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: Pattern matching, Index structure, Trie, Suffix trie, Suffix tree, DAWG, CDAWG, Linear time algorithm

1 Introduction

Crochemore and Vérin displayed a relationship among suffix tries, suffix trees, Directed Acyclic Word Graphs (DAWGs), and Compact Directed Acyclic Word Graphs (CDAWGs) [CV97]. It states 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, except suffix tries, 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.

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

Their algorithm can also construct a CDAWG for a set S of strings in linear time with respect to the total length ℓ of the strings in S. In this paper, we consider the case that the set S is given in the form of 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 ℓ. We show a non-trivial extension of the algorithm that constructs CDAWG for a trie in $O(n)$ time and space.

Some related work 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 as in [IHS+01a]. That means, all of these indexing structures for a trie can be constructed in linear time with respect to the number of nodes in 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, compared with those of 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 *Prefix*(w), *Factor*(w), and *Suffix*(w), respectively. The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. 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 *Prefix*(S), *Factor*(S), and *Suffix*(S), respectively.

2.2 Compact Directed Acyclic Word Graphs

Here we show the properties of CDAWGs. For comparison, we first recall the properties of suffix trees. The suffix tree for a set S of strings is a rooted tree whose edges are labeled with strings in *Factor*(S) (see Fig. 1). We denote by *STree*(S) the suffix tree for S. We here assume that each string w_i in $S = \{w_1, \ldots, w_k\}$ ends with a unique *endmarker* $\$_i \notin \Sigma$, where $1 \leq i \leq k$. Let $S' = \{w_1\$_1, \ldots, w_k\$_k\}$. On the above assumption, every string in *Suffix*(S') is associated with a leaf node in *STree*(S'). *STree*(S') has the following properties:

1. It has a *root* node, at most $\|S\| - 1$ *internal* nodes, and $\|S\| + |S|$ *leaf* nodes.
2. The root node and any internal nodes have at least two outgoing edges.
3. Labels of any two edges leaving the same node do not begin with the same letter.
4. Any string in *Factor*(S) is represented by a path starting at the root node.
5. Any string in *Suffix*(S') is represented by a path starting at the root node and ending at a leaf node.

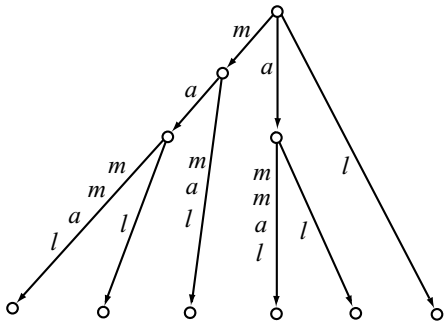

Figure 1: *STree*(S) for $S = \{mammal\}$. Since the last letter l plays the role of the endmarker, the endmarker is omitted.

The compact directed acyclic word graph (CDAWG) was first introduced by Blumer et al. [BBH$^+$87]. The CDAWG for a set S of strings is a directed acyclic graph with edges labeled by strings in *Factor*(S) (see Fig. 2). We denote by *CDAWG*(S) the CDAWG for S. We similarly assume that each string in S ends with a unique endmarker. It is because the algorithm for constructing the CDAWG for a set of strings, which was given in [IHS$^+$01b], requires the endmarkers. Remark that for any set S of strings *CDAWG*(S'), whose edges labeled with $\$_i$ are removed for any i ($1 \leq i \leq |S|$), is equal to the CDAWG of S. *CDAWG*(S') has the following properties:

1. It has an *initial* node, at most $\|S\|-1$ *internal* nodes, and $|S|$ *final* nodes.
2. The initial node and any internal nodes have at least two outgoing edges.
3. Labels of any two edges leaving the same node do not begin with the same letter.
4. Any string in *Factor*(S) is represented by a path starting at the initial node.
5. Any string in *Suffix*(S') is represented by a path starting at the initial node and ending at a final node.
6. Suppose that a path spelling out $\alpha \in \Sigma^*$ ends at a node v. If a string β is always preceded by $\gamma \in \Sigma^*$ and $\alpha = \gamma\beta$ in any string $x \in S'$ such that $\beta \in$ *Factor*(x), the path spelling out β also ends at node v.

