Ternary Directed Acyclic Word Graphs

Satoru Miyamoto [a] Shunsuke Inenaga [b] Masayuki Takeda [a,c]
Ayumi Shinohara [a,c]

[a] *Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan*

[b] *Department of Computer Science, P.O.Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland*

[c] *SORST, Japan Science and Technology Agency (JST)*

Abstract

Given a set S of strings, a DFA accepting S offers a very time-efficient solution to the pattern matching problem over S. The key is how to implement such a DFA in the trade-off between time and space, and especially the choice of how to implement the transitions of each state is critical. Bentley and Sedgewick proposed an effective tree structure called *ternary trees*. The idea of ternary trees is to 'implant' the process of binary search for transitions into the structure of the trees themselves. This way the process of binary search becomes visible, and the implementation of the trees becomes quite easy. The *directed acyclic word graph* (*DAWG*) of a string w is the smallest DFA that accepts all suffixes of w, and requires only linear space. We apply the scheme of ternary trees to DAWGs, introducing a new data structure named *ternary DAWGs* (*TDAWGs*). Furthermore, the scheme of *AVL trees* is applied to the TDAWGs, yielding a more time-efficient structure *AVL TDAWGs*. We also perform some experiments that show the efficiency of TDAWGs and AVL TDAWGs, compared to DAWGs in which transitions are implemented by linked lists.

Key words: deterministic finite state automata, pattern matching on strings, directed acyclic word graphs, ternary search trees, AVL trees

1 Introduction

Due to rapid advance in information technology and global growth of computer networks, we can utilize a large amount of data today. In most cases, data

Email addresses: s-miya@i.kyushu-u.ac.jp (Satoru Miyamoto), inenaga@cs.helsinki.fi (Shunsuke Inenaga), takeda@i.kyushu-u.ac.jp (Masayuki Takeda), ayumi@i.kyushu-u.ac.jp (Ayumi Shinohara).

are stored and manipulated as *strings*. Therefore the development of efficient data structures for searching strings has for decades been a particularly active research area in computer science.

Given a set S of strings, we want some efficient data structure that enables us to search S very quickly. Obviously a DFA that accepts S is the one. The problem arising in implementing such an automaton is how to store the information of the transitions in each state. The most basic idea is to use tables, with which searching S for a given pattern p is feasible in $O(|p|)$ time, where $|p|$ denotes the length of p. However, the significant drawback is that the size of the tables is proportional to the size of the alphabet Σ used. In particular, it is crucial when the size of Σ is thousands large like in Asian languages such as Japanese, Korean, Chinese, and so on. Using linked lists is one apparent means of escape from this waste of memory space by tables. Although this surely reduces space requirement, searching for pattern p takes $O(|\Sigma| \cdot |p|)$ time in both worst and average cases. It is easy to imagine that this should be a serious disadvantage when searching texts of a large alphabet.

Bentley and Sedgewick [3] introduced an effective tree structure called *ternary search trees* (to be simply called *ternary trees* in this paper), for storing a set of strings. The idea of ternary trees is to 'implant' the process of binary search for transitions into the structure of the trees themselves. This way the process of binary search becomes visible, and the implementation of the trees becomes quite easy since each and every state of ternary trees has at most three transitions. Bentley and Sedgewick gave an algorithm that, for any set S of strings, constructs its ternary tree in $O(|\Sigma| \cdot \|S\|)$ time with $O(\|S\|)$ space, where $\|S\|$ denotes the total length of the strings in S. They also showed several nice applications of ternary trees [2].

This paper considers the most fundamental pattern matching problem on strings, the *substring pattern matching problem*, which is described as follows: *Given a text string w and pattern string p, examine whether or not p is a substring of w.* Clearly, a DFA that recognizes the set of all suffixes of w permits us to solve this problem very quickly. The smallest DFA of this kind was introduced by Blumer et al. [4], called the *directed acyclic word graph* (*DAWG*) of string w, that only requires $O(|w|)$ space.

In this paper, we apply the scheme of ternary trees to DAWGs, yielding a new data structure called *ternary DAWGs* (*TDAWGs*). By the use of a TDAWG of w, searching text w for patten p takes $O(|\Sigma| \cdot |p|)$ time in the worst case, but the time complexity in the average case is $O(\log |\Sigma| \cdot |p|)$, which is an advantage over DAWGs implemented with linked lists that require $O(|\Sigma| \cdot |p|)$ expected time. Therefore, the key is how to construct TDAWGs quickly. Note that the set of all suffixes of a string w is of size quadratic in $|w|$. Namely, simply applying the algorithm by Bentley and Sedgewick [3] merely allows us

to construct a TDAWG of w in $O(|\Sigma|\cdot|w|^2)$ time. However, using a modification of the on-line algorithm of Blumer et al. [4], pleasingly, the TDAWG of w can be constructed in $O(|\Sigma| \cdot |w|)$ time.

In addition, we have tackled the application of the scheme of *AVL trees* [1] to our TDAWGs. AVL trees are a kind of binary trees on which searching for any single character can be done in $O(\log|\Sigma|)$ time even in the worst case. Our new structure is a combination of AVL trees and TDAWGs, named *AVL TDAWGs*. Using the AVL TDAWG for a string w, it can be examined in $O(\log|\Sigma| \cdot |p|)$ time whether p is a substring of w or not, even in the worst case. Another nice feature of AVL TDAWGs is that the AVL TDAWG of any string w can be built in $O(\log|\Sigma| \cdot |w|)$ time.

We also performed some computational experiments to evaluate the efficiency of TDAWGs and AVL DAWGs using English and Japanese texts, by the comparison with DAWGs implemented by linked lists. The most exciting result is that the construction times of TDAWGs and AVL TDAWGs for the Japanese text are dramatically shorter than those of DAWGs with linked lists. This is typically shows our TDAWGs and AVL TDAWGs work very well for texts over a large size alphabet. Plus, search times by TDAWGs and AVL TDAWGs are much faster than those by DAWGs with linked lists. Our experiment also reveals that AVL TDAWGs are the fastest in searching for patterns both for English text and Japanese text, and this is surely the effect of AVL balancing for speeding up binary searches.

