Space-Economical Construction of Index Structures for All Suffixes of a String

Shunsuke Inenaga[1], Ayumi Shinohara[1,2], Masayuki Takeda[1,2], Hideo Bannai[3], and Setsuo Arikawa[1]

[1] Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan,
[2] PRESTO, Japan Science and Technology Corporation (JST),
{s-ine,ayumi,takeda,arikawa}@i.kyushu-u.ac.jp
[3] Human Genome Center, University of Tokyo, Tokyo 108-8639, Japan,
bannai@ims.u-tokyo.ac.jp

Abstract. The *minimum all-suffixes directed acyclic word graph* (*MASDAWG*) of a string w has $|w|+1$ initial nodes, where the dag induced by all reachable nodes from the k-th initial node conforms with the DAWG of the k-th suffix of w. A new space-economical algorithm for the construction of $MASDAWG(w)$ is presented. The algorithm reads a given string w from right to left, and constructs $MASDAWG(w)$ *without suffix links*. It performs in time linear in the output size. Furthermore, we introduce the *minimum all-suffixes compact DAWG* (*MASCDAWG*). CDAWGs are known to be more space-economical than DAWGs, and thus $MASCDAWG(w)$ requires smaller space than $MASDAWG(w)$. We present an on-line (right-to-left) algorithm to build $MASCDAWG(w)$ without suffix links, whose running time is also linear in its size.

1 Introduction

Pattern matching on strings is one of the most fundamental and important problems in Theoretical Computer Science. When a pattern is flexible and a text is fixed, the problem can be solved in time proportional to the length of the pattern by using a suitable *index structure*.

An example of widely explored patterns is the *variable-length-don't-care pattern* (*VLDC-pattern*) which includes a symbol $\star$, a *wildcard* matching any string. Formally, when Σ is an alphabet, a VLDC-pattern is an element of set $(\Sigma \cup \{\star\})^*$. For example, $a\star ab\star$ is a VLDC-pattern, where $a, b \in \Sigma$. VLDC-patterns are sometimes called *regular patterns* as in [11]. The language of a VLDC-pattern (or a regular pattern) is the set of strings obtained by replacing $\star$'s in the pattern by arbitrary strings. This language corresponds to a class of the *pattern languages* proposed in [1].

The smallest automaton to recognize all VLDC-patterns matching a given text string was introduced in [8]. It is essentially the same structure as the minimum dag representing all substrings of every suffix of a string, which is called the *minimum all-suffixes directed acyclic word graph* (*MASDAWG*). The MASDAWG for a string w is the minimization of the DAWGs for all suffixes of

K. Diks et al. (Eds): MFSC 2002, LNCS 2420, pp. 341–352, 2002.

w. It has $|w| + 1$ initial nodes, in which the dag induced by all reachable nodes from the k-th initial node conforms with the DAWG of the k-th suffix of w. Some applications of MASDAWGs were presented in [8].

The size of the DAWG for a string w is $O(|w|)$ [2]. This implies that the total size of the DAWGs of all suffixes of w is $O(|w|^2)$. Hence, the MASDAWG for w can be constructed in $O(|w|^2)$ time by minimizing the DAWGs [10]. On the other hand, it has been proven that the size of the MASDAWG of w is $\Theta(|w|^2)$ [8]. The direct construction of MASDAWGs that avoids the creation of redundant nodes and edges is therefore important, considering the reduction of space requirements. The first algorithm to directly build the MASDAWG of a string was given in [8]. It performs in *on-line* manner, that is, it processes a given string from left to right, a character by a character, and converts the MASDAWG of w to the MASDAWG of wa.

The algorithm of [8] can efficiently construct MASDAWGs by means of *suffix links*, kinds of failure transitions, like most linear-time algorithms constructing index structures (e.g., see [13,9,12,2,3,5,7,4,6]). On the other hand, it is also the fact that the memory space required by suffix links is non-ignorable. Moreover, for each node, the algorithm additionally requires to keep the length of the longest string that reaches to the node, in the construction phase. These values are unnecessary in order to examine whether a given pattern occurs or not in the specified suffix. In this paper, we present a new algorithm to construct MASDAWGs *without suffix links nor length information*, which thus permits us to save memory space. The algorithm is best understood as one constructing MASDAWGs in 'right-to-left' on-line manner. Namely, it builds the MASDAWG of aw by adding some nodes and edges to the MASDAWG of w.

Furthermore, we aim to reduce the space requirement by *compacting* the structure itself. We focus on the *compact DAWG* (*CDAWG*) whose space requirement is strictly smaller than that of the DAWG, both theoretically and practically [3,5]. Its all-suffixes version, named the *minimum all-suffixes CDAWG* (*MASCDAWG*), is introduced in this paper. We also present an on-line (right-to-left) algorithm to construct the MASCDAWG in linear time with respect to its size, without using suffix links nor length information.

2 Minimum All-Suffixes Directed Acyclic Word Graphs

Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of string $w = xyz$, 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 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 $w[i:] = w[i:|w|]$ for $1 \leq i \leq |w| + 1$. Assume S is a subset of Σ^*. For any string $u \in \Sigma^*$, $u^{-1}S = \{x \mid ux \in S\}$.

