On-Line Construction of Compact Directed Acyclic Word Graphs

Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa

Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
{s-ine, hoshino, ayumi, takeda, arikawa}@i.kyushu-u.ac.jp

Abstract. Directed Acyclic Word Graph (DAWG) is a space efficient data structure that supports indices of a string. Compact Directed Acyclic Word Graph (CDAWG) is a more space efficient variant of DAWG. Crochemore and Vérin gave the first direct algorithm to construct CDAWGs from given strings, based on the McCreight's algorithm for suffix trees. In this paper, we give an Ukkonen's counterpart for CDAWGs. That is, we show an *on-line* algorithm that constructs CDAWGs from given strings directly.

1 Introduction

A Directed Acyclic Word Graph (DAWG) is the smallest finite state automaton that recognizes all suffixes of a given string [1]. DAWG is involved in several combinatorial algorithms on strings, because it serves as indices on the string, as well as other indexing structures like suffix trie, suffix tree, and suffix array (see eg. [2, 3, 8]). All of these indexing structures except for suffix trie can be constructed in linear time with respect to the size of a given string, and the space requirements are also linear. The hidden constants behind the big-O notation of space complexity are critical in practice, and much attention has been paid to reduce these constants recently.

Blumer et al. [2] first introduced the *Compact Directed Acyclic Word Graph* (CDAWG), a space efficient variant of DAWG, that is obtained by deleting all nodes of out-degree one and their corresponding edges. They showed a linear-time algorithm to construct CDAWGs, that actually shrinks DAWGs into CDAWGs.

Crochemore and Vérin developed an algorithm that builds CDAWGs from given strings *directly*, which avoids constructing DAWGs as intermediates [4, 5]. The algorithm is, for some reason, based on McCreight's algorithm [9] for suffix trees, that processes a given string from right to left. As a result, unfortunately, the proposed algorithm does not have an "on-line" property that may be useful in some situations.

As is well-known, for constructing suffix trees, Ukkonen's algorithm [11] has the on-line property, and is easier to understand [3, 6, 7]. The Ukkonen algorithm is based on an intuitive on-line construction of suffix tries, and an invention of "open-transition" enables it to run in linear time. Moreover, Ukkonen remarked that the on-line construction algorithm for DAWGs by Blumer et al. [1] also can be naturally derived from that for suffix tries. As Crochemore and Vérin explained [5], suffix tree is the compact version of suffix trie, and DAWG is the minimized version of suffix trie. CDAWG can be obtained either by minimizing suffix tree or by compacting DAWG (Fig. 1). In this sense, a missing piece, which we have been looking for, is an *on-line construction algorithm for CDAWGs.*

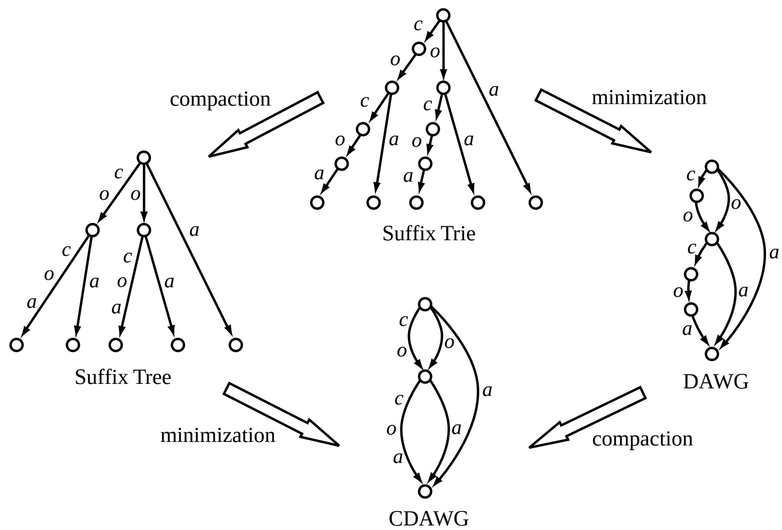

Fig. 1. Relationship among suffix trie, suffix tree, DAWG, and CDAWG for the string "*cocoa*".

In this paper, we give the very one. We show an on-line algorithm that constructs CDAWGs from given strings directly, based on the Ukkonen algorithm. We believe that our algorithm is clearer than that by Crochemore and Vérin [4, 5], as is Ukkonen's than McCreight's. As a delightful consequence, we now have a unified view of all these on-line construction algorithms for suffix tries, suffix trees, DAWGs, and CDAWGs.

2 Unified view of Suffix trie, suffix tree, DAWG, and CDAWG

We in this section clarify the relationship among the suffix trie, the suffix tree, the DAWG, and CDAWG, which is based on equivalence relations on strings. This is needed both for description of our algorithm for on-line construction of the CDAWG, which will be presented in the next section, and for an unified view of these indexing structures.

