Linear-size CDAWG: new repetition-aware indexing and grammar compression

Takuya Takagi\textsuperscript{1} Keisuke Goto\textsuperscript{2} Yuta Fujishige\textsuperscript{3}
Shunsuke Inenaga\textsuperscript{3} Hiroki Arimura\textsuperscript{1}

\textsuperscript{1} Graduate School of IST, Hokkaido University, Japan
\text{\{tkg,arim\}@ist.hokudai.ac.jp}
\textsuperscript{2} Fujitsu Laboratories Ltd., Japan
\text{goto.keisuke@jp.fujitsu.com}
\textsuperscript{3} Department of Informatics, Kyushu University, Japan
\text{\{yuta.fujishige,inenaga\}@inf.kyushu-u.ac.jp}

July 28, 2017

Abstract

In this paper, we propose a novel approach to combine compact directed acyclic word graphs (CDAWGs) and grammar-based compression. This leads us to an efficient self-index, called Linear-size CDAWGs (L-CDAWGs), which can be represented with $O(\tilde{e}_T \log n)$ bits of space allowing for $O(\log n)$-time random and $O(1)$-time sequential accesses to edge labels, and $O(m \log \sigma + \text{occ})$-time pattern matching. Here, $\tilde{e}_T$ is the number of all extensions of maximal repeats in $T$, $n$ and $m$ are respectively the lengths of the text $T$ and a given pattern, $\sigma$ is the alphabet size, and $\text{occ}$ is the number of occurrences of the pattern in $T$. The repetitiveness measure $\tilde{e}_T$ is known to be much smaller than the text length $n$ for highly repetitive text. For constant alphabets, our L-CDAWGs achieve $O(m + \text{occ})$ pattern matching time with $O(e_T \log n)$ bits of space, which improves the pattern matching time of Belazzougui et al.’s run-length BWT-CDAWGs by a factor of $\log \log n$, with the same space complexity. Here, $e_T$ is the number of right extensions of maximal repeats in $T$. As a byproduct, our result gives a way of constructing a straight-line program (SLP) of size $O(\tilde{e}_T)$ for a given text $T$ in $O(n + \tilde{e}_T \log \sigma)$ time.

1 Introduction

Background: Text indexing is a fundamental problem in theoretical computer science, where the task is to preprocess a given text so that subsequent pattern matching queries can be answered quickly. It has wide applications such as information retrieval, bioinformatics, and big data analytics \cite{10,14}. There have been a lot of recent research on compressed text indexes \cite{14,16} that store a text $T$ supporting \texttt{extract} and \texttt{find} operations in space significantly smaller than the total size $n$ of texts. Operation \texttt{extract} returns any substring $T[i..j]$ of the text. Operation \texttt{find} returns the list of all $\text{occ}$ occurrences of
a given pattern $P$ in $T$. For instance, Grossi, Gupta, and Vitter [9] gave a compressed text index based on compressed suffix arrays, which takes $s = nH_k + O(n \log \log n \log \sigma / \log n)$ bits of space and supporting $O(m \log \sigma + \text{polylog}(n))$ pattern match time, where $H_k$ is the $k$-th order entropy of $T$ and $m$ is the length of the pattern $P$.

**Compression measures for highly repetitive text:** Recently, there has been an increasing interest in indexed searches for highly repetitive text collections. Typically, the compression size of such a text can be described in terms of some measure of repetition. The followings are examples of such repetitiveness measures for $T$:

- the number $g_T$ of rules in a grammar (SLP) representing $T$,
- the number $z_T$ of phrases in the LZ77 parsing of $T$,
- the number $r_T$ of runs in the Burrows-Wheeler transform of $T$, and
- the number $\tilde{e}_T = e_T^r + e_T^l$ of right- and left-extensions of maximal repeats of $T$.

Belazzougui et al. [1] observed close relationship among these measures. Specifically, the authors empirically observed that all of them showed similar logarithmic growth behavior in $|T|$ on a real biological sequence, and also theoretically showed that both $z_T$ and $r_T$ are upper bounded by $\tilde{e}_T$. These repetitive texts are formed from many repeated fragments nearly identical. Therefore, one can expect that compressed index based on these measures such as $g_T$, $z_T$, $r_T$, and $\tilde{e}_T$ can effectively capture the redundancy inherent to these highly repetitive texts than conventional entropy-based compressed indexes [13].

**Repetition-aware indexes:** There has been extensive research on a family of repetition-aware indexes [3, 7, 10, 11] since the seminal work by Claude and Navarro [4]. They proposed the first compressed self-index based on grammars, which takes $s = g \log n + O(g \log g)$ bits supporting $O((m^2 + h(m + \text{occ})) \log g)$ pattern match time, where $g = g_T$ and $h$ are respectively the size and height of a grammar. Kreft and Navarro [10] gave the first compressed self-index based on LZ77, which takes $s = 3z \log n + 5n \log \sigma + O(z) + o(n)$ bits supporting $O(m^2 d + (m + \text{occ}) \log z)$ pattern match time. Here, $d$ is the height of the LZ parsing. Makinen, Navarro, Siren, and Valimaki [11] gave a compressed index based on RLBBT, which takes $s = r \log \sigma \log (2n/r)(1 + o(1)) + O(r \log \sigma \log \log (2n/r)) + O(\sigma \log n)$ bits supporting $O(m f(r \log \sigma, n \log \sigma))$ pattern match time, where $f(b, u)$ is the time for a binary searchable dictionary which is $O((\log b)^{0.5})$ and $o((\log \log u)^2)$ for example [11].