Let $w \in \Sigma^*$. We define an equivalence relation $\equiv_w$ on Σ^* by

$$x \equiv_w y \Leftrightarrow x^{-1}\mathit{Suffix}(w) = y^{-1}\mathit{Suffix}(w).$$

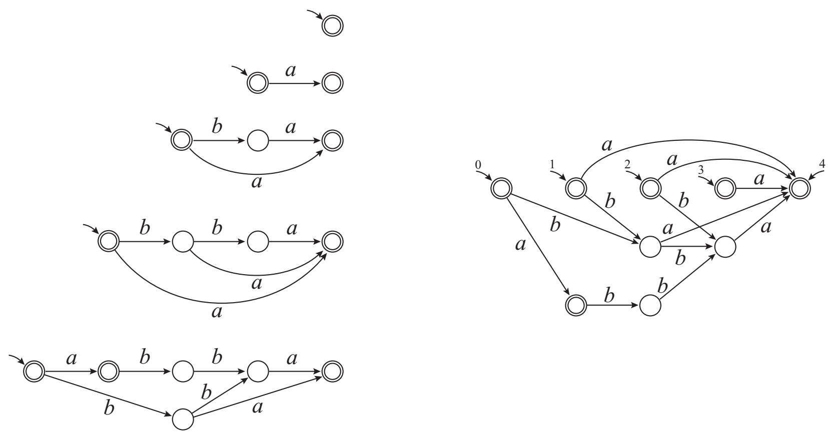

Fig. 1. The naive $ASDAWG(w)$ is shown on the left, where $w = abba$. $MASDAWG(w)$ is displayed on the right.

Let $[x]_w$ denote the equivalence class of a string $x \in \Sigma^*$ under $\equiv_w$. The longest element in the equivalence class $[x]_w$ for $x \in Factor(w)$ is called its *representative*.

Definition 1. *$DAWG(w)$ is the dag (V, E) such that*

$$V = \{[x]_w \mid x \in Factor(w)\},$$
$$E = \{([x]_w, a, [xa]_w) \mid x, xa \in Factor(w), a \in \Sigma\}.$$

Definition 2. *$ASDAWG(w)$ is a kind of dag with $|w| + 1$ initial nodes, designated by $0, 1, \ldots, |w|$, in which the subgraph consisting of the nodes reachable from the k-th initial node and their out-going edges is $DAWG(w[k + 1 :])$.*

The simple collection of $DAWG(w[1 :])$, $DAWG(w[2 :])$,..., $DAWG(w[n])$, $DAWG(w[n + 1 :])$ ($n = |w|$) is an example of $ASDAWG(w)$, referred to as the *naive* $ASDAWG(w)$. The number of nodes of the naive $ASDAWG(w)$ is $O(|w|^2)$. By minimizing the naive $ASDAWG(w)$, we can obtain the *minimum* $ASDAWG(w)$, which is denoted by $MASDAWG(w)$. The naive $ASDAWG(abba)$ and $MASDAWG(abba)$ are shown in Fig. 1. The minimization is performed based on the equivalence relation defined as follows. Each node of $ASDAWG(w)$ is represented by a pair $\langle u, [x]_u \rangle$ with $u \in Suffix(w)$ and $x \in Factor(u)$. The equivalence relation, denoted by $\sim_w$, is defined by

$$\langle u, [x]_u \rangle \sim_w \langle v, [y]_v \rangle \Leftrightarrow x^{-1} Suffix(u) = y^{-1} Suffix(v).$$

A node of $MASDAWG(w)$ corresponds to an equivalence class under $\sim_w$. We write $\langle u, [x]_u \rangle$ simply as $\langle u, [x] \rangle$ in case no confusion occurs.

Theorem 1 ([8]). *When $|\Sigma| \geq 2$, the number of nodes of $MASDAWG(w)$ for a string w is $\Theta(|w|^2)$. It is $\Theta(|w|)$ for a unary alphabet.*

Proposition 1 ([8]). *Let $u \in Suffix(w)$. Let x be a nonempty factor of u. We factorize u as $u = hxt$ and assume h is the shortest such string. Then, $\langle hxt, [x] \rangle$ is equivalent to $\langle sxt, [x] \rangle$ for every suffix s of h. (NOTE: The string x is not necessarily the representative of $[x]_u$.)*

Let $h_0, h_1, \ldots, h_r$ be the suffixes of the string h arranged in the decreasing order of their length. The above proposition implies the existence of the chain of equivalent nodes $\langle h_0xt, [x] \rangle, \langle h_1xt, [x] \rangle, \ldots, \langle h_rxt, [x] \rangle$.

Lemma 1 ([8]). *Let $h \in \Sigma^+$ and $u, hu \in Suffix(w)$. If a node of $DAWG(u)$ is equivalent to some node of $DAWG(hu)$, then it is also equivalent to some node of $DAWG(au)$ where a is the right-most character of the string h.*

The above lemma guarantees that the DAWGs sharing a node of $MASDAWG(w)$ are 'consecutive'. We can therefore concentrate on the relation between two consecutive DAWGs.