2.1 Preliminaries

Let Σ be a finite alphabet. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of the string $u = xyz$, respectively. The sets of prefixes, factors, and suffixes of a string w are denoted by $\mathit{Prefix}(w)$, $\mathit{Factor}(w)$, and $\mathit{Suffix}(w)$, respectively. The length of a string u is denoted by $|u|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. The ith symbol of a string u is denoted by $u[i]$ for $1 \le i \le |u|$, and the factor of a string u that begins at position i and ends at position j is denoted by $u[i:j]$ for $1 \le i \le j \le |u|$. For convenience, let $u[i:j] = \varepsilon$ for $j < i$. For an arbitrary equivalence relation $\equiv$ on Σ^*, let $\Sigma^*/\!\equiv$ denote the quotient of Σ^* by $\equiv$.

For strings $x, y \in \Sigma^*$, we write as $x \equiv_w^L y$ (resp. $x \equiv_w^R y$) if the sets of positions in w at which x and y begin (resp. end) are identical. The equivalence class of a string $x \in \Sigma^*$ with respect to $\equiv_w^L$ (resp. $\equiv_w^R$) is denoted by $[x]_w^L$ (resp. $[x]_w^R$).

Note that all strings that are not in *Factor*(w) form one equivalence class under $\equiv_w^L$ ($\equiv_w^R$). This equivalence class is called the *degenerate* class. It follows from the definition of $\equiv_w^L$ that if two factors x and y of w are in a single equivalent class under $\equiv_w^L$, then either x is a prefix of y, or vice versa. Therefore, each equivalence class in $\equiv_w^L$ other than the degenerate class has a unique longest member. Similar discussion holds for $\equiv_w^R$.

For any factor x of a string $w \in \Sigma^*$, let $\overset{w}{\overrightarrow{x}}$ and $\overset{w}{\overleftarrow{x}}$ denote the unique longest members of $[x]_w^L$ and $[x]_w^R$, respectively.

2.2 Suffix trie, suffix tree, DAWG, and CDAWG

We use the terminology of automata and graph theories in description of property on strings.

Definition 1. *Let $w \in \Sigma^*$. The* out-degree *of a string $x \in Factor(w)$ w.r.t. w, denoted by $out\text{-}deg_w(x)$, is defined to be the number of distinct symbols c such that $xc \in Factor(w)$. A string $x \in Factor(w)$ is said to be*

- branching *w.r.t. w if it has out-degree more than one;*
- accepting *w.r.t. w if it is a suffix of w; and*
- proper *w.r.t. w if it is branching or accepting.*

Let *Branching*(w) (resp. *Proper*(w)) denote the set of factors of w that are branching (resp. proper) w.r.t. a string $w \in \Sigma^*$.

We here recall definitions of the suffix trie, the suffix tree, the DAWG, and CDAWG for a string $w \in \Sigma^*$, which are denoted by *STrie*(w), *STree*(w), *DAWG*(w), and *CDAWG*(w), respectively. This is necessary for a unified view of construction of these four indexing structures, which will be mentioned later.

$$
STrie(w) \;:\; \begin{cases} V = Factor(w), \\ E = \left\{ x \xrightarrow{\sigma} y \;\middle|\; x, y \in Factor(w),\ \sigma \in \Sigma, \text{ and } y = x\sigma \right\} \end{cases}
$$

$$
STree(w) \;:\; \begin{cases} V = Proper(w), \\ E = \left\{ x \xrightarrow{\sigma\gamma} y \;\middle|\; \begin{array}{l} x, y \in Proper(w),\ \sigma \in \Sigma,\ \gamma \in \Sigma^*, \\ \text{and } y = \overset{w}{\overrightarrow{x\sigma}} = x\sigma\gamma \end{array} \right\} \end{cases}
$$

$$
DAWG(w) \;:\; \begin{cases} V = \left\{ [x]_w^R \mid x \in Factor(w) \right\} = Factor(w)/\equiv_w^R, \\ E = \left\{ [x]_w^R \xrightarrow{\sigma} [y]_w^R \;\middle|\; \begin{array}{l} x, y \in Factor(w),\ \sigma \in \Sigma, \\ \text{and } y = x\sigma \end{array} \right\} \end{cases}
$$

$$
CDAWG(w) : \begin{cases} V = \left\{ [x]_w^R \mid x \in Proper(w) \right\} = Proper(w)/\equiv_w^R, \\ E = \left\{ [x]_w^R \xrightarrow{\sigma\gamma} [y]_w^R \;\middle|\; \begin{array}{l} x, y \in Proper(w),\ \sigma \in \Sigma,\ \gamma \in \Sigma^*, \\ \text{and } y = \overset{w}{\overrightarrow{x\sigma}} = x\sigma\gamma \end{array} \right\} \end{cases}
$$