**Previous approaches:** Considering the above results, we notice that in compression ratio, all indexes above achieve good performance depending on the repetitive measures, while in terms of operation time, most of them except the RLBWT-based one [11] have quadratic dependency in pattern size $m$. Hence, a challenge here is to develop repetition-aware text indexes to achieve good compression ratio for highly repetitive texts in terms of repetition measures, while supporting faster extract and find operations. Belazzougui et al. [1] proposed a repetition-aware index which combines CDAGWs [3, 7] and the run-length encoded BWT [11], to which we refer as RLBWT-CDAGWs. For a given text $T$ of the length $n$ and a pattern $P$ of the length $m$, their index uses
\(O(e_T \log n)\) bits of space and supports \texttt{find} operation in \(O(m \log \log n + \text{occ})\) time.

**Main results:** In this paper, we propose a new repetition-aware index based on combination of CDAWGs and grammar-based compression, called the **Linear-size CDAWG** (L-CDAWG, for short). The L-CDAWG of a text \(T\) of length \(n\) is a self-index for \(T\) which can be stored in \(O(e_T \log n)\) bits of space, and support \(O(\log n)\)-time random access to the text, \(O(1)\)-time sequential character access from the beginning of each edge label, and \(O(m \log \sigma + \text{occ})\)-time pattern matching. For constant alphabets, our L-CDAWGs use \(O(e_T \log n)\) bits of space and support pattern matching in \(O(m + \text{occ})\) time, hence improving the pattern matching time of Belazzougui et al.’s RLBWT-CDAWGs by a factor of \(\log \log n\). We note that RLBWT-CDAWGs use hashing to retrieve the first character of a given edge label, and hence RLBWT-CDAWGs seem to require \(O(m \log \log n + \text{occ})\) time for pattern matching even for constant alphabets.

From the context of studies on suffix indices, our L-CDAWGs can be seen as a successor of the **linear-size suffix trie (LSTries)** by Crochemore et al. \cite{5}. The LSTrie is a variant of the suffix tree \cite{6}, which need not keep the original text by elegant scheme of linear time decoding using suffix links and a set of auxiliary nodes. However, it is a challenge to generalize their result for the CDAWG because the paths between a given pair of endpoints are not unique. By combining the idea of LSTries, an SLP-based compression with direct access \cite{2, 8}, we successfully devise a text index of \(O(e_T \log n)\) bits by improving functionalities of LSTries. As a byproduct, our result gives a way of constructing an SLP of size \(O(e_T \log e_T)\) bits of space for a text \(T\). Moreover, since the L-CDAWG of \(T\) retains the topology of the original CDAWG for \(T\), the L-CDAWG is a compact representation of all maximal repeats \cite{15} that appear in \(T\).

## 2 Preliminaries

In this section, we give some notations and definitions to be used in the following sections. In addition, we recall string data structures such as suffix tries, suffix trees, CDAWGs, linear-size suffix tries and straight-line programs, which are the data structures to be considered in this paper.

### 2.1 Basic definitions and notations

**Strings:** Let \(\Sigma\) be a general ordered alphabet of size \(\sigma \geq 2\). An element \(T = t_1 \cdots t_n\) of \(\Sigma^*\) is called a string, where \(|T| = n\) denotes its length. We denote the empty string by \(\varepsilon\) which is the string of length 0, namely, \(|\varepsilon| = 0\). Let \(\Sigma^* = \Sigma^* \setminus \{\varepsilon\}\). If \(T = XYZ\), then \(X\), \(Y\), and \(Z\) are called a **prefix**, a **substring**, and a **suffix** of \(T\), respectively. Let \(T = t_1 \cdots t_n \in \Sigma^n\) be any string of length \(n\). For any \(1 \leq i \leq j \leq n\), let \(T[i..j] = t_i \cdots t_j\) denote the substring of \(T\) that begins and ends at positions \(i\) and \(j\) in \(T\), and let \(T[i] = t_i\) denote the \(i\)th character of \(T\). For any string \(T\), we denote by \(\overline{T}\) the reversed string of \(T\), i.e., \(\overline{T} = T[n] \cdots T[1]\). Let \(\text{Suffix}(T)\) denote the set of suffixes of \(T\). For a string \(x\), the number of occurrences of \(x\) in \(T\) means the number of positions where \(x\) is a substring in \(T\).

