Compact Directed Acyclic Word Graphs for a Sliding Window

SHUNSUKE INENAGA, *Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan.*
PRESTO, Japan Science and Technology Corporation (JST)
E-mail: s-ine@i.kyushu-u.ac.jp

AYUMI SHINOHARA, *Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan.*
PRESTO, Japan Science and Technology Corporation (JST)
E-mail: ayumi@i.kyushu-u.ac.jp

MASAYUKI TAKEDA, *Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan.*
PRESTO, Japan Science and Technology Corporation (JST)
E-mail: takeda@i.kyushu-u.ac.jp

SETSUO ARIKAWA, *Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan.*
E-mail: arikawa@i.kyushu-u.ac.jp

ABSTRACT: ***Suffix trees*** are a well-known and widely-studied data structure highly useful for string matching. The suffix tree of a string w can be constructed in $O(n)$ time and space, where n denotes the length of w. Larsson achieved an efficient algorithm to maintain suffix trees for a sliding window. It contributes to *prediction by partial matching* (*PPM*) style statistical data compression scheme. *Compact directed acyclic word graphs* (*CDAWGs*) are a more space-economical data structure for indexing strings. In this paper we propose a linear-time algorithm to maintain CDAWGs for a sliding window.

Keywords: On-line Text Compression, Linear-Time Algorithm, Sliding Window, Compact Directed Acyclic Word Graphs

1 Introduction

Due to rapid advance in information technology and global growth of computer networks, various data are available today. This benefit, on the other hand, can turn out to be a serious matter that we have to sacrifice a large amount of memory space for the storage of huge data. *Data compression* is the practical technique to save memory

space required to store the information we need. Since *string* is the most fundamental and basic form of data, string matching plays central tasks in many data compression schemes. A straightforward method for matching strings would be to construct an index structure for the full text to be compressed. It is, however, easy to imagine that indexing the whole string requires too much space for the storage. This fact implies that an index structure needs to be *dynamic* in order to be suitable for *processing part of the text*, so that we can treat huge text data with a limited amount of space.

Suffix trees are a highly efficient data structure, therefore being extensively used in various applications in string matching [19, 16, 1, 5, 8]. The suffix tree of a string w, denoted by *STree*(w), represents all factors of w and can be constructed in linear time and space. The on-line algorithm by Ukkonen [18] processes a given string w left to right, and at each j-th phase it maintains *STree*$(w[1:j])$, where $w[1:j]$ denotes the prefix of w of length j. Larsson [14] modified Ukkonen's algorithm so as to maintain *STree*$(w[i:j])$ with $0 \leq i \leq j \leq |w|$, for any factor of length $j - i + 1 = M$. The width M of indexed factors is called the *window size*. That is, Larsson presented an algorithm to maintain a suffix tree for a *sliding window* mechanism where the values of i and j are incremented.

Larsson addressed that an application of suffix trees for a sliding window is the *prediction by partial matching* (*PPM*) style statistical data compression model [4, 17]. PPM* [3] is an improvement that allows unbounded context length. PPM* employs a tree structure called the *context trie*, which supports indexes of the input string. The drawback of PPM* is, however, its too much computational resources in both time and space, which weakens its practical usefulness. In particular, the context trie occupies major part of the space requirement. Larsson's suffix tree for a sliding window offered a variant of PPM*, feasible in practice since its space requirement is bounded by the window size M and the running time is linear in the length of the input string w.

In this paper, we take another approach to reducing the space requirement in PPM*-style statistical compression. We propose an algorithm to maintain *compact directed acyclic word graphs* (*CDAWGs*) for a sliding window, which performs in linear time and space. CDAWGs require less space than suffix trees in both theory and practice [2, 6]. In our previous work [11], we presented an on-line algorithm that constructs *CDAWG*(w) in linear time and space. Moving the rightmost position of a sliding window can be accomplished by the algorithm. In case of a suffix tree, it is also rather straightforward to advance the leftmost position of a sliding window: basically we have only to remove the leaf node and its in-coming edge corresponding to the longest suffix. However, since a CDAWG is a graph, the matter is much more complex and technically difficult. Thus more detailed and precise discussions are necessary. In addition, we have to ensure that no edge labels refer to positions outside a sliding window. To guarantee it, Larsson utilized the technique of *credit issuing* first introduced in [7], which takes amortized constant time. We introduce an extended version of credit issuing that is modified to be suitable for treating CDAWGs.

A preliminary version of this article appeared in [12].

2 Compact Directed Acyclic Word Graphs

2.1 Definitions

Let Σ be an *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of string $w = xyz$, respectively. The sets of the prefixes, factors, and suffixes of a string w are denoted by ***Prefix***(w), ***Factor***(w), and ***Suffix***(w), respectively. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. The factor of a string w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i:j] = \varepsilon$ for $j < i$. Let $S \subseteq \Sigma^*$. The cardinality of S is denoted by $|S|$. For any string $x \in \Sigma^*$, $Sx^{-1} = \{u \mid ux \in S\}$ and $x^{-1}S = \{u \mid xu \in S\}$. We define equivalence relations $\equiv_w^L$ and $\equiv_w^R$ on Σ^* by

$$\begin{aligned} x \equiv_w^L y \quad &\Leftrightarrow \textit{Prefix}(w)x^{-1} = \textit{Prefix}(w)y^{-1}, \\ x \equiv_w^R y \quad &\Leftrightarrow x^{-1}\textit{Suffix}(w) = y^{-1}\textit{Suffix}(w), \end{aligned}$$

respectively. Let $[x]_w^L$ and $[x]_w^R$ denote the equivalence classes of a string $x \in \Sigma^*$ under $\equiv_w^L$ and $\equiv_w^R$, respectively. The longest elements in the equivalence classes $[x]_w^L$ and $[x]_w^R$ for $x \in \textit{Factor}(w)$ are called their *representatives* and denoted by $\overrightarrow{x}^w$ and $\overleftarrow{x}^w$, respectively. For any string $x \in \textit{Factor}(w)$, there uniquely exist strings $\alpha, \beta \in \Sigma^*$ such that $\overrightarrow{x}^w = x\alpha$ and $\overleftarrow{x}^w = \beta x$.

We now introduce a relation X_w over Σ^* such that

$$X_w = \{(x, xa) \mid x \in \textit{Factor}(w) \text{ and } a \in \Sigma \text{ is unique such that } xa \in \textit{Factor}(w)\},$$

and let $\equiv'^L_w$ be the equivalence closure of X_w, i.e., the smallest superset of X_w that is symmetric, reflexive, and transitive. It can be readily shown that $\equiv_w^L$ is a refinement of $\equiv'^L_w$, namely, every equivalence class under $\equiv'^L_w$ is a union of one or more equivalence classes in $\equiv_w^L$. For a string $x \in \textit{Factor}(w)$, let $\overrightarrow{\overrightarrow{x}}^w$ denote the longest string in the equivalence class to which x belongs under the equivalence relation $\equiv'^L_w$.

Note that $\overrightarrow{x}^w$ and $\overrightarrow{\overrightarrow{x}}^w$ are not always equal. For example, consider the case that $w = \texttt{abab}$ and $x = \texttt{ab}$, where $\overrightarrow{x}^w = \texttt{ab}$ but $\overrightarrow{\overrightarrow{x}}^w = \texttt{abab}$. More formally:

PROPOSITION 2.1 ([10])
Let $w \in \Sigma^*$. For any string $x \in \textit{Factor}(w)$, $\overrightarrow{x}^w$ is a prefix of $\overrightarrow{\overrightarrow{x}}^w$. If $\overrightarrow{x}^w \neq \overrightarrow{\overrightarrow{x}}^w$, then $\overrightarrow{x}^w \in \textit{Suffix}(w)$.

In the following, we define the suffix tree and the CDAWG of w, denoted by ***STree***(w) and ***CDAWG***(w), respectively. We define them as edge-labeled graphs (V, E) with $E \subseteq V \times \Sigma^+ \times V$ where the second component of each edge represents its label. We also give definitions of the *suffix links*, frequently used for time-efficient construction of the index structures [19, 16, 18, 2, 6, 11, 9].

