Sparse Directed Acyclic Word Graphs

Shunsuke Inenaga[1,2] and Masayuki Takeda[2,3]

[1] Japan Society for the Promotion of Science
[2] Department of Informatics, Kyushu University, Fukuoka 812-8581, Japan
{shunsuke.inenaga, takeda}@i.kyushu-u.ac.jp
[3] SORST, Japan Science and Technology Agency (JST)

Abstract. The suffix tree of string w is a text indexing structure that represents all suffixes of w. A sparse suffix tree of w represents only a subset of suffixes of w. An application to sparse suffix trees is composite pattern discovery from biological sequences. In this paper, we introduce a new data structure named *sparse directed acyclic word graphs* (*SDAWGs*), which are a sparse text indexing version of directed acyclic word graphs (DAWGs) of Blumer et al. We show that the size of SDAWGs is linear in the length of w, and present an on-line linear-time construction algorithm for SDAWGs.

1 Introduction

Text indexing is a classical technique for pattern matching. Probably, the most widely known structure for text indexing is suffix trees [1, 2]. Indeed, quite a lot of applications for suffix trees have been introduced so far, and many problems are efficiently solved by using suffix trees [3]. Some of those problems require 'variants' of suffix trees that are modified for the specific purposes. This paper focuses on one of such problems, named the *sparse text indexing problem*, described as follows:

Input: Pattern string p, text string t of length n, and subset $S \subseteq \{1, \ldots, n\}$ of all positions in t.

Output: Whether or not there is any occurrence of p in t which begins at a position in S.

Kärkkäinen and Ukkonen [4] introduced the *sparse suffix tree* which stores only the suffixes of t beginning at the positions in S. Sparse suffix trees enable us to solve the above problem in time proportional to the pattern length (for fixed alphabets). An example of applications to sparse suffix trees is composite pattern discovery from biological sequences [5, 6, 7]. It is not difficult to see that sparse suffix trees can be constructed in $O(n)$ time and space, in such a way that we firstly construct a normal suffix tree of t in $O(n)$ time, and then prune the leaves for the suffixes which do *not* begin with positions in S. Now, a natural interest is whether or not the sparse suffix tree for t w.r.t. S is directly constructible in $O(n)$ time using $O(k)$ space, where $k = |S|$. (Note that $O(n)$ time consumption is unavoidable due to the necessity of reading entire t at least once.) To the best of our knowledge, this is still an open problem.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 61–73, 2006.
© Springer-Verlag Berlin Heidelberg 2006

However, another representation of S and some simple alteration to t make it possible to construct its sparse suffix tree efficiently. Let # be a unique symbol not appearing anywhere in t, and let us insert # into t at the positions listed in S. Now we get a string of length $n + k$, and let us denote this string by w. Remark that since $k \leq n$, w is at most twice long as t. Now, if we are able to construct a sparse suffix tree representing only the suffixes of w which begin immediately after #, then this tree is an alternative to the sparse suffix tree for t. At a matching phase of pattern p we simply ignore any #'s in the edge labels of the tree.

This tree is also known as the *word suffix tree* whose concept was first introduced in [8]. Let Σ be an alphabet, and let $D = \Sigma^*\#$ be a dictionary of words over Σ, each followed by #. Now assume that w is a string in D^+, namely, w is a sequence $w_1 \cdots w_k$ of k words in D. Then, the word suffix tree of w w.r.t. D is a tree structure which represents only the k suffixes in the form of $w_i \cdots w_k$. We can associate the special symbol # with, e.g., a 'word separator' such as the blank symbol in the European languages, and then the word suffix tree of w represents only the suffixes of w starting at the beginning of words. In this way, we can avoid unwanted matches such as 'other' in 'mother'. Andersson et al. [9] introduced an algorithm to build the word suffix tree for w w.r.t. D with $O(k)$ space, but in $O(n)$ *expected* time. Very lately, we invented an algorithm that constructs word suffix trees with $O(k)$ space and in $O(n)$ time *in the worst cases* [10].

In this paper, we introduce a new data structure named *sparse directed acyclic word graphs* (*SDAWGs*) as an alternative to the word suffix trees and to the sparse suffix trees thereby. SDAWGs are a sparse text indexing version of *directed acyclic word graphs* (*DAWGs*) of Blumer et al. [11]. We give a formal definition of SDAWGs based on an equivalence relation on string w and dictionary D, and show the size of SDAWGs to be linear in n. One might concern that $O(n)$ space consumption is a disadvantage of SDAWGs against sparse suffix trees requiring only $O(k)$ space, but we recall that the edge labels of sparse suffix trees are implemented as pairs of pointers to the positions of w, and therefore the input string w has to be kept stored. Consequently, the total space requirement is also $O(n)$ for using the sparse suffix tree. On the other hand, any edge label of SDAWGs is a single symbol, and thus the input string w can be discarded after the SDAWG is constructed.

Finally, we present an on-line linear-time algorithm for constructing SDAWGs. Our algorithm is based on, and generalizes, the on-line construction algorithm for DAWGs invented by Blumer et al. [11]. Our algorithm directly constructs SDAWGs without building normal DAWGs as intermediate structures, by using the minimum DFA accepting dictionary D as suggested in [10]. We emphasize that SDAWGs can be obtained by first constructing the corresponding normal DAWGs, removing the unnecessary suffixes from the DAWGs, and then minimizing the resulting graphs. However, since these are non-tree DAGs, removing only one suffix may take linear time. Therefore, the algorithm presented in this paper is the only known one capable of building SDAWGs in $O(n)$ time.

