Ternary Directed Acyclic Word Graphs

Satoru Miyamoto[1], Shunsuke Inenaga[1,2],
Masayuki Takeda[1,2], and Ayumi Shinohara[1,2]

[1] Department of Informatics, Kyushu University 33,
Fukuoka 812-8581, Japan
[2] PRESTO, Japan Science and Technology Corporation (JST)
{s-miya,s-ine,takeda,ayumi}@i.kyushu-u.ac.jp

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*). We perform some experiments that show
the efficiency of TDAWGs, compared to DAWGs in which transitions are
implemented by tables and linked lists.

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 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].

We in this paper consider 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 for 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. We also performed some computational experiments to evaluate the efficiency of TDAWGs, using English texts, by the comparison with DAWGs implemented by tables and linked lists. The most interesting result is that the construction time of TDAWGs is dramatically faster than those of DAWGs with tables and linked lists. Plus, it is evaluated that search time by TDAWGs is also faster than that by DAWGs with linked lists.

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 *Prefix*(w), *Substr*(w), and *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 $\|S\|$.
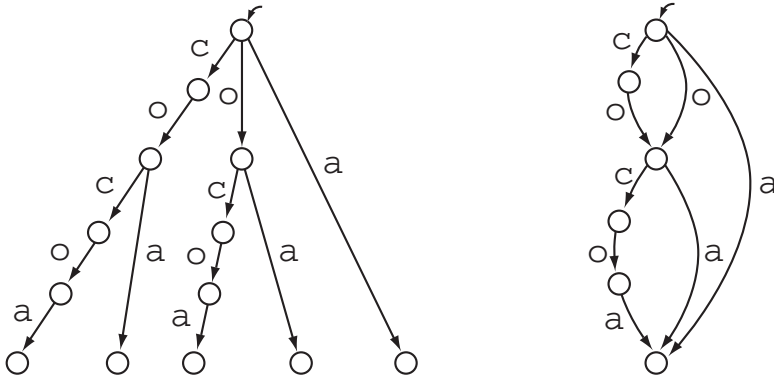
Fig. 1. *STrie*(cocoa) is shown on the left, where all the states are accepting. By minimizing this automaton we obtain *DAWG*(cocoa), on the right

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.*

It is clear that this problem is solvable in time proportional to the length of p, by using an automaton that accepts $Substr(w)$. 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 $ya = x_k$, 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, and 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).*

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.*

It is a trivial fact that $DAWG(w)$ can be constructed in time proportional to the number of transitions in $STrie(w)$ by the DAG-minimization algorithm
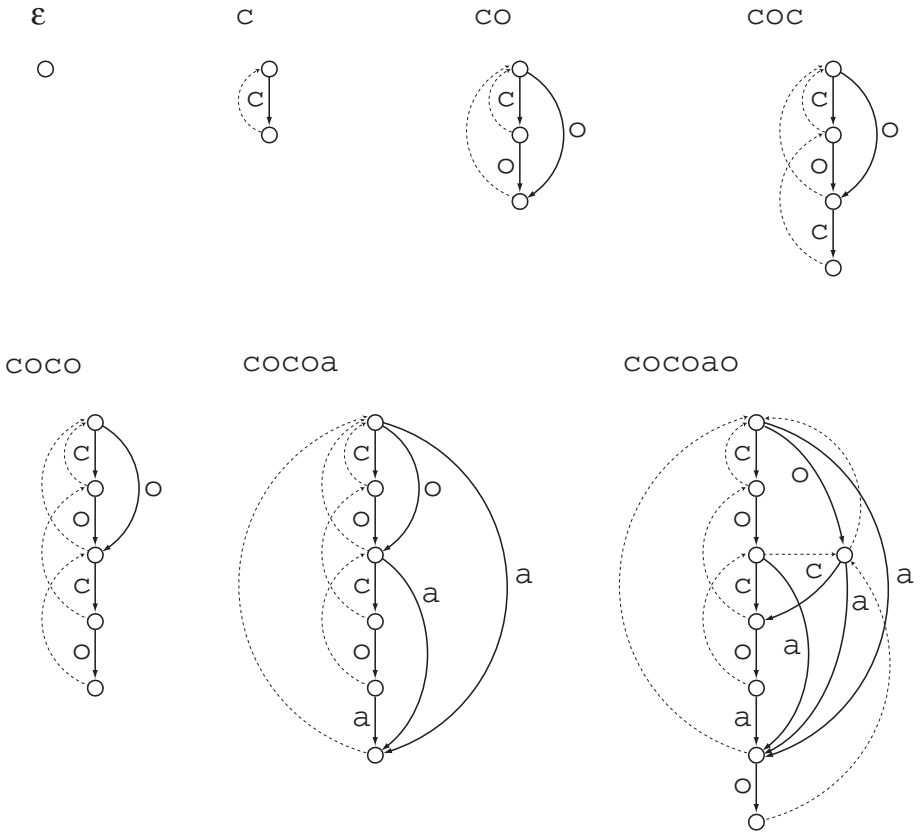
Fig. 2. The on-line construction of $DAWG(w)$ with $w = $ `cocoao`. The solid arrows are the transitions, and the dashed arrows are the suffix links. Note that the state pointed by the suffix link of the sink state will be the active state of the next phase. In the process of updating $DAWG(\texttt{cocoa})$ to $DAWG(\texttt{cocoao})$, the state accepting $\{\texttt{co}, \texttt{o}\}$ is separated into two states for $\{\texttt{co}\}$ and $\{\texttt{o}\}$