From now on, we consider what happens when constructing $MASDAWG(au)$ from $MASDAWG(u)$. Due to Lemma 1, we only investigate the relationship between $DAWG(au)$ and $DAWG(u)$.

Lemma 2. *Let $a \in \Sigma$ and $u \in \Sigma^*$. For any string $x \in Factor(u) - Prefix(au)$, it holds that $\langle au, [x] \rangle \sim_{au} \langle u, [x] \rangle$.*

Proof. $x^{-1}Suffix(au) = x^{-1}(\{au\} \cup Suffix(u)) = x^{-1}\{au\} \cup x^{-1}Suffix(u) = x^{-1}Suffix(u)$, because $x^{-1}\{au\} = \emptyset$ for $x \notin Prefix(au)$. $\square$

The above lemma implies that we have only to care about the prefixes of au in order to construct $MASDAWG(au)$ from $MASDAWG(u)$. We need not modify nor change the structure of $MASDAWG(u)$: it is kept static.

Lemma 3. *Let $a \in \Sigma$ and $u \in \Sigma^*$. For any $x \in Prefix(u)$ and $y \in \Sigma^*$, if $\langle au, [ax] \rangle \sim_{au} \langle u, [y] \rangle$ then $[x]_u = [y]_u$.*

Proof. Since $x \in Prefix(u)$, there exists $s \in \Sigma^*$ such that $u = xs$. By the assumption, $(ax)^{-1}Suffix(au) = y^{-1}Suffix(u)$. Since s is included in the left set, s is also included in the right set, i.e. $s \in y^{-1}Suffix(u)$, which implies $ys \in Suffix(xs)$, thus $y \in Suffix(x)$. We have two cases according to $x \in Prefix(au)$.

(Case 1) When $x \in Prefix(au)$. Since $x \in Prefix(axs)$, $x = a^i$ and $y = a^j$ for some integers $j \leq i$. Suppose $j < i$, and let $k = i - j > 0$. Then $a^k s \in y^{-1}Suffix(u)$ while $a^k s \notin (ax)^{-1}Suffix(au)$, that contradicts with the assumption that $(ax)^{-1}Suffix(au) = y^{-1}Suffix(u)$. Thus $j = i$, which yields $y = x = a^i$.

(Case 2) When $x \notin Prefix(au)$.

$$\begin{aligned} y^{-1}Suffix(u) &= (ax)^{-1}Suffix(au) && \text{by the assumption} \\ &\subseteq x^{-1}Suffix(au) && \text{since } x \in Suffix(ax) \\ &= x^{-1}Suffix(u) && \text{since } x \notin Prefix(au) \\ &\subseteq y^{-1}Suffix(u) && \text{since } y \in Suffix(x) \end{aligned}$$

Thus we have $x^{-1}Suffix(u) = y^{-1}Suffix(u)$, that is, $[x]_u = [y]_u$. $\square$

The path in $MASDAWG(u)$ spelling out u is called its 'backbone'. The above lemma shows that if a node $\langle au, [ax]\rangle$ on the 'backbone' of $MASDAWG(au)$ is equivalent to a node of $MASDAWG(u)$, the node $\langle au, [ax]\rangle$ is also on the 'backbone' of $MASDAWG(u)$. This fact is crucial in order that our algorithm, which will be given in the sequel, performs in time linear in the size of $MASDAWG(u)$.

For the prefixes of string au, we have the following lemma.

Lemma 4. *Let $a \in \Sigma$ and $u \in \Sigma^*$. Let $ax \in Prefix(au)$ be the shortest string which satisfies $\langle au, [ax]\rangle \sim_{au} \langle u, [x]\rangle$. Then for any longer prefix $axv \in Prefix(au)$, it holds that $\langle au, [axv]\rangle \sim_{au} \langle u, [xv]\rangle$.*

Proof. Since $\langle au, [ax]\rangle \sim_{au} \langle u, [x]\rangle$, $(ax)^{-1}Suffix(au) = x^{-1}Suffix(u)$. Thus, $(axv)^{-1}Suffix(au) = v^{-1}((ax)^{-1}Suffix(au)) = v^{-1}(x^{-1}Suffix(u)) = (xv)^{-1}Suffix(u)$. □

Remark that the node $\langle u, [xv]\rangle$ already exists in $MASDAWG(u)$, since $xv \in Prefix(u)$. Thus the above lemma guarantees that all nodes we have to newly create in $MASDAWG(au)$ are $\langle au, [t]\rangle$ for strings $t \in Prefix(z)$, where z is the longest prefix of au which does *not* satisfy $\langle au, [ax]\rangle \sim_{au} \langle u, [x]\rangle$. Now the next question is how to *efficiently* check whether $\langle au, [ax]\rangle \sim_{au} \langle u, [x]\rangle$ or not for each $x \in Prefix(u)$. Our idea is to count the cardinality of the set $x^{-1}Suffix(u)$.