**Maximal repeats and other measures of repetition:** A substring \(w\) of \(T\) is called a **repeat** if the number of occurrences of \(w\) in \(T\) more than one. A
right extension (resp. a left extension) of \( w \) of \( T \) is any substring of \( T \) with the form \( aw \) (resp. \( aw \)) for some letter \( a \in \Sigma \). A repeat \( w \) of \( T \) is a maximal repeat if both left- and right-extensions of \( w \) occur strictly fewer times in \( T \) than \( w \).

In what follows, we denote by \( \mu_T \), \( e_T \), \( e_T^* \), and \( \tilde{e}_T = e_T^* + e_T \) the numbers of maximal repeats, right-extensions, left-extensions, and all extensions of maximal repeats appearing in \( T \), respectively. Recently, it has been shown in [1] that the number \( \tilde{e}_T \) is an upper bound on the number \( \tau_T \) of runs in the Burrows-Wheeler transform for \( T \) and the number \( z_T \) of factors in the Lempel-Ziv parsing of \( T \).

It is also known that \( \tilde{e}_T \leq 4n - 4 \) and \( \mu_T < n \), where \( n = |T| \).

**Notations on graphical indexes:** All index structures dealt with in this paper, such as suffix tries, suffix trees, CDAWGs, linear-size suffix tries (LSTries), and linear-size CDAWGs (L-CDAWGs), are graphical indexes in the sense that an index is a pointer-based structure built on an underlying DAG \( G_L = (V(L), E(L)) \) with a root \( r \in V(L) \) and mapping \( \text{lab} : E(L) \to \Sigma^+ \) that assign a label \( \text{lab}(e) \) to each edge \( e \in E(L) \). For an edge \( e = (u,v) \in E(L) \), we denote its end points by \( e.\text{hi} := u \) and \( e.\text{lo} := v \), respectively. The label string of \( e \) is \( \text{lab}(e) \in \Sigma^+ \). The string length of \( e \) is \( \text{slen}(e) := |\text{lab}(e)| \geq 1 \).

An edge is called atomic if \( \text{slen}(e) = 1 \), and thus, \( \text{lab}(e) \in \Sigma \). For a path \( p = (e_1, \ldots, e_k) \) of length \( k \geq 1 \), we extend its end points, label string, and string length by \( p.\text{hi} := e_1.\text{hi}, p.\text{lo} := e_k.\text{lo}, \text{lab}(p) := \text{lab}(e_1) \ldots \text{lab}(e_k) \in \Sigma^* \), and \( \text{slen}(p) := \text{slen}(e_1) + \cdots + \text{slen}(e_k) \geq 1 \), respectively.

### 2.2 Suffix tries and suffix trees

The suffix trie [6] for a text \( T \) of length \( n \), denoted \( STrie(T) \), is a trie which represents \( \text{Suffix}(T) \). The size of \( STrie(T) \) is \( O(n^2) \). The path label of a node \( v \) is the string \( \text{str}(v) := \text{lab}(\pi_v) \) formed by concatenating the edge labels on the unique path \( \pi_v \) from the root to \( v \). If \( x = \text{str}(v) \), we denote \( v \) by \([x]\). We may identify \( v = [x] \) with its label \( x \) if it is clear from context. A substring \( x \) of \( T \) is said to be branching if there exists two distinct characters \( a, b \in \Sigma \) such that both \( xa \) and \( xb \) are substrings of \( T \). For any \( a \in \Sigma \), \( x \in \Sigma^* \), we define the suffix link of node \([ax]\) by \( \text{slink}([ax]) = [x] \) if \([ax]\) is defined.

The suffix tree [6,14] for a text \( T \), denoted \( STree(T) \), is a compacted trie which also represents \( \text{Suffix}(T) \). \( STree(T) \) can be obtained by compacting every path of \( STrie(T) \) which consists of non-branching internal nodes (see Fig. 1). Since every internal node of \( STree(T) \) is branching, and since there are at most
n leaves in \( STree(T) \), the numbers of edges and nodes are \( O(n) \). The edges of \( STree(T) \) are labeled by non-empty substrings of \( T \). By representing each edge label \( \alpha \) with a pair \((i, j)\) of integers such that \( T[i..j] = \alpha \), \( STree(T) \) can be stored in \( O(n \log n) \) bits of space.

### 2.3 CDAWGs

The \textit{compact directed acyclic word graph} for a text \( T \), denoted \( CDAWG(T) \), is the minimal compact automaton which represents \( Suffix(T) \). \( CDAWG(T) \) can be obtained from \( STree(T) \) by merging isomorphic subtrees and deleting associated endmarker \( \$$ \notin \Sigma \). Since \( CDAWG(T) \) is an edge-labeled DAG, we represent a directed edge from node \( u \) to \( v \) with label string \( x \in \Sigma^+ \) by a triple \( f = (u, x, v) \). For any node \( u \), the label strings of out-going edges from \( u \) start with mutually distinct characters.

