Given e and z are the number of CDAWG edges and the number of LZ78 factors, respectively, in that line, [5] proposed a straight-line program (SLP), which can be computed in $\mathcal{O}(e)$ time taking $\mathcal{O}(e)$ space. Given an SLP of size $\mathcal{O}(g)$, [3] 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, [1] showed how to compute, among others, the RLBWT and LZ77 in $\mathcal{O}(e)$ time and space.

2 Preliminaries

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

Let T be a text of length n whose characters are drawn from an integer alphabet $\Sigma = [1..\sigma]$ with $\sigma \leq n^{\mathcal{O}(1)}$. Given $X, Y, Z \in \Sigma^*$ with $T = XYZ$, then X, Y and Z are called a *prefix*, *substring* and *suffix* of T, respectively. We call $T[i..]$ the i-th suffix of T, and denote a substring $T[i] \cdots T[j]$ with $T[i..j]$. A *parsing dictionary* is a set of strings. A parsing dictionary $\mathcal{D}$ is called *prefix-closed* if it contains, for each string $S \in \mathcal{D}$, all prefixes of S as well. A *factorization* of T of size z partitions T into z substrings $F_1 \cdots F_z = T$. Each such substring F_x is called a *factor* 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* [33] of T iff, for all $x \in [1..z]$, the factor F_x is the longest prefix of $T[|F_1 \cdots F_{x-1}| + 1..]$ such that $F_x = F_y \cdot c$ for some $y \in [0..x-1]$ and $c \in \Sigma$, that is, F_x is the longest possible previous factor F_y appended by the following character in the text. The dictionary for computing F_x is $\mathcal{D}_x := \{F_y \cdot c : y \in [0..x-1], c \in \Sigma\}$, 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.

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* [32] 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. 1 gives an example of ST.

Reading the suffix numbers stored in the leaves of ST in leaf-rank order gives the suffix array [24]. We denote the suffix array of T by SA. Since the ST leaves are sorted in SA 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 [14] 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, we 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 [9,11]. The

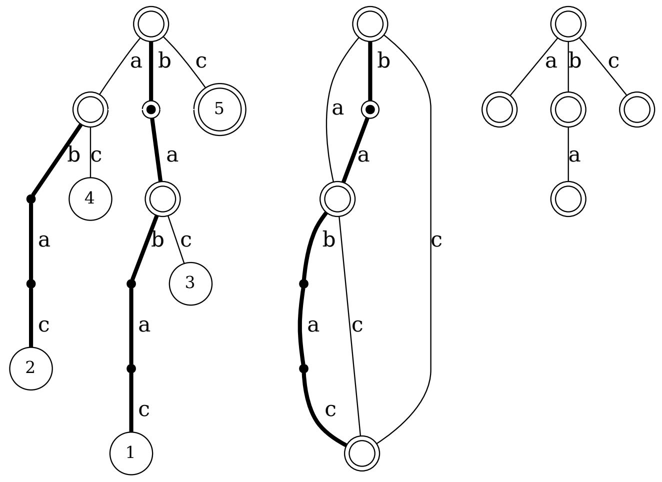

Fig. 1. ST (left), CDAWG (center), and the LZ trie (right) for $T :=$ babac. The LZ78 factorization of T is $F_0, F_1, \ldots, F_4 = \epsilon, \mathtt{b}, \mathtt{a}, \mathtt{ba}, \mathtt{c}$. We superimpose the suffix trie and the DAWG on ST and CDAWG, respectively, by drawing implicit nodes with black 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 $\{\mathtt{c}, \mathtt{ac}, \mathtt{bac}, \mathtt{abac}, \mathtt{babac}\}$, which can be read on the root-sink paths. Only a part of these strings are LZ78 factors.

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 [28], 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 [5, Lemma 1]. Furthermore, there is a string family that achieves $e \in \Theta(\lg n)$ [29]. The right of Fig. 1 gives an example of CDAWG.

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 $[i..j] \subset [1..n]$, 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 drop 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 [25] to CDAWG. The main observation of [25, Sect. 3] is that the LZ trie is a connected subgraph of the suffix trie containing its root because LZ78 is prefix-closed. The compacted suffix trie, i.e., ST, however contains not 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 about 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 have in general no information about the actual string length of v because v' can have multiple paths leading to root. Fig. 1 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 are not overlapping 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. 1, we end with 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 ([27]). *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 in $\mathcal{O}(\lg z/\lg\lg z)$ time. This data structure supports insertions and deletions of a weighted interval in $\mathcal{O}(\lg z)$ and $\mathcal{O}(\lg z/\lg\lg z)$ amortized time, respectively.*

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 [17,7].