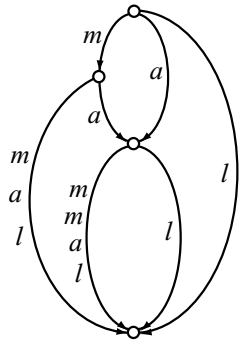

Figure 2: *CDAWG*(S) for $S = \{mammal\}$. Since the last letter l plays the role of the endmarker, the endmarker is omitted.

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

Hereafter, we denote by (u, α, v) an edge labeled with 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 $1 \leq i \neq j \leq k$, consider the set $S'' = \{w_1\$ \ldots w_k\$\}$ where \$ denotes the common endmarker. Then the *reversed trie* for S'' is a tree in which strings in $\mathit{Suffix}(S'')$ are merged as long and many as possible [Bre98] (see Fig. 3). 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 represented by a path beginning from a leaf node. The number of nodes in $\mathit{Trie}^R(S'')$ is at most $\|S''\| - |S''| + 2$. If the strings in S'' have long and many common suffixes, the number of nodes in $\mathit{Trie}^R(S'')$ is by far smaller than the upper bound $\|S''\| - |S''| + 2$.

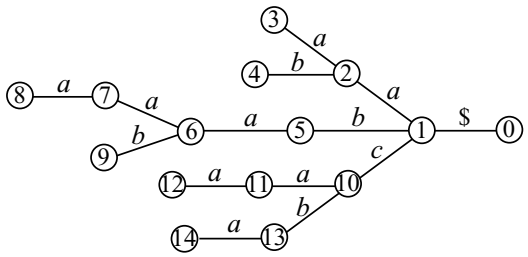

Figure 3: $\mathit{Trie}^R(S'')$ for $S'' = \{aaab\$, aac\$, aa\$, abc\$, bab\$, ba\$\}$

On the other hand, for a set $S' = \{w_1\$_1, \ldots, w_k\$_k\}$ where $\$_i$ denotes the endmarker for w_i $(1 \leq i \leq k)$, the *trie* for a set S' of strings is a tree in which strings in $\mathit{Prefix}(S')$ are merged as long and many as possible (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, even if a string $x \in S'$ belongs to $\mathit{Prefix}(y)$ for some string $y \in S'$, the path spelling out x always 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.

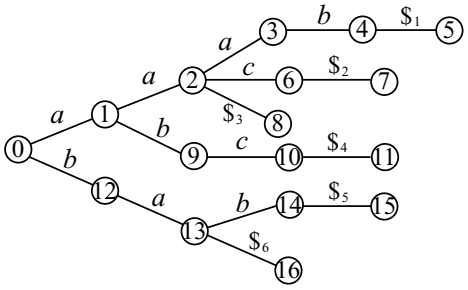

Figure 4: $\mathit{Trie}(S')$ for $S' = \{aaab\$_1, aac\$_2, aa\$_3, abc\$_4, bab\$_5, ba\$_6\}$

Tries are used as inputs of our algorithm that will be introduced in Section 4.

3 Algorithm to Construct the CDAWG for a Set of Strings

This section is devoted to recalling the algorithm to construct the CDAWG for a set of strings, which was proposed in [IHS+01b]. By illustrating the construction of *CDAWG*(*ababcbc*\$) in Fig. 5, we roughly show how the algorithm builds a CDWAG. More detailed description of the algorithm can be seen in [IHS+01a]. For simplicity, we have put a single string to the input of the algorithm in Fig. 5.