2 Preliminaries

2.1 Notations

Let Σ be a finite set of symbols, called an *alphabet*. Throughout this paper we assume that Σ is fixed. A finite sequence of symbols is called a *string*. We denote the length of string w by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let Σ^* be the set of strings over Σ. For any symbol $a \in \Sigma$, we define a^{-1} such that $a^{-1}a = \varepsilon$.

Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. A prefix, substring, and suffix of string w are said to be *proper* if they are not w. Let $\mathit{Prefix}(w)$ be the set of the prefixes of string w, and let $\mathit{Prefix}(S) = \bigcup_{w \in S} \mathit{Prefix}(w)$ for set S of strings.

Definition 1 (Prefix property). *A set L of strings is said to satisfy the* prefix property *if no string in L is a proper prefix of another string in L.*

The i-th symbol of string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of 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 any strings $x, w \in \Sigma^*$, let

$$\mathit{Endpos}_w(x) = \{j \mid x = w[j - |x| + 1..j]\}.$$

Let D be a set of strings called a *dictionary*. A *factorization* of string w w.r.t. D is a list $w_1, \ldots, w_k$ of strings in D such that $w = w_1 \cdots w_k$ and $w_i \in D$ for each $1 \leq i \leq k$. In the rest of the paper, we assume that $D = \Sigma^*\#$ where $\#$ is a special symbol not belonging to Σ, and that $w \in D^+$. Then, a factorization of w w.r.t. D is always unique, since D clearly satisfies the prefix property because of $\#$ not being in Σ. Let M_D denote the minimum DFA which accepts $D = \Sigma^*\#$. It is easy to see that M_D requires only constant space (refer to the left of Fig. 2).

Let

$$\mathit{Suffix}_D(w) = \{w_i \cdots w_k \mid 1 \leq i \leq k+1\}.$$

Then, $\mathit{Suffix}_D(w)$ consists only of the original string w, the suffixes which immediately follow $\#$ in w, and the empty string ε intended by $w_{k+1}w_k$. We define set $\mathit{Wordpos}_D(w)$ of the word-starting positions in w as follows:

$$\mathit{Wordpos}_D(w) = \{|w| - |s| + 1 \mid s \in \mathit{Suffix}_D(w)\}.$$

2.2 Equivalence Class on Strings over D

For set S of integers and integer i, we denote $S \oplus i = \{j + i \mid j \in S\}$ and $S \ominus i = \{j - i \mid j \in S\}$. Now we define the end-equivalence relation $\equiv_w$ on $w \in D^+$ by:

$$x \equiv_w y \Leftrightarrow \mathit{Endpos}_w(x) \cap (\mathit{Wordpos}_D(w) \oplus |x| \ominus 1) = \mathit{Endpos}_w(y) \cap (\mathit{Wordpos}_D(w) \oplus |y| \ominus 1).$$

We note that the above end-equivalence relation is a 'word-position-sensitive' version of the equivalence relation introduced by Blumer et al. [11], where the intersection with $Wordpos_D(w)$ makes it word-position-sensitive. We denote by $[x]_w$ the equivalence class of x w.r.t. $\equiv_w$.

Proposition 1. *All strings that are not in $Prefix(Suffix_D(w))$ form one equivalence class under $\equiv_w$, called the* degenerate class.

Proof. Since for any string $x \notin Prefix(Suffix_D(w))$ we have $Wordpos_D(w) = \emptyset$, we consequently obtain $Endpos_w(x) \cap (Wordpos_D(w) \oplus |x| \ominus 1) = \emptyset$. Moreover, for any string $y \in Prefix(Suffix_D(w))$, it is easy to observe that $Endpos_w(y) \cap (Wordpos_D(w) \oplus |x| \ominus 1) \neq \emptyset$. □

It follows from the definition of $\equiv_w$ that if two strings x, y are in a same non-degenerate equivalence class under $\equiv_w$, then either x is a suffix of y, or vice versa. Thus, each non-degenerate equivalence class under $\equiv_w$ has a unique longest member, which is called the *representative* of it. The representative of $[x]_w$ is denoted by $\overset{w}{\overleftarrow{x}}$.

3 Sparse Directed Acyclic Word Graphs

3.1 Definitions

Here we define the *sparse directed acyclic word graphs* (*SDAWGs* in short) as edge-labeled DAGs (V, E) with $E \subseteq V \times \Sigma^+ \times V$ where the second component of each edge represents its label.

Definition 2 (Sparse directed acyclic word graph). *The* sparse directed acyclic word graph *of string $w \in D^+$, denoted by $SDAWG_D(w)$, is a DAG (V, E) such that*

$$V = \{[x]_w \mid x \in Prefix(Suffix_D(w))\},$$
$$E = \{([x]_w, a, [xa]_w) \mid x, xa \in Prefix(Suffix_D(w)) \text{ and } a \in \Sigma \cup \{\#\}\}.$$

$SDAWG_D(w)$ has single *source node* $[\varepsilon]_w$ of in-degree zero, and single *sink node* $[w]_w$ of out-degree zero.

We associate each node $[x]_w$ of $SDAWG_D(w)$ with $length([x]_w) = |\overset{w}{\overleftarrow{x}}|$. For any edge $([x]_w, a, [xa]_w)$, if $length([xa]_w) = length([x]_w) + 1$, this edge is called *primary*; otherwise, it is called *secondary*.