The nodes $[\varepsilon]^R_w$ and $[w]^R_w$ of *DAWG*(w) (or *CDAWG*(w)) are often referred to as the *source* and the *sink* nodes, respectively. While the nodes of *STrie*(w) are all strings in *Factor*(w), the nodes of *STree*(w) are limited to the strings in *Proper*(w). In this sense *STree*(w) is obtained from *STrie*(w) by removing the non-proper nodes. Similarly, while the nodes of *DAWG*(w) are the equivalence classes in *Factor*$(w)/\equiv^R_w$, the nodes of *CDAWG*(w) are limited to those in *Proper*$(w)/\equiv^R_w$. We can say that *CDAWG*(w) is obtained from *DAWG*(w) by removing the non-proper nodes. We remark that the removal corresponds to the compaction of Fig. 1, and it is based on the equivalence relation $\equiv^L_w$, namely:

Proposition 1. *Let $w \in \Sigma^*$. A string $x \in$ Factor(w) is proper w.r.t. w iff $\overrightarrow{x}^{w} = x$.*

Let us define the mapping Φ_w that maps $x \in$ *Factor*(w) to the equivalence class $[x]^R_w$. Then, Φ_w maps the nodes of *STrie*(w) to the nodes of *DAWG*(w), and it induces the onto-mapping from the edges of *STrie*(w) to the edges of *DAWG*(w) without changing edge labels. In this sense it can be said that Φ_w converts *STrie*(w) into *DAWG*(w). Similarly, Φ_w converts *STree*(w) into *CDAWG*(w). The conversion corresponds to the minimization of Fig. 1. It is, of course, based on the equivalence relation $\equiv^R_w$, and therefore *CDAWG*(w) is the very structure that is based on both the equivalence relations. The observation will play a central role in development of our algorithm for CDAWGs from Ukkonen's algorithm for suffix trees.

Next we define the suffix links of these index structures.

Definition 2. *Let $w \in \Sigma^*$. For any $x \in$ Factor(w) with $x \neq \varepsilon$, let $f(x)$ be the suffix y of x such that $|x| = |y|+1$. Define the function $f^\star_w$ from Factor$(w)/\equiv^R_w - \{[\varepsilon]^R_w\}$ to Factor$(w)/\equiv^R_w$ by letting $f^\star_w([x]^R_w)$ be $[f(z)]^R_w$, where z is the shortest member of $[x]^R_w$.*

Note that the function f from *Factor*$(w) - \{\varepsilon\}$ to *Factor*(w) gives the suffix links of *STrie*(w). Moreover we have:

Proposition 2. *Let $w \in \Sigma^*$. For any $x \in$ Factor(w) with $x \neq \varepsilon$, if x is branching (resp. accepting), then $f(x)$ is branching (resp. accepting).*

This implies that every node of *STree*(w) other than the root node is mapped to a node of *STree*(w) via f. Thus the limitation of f to the domain *Proper*$(w) - \{\varepsilon\}$ gives the suffix links of *STree*(w). Similarly, the function $f^\star_w$ gives the suffix links of *DAWG*(w). We have:

Proposition 3. *Let $w \in \Sigma^*$. For any $X \in$ Factor$(w)/\equiv^R_w$ with $X \neq [\varepsilon]^R_w$, if X is branching (resp. accepting), then $f^\star_w(X)$ is branching (resp. accepting).*

This means that every node of *CDAWG*(w) other than the source node is mapped to a node of *CDAWG*(w) via $f^\star_w$. Thus the limitation of $f^\star_w$ to the domain *Proper*$(w)/\equiv^R_w - \{[\varepsilon]^R_w\}$ gives the suffix links of *CDAWG*(w).

Recall that all strings in *Factor*(w) are represented as nodes of *STrie*(w) but the situation is different in *STree*(w). That is, the nodes of *STrie*(w) representing the strings in *Factor*$(w)-$*Proper*(w) are not present in *STree*(w). These invisible nodes are often called the *implicit nodes* of *STree*(w). In [11], an explicit or implicit node r of suffix tree is referred to by a *reference pair* (s, u), where s is some explicit node that is an ancestor of r and u is the string spelled out by the transitions from s to r in the corresponding suffix trie. This can be written as $r = su$ since the nodes s and r both strings. A reference pair is said to be *canonical* if u is the shortest possible.