The rest of the paper is organized as follows. In Section 2, we recall the definition and the on-line construction algorithm of DAWGs. In Section 3, we introduce our new structure TDAWGs and show how they work. Section 4 is devoted to the introduction of the enhanced version of our new structure, AVL TDAWGs. We give the results of our experiments in Section 5 and conclude in Section 6.

2 Directed Acyclic Word Graphs

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*, *substring*, and *suffix* of string $w = xyz$, respectively. The sets of prefixes, substrings, and suffixes of a string w are denoted by $\mathit{Prefix}(w)$, $\mathit{Substr}(w)$, and $\mathit{Suffix}(w)$, respectively. The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Let $S \subseteq \Sigma^*$. The number of strings in S is denoted by $|S|$, and the sum of the lengths of strings in S by $\|S\|$.

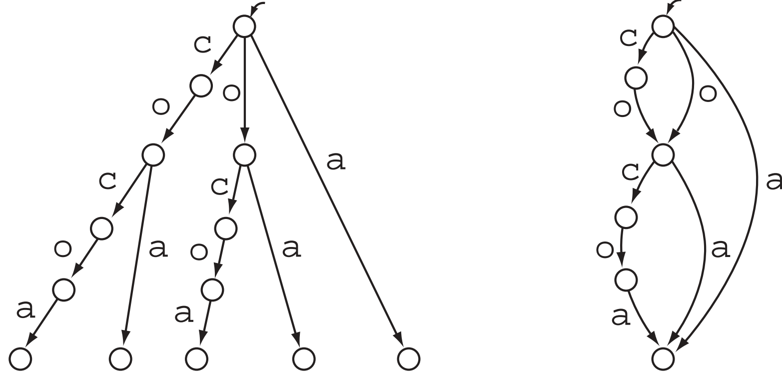

Fig. 1. *STrie*(cocoa) is shown on the left, where all the states are accepting. By minimizing this automaton we obtain *DAWG*(cocoa), on the right.

The following problem is the most fundamental and important in string processing.

Definition 1 (Substring Pattern Matching Problem)
Instance: *a text string* $w \in \Sigma^*$ *and pattern string* $p \in \Sigma^*$.
Determine: *whether* p *is a substring of* w.

Obviously, an automaton that accepts $Substr(w)$ is pretty useful to solve this problem. The most basic automaton of this kind is the *suffix trie*. The suffix trie of a string $w \in \Sigma^*$ is denoted by $STrie(w)$. What is obtained by minimizing $STrie(w)$ is called the *directed acyclic word graph* (*DAWG*) of w [9], denoted by $DAWG(w)$. In Fig. 1 we show $STrie(w)$ and $DAWG(w)$ with $w =$ cocoa.

The initial state of $DAWG(w)$ is also called the *source state*, and the state accepting w is called the *sink state* of $DAWG(w)$. Each state of $DAWG(w)$ other than the source state has a *suffix link*. Assume $x_1, \ldots, x_k$ are the substrings of w accepted in one state of $DAWG(w)$, arranged in the decreasing order of their lengths. Let $ay = x_k$, where $y \in \Sigma^*$ and $a \in \Sigma$. Then the suffix link of the state accepting $x_1, \ldots, x_k$ points to the state in which y is accepted.

DAWGs were first introduced by Blumer et al. [4], and have widely been used for solving the substring pattern matching problem as well as in various applications [7,8,15].

Theorem 1 (Crochemore [6]) *For any string* $w \in \Sigma^*$, $DAWG(w)$ *is the smallest (partial) DFA that recognizes* $Suffix(w)$.

Proposition 1 *Using* $DAWG(w)$ *whose transitions are implemented with linked lists, the substring pattern matching problem of Definition 1 is solvable in* $O(|\Sigma| \cdot |p|)$ *time in the worst and average cases.*

Theorem 2 (Blumer et al. [4]) *For any string* $w \in \Sigma^*$ *with* $|w| > 1$, $DAWG(w)$ *has at most* $2|w| - 1$ *states and* $3|w| - 3$ *transitions.*

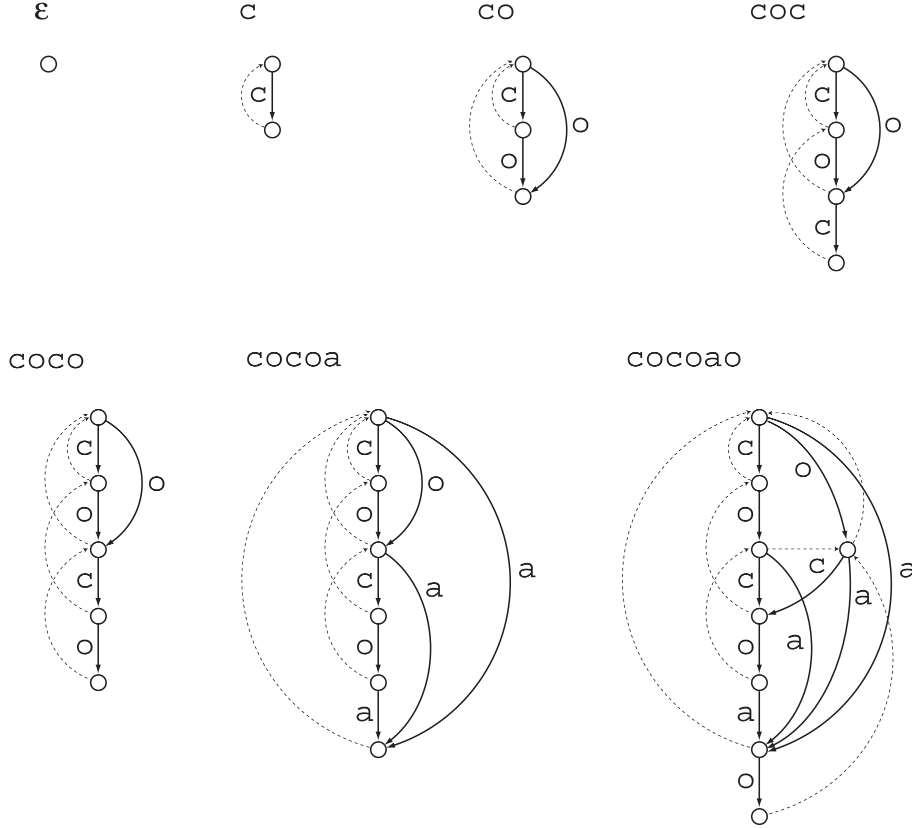

Fig. 2. On-line construction of $DAWG(w)$ with $w =$ cocoao. The solid arrows are the transitions, and the dashed arrows are the suffix links. In the process of updating $DAWG($cocoa$)$ to $DAWG($cocoao$)$, the state accepting $\{$co$,$ o$\}$ is separated into two states for $\{$co$\}$ and $\{$o$\}$.