Fig. 1 shows $SDAWG_D(w)$ with $w = \mathtt{a\#b\#a\#bab\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}^*\#$, together with the normal $DAWG(w)$ representing all the suffixes of w. Observe that $SDAWG_D(w)$ only represents the suffixes $\mathtt{a\#b\#a\#bab\#}$, $\mathtt{b\#a\#bab\#}$, $\mathtt{a\#bab\#}$, $\mathtt{bab\#}$, and ε, all from $Suffix_D(w)$.

Also, observe that substrings $\mathtt{a\#b}$ and $\mathtt{b}$ are in distinct nodes of $DAWG(w)$, while they are in the same node of $SDAWG_D(w)$. It is because $Endpos_w(\mathtt{a\#b}) = \{3, 7\} \neq Endpos_w(\mathtt{b}) = \{3, 7, 9\}$, but $Endpos_w(\mathtt{a\#b}) \cap (Wordpos_D(w) \oplus 2) =$

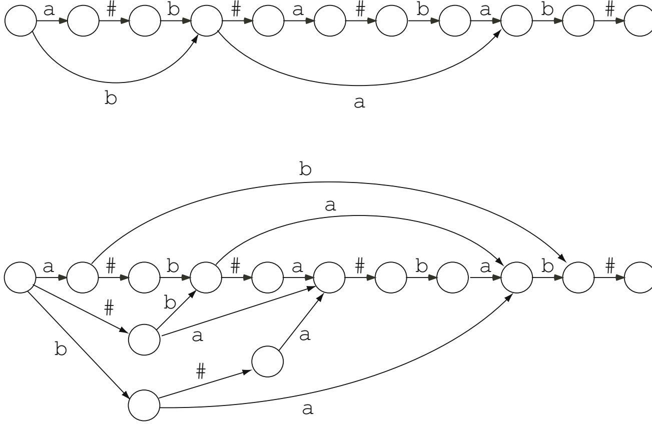

Fig. 1. $SDAWG_D(w)$ with $w = \mathtt{a\#b\#a\#bab\#}$ and $D = \{\mathtt{a},\mathtt{b}\}^*\#$ is shown on the upper, and normal $DAWG(w)$ is shown on the lower for comparison. Observe that $SDAWG_D(w)$ contains only suffixes of $Suffix_D(w)$, while $DAWG(w)$ has all the suffixes of $Suffix(w)$. For instance, $\mathtt{ab\#}$ is a suffix of w and is in normal $DAWG(w)$, but is not in $SDAWG_D(w)$.

$Endpos_w(\mathtt{b}) \cap (Wordpos_D(w) \oplus 0) = \{3, 7\}$, as $Wordpos_D(w) = \{1, 3, 5, 7\}$. Similar discussion holds for the pair of strings $\mathtt{a\#b\#}$ and $\mathtt{b\#}$.

Now we define the *suffix links* of $SDAWG_D(w)$, which are extensively used for on-line linear-time construction algorithm to be given later on. Also, they play an important role to bound the size of $SDAWG_D(w)$ within $O(n)$ space. For any string $x \in Prefix(Suffix_D(w))$, we consider a partition $x = x_1x_2$ such that $x_1 \in D^*$ and x_2 is a *proper* prefix of some string in D. Then, it is easy to see that the partition x_1x_2 is unique for any $x \in Prefix(Suffix_D(w))$.

The following proposition is clear from the definition of the end-equivalence.

Proposition 2. *For any $x, y \in Prefix(Suffix_D(w))$ such that $x \equiv_w y$ and $|x| > |y|$, we have $x_1 = vy_1$ with $v \in D^+$ and $x_2 = y_2$.*

Definition 3 (Suffix links of SDAWGs). *For any node $[x]_w$ of $SDAWG_D(w)$, let $x' = x'_1x'_2$ be the shortest member of $[x]_w$.*

1. *If $x'_1 \in D^+$, the suffix link from node $[x]_w$ goes to node $[u]_w$ such that $\overset{w}{\overleftarrow{u}} = u = u_1u_2$, $u_1 \in D^*$, $u_2 = x'_2$, and $x'_1 = hu_1$ for some $h \in D$;*
2. *Otherwise (If $x'_1 = \varepsilon$), the suffix link from $[x]_w$ goes to the initial state of M_D.*

Fig. 2 displays $SDAWG_D(w)$ and its suffix links, with $w = \mathtt{a\#b\#a\#bab\#}$. For instance, see Node 8 that is $[x]_w = \{\mathtt{a\#b\#a\#b}, \mathtt{b\#a\#b}\}$, where $x' = \mathtt{b\#a\#b}$, $x'_1 = \mathtt{b\#a\#}$, and $x'_2 = \mathtt{b}$. The suffix link of Node 8 goes to Node 4 that is $[u]_w = \{\mathtt{a\#b}, \mathtt{b}\}$, where $\overset{w}{\overleftarrow{u}} = \mathtt{a\#b}$, $u_1 = \mathtt{a\#}$, and $u_2 = \mathtt{b}$. Observe that $h = \mathtt{b\#}$, $x'_1 = hu_1 = \mathtt{b\#a\#}$ and $x'_2 = u_2 = \mathtt{b}$.