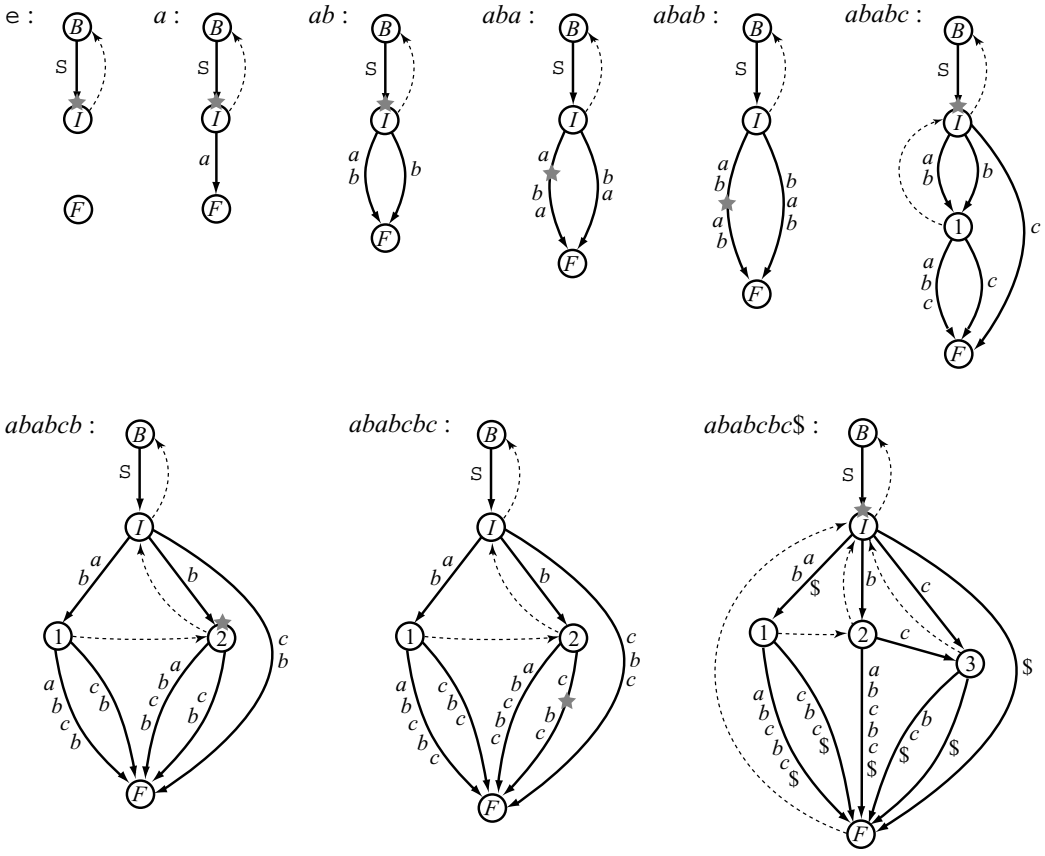

Figure 5: Construction of *CDAWG*(w) for w = *ababcbc*\$. The broken lines represent the suffix links, and the gray starred points represent the active points.

3.1 Suffix Links

As in literature [Wei73, McC76, Ukk95] about the construction of suffix trees, this algorithm to construct CDAWGs also utilizes *suffix links*. By means of the suffix links, the time complexity of these algorithms can be linear.

Let us assume that in a CDAWG 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, node v has the suffix link that points to a node u such that the path spelling out β is the longest path from the initial node to the node u. The algorithm determines the suffix link of each node during the construction of a CDAWG. For example, at 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 the empty string ε corresponds to. The suffix link of the initial node points to a special node, the *bottom* node, as in [Ukk95]. It has an edge labeled with Σ that represents an arbitrary letter in the alphabet Σ in this case. With this bottom node, we do not need to treat the initial node as an exception during the construction of a CDAWG. In Fig. 5, the bottom node, the initial node, and the final node are expressed by B, I, and F, respectively. Until the construction of the CDAWG of the whole input string is finished, the suffix link of the final node is left undefined (see the 1st phase to 8th in Fig 5), since the node to which the suffix link of the final node points can change during the construction. This implies that we cannot achieve a linear time algorithm if we update the suffix link of the final node at every phase. It can happen that a node r, 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 r should point.

3.2 The Active Point

The gray starred point in Fig. 5 denotes the location from which the algorithm starts to update the current CDAWG at the next phase. This is called *active point* similarly in [Ukk95]. An active point p is represented by the pair of a node v and a string α such that p can be reached from node v along the edge whose label begins with α. In the running example, the active point is represented by $(2, \varepsilon)$ at phase *ababcb*, whereas it is represented by $(2, c)$ at phase *ababcbc*.