It is a trivial fact that $DAWG(w)$ can be constructed in time proportional to the number of transitions in $STrie(w)$ using the DAG-minimization algorithm by Revuz [13]. However, the number of transitions of $STrie(w)$ is unfortunately quadratic in $|w|$. The direct construction of $DAWG(w)$ in linear time is therefore significant, in order to avoid creating redundant states and transitions that are deleted in the process of minimizing $STrie(w)$. Blumer et al. [4] indeed presented an algorithm that directly constructs $DAWG(w)$ and runs in linear time if Σ is fixed, by means of suffix links. Their algorithm is *on-line*, namely, for any $w \in \Sigma^*$ and $a \in \Sigma$ it allows us to update $DAWG(w)$ to $DAWG(wa)$ in amortized constant time, meaning that we need not construct $DAWG(wa)$ from scratch. On-line construction of $DAWG($cocoao$)$ is illustrated in Fig. 2.

We here briefly recall the on-line algorithm by Blumer et al. A more detailed description and pseudo-code of the algorithm can be found in [4]. The algorithm updates $DAWG(w)$ to $DAWG(wa)$ by inserting suffixes of wa into $DAWG(w)$ in decreasing order of their lengths. Let z be the longest string in $Substr(w) \cap Suffix(wa)$. Then z is called the *longest repeated suffix* of wa and

denoted by $LRS(wa)$. Let $z' = LRS(w)$. Let $|wa| = l$ and $u_1, u_2, \ldots, u_l, u_{l+1}$ be the suffixes of wa ordered by their lengths, that is, $u_1 = wa$ and $u_{l+1} = \varepsilon$. We categorize these suffixes of wa into the three following groups.

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

Note all suffixes in Group 3 are already represented in $DAWG(w)$. We can insert all the suffixes of Group 1 into $DAWG(w)$ by creating a new transition labeled by a from the current sink state to the new sink state. Therefore, we have only to care about those in Group 2. Let $v_i, \ldots, v_{j-1}$ be the suffixes of w such that, for any $i \leq k \leq j-1$, $v_k a = u_k$. We start from the state corresponding to $LRS(w) = z' = v_i$ in $DAWG(w)$, which is called the *active state* of the current phase. A new transition labeled by a is inserted from the active state to the new sink state. The state to be the next active state is found simply by traversing the suffix link of the state for v_i, in constant time, and a new transition labeled by a is created from the new active state to the sink state. After we insert all the suffixes of Group 2 this way, the automaton represents all the suffixes of wa. We now pay attention to $LRS(wa) = z = u_j$. The suffix link of the new sink state is set to point to the state that accepts u_j.
Let us see a concrete example from Fig. 2. See the conversion of $DAWG(\texttt{coco})$ to $DAWG(\texttt{cocoa})$. Regarding the suffixes of `cocoa`, we have Group 1: `cocoa`, `ocoa`; Group 2: `coa`, `oa`, `a`; Group 3: ε. By creating a new transition labeled by `a` from the old sink state to the new sink state, the suffixes `cocoa` and `ocoa` in Group 1 get to be accepted by the automaton. Notice the current active state is the state corresponding to $\{\texttt{co}, \texttt{o}\}$. From this state we create a new transition labeled by `a` to the new sink state. Then two suffixes `coa` and `oa` in Group 2 are now accepted. After that, we go up to the source state by traversing the suffix link of the state accepting $\{\texttt{co}, \texttt{o}\}$. We create a new transition labeled with `a` from the source state to the new sink state, and now all the suffixes of `cocoa` are accepted by the automaton. Finally, we set the suffix link of the new sink state so that it points to the source state that accepts ε.

We note that an event so called *node separation* can happen at the last stage of updating $DAWG(w)$ to $DAWG(wa)$. Let s be the state which accepts u_j, and let x be the longest string accepted by state s. We then check whether $x = u_j$ or not. If so, we are finished. Otherwise, state s is *separated* into two states, s and its duplication s', where s becomes to accept the strings longer than u_j and s' accepts the rest. A concrete example can be seen in the conversion of $DAWG(\texttt{cocoa})$ into $DAWG(\texttt{cocoao})$ shown in Fig. 2. Here, $u_j = \texttt{o}$ and $x = \texttt{co}$, thus we have $u_j \neq x$. Then state s which in this case accepts $\{\texttt{co}, \texttt{o}\}$ is separated into two states accepting $\{\texttt{co}\}$ and $\{\texttt{o}\}$, respectively. All the

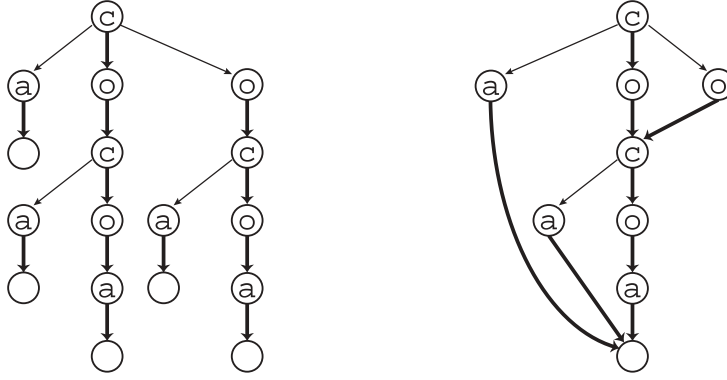

Fig. 3. *TSTrie*(w) is on the left, and *TDAWG*(w) on the right, with $w = \mathsf{cocoa}$.