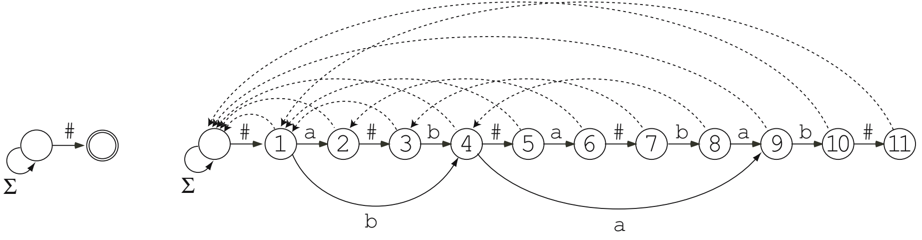

Fig. 2. To the left is the minimum DFA M_D accepting $D = \Sigma^*\#$, and to the right is $SDAWG_D(w)$ for $w = \texttt{a\#b\#a\#bab\#}$, with M_D and its suffix links (broken arrows) attached. Nodes 3, 5, 6, 7, 8, and 11 are in Group 1 of Definition 3, and nodes 1, 2, 4, 9, and 10 are in Group 2.

3.2 Size Bound

Here we analyze the size of $SDAWG_D(w)$. Firstly, we show that any distinct prefixes of w are associated with distinct nodes of $SDAWG_D(w)$.

Lemma 1. *For any strings $x, xa \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$, $x \not\equiv_w xa$.*

Proof. By the length argument, $|x| \in Endpos_w(x)$ but $|x| \notin Endpos_w(xa)$. Since $1 \in Wordpos_D(w)$ for any $w \in D^+$, $|x| \in Endpos_w(x) \cap (Wordpos_D(w) \oplus |x| \ominus 1)$ but $|x| \notin Endpos_w(xa) \cap (Wordpos_D(w) \oplus |xa| \ominus 1)$. Thus we have $x \not\equiv_w xa$. □

According to the above lemma, $SDAWG_D(w)$ has at least $n+1$ nodes, each corresponding to a certain prefix of w. In addition, for any proper prefix x of w, there exists primary edge $([x]_w, a, [xa]_w)$ in $SDAWG_D(w)$ with $xa \in Prefix(w)$.

To show the upper bound for the size of $SDAWG_D(w)$, we consider the suffix link tree $T_D(w) = (V \cup \{q_s\}, E_\ell)$ where q_s is the initial state of M_D and is the root of $T_D(w)$, and E_ℓ is the set of the 'reversed' suffix links of $SDAWG_D(w)$.

The following lemma is critical to bound the size of $SDAWG_D(w)$ within linear space w.r.t. n.

Lemma 2. *If $\overset{w}{\overleftarrow{x}} \notin Prefix(w)$, node $[x]_w$ of $T_D(w)$ is branching (has at least two children).*

Proof. Since $\overset{w}{\overleftarrow{x}} \notin Prefix(w)$, there exist some distinct strings $u, v \in D$ such that

- both $u\overset{w}{\overleftarrow{x}}$ and $v\overset{w}{\overleftarrow{x}}$ are substrings of w,
- $Endpos_w(u\overset{w}{\overleftarrow{x}}) \cap (Wordpos_D(w) \oplus |u\overset{w}{\overleftarrow{x}}| \ominus 1) \neq \emptyset$,
- $Endpos_w(v\overset{w}{\overleftarrow{x}}) \cap (Wordpos_D(w) \oplus |v\overset{w}{\overleftarrow{x}}| \ominus 1) \neq \emptyset$, and
- $Endpos_w(u\overset{w}{\overleftarrow{x}}) \cap (Wordpos_D(w) \oplus |u\overset{w}{\overleftarrow{x}}| \ominus 1) \neq$
  $Endpos_w(v\overset{w}{\overleftarrow{x}}) \cap (Wordpos_D(w) \oplus |v\overset{w}{\overleftarrow{x}}| \ominus 1)$.

Then, no two strings of $u\overset{w}{\overleftarrow{x}}$, $v\overset{w}{\overleftarrow{x}}$, or $\overset{w}{\overleftarrow{x}}$ belong to the same end-equivalence class. By Proposition 2 and Definition 3, the suffix links of $[u\overset{w}{\overleftarrow{x}}]_w$ and $[v\overset{w}{\overleftarrow{x}}]_w$ both go to $[\overset{w}{\overleftarrow{x}}]_w = [x]_w$. □

Now we show the upper bound of the size of SDAWGs, based on a similar idea to the case of DAWGs by Blumer et al. [11].

Theorem 1. *For any string $w \in D^+$ of length n, $SDAWG_D(w)$ has $O(n)$ nodes and edges.*

Proof. By Lemmas 1 and 2, $T_D(w)$ can have at most $n + 1$ leaves which correspond to the nodes having the prefixes of w. Since $T_D(w)$ is a tree, it can have at most n branching nodes, and therefore the total number of nodes in $SDAWG_D(w)$ is bounded by $O(n)$.

Now we bound the number of edges in $SDAWG_D(w)$. It is not difficult to see that for any $w \in D^+$, $SDAWG_D(w)$ has a spanning tree rooted at the source node $[\varepsilon]_w$, and let us fucus on one such spanning tree. With each edge of $SDAWG_D(w)$ *not* in the spanning tree, we associate one of the $k-1$ non-empty proper suffixes of $Suffix_D(w)$. This suffix can be obtained by spelling out a path from the source through the spanning tree until one of its leaves, across the omitted edge, and finally to the sink node $[w]_w$ in any convenient way. Then, distinct omitted edges are associated with distinct non-empty suffixes of $Suffix_D(w)$, since they are associated with distinct source-to-sink paths (the paths differ in the first edge traversed outside the spanning tree). Thus, the number of edges not in the spanning tree is bounded by $k-1$, and the total number of edges in $SDAWG_D(w)$ by $O(n)$. □

One might concern that Theorem 1 suggests a disadvantage of the SDAWGs against the sparse (word) suffix trees which have only $O(k)$ nodes and edges, but we recall that the edge labels of those suffix trees are implemented as pairs of pointers to the positions of w. To do so, the input string w has to be kept stored and therefore the total space requirement for using those suffix trees is also $O(n)$. On the other hand, any edge label of SDAWGs is a single symbol from $\Sigma \cup \{\#\}$, and therefore the input string w can be discarded after $SDAWG_D(w)$ is constructed.