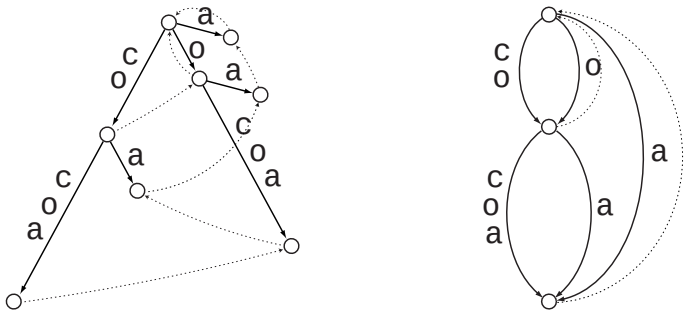

FIG. 1: The left and right figures are *STree*(cocoa) and *CDAWG*(cocoa), respectively. Solid arrows represent edges, and dotted arrows represent suffix links.

DEFINITION 2.2
STree(w) is the tree (V, E) such that

$$\begin{aligned} V &= \{\overset{w}{\Longrightarrow}\overline{x} \mid x \in \mathit{Factor}(w)\}, \\ E &= \{(\overset{w}{\Longrightarrow}\overline{x}, a\beta, \overset{w}{\Longrightarrow}\overline{xa}) \mid x, xa \in \mathit{Factor}(w), a \in \Sigma, \beta \in \Sigma^*, \overset{w}{\Longrightarrow}\overline{xa} = xa\beta, \overset{w}{\Longrightarrow}\overline{x} \neq \overset{w}{\Longrightarrow}\overline{xa}\}, \end{aligned}$$

and its suffix links are the set

$$F = \{(\overset{w}{\Longrightarrow}\overline{ax}, \overset{w}{\Longrightarrow}\overline{x}) \mid x, ax \in \mathit{Factor}(w),\ a \in \Sigma,\ \overset{w}{\Longrightarrow}\overline{ax} = a \cdot \overset{w}{\Longrightarrow}\overline{x}\}.$$

DEFINITION 2.3
CDAWG(w) is the dag (V, E) such that

$$\begin{aligned} V &= \{[\overset{w}{\Longrightarrow}\overline{x}]^R_w \mid x \in \mathit{Factor}(w)\}, \\ E &= \{([\overset{w}{\Longrightarrow}\overline{x}]^R_w, a\beta, [\overset{w}{\Longrightarrow}\overline{xa}]^R_w) \mid x, xa \in \mathit{Factor}(w), a \in \Sigma, \beta \in \Sigma^*, \overset{w}{\Longrightarrow}\overline{xa} = xa\beta, \overset{w}{\Longrightarrow}\overline{x} \neq \overset{w}{\Longrightarrow}\overline{xa}\}, \end{aligned}$$

and its suffix links are the set

$$F = \{([\overset{w}{\Longrightarrow}\overline{ax}]^R_w, [\overset{w}{\Longrightarrow}\overline{x}]^R_w) \mid x, ax \in \mathit{Factor}(w),\ a \in \Sigma,\ \overset{w}{\Longrightarrow}\overline{ax} = a \cdot \overset{w}{\Longrightarrow}\overline{x},\ [\overset{w}{\Longrightarrow}\overline{x}]^R_w \neq [\overset{w}{\Longrightarrow}\overline{ax}]^R_w\}.$$

One can see that *CDAWG*(w) is the "minimization" of *STree*(w) due to the "$[(\cdot)]^R_w$ operation". Compare *STree*(w) with *CDAWG*(w) for $w =$ tt cocoa shown in FIG. 1.

The nodes $[\overset{w}{\Longrightarrow}\overline{\varepsilon}]^R_w = [\varepsilon]^R_w$ and $[\overset{w}{\Longrightarrow}\overline{w}]^R_w = [w]^R_w$ are called the *source* node and the *sink* node of *CDAWG*(w), respectively. For any $x \in \mathit{Factor}(w)$ such that $x = \overset{w}{\Longrightarrow}\overline{x}$, x is said to be represented by the *explicit* node $[\overset{w}{\Longrightarrow}\overline{x}]^R_w$. If $x \neq \overset{w}{\Longrightarrow}\overline{x}$, x is said to be on an *implicit* node. The implicit node is represented by a *reference* pair $([\overset{w}{\Longrightarrow}\overline{z}]^R_w, y)$ such

that $z \in \mathit{Prefix}(x)$, $y \in \Sigma^*$ and $\overset{w}{\Longrightarrow} \cdot y = x$. When $|y|$ is minimum, the pair $([\overset{w}{\Longrightarrow}]^R_w, y)$ is called the *canonical* reference pair of x. Note that an explicit node can also be represented by a reference pair.

The *out-degree* of a node v of a suffix tree (or of a CDAWG) is denoted by $\mathit{OutDeg}(v)$.

PROPOSITION 2.1 implies that, for a string $x \in \mathit{Factor}(w)$, $\overset{w}{x}$ is not always represented on an explicit node in $\mathit{CDAWG}(w)$. Actually, in $\mathit{CDAWG}(\texttt{coco})$ displayed at FIG. 2, string $\overset{w}{\texttt{co}} = \texttt{co}$ is on an implicit node, where $w = \texttt{coco}$.

To implement $\mathit{CDAWG}(w)$ using only $O(|w|)$ space, labels of edges are represented by two integers indicating their beginning and ending positions in w, respectively. Suppose that $([\overset{w}{\Longrightarrow}]^R_w, y, [\overset{w}{\Longrightarrow}]^R_w)$ is an edge of $\mathit{CDAWG}(w)$. Then the edge label y is actually represented by a pair (i, j) of integers such that $w[i : j] = y$. A reference pair can be represented in a similar way.

2.2 On-Line Algorithm to Construct CDAWGs

We here recall the on-line algorithm to construct CDAWGs, introduced in our previous work [11]. By that algorithm we can move ahead the rightmost position of the sliding window. The algorithm is based on Ukkonen's on-line suffix tree construction algorithm [18]. It updates $\mathit{CDAWG}(w)$ to $\mathit{CDAWG}(wa)$ by inserting suffixes of wa into $\mathit{CDAWG}(w)$ in decreasing order of their lengths. Let z be the longest string in $\mathit{Factor}(w) \cap \mathit{Suffix}(wa)$. Then z is called the *longest repeated suffix* of wa and denoted by $\mathit{LRS}(wa)$. Let $z' = \mathit{LRS}(w)$. Let $|wa| = l$ and $u_1, u_2, \ldots, u_l, u_{l+1}$ be the suffixes of wa ordered in their length, that is, $u_1 = wa$ and $u_{l+1} = \varepsilon$. We categorize these suffixes of wa into the following three groups.

(Group 1) $u_1, \ldots, u_{i-1}$
(Group 2) $u_i, \ldots, u_{j-1}$ where $u_i = z'a$
(Group 3) $u_j, \ldots, u_{l+1}$ where $u_j = z$

Note all suffixes in Group 3 have already been represented in $\mathit{CDAWG}(w)$. Let $v_1, \ldots, v_{i-1}$ be the suffixes of w such that, for any $1 \leq k \leq i-1$, $v_k a = u_k$. Then we can insert all the suffixes of Group 1 into $\mathit{CDAWG}(w)$ by appending the character a at the end of edges leading to the sink node. Moreover, we can update those edges in constant time, by setting the ending position of the labels so to refer to a global variable e indicating the length of the scanned part of the input string. In this case, $e = l$. It therefore results in that we have only to care about those in Group 2. We start from the locus corresponding to $z' = v_i$ in $\mathit{CDAWG}(w)$, which is called the *active point*. If the active point is on an edge (on an implicit node), a new explicit node is created and from there a new edge labeled by a is inserted leading to the sink node. The locus of v_{i+1} can be found by following the suffix link and possibly some downward edges. For an example, see the first and second phase of the conversion of $\mathit{CDAWG}(\texttt{coco})$ to $\mathit{CDAWG}(\texttt{cocoa})$ in FIG. 2.

Assume v_{i+1} is also on an edge (on an implicit node) when it is found. Sometimes the edge can be *redirected* to the node created when u_i was inserted (see the third

phase of the conversion of *CDAWG*(cocо) to *CDAWG*(cocoa) in FIG. 2). For the detailed condition of whether the edge should be merged or not, see [11]. After the last suffix u_{j-1} is inserted to the CDAWG, all suffixes of wa are represented in the CDAWG.