Lemma 5. *Let $a \in \Sigma$ and $u \in \Sigma^*$. For any $x \in Factor(u)$, $\langle au, [ax]\rangle \sim_{au} \langle u, [x]\rangle$ if and only if $|(ax)^{-1}Suffix(au)| = |x^{-1}Suffix(u)|$.*

Proof. We first show that $(ax)^{-1}Suffix(au) \subseteq x^{-1}Suffix(u)$. Let us choose $s \in (ax)^{-1}Suffix(au)$ arbitrarily. Then $axs \in Suffix(au) = \{au\} \cup Suffix(u)$. If $axs = au$, then $xs = u$. Otherwise, $axs \in Suffix(u)$. Since xs is a suffix of axs, we know that xs is also a suffix of u. In both cases, we have $xs \in Suffix(u)$, which implies that $s \in x^{-1}Suffix(u)$. Thus $(ax)^{-1}Suffix(au) \subseteq x^{-1}Suffix(u)$. It yields that $(ax)^{-1}Suffix(au) = x^{-1}Suffix(u)$ if and only if $|(ax)^{-1}Suffix(au)| = |x^{-1}Suffix(u)|$. By the definition of $\sim_{au}$, we have proved the lemma. □

We associate each node $\langle u, [x]\rangle$ with the cardinality of the set, $|x^{-1}Suffix(u)|$, denoted by $\#\langle u, [x]\rangle$. Note that $\#\langle u, [u]\rangle = 1$ since $u^{-1}Suffix(u) = \{\varepsilon\}$, and that $\#\langle u, [\varepsilon]\rangle = |u| + 1$ since $\varepsilon^{-1}Suffix(u) = Suffix(u)$.

Lemma 6. *Let $a \in \Sigma$ and $u \in \Sigma^*$. For any $x \in Prefix(u)$, $\#\langle au, [ax]\rangle = \#\langle u, [ax]\rangle + 1$.*

Proof. Since $x \in Prefix(u)$, $\#\langle au, [ax]\rangle = |(ax)^{-1}Suffix(au)| = |(ax)^{-1}(\{au\} \cup Suffix(u))| = |(ax)^{-1}\{au\} \cup (ax)^{-1}Suffix(u))| = \#\langle u, [ax]\rangle + 1$. □

The whole algorithm is shown in Fig. 2. Since the algorithm manipulates an input string w from right to left, we number the characters in w as $w = w_n w_{n-1} \dots w_1$. An edge is represented by a triple (r, w_i, s), where s, r are nodes and w_i is the character for the label of the edge.

Theorem 2. *For any string $w \in \Sigma^*$, our algorithm constructs $MASDAWG(w)$ in time linear in its size.*

The on-line (right-to-left) construction of $MASDAWG(w)$ where $w = abaa\$$ is displayed in Fig. 3.

```
Algorithm Construction of MASDAWG(w = w_n w_{n-1} ... w_1).
  create new nodes s_0;
  #(s_0) := 1;    #(nil) := 0;
  initNode[0] := s_0;  node := s_0;
  for i := 1 to n do
      s := Find(node, w_i);
      target := NewTargetNode(s, i − 1, node);
      newNode := create a new node with copying all out-going edges of node;
      add or overwrite edge (newNode, w_i, target);
      #(newNode) := i;
      initNode[i] = newNode;
      node = newNode;

function NewTargetNode(Node s, int j, Node backbone) : Node
  nextNumSuf := #(s) + 1;
  if nextNumSuf = #(backbone) then return backbone;    /* redirection */
  nextBackbone := Find(backbone, w_j);
  newNode := create a new node with copying all out-going edges of s;
  s := Find(s, w_j);
  target := NewTargetNode(s, j − 1, nextBackbone);
  add or overwrite edge (newNode, w_j, target);
  #(newNode) := nextNumSuf;
  return newNode;

function Find(Node s, char c) : Node
  if s has the c-edge then
      let (s, c, r) be the c-edge from s;
      return r;
  else return nil;
```

Fig. 2. The algorithm to construct $MASDAWG(w)$.

3 Minimum All-Suffixes Compact Directed Acyclic Word Graphs

To achieve a more space-economical index structure for all suffixes of a string, we turn our attention to a compact directed acyclic word graph (CDAWG) and consider its all-suffixes version.

Assume S is a subset of Σ^*. For any string $u \in \Sigma^*$, $Su^{-1} = \{x \mid xu \in S\}$. Let $w \in \Sigma^*$. We define an equivalence relation $\equiv'_w$ on Σ^* by

$$x \equiv'_w y \Leftrightarrow \mathit{Prefix}(w)x^{-1} = \mathit{Prefix}(w)y^{-1}.$$

Let $[x]'_w$ denote the equivalence class of a string $x \in \Sigma^*$ under $\equiv'_w$. The longest element in the equivalence class $[x]'_w$ for $x \in \mathit{Factor}(w)$ is also called its *representative*, and is denoted by $\overset{w}{\overrightarrow{x}}$. For any string $x \in \mathit{Factor}(w)$, there uniquely exists string $\alpha \in \Sigma^*$ such that $\overset{w}{\overrightarrow{x}} = x\alpha$.

