Construction of the CDAWG for a Trie

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

Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan

e-mail: {s-ine, hoshino, ayumi, takeda, arikawa}@i.kyushu-u.ac.jp

Abstract. Trie is a tree structure to represent a set of strings. When the strings have many common prefixes, the number of nodes in the trie is much less than the total length of the strings. In this paper, we propose an algorithm for constructing the Compact Directed Acyclic Word Graph for a trie, which runs in linear time and space with respect to the number of nodes in the trie.

Key words: Trie, Suffix trie, Suffix tree, DAWG, CDAWG

1 Introduction

Crochemore and Vérin displayed the relationship among suffix tries, suffix trees, Directed Acyclic Word Graphs (DAWGs), and Compact Directed Acyclic Word Graphs (CDAWGs) [CV97]. It implies that a suffix tree (DAWG, resp.) can be obtained by compacting (minimizing, resp.) the corresponding suffix trie. Similarly, a CDAWG can be obtained by either compacting the corresponding DAWG or minimizing the corresponding suffix tree.

It is known that all of these indexing structures for a string, except suffix trie that requires quadratic space, can be constructed in linear time and space: Weiner [Wei73], McCreight [McC76], and Ukkonen [Ukk95] for suffix trees, and Blumer et al. [BBH+85] for DAWGs.

Blumer et al. [BBH+87] gave an algorithm for constructing CDAWGs by compacting the corresponding DAWGs. Direct construction of CDAWGs from a given string is also important, since the hidden constant of the size complexity of CDAWGs is strictly smaller than those of suffix trees and DAWGs [BBH+87]. Actually, Crochemore and Vérin [CV97] gave the first algorithm that constructs CDAWGs directly from a given string, based on McCreight algorithm for suffix trees. Recently, Inenaga et al. [IHS+01b] developed an on-line algorithm for the direct construction of CDAWGs, which is based on the Ukkonen algorithm.

Their algorithm can also construct a CDAWG for a set S of strings in linear time with respect to the total length of the strings in S. In this paper, we consider the case that the set S is given as a trie, as input. Since the trie shares common prefixes of the strings in S, the number n of nodes of the trie is less than the total length of the strings. We show a non-trivial extension of the algorithm that constructs CDAWG for a trie, which runs in $O(n)$ time and space.

Some related works can be seen in literature: Kosaraju [Kos89] introduced the suffix tree for a *reversed* trie, and showed an algorithm to construct it in $O(n \log n)$

time. Breslauer [Bre98] reduced it to $O(n)$ time. On the other hand, our algorithm constructs a CDAWG for a (normal) trie. We remark that our algorithm can be easily adopted to construct Suffix trees and DAWGs instead of CDAWGs, with a slight modification in the same way that is mentioned in [IHS+01a]. That means, all of these indexing structures for trie can be constructed in linear time with respect to the number of nodes of the trie.

2 Preliminaries

The Compact Directed Acyclic Word Graph (CDAWG) can be seen as either the compaction of the Directed Acyclic Word Graph (DAWG), or the minimization of the suffix tree [BBH+87, CV97]. In this section, we recall the properties of CDAWGs, comparing with suffix trees.

2.1 Notations

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

Given a set S of strings, let $\|S\|$ represent the total length of the strings in S, and $|S|$ the cardinality of S. The sets of prefixes, factors, and suffixes of the strings in S are denoted by $\mathit{Prefix}(S)$, $\mathit{Factor}(S)$, and $\mathit{Suffix}(S)$, respectively.

2.2 Compact Directed Acyclic Word Graphs

We here recall the properties of CDAWGs, comparing with those of suffix trees. The suffix tree for a set S of strings is a rooted tree whose edges are labeled with strings in $\mathit{Factor}(S)$ (see Fig. 1). We denote by $\mathit{STree}(S)$ the suffix tree for S. We here assume that each string in S ends with a unique *endmarker* which never occurs in the string, so that any path from the root node spelling out a suffix of the string can end at a unique leaf in $\mathit{STree}(S)$. $\mathit{STree}(S)$ has the following properties:

1. It has one *root* node and $|\mathit{Suffix}(S)|$ *leaf* nodes.
2. Labels of any two edges leaving the same node do not begin with the same letter.
3. Any node but the leaf nodes has at least two outgoing edges.
4. Any string in $\mathit{Factor}(S)$ is spelled out along a certain path starting at the root node.
5. For any string $w \in S$, every string in $\mathit{Suffix}(w)$ is spelled out along a path starting at the root node and ending at the leaf node which corresponds to w.