We now pay attention to *LRS*$(wa) = z = u_j$. Consider the case that $\overset{wa}{\Longrightarrow} u_j = u_j$, that is, the case that u_j is now represented in an explicit node. Then we have to check whether or not u_j is the representative of $[u_j]^R_{wa}$. If not, the explicit node is *separated* into two explicit nodes $[x]^R_{wa}$ and $[u_j]^R_{wa}$, where $x \in [u_j]^R_w$ and $x \neq u_j$. See *CDAWG*(cocoao) for an example of this.

THEOREM 2.4 ([11])
For any string $w \in \Sigma^*$, *CDAWG*(w) can be constructed on-line and in $O(|w|)$ time and space.

The on-line construction of *CDAWG*(w) with $w =$ cocoao is shown in FIG. 2.

3 Suffix Trees for a Sliding Window

In this section we briefly recall Larsson's algorithm for maintaining suffix trees for a *sliding window* of width $M > 0$ [14]. Let i (resp. j) be the leftmost (resp. rightmost) position of the window sliding in w, that is, $j - i + 1 = M$. To move the sliding window ahead, we need to increment i and j. Incrementing j can be accomplished by Ukkonen's on-line algorithm. On the other hand, incrementing i means to delete the leftmost character of the currently scanned string, that is, to convert *STree*(bw) into *STree*(w) with some $b \in \Sigma$ and $w \in \Sigma^*$. We focus on the path of *STree*(bw) which spells out bw from the root node. This path is called the *backbone* of *STree*(bw). Let x be the longest string in *Prefix*$(bw) - \{bw\}$ such that $\overset{bw}{\Longrightarrow} x = x$. The locus of x in *STree*(bw) is called the *deletion point* and denoted by *DelPoint*(bw). On the other hand, let z be the longest string in *Prefix*$(bw) - \{bw\}$ such that $\overset{bw}{\Longrightarrow} z = z$. The string z is called the *last node* in the backbone and denoted by *LastNode*(bw).

We first consider the case that *DelPoint*$(bw) =$ *LastNode*(bw). Now assume that *OutDeg*$(\overset{bw}{\Longrightarrow} x) \geq 3$. Then there exists an edge $(\overset{bw}{\Longrightarrow} x, y, \overset{bw}{\Longrightarrow} bw)$ in *STree*(bw) where $y \in \Sigma^+$ and $xy = bw$. Then, $\overset{w}{\Longrightarrow} x = x$ and *OutDeg*$(\overset{w}{\Longrightarrow} x) \geq 2$. This implies that, only by removing this edge from *STree*(bw), we can obtain *STree*(w). An example is shown in FIG. 3.

Now we assume *OutDeg*$(\overset{bw}{\Longrightarrow} x) = 2$ in case *DelPoint*$(bw) =$ *LastNode*(bw). We delete the edge $(\overset{bw}{\Longrightarrow} x, y, \overset{bw}{\Longrightarrow} bw)$ from *STree*(bw) such that $y \in \Sigma^+$ and $xy = bw$, in order to obtain *STree*(w) as well. Then the explicit node for x has to become implicit, since $\overset{w}{\Longrightarrow} x \neq x$ due to the fact that $OutDeg(x)$ is now 1 (by the definition no internal node of out-degree one must not exist in a suffix tree). As a result we obtain *STree*(w). An example can be seen in FIG. 4.

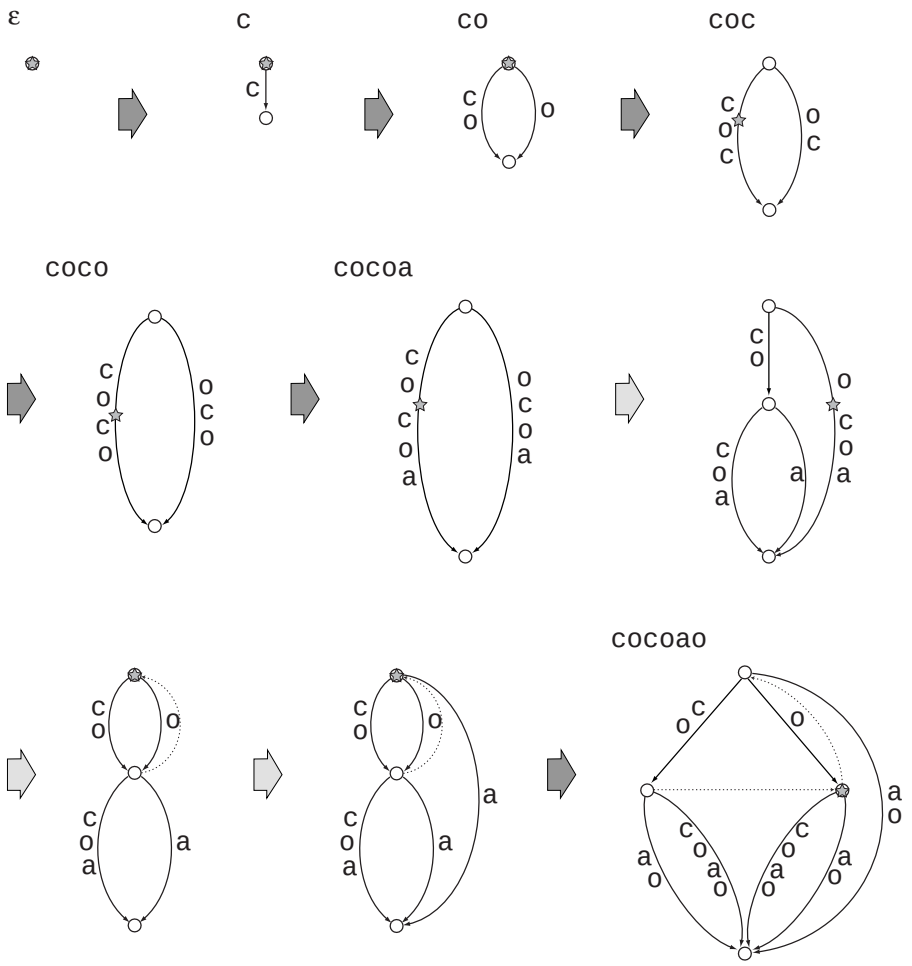

FIG. 2: The on-line construction of *CDAWG*(w) with $w =$ `cocoao`. The dotted lines represent the suffix links. The gray star indicates the active point of each step.

When *DelPoint*$(bw) \neq$ *LastNode*(bw), it follows from PROPOSITION 2.1 that $x \in$ *Suffix*(bw). Moreover, $x =$ *LRS*(bw), as to be proven by LEMMA 4.3 in Section 4.1. Namely, the active point is on the locus for x in *STree*(bw). Let $(\overset{bw}{\Longrightarrow}s, y, \overset{bw}{\Longrightarrow}bw)$ be the edge on which x is represented. Let $\overset{bw}{\Longrightarrow}s \cdot t = x$, where $t \in$ *Prefix*(y). We *shorten* the edge to $(\overset{w}{\Longrightarrow}s, t, \overset{w}{\Longrightarrow}x)$, and move the active point to the locus for the one-character shorter suffix of x. Let v be the suffix of bw which is one-character longer than *LRS*$(bw) = x$. Then, as seen in the example shown in FIG. 5, a new suffix link is

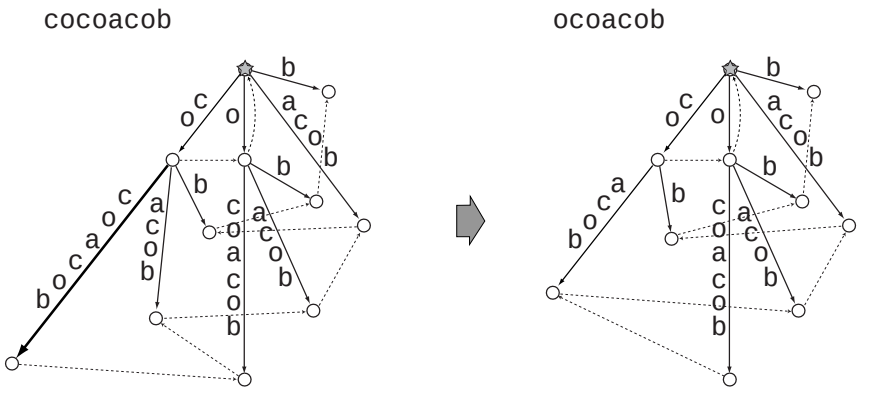

FIG. 3: To the left is *STree*(cocoacob), in which the thick edge is to be deleted. To the right is the resulting structure, *STree*(ocoacob). The gray star indicates the active point for each.

created from the leaf node v to $\overset{w}{\Longrightarrow} x$.