Formally, \( CDAWG(T) \) is defined as follows. For any strings \( x, y \), we denote \( x \equiv_L y \) (resp. \( x \equiv_R y \)) iff the beginning positions (resp. ending positions) of \( x \) and \( y \) in \( T \) are equal. Let \([x]_L \) (resp. \([x]_R \)) denote the equivalence class of strings w.r.t. \( \equiv_L \) (resp. \( \equiv_R \)). All strings that are not substrings of \( T \) form a single equivalence class, and in the sequel we will consider only the substrings of \( T \). Let \( \overline{x} \) (resp. \( \overline{x}^R \)) denote the longest member of the equivalence class \([x]_L \) (resp. \([x]_R \)). Notice that each member of \([x]_L \) (resp. \([x]_R \)) is a prefix of \( \overline{x} \) (resp. a suffix of \( \overline{x}^R \)). Let \( \overline{x}_R = (\overline{x}) = (\overline{x}^R) \). We denote \( x \equiv y \) iff \( \overline{x}_R = \overline{y}_R \), and let \([x] \) denote the equivalence class w.r.t. \( \equiv \). The longest member of \([x] \) is \( \overline{x} \) and we will also denote it by \( value([x]) \). We define \( CDAWG(T) \) as an edge-labeled DAG \((V, E)\) such that \( V = \{[\overline{x}]_R \mid x \text{ is a substring of } T\} \) and \( E = \{([\overline{x}]_R, \alpha, [\overline{\alpha}]_R) \mid \alpha \in \Sigma^+, \overline{x} \neq \overline{\alpha}\} \). The \( \rightarrow \) operator corresponds to compacting non-branching edges (like conversion from \( STric(T) \) to \( STree(T) \)) and the \( [\cdot]_R \) operator corresponds to merging isomorphic subtrees of \( STree(T) \).

For simplicity, we abuse notation so that when we refer to a node of \( CDAWG(T) \) as \([x] \), this implies \( x = \overline{x} \) and \([x] = [\overline{x}]_R \).

Let \([x] \) be any node of \( CDAWG(T) \) and consider the suffixes of \( value([x]) \) which correspond to the suffix tree nodes that are merged when transformed into the CDAWG. We define the \textit{suffix link} of node \([x] \) by \( slink([x]) = [y] \), if \( y \) is the longest suffix of \( value([x]) \) that does not belong to \([x] \).

It is shown that all nodes of \( CDAWG(T) \) except the sink correspond to the maximal repeats of \( T \). Actually, \( value([x]) \) is a maximal repeat in \( T \) [13]. Following this fact, one can easily see that the numbers of edges of \( CDAWG(T) \) and \( CDAWG(T) \) coincide with the numbers \( e_T^R \) and \( e_T^L \) of right- and left-extensions of maximal repeats of \( T \), respectively [11][13].

By representing each edge label \( \alpha \) with pairs \((i, j)\) of integers such that \( T[i..j] = \alpha \), \( CDAWG(T) \) can be stored in \( O(e_T^R \log n + n \log \sigma) \) bits of space.

### 2.4 LSTrie

Recently, Crochemore \textit{et al.} [5] proposed a compact variant of a suffix trie, called \textit{linear-size suffix trie} (or LSTrie, for short), denoted \( LSTrie(T) \). It is a compacted tree with the topology and the size similar to \( STree(T) \), but has no indirect references to a text \( T \) (See Fig. 2). \( LSTrie(T) \) is obtained from \( STree(T) \) by adding all nodes \( v \) such that their suffix links \( slink(v) \) appear also in \( STree(T) \). Unlike \( STree(T) \), each edge \((u, v)\) of \( LSTrie(T) \) stores the first
character and the length of the corresponding suffix tree edge label (see Fig. 2).

Using auxiliary links called the \textit{jump pointers} the following theorem is proved.

**Proposition 1** (Crochemore et al. [5]). For a text $T$ of length $n$, the linear-size suffix trie $\text{LSTrie}(T)$ for $T$ can be stored in $O(n \log n)$ bits of space supporting reconstruction of the label of a given edge in $O(\ell)$ time, where $\ell$ is the length of the edge label.

Crochemore et al.'s method [5] does not regard the order of decoding characters on an edge label. This implies that $\text{LSTrie}(T)$ needs $O(\ell)$ worst case time to read any prefix of an edge label of length $\ell$. This may cause troubles in some applications including pattern matching. In particular, it does not seem straightforward to match a pattern $P$ against a prefix of the label of an edge $e$ in $O(|P|)$ time when $|P| < |\text{lab}(e)|$. We will solve these problems in Section 3 later.

### 2.5 Straight-line programs

A straight-line program (SLP) is a context-free grammar (CFG) in the Chomsky normal form generating a single string. SLPs are often used in grammar compression algorithms [14].

Consider an SLP $R$ with $n$ variables. Each production rule is either of form $X \rightarrow a$ with $a \in \Sigma$ or $X \rightarrow YZ$ without loops. Thus an SLP produces a single string. The \textit{phrase} of each $X_i$, denoted $F(X_i)$, is the string that $X_i$ produces.