(outgoing) transitions of s are also duplicated for s', namely, the target state of the transitions of s' is the same as the target state of the transitions of s (see Fig. 2). The suffix links of s and s' also have to be adjusted. Let t be the state to which the suffix link of s is directed. Then the suffix link of s is redirected to s', and the suffix link of s' is directed to t. Associating each state with the length of the longest string accepted in it, we can deal with this state separation in amortized constant time.

Theorem 3 (Blumer et al. [4]) *For any string $w \in \Sigma^*$, DAWG(w) can be constructed on-line and in $O(|\Sigma| \cdot |w|)$ time using $O(|w|)$ space, if the transitions are implemented by linked lists.*

The $|\Sigma|$ factor in the time complexity of the above theorem comes from the fact that searching transitions in each state takes $O(|\Sigma|)$ time if we implement the transitions by linked lists, as stated in Proposition 1. Therefore, efficient implementation of the transitions is crucial in order to achieve faster search and construction of DAWGs. In the following sections, we will show our new automata which enable us faster search and construction.

3 Ternary Directed Acyclic Word Graphs

In this section, we present a new kind of automata called *ternary directed acyclic word graphs* (*TDAWGs*). The idea is to implement the transitions of DAWGs by using *ternary search trees* [3,2] (in short, ternary trees). Ternary trees are quite useful for storing a set of strings from both viewpoints of space efficiency and search speed. The idea of ternary trees is to 'implant' the process of binary search for linked lists into the trees themselves. This way the process of binary search becomes visible, and the implementation of the trees becomes quite easy since each and every state of ternary trees has at most three transitions.

The left of Fig. 3 is a ternary tree for $\mathit{Suffix}(w)$ with $w = \texttt{cocoa}$. We can see that this corresponds to $\mathit{STrie}(w)$ in Fig. 1, and therefore, the tree is called a *ternary suffix trie* (*TSTrie*) of string cocoa.

For a substring x of a string $w \in \Sigma^*$, we consider set $\mathit{CharSet}_w(x) = \{a \in \Sigma \mid xa \in \mathit{Substr}(w)\}$ of characters. In $\mathit{STrie}(w)$, each character of $\mathit{CharSet}_w(x)$ is associated with a transition from state x (see $\mathit{STrie}(\texttt{cocoa})$ in Fig. 1). However, in a TSTrie of w, each character in $\mathit{CharSet}_w(x)$ corresponds to a *state*. This means that we can regard $\mathit{CharSet}_w(x)$ as a set of the states that immediately follow string x in the TSTrie of w, where elements of $\mathit{CharSet}_w(x)$ are arranged in lexicographical order, top-down. There are many variations of the arrangement of elements in $\mathit{CharSet}_w(x)$, but we arrange them in increasing order of their leftmost occurrences in w, top-down. Thus the arrangement of the states is uniquely determined, and the resulting structure is called *the* TSTrie of w, denoted by $\mathit{TSTrie}(w)$. The state corresponding to the character in $\mathit{CharSet}_w(x)$ with the earliest occurrence, is called the *top* state with respect to $\mathit{CharSet}_w(x)$, since it is arranged on the top of the states for characters in $\mathit{CharSet}_w(x)$.

We now describe how searching for a pattern takes place in $\mathit{TSTrie}(w)$. Given a pattern p, at any node of $\mathit{TSTrie}(w)$ we examine if the character a in p we currently focus on is lexicographically larger than the character b stored in the state. If $a < b$, then we take the left transition from the state and compare a to the character in the next state. If $a > b$, then we take the right transition from the state and compare a to the character in the next state. If $a = b$, then we take the center transition from the state, now the character a has been recognized, and we compare the next character in p to the character in the next state. We give a concrete example of searching for pattern oa using $\mathit{TSTrie}(\texttt{cocoa})$ in Fig. 3. We start from the initial state of the tree and have $\texttt{o} > \texttt{c}$, and thus go down to the next state via the right transition. At the next state we have $\texttt{o} = \texttt{o}$, and thus we take the center transition from the state and arrive at the next state, with the character o recognized. We then compare the next character a in the pattern with c in the state where we are. Now we have $\texttt{a} < \texttt{c}$, we go down along the left transition of the state and arrive at the next state, where we have $\texttt{a} = \texttt{a}$. Then we take the center transition and arrive at the next state, where finally oa is accepted. This way, for any pattern $p \in \Sigma^*$ we can solve the substring pattern matching problem of Definition 1 in $O(\log|\Sigma| \cdot |p|)$ expected time.

We now consider to apply the above scheme to $\mathit{DAWG}(w)$. What is obtained here is the *ternary DAWG* (*TDAWG*) of w, denoted by $\mathit{TDAWG}(w)$. The right of Fig. 3 is $\mathit{TDAWG}(\texttt{cocoa})$. Compare it to $\mathit{DAWG}(\texttt{cocoa})$ in Fig. 1 and $\mathit{TSTrie}(\texttt{cocoa})$ in Fig. 3.