One can see that any of the above-mentioned procedure takes only constant time. The detection of *DelPoint*(w) after the conversion of *STree*(bw) to *STree*(w) is also feasible in constant time simply by moving via the suffix link of the leaf node that is deleted in the conversion. In the example of FIG. 5, we start from the leaf node of cocoacoc and traverse its suffix link, arriving at the leaf node of ocoacoc. After the suffix link is updated with a new character added to the right of the current string ocoacoc, we check the incoming edge of the leaf node. If the active point is on it, the edge is going to be shortened, and otherwise, deleted.

The last thing we have to care is that every edge label of a suffix tree is actually implemented by a pair of integers that indicate beginning and ending positions of a substring of the input string. Namely, we have to guarantee that no edge labels refer to positions that are already out of the sliding window. Otherwise we will not be able to maintain a suffix tree in $O(M)$ space where M is the window size. Larsson utilized the technique called *credit issuing*, introduced by Fiala and Greene [7], with which we can in linear time maintain the labels of edges appropriately. As a consequence Larsson achieved the following:

THEOREM 3.1 ([14])
Let $w \in \Sigma^*$ and M be the window size. Larsson's algorithm runs in $O(|w|)$ time using $O(M)$ space.

4 CDAWGs for a Sliding Window

In this section, we consider the maintenance of a CDAWG for a sliding window. Advancing the rightmost position of the window can be done by the on-line algorithm recalled in Section 2.2. Thus the matter is to move ahead the leftmost position of the

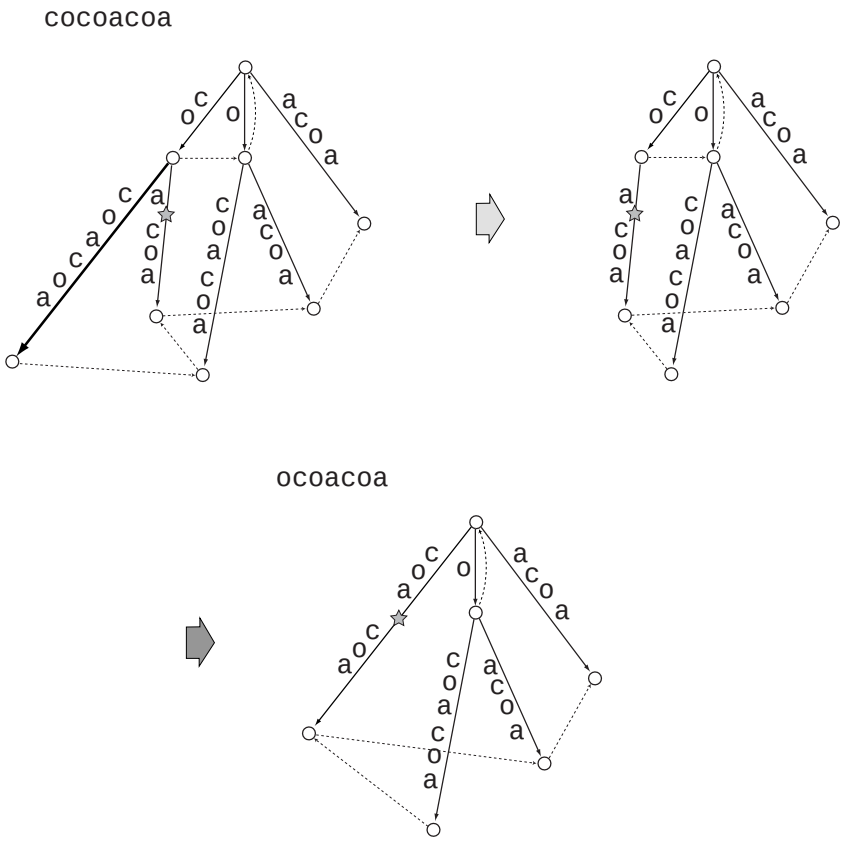

FIG. 4: The upper left is *STree*(cocoacoa). The upper right is the tree obtained by deleting the thick edge from the suffix tree. The lower is the resulting structure, *STree*(ocoacoa), in which the explicit node of out-degree one has become implicit. The gray star indicates the active point for each.

window.

4.1 Edge Deletion

Given *CDAWG*(w), we also focus on its backbone, the path spelling out w from the source node. Let $x =$ *DelPoint*(w). If *DelPoint*$(w) =$ *LastNode*(w), we remove the edge $([\overrightarrow{x}]^R_w, y, [\overrightarrow{w}]^R_w)$ such that $xy = w$. However, notice that this method might remove other suffixes of w from the CDAWG. More precise arguments follow.

LEMMA 4.1
Let $w \in \Sigma^+$, $x =$ *DelPoint*(w), and $z =$ *LastNode*(w). Assume $x = z$. Let s be

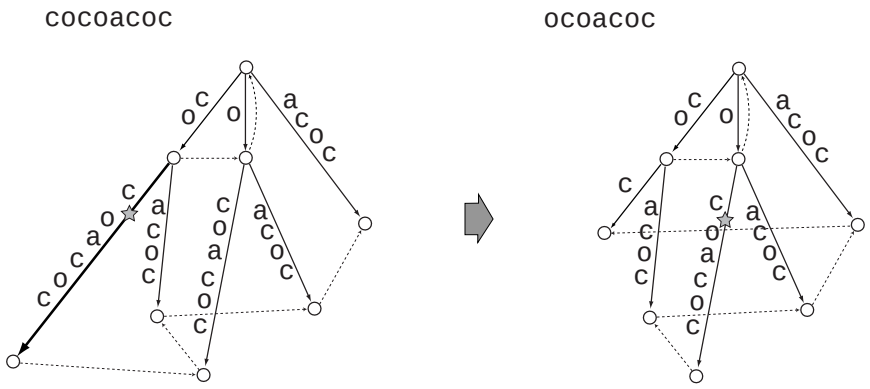

FIG. 5: To the left is *STree*(cocoacoc), in which the thick edge is to be shortened. To the right is the resulting structure, *STree*(ocoacoc). The gray star indicates the active point for each.

any string in $[\overrightarrow{\overrightarrow{x}}^{w}]^R_w = [\overrightarrow{\overrightarrow{z}}^{w}]^R_w$. Then there uniquely exists a string $y \in \Sigma^+$ such that $sy \in$ *Suffix*(w).

PROOF. Since $x =$ *DelPoint*(w), there uniquely exists a character $a \in \Sigma$ such that $xa \in$ *Factor*(w) and $\overrightarrow{\overrightarrow{xa}}^{w} = w$. Let y be the string such that $xy = w$ with $y \in \Sigma^+$, where the first character of y is a. Let s be an arbitrary element in $[x]^R_w$. Since $x \in$ *Prefix*(w), $\overleftarrow{x}^{w} = x$. Thus $s \in$ *Suffix*(x), which implies $sy \in$ *Suffix*(w). ∎

In case that *DelPoint*$(w) \neq$ *LastNode*(w), we have the following.

LEMMA 4.2
Let $w \in \Sigma^+$, $x =$ *DelPoint*(w), and $z =$ *LastNode*(w). Assume $x \neq z$. Let s be any string in $[\overrightarrow{\overrightarrow{z}}^{w}]^R_w$. Then there uniquely exist strings $t, u \in \Sigma^+$ such that $st \in$ *Suffix*(x) and $stu \in$ *Suffix*(w).