The string defined by SLP $R$ is $F(X_n)$. We will use the following results.

**Proposition 2** (Gasieniec et al. [8]). For an SLP $R$ of size $g$ for a text of length $n$, there exist a data structure of $O(g \log n)$ bits of space which supports expansion of a prefix of $F(X_i)$ for any variable $X_i$ in $O(1)$ time per character, and can be constructed in $O(g)$ time.
Proposition 3 (Bille et al. [2]). For an SLP $R$ of size $g$ representing a text of length $n$, there exists a data structure of $O(g \log n)$ bits of space which supports to access consecutive $m$ characters at arbitrary position of $F(X_i)$ for any variable $X_i$ in $O(m + \log n)$ time, and can be constructed in $O(g)$ time.

3 The proposed data structure: L-CDAWG

In this section, we present the Linear-size CDAWG (L-CDAWG, for short). The L-CDAWG can support CDAWG operations in the same time complexity without holding the original input text and can reduce the space complexity from $O(e_T \log n + n \log \sigma)$ bits of space to $O(\tilde{e}_T \log n)$ bits of space, where $\tilde{e}_T = e_{lT} + e_{rT}$ is the number of extensions of maximal repeats. From now on, we assume that an input text $T$ terminates with a unique character $\$ which appears nowhere else in $T$.

3.1 Outline

The Linear-size CDAWG for a text $T$ of length $n$, denoted $L$-CDAWG$(T)$, is a DAG whose edges are labeled with single characters. $L$-CDAWG$(T)$ can be obtained from CDAWG$(T)$ by the following modifications. From now on, we refer to the original nodes appearing in CDAWG$(T)$ as type-1 nodes, which are always branching except the sink.

1. First, we add new non-branching nodes, called type-2 nodes to CDAWG$(T)$.
   Let $u = value([x])$ for any type-1 node $[x]$ of CDAWG$(T)$. If $au$ is a substring of $T$ but the path spelling out $au$ ends in the middle of an edge, then we introduce a type-2 node $v$ representing $au$. We add the suffix link $u = slink(v)$ as well. Adding type-2 nodes splits an edge into shorter ones. Note that more than one type-2 nodes can be inserted into an edge of CDAWG$(T)$.

2. Let $(u, x, v)$ be any edge after all the type-2 nodes are inserted, where $x \in \Sigma^+$. We represent this edge by $e = (u, c, v)$ where $c$ is the first character $c = x[1] \in \Sigma$ of the original label. We also store the original label length $slen(e) = |x|$.

3. We will augment $L$-CDAWG$(T)$ with a set of SLP production rules whose nonterminals correspond to edges of $L$-CDAWG$(T)$. The definition and construction of this SLP will be described later in Section 3.3.

If non-branching type-2 nodes are ignored, then the topology of $L$-CDAWG$(T)$ is the same as that of CDAWG$(T)$. For ease of explanation, we denote by $lab(e)$ the original label of edge $e$. Namely, for any edge $e = (u, c, v)$, $lab(e) = x$ iff $(u, x, v)$ is the original edge for $e$.

The following lemma gives an upper bound of the numbers of nodes and edges in $L$-CDAWG$(T)$. Recall that $\mu_T$ is the number of maximal repeats in $T$, $e_{lT}$ and $e_{rT}$ are respectively the number of left- and right-extensions of maximal repeats in $T$, and $\tilde{e}_T = e_{lT} + e_{rT}$.

Lemma 1. For any string $T$, let $L$-CDAWG$(T) = (V, E)$, then $|V| = O(\mu_T + e_{lT})$ and $|E| = O(\tilde{e}_T)$.
Proof. Let $\text{CDAWG}(T) = (V_0, E_0)$ and $\text{CDAWG}(\overline{T}) = (\overline{V}_0, \overline{E}_0)$. It is known that $|V_0| = |\overline{V}_0| = e_T$, $|E_0| = e_T^2$ and $|\overline{E}_0| = e_T^3$ (see [8] and [15]). Let $V_1$ and $V_2$ be the set of type-1 and type-2 nodes in $L$-$\text{CDAWG}(T)$, respectively. Clearly, $V_1 \cap V_2 = \emptyset$, $V = V_1 \cup V_2$, and $V_1 = V_0$. Let $[x] \in V_1$ and $u = \text{value}(x)$. Note that $u$ is a maximal repeat of $T$. For any character $a \in \Sigma$ such that $au$ is a substring of $T$, clearly $au$ is a left-extension of $u$. By the definition of $L$-$\text{CDAWG}(T)$, it always has a (type-1 or type-2) node which corresponds to $au$. Hence $|V_2| \leq e_T^f$. This implies $|V| = |V_1| + |V_2| = O(\mu_T + e_T^f)$. Since each type-2 node is non-branching, clearly $|E| = O(e_T^f + e_T^r) = O(e_T)$. \[
\]

Corollary 4. For any string of $T$ over a constant alphabet, $|V| = O(\mu_T + e_T^f)$ and $|E| = O(e_T^f)$, where $L$-$\text{CDAWG}(T) = (V, E)$.