From now on, we sketch how the active point moves and stops in a phase. Suppose that the algorithm has already finished a phase γ and now faces phase γc with $c \in \Sigma$, that is, a letter c follows γ in the input string. Then there can be four cases, that is, the active point is now:

(1) on a node that has an outgoing edge whose label begins with letter c;

(2) on a node that has no outgoing edge whose label begins with letter c;

(3) on the middle of an edge and followed by letter c in the label of the edge;

(4) on the middle of an edge and not followed by letter c in the label of the edge.

In case (1), the active point advances by letter c along the edge and then stops over there. This can be seen between phase *ab* and phase *aba* in Fig. 5. If it faces case (3) in the following phases, the active point keeps on moving and stopping along the edge, as seen in phase *abab*. In case (2), a new edge labeled with c is created and then connected from the node the active point is now on to the final node. The active point then moves to another node via the suffix link in order to check if there is an outgoing edge whose label begins with letter c. Finally, in case (4), a new node is created where the active point presents, splits the edge into two over there, and creates a new edge labeled with c from the new node to the final node. Then the algorithm has to look for the location where the active point will move next.

We illustrate how the algorithm behaves after facing case (4). See phase *abab* and phase *ababc* in Fig. 5. Since the active point of phase *abab* can not move along the edge any longer because c dose not follow *ab* there, new node 1 is created and then a

new edge labeled with c is created and connected to the final node. After that, the algorithm has to find the location where the active point next moves. Since node 1 does not have the suffix link yet, the active point moves backwards to node I that has the suffix link. After arriving at node B via the suffix link of node I, it resumes moving along the path corresponding to ab, where ab is the part of the label of the edge that the active point moved backwards. Remark that, although the one spelling out ab from node I consists of an edge, the path spelling out ab from node B 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.

3.3 Edge Merging

(The above story still continues here.) Since the active point cannot move with spelling out c from the current location, it seems necessary to create a new node and a new edge labeled with c over there. However, the fact is that letter b is always preceded by letter a in string $ababc$ and that node 1 which corresponds to ab obviously has an edge labeled 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 node B via the suffix link and then moves and stops on node I while spelling out c.

As seen above, thanks to the bottom node, 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 case of a set of strings, as well.

3.4 Node Separation

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

3.5 Update of Edges Entering to Final Node

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 $w_h[i:j]$, where $1 \leq h \leq k$. 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 in the final node. By 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.

As a result of the above discussion, the following theorem holds.

Theorem 1 (Inenaga et al., [IHS+01b]) *For a fixed alphabet, the CDAWG for a set S of strings can be directly constructed on-line, in linear time and space with respect to $\|S\|+|S|$.*

4 Algorithm to Construct the 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 the CDAWG of S for any set S of string, except only one thing. While the label of an edge in *CDAWG*(S) is implemented by a 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 in the CDAWG for *Trie*(S) refers to a pair of nodes in *Trie*(S), between of which there is the string corresponding to the label.

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

(1) Trace the *advanced point* q in the CDAWG so that the path from the root node to q coincides with the path from the root node to node v, where v is the node currently visited in the trie.

(2) Create a new node in the CDAWG where the advanced point q is, before stepping into the first branch at each branching node in the trie.

We will explain the detail in the sequel. Suppose that, after having traveled nodes with scanning $\alpha \in \Sigma^*$ in *Trie*(S), the algorithm encounters a node v having k (≥ 2) branches in *Trie*(S). Moreover suppose that it then chooses an edge with which a path spelling out β and ending at a leaf node begins. After updating the CDAWG with string $\alpha\beta$, the algorithm has to update it with the other strings represented in *Trie*(S). Notice that the current CDAWG already has the path representing α from the initial node, which corresponds to prefixes of at least k strings in S. Thus the algorithm has only to restart updating the CDAWG from the location to which α corresponds and to continue traversing *Trie*(S) from the node v. For that purpose, we trace the *advanced point* q mentioned in (1) above.