Similarly, while the equivalence classes in $\mathit{Factor}(w)/{\equiv^R_w}$ are represented as nodes of $\mathit{DAWG}(w)$, the nodes of $\mathit{DAWG}(w)$ representing those in $\big(\mathit{Factor}(w) - \mathit{Proper}(w)\big)/{\equiv^R_w}$ are not explicitly present in $\mathit{CDAWG}(w)$. Moreover, such an invisible node is not necessarily on a unique edge. Namely, the members of the equivalence class are possibly dispersed on more than one edge of $\mathit{CDAWG}(w)$. This corresponds to the fact that a reference pair $([x]^R_w, u)$ of an explicit or implicit node of $\mathit{CDAWG}(w)$ specifies the strings in $[x]^R_w \cdot u$, and $[x]^R_w \cdot u$ is not always identical to $[xu]^R_w$.

3 On-line construction Algorithm

In this section, we give an on-line algorithm for constructing the CDAWG on the basis of Ukkonen's on-line suffix tree construction algorithm. Hereafter, let us assume readers to be familiar with Ukkonen's algorithm.

3.1 Modifications in definitions of $\boldsymbol{STree(w)}$ and $\boldsymbol{CDAWG(w)}$

It is technically convenient to omit accepting nodes in the definition of suffix tree as in [11].

$$\mathit{STree}_{mod}(w) :$$
$$\begin{cases} V = \mathit{Branching}(w) \cup [w]^R_w, \\ E = \left\{ x \xrightarrow{\sigma\gamma} y \;\middle|\; \begin{array}{l} x, y \in \mathit{Branching}(w) \cup [w]^R_w,\ \sigma \in \Sigma,\ \gamma \in \Sigma^*, \\ y = x\sigma\gamma, \text{ and every prefix } z \text{ of } y \\ \text{with } |x| < |z| < |y| \text{ has out-degree } 1 \end{array} \right\} \end{cases}$$

One can observe that not only V but also E is modified appropriately. We also modify the definition of CDAWG in a similar way.

$$\mathit{CDAWG}_{mod}(w) :$$
$$\begin{cases} V = \big(\mathit{Branching}(w) \cup [w]^R_w\big)/{\equiv^R_w}, \\ E = \left\{ [x]^R_w \xrightarrow{\sigma\gamma} [y]^R_w \;\middle|\; \begin{array}{l} x, y \in \mathit{Branching}(w) \cup [w]^R_w,\ \sigma \in \Sigma,\ \gamma \in \Sigma^*, \\ y = x\sigma\gamma, \text{ and every prefix } z \text{ of } y \\ \text{with } |x| < |z| < |y| \text{ has out-degree } 1 \end{array} \right\} \end{cases}$$

In spite of the modification in these definitions, $\mathit{STree}_{mod}(w\#)$ and $\mathit{CDAWG}_{mod}(w\#)$ coincide with $\mathit{STree}(w\#)$ and $\mathit{CDAWG}(w\#)$, respectively, where # is a symbol that never appears in string w. The Ukkonen algorithm builds $\mathit{STree}_{mod}(w)$ for a string w, and similarly we will give an algorithm that builds $\mathit{CDAWG}_{mod}(w)$. We remark that the function Φ_w induces the onto-mapping from the edges of $\mathit{STree}_{mod}(w)$ to $\mathit{CDAWG}_{mod}(w)$ as in the case of $\mathit{STree}(w)$ and $\mathit{CDAWG}(w)$.

3.2 What happens on CDAWG when a new symbol is added?

The next proposition gives the condition that string x that was not proper becomes proper after a symbol a is added to the input string w.

Proposition 4. *Let $w \in \Sigma^*$ and $a \in \Sigma$. Let $x \in Factor(w)$.*

1. *If $out\text{-}deg_w(x) = 0$, then $out\text{-}deg_{wa}(x) = 1$. In this case x must be in the old sink $[w]^R_w$ and it becomes implicit in $CDAWG_{mod}(wa)$.*
2. *If $out\text{-}deg_w(x) > 1$, then $out\text{-}deg_{wa}(x) > 1$.*
3. *If $out\text{-}deg_w(x) = 1$ and $x \in Suffix(w)$, then:*
   (a) *If $xa \in Factor(w)$, then $out\text{-}deg_{wa}(x) = 1$.*
   (b) *Otherwise, $out\text{-}deg_{wa}(x) = 2$.*
   That is, x becomes branching w.r.t. wa iff $xa \notin Factor(w)$. Such strings xa form the new sink $[wa]^R_{wa}$.
4. *If $out\text{-}deg_w(x) = 1$ and $x \notin Suffix(w)$, then $out\text{-}deg_{wa}(x) = 1$.*

It turns out from Proposition 4 that we have to care only about the suffixes of the input string. We remark that since the new proper strings in (3-b) were originally implicit, they must be merged into new explicit nodes. It should also be noted that node separation may occur in (2). In addition, if we merge implicit nodes in (3-b) according to $\equiv^R_w$, not to $\equiv^R_{wa}$, then we may also perform the node separation after the merge is completed.