PROOF. Since $\overrightarrow{\overrightarrow{z}}^{w} = z$, $\overrightarrow{z}^{w} = z$. By the assumption that $z \neq x$, we have $z \in$ *Prefix*(x). Since $x =$ *DelPoint*(w), there uniquely exists a character $a \in \Sigma$ such that $\overrightarrow{za}^{w} = x$. Thus there is a unique string $t \in \Sigma^+$ such that $zt = x$. Since $z \in$ *Prefix*(w), $z = \overleftarrow{z}^{w}$. Therefore, for any string $s \in [z]^R_w$ it holds that $st \in$ *Suffix*(x). Moreover, there uniquely exists a character $b \in \Sigma$ such that $\overrightarrow{xb}^{w} = w$. Let $u \in \Sigma^+$ be the string satisfying $\overrightarrow{xb}^{w} = xu$. Now we have $ztu = w$, and for any $s \in [z]^R_w$, it holds that $stu \in$ *Suffix*(w). ∎

LEMMA 4.3
Let $w \in \Sigma^+$, $x =$ *DelPoint*(w), and $z =$ *LastNode*(w). Assume $x \neq z$. Then $x =$ *LRS*(w).

PROOF. Since $x \neq z$, $\overset{w}{\Longrightarrow}{x} \neq \overset{w}{x}$. Hence $\overset{w}{x} = x \in \mathit{Suffix}(w)$ by PROPOSITION 2.1. It is not difficult to show that x occurs in w just twice. Let $y = ax$ with $a \in \Sigma$, such that $y \in \mathit{Suffix}(w)$. Assume, for a contradiction, $y = \mathit{LRS}(w)$. On the assumption, y appears in w at least twice. If $y \notin \mathit{Prefix}(w)$, y must also occur in w as neither a prefix nor a suffix of w. It turns out that x appears three times in w: a contradiction. If $y \in \mathit{Prefix}(w)$, x is of the form a^{ℓ}. Then $y = \mathit{DelPoint}(w)$, which contradicts the assumption that $x = \mathit{DelPoint}(w)$. Consequently, $x = \mathit{LRS}(w)$. ∎

According to the above three lemmas, we obtain the following theorem.

THEOREM 4.4
Let $w \in \Sigma^+$, $x = \mathit{DelPoint}(w)$, and $z = \mathit{LastNode}(w)$. Let $k = |[\overset{w}{\Longrightarrow}{z}]^R_w|$. Suppose $u_1, u_2, \ldots, u_k$ be the suffixes of w arranged in decreasing order of their length, where $u_1 = w$.

1. When $x = z$: Let $xy = w$. Assume that the edge $([\overset{w}{\Longrightarrow}{x}]^R_w, y, [\overset{w}{\Longrightarrow}{w}]^R_w)$ is deleted from $\mathit{CDAWG}(w)$.
2. When $x \neq z$: Let $zt = x$ and $ztu = w$. Assume that the edge $([\overset{w}{\Longrightarrow}{z}]^R_w, tu, [\overset{w}{\Longrightarrow}{w}]^R_w)$ of $\mathit{CDAWG}(w)$ is shortened into the edge $([\overset{w}{\Longrightarrow}{z}]^R_w, t, [\overset{w}{\Longrightarrow}{x}]^R_w)$.

In both cases, the suffixes $u_1, \ldots, u_k$ are removed from the CDAWG.

What the above theorem implies is that after deleting or shortening the last edge in the backbone of $\mathit{CDAWG}(w)$, the leftmost position of a sliding window "skips" k characters at once. Let $\mathit{DelSize}(w) = k$. The next question is the exact upper bound of $\mathit{DelSize}(w)$. Fortunately, we achieve a reasonable result such that $\mathit{DelSize}(w)$ is at most about half of $|w|$. A more precise evaluation will be performed in Section 4.4.

One may wonder weather or not it is possible to delete only the leftmost character of w in (amortized) constant time. We strongly believe the answer is "No". The reason is as follows. Let $|w| = n$ where $w \in \Sigma^*$. Let $u_1, u_2, \ldots, u_{n+1}$ be all the suffixes of w arranged in decreasing order of their length. In [13], it has been proven that the total number of nodes necessary to keep $\mathit{CDAWG}(u_i)$ for every $1 \leq i \leq n+1$ is $\Theta(n^2)$, even if we minimize the CDAWGs so to share as many nodes and edges as possible. Therefore, the amortized time complexity to delete the leftmost character of w would be proportional to n.

4.2 Maintaining the Structure of CDAWG

Suppose the last edge of the backbone of $\mathit{CDAWG}(w)$ is deleted or shortened right now. Let $k = \mathit{DelSize}(w)$. Let $u = w[k+1 : n]$ where $n = |w|$. We sometimes need to modify the structure of the current graph, so that it exactly becomes $\mathit{CDAWG}(u)$. Let $x = \mathit{DelPoint}(w)$ of $\mathit{CDAWG}(w)$.

Firstly, we consider when $\mathit{OutDeg}([\overset{w}{\Longrightarrow}{x}]^R_w) \geq 3$ in the first case of THEOREM 4.4. In this case, $\overset{u}{\Longrightarrow}{x} = x$ and $\mathit{OutDeg}([\overset{u}{\Longrightarrow}{x}]^R_u) \geq 2$. It does not contradict DEFINITION 2.3, and thus no more maintenance is required. An example of the case is shown in FIG. 6.

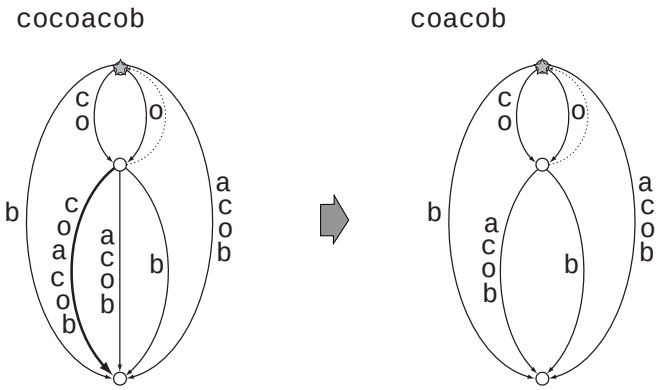

FIG. 6: On the left, *CDAWG*(cocoacob) is shown. The thick edge is to be deleted. The resulting structure is *CDAWG*(coacob), shown on the right. The gray star indicates the active point for each.

Secondly, we consider when *OutDeg*$([\overrightarrow{\overrightarrow{x}}^{w}]^R_w) = 2$ in the first case of THEOREM 4.4. Let $([\overrightarrow{\overrightarrow{r}}^{w}]^R_w, s, [\overrightarrow{\overrightarrow{x}}^{w}]^R_w)$ be an arbitrary in-coming edge of the node $[\overrightarrow{\overrightarrow{x}}^{w}]^R_w$ in *CDAWG*(w). Assume $\overrightarrow{\overrightarrow{r}}^{w} = r$, that is, $rs \in$ *Suffix*(x). Let $([\overrightarrow{\overrightarrow{x}}^{w}]^R_w, t, [\overrightarrow{\overrightarrow{w}}^{w}]^R_w)$ be the edge which is to be the sole remaining out-going edge of the node $[\overrightarrow{\overrightarrow{x}}^{u}]^R_u$ after the deletion. Notice that, however, $\overrightarrow{\overrightarrow{x}}^{u} = u$. Thus the edge $([\overrightarrow{\overrightarrow{r}}^{w}]^R_w, s, [\overrightarrow{\overrightarrow{x}}^{w}]^R_w)$ is modified to $([\overrightarrow{\overrightarrow{r}}^{u}]^R_u, st, [\overrightarrow{\overrightarrow{u}}^{u}]^R_u)$. The total time required in the operations is proportional to the number of in-coming edges of the node $[\overrightarrow{\overrightarrow{x}}^{w}]^R_w$ in *CDAWG*(w). It is bounded by *DelSize*(w).

Moreover, we might need a maintenance of the active point. Let $v =$ *LRS*(w). Supposing that $v \in$ *Prefix*(xt), v is represented on the edge $([\overrightarrow{\overrightarrow{x}}^{w}]^R_w, t, [\overrightarrow{\overrightarrow{w}}^{w}]^R_w)$ in *CDAWG*(w). The active point is actually referred to as the pair $([\overrightarrow{\overrightarrow{x}}^{w}]^R_w, p)$, where $p \in$ *Prefix*(t) and $xp = v$. The reference pair is modified to $([\overrightarrow{\overrightarrow{r}}^{u}]^R_u, sp)$ in *CDAWG*(u). Note that $\overleftarrow{\overrightarrow{r}}^{u} \cdot sp = v$. An example of the case is shown in FIG. 7.