Let us now clarify the aim of (2). The aim is to make the advanced point q be an *explicit* node whenever the algorithm encounters a branching node in *Trie*(S). That is, the reference pair of q should then become of the form (s, ε) for some node s. What is the matter if the advanced point q is not explicit before stepping into the first branch? Assume that the advanced point q was referred as (u, γ) with some node u and string $\gamma \neq \varepsilon$ 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

on v and $q=(u,\gamma)$. The matter is that the reference (u,γ) might not be *canonical* any longer: the path spelling out γ may contain extra nodes. Namely, the path spelling out γ may have been split while the algorithm updated the CDAWG with string β. A concrete example is shown in Fig. 6.

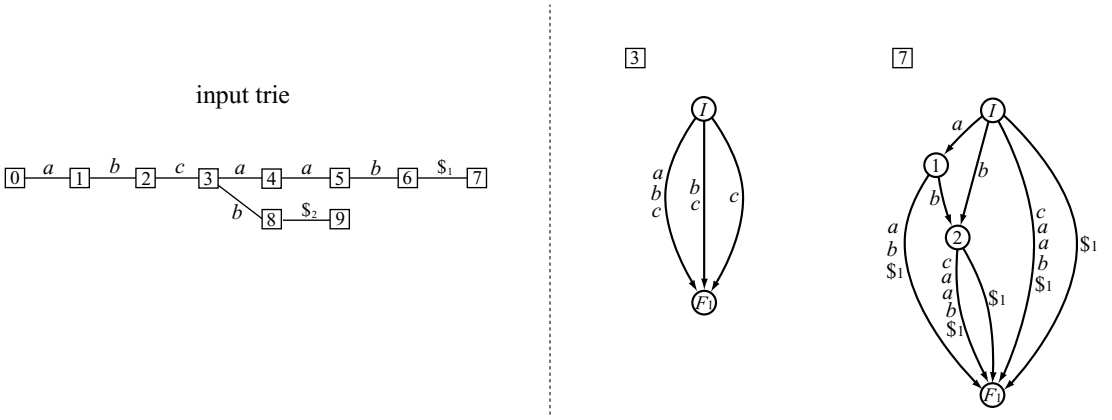

Figure 6: *Trie*(S) for $S=\{abcaab\$_1, abcb\$_2\}$ is shown left. When the algorithm focuses on node 3 in *Trie*(S), it needs to memorize the location in the CDAWG corresponding to *abc*. Since there is no node but F_1 at the location, it is memorized by a reference pair (I, abc). After having visited node 7 in *Trie*(S), the algorithm updates the CDAWG from (I, abc), and with node 3 in the trie. However, since the path spelling *abc* dose not consist of an edge any more, the algorithm has to find the nearest node from the location the path ends on, that is, node 2. We have to avoid this, because traversing the path spelling *abc* in the CDAWG just deserves traversing *Trie*(S) from node 0 to 3.

If the algorithm scans such extra nodes, its time complexity can become quadratic with respect to the number of nodes in *Trie*(S). In order to avoid this matter, the algorithm creates a new node s so that the active point is guaranteed to be on an explicit node. However, the algorithm dose not merge any other edges because at the moment it is unknown how many edges should be merged into the new node s. Of course, if $\gamma=\varepsilon$, there is no need to create any new node.

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 on in *Trie*(S). The variable *advanced_point* is of the form of a reference pair (u, β), where u is the parent node nearest to *advanced_point*. As mentioned above, the string β is actually implemented by a pair of nodes in *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 stemming from the node latest created by function *create-node*.

An example of the construction of the CDAWG for a trie is shown in Fig. 7.

Finally, we have the following theorem.

Theorem 2 *The proposed algorithm constructs the CDAWG for a trie in linear time and space with respect to the number of nodes in the trie.*

Proof. We first explain that the modification of the function *update-CDAWG* and the function *create-node* itself do not affect the linearity of the algorithm.

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 suppose 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 $O(m)$. This implies the modification, creating new edges due to the nodes made by function *create-node*, takes $O(m)$ time as well.