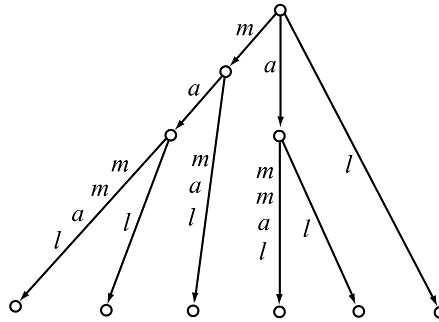

Figure 1: $STree(S)$ for $S = \{mammal\}$

The compact directed acyclic word graph (CDAWG) was first introduced by Blumer et al. [BBH$^+$87]. The CDAWG for a set S of strings is a directed acyclic graph with edges labeled by strings in $Factor(S)$ (see Fig. 2). We denote by $CDAWG(S)$ the CDAWG for S. $CDAWG(S)$ has the following properties:

1. It has an *initial* node and $|S|$ *final* nodes.
2. Labels of any two edges leaving the same node do not begin with the same letter.
3. Any node but the final nodes has at least two outgoing edges.
4. Any string in $Factor(S)$ is spelled out along a certain path starting at the initial node.
5. For any string $w \in S$, every string in $Suffix(w)$ is spelled out along a path starting at the initial node and ending at the final node which corresponds to w.
6. Suppose that the path spelling α ends at a node v. If a string β is always preceded by $\gamma \in \Sigma^*$ and $\alpha = \gamma\beta$ in any string $w \in S$ such that $\beta \in Factor(w)$, the path spelling out β also ends at the same node v.

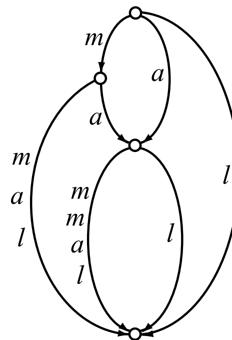

Figure 2: $CDAWG(S)$ for $S = \{mammal\}$

In Fig. 2, one can see that the path spelling a ends at the same node as the one spelling ma, with regard to the property 6 above. The reason is that a is always preceded by m in string $mammal$.

Theorem 1 (Blumer et al., [BBH+87]) *For any set S of strings, $CDAWG(S)$ has at most $\|S\| + |S|$ nodes.*

We hereafter denote by (u, α, v) the edge labeled α which starts at node u and ends at node v, both in CDAWGs and suffix trees.

2.3 Trie and Reversed Trie

Given a set $S = \{w_1\$, \ldots, w_k\$\}$ such that $w_i \notin \mathit{Suffix}(w_j)$ for any $1 \leq j \neq i \leq k$, the *reversed trie* for S is a rooted tree in which strings in $\mathit{Suffix}(S)$ are merged as long as possible [Bre98] (see Fig. 3), where \$ denotes an endmarker. We associate each node in a reversed trie with a unique number, as in Fig. 3. We write as $\mathit{Trie}^R(S)$ the reversed trie for a set S of strings. Every string in $\mathit{Prefix}(S)$ is spelled out along a path from a certain leaf node. The number of nodes in $\mathit{Trie}^R(S)$ is at most $\|S\| - |S| + 2$. In the case that the strings in S have long and many common suffixes, the number of nodes in $\mathit{Trie}^R(S)$ is by far smaller than $\|S\| - |S| + 2$.