3.3 On-Line Linear-Time Construction Algorithm

In this section we present our on-line linear-time construction algorithm for SDAWGs. Since our algorithm is on-line, it sequentially processes the input string $w \in D^+$ from left to right. To discuss this on-line construction, we extend the definition of $Suffix_D(u)$ to any prefix u of $w \in D^+$, as follows. For any prefix u of $w = w_1 \dots w_k \in D^+$ such that $u = w_1 \cdots w_\ell v$, $1 \le \ell < k$, and v is a proper prefix of $w_{\ell+1}$, let $u_i = w_i \cdots w_\ell v$. For convenience, let $u_{\ell+1} = v$ and $u_{\ell+2} = \varepsilon$. Now, let

$$Suffix_D(u) = \{u_i \mid 1 \le i \le \ell + 2\}.$$

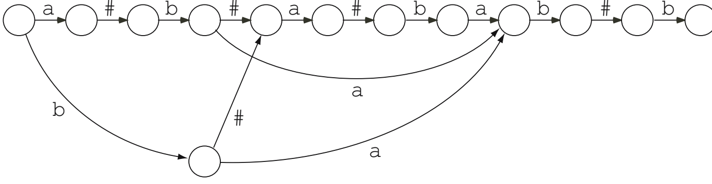

Fig. 3. The SDAWG for string a#b#a#bab#b w.r.t. $D = \Sigma^*\#$. Compare this with the SDAWG for string a#b#a#bab# w.r.t. D in Fig. 1 (upper).

Then, the definitions of $Wordpos_D(u)$, the end-equivalence relation $\equiv_u$, and $SDAWG_D(u)$ are naturally extended to any prefix u of w.

The following proposition and lemma state how to update the nodes of $SDAWG_D(u)$ when we read a new symbol a and construct $SDAWG_D(ua)$.

Proposition 3. *Let $w \in D^+$ and $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$. Then,*

$$Wordpos_D(ua) = \begin{cases} Wordpos_D(u) \cup \{|ua|\}, & \text{if } u[|u|] = \#; \\ Wordpos_D(u), & \text{otherwise.} \end{cases}$$

Also, for any string $x \in (\Sigma \cup \{\#\})^*$,

$$Endpos_{ua}(x) = \begin{cases} Endpos_u(x) \cup \{|ua|\}, & \text{if } x \in Suffix(ua); \\ Endpos_u(x), & \text{otherwise.} \end{cases}$$

Lemma 3. *Let $w \in D^+$, and let $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$. Let z be the longest string in $Suffix_D(ua) \cap Prefix(Suffix_D(u))$. Then, for any $x \in Prefix(Suffix_D(u))$, we have*

$$[x]_u = \begin{cases} [\overset{u}{\overleftarrow{x}}]_{ua} \cup [z]_{ua}, & \text{if } z \in [x]_u \text{ and } z \neq \overset{u}{\overleftarrow{x}}; \\ [x]_{ua}, & \text{otherwise.} \end{cases}$$

Proof (Sketch). By Proposition 3, it is not difficult to see that only if $z \in [x]_u$ and $z \neq \overset{u}{\overleftarrow{x}}$, it happens that $[x]_u \neq [x]_{ua}$. In any other cases, we have $[x]_u = [x]_{ua}$. By Proposition 3, for any string $s \in [x]_u$ with $|s| > |z|$, $Endpos_{ua}(s) = Endpos_u(s)$, and for any string $t \in [x]_u$ with $|t| \leq |z|$, $Endpos_{ua}(t) = Endpos_u(t) \cup \{|ua|\}$. No matter if $Wordpos_D(ua) = Wordpos_D(u) \cup \{|ua|\}$ or $Wordpos_D(ua) = Wordpos_D(u)$, we have $[x]_u = [\overset{u}{\overleftarrow{x}}]_{ua} \cup [z]_{ua}$. □

To see a concrete example of the above lemma, compare $SDAWG_D(u)$ of Fig. 1 (upper) and $SDAWG_D(u\mathtt{b})$ of Fig. 3, where u = a#b#a#bab#. Observe that $z = \mathtt{b}$. Now, node $[\mathtt{a\#b}]_u$ of $SDAWG_D(u)$ is split when it is updated to $SDAWG_D(u\mathtt{b})$, as follows:

$$[\mathtt{a\#b}]_u = \{\mathtt{a\#b}, \mathtt{b}\} = \{\mathtt{a\#b}\} \cup \{\mathtt{b}\} = [\mathtt{a\#b}]_{u\mathtt{b}} \cup [\mathtt{b}]_{u\mathtt{b}}.$$