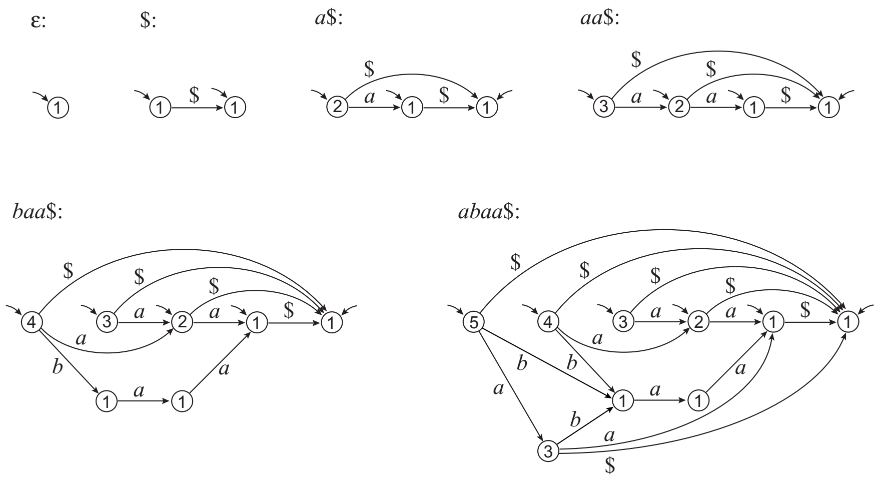

Fig. 3. Construction of $MASDAWG(abaa\$)$. Each node is marked by $\#\langle u, [x]\rangle$ where $u = abaa\$$ and $x \in Factor(u)$.

Proposition 2. *Let $x \in Factor(w)$. Assume $\overset{w}{\overrightarrow{x}} \notin Suffix(w)$. Then, x occurs in w at least twice.*

Proof. For a contradiction, assume x occurs in w only once. Then, we have $|Prefix(w)x^{-1}| = 1$. Let $w = hxy$. Since x occurs in w only once, $|Prefix(w)x^{-1}| = |Prefix(w)(xy)^{-1}|$. Thus $x \equiv'_w xy$ and $\overset{w}{\overrightarrow{x}} = xy$. However, $xy \in Suffix(w)$, a contradiction. Consequently, x appears in w at least twice. $\square$

Definition 3. *$CDAWG(w)$ is the dag (V, E) such that*

$$V = \{[\overset{w}{\overrightarrow{x}}]_w \mid x \in Factor(w)\},$$

$$E = \{([\overset{w}{\overrightarrow{x}}]_w, a\beta, [\overset{w}{\overrightarrow{xa}}]_w) \mid x, xa \in Factor(w),\ a \in \Sigma,\ \beta \in \Sigma^*,\ \overset{w}{\overrightarrow{xa}} = xa\beta,\ \overset{w}{\overrightarrow{x}} \neq \overset{w}{\overrightarrow{xa}}\}.$$

The following corollary derives from Lemma 2.

Corollary 1. *Assume that w terminates with a unique symbol \$. Then, for any string $x \in Factor(w) - Suffix(w)$, node $[\overset{w}{\overrightarrow{x}}]_w$ is of out-degree more than one.*

Namely, $CDAWG(w)$ is the compaction of $DAWG(w)$ where any nodes of out-degree one are removed and their edges are modified accordingly.

Definition 4. *$ASCDAWG(w)$ is a kind of dag with $|w| + 1$ initial nodes, designated by $0, 1, \ldots, |w|$, in which the subgraph consisting of the nodes reachable from the k-th initial node and their out-going edges is $CDAWG(w[k + 1 :])$.*

We now introduce the minimized version of $ASCDAWG(w)$, which is well defined similarly to MASDAWGs. Each node of $ASCDAWG(w)$ can be represented by a pair $\langle u, [\overset{u}{\overrightarrow{x}}]_u \rangle$ with $u \in \mathit{Suffix}(w)$ and $x \in \mathit{Factor}(u)$. We write $\langle u, [\overset{u}{\overrightarrow{x}}]_u \rangle$ simply as $\langle u, [\overrightarrow{x}] \rangle$ when no confusion occurs. If $\langle u, [\overset{u}{\overrightarrow{x}}]_u \rangle \sim_w \langle v, [\overset{v}{\overrightarrow{y}}]_v \rangle$, we merge these nodes and the resulting structure is the *minimum ASCDAWG*(w), denoted by *MASCDAWG*(w).

Theorem 3. *When* $|\Sigma| \geq 2$*, the number of nodes in MASCDAWG*(w) *for a string* w *is* $\Theta(|w|^2)$*. It is* $\Theta(|w|)$ *for a unary alphabet.*

Here, we have only to consider a string $x \in \mathit{Factor}(w)$ such that $\overset{w}{\overrightarrow{x}} = x$. Since Proposition 1 and Lemma 1 hold for an arbitrary string in $\mathit{Factor}(w)$, it is guaranteed that the CDAWGs sharing a node in *MASCDAWG*(w) are also 'consecutive'. Therefore, we only consider the relationship between *CDAWG*(au) and *CDAWG*(u), two consecutive CDAWGs.