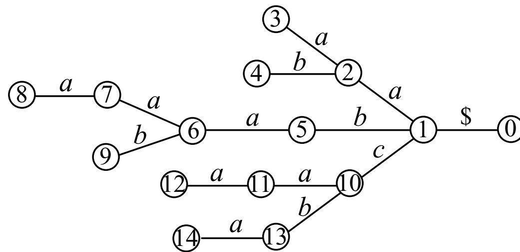

Figure 3: $\mathit{Trie}^R(S)$ for $S = \{aaab\$, aac\$, aa\$, abc\$, bab\$, ba\$\}$

On the other hand, given a set $S = \{w_1\$_1, \ldots, w_k\$_k\}$, where $\$_j$ denotes the endmarker for w_j $(1 \leq j \leq k)$, the *trie* for a set S of strings is a tree where possibly most strings in $\mathit{Prefix}(S)$ are merged with (see Fig. 4). We denote the trie for a set S by $\mathit{Trie}(S)$. It is easy to see that the number of nodes in $\mathit{Trie}(S)$ is at most $\|S\| + 1$. Thanks to the unique endmarkers, tries do not require the condition that reversed tries instead do. That is, given a set S of strings, even if a string $w \in S$ belongs to $\mathit{Prefix}(u)$ for some string $u \in S$, the path spelling out w ends at a leaf node in $\mathit{Trie}(S)$. For example, although string aa is a prefix of $aaab$ in Fig. 4, the path spelling out $aa\$_3$ ends at leaf node 8.

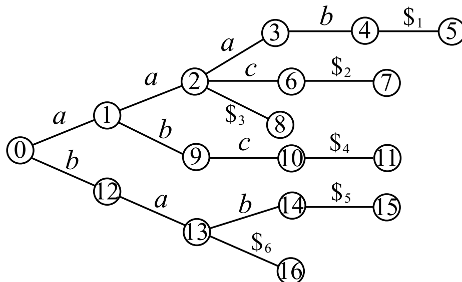

Figure 4: $\mathit{Trie}(S)$ for $S = \{aaab\$_1, aac\$_2, aa\$_3, abc\$_4, bab\$_5, ba\$_6\}$

3 Algorithm

In this section, we give an algorithm for constructing the CDAWG for a trie. The main idea of the algorithm is in the basis of the existing Inenaga's algorithm for constructing the CDAWG for a set of strings, and the depth-first search for a trie.

3.1 Algorithm Constructing CDAWG for a Set of Strings

By illustrating the construction of $CDAWG(ababcbc\$)$ in Fig. 5, we roughly recall the algorithm for the constructing the CDAWG for a set of strings, which was given in [IHS+01b]. More detailed description of the algorithm can be seen in [IHS+01a]. For simplicity, we put a single string to the input for the algorithm in Fig. 5.

Given a CDAWG, let us assume that the shortest path from the initial node to a node v spells out $\alpha = c\beta$, where $\alpha, \beta \in \Sigma^*$, and $c \in \Sigma$. Then, v has the *suffix link* that points to the node at which the path spelling out β ends. For example, at the phase *ababc* in Fig. 5, one can see that the suffix link of node 1, at which the path spelling out b ends, accordingly points to the initial node, to which ε corresponds. The suffix link of the initial node points to a special node, the *bottom* node, from which there are $|\Sigma|$ edges to the initial node, each of which is labeled with a letter in Σ. With the bottom node, we do not need to treat the initial node as an exception during the construction of the CDAWG. In Fig. 5, the bottom node, the initial node, and the final node are written as B, I, and F, respectively. Until the construction of the CDAWG for the whole input string is finished, the suffix link of the final node is left undefined (see the 1st phase to 8th), since it is possible that the node to which the suffix link of the final node points can change during the construction. If we update the suffix link of the final node at every phase, we cannot achieve the linear time algorithm. It can happen that a node u, which was the latest created in some phase, does not have the suffix link until another node is newly created in the phase, because the new node is just the one to which the suffix link of u should point.