Proof. It clearly holds that $\mu_T \geq e_T^f / \sigma$ and $e_T^r \geq \mu_T$. Thus we have $e_T^r \leq \sigma e_T^f$. The corollary follows from Lemma 1 when $\sigma = O(1)$.

3.2 Constructing type-2 nodes and edge suffix links

Lemma 2. Given $\text{CDAWG}(T)$ for a text $T$, we can compute all type-2 nodes of $L$-$\text{CDAWG}(T)$ in $O(e_T \log \sigma)$ time.

Proof. We create a copy $G$ of $\text{CDAWG}(T)$. For each edge $(u, x, v)$ of $\text{CDAWG}(T)$, we compute node $u' = \text{slink}(u)$ and the path $Q$ that spells out $x$ from $u'$. The number of type-1 nodes in this path $Q$ is equal to the number of type-2 nodes that need to be inserted on edge $(u, x, v)$, and hence we insert these nodes to $G$. After the above operation is done for all edges, $G$ contains all type-2 nodes of $L$-$\text{CDAWG}(T)$. Since there always exists such a path $Q$, to find $Q$ it suffices to check the first characters of out-going edges. Hence we need only $O(\log \sigma)$ time for each node in $Q$. Overall, it takes $O(e_T \log \sigma)$ time. \[
\]

The above lemma also indicates the notion of the following edge suffix links in $L$-$\text{CDAWG}(T)$ which are virtual links, and will not be actually created in the construction.

Definition 1 (Edge suffix links). For any edge $e$ with $\text{slen}(e) \geq 2$, $e$-suf$(e) = (e_1, \ldots, e_k)$ is the path, namely a list of edges, from $e_1.hi = \text{slink}(e.hi)$ to $e_k.lo$ that can be reachable from $e_1.hi$ by scanning $\text{lab}(e)$.

Edge suffix links have the following properties.

Lemma 3. For any edge $e$ such that $\text{slen}(e) \geq 2$ and its edge suffix link $e$-suf$(e) = (e_1, \ldots, e_k)$, (1) both $e_1.hi$ and $e_k.lo$ are type-1 nodes, and (2) all nodes in the path $e_1.lo = e_2.hi, \ldots, e_k-1.lo = e_k.hi$ are type-2 nodes.

Proof. From the definition of edge suffix links, we have $e_1.hi = \text{slink}(e.hi)$ and the path from $e_1.hi$ to $e_k.lo$ spells out $\text{lab}(e)$. (1) By the definitions of type-2 nodes and edge suffix links, $e_1.hi$ is always of type-1. Hence it suffices to show that $e_k.lo$ is of type-1. There are two cases: (a) If $e.lo$ is a type-2 node, then by the definition of type-2 nodes, $e_k.lo$ must be the node pointed by $\text{slink}(e.lo)$. Therefore, $e_k.lo$ is a type-1 node. (b) If $e.lo$ is a type-1 node, then let $ax$ be the shortest string represented by $e.hi$ with $a \in \Sigma$ and $x \in \Sigma^*$. Then, string $x$-lab$(e)$ is spelled out by a path from the source to $e_1.hi, \ldots, e_k.lo$, where either $e_k.lo = e.lo$ or $e_k.lo = \text{slink}(e.lo)$. Since $e.lo$ is of type-1, $\text{slink}(e.lo)$ is also of type-1.
Proof. We give an SLP of size $3.3$ Construction of the SLP for L-CDAWG

Lemma $3$ says that the label of any edge $e = (u, c, v)$ with $slen(e) \geq 2$ can be represented by a path $p = (e_1, \ldots, e_k) = e \cdot suf(e)$. In addition, since the path $p$ includes type-1 nodes only at the end points and since type-2 nodes are non-

branching, $p$ is uniquely determined by a pair of $(slink(u), c)$. We can compute all edges $e_i \in p$ for $1 \leq i \leq k$ in $O(k + \log \sigma)$ per query, as follows. Firstly, we compute $p.hi = slink(u)$ and then select the out-going edge $e_1$ starting with the character $c$ in $O(\log \sigma)$ time. Next, we blindly scan the downward path from $e_1$ while the lower end of the current edge $e_i$ has type-2. This scanning terminates when we reach an edge $e_k$ such that $e_k.lo$ is of type-1.

3.3 Construction of the SLP for L-CDAWG

We give an SLP of size $O(\tilde{e}_T)$ which represents $T$ and all edge labels of $L = L\cdot CDAWG(T)$ based on the jump links.

Jumping from an edge to a path: First, we define jump links, by which we can jump from a given edge $e$ with $slen(e) \geq 2$ to the path consisting of at least two edges, and having the same string label. Although our jump link is based on that of LSTries [5], we need a new definition since a path in CDAWG($T$) (and hence in $L\cdot CDAWG(T)$) cannot be uniquely determined by a pair of nodes, unlike STree($T$) (or LSTrie($T$)).