For any other nodes $[x]_u$, we have $[x]_u = [x]_{u\mathtt{b}}$.

```
Input:   w = w[1..n] ∈ D⁺ and M_D with initial state q_s and final state q_f.
Output:  SDAWG_D(w).
{
    length(q_f) = 0;   length(q_s) = −1;
    source = q_f;   link(source) = q_s;
    sink = source;
    for (i = 1; i ≤ n; i++)  sink = Update(sink, i);
}

node Update(sink, i) {
    c = w[i];
    create new node newsink;   length(newsink) = i;
    create new edge (sink, c, newsink);
    for (s = link(sink); no c-edge from s; s = link(s))
        create new edge (s, c, newsink);
    s' = SplitNode(s, c);
    link(newsink) = s';
    return newsink;
}

node SplitNode(s, c) {
    let s' be the head of the c-edge from s;
    if (length(s') == length(s) + 1) return s';
    create node r' as a duplication of s' with the out-going edges;
    link(r') = link(s');   link(s') = r';
    length(r') = length(s) + 1;
    do {
        replace edge (s, c, s') by edge (s, c, r');
        s = link(s);
    } while the head of the c-edge from s is s';
    return r';
}
```

Fig. 4. SDAWG construction algorithm. For any node v, $link(v)$ indicates the node to which the suffix link of v goes. Only the initialization steps using M_D is different from the normal DAWG construction algorithm by Blumer et al. [11].

Fig. 4 shows a pseudo code of our on-line algorithm to build SDAWGs, with the help of the DFA M_D and the suffix links of Definition 3. The only difference between our algorithm and the algorithm of Blumer et al. [11] for constructing normal DAWGs is the initialization steps of the main routine where we set the source of the SDAWG to the final state q_f of M_D and the suffix link of the source to the initial state q_s of M_D. These simple modifications make a difference in the resulting data structures. In Fig. 5 we illustrate on-line construction of $SDAWG_D(w)$ with $w = \mathtt{ab\#b\#ba\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}\#$.

We remark that our algorithm generalizes the normal DAWG construction algorithm of Blumer et al. [11]. Assume just for now $D = \Sigma$, and consider a DFA which accepts Σ with only two states that are a single initial state and a

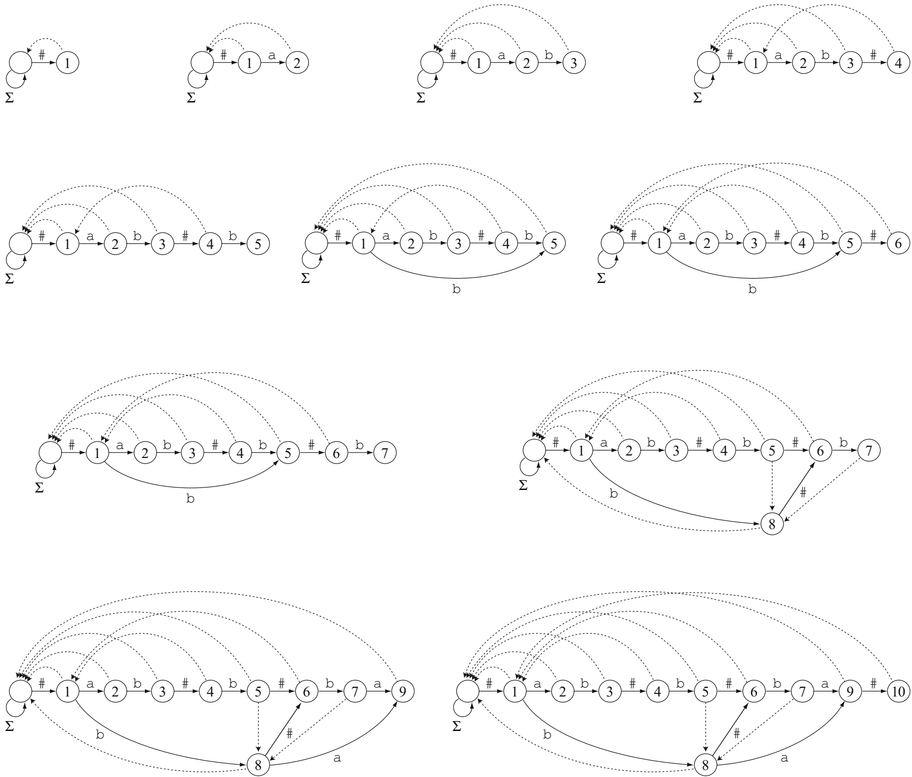

Fig. 5. A snapshot of on-line construction of $SDAWG_D(w)$ with $w = \mathtt{ab\#b\#ba\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}\#$. The broken arrows represent suffix links. The update from $SDAWG_D(\mathtt{ab\#})$ to $SDAWG_D(\mathtt{ab\#b})$ is shown in two rounds, as two new edges are created here. Also, the update from $SDAWG_D(\mathtt{ab\#b\#})$ to $SDAWG_D(\mathtt{ab\#b\#b})$ is shown in two rounds, as a node is here split into two nodes.

single final state. Then this DFA plays the same role as the auxiliary '⊥' node used in Ukkonen's on-line suffix tree construction algorithm [12], and this alters our algorithm so that it builds normal DAWGs.

Theorem 2. *The algorithm of Fig. 4 correctly constructs $SDAWG_D(w)$ for any string $w \in D^+$.*

To establish the above correctness theorem, we show the following claim:

Claim. Let $w \in D^+$ and $w_1, \ldots, w_k$ be a unique factorization of w w.r.t. D. Let $u = w_1 \cdots w_\ell v$ be the prefix of w of length j, where v is a proper prefix of $w_{\ell+1}$. After the j-th call of the *Update* function, we have $SDAWG_D(u)$ representing $Suffix_D(u)$ together with the suffix links of Definition 3.