Proposition 2 *Using* $\mathit{TDAWG}(w)$*, the substring pattern matching problem*

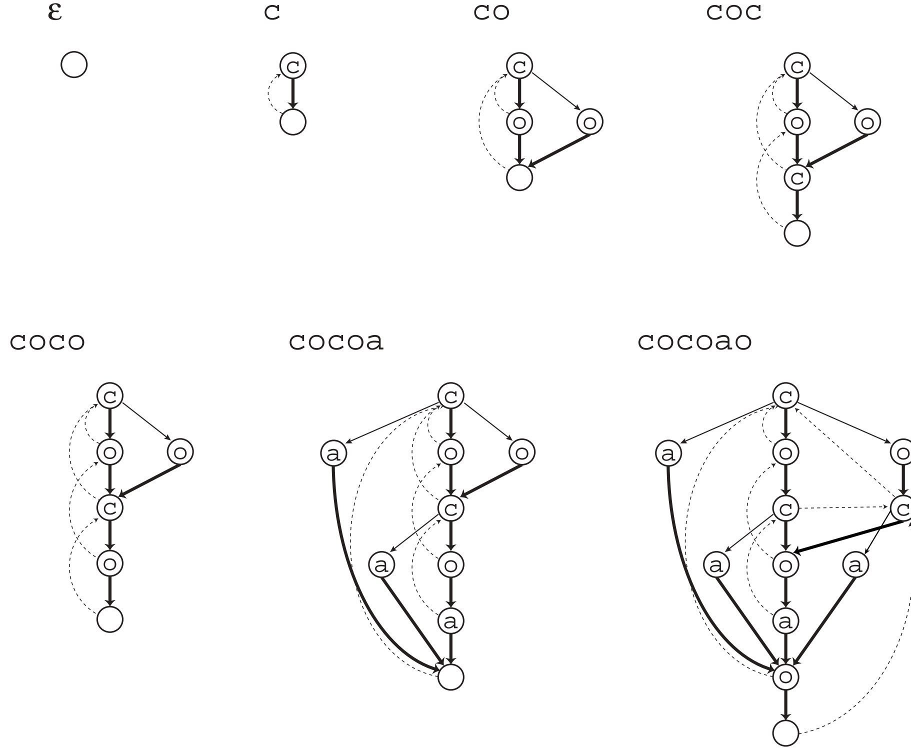

Fig. 4. On-line construction of $\mathit{TDAWG}(w)$ with $w =$ `cocoao`. The dashed arrows are the suffix links. Notice only top states have suffix links, and only top states can be the target of suffix links of other top states. In the process of updating $\mathit{TDAWG}($`cocoa`$)$ to $\mathit{TDAWG}($`cocoao`$)$, the state accepting $\{$`co`$,$ `o`$\}$ is separated into two states for $\{$`co`$\}$ and $\{$`o`$\}$, as well as the case of DAWGs shown in Fig 2.

of Definition 1 is solvable in $O(|\Sigma| \cdot |p|)$ *time in the worst case, and in* $O(\log |\Sigma| \cdot |p|)$ *time in the average case.*

Notice the advantage in the average case of TDAWGs against DAWGs on searching for patterns (See Proposition 1).

On-line construction of TDAWGs can be done based on the on-line DAWG construction algorithm by Blumer et al. [4], which was recalled in Section 2. On-line construction of $\mathit{TDAWG}($`cocoao`$)$ is illustrated in Fig. 4.

Here are two small remarks about on-line construction of TDAWGs: The first is about suffix links. In TDAWGs only top states have suffix links, and only top states can be the target of suffix links of other top states. The second is about state separation. When a top state is separated, then the other states belonging to the same *CharSet* as the top state have to be duplicated. A concrete example can be seen in the conversion of $\mathit{TDAWG}($`cocoa`$)$ to $\mathit{TDAWG}($`cocoao`$)$ in Fig. 4, where top state accepting $\{$`co`$,$ `o`$\}$ is separated into two top states accepting $\{$`co`$\}$ and $\{$`o`$\}$.

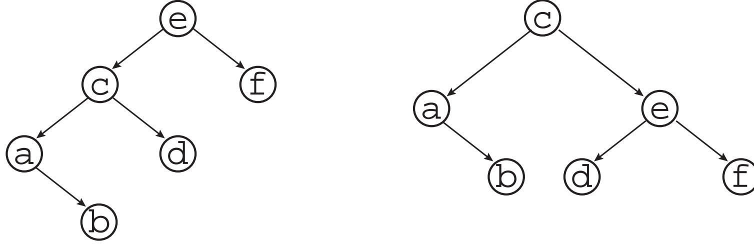

Fig. 5. Examples of a non-AVL tree on the left and an AVL tree on the right.

Now we have the following theorem:

Theorem 4 *For any string $w \in \Sigma^*$, $TDAWG(w)$ can be constructed on-line, in $O(|\Sigma| \cdot |w|)$ time using $O(|w|)$ space.*

4 AVL Ternary Directed Acyclic Word Graphs

AVL trees [1] are a well-known fast tree structure in binary searches. The idea is to balance each subtree so that the worst case time complexity for binary search becomes $O(\log |\Sigma|)$.

Definition 2 (AVL Tree) *Define the height of a tree as the maximum length of any path from its root to a leaf. An* AVL tree *is a binary search tree such that the height of the left and right subtrees of any node differs by at most one.*

See Fig. 5 for examples of a non-AVL tree and AVL tree.

Now our idea is to apply this scheme to our TDAWGs so that, for any top state of $TDAWG(w)$ corresponding to a substring x of w, the tree for $CharSet_w(x)$ consisting of the left and right transitions is an AVL tree. We call it the *AVL TDAWG* of w, and denote by $avl_TDAWG(w)$. $avl_TDAWG(w)$ is superior to $TDAWG(w)$ on searching for a pattern p, as stated in the following proposition.

Proposition 3 *Using $avl_TDAWG(w)$, the substring pattern matching problem of Definition 1 is solvable in $O(\log |\Sigma| \cdot |p|)$ time in the worst and average cases.*

Moreover, we can construct $avl_TDAWG(w)$ in on-line manner, by examining the AVL condition each time a new state (character) is inserted into the tree for $CharSet_w(x)$ consisting of the left and right transitions. The addition of a new state to the tree can sometimes violate the AVL condition, and then we rotate those states so that the tree can still remain an AVL tree. It is a well-known fact that:

Lemma 1 (G. Adelson-Velskii and E. Landis [1]) *Inserting a new node into an AVL tree takes $O(\log N)$ time, where N is the number of nodes of the AVL tree.*

Due to the above lemma, we obtain the following theorem.