The gray starred points in Fig. 5 represent *active points*, from which the algorithm begins updating the CDAWG at each phase. An active point p is represented by a pair of a node v and a string α, such that p can be reached from v by spelling α. For example, at phase *ababcb*, the active point is represented by $(2, c)$, whereas it is represented by $(2, \varepsilon)$ at phase *ababcb*. The active point of the current phase starts from the location where the active point stopped in the last phase. From now on, we sketch how the active point moves and stops in a phase. Assuming that the algorithm has already finished a phase γ, and it now faces the phase γc with $c \in \Sigma$, that is, a character c follows γ in the input string. If the active point is now on a node having an outgoing edge whose label is initialized by c, the active point advances just one letter a along the edge and stops over there. For example, it can be observed at phase *ab* and phase *aba* in Fig. 5. In the following phases, the active point keeps on stopping along the edge as long as possible (see the phase *abab*). If the active point is now on a node having no outgoing edge that begins with c, a new edge labeled c is created and connected from the node to the final node. The active point then moves to another node via the suffix link in order to check if it has an outgoing edge whose label begins with letter c. Otherwise, the active point is in the middle of an edge and is not followed by c in its label. In this case, a new node is created at the location

where the active point is, splits the edge into two over there, and creates a new edge labeled c to the final node. Then, the algorithm has to look for the location where the active point will move next.

See the phase *abab* and phase *ababc* in Fig. 5. As the active point of the phase *abab* can not move along the edge any longer because c dose not follow ab over there, a new node 1 is created and is then connected to the final node with an edge labeled c. After that, the algorithm has to find the location where the active point next moves. Since the node 1 does not have the suffix link yet, the active point at first moves backwards to node I that has the suffix link. After arriving at B via the suffix link of I, it resumes moving along the path spelling ab, where ab is the part of the label of the edge that the active point moved backwards. Notice that B has an edge labeled a. Although the path spelling ab from I consists of one edge, there is no guarantee that the path spelling ab from the bottom node to which the suffix link of I points also consists of one edge. In fact, as seen at the phase *abab*, the path spelling ab from the bottom node consists of two edges. Anyway, the active point finally arrives in the middle of edge (I, bab, F) while spelling out ab from the bottom node. Since the active point cannot move with spelling c from the current location, it seems necessary to create a new node and a new edge labeled c at the location, and it should enter into F. However, the fact is that b is always preceded by a in string *ababc* and it is obvious that the node 1 corresponding to ab has an edge having a label beginning with c. That is why edge (I, bab, F) is *merged* into node 1 with label b, that is, it becomes $(I, b, 1)$. After that, the active point again moves backwards to I and arrives at B via the suffix link of I. It then stops just on node I spelling out b. Creating a new edge (I, c, F), the active point moves to B via the suffix link, and then stops on I traversing along the edge labeled c. Thanks to the bottom node B, we can obtain the following lemma, similarly in [Ukk95].

Lemma 1 *For any string w and any i $(1 \leq i \leq |w|)$, CDAWG$(w[1:i-1])$ always has the location on which the active point of phase $w[1:i]$ stops.*

The above lemma holds in the case of a set of strings, as well.

The completely opposite thing to the edge merging above, the node *separation* can also happen, as seen at phase *ababcb* in Fig. 5. Recall that the active point was on I at the phase *ababc*. Then, as the letter b follows *ababc*, the active point moves to node 1. Note that it arrives at node 1 along the edge labeled b which does not compose the longest path from I to node 1. Then, node 1 is separated into two, that is, a new node 2 is created with the same outgoing edges as those of 1, and edge $(I, b, 1)$ becomes $(I, b, 2)$. The reason of the above is that b is not always preceded by a in string *ababcb*, though it was so in string *ababc*. If node 1 had incoming edges composing shorter paths from I to node 1 than the one which contains the edge the active point traveled, all of them would be also redirected to node 2.

Given a set $S = \{w_1, \ldots, w_k\}$, a label of any edge in $CDAWG(S)$ is implemented with a triple of integers (h, i, j) such that the label corresponds to $w_h[i:j]$. Let us hereafter call i and j *starting position* and *ending position*, respectively. We make the ending position of every edge which enters to the final node refer to the integer e set in the final node. With increasing e each time the CDAWG is extended with a new letter, we obtain the constant time update of all the edges entering to the final node.

Theorem 2 (Inenaga et al., [IHS+01b]) *For a fixed alphabet, the CDAWG for a*

set S of strings can be directly constructed in linear time and space with respect to $\|S\| + |S|$.