Proof. By induction on $j = |u|$. When $|u| = 0$, the lemma trivially holds. We now consider $|u| > 0$. Let $u_i = w_i \cdots w_\ell v$ for $1 \le i \le \ell$, and for convenience, let

$u_{\ell+1} = v$, $u_{\ell+2} = \varepsilon$ and $u_{\ell+3} = c^{-1}$. For the induction hypothesis, assume that, after the j-th call of the *Update* function, we have $SDAWG_D(u)$ representing

$$Suffix_D(u) = \{u_i \mid 1 \le i \le \ell + 2\}.$$

At the $(j+1)$-th call of *Update*, due to Lemmas 1 and 3, $sink = [u]_u = [u]_{uc}$ and thus $newsink$ is created as node $[uc]_{uc}$, together with edge $(sink, c, newsink)$ $= ([u]_{uc}, c, [uc]_{uc})$. Now let h $(1 < h \le \ell + 3)$ be the smallest integer satisfying $w_h \cdots w_\ell vc = u_h c \in Prefix(Suffix_D(u))$. Note that such h always exists, since $u_{\ell+3}c = c^{-1}c = \varepsilon$ is always in $Prefix(Suffix_D(u))$. Then, $u_h c$ is the longest element of $Suffix_D(uc)$ represented by $SDAWG_D(u)$. In the iteration of the **for** loop, we traverse the suffix links starting from $sink$, each time creating edge $([u_i]_{uc}, c, [uc]_{uc})$ for $i = 2, \ldots, h-1$. (Note that for some consecutive i's, the strings u_i may belong to the same end-equivalence class under u, and in this case only one edge is created for all such consecutive i's.) This is justified by the definition of the end-equivalence relation, as we do have $u_i c \equiv_{uc} uc$ for all $i = 1, \ldots, h-1$. Hence, the current DAG represents $Suffix_D(uc)$.

For the suffix link of $newsink$, there are two possible cases to happen:

- When $u_h c = \overleftarrow{u_h c}^{u}$. In this case, $([u_h]_u, c, [u_h c]_u)$ is a primary edge. Due to Lemma 3, we have $[u_h c]_u = [u_h c]_{uc}$ and the suffix link from $newsink = [uc]_{uc}$ is set to node $[u_h c]_{uc}$. This operation is justified by Definition 3.
- When $u_h c \neq \overleftarrow{u_h c}^{u}$. In this case, $([u_h]_u, c, [u_h c]_u)$ is a secondary edge. Due to Lemma 3, we have $[u_h c]_u = [y u_h c]_{uc} \cup [u_h c]_{u_h c}$ where $\overleftarrow{u_h c}^{u} = y u_h c$ and $y \in D^+$. This is done by the function *SplitNode*, and the suffix link of $[u_h c]_{uc}$ is set to $link([u_h c]_u)$, and that of $[y u_h c]_{uc}$ is set to $[u_h c]_{uc}$. Then, the suffix link of $newsink = [uc]_{uc}$ is also set to node $[u_h c]_{uc}$. These operations are justified by Definition 3.

Judging whether $u_h c = \overleftarrow{u_h c}^{u}$ or not, namely, whether edge $([u_h]_u, c, [u_h c]_u)$ is primary or secondary, is done by the **if** condition in *SplitNode* checking the lengths of the nodes $[u_h]_u$ and $[u_h c]_u$. The resulting structure is $SDAWG_D(uc)$ with its suffix links. □

Now the only remaining matter is the time complexity of the algorithm.

Let $u, ua \in Prefix(w)$ with $w \in D^+$ and $a \in \Sigma \cup \{\#\}$. For any $x \in Prefix(Suffix_D(u))$ with $\overleftarrow{x}^{u} = x$, let $SC_u(x)$ be the list of nodes contained in the suffix-link path from node $[x]_u$ to the root of $T_D(u)$. We can establish the following lemma, similarly to [11].

Lemma 4. *Let $u \in Prefix(w)$ with $w \in D^+$. Assume that there is a primary edge $([x]_u, a, [xa]_u)$ in $SDAWG_D(u)$, with $\overleftarrow{x}^{u} = x$ and $\overleftarrow{xa}^{u} = xa$. Then $|SC_u(xa)| = |SC_u(x)| - m + 1$ where m is the number of secondary edges from nodes in $SC_u(x)$ to nodes in $SC_u(xa)$.*

We are ready to prove the following theorem, based on a similar idea to [11].

Theorem 3. *The execution time of the algorithm of Fig. 4 is linear in the input string length.*

Proof. Let $w \in D^+$ be the input string and let $u, ua \in \mathit{Prefix}(w)$ with $a \in \Sigma \cup \{\#\}$. Consider a single call to *Update* creating new node *newsink*, where $\mathit{sink} = [u]_u$ and $\mathit{newsink} = [ua]_{ua}$. Let l be the total number of iterations by the **for** and **do while** loops, except for the first execution of the **do while** loop that generates the primary edge from node s to r'. In any other iteration of either of these loops, a secondary edge is created from a node in $SC_{ua}(u)$ to either $[ua]_{ua}$ or $[z]_{ua}$, where z is the longest string in $\mathit{Suffix}_D(ua) \cap \mathit{Prefix}(\mathit{Suffix}_D(u))$. Since $z \in \mathit{Suffix}_D(ua)$, we have $[z]_{ua} \in SC_{ua}(ua)$. Therefore, by Lemma 4, we have $|SC_{ua}(ua)| \leq |SC_{ua}(u)| - l + 1$.