This approach can be directly rewritten for CDAWG. To this end, we make use of the $\mathcal{O}(e + \bar{e})$-words representation of [6] and [5], 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.

Additionally, it is known that the number of runs r in the BWT is upper bounded by e [6]. Hence, in $\mathcal{O}(e)$ space, we can store the run-length compressed FM-index (RLFM)-index [23]. 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 for matching 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 information for adding the interval of F_x to Stab, and 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 $\mathcal{I} \subset [1..n]$, 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 Upwards

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 made 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 $\mathcal{O}(\lg n)$. [5] 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. 1 gives an example of the centroid path decomposition and the correspondence between ST and CDAWG.

To speed up the CDAWG traversal for the computation of the factorization, 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 such that we only require 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 by the fact that 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 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 $u = \mathsf{root}$, 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 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 the 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 accumulating 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 ([8, 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 [12,4], 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 $B.$ for any node v, during a sink-root traversal, a query of $B.$ always leads us to a higher node v such that the next search in $B.$ 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 $\mathcal{I} \subset [1..n]$, compute the LZ78 factorization of $T[\mathcal{I}]$ in $\mathcal{O}(z \lg n + z \lg z) \subset \mathcal{O}(z \lg n)$ time and $\mathcal{O}(z)$ extra space, where z is the number of computed factors.*

4 Experiments

In what follows, we empirically evaluate CDAWGs on real-world text strings for computing LZ78 substring compression. To this end, we first highlight details of our CDAWG 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 on 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 at the implementation of the stabbing-max data structure, for which we use splay trees [30]. With a splay tree built on z intervals, the times for answering a query or adding 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 rearrange 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 [21] 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 is aimed at reducing 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.

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} 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

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

	CDAWG size				memory usage	
dataset	$\|V\|$	$\|E\|$	$\|V\|/n$	$\|E\|/n$	CDAWG	ST
SOURCES	19.70e6	66.33e6	0.147	0.494	984.5	3970.6
DNA	68.39e6	178.91e6	0.510	1.333	2830.2	4069.3
ENGLISH	30.49e6	102.21e6	0.227	0.761	1518.6	3920.0
FIB	38	74	2.831e-7	5.513e-7	1.28e-3	4736.0

E: set of edges, V: set of nodes, $n = 2^{27}$

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 upwards to the returned parent, 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 overall memory consumption is $4p|V| + n \lg \sigma$ bits.

4.2 Experimental Settings

We have implemented our CDAWG- and ST-based 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 byte alphabet ($\lg \sigma = 8$) and $n \leq 2^{32}$ (thus $p = 32$). Table 3 gives characteristics of the used input texts.

In one experiment instance, we construct the CDAWG or the ST of an input text and answer some LZ78 substring compression queries. As input texts, we

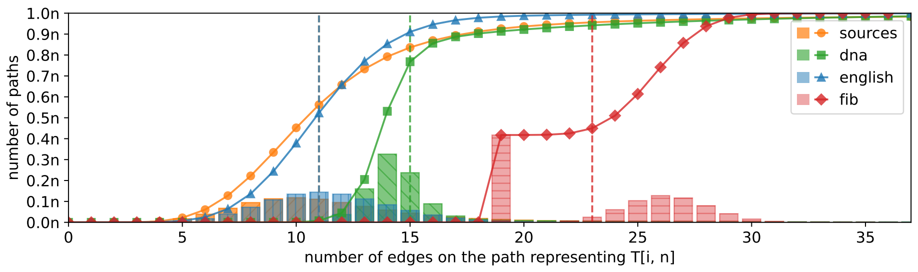

Fig. 2. The histogram of the number of edges on all paths between root and sink. The dashed vertical line for each dataset represents the average number of edges on all root-sink paths. The curve represents the cumulative sum of the histogram. Note that the number of all root-sink paths is $n = 134,217,728$ for all datasets.

Table 3. The alphabet size and repetitive measures on the first 128MiB of each dataset ($n = 134,217,728$). Columns σ, e, r, z_{77}, and z_{78} represent the alphabet size, the number of edges in the CDAWG, the number of phrases of run-length encoded 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 [26].

dataset	σ	e	r	z_{77}	z_{78}
SOURCES	227	$0.494n$	$0.233n$	$0.058n$	$0.102n$
DNA	16	$1.333n$	$0.626n$	$0.069n$	$0.080n$
ENGLISH	218	$0.761n$	$0.360n$	$0.072n$	$0.105n$
FIB	2	74	20	41	267813

used SOURCES, DNA, and ENGLISH from the Pizza&Chilli Corpus [13], and the length-n prefix of the (infinite) Fibonacci string FIB. It is known that the CDAWG of the length-n prefix of the Fibonacci string has only $O(\lg n)$ edges [29]. We fixed $n = 2^{27} = 134,217,728$, and generated our input texts by extracting the first 2^{27} bytes (=128MiB) from each dataset of 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 CDAWG and the ST 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 CDAWG and ST answer LZ78 substring compression queries. For each $\alpha \in \{2^3, 2^4, \dots, 2^{27}\}$, 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.