Lemma 7. *Let* $a \in \Sigma$ *and* $u \in \Sigma^*$*. For any string* $x \in \mathit{Factor}(u) - \mathit{Prefix}(au)$*,* $\overset{u}{\overrightarrow{x}} = \overset{au}{\overrightarrow{x}}$*.*

Proof. Since $x \notin \mathit{Prefix}(au)$, there is no new occurrence of x in au. It implies that $a(\mathit{Prefix}(u)x^{-1}) = \mathit{Prefix}(au)x^{-1}$. Thus we have $[x]'_u = [x]'_{au}$. Consequently, $\overset{u}{\overrightarrow{x}} = \overset{au}{\overrightarrow{x}}$. □

The above lemma ensures that any implicit node of *CDAWG*(u) does not become explicit in *CDAWG*(au) if it is not associated with a prefix of au. It follows from this lemma and Lemma 2 that we do not need to modify nor change the structure of *MASCDAWG*(u) when constructing *MASCDAWG*(au).

Lemma 8. *Let* $a \in \Sigma$ *and* $u \in \Sigma^*$*. For any* $x, z \in \mathit{Factor}(u)$*, if* $\overset{au}{\overrightarrow{ax}} = az$ *then* $\overset{u}{\overrightarrow{z}} = z$*.*

Proof. Suppose contrarily that $\overset{u}{\overrightarrow{z}} \neq z$. That means there exists $y \in \Sigma^*$ such that $\mathit{Prefix}(u)y^{-1} = \mathit{Prefix}(u)z^{-1}$ and $|y| > |z|$. Then $\mathit{Prefix}(au)(ay)^{-1} = (\mathit{Prefix}(au)y^{-1})a^{-1} = (a(\mathit{Prefix}(u)y^{-1}))a^{-1} = (a(\mathit{Prefix}(u)z^{-1}))a^{-1} = \mathit{Prefix}(au)(az)^{-1} = \mathit{Prefix}(au)(ax)^{-1}$. Thus $ay \equiv'_{au} ax$ and $|ay| > |az|$. It contradicts the assumption $\overset{au}{\overrightarrow{ax}} = az$. □

Lemma 9. *Let* $a \in \Sigma$ *and* $u \in \Sigma^*$*. For any* $x \in \mathit{Prefix}(u)$ *and* $y \in \Sigma^*$ *satisfying* $\langle au, [\overset{au}{\overrightarrow{ax}}]_{au} \rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{y}}]_u \rangle$*, there exists* $z \in \mathit{Prefix}(u)$ *such that* $[\overset{u}{\overrightarrow{z}}]_u = [\overset{u}{\overrightarrow{y}}]_u$*.*

Proof. Let z be the string with $\overset{au}{\overrightarrow{ax}} = az$. Then we have $\overset{u}{\overrightarrow{z}} = z$ by Lemma 8. Moreover, $z \in \mathit{Prefix}(u)$ since $x \in \mathit{Prefix}(u)$. Since $\langle au, [az]_{au} \rangle = \langle au, [\overset{au}{\overrightarrow{ax}}]_{au} \rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{y}}]_u \rangle$, we have $[z]_u = [\overset{u}{\overrightarrow{y}}]_u$ by Lemma 3. Thus $[\overset{u}{\overrightarrow{z}}]_u = [\overset{u}{\overrightarrow{y}}]_u$. □

Lemma 9 shows that if node $\langle au, [\overset{au}{\overrightarrow{ax}}]_{au} \rangle$ on the 'backbone' of *MASCDAWG*(au) is equivalent to a node of *MASCDAWG*(u), the node $\langle au, [\overset{au}{\overrightarrow{ax}}]_{au} \rangle$ is also on the 'backbone' of *MASCDAWG*(u). It corresponds to Lemma 3.

We have the following lemma which corresponds to Lemma 4.

Lemma 10. *Let $a \in \Sigma$ and $u \in \Sigma^*$. Let $ax \in Prefix(au)$. Let $\overset{au}{\overrightarrow{ax}}$ be the shortest string for which there exists $z \in Prefix(u)$ such that $\langle au, [\overset{au}{\overrightarrow{ax}}]_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{z}}]_u\rangle$. Let $\overset{au}{\overrightarrow{ax}} = ay$. Then for any longer prefix $ayv \in Prefix(au)$, there exists $s \in Prefix(u)$ such that $\langle au, [\overset{au}{\overrightarrow{ayv}}]_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{s}}]_u\rangle$.*

Proof. Let $\overset{au}{\overrightarrow{ayv}} = as$. By Lemma 8, $\overset{u}{\overrightarrow{s}} = s$. Since $yv \in Prefix(u)$, $s \in Prefix(u)$. Let $\overset{u}{\overrightarrow{z}} = t$. By the assumption $\langle au, [\overset{au}{\overrightarrow{ax}}]_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{z}}]_u\rangle$, we have $\langle au, [ay]\rangle \sim_{au} \langle u, [t]\rangle$. Since $y \in Prefix(u)$, $\langle au, [ay]\rangle \sim_{au} \langle u, [y]\rangle$ by Lemma 3. Note that $y \in Prefix(s)$. Hence we have $\langle au, [as]\rangle \sim_{au} \langle u, [s]\rangle$ by Lemma 4. Because $as = \overset{au}{\overrightarrow{ayv}}$ and $s = \overset{u}{\overrightarrow{s}}$, it holds that $\langle au, [\overset{au}{\overrightarrow{ayv}}]_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{s}}]_u\rangle$. □

We remark that the equivalence $\langle au, [\overset{au}{\overrightarrow{ax}}]_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{z}}]_u\rangle$ can also be examined by checking the cardinalities of the corresponding sets, as is the case of MASDAWGs. Hereby we have shown that *MASCDAWG*(w) can be constructed in a similar way to *MASDAWG*(w). The only thing not clarified yet is whether or not *MASCDAWG*(w) can be built in time linear in its size. We establish the following lemmas to support the linearity.