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.

We here briefly recall the on-line algorithm by Blumer et al. It 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 in their lengths, that is, $u_1 = wa$ and $u_{l+1} = \varepsilon$. We categorize these suffixes of wa into the following three groups.

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

Note all suffixes in Group 3 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. It obviously takes only constant time. 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 $LRS(wa) = z = u_j$. Let x be the longest string in the state where u_j is accepted. We then have to check whether $x = u_j$ or not. If not, the state is *separated* into two states, where one accepts the longer strings than u_j, and the other accepts the rest. Associating each state with the length of the longest string accepted in it, we can deal with this state separation in constant time.

The on-line construction of $DAWG(\texttt{cocoao})$ is shown in Fig. 2.

3   Ternary Directed Acyclic Word Graphs

Bentley and Sedgewick [3, 2] introduced a new data structure called *ternary trees*, which 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 figure in Fig. 3 is a ternary tree for $Suffix(w)$ with $w = \texttt{cocoa}$. We can see that this corresponds to $STrie(w)$ in Fig. 1, and therefore, the tree is called a *ternary suffix trie* (*TSTrie*) of string $\texttt{cocoa}$.

For a substring x of a string $w \in \Sigma^*$, we consider set $CharSet_w(x) = \{a \in \Sigma \mid xa \in Substr(w)\}$ of characters. In $STrie(w)$, each character of $CharSet_w(x)$ is associated with a transition from state x (see $STrie(\texttt{cocoa})$ in Fig. 1). However, in a TSTrie of w, each character in $CharSet_w(x)$ corresponds to a *state*. This means that we can regard $CharSet_w(x)$ as a set of the states that immediately
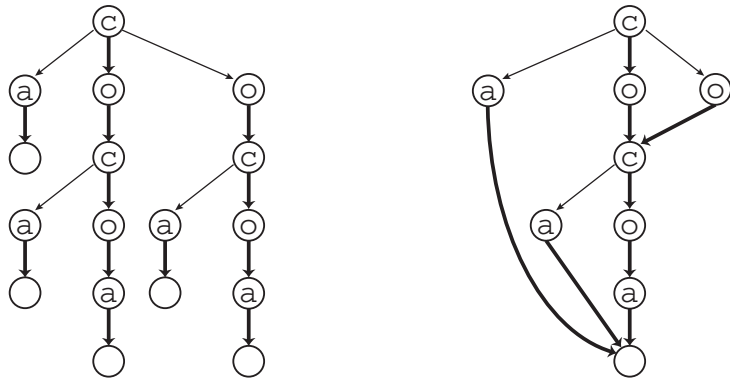
Fig. 3. *TSTrie(w)* is on the left, and *TDAWG(w)* on the right, with $w = \texttt{cocoa}$

follows string x in the TSTrie of w, where elements of $CharSet_w(x)$ are arranged in lexicographical order, top-down. There are many variations of the arrangement of elements in $CharSet_w(x)$, 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 *TSTrie(w)*. The state corresponding to the character in $CharSet_w(x)$ with the earliest occurrence, is called the *top* state with respect to $CharSet_w(x)$, since it is arranged on the top of the states for characters in $CharSet_w(x)$.