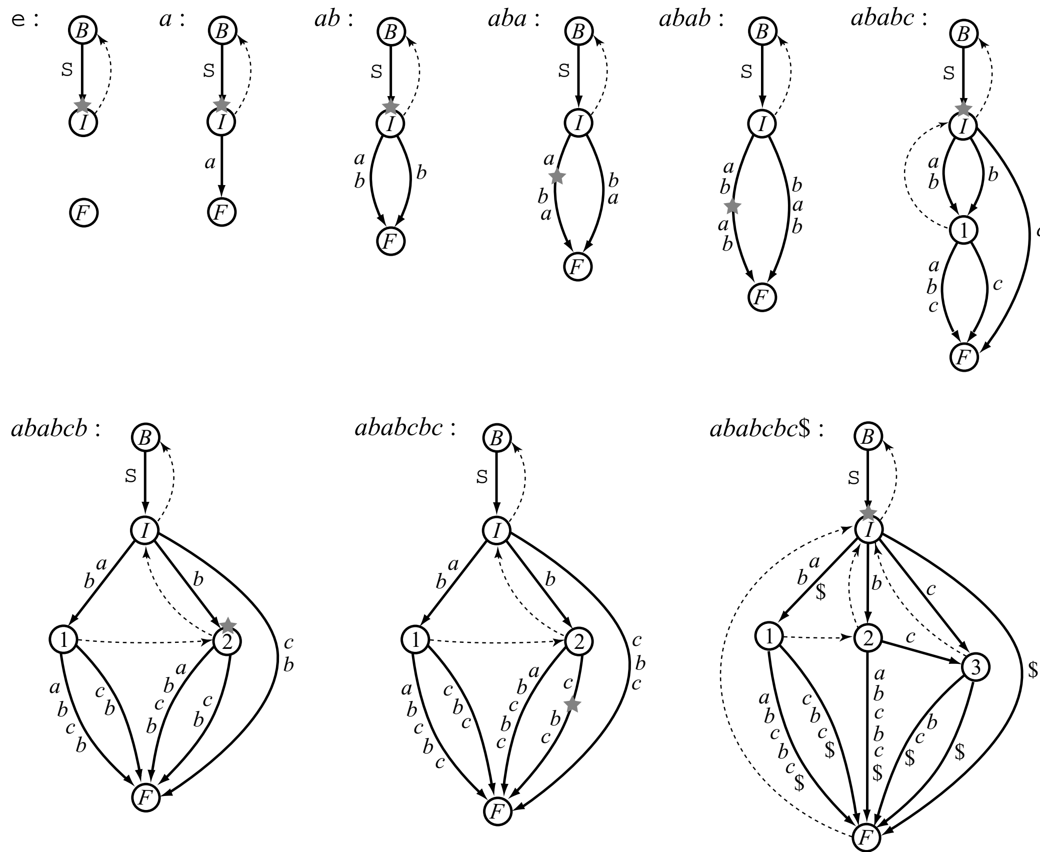

Figure 5: Construction of $CDAWG(w)$ for $w = ababcbc\$$. Broken lines represent the suffix links. Gray starred points represent the active points.

3.2 Algorithm Constructing CDAWG for a Trie

We are now ready to show our main algorithm that constructs CDAWGs for tries. First we note that the CDAWG for *Trie*(S) is the same as CDAWG for S, except only one thing. While the label of each edge in $CDAWG(S)$ is implemented with the triple of integers (h, i, j) representing the starting position i and ending position j of the label in the h-th string in S, that of the CDAWG for *Trie*(S) refers to the pair of nodes in *Trie*(S) such that the label corresponds to the path between them.

The basic action of the algorithm for *Trie*(S) is to update CDAWG incrementally, synchronized with the depth-first traversal of *Trie*(S). The key idea to achieve the linear time construction is the following:

(1) To trace the *advanced point* p in the CDAWG corresponding to the currently visited node v in the trie, so that the spell of p coincides with that of v.

(2) To create a new node in the CDAWG at the advanced point *beforehand* when stepping into the first branch at each branching node of the trie.