Merging implicit nodes. The equivalence test can be done by the next proposition.

Proposition 5. *Let $x, y \in Factor(w)$. Let X and Y, respectively, be the equivalence classes of $\overset{w}{\overrightarrow{x}}$ and $\overset{w}{\overrightarrow{y}}$ under $\equiv^R_w$. If y is a suffix of x, then $x \equiv^R_w y \Leftrightarrow X = Y$.*

Proof. Omitted.

For $x \in Factor(w)$, the equivalence class of the string $\overset{w}{\overrightarrow{x}}$ under $\equiv^R_w$ is the first encountered node in transitions from the node $[x]^R_w$ in $DAWG(w)$ which is accepting or has out-degree other than 1. Thus we could test the equivalence for two suffixes of w with the proposition.

Separating explicit nodes. For $w \in \Sigma^*$ and $a \in \Sigma$, $\equiv^R_{wa}$ is a refinement of $\equiv^R_w$. Furthermore, we have:

Proposition 6 ([1]). *Let $w \in \Sigma^*$ and $a \in \Sigma$. Let z be the longest suffix of wa that is in $Factor(w)$. Let $x \in Factor(w)$ such that x is the longest string in $[x]^R_w$. Then,*

$$[x]^R_w = \begin{cases} [x]^R_{wa} \cup [z]^R_{wa}, & \text{if } z \in [x]^R_w \text{ and } x \neq z; \\ [x]^R_{wa}, & \text{otherwise.} \end{cases}$$

This proposition gives us the condition that an equivalence class in $Factor(w)/{\equiv^R_w}$ is split into two classes under $\equiv^R_{wa}$.

3.3 Algorithm

We recall the essence of the Ukkonen algorithm briefly.

Proposition 7. *For any $x \in Factor(w)$ with $x \neq \varepsilon$, $out\text{-}deg_w(x) \leq out\text{-}deg_w(f(x))$.*

Let $u_0, u_1, \ldots, u_\ell$ be the suffixes of a string w such that $u_0 = w$, $u_\ell = \varepsilon$, and $u_{i+1} = f(u_i)$ for each $i = 0, \ldots, \ell - 1$. By Proposition 7, we can partition them into three groups:

- $u_0, \ldots, u_j$ having out-degree 0.
- $u_{j+1}, \ldots, u_k$ having out-degree 1.
- $u_{k+1}, \ldots, u_\ell$ having out-degree ≥ 2.

The suffixes in the first group are represented as the leaves of $STree_{mod}(w)$. The suffixes in the second group are not explicitly present in $STree_{mod}(w)$, whereas those in the third group are represented as nodes of $STree_{mod}(w)$. The u_{j+1} is referred to as the *active point* in [11]. The Ukkonen algorithm does nothing for the suffixes $u_0, \ldots, u_j$ in the first group in $STree(w)$ based on the idea of *open edges* (open transitions). For the suffixes $u_{j+1}, \ldots, u_\ell$, it proceeds as follows: For each $i \geq j+1$, it tests whether or not there is an *a-edge* (i.e. an edge whose label begins with a) from the node u_i, and makes a new branch labeled by a, until either it encounters a node u_r having a-edge or i exceeds ℓ. The u_r is referred to as the *end point*. This process is thus a search task for the end point. If u_r is found then the string $u_r a$ becomes the new active point. Otherwise, the new active point is ε (the root).

Our algorithm for constructing CDAWGs works in the very same way as the Ukkonen algorithm for suffix trees. It processes the input string *text* symbol by symbol from left to right, and builds $CDAWG_{mod}(text[1:i])$ from $CDAWG_{mod}(text[1:i-1])$ in the ith cycle of the algorithm. The difference can be summarized as follows.

- All the suffixes $u_0, \ldots, u_j$ in the first group are equivalent under $\equiv^R_w$, so that they can be represented by the sink $[w]^R_w$. Namely, the destinations of the open edges are all the same. Our algorithm also does nothing for the open edges when new symbol is added.
- Recall that the suffixes $u_{j+1}, \ldots, u_k$ have out-degree 1, and hence they are not represented as explicit nodes. There might be i_1, i_2 with $j+1 \leq i_1 < i_2 \leq k$ such that u_{i_1} and u_{i_2} are equivalent under $\equiv^R_w$ but implicitly lie on distinct edges. Such suffixes must be merged into one node if they become branching because of newly added symbol. This can be performed in the task of searching for the end point. The equivalence test can be carried out on the basis of Proposition 5. We have only to compute the node associated with the string $\overset{w}{\overrightarrow{u_i}}$ for every u_i with $j+1 \leq i < r$. But there is still some difficulty because not all proper nodes are present in $CDAWG_{mod}(w)$. We therefore must care about the case where more than two accepting nodes lie on the same edge implicitly. The problem, however, does not matter because we can process the suffixes in the decreasing order of their length.