Thirdly, we consider the second case in THEOREM 4.4. In this case the last edge in the backbone $([\overrightarrow{\overrightarrow{z}}^{w}]^R_w, tu, [\overrightarrow{\overrightarrow{w}}^{w}]^R_w)$ is shortened to $([\overrightarrow{\overrightarrow{z}}^{u}]^R_u, t, [\overrightarrow{\overrightarrow{x}}^{u}]^R_u) = ([\overrightarrow{\overrightarrow{z}}^{u}]^R_u, t, [\overrightarrow{\overrightarrow{u}}^{u}]^R_u)$ in *CDAWG*(u). It implies that $x \neq$ *LRS*(u), although $x =$ *LRS*(w). The active point of *CDAWG*(w) is represented by $([\overrightarrow{\overrightarrow{z}}^{w}]^R_w, t)$, since $zt = x$ (by LEMMA 4.3). Let *SufLink*$([\overrightarrow{\overrightarrow{z}}^{w}]^R_w) = [\overrightarrow{\overrightarrow{s}}^{w}]^R_w$. Assuming $\overrightarrow{\overrightarrow{s}}^{w} = s$, s is the longest string such that $s \in$ *Suffix*(z) and $s \notin [\overrightarrow{\overrightarrow{z}}^{w}]^R_w$. Notice that *LRS*$(u) = st$. Hereby, the reference pair of the active point is changed to $([\overrightarrow{\overrightarrow{s}}^{u}]^R_u, t)$. If $[\overrightarrow{\overrightarrow{s}}^{u}]^R_u$ is the explicit parent node nearest

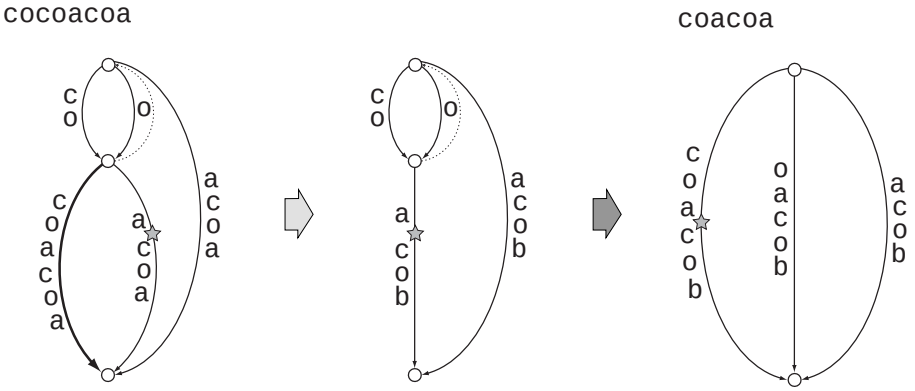

FIG. 7: On the left, *CDAWG*(cocoacoa) is shown, where the thick edge is to be deleted. The center is the intermediate structure in which the edge is deleted. After the modifications, we obtain *CDAWG*(coacoa), shown on the right. The gray star indicates the active point for each.

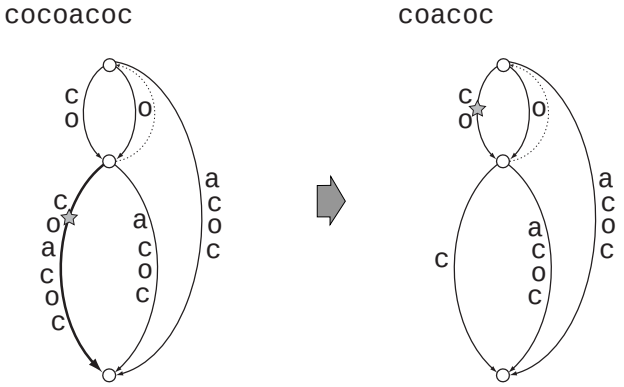

FIG. 8: On the left, *CDAWG*(cocoacoc) is shown. The thick edge is to be shortened. The resulting structure is *CDAWG*(coacoc), shown on the right. The gray star indicates the active point for each.

the locus of st, we are done. If not, the reference pair is canonized to $([\overset{u}{\Longrightarrow} r]_u^R, p)$ such that $s \in$ *Prefix*$(\overset{u}{\Longrightarrow} r)$, $st = \overset{u}{\Longrightarrow} r \cdot p$, and $|p|$ is minimum. An example of the case is shown in FIG. 8.

4.3 Detecting DelPoint(u)

Suppose that after the edge deletion or shortening of *CDAWG*(w), we got *CDAWG*(u), where $u \in$ *Suffix*(w). The problem is how to locate *DelPoint*(u) in *CDAWG*(u).

A naive solution is to traverse the backbone of *CDAWG*(u) from the source node. However, it takes $O(|u|)$ time, which leads to quadratic time complexity in total.

Our approach is to keep track of the *advanced point* that corresponds to the locus of $w[1 : n-1]$, where $n = |w|$. Let $x = \mathit{LastNode}(w)$ and $xy = w$, that is, $([\overrightarrow{x}^{w}]^R_w, y, [\overrightarrow{w}^{w}]^R_w)$ is the edge for deletion or shortening. The canonical reference pair for the advanced point is $([\overrightarrow{x}^{w}]^R_w, t)$, where $t \in \mathit{Prefix}(y)$ and $\overrightarrow{x}^{w} \cdot t = w[1 : n-1]$. We move to node $[\overrightarrow{s}^{w}]^R_w = \mathit{SufLink}([\overrightarrow{x}^{w}]^R_w)$. Suppose *CDAWG*$(w)$ has already been converted to *CDAWG*(u). Assume $[\overrightarrow{s}^{u}]^R_u = [\overrightarrow{s}^{w}]^R_w$. Since $\overrightarrow{s}^{u} \cdot t = u[1 : m-1]$ where $m = |u|$, $([\overrightarrow{s}^{u}]^R_u, t)$ is a reference pair of the next advanced point, and then it is canonized. Let $([\overrightarrow{s'}^{u}]^R_u, t')$ be the canonical reference pair of the advanced point. Then $\overrightarrow{s'}^{u} = \mathit{LastNode}(u)$. If $\mathit{LastNode}(u) \notin \mathit{Prefix}(\mathit{LRS}(u))$, that is, if the active point is not on the longest out-going edge from the node $[\overrightarrow{s'}^{u}]^R_u$, $\mathit{DelPoint}(u) = \mathit{LastNode}(u)$. Otherwise, $\mathit{DelPoint}(u) = \mathit{LRS}(u)$. In the case that $[\overrightarrow{s}^{u}]^R_u \neq [\overrightarrow{s}^{w}]^R_w$, we perform the same procedure from its closest parent node (see FIG. 7). If Σ is fixed, the cost of canonizing the reference pair is only proportional to the number of nodes included in the path. The amortized number of such nodes is constant.

4.4 On Buffer Size

The following theorem is the main result of this section, which shows an exact estimation of the upperbound of $\mathit{DelSize}(w)$. For an alphabet Σ and an integer n, let $\mathit{MaxDel}_\Sigma(n) = \max\{\mathit{DelSize}(w) \mid w \in \Sigma^*, |w| = n\}$.

THEOREM 4.5
If $|\Sigma| \geq 3$, $\mathit{MaxDel}_\Sigma(n) = \lceil \frac{n}{2} \rceil - 1$.

By this theorem, edge deletion or edge shortening can shrink the window size upto the half of the original size. Therefore, in order to keep the window size at least M, a buffer of size $2M + 1$ is necessary and sufficient.

We will prove the above theorem in the sequel.

LEMMA 4.6
Let $w \in \Sigma^*$. For any string $x \in \mathit{Factor}(w)$, let $\mathit{SufLink}([\overrightarrow{x}^{w}]^R_w) = [\overrightarrow{s}^{w}]^R_w$. Then $|[\overrightarrow{x}^{w}]^R_w| = |\overrightarrow{x}^{w}| - |\overrightarrow{s}^{w}|$.