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 u where *advanced_point* is in the CDAWG. Then *active_point* may traverse at most $k|\alpha|$ nodes from p to the initial node via suffix links 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. □

5 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. The truth is, with a slight modification, the proposed algorithm can be adopted to construct the suffix tree and the DAWG for a trie. When 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

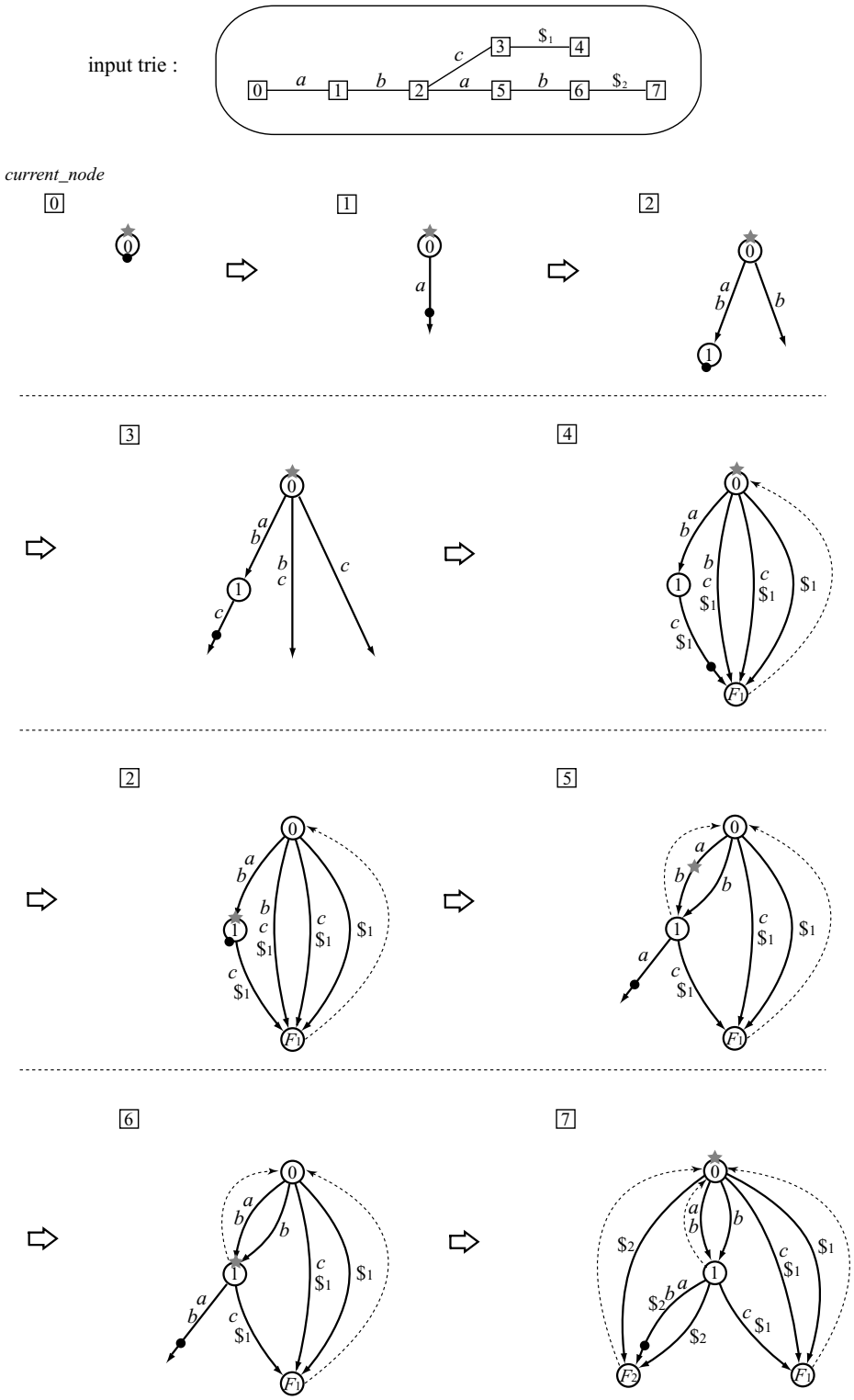

Figure 7: Construction of the CDAWG for $Trie(S)$, where $S = \{abc\$_1, abab\$_2\}$. The gray starred point represents *active_point*, and the black dotted point represents *advanced_point*. For simplicity, the bottom node is omitted. As node 2 in the trie is branching, a new node 1 is created in the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

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. *Theoretical 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.