– Assume strings $x, y \in \mathit{Factor}(w)$ are equivalent under $\equiv_w^R$. This can be violated by adding a new symbol a to the right of w. For this reason, a node of $\mathit{DAWG}(w)$ can be separated into two nodes in $\mathit{DAWG}(wa)$. The separation occurs only when $x \notin \mathit{Suffix}(wa)$ but $y \in \mathit{Suffix}(wa)$, so that it can be done after the end point is found. The separation procedure differs from that of construction of DAWGs in the respect that the end point and its suffixes are not necessarily represented as an explicit node.

As in [11] we represent an edge label as a pair of integers. Namely, edge $s \xrightarrow{u} t$ is represented as $s \xrightarrow{(k,p)} t$, where $u = \mathit{text}[k : p]$. We refer to an explicit or implicit node r of CDAWG by a reference pair, like in [11]. The proposed algorithm is described in Fig. 2 and Fig. 3. The function *Canonize* is borrowed from the Ukkonen algorithm, that canonizes the reference pair $(s, (k, p))$ of a (possibly implicit) node. The functions *Check_End_Point* and *Split_Edge* correspond to the function *test–and–split* in the Ukkonen algorithm. The table *Suf* represents the suffix links, and *Length* is the table that stores the length of the longest path from the source to each node.

Fig. 4 shows $\mathit{CDAWG}_{mod}(w)$ and $\mathit{CDAWG}_{mod}(wa)$ for $w = \mathit{abcabcab}$. For comparison, $\mathit{STree}_{mod}(w)$ and $\mathit{STree}_{mod}(wa)$ are also supplied. It can be observed that the implicit nodes associated with the strings *abcab*, *bcab*, and *cab* are merged into a single node, and the implicit nodes associated with the strings *ab* and *b* are also merged into another single node. As stated before, the implicit nodes are merged only when they are equivalent under $\equiv_w^R$ and become explicit due to a newly added symbol. One can observe that the accepting nodes associated with *abcab* and *ab* are on a single edge and those associated with *bcab* and *b* are on another single edge, but no confusion occurs as stated above. See Fig. 5 which illustrates well the situation.

The function *Separate_Node* in the algorithm summarizes the procedure of node separation. This is essentially the same as the separation procedure for $\mathit{DAWG}(w)$ given by Blumer et al. [1], except that we have to handle implicit nodes and implicit suffix links.

We conclude the complexity of our algorithm as follows.

Theorem 1. *The proposed algorithm runs in linear time in the length of an input string.*

Proof. The linearity proof is very similar to those for the DAWG in [1] and for the suffix tree in [11]. We divide the time requirement into two components, both turn out to be $O(n)$. The first component consists of the total time for *Canonize*. The second component consists of the rest. Let us define the *suffix chain* started at x on w, denoted by $SC_w(x)$, to be the sequence of (possibly implicit) nodes that form the path via suffix links from the (possibly implicit) node associated with x to the source in $\mathit{CDAWG}_{mod}(w)$ as in [1]. $|SC_w(x)|$ denotes the length of this sequence. Let k_1 be the number of iterations of the while loop in *Update* and let k_2 be the number of iterations of the repeat-until loop in *Separate_Node*, in updating $\mathit{CDAWG}(w)$ to $\mathit{CDAWG}(wa)$. By a similar argument in [1], we can prove $|SC_{wa}(wa)| \leq |SC_w(w)| - (k_1 + k_2) + 2$. Initially $|SC_w(w)| = 1$ because $w = \varepsilon$, and then it grows at most two (possibly

```
Algorithm Construction of CDAWG(text#)
    in alphabet Σ = {text[−1], text[−2], ..., text[−m]},
    and # is the end marker not appearing elsewhere in text.
create nodes source, sink, and ⊥;
for j := 1 to m do create a new edge ⊥ --(−j,−j)--> source;
Suf(source) := ⊥;     /* suffix link */
Length(source) := 0; Length(⊥) := −1; /* length of the longest path from source */
(s, k) := (source, 1); i := 0;
repeat
    i := i + 1;
    (s, k) := Update(s, (k, i));
until text[i] = #;

function Update(s, (k, i)): pair of integers;
/* (s, (k, i − 1)) is the canonical reference pair for the active point. */
c := text[i]; oldr := nil; e := nil;
while not Check_End_Point(s, (k, i − 1), c) do
    if k ≤ i − 1 then     /* implicit */
        if e = Extension(s, (k, i − 1)) then
            Redirect(s, (k, i − 1), r);
            (s, k) := Canonize(Suf(s), (k, i − 1));
            continue;
        else
            e := Extension(s, (k, i − 1));
            r := Split_Edge(s, (k, i − 1));
    else     /* explicit */
        r := s;
    create a new edge r --(i,∞)--> sink;
    if oldr ≠ nil then Suf(oldr) := r;
    oldr := r;
    (s, k) := Canonize(Suf(s), (k, i − 1));
if oldr ≠ nil then Suf(oldr) := s;
return Separate_Node(s, (k, i));

function Check_End_Point(s, (k, p), c): boolean;
if k ≤ p then     /* implicit */
    let s --(k',p')--> s' be the text[k]-edge from s;
    return (c = text[k' + p − k + 1]);
else     /* explicit */
    return (there is a c-edge from s);
```