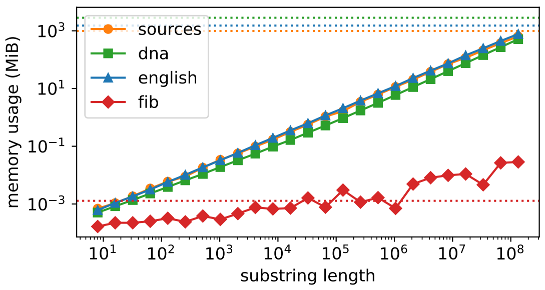

Fig. 3. Average memory usages of the stabbing-max data structure depicted by solid lines. The dotted line with the same respective color represents the memory usage of the CDAWG of the respective dataset.

4.3 Experimental Results

Table 2 indicates the size and memory consumption of the CDAWG and ST approaches. For all datasets, CDAWG consumes less memory than the ST. However, 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, CDAWG compresses the input text exponentially. The memory usage is about 10^5 and 3.7×10^6 times less than the raw text and ST, respectively.

Fig. 2 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 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. 3 plots the memory usage of the used stabbing-max data structures. The 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 major memory bottleneck is the CDAWG.

Fig. 4 shows the elapsed time for LZ78 substring compression using STs and CDAWGs, and Fig. 5 shows the elapsed time divided by the number of computed factors. The elapsed time increases almost linearly with the increase in substring length. In the real-world datasets, the CDAWG is about 2–20 times slower than the ST. This is because the traversal of the ST is simpler than that of the CDAWG. In contrast, CDAWG computes the LZ78 substring compression about 5–300 times faster than ST in FIB. We speculate that an effective CDAWG-based compression has a positive impact on cache-friendly memory access.

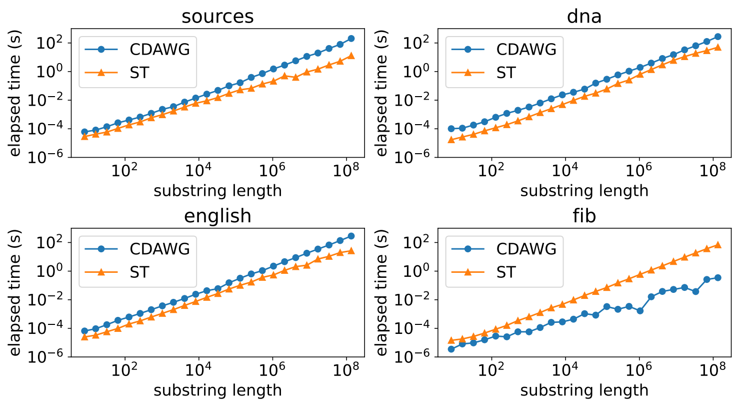

Fig. 4. Average elapsed time for LZ78 substring compression with CDAWG or ST.

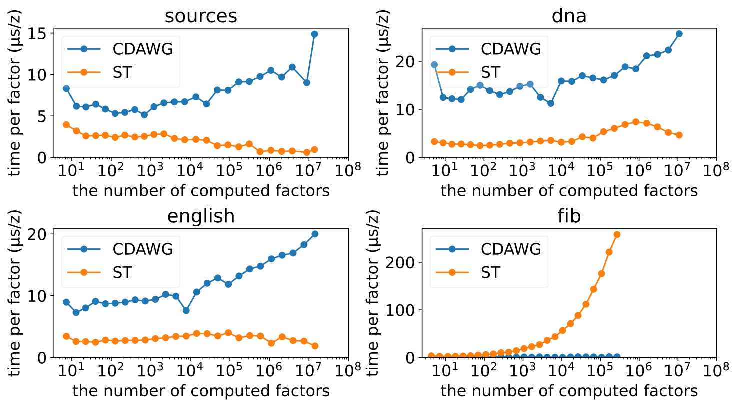

Fig. 5. Average elapsed time divided by the number of computed factors for LZ78 substring compression with CDAWG or ST.