Lemma 11. *Let $a \in \Sigma$ and $w \in \Sigma^*$. For any $x, z \in Factor(w)$, if $\overset{w}{\overrightarrow{ax}} = az$ then $\overset{w}{\overrightarrow{z}} = z$.*

Proof. For a contradiction, assume $\overset{w}{\overrightarrow{z}} \neq z$. Then there exists $y \in \Sigma^*$ such that $Prefix(w)y^{-1} = Prefix(w)z^{-1}$ and $|y| > |z|$. Then $Prefix(w)(ay)^{-1} = (Prefix(w)y^{-1})a^{-1} = (Prefix(w)z^{-1})a^{-1} = Prefix(w)(az)^{-1}$. Thus $ay \equiv'_{au} az$ and $|ay| > |az|$. It contradicts the assumption $\overset{w}{\overrightarrow{ax}} = az$. □

Note that the statement of the above lemma slightly differs from that of Lemma 8.

Lemma 12. *Let $a, b \in \Sigma$ and $w \in \Sigma^*$. Let $x, y \in Factor(w)$ such that $\overset{w}{\overrightarrow{xb}} = xby \neq w$. If $axb \in Factor(w)$, then $axby \in Factor(w)$, and $\overset{w}{\overrightarrow{axby'}} = \overset{w}{\overrightarrow{axby}}$ for any $y' \in Prefix(y)$.*

Proof. Since $axb \in Factor(w)$ and $xby \neq w$, there always exists $z \in \Sigma^*$ such that $\overset{w}{\overrightarrow{axb}} = axbz \in Factor(w)$. By Lemma 11, $\overset{w}{\overrightarrow{xbz}} = xbz$. Since $\overset{w}{\overrightarrow{xb}} = xby$, $y \in Prefix(z)$. Because $axbz \in Factor(w)$, $axby \in Factor(w)$. For any $y' \in Prefix(y)$, $axbz \equiv'_w axby'$ since $\overset{w}{\overrightarrow{axb}} = axbz$. Therefore $\overset{w}{\overrightarrow{abxy'}} = abxz = \overset{w}{\overrightarrow{abxy}}$. □

Suppose $\overset{w}{\overrightarrow{x}} = x$. If we in advance know node $[\overset{w}{\overrightarrow{x}}]_w$ has an out-going edge labeled with by, we can avoid to scan the whole string xby in traversing the path $axby$ from the initial node of *CDAWG*(w). Moreover, it is guaranteed that the path by from the (explicit or implicit) node for ax consists of one edge: no explicit node is contained in the path. This is a key to achieve an algorithm that constructs *MASCDAWG*(w) in linear time with respect to its size.

The whole algorithm is shown in Fig. 4. Here we also read an input string w from right to left, and thus w is written as $w = w_n w_{n-1} \dots w_1$. The label

```
Algorithm Construction of MASCDAWG(w = w_n w_{n-1} ... w_1).
  create new nodes s_0,s_1,s_2;
  #(s_0) := 1;  #(s_1) := 1;  #(s_2) := 2;  #(nil) := 0;
  endpos(s_0) := 0;  endpos(s_1) := 1;  endpos(s_2) := 2;  endpos(nil) := 0;
  add edges (s_1, 1, s_0), (s_2, 1, s_0), (s_2, 2, s_0);
  initNode[0] := s_0;  initNode[1] := s_1;  initNode[2] := s_2;  node := s_2;
  for i := 3 to n do
      (s, k, p, r) := Canonize(FastFind(node, i, 1));
      target := NewTargetNode((s, k, p, r), i − 1, node);
      newNode := create a new node with copying all out-going edges of node;
      add or overwrite edge (newNode, i, target);
      #(newNode) := i;    endpos(newNode) := i;
      initNode[i] = newNode;
      node = newNode;

function NewTargetNode(refQuartet (s, k, p, r), int j, Node backbone) : Node
  nextNumSuf := #(r) + 1;
  if nextNumSuf = #(backbone) then return backbone;    /* redirection */
  let (backbone, ℓ, nextBackbone) be the w_j-edge from backbone;
  m := ℓ − endpos(nextBackbone);    /* length of this edge */
  if k = p then    /* explicit node */
      newNode := create a new node with copying all out-going edges of s;
      (s, k, p, r) := Canonize(FastFind(s, j, m));
      target := NewTargetNode((s, k, p, r), j − m, nextBackbone);
      add or overwrite edge (newNode, j, target);
      #(newNode) := nextNumSuf;    endpos(newNode) := j;
      return newNode;
  else if w_p = w_j then /* implicit and next characters are the same */
      (s, k, p, r) := Canonize(s, k, p − m, r);    /* skip m characters */
      return NewTargetNode((s, k, p, r), j − m, nextBackbone);
  else    /* implicit and next characters are different */
      newNode := create a new node;    /* edge split */
      add new edges (newNode, p, r) and (newNode, j, s_0);
      #(newNode) := nextNumSuf;    endpos(newNode) := j;
      return newNode;

function FastFind(Node s, int i, int length) : refQuartet
/* compute the position from s along the string w_i w_{i-1} ... w_{i-length+1} */
/* remark that the first character w_i is only compared */
  if s has the w_i-edge then
      let (s, ℓ, r) be the w_i-edge from s;
      return (s, ℓ, ℓ − length, r);
  else return (s, i, i − length, nil);

function Canonize( refQuartet (s, k, p, r) ) : refQuartet
/* when the referenced position is an explicit node, canonize the expression */
  if k > p and p = endpos(r) then return (r, p, p, r);
  else return (s, k, p, r);
```