Given a pattern p, at any node of *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 is 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 $\texttt{oa}$ using *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 $\texttt{o}$ recognized. We then compare the next character $\texttt{a}$ in the pattern with $\texttt{c}$ 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 $\texttt{oa}$ 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 $DAWG(w)$. What is obtained here is the *ternary DAWG* (*TDAWG*) of w, denoted by $TDAWG(w)$. The right figure in Fig. 3 is $TDAWG(\texttt{cocoa})$. Compare it to $DAWG(\texttt{cocoa})$ in Fig. 1 and *TSTrie*($\texttt{cocoa}$) in Fig. 3. It is quite obvious that using $TDAWG(w)$ we can examine if $p \in Substr(w)$ in $O(\log |\Sigma| \cdot |p|)$ expected time, as well. Reasonably,
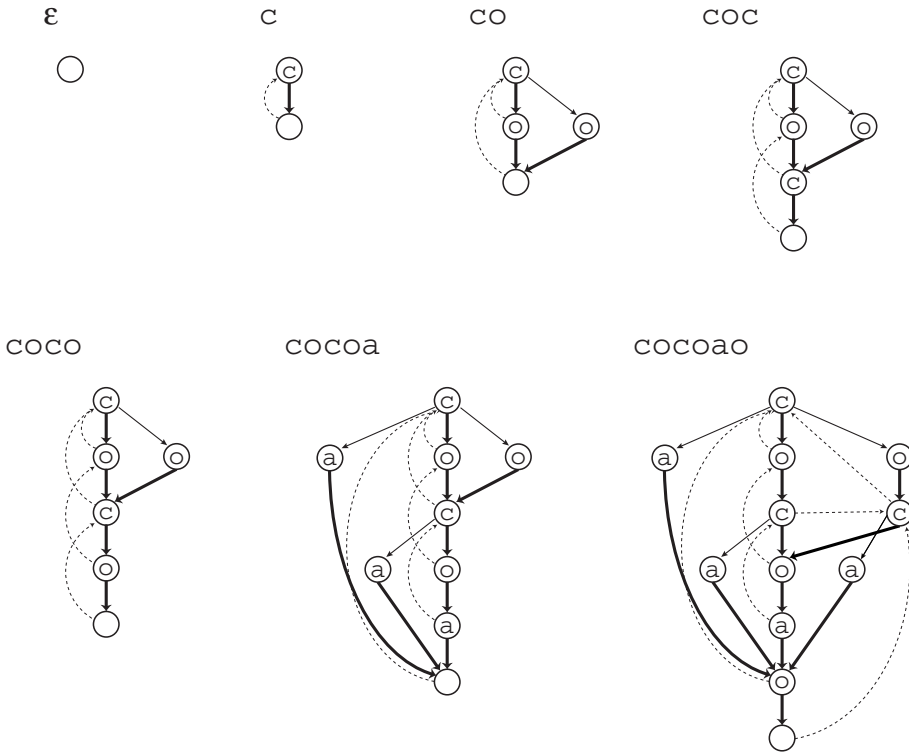
Fig. 4. The on-line construction of *TDAWG(w)* with *w* = cocoao. The dashed arrows are the suffix links. Notice only top states have suffix links, and are pointed by suffix links of other top states. In the process of updating *TDAWG*(cocoa) to *TDAWG*(cocoao), the state accepting {co, o} is separated into two states for {co} and {o}, as well as the case of DAWGs shown in Fig 2

TDAWG(w) can be constructed by the on-line algorithm of Blumer et al. [4] that was recalled in Section 2. In *TDAWG(w)* only top states have suffix links, and can be directed by suffix links of other states.

The on-line construction of *TDAWG*(cocoa) is shown in Fig. 4.