Definition 2 (Jump links). For an edge $e$ with $slen(e) \geq 2$ and $e \cdot suf(e) = (e_1, \ldots, e_k)$, jump($e$) is recursively defined as follows:

1. jump($e$) := jump($e_1$) if $k = 1$ (thus $e \cdot suf(e) = (e_1)$), and
2. jump($e$) := $(e_1, \ldots, e_k)$ if $k \geq 2$.

Note that lab($e$) equals lab($e_1$) \cdots lab($e_k$) for jump($e$) = $(e_1, \ldots, e_k)$.

Lemma 4. For any edge $e$ with $slen(e) \geq 2$, jump($e$) consists of at least two edges.

Proof. Assume on the contrary that jump($e$) = $e'$ for some edge $e'$. This implies $slen(e') \geq 2$. By definition, $e'.hi$ is a proper suffix of $e.hi$, namely, there exists an integer $k \geq 1$ such that $slink''(e.hi) = e'.hi$. For any character $c$ which appears in $T$, there is a (type-1 or type-2) node which represents $c$ as a child of the source of L-CDAWG($T$). This implies that there is an out-going edge $e''$ of length 1 from the source representing the first character of $e.hi$. This contradicts that jump($e$) only contains a single edge $e'$ with $slen(e') \geq 2$.

Theorem 5. For a given L-CDAWG($T$), there is an algorithm that computes all jump links in $O(\tilde{e}_T \log \sigma)$ time.

Proof. We explain how to obtain jump($e$) for an edge $e$ with $slen(e) \geq 2$. For all edge $e$ with $slen(e) \geq 2$, we manage a pointer to the first edge $e'$ of jump($e$) by $P[e] = e'$. We initially set $P[e] = e$ for all $e$. For all nodes $e$ with $slen(e) \geq 2$, let $u$ be an outgoing edge of $slink(e.hi)$ with the same label character of $e$. We check whether $P[e] = e$ and, if so, we recursively compute $P[u]$, and then set $P[e] = P[u]$. In this way all $P[e]$ can be computed in $O(\tilde{e}_T \log \sigma)$ time in total,
where the log\(\sigma\) is needed for selecting the outgoing edge. From Lemma 3, since there does not exist branching edge on each jump link, \(\text{jump}(e)\) can be easily obtained from \(P[e]\) by traversing the path until encountered a type-1 node.

**An SLP for the L-CDAWG:** We build an SLP which represents all edge labels in \(L\text{-CDAWG}(T) = (V, E)\) based on jump links. For each edge \(e\), let \(X(e)\) denote the variable which generates the string label \(lab(e)\). Let \(E = \{e_1, \ldots, e_n\}\).

For any \(e_i \in E\) with \(slen(e_i) = 1\), we construct a production \(X(e_i) \rightarrow c\) where \(c \in \Sigma\) is the label. For any \(e_i \in E\) with \(slen(e_i) \geq 2\), let \(\text{jump}(e_i) = (e'_1, \ldots, e'_k)\).

We construct productions \(X(e_i) \rightarrow X(e'_1)Y_1, Y_1 \rightarrow X(e'_2)Y_2, \ldots, Y_{k-3} \rightarrow X(e'_{k-2})Y_{k-2}, \text{ and } Y_{k-2} \rightarrow X(e'_{k-1})X(e'_k)\). We call a production whose left-hand size is \(Y_i\) an intermediate production. It is clear that \(X(e_i)\) generates \(lab(e)\) and we introduced \(k-1\) productions. If there is another edge \(e_j\) (\(i \neq j\)) such that \(\text{jump}(e_j) = (e'_1, \ldots, e'_k)\), then we construct a new production \(X(e_j) \rightarrow X(e'_1)Y_1\) and reuse the other productions. Let \(p\) be the path that spells out the text \(T\).

We create productions which generates \(T\) using the same technique as above for this path \(p\). Overall, the total number of intermediate productions is linear in the number of type-2 nodes in \(L\text{-CDAWG}(T)\). Since there are \(O(|E|)\) non-intermediate productions, this SLP consists of \(O(\tilde{e}_T)\) productions.

Now, we have the main result of this subsection.

**Theorem 6.** For a given \(L\text{-CDAWG}(T)\), there is an algorithm that constructs an SLP which represents all edge labels in \(O(\tilde{e}_T \log \sigma)\) time.

**Proof.** By the above algorithm, if jump links are computed, we can obtain an SLP which represents all edge labels in \(O(\tilde{e}_T)\) time. From Theorem 3, we can compute all jump links in \(O(\tilde{e}_T \log \sigma)\) times. Overall, the total time of this algorithm is \(O(\tilde{e}_T \log \sigma)\).

Fig. 2 shows \(L\text{STrie}(T)\) and \(L\text{-CDAWG}(T)\) enhanced with the SLP for string \(T = abedbeda\).