PROOF. $|[\overrightarrow{x}^{w}]^R_w| = |\mathit{Suffix}(\overrightarrow{x}^{w})| - |\mathit{Suffix}(\overrightarrow{s}^{w})| = (|\overrightarrow{x}^{w}|+1) - (|\overrightarrow{s}^{w}|+1) = |\overrightarrow{x}^{w}| - |\overrightarrow{s}^{w}|$. ∎

LEMMA 4.7
Let $w \in \Sigma^*$ and $n = |w|$. For any $x \in \mathit{Prefix}(w) - \{w\}$ with $\overrightarrow{x}^{w} = x$, $|[\overrightarrow{x}^{w}]^R_w| \leq \lceil \frac{n}{2} \rceil - 1$.

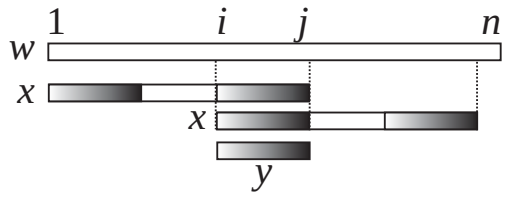

FIG. 9: The case $j > \frac{n}{2}$. x occurs at least twice in w, and the overlap y is in fact both a prefix and a suffix of x.

PROOF. Let $j = |x| = |\overrightarrow{x}^{w}|$. Let *SufLink*$([\overrightarrow{x}^{w}]^{R}_{w}) = [\overrightarrow{s}^{w}]^{R}_{w}$. We have the following three cases.

(1) When $j < \frac{n}{2}$: Since j is an integer, $j \leq \lceil \frac{n}{2} \rceil - 1$, and $|[\overrightarrow{x}^{w}]^{R}_{w}| = |\overrightarrow{x}^{w}| - |\overrightarrow{s}^{w}| \leq \lceil \frac{n}{2} \rceil - 1$ by LEMMA 4.6.

(2) When $j > \frac{n}{2}$: (See FIG. 9.) The equivalences $x = w[1 : j]$ and $\overrightarrow{x}^{w} = x$ imply that $x = w[i : i+j-1]$ for some $i \geq 2$ and $i+j-1 \leq n$. Then $i-j \leq n-2j+1 < 1$, that is, $i \leq j$. Let $y = w[i : j]$. Its length is $|y| = j - i + 1 \geq 1$, and $y = x[i : j] \in$ *Suffix*(x). Since $\overrightarrow{x}^{w} = x$ and $y \in$ *Suffix*(x), $\overrightarrow{y}^{w} = y$. On the other hand, $y = w[i : j] = x[1 : j - i + 1] = w[1 : j - i + 1] \in$ *Prefix*(w), which implies $\overleftarrow{y}^{w} = y$. Thus y is the longest element of $[\overrightarrow{y}^{w}]^{R}_{w}$. Since $|x| > |y|$, $x \notin [\overrightarrow{y}^{w}]^{R}_{w}$. Therefore $|\overrightarrow{s}^{w}| \geq |y|$, which yields $|[\overrightarrow{x}^{w}]^{R}_{w}| = |\overrightarrow{x}^{w}| - |\overrightarrow{s}^{w}| \leq |x| - |y| = j - (j - i + 1) = i - 1 \leq n - j < n - \frac{n}{2} = \frac{n}{2}$. Thus $|[\overrightarrow{x}^{w}]^{R}_{w}| \leq \lceil \frac{n}{2} \rceil - 1$.

(3) When $j = \frac{n}{2}$: Since $\overrightarrow{x}^{w} = x$, x occurs in w at least twice. If $x = w[i : i + j - 1]$ for some i with $2 \leq i \leq j$, we can show the inequality holds in the same way as (2). Otherwise, $x = w[j + 1 : 2j] = w[j + 1 : n] = w[1 : j]$, that is $w = xx$. Then $\overrightarrow{x}^{w} = w \neq x$, which does not satisfy the precondition of the lemma.

In any cases, we have got the result. ∎

We are ready to prove the upperbound of *MaxDel*$_{\Sigma}(n)$.

LEMMA 4.8
MaxDel$_{\Sigma}(n) \leq \lceil \frac{n}{2} \rceil - 1$ for any Σ and any $n \geq 3$.

PROOF. Let $x =$ *DelPoint*(w) and $z =$ *LastNode*(w). First we consider the case $x = z$. Since $\overrightarrow{x}^{w} = x$ and $x \in$ *Prefix*$(w) - \{w\}$, *DelSize*$(w) = |[\overrightarrow{z}^{w}]^{R}_{w}| = |[\overrightarrow{x}^{w}]^{R}_{w}| \leq \lceil \frac{n}{2} \rceil - 1$ by LEMMA 4.7.

We now assume $x \neq z$. Then $z \in$ *Prefix*(x), and $x =$ *DelPoint*(w) implies that $x \in$ *Prefix*$(w) - \{w\}$, which yields $z \in$ *Prefix*$(w) - \{w\}$. Thus by LEMMA 4.7, *DelSize*$(w) = |[\overrightarrow{z}^{w}]^{R}_{w}| \leq \lceil \frac{n}{2} \rceil - 1$. ∎

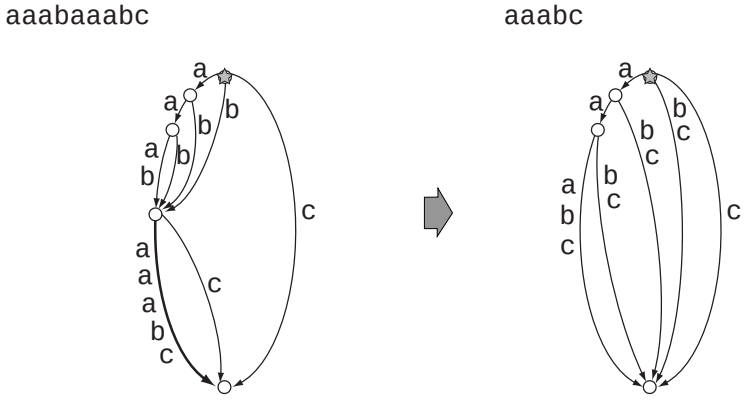

FIG. 10: To the left is *CDAWG*(aaabaaabc), in which the thick edge is to be deleted. To the right is the resulting structure *CDAWG*(aaabc). The gray star indicates the active point.

On the other hand, the lowerbound is given by the following lemma.

LEMMA 4.9
If $|\Sigma| \geq 3$, $\mathit{MaxDel}_\Sigma(n) \geq \lceil \frac{n}{2} \rceil - 1$ for any $n \geq 1$.

PROOF. For each $1 \leq n \leq 4$, the inequality trivially holds since $\lceil \frac{n}{2} \rceil - 1 \leq 1$. Let $\mathsf{a}, \mathsf{b}, \mathsf{c}$ be distinct symbols in Σ. For each odd $n \geq 5$, let $w_n = \mathsf{a}^k\mathsf{ba}^k\mathsf{bc}$, where $k = \frac{n-3}{2}$. Remark that $\mathit{DelPoint}(w_n) = \mathsf{a}^k\mathsf{b}$ (see FIG. 10). Let $x = \mathsf{a}^k\mathsf{b}$. We can see that any suffix of x except ε belongs to $\lceil \overrightarrow{x} \rceil^R_w$, so that $\mathit{SufLink}(x) = \varepsilon$. Thus $\mathit{DelSize}(w_n) = |x| - |\varepsilon| = |\mathsf{a}^k\mathsf{b}| - 0 = k + 1 = \frac{n-1}{2} = \lceil \frac{n}{2} \rceil - 1$, since n is odd.

For each even $n \geq 6$, let $w'_n = \mathsf{a}^{k-1}\mathsf{ba}^k\mathsf{bc}$, where $k = \frac{n}{2} - 1$, and we can verify that $\mathit{DelSize}(w'_n) = \lceil \frac{n}{2} \rceil - 1$ similarly. ∎

Consequently, THEOREM 4.5 is proved by LEMMA 4.8 and LEMMA 4.9. We note that for a binary alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}\}$, two series of the strings $\mathsf{a}^k\mathsf{ba}^k\mathsf{bab}$ and $\mathsf{a}^{k-1}\mathsf{ba}^k\mathsf{bab}$ give the lowerbound $\mathit{MaxDel}_\Sigma(n) \geq \lceil \frac{n-3}{2} \rceil$.