Theorem 5 *For any string $w \in \Sigma^*$, avl_TDAWG(w) can be constructed on-line, in $O(\log |\Sigma| \cdot |w|)$ time using $O(|w|)$ space.*

On-line construction of $avl_TDAWG(w)$ is illustrated in Fig. 6. One might suspect that it is sometimes necessary to redirect suffix links after rotating nodes, as seen in the node rotation of $avl_TDAWG(\mathtt{bef})$ in Fig. 6. The number of the suffix links for each $CharSet_w(x)$ is $O(|\Sigma|)$, and thus, if such redirection can happen every time a new character is added, we no longer can construct $avl_TDAWG(w)$ in $O(\log |\Sigma| \cdot |w|)$ time. However, if we use an auxiliary state connected to the current top state for each $CharSet_w(x)$ and associate the suffix links with this auxiliary state, we can redirect all the suffix links in $O(1)$ time by reconnecting the auxiliary state to the new top state after the rotation. Hence we can achieve the improved time complexity mentioned in Theorem 5.

5 Experiments

In this section we show some experimental results that reveal the advantage of our TDAWGs and AVL TDAWGs, compared to DAWGs whose transitions are implemented with linked lists (denoted list_DAWGs). The linked lists were linearly searched at any state of the list_DAWGs. All the three algorithms to construct TDAWGs, AVL TDAWGs, and list_DAWGs were implemented in the C language. All calculations were performed on a Desktop PC with Pentium4-1.7GHz CPU and 768MB main memory running Windows XP Professional. We used the English text "ohsumed.91" available at `http://trec.nist.gov/data.html`, and the Japanese texts from novels of Soseki Natsume available at `http://www.aozora.gr.jp/`.

The first test was to compare memory space requirements of TDAWGs, AVL TDAWGs, and list_DAWGs. The left chart of Fig. 7 shows memory requirements of TDAWGs, AVL TDAWGs, and list_DAWGs for the English texts, where the memory requirements grow linearly, as expected. TDAWGs require about 19% more memory than list_DAWGs. Also, AVL TDAWGs require about 24% and 4% more memory than list_DAWGs and TDAWGs, respectively. The right chart of Fig. 7 shows memory requirements of TDAWGs, AVL TDAWGs, and list_DAWGs for the Japanese texts. Here again, the memory requirements grow linearly as expected. TDAWGs require about 28% more

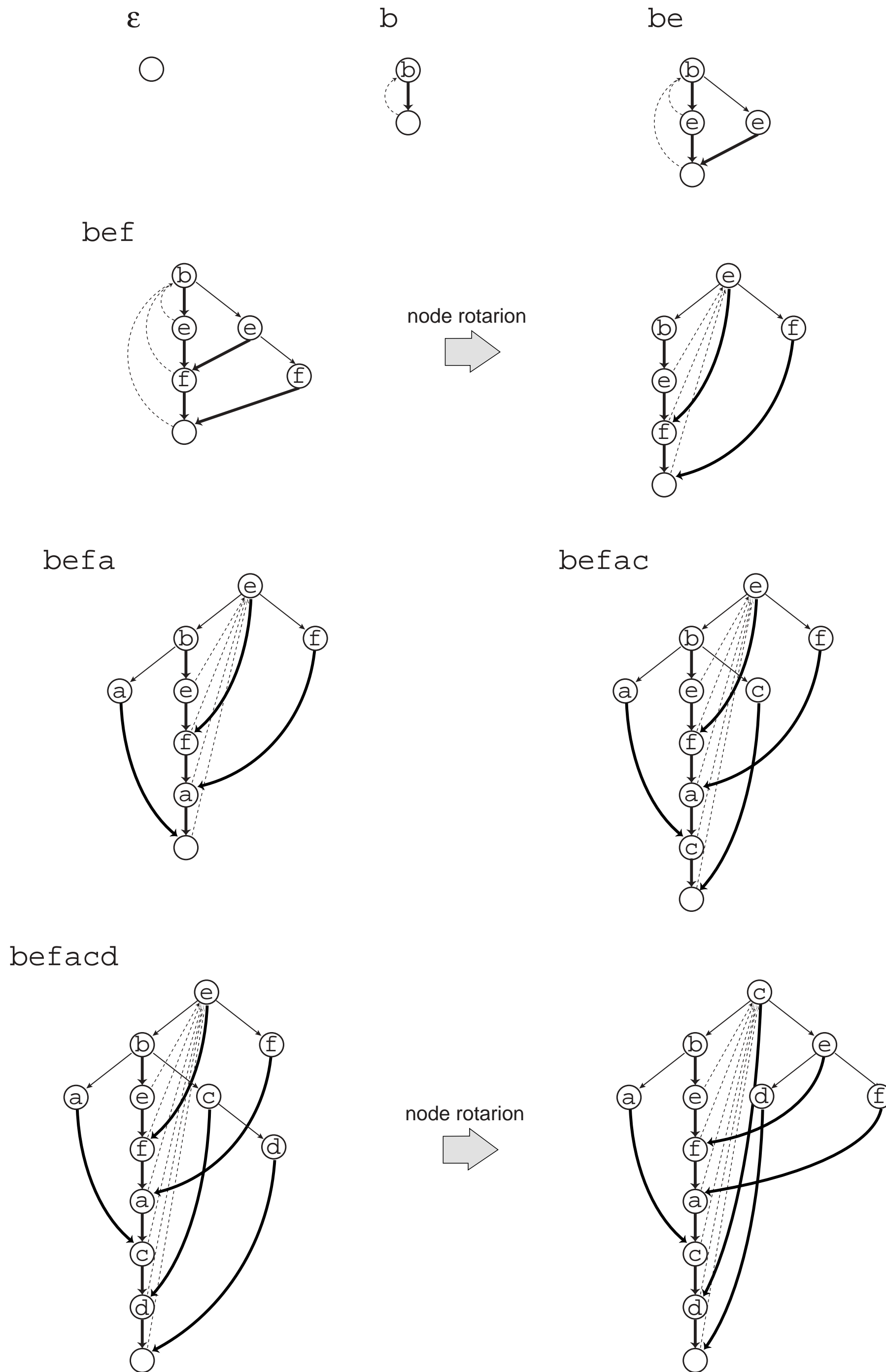

Fig. 6. On-line construction of $avl_TDAWG(w)$ with w = `befacd`. The dashed arrows are the suffix links. After a new character `f` is inserted into the source state of $avl_TDAWG($`be`$)$, the tree for the source state becomes a non-AVL tree. Therefore we rotate the nodes for characters `b`, `e`, and `f` so that the tree remains an AVL tree (second upper right). Another type of node rotation happens in inserting a new character `d` into the source state of $avl_TDAWG($`befac`$)$. The resulting structure (lower right) is $avl_TDAWG($`befacd`$)$ in which the trees for all $CharSet_w(x)$ are AVL balanced.