5 Conclusion

We propose a method to compute the LZ78 substring compression in CDAWG-compressed space. Our method compresses substrings with z LZ78 factors in $\mathcal{O}(z \lg n)$ time and $\mathcal{O}(e)$ space. We conducted experiments on various types of data, and empirically evaluated that our method performs LZ78 substring compression efficiently with space improvements compared to ST-based methods. For that, we slightly deviated from theory by omitting the centroid path decomposition and sophisticated data structures. As future work, we plan to make our approach based on stabbing-max queries independent of the CDAWG, which is possible since we require only access to the input text, the suffix array, and its inverse. Elaborated compressed index data structures like [16,19] seem to support these access operations, which can take space asymptotically smaller than the CDAWG.

Acknowledgments. This work was supported by JSPS KAKENHI Grant Number JP23H04378.

References

1. H. Arimura, S. Inenaga, Y. Kobayashi, Y. Nakashima, and M. Sue. Optimally computing compressed indexing arrays based on the compact directed acyclic word graph. In *Proc. SPIRE*, volume 14240 of *LNCS*, pages 28–34, 2023.
2. M. A. Babenko, P. Gawrychowski, T. Kociumaka, and T. Starikovskaya. Wavelet trees meet suffix trees. In *Proc. SODA*, pages 572–591, 2015.
3. H. Bannai, P. Gawrychowski, S. Inenaga, and M. Takeda. Converting SLP to LZ78 in almost linear time. In *Proc. CPM*, volume 7922 of *LNCS*, pages 38–49, 2013.
4. D. Belazzougui, P. Boldi, and S. Vigna. Predecessor search with distance-sensitive query time. *ArXiv CoRR*, abs/1209.5441, 2012.
5. D. Belazzougui and F. Cunial. Representing the suffix tree with the CDAWG. In *Proc. CPM*, volume 78 of *LIPIcs*, pages 7:1–7:13, 2017.
6. D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. Composite repetition-aware data structures. In *Proc. CPM*, volume 9133 of *LNCS*, pages 26–39, 2015.
7. D. Belazzougui, D. Kosolobov, S. J. Puglisi, and R. Raman. Weighted ancestors in suffix trees revisited. In *Proc. CPM*, volume 191 of *LIPIcs*, pages 8:1–8:15, 2021.
8. P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.
9. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985.
10. G. Cormode and S. Muthukrishnan. Substring compression problems. In *Proc. SODA*, pages 321–330, 2005.
11. M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In *Proc. CPM*, volume 1264 of *LNCS*, pages 116–129, 1997.
12. M. Ehrhardt and W. Mulzer. Delta-fast tries: Local searches in bounded universes with linear space. In F. Ellen, A. Kolokolova, and J. Sack, editors, *Proc. WADS*, volume 10389 of *Lecture Notes in Computer Science*, pages 361–372. Springer, 2017.
13. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13:1.12:1 – 1.12:31, 2008.
14. P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter. On searching compressed string collections cache-obliviously. In *Proc. PODS*, pages 181–190, 2008.
15. J. Fischer, T. I, D. Köppl, and K. Sadakane. Lempel–Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018.
16. T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018.
17. P. Gawrychowski, M. Lewenstein, and P. K. Nicholson. Weighted ancestors in suffix trees. In *Proc. ESA*, volume 8737 of *LNCS*, pages 455–466, 2014.
18. O. Keller, T. Kopelowitz, S. L. Feibish, and M. Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:42–54, 2014.
19. D. Kempa and T. Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *Proc. FOCS*, pages 1877–1886, 2023.
20. T. Kociumaka. *Efficient Data Structures for Internal Queries in Texts.* PhD thesis, University of Warsaw, 2018.

21. D. Köppl. Non-overlapping LZ77 factorization and LZ78 substring compression queries with suffix trees. *Algorithms*, 14(2)(44):1–21, 2021.
22. D. Köppl. Computing LZ78-derivates with suffix trees. In *Proc. DCC*, pages 133–142, 2024.
23. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.
24. U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
25. Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda. Constructing LZ78 tries and position heaps in linear time for large alphabets. *Inf. Process. Lett.*, 115(9):655–659, 2015.
26. G. Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2021.
27. Y. Nekrich. A dynamic stabbing-max data structure with sub-logarithmic query time. In *Proc. ISAAC*, volume 7074 of *LNCS*, pages 170–179, 2011.
28. M. Raffinot. On maximal repeats in strings. *Inf. Process. Lett.*, 80(3):165–169, 2001.
29. W. Rytter. The structure of subword graphs and suffix trees of Fibonacci words. *Theor. Comput. Sci.*, 363(2):211–223, 2006.
30. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
31. J. A. Storer and T. G. Szymanski. The macro model for data compression (extended abstract). In *Proc. STOC*, pages 30–39, 1978.
32. P. Weiner. Linear pattern matching algorithms. In *Proc. SWAT*, pages 1–11, 1973.
33. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978.