Moreover, consider the special case where $z \in [u]_u$ and $z \neq \overset{u}{\overleftarrow{u}}$. Recall that the occurrence of z as a suffix of ua immediately follows $\#$ in ua, namely, $ua[|ua| - |z|] = \#$. Since now $z \in \mathit{Suffix}_D(u)$, by the periodicity of z, this special case can happen only when $z = \#^{|z|}$. By Lemma 3, node $[u]_u$ is split into two nodes $[u]_{ua}$ and $[z]_{ua}$, increasing $|SC_{ua}(u)|$ by one from $|SC_u(u)|$. Consequently, we obtain $|SC_{ua}(ua)| \leq |SC_{ua}(u)| - l + 2$.

The above formula implies that at each call to *Update*, the suffix chain of *newsink* of $\mathit{SDAWG}_D(ua)$ can grow by at most two from the suffix chain of *sink* of $\mathit{SDAWG}_D(u)$. On the other hand, at each call to *Update*, the length of this suffix chain decreases by l, which is the number of iterations of the **for** and **do while** loops minus one. Note that the length of this suffix chain never gets zero, since at the beginning of the construction, $SC_D(\varepsilon)$ already has the initial state q_s of M_D. Hence, the total number of iterations of these loops when we have processed the entire string w is linear in $|w|$. □

4 Conclusions and Further Work

In this paper we introduced a new data structure $\mathit{SDAWG}_D(w)$ which supports a sparse text indexing of string w w.r.t. dictionary $D = \Sigma^*\#$. Namely, for any string $w = w_1 \cdots w_k$ with $w_i \in D$ for each $1 \leq i \leq k$, $\mathit{SDAWG}_D(w)$ represents the suffixes of w of the form $w_i \cdots w_k$. A typical application to SDAWGs is word- and phrase-level search on texts written in natural languages such as the European languages, where the blank character can be regarded as $\#$. Also, by using SDAWGs, the sparse text indexing problem stated in Section 1 is solvable in time proportional to the pattern length. Further, we showed that $\mathit{SDAWG}_D(w)$ has $O(n)$ nodes and edges, where $n = |w|$, and finally we presented an on-line algorithm that constructs $\mathit{SDAWG}_D(w)$ in $O(n)$ time and space.

Our future work includes the followings: The first one is to show the exact, tight bound on the size of SDAWGs. Blumer et al. [11] showed that for any string $w \in \Sigma^*$, $\mathit{DAWG}(w)$ has at most $2n - 1$ nodes and $3n - 4$ edges, where $n = |w|$. Since SDAWGs are a sparse version of DAWGs, SDAWGs should have strictly less nodes and edges, but this has to be explored in more details.

The second one is to extend this work to compact directed acyclic word graphs (CDAWGs) [13]. The idea of using the minimum DFA M_D is applicable to the

on-line construction algorithm for CDAWGs [14], yielding a sparse text indexing version of CDAWGs. We will then need to define this new data structure, show its size bound, and prove that the modified algorithm correctly constructs the desired structure in linear time.

References

1. Weiner, P.: Linear pattern-matching algorithms. In: Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory. (1973) 1–11
2. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2002)
3. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
4. Kärkkänen, J., Ukkonen, E.: Sparse suffix trees. In: Proc. 2nd International Computing and Combinatorics Conference (COCOON'96). Volume 1090 of Lecture Notes in Computer Science., Springer-Verlag (1996) 219–230
5. Inenaga, S., Kivioja, T., Mäkinen, V.: Finding missing patterns. In: Proc. 4th Workshop on Algorithms in Bioinformatics (WABI'04). Volume 3240 of Lecture Notes in Bioinformatics., Springer-Verlag (2004) 463–474
6. Bannai, H., Hyyrö, H., Shinohara, A., Takeda, M., Nakai, K., Miyano, S.: An $O(N^2)$ algorithm for discovering optimal boolean pattern pairs. IEEE/ACM Transactions on Computational Biology and Bioinformatics **1** (2004) 159–170
7. Inenaga, S., Bannai, H., Hyyrö, H., Shinohara, A., Takeda, M., Nakai, K., Miyano, S.: Finding optimal pairs of cooperative and competing patterns with bounded distance. In: Proc. 7th International Conference on Discovery Science (DS'04). Volume 3245 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2004) 32–46
8. Baeza-Yates, R., Gonnet, G.H.: Efficient text searching of regular expressions. In: Proc. 16th International Colloquium on Automata, Languages and Programming (ICALP'89). Volume 372 of Lecture Notes in Computer Science., Springer-Verlag (1989) 46–62
9. Andersson, A., Larsson, N.J., Swanson, K.: Suffix trees on words. Algorithmica **23** (1999) 246–260
10. Inenaga, S., Takeda, M.: On-line linear-time construction of word suffix trees. In: Proc. 17th Ann. Symp. on Combinatorial Pattern Matching (CPM'06). Lecture Notes in Computer Science, Springer-Verlag (2006) To appear.
11. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.: The smallest automaton recognizing the subwords of a text. Theoretical Computer Science **40** (1985) 31–55
12. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14** (1995) 249–260
13. Blumer, A., Blumer, J., Haussler, D., McConnell, R., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. Journal of the ACM **34** (1987) 578–595
14. Inenaga, S., Hoshino, H., Shinohara, A., Takeda, M., Arikawa, S., Mauri, G., Pavesi, G.: On-line construction of compact directed acyclic word graphs. Discrete Applied Mathematics **146** (2005) 156–179