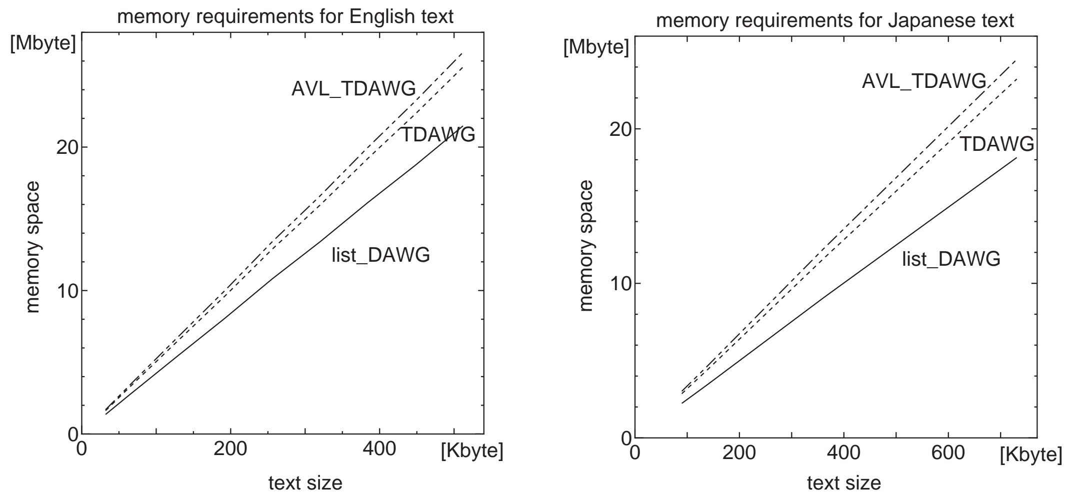

Fig. 7. The left and right charts show the space requirements of the TDAWG, AVL TDAWG, and list_DAWG for the English and Japanese texts, respectively.

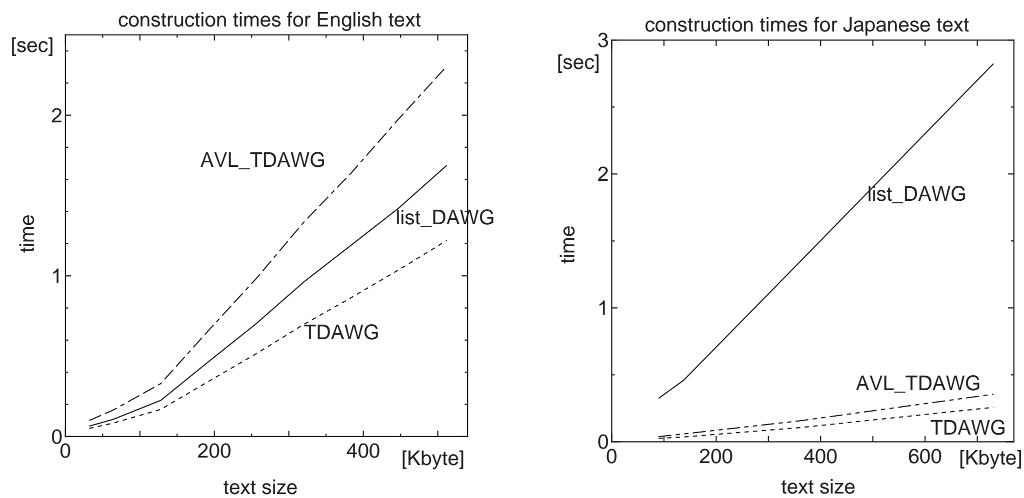

Fig. 8. The left and right charts show the construction times of the TDAWG, AVL TDAWG, and list_DAWG for the English and Japanese texts, respectively.

memory than list_DAWGs, and AVL TDAWGs require about 35% and 5% more memory than list_DAWGs and TDAWGs, respectively.

The second test was to compare construction times of TDAWGs, AVL TDAWGs, and list_DAWGs. The left chart of Fig. 8 shows construction times of the TDAWGs, AVL TDAWGs, and list_DAWGs for the English texts. One can see that the constructions of TDAWGs were done about 1.2 times faster than that of list_DAWGs. This should be the effect of binary search in the TDAWGs, while the linked lists were linearly searched in the list_DAWGs. AVL TDAWGs were constructed about twice slower than TDAWGs. It seems that this comes from the cost of node rotations for balancing each AVL tree in the AVL TDAWGs. The right chart of Fig. 8 shows construction times of TDAWGs, AVL TDAWGs, and list_DAWGs for the Japanese texts, which is