We will explain the detail in the sequel. Suppose that, after having traversed α along edges in *Trie*(S), it encounters a node v with k (≥ 2) branches in *Trie*(S). Moreover suppose that it then chooses an edge with which a path spelling β and ending at a leaf node begins. After updating the CDAWG according to the string $\alpha\beta$, the algorithm has to update it according to another string represented in *Trie*(S). Notice that the current CDAWG already has the path spelling α, which corresponds to prefixes of at least k strings in S. Therefore, the algorithm has only to restart updating the CDAWG at a node p which spells α and to continue traversing *Trie*(S) from the node v. For that purpose, we trace the *advanced point* p mentioned in (1) above.

Let us now clarify the aim of (2). The aim is to make the advanced point p be an *explicit node*. That is, the reference pair of p should be of the form (u, ϵ). What is the matter if the advanced point p is not explicit before stepping into the first branch? Assuming that the advanced point p is referred as (u, γ) with $\gamma \neq \varepsilon$, when the algorithm encountered the node v corresponding to α in *Trie*(S). After finishing updating the CDAWG with $\alpha\beta$, the algorithm focuses back to v and $p = (u, \gamma)$. The problem is that the reference (u, γ) might not be *canonized* no longer: the path spelling γ may contain extra nodes. Namely, the path spelling γ may have been split while the algorithm updated the CDAWG with string β. A concrete example is shown in Fig. 6.

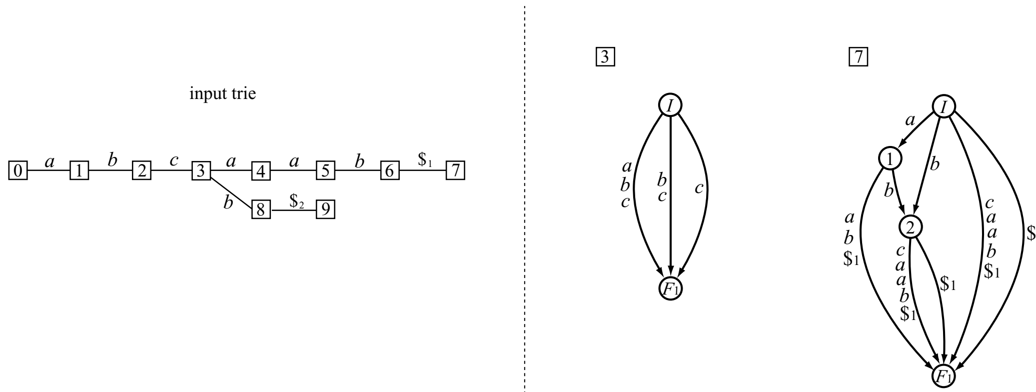

Figure 6: *Trie*(S) for $S = \{abcaab\$_1, abcb\$_2\}$ is shown left. When the algorithm focuses on node 3 in *Trie*(S), it needs to memorize the location in the CDAWG corresponding to *abc*. Since there is no node but F_1 at the location, it is memorized as a reference pair (I, abc). After having visited node 7 in *Trie*(S), the algorithm begins to update the CDAWG with (I, abc), and with node 3 in the trie. However, since the path spelling *abc* dose not consist of one edge any more, the algorithm has to find the nearest node from the location the path ends on, that is, node 2. As traversing the path spelling *abc* in the CDAWG just deserves traversing *Trie*(S) from node 0 to 3, the time complexity becomes $O(||S||)$, but not $n = |\mathit{Trie}(S)|$.

If the algorithm scans such extra nodes, its time complexity becomes quadratic with respect to the number of nodes in *Trie*(S). In order to avoid this, the algorithm creates the new node so that the active point is guaranteed to be an explicit node. However, it dose not merge any other edges because at the moment it is unknown how many edges should be merged into the new node. An example of the construction of the CDAWG for a trie is shown in Fig. 7.