Fig. 2. Main routine, function *Update*, and function *Check_End_Point*.

implicit) nodes longer in each call of *Update*. Since $|SC_w(w)|$ decreases by an amount proportional to the sum of numbers of iterations of the while loop and the repeat-until loop on each call of *Update*, this implies that the second time component is $O(n)$.

For an analysis of the first time component we have only to consider the number of iterations of the while loop in *Canonize*. Concerning the calls of *Canonize* from the while loop in *Update*, the total number of the iterations is linear by the same argument in [11]. Thus we shall consider the number of

function *Extension*$(s, (k, p))$: node;
/* $(s, (k, p))$ *is a canonical reference pair.* */
if $k > p$ **then return** s; /* *explicit* */
find the $text[k]$-edge $s \xrightarrow{(k',p')} s'$ from s; **return** s'; /* *implicit* */

function *Redirect*$(s, (k, p), r)$;
let $s \xrightarrow{(k',p')} s'$ be the $text[k]$-edge from s;
replace this edge by edge $s \xrightarrow{(k',k'+p-k)} r$;

function *Canonize*$(s, (k, p))$: pair of **integers**;
/* *identical to the one of Ukkonen's algorithm.* */
if $k > p$ **then return** (s, k); /* *explicit* */
/* $(s, (k, p))$ *is an implicit node.* */
find the $text[k]$-edge $s \xrightarrow{(k',p')} s'$ from s;
while $p' - k' \leq p - k$ **do**
  $k := k + p' - k' + 1$; $s := s'$;
  if $k \leq p$ **then** find the $text[k]$-edge $s \xrightarrow{(k',p')} s'$ from s;
return (s, k);

function *Split_Edge*$(s, (k, p))$: node;
let $s \xrightarrow{(k',p')} s'$ be the $text[k]$-edge from s;
replace this edge by edges $s \xrightarrow{(k',k'+p-k)} r$ and $r \xrightarrow{(k'+p-k+1,p')} s'$,
where r is a new node;
$Length(r) := Length(s) + (p - k + 1)$;
return r;

function *Separate_Node*$(s, (k, p))$: pair of **integers**;
$(s', k') := Canonize(s, (k, p))$;
if $k' \leq p$ **then return** (s', k'); /* *implicit* */
/* $(s', (k', p))$ *is an explicit node.* */
if $Length(s') = Length(s) + (p - k + 1)$ **then return** (s', k'); /* *solid edge* */
/* *non-solid case* */
create a new node r' as a duplication of s';
$Suf(r') := Suf(s')$; $Suf(s') := r'$;
$Length(r') := Length(s) + (p - k + 1)$;
repeat
  replace the $text[k]$-edge from s to s' by edge $s \xrightarrow{(k,p)} r'$;
  $(s, k) := Canonize(Suf(s), (k, p - 1))$;
until $(s', k') \neq Canonize(s, (k, p))$;
return $(r', p + 1)$;

Fig. 3. Other functions.

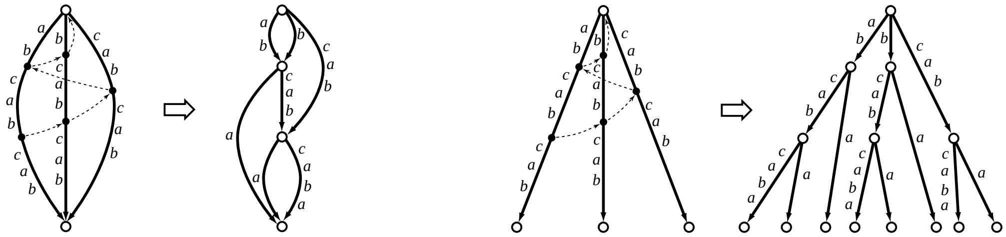

Fig. 4. The left two graphs are $CDAWG_{mod}(w)$ and $CDAWG_{mod}(wa)$, and the right two graphs are $STree_{mod}(w)$ and $STree_{mod}(wa)$ for $w = abcabcab$. The white circles and the solid arrows mean the explicit nodes and the edges, respectively. The small black points on the solid arrows are implicit nodes. The broken arrows are the suffix links represented implicitly.