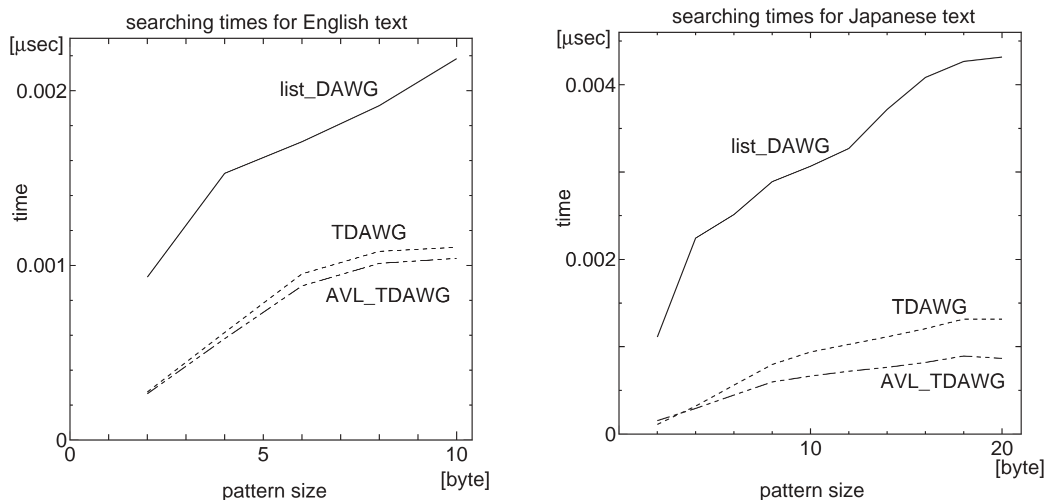

Fig. 9. The left and right charts show the search times of the TDAWG, AVL TDAWG, and list_DAWG for the English and Japanese texts, respectively.

one of the most dramatic results from our experiments. TDAWGs were constructed thirteen times faster than list_DAWGs, and even AVL DAWGs were constructed eight times faster than list_DAWGs! This is obviously because the linked lists of the list_DAWGs were linearly searched, while binary searches were operated in the others. This result typically shows one effectiveness of TDAWGs and AVL DAWGs for texts over a large alphabet.

The third test was searching times for patterns of different lengths. We used 9 texts of different lengths, in the range from 32 to 512 Kbytes. For each text, we randomly chose 100 of their substrings of each length, and searched for each of these substrings 1 million times. The result shown in the left chart of Fig. 9 is the average time of searching for a pattern, using the English texts. One can see both TDAWGs and AVL TDAWGs are more than twice faster than list_DAWGs. Moreover, search on AVL TDAWGs is slightly faster than on TDAWGs, due to the effect of balancing trees. In the right chart of Fig. 9 that shows the average time of searching for a pattern in the Japanese texts, the effect of TDAWGs and AVL TDAWGs is visualized better; TDAWGs are more than three times faster than list_DAWGs, and AVL TDAWGs are about five times faster than list_DAWGs. Moreover, searching on AVL TDAWGs is about 1.5 times faster than TDAWGs, where balancing trees on AVL TDAWGs took effect.

6 Conclusions and Further Work

Table 1 summarizes the space requirements, construction times, worst-case search times, and average search times of DAWGs whose transitions are im-

Table 1
Comparison of DAWG implemented with tables (denoted table_DAWG), DAWG implemented with linked lists (denoted list_DAWG), TDAWGs, and AVL TDAWGs, where Σ is the alphabet, and n and m are the length of the text and pattern, respectively.

type of DAWG	*space requirement*	*construction time*	*worst-case search time*	*average search time*
table_DAWG	$O(\|\Sigma\|n)$	$O(\|\Sigma\|n)$	$O(m)$	$O(m)$
list_DAWG	$O(n)$	$O(\|\Sigma\|n)$	$O(\|\Sigma\|m)$	$O(\|\Sigma\|m)$
TDAWG	$O(n)$	$O(\|\Sigma\|n)$	$O(\|\Sigma\|m)$	$O(\log\|\Sigma\|m)$
AVL TDAWG	$O(n)$	$O(\log\|\Sigma\|n)$	$O(\log\|\Sigma\|m)$	$O(\log\|\Sigma\|m)$

plemented with tables (table_DAWG), DAWGs whose transitions are implemented with linked lists (list_DAWG), TDAWGs and AVL TDAWGs. Although table_DAWGs are surely very fast in search, they actually consume too much space, $O(|\Sigma|n)$. Especially for texts over a large sized alphabet such as Japanese, Korean, Chinese etc., table_DAWGs are absolutely unrealistic. TDAWGs are better in average search time than list_DAWGs, where the linked lists in list_DAWGs are linearly searched but binary searches take place in TDAWGs. AVL DAWGs have good complexities for all of space requirement, construction time, worst-case search time, and average search time. The results of our experiments in Section 5 have shown that our new structures TDAWGs and AVL TDAWGs are useful in practice as well, especially for texts over a large alphabet.

We emphasize that the benefit of the ternary-based implementation is not limited to DAWGs. Namely, it can be applied to any automata-oriented index structure such as *suffix trees* [16,12,14,10] and *compact directed acyclic word graphs* (*CDAWGs*) [5,9,11]. Therefore, we can also consider *ternary suffix trees* and *ternary CDAWGs*. Concerning the experimental results on TDAWGs, ternary suffix trees and ternary CDAWGs promise to perform very well in practice.

Moreover, there is another variation of TDAWGs that is more space-economical. Note that Fig. 3 of Section 3 can be minimized by the algorithm of Revuz [13], and the resulting structure is shown in Fig. 10, which is called the *minimum TDAWG* (*MTDAWG*) of the string. To use Revuz's algorithm we have to maintain the reversed transition for every transition, and it certainly requires too much space. Thus we are now interested in an on-line algorithm to construct MTDAWGs *directly*, and it is our future work. We expect that search time on MTDAWGs will be in practice faster than using TDAWGs, since memory allocation for MTDAWGs is likely to be quicker.

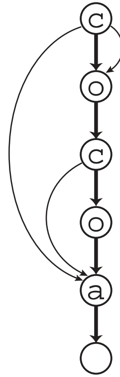

Fig. 10. The MTDAWG of string cocoa.

References

[1] G. Andelson-Velskii and E. Landis. An algorithm for the organisation of information. *Soviet. Math.*, 3:1259–1262, 1962.

[2] J. Bentley and B. Sedgewick. Ternary search trees. *Dr. Dobb's Journal*, 1998. `http://www.ddj.com/`.

[3] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 360–369. ACM/SIAM, 1997.

[4] 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.

[5] 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.

[6] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[7] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, New York, 1994.

[8] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.

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

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

[11] 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 *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*,

volume 2089 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, 2001.

[12] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[13] D. Revuz. Minimization of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.

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

[15] E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, 1993.

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