Theorem 3. *For any string* $w \in \Sigma^*$, *TDAWG(w) can be constructed on-line, in* $O(|\Sigma| \cdot |w|)$ *time using* $O(|w|)$ *space.*

4   Experiments

In this section we show some experimental results that reveal the advantage of our TDAWGs, compared to DAWGs with tables (table_DAWGs) and DAWGs with linked lists (list_DAWGs). The tables were implemented by arrays of length 256. The linked lists were linearly searched at any state of the
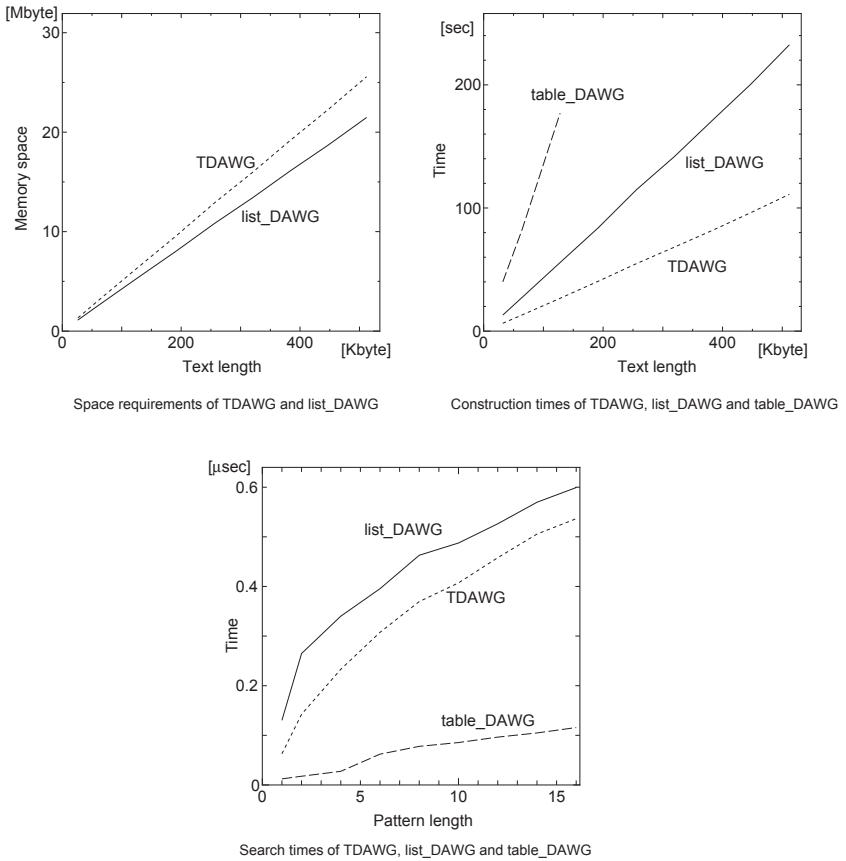
Space requirements of TDAWG and list_DAWG

Construction times of TDAWG, list_DAWG and table_DAWG

Search times of TDAWG, list_DAWG and table_DAWG

Fig. 5. The upper left chart is the memory requirements (in Mbytes) of TDAWGs and list_DAWGs. The upper right chart is the construction times (in seconds) of TDAWGs, list_DAWGs and table_DAWGs. The lower chart is the searching time (in micro seconds) of TDAWGs, list_DAWGs and table_DAWGs

list_DAWGs. All the three algorithms to construct TDAWGs, table_DAWGs and list_DAWGs were implemented in the C language. All calculations were performed on a Laptop PC with PentiumIII-650MHz CPU and 256MB main memory running VineLinux2.6r2. We used the English text "ohsumed.91" available at http://trec.nist.gov/data.html.

The upper left chart of Fig. 5 shows memory requirements for TDAWGs and list_DAWGs, where memory spaces for both grow linearly, as expected. One can see that TDAWGs require about 20% more memory than list_DAWGs. The memory requirement of table_DAWGs is not shown since it is too much for the scale of the chart. The table_DAWG for the text of size 64KB required 98.82MB of memory space, and that for the text of size 128KB needed 197.53MB. Thus table_DAWGs are rather unusable in reality.