On the on-line algorithm of [11], each node $\lceil \overrightarrow{x} \rceil^R_w$ of *CDAWG*(w) stores the value of $|\overrightarrow{x}|$. By LEMMA 4.6, it is guaranteed that we can calculate $\mathit{DelSize}(w)$ in constant time with no additional information.

4.5 Keeping Edge Labels Valid

As mentioned previously, an edge label is actually represented by a pair of integers indicating its beginning and ending positions in input string w, respectively. We must ensure that no edge label becomes "out of date" after the window slides, e.g., that no integer refers to a position outside the sliding window. In case of a suffix tree,

when a new edge is created, we can guarantee the above regulation by traversing from the leaf node toward the root node while updating all edge labels encountered. However, this would yield quadratic time complexity in the aggregate. Larsson [14, 15] utilized *credit issuing*, an update-number-restriction technique, originally proposed in [7], which takes in total $O(|w|)$ time and space. In the following, we introduce an extended credit issuing technique for CDAWGs. Our basic strategy is to show that we can handle the credit issuing as well as in case of suffix trees.

We assign each internal node s of *CDAWG*(w) a binary counter called *credit*, denoted by *Cred*(s). This credit counter is initially set to zero when s is created. When a node s receives a credit, we update the labels of in-coming edges of s. Then, if *Cred*$(s) = 0$, we set it to one, and stop. If *Cred*$(s) = 1$, after setting it to zero, we let the node s issue a credit to its parent nodes.

When s is newly created, *Cred*$(s) = 0$. The creation of the new node s implies that a new edge is to be inserted from s to the sink node. When the new edge is created leading to the sink node, the sink node *issues* a credit to the parent node s. Assume the new edge is labeled by pair (i, j) where i, j are some integers with $i < j$. Let ℓ be the length of the label of the in-coming edge of s. After s received a credit from the sink node, we reset its in-coming edge label to $(i - \ell, i - 1)$. Remember the edge redirection happening in the construction of a CDAWG (see Section 2.2 or [11]). If some edge is actually redirected to node s, its label is updated as well. Note that we need not change the value of *Cred*(s) again.

Suppose a node r has right now received a credit from one of its child nodes. Assume *Cred*(r) is currently one. We need to update all in-coming edge labels of r. We store a list in r to maintain its in-coming edges arranged in the order of the length of the path they correspond to. The maintenence of the list is an easy matter, since the on-line algorithm of [11] inserts edges to r in such order. Let t be an arbitrary parent node of r. Let k be the number of the in-coming edges of r connected from t. One might wonder that r must issue k credits to t, but there is the following time-efficient method. In case k is even, *Cred*(t) need not to be changed because it is a *binary* counter. Contrarily, in case k is odd, we always change the value of *Cred*(t). If *Cred*(t) was one, we also have to update the in-coming edge of t. To do it, we focus on the *shortest* in-coming edge of r connected from t, which is in turn the shortest out-going edge of t leading to r. In updating the in-coming edges of t, we should utilize the label of the shortest edge, since the label corresponds to the possibly newest occurrence of the factors represented in node t. We continue updating edge labels by traversing the reversed graph rooted at r in width-first manner while issuing credits.

Recall the node separation in constructing a CDAWG (see Section 2.2 or [11]). Assume a node r has right now been created owing to the separation of a node s. The subgraph rooted at r is currently the same as the one rooted at s, since r was created as a clone of s. Thus we simply set *Cred*$(r) =$ *Cred*(s).

Now consider a node u to be deleted, corresponding to the second case of Section 4.2. It might have received a credit from its newest child node (that is not deleted), which has not been issued to its parent node yet. Therefore, when a node u is scheduled for deletion and *Cred*$(u) = 1$, node u issues credits to its parent nodes. However, this complicates the update of edge labels: several waiting credits may aggregate,

causing nodes upper in the CDAWG to receive a credit *older* than the one it has already received from its another child node. Therefor, before updating an edge label, we compare its previous value against the one associated with the received credit, and refer to the newer one. As well as the case of edge insertion mentioned in the above paragraph, we traverse the reversed graph rooted at u in width-first fashion to update edge labels. In the worst case, the updating cost is proportional to the number of paths from the source node to node u. Nevertheless, it is bounded by *DelSize*(w).

By analogous arguments to [7, 14, 15], we can establish the following lemma.

LEMMA 4.10
All edge labels of a CDAWG can be kept valid in a sliding window, in linear time and space with respect to the length of an input string.

As a conclusion of Section 4, we finally obtain the following.

THEOREM 4.11
Let $w \in \Sigma^*$ and M be the window size. The proposed algorithm runs in $O(|w|)$ time using $O(M)$ space.

5 Conclusion

We introduced an algorithm to maintain CDAWGs for a sliding window, which runs in linear time and space. It can be an alternative of Larsson's suffix tree algorithm in [14]. Moreover, CDAWGs are known to be more space-economical than suffix trees [2, 6], and thus our algorithm seems to contribute to reducing the space requirement in PPM*-style data compression scheme.

It is still an open problem whether conversion of *CDAWG*(bu) to *CDAWG*(u) can be done in (amortized) constant time for any character b and string u. Also, it is surely worth considering *DAWGs for a sliding window* where labels of edges of DAWGs are single characters. This is really a big advantage in the scheme of a sliding window since we would not need credit issuing then, which is surely time-consuming and makes the algorithm rather complicated. However, we hold a strong belief that conversion of *DAWG*(bu) into *DAWG*(u) cannot be done in (amortized) constant time, either. Thus we will need some alternative way, like in case of CDAWGs.

Acknowledgment

We wish to thank Ricardo Garcia and Veli Mäkinen who suggested us to apply our previous work [11] to the sliding window mechanism. We also appreciate Ricardo Garcia's fruitful comments and suggestions in the early stage of this work.

References

[1] Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithm on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 85–96. Springer-Verlag, 1985.

[2] Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[3] John G. Cleary, W. J. Teahan, and Ian H. Witten. Unbounded length contexts for PPM. In *Proc. Data Compression Conference '95 (DCC'95)*, pages 52–61. IEEE Computer Society, 1995.

[4] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, 32(4):396–402, 1984.

[5] Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, New York, 1994.

[6] Maxime Crochemore and Renaud Vérin. On compact directed acyclic word graphs. In *Structures in Logic and Computer Science*, volume 1261 of *Lecture Notes in Computer Science*, pages 192–211. Springer-Verlag, 1997.

[7] Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989.

[8] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.

[9] Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Construction of the CDAWG for a trie. In *Proc. The Prague Stringology Conference '01 (PSC'01)*. Czech Technical University, 2001.

[10] Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. On-line construction of symmetric compact directed acyclic word graphs. In *Proc. 8th International Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 96–110. IEEE Computer Society, 2001.

[11] Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa, Giancarlo Mauri, and Giulio Pavesi. On-line construction of compact directed acyclic word graphs. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, 2001.

[12] Shunsuke Inenaga, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Compact directed acyclic word graphs for a sliding window. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE'02)*, volume 2476 of *Lecture Notes in Computer Science*, pages 310–324. Springer-Verlag, 2002.

[13] Shunsuke Inenaga, Ayumi Shinohara, Masayuki Takeda, Hideo Bannai, and Setsuo Arikawa. Space-economical construction of index structures for all suffixes of a string. In *Proc. 27th International Symposium on Mathematical Foundations of Computer Science (MFCS'02)*, volume 2420 of *Lecture Notes in Computer Science*, pages 341–352. Springer-Verlag, 2002.

[14] N. Jesper Larsson. Extended application of suffix trees to data compression. In *Proc. Data Compression Conference '96 (DCC'96)*, pages 190–199. IEEE Computer Society, 1996.

[15] N. Jesper Larsson. *Structures of String Matching and Data Compression*. PhD thesis, Lund University, 1999.

[16] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[17] Alistair Moffat. Implementing the PPM data compression scheme. *IEEE Trans. Commun.*, 38(11):1917–1921, 1990.

[18] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[19] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Received XX YY 2003