Fig. 4. The algorithm to construct *MASCDAWG*(w).

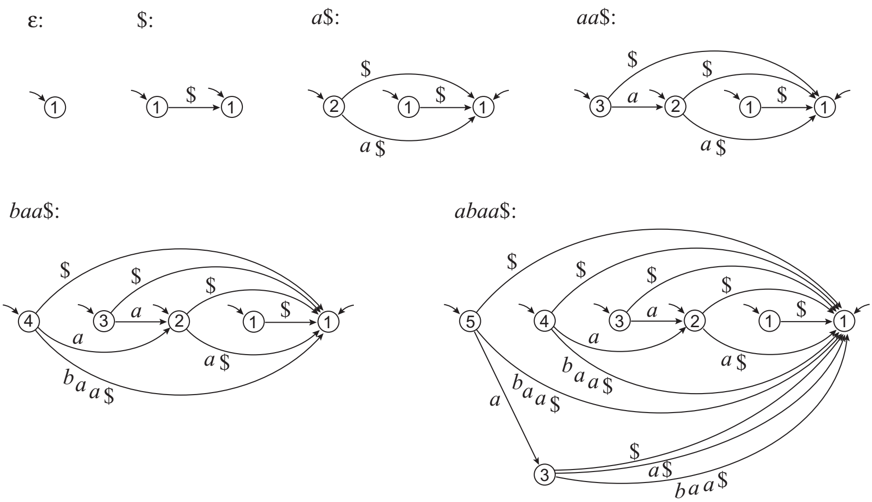

Fig. 5. Construction of *MASCDAWG*(*abaa*$).

$w_i w_{i-1} \dots w_j$ of each edge can be represented by a pair of the beginning position i and the ending position $j-1$. $(i > j-1)$ If the string corresponding to the label appears in w more than once, we represent it by the leftmost occurrence. This way we can assign *endpos*(s) to a node s, where *endpos*(s) indicates the ending position of every in-coming edge of s. Thereby, we represent each edge by a triple (r, i, s), where r, s are explicit nodes. An implicit node corresponding to some factor x of w can be represented by a triple (r, k, p), where r is an explicit parent node of the implicit node. Assuming the representative of the equivalence class associated with r is y, $x = yu$ where $u = w_k w_{k-1} \dots w_p$. The quartet (r, k, p, s) is called the *reference quartet*, where s is the closest explicit child node of r reachable via the w_k-edge from r. When $|p-k|$ is minimum, the quartet (r, k, p, s) is called the *canonical reference quartet.*

Theorem 4. *For any string $w \in \Sigma^*$, our algorithm constructs MASCDAWG(w) in time linear in its size.*

The on-line (right-to-left) construction of *MASCDAWG*(w) where $w = abaa\$$ is displayed in Fig. 5.

4 Concluding Remarks

We proposed a new space-economical algorithm to construct MASDAWGs without suffix links, running in time linear in the output size. As shown in [8], there are several important applications for MASDAWGs. Therefore, reducing memory space needed in the construction of MASDAWGs is considerably significant. We have also accomplished further reduction of the space requirement, by introducing the MASCDAWG and its construction algorithm, which runs in linear time with respect to the size of the structure.

It is easy to construct the *minimum all-suffixes suffix trie* in time proportional to its size, by a slightly modified algorithm for the MASDAWG. We only need to care not to merge subtrees of the same suffix trie, so that the resulting structure does *not* become a dag. Similarly, the *minimum all-suffixes suffix tree* can also be built in time linear to its size, by modifying the algorithm for the MASCDAWG.

References

1. D. Angluin. Finding patterns common to a set of strings. *J. Comput. Sys. Sci.*, 21:46–62, 1980.
2. 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.
3. 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.
4. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, 1994.
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. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.
7. S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. In A. Amir and G. M. Landau, editors, *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, 2001.
8. S. Inenaga, M. Takeda, A. Shinohara, H. Hoshino, and S. Arikawa. The minimum dawg for all suffixes of a string and its applications. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, 2002. (to appear).
9. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
10. D. Revuz. Minimization of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.
11. T. Shinohara. Polynomial-time inference of pattern languages and its applications. In *Proc. 7th IBM Symp. Math. Found. Comp. Sci.*, pages 191–209, 1982.
12. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
13. P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.