We associate to each edge label the corresponding variable of the SLP. By applying algorithms of Gasieniec et al. \([8]\) (in Proposition 2) and Bille et al. \([2]\) (in Proposition 3), we can show the following theorems.

**Theorem 7.** For a text \(T\), \(L\text{-CDAWG}(T)\) can support pattern matching for a pattern \(P\) of length \(m\) in \(O(m \log \sigma + \text{occ})\) time.

**Proof.** From Proposition 2, any consecutive \(m\) characters from the beginning of an edge in \(L\text{-CDAWG}(T)\) can be sequentially read in \(O(m)\) time. \(CDAWG(T)\) can support pattern matching by traversing the path from the source with \(P\) in \(O(m \log \sigma + \text{occ})\) time \([3]\). Since \(L\text{-CDAWG}(T)\) contains the topology of \(CDAWG(T)\), it can also support pattern matching in \(O(m \log \sigma + \text{occ})\) time.

**Theorem 8.** For a text \(T\) of length \(n\), \(L\text{-CDAWG}(T)\) has an SLP that derives \(T\). In addition, we can read any substring \(T[i..i+m]\) can be read in \(O(m + \log n)\) time.

**Proof.** The text \(T\) of \(L\text{-CDAWG}(T)\) is represented by the longest path \(p\) from the source to the sink. Remembering \(p\) makes it possible to read any position of \(T\) by using the Proposition 3.
3.4 The main result

It is known that for a given string $T$ of length $n$ over an integer alphabet of size $n^{O(1)}$, CDAWG($T$) can be constructed in $O(n)$ time [12]. Combining this with the preceding discussions, we obtain the main result of this paper.

**Theorem 9.** For a text $T$ of length $n$, L-CDAWG($T$) supports pattern matching in $O(m \log \sigma + \text{occ})$ time for a given pattern of length $m$ and substring extraction in $O(m + \log n)$ time for any substring of length $m$, and can be stored in $O(\tilde{e}_T \log n)$ bits of space (or $O(\tilde{e}_T)$ words of space). If CDAWG($T$) is already constructed, then L-CDAWG($T$) can be constructed in $O(\tilde{e}_T \log \sigma)$ total time. If $T$ is given as input, then L-CDAWG($T$) can be constructed in $O(n + \tilde{e}_T \log \sigma)$ total time for integer alphabets of size $n^{O(1)}$. After L-CDAWG($T$) has been constructed, the input string $T$ can be discarded.

4 Conclusions and further work

In this paper, we presented a new repetition-aware data structure called Linear-size CDAWGs. L-CDAWG($T$) takes linear space in the number of the left- and right-extensions of the maximal repeats in $T$, which is known to be small for highly repetitive strings. The key idea is to introduce type-2 nodes following LS-Tries proposed by Crochemore et al. [5]. Using a small SLP induced from edge-suffix links that is enhanced with random access and prefix extraction data structures, our L-CDAWG($T$) supports efficient pattern matching and substring extraction. This SLP is repetition-aware, i.e., its size is linear in the number of left- and right-extensions of the maximal repeats in $T$. We also showed how to efficiently construct L-CDAWG($T$).

Our future work includes implementation of L-CDAWG($T$) and evaluation of its practical efficiency, when compared with previous compressed indexes for repetitive texts. An interesting open question is whether we can efficiently construct L-CDAWG($T$) in an on-line manner for growing text.

References

[1] D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. Composite repetition-aware data structures. In Combinatorial Pattern Matching, pages 26–39. Springer, 2015.

[2] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings and trees. SIAM Journal on Computing, 44(3):513–539, 2015.

[3] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. Journal of the ACM (JACM), 34(3):578–595, 1987.

[4] F. Claude and G. Navarro. Self-indexed grammar-based compression. Fundamenta Informaticae, 111(3):313–337, 2011.

[5] M. Crochemore, C. Epifanio, R. Grossi, and F. Mignosi. Linear-size suffix tries. Theoretical Computer Science, 638:171–178, 2016.
[6] M. Crochemore and W. Rytter. *Jewels of stringology: text algorithms*. World Scientific, 2003.

[7] M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In *Combinatorial Pattern Matching*, pages 116–129. Springer, 1997.

[8] L. Gasieniec, R. M. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Data Compression Conference*, page 458, 2005.

[9] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

[10] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[11] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[12] K. Narisawa, S. Inenaga, H. Bannai, and M. Takeda. Efficient computation of substring equivalence classes with suffix arrays. *Algorithmica*, 2016.

[13] G. Navarro. A self-index on block trees. *arXiv preprint arXiv:1606.06617*, 2016.

[14] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.

[15] M. Raffinot. On maximal repeats in strings. *Information Processing Letters*, 80(3):165 – 169, 2001.

[16] Y. Takabatake, Y. Tabei, and H. Sakamoto. Improved ESP-index: a practical self-index for highly repetitive texts. In *International Symposium on Experimental Algorithms*, pages 338–350. Springer, 2014.

[17] P. Weiner. Linear pattern-matching algorithms. In *IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.