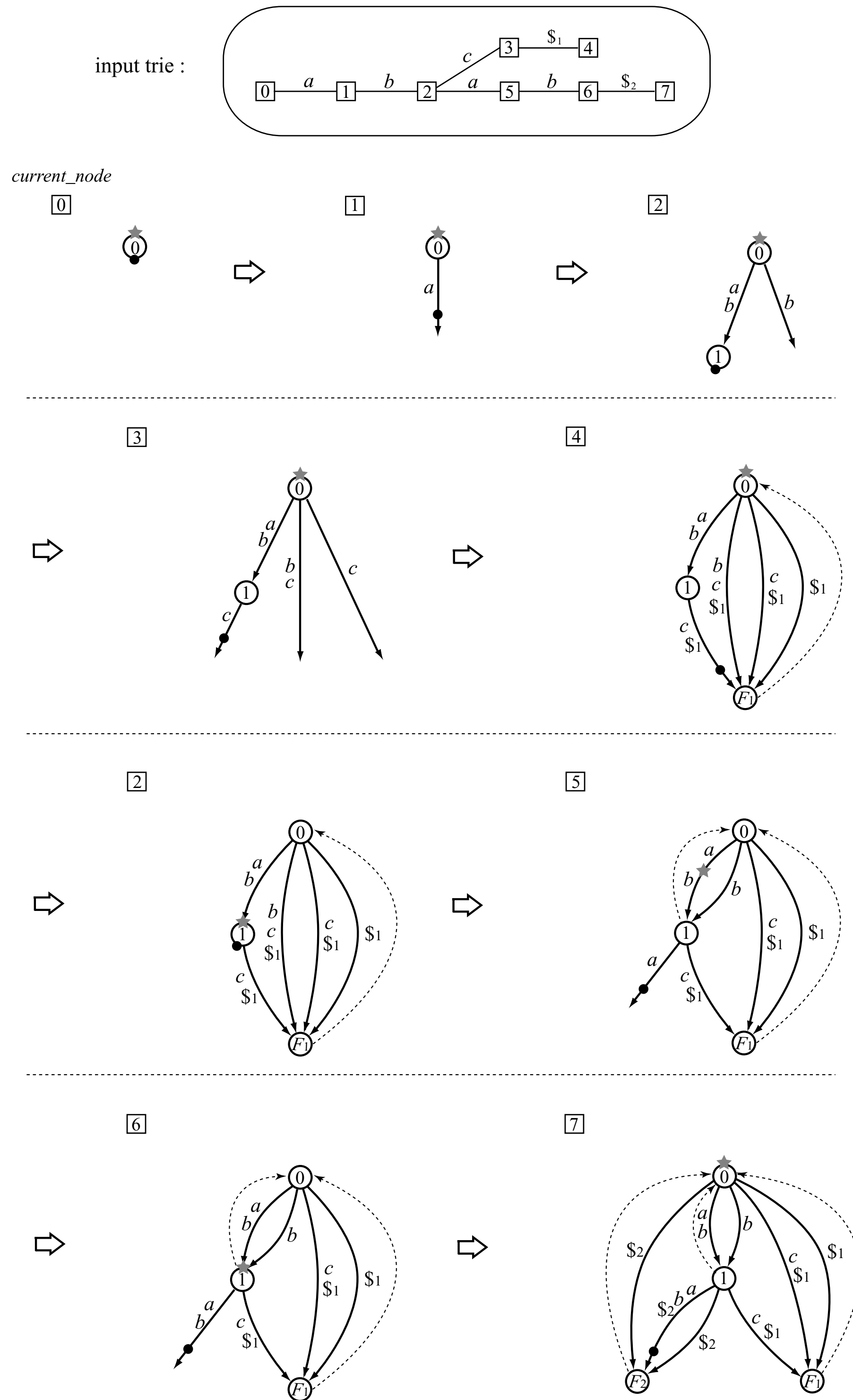

Figure 7: Construction of the CDAWG for $Trie(S)$, where $S = \{abc\$_1, abab\$_2\}$. The gray starred point represents *active_point*, and the black dotted point represents *advanced_point*. For simplicity, the bottom node is omitted. As node 2 in the trie is branching, a new node 1 is created in the CDAWG when *current_node* arrives at node 2 for the first time. After *current_node* visits node 4, the algorithm updates the CDAWG with *current_node* $= 2$ and *advanced_point* $= 1$.