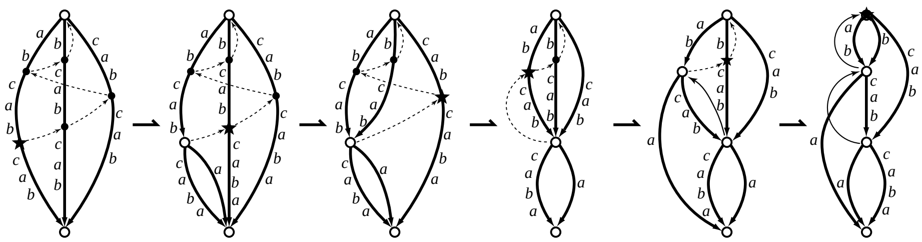

Fig. 5. Detailed conversion of $CDAWG_{mod}(w)$ into $CDAWG_{mod}(wa)$ for $w = abcabcab$. The star $\star$ expresses the focused points. The thin solid arrows are the suffix links represented explicitly.

iterations of the while loop in *Canonize* called from *Separate_Node*. There are two cases to consider. When the end point is explicit, the call of *Canonize* is executed in constant time. Consider the case where the end point is implicit. Notice that this case occurs only when the active point is identical to the end point. Hence the active point has advanced along the edge that it lies on without transitions via suffix links reading text symbols. The total number of iterations of the while loop of *Canonize* in each call of *Separate_Node* is at most the length of the edge on which the end point lies. This can be bounded by the text length in total. □

4 Conclusion

We developed an on-line algorithm for constructing CDAWGs. It is a generalization of Ukkonen's algorithm designed for suffix trees. In actual fact, our algorithm builds the suffix tree for an input string if we replace the 13th line of the function *Update* in Fig. 2 by

create a new edge $r \xrightarrow{(i,\infty)} r'$ where r' is a new node;

With this slight alteration, the behavior of our algorithm becomes completely the same as that of Ukkonen's algorithm. Also, DWAGs can be constructed with altering the 13th line as follows.

create a new edge $r \xrightarrow{(i,i)} sink$;

In this case, an additional maintenance to update the sink is necessary. Furthermore, our algorithm can construct suffix tires with following alteration for the 13th line.

create a new edge $r \xrightarrow{(i,i)} r'$ where r' is a new node;

Similarly to the case of DAWGs, we need to append a maintenance to update the leaves in a suffix tree.

It is slight to add these two maintenances into the algorithm. Recall the definitions of suffix tries, suffix trees, DAWGs, and CDAWGs, which are displayed in Section 2. Then, notice that the above alteration for every structure totally corresponds to the definition for it. We now have a unified view of suffix tires, suffix trees, DAWGs, and CDAWGs, basing on our general algorithm.

The suffix tree and the DAWG achieve the linear space complexity on the basis of the equivalence relations $\equiv_w^L$ and $\equiv_w^R$ on Σ^*, respectively. The CDAWG is also based on an equivalence relation that is the transitive closure of $\equiv_w^L \cup \equiv_w^R$ [2]. Not only the CDAWG is attractive as indexing structure, but also the underlying equivalence relation is useful in data mining or machine discovery from textual databases. In fact, the equivalence relation played a central role in supporting human experts who involved in evaluation/interpretation task for mined expressions from anthologies of classical Japanese poems [10].

References

1. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.
2. A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987. Preliminary version in: STOC'84.
3. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, 1994.
4. M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In A. Apostolico and J. Hein, editors, *Proc. 8th Annual Symposium on Combinatorial Pattern Matching*, volume 1264 of *Lecture Notes in Computer Science*, pages 116–129. Springer-Verlag, 1997.
5. M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, volume 1261 of *Lecture Notes in Computer Science*, pages 192–211. Springer-Verlag, 1997.
6. R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19:331–353, 1997.
7. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
8. J. Holub and B. Melichar. Approximate string matching using factor automata. *Theoretical Computer Science*, 249:305–311, 2000.
9. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, Apr. 1976.
10. M. Takeda, T. Matsumoto, T. Fukuda, and I. Nanri. Discovering characteristic expressions from literary works: A new text analysis method beyond n-gram and KWIC. In S. Arikawa and S. Morishita, editors, *Proc. 3rd International Conference on Discovery Science*, volume 1967 of *Lecture Notes in Artificial Intelligence*, pages 112–126. Springer-Verlag, 2000.
11. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.