The second test was the construction times for TDAWGs, table_TDAWGs, and list_DAWGs, shown upper right of Fig. 5. One can see that the TDAWGs were constructed about twice faster than the list_DAWGs. This seems the effect of binary search in the TDAWGs, while the linked lists were linearly searched in the list_DAWGs. As for the table_DAWGs, though searching the transition can be done in constant time, the memory allocation for the tables seemed to take too much time.

The third test was searching times for patterns of different lengths. We randomly chose 100 substrings of the text for each length, and searched for every of them 1 million times. The result shown in the lower chart in Fig. 5 is the average time of searching for a pattern once. Remark that the TDAWGs are faster than list_DAWGs, even for longer patterns.

5   Conclusions and Further Work

In this paper we introduced a new data structure called *ternary directed acyclic word graphs* (*TDAWGs*). The process of binary search for the transitions is 'implanted' in each state of TDAWGs. For any string $w \in \Sigma^*$, $TDAWG(w)$ can be constructed in $O(|\Sigma| \cdot |w|)$ time using $O(|w|)$ space, in on-line fashion. Our experiments showed that TDAWGs can be constructed much faster than both DAWGs with tables and DAWGs with linked lists, for English texts. Moreover, searching time of TDAWGs is also better than that of DAWGs with linked lists. Thinking over the fact that TDAWGs are better in speed than the two other types of DAWGs though the alphabet size of the English text is only 256, TDAWGs should be a lot more effective when applied to texts of large alphabet such as Japanese, Korean, Chinese, and so on. 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 are promising to perform very well in practice.

As previously mentioned, the search time for pattern p using DAWGs with linked lists is $O(|\Sigma| \cdot |p|)$ in the worst case, but there is a way to improve it to $O(\log |\Sigma| \cdot |p|)$ time with additional effort for performing binary search. However, it is not practical since the additional work consumes a considerable amount of time, and is not implementation-friendly. Further, it does not allow us to update the input text sting since it is just an off-line algorithm. However, as for TDAWGs, we can apply the technique of *AVL trees* [1] for balancing the states of TDAWGs. In this scheme, the states for $CharSet_w(x)$ for any substring x are "AVL-balanced", and thus the time to search for pattern p is $O(\log |\Sigma| \cdot |p|)$ even in the worst case. The difference from DAWGs with linked lists is that we can construct "AVL-balanced" TDAWGs in *on-line* manner, *directly*, and can update the input text string easily. Moreover, the construction algorithm runs in $O(\log |\Sigma| \cdot |w|)$ time unlike the case of linked lists requiring $O(|\Sigma| \cdot |w|)$ time.
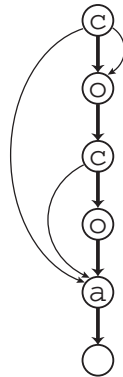
Fig. 6. The MTDAWG of string `cocoa`

Therefore, TDAWGs are more practical and flexible for guaranteeing $O(\log|\Sigma| \cdot |p|)$-time search in the worst case.

Moreover, there is a 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. 6, 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 for sure requires too much space. Thus we are now interested in an on-line algorithm to construct MTDAWGs directly, but it is still incomplete. We expect that searching for pattern strings using MTDAWGs will be faster than using TDAWGs, since memory allocation for MTDAWGs is likely to be more quick.

References

[1] G. M. Andelson-Velskii and E. M. Landis. An algorithm for the organisation of information. *Soviet. Math.*, 3:1259–1262, 1962. 128

[2] J. Bentley and B. Sedgewick. Ternary search trees. *Dr. Dobb's Journal*, 1998. `http://www.ddj.com/`. 121, 124

[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. 121, 124

[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. 121, 122, 123, 126

[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. 128

[6] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986. 122

[7] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, 1994. 122

[8] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002. 122

[9] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In *Structures in Logic and Computer Science*, volume 1261 of *LNCS*, pages 192–211. Springer-Verlag, 1997. 122, 128

[10] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997. 128

[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 A. Amir and G. M. Landau, editors, *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *LNCS*, pages 169–180. Springer-Verlag, 2001. 128

[12] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. 128

[13] D. Revuz. Minimization of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992. 123, 129

[14] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. 128

[15] E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, 1993. 122

[16] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973. 128