The algorithm is described as follows.

main routine
current_node := 0; /* *the root node in the trie* */
active_point := (I, ε); /* *the initial node in the CDAWG* */
advanced_point := (I, ε); /* *the initial node in the CDAWG* */
traverse-and-update(*current_node*, *active_point*, *advanced_point*);

procedure *traverse-and-update*(*current_node*, *active_point*, *advanced_point*)
Let *label_set* be the set of labels of the outgoing edges of *current_node*;
if $|label_set| = 0$ **then return**;
else if $|label_set| \geq 2$ **then** *create-node*(*advanced_point*);
for each $c \in$ *label_set* **do**
new_active_point := *update-CDAWG*(c, *active_point*);
Let *new_advanced_point* be the location where *active_point* advances with c;
Let v be the node to which the edge labeled c points;
traverse-and-update(v, *new_active_point*, *new_advanced_point*);

The variable *current_node* indicates the node that the algorithm currently focuses in *Trie*(S). The variable *advanced_point* is of the form of a reference pair (q, β), where q is the parent node nearest to *advanced_point*. As mentioned above, the string β is actually implemented as a pair of nodes of *Trie*(S).

In the procedure *traverse-and-update*, the function *update-CDAWG* updates the CDAWG with a letter c. *update-CDAWG* is the same as the one for the construction of the CDAWG for a set of strings [IHS+01b, IHS+01a], excepting that *update-CDAWG* creates a new edge from the node latest created by function *create-node*. We now have the following theorem.

Theorem 3 *The proposed algorithm constructs the CDAWG for a trie in linear time and space complexity with respect to the number of nodes in the trie.*

Proof. We first explain that the modifications of the functions *update-CDAWG* and *create-node* themselves do not disturb the linearity.

Suppose that an input trie has n nodes. It is clear that the number of nodes visited by *advanced_point* in the CDAWG is at most n. Hence, it takes $O(n)$ time to calculate *advanced_point* all through the construction. Furthermore supposing that m nodes in *Trie*(S) are branching. It is clear that $m < n$, because any trie has at least one leaf node. Therefore, function *create-node* creates at most m nodes in the CDAWG, and it implies that the time complexity of *create-node* is $O(m)$.

We from now on verify the overall linearity of the proposed algorithm. The matter we have to clarify is the upper bound of the number of nodes *active_point* visits throughout the construction. Assume that a node v in the trie has k branches and there is a path spelling α between the root and v. When *current_node* arrives at node v in the trie for the first time, function *create-node* creates a new node p where *advanced_point* is in the CDAWG. Then, *active_point* may visit at most $k|\alpha|$ nodes from p to the initial node until finding the location it can stop on. However, $k \leq |\Sigma|$. Therefore, for a trie with n nodes, the number of nodes *active_point* visits throughout

the construction is $O(|\Sigma|n)$. Thus, if Σ is a fixed alphabet, the proposed algorithm constructs the CDAWG for a trie in $O(n)$ time and space. □

4 Conclusion

We gave an algorithm for constructing the CDAWG for a trie in linear time and space with respect to the number of nodes in the trie. With a slight modification, the proposed algorithm can be adopted to construct the suffix tree as well as DAWG for a trie. When the input strings are given in the form of a trie, the proposed algorithm constructs the CDAWG for the strings faster than the one presented in [IHS+01b] directly does from a set of the strings, especially when the strings have many common prefixes. As the space complexity of CDAWGs is bounded strictly lower than that of suffix trees, the algorithm presented in this paper also allows to save memory space.

References

[BBH+85] Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[BBH+87] Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987. Preliminary version in: STOC'84.

[Bre98] Dany Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theoritical Computer Science*, 191:131–144, 1998.

[CV97] Maxime Crochemore and Renaud Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, volume 1261 of *Lecture Notes in Computer Science*, pages 192–211. Springer-Verlag, 1997.

[IHS+01a] Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. On-line construction of compact directed acyclic word graphs. Technical Report DOI-TR-CS-183, Department of Informatics, Kyushu University, January 2001.

[IHS+01b] Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa, Giancarlo Mauri, and Giulio Pavesi. On-line construction of compact directed acyclic word graphs (to appear). In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, July 2001.

[Kos89] S. Rao Kosaraju. Fast pattern matching in trees. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 178–183, 1989.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, October 1973.