Electronic Preprint for Journal of Information Processing Vol.29

60th Anniversary Best Paper

Towards a Complete Perspective on Labeled Tree Indexing: New Size Bounds, Efficient Constructions, and Beyond

Shunsuke Inenaga[1,2,a)]

Received: March 9, 2020, Accepted: August 28, 2020

Abstract: A labeled tree (or a trie) is a natural generalization of a string which can also be seen as a compact representation of a set of strings. This paper considers the labeled tree indexing problem, and provides a number of new results on space bound analysis and on algorithms for efficient construction and pattern matching queries. Kosaraju [FOCS 1989] was the first to consider the labeled tree indexing problem and he proposed the suffix tree for a backward trie, where the strings in the trie are read in the leaf-to-root direction. In contrast to a backward trie, we call an ordinary trie as a forward trie. Despite a few follow-up works after Kosaraju's paper, indexing forward/backward tries is not well understood yet. In this paper, we show a full perspective on the sizes of indexing structures such as suffix trees, DAWGs, CDAWGs, suffix arrays, affix trees, affix arrays for forward and backward tries. Some of them take $O(n)$ space in the size n of the input trie, while the others can occupy $O(n^2)$ space in the worst case. In particular, we show that the size of the DAWG for a forward trie with n nodes is $\Omega(\sigma n)$, where σ is the number of distinct characters in the trie. This becomes $\Omega(n^2)$ for an alphabet of size $\sigma = \Theta(n)$. Still, we show that there is a compact $O(n)$-space implicit representation of the DAWG for a forward trie, whose space requirement is independent of the alphabet size. This compact representation allows for simulating each DAWG edge traversal in $O(\log \sigma)$ time, and can be constructed in $O(n)$ time and space over any integer alphabet of size $O(n)$. In addition, this readily extends to the *first* indexing structure that permits *bidirectional pattern searches* over a trie within linear space in the input trie size. We also discuss the size of the DAWG built on a *labeled DAG* or on an *acyclic DFA*, and present a quadratic lower bound for its size.

Keywords: string indexing structures, pattern matching, indexing labeled trees, SDDs

1. Introduction

Strings are an abstract data type for any data in which the order of the items matters. Strings cover a wide range of sequential data, e.g., natural language text, biological sequences, temporal data, time series, event sequences, and server logs. Due to recent developments of sensor networks, M2M communications, and high-throughput sequencing technologies, string data have been increasing more rapidly than ever before.

String indexing is a fundamental problem in theoretical computer science, where the task is to preprocess a text string so that subsequent pattern matching queries on the text can be answered quickly. This is the basis of today's large-scale information retrieval systems and databases. The first of such string indexing structures was the suffix tree, which was introduced by Weiner in 1973 [43]. Suffix trees also have numerous other applications including string comparisons [43], data compression [3], data mining [37], and bioinformatics [21], [33].

A *labeled tree* is a static rooted tree where each edge is labeled by a single character. A *trie* is a kind of labeled tree such that the out-going edges of each node are labeled by mutually distinct characters. A trie can be naturally seen as an *acyclic deterministic finite-state automaton* (*acyclic DFA*) in a tree shape, which accepts a finite set of strings. In this sense, a trie can be seen as a compact representation of a set of strings. In another view, a trie is a generalization of a string that is a labeled single-path tree.

This paper considers the *labeled tree indexing problem*, where the task is to build a data structure that supports sub-path queries to report all sub-paths in the trie that match a given string pattern. Such sub-path queries on labeled trees are primitive in data base searches where the XML has been used as a *de facto* format of data storage. It is well known that each XML document forms a tree[*1]. Other important uses of labeled trees includes SQL queries [23], dictionaries [1], [45], and data compression [44], [47], to mention just a few.

A *backward* trie is an edge-reversed trie, where we read the path strings in the leaf-to-root direction. Kosaraju [28] was the first to consider the trie indexing problem, and he proposed the suffix tree of a backward trie that takes $O(n)$ space[*2], where n is the number of nodes in the backward trie. Kosaraju also claimed

1 Department of Informatics, Kyushu University, Fukuoka 819–0395, Japan
2 PRESTO, Japan Science and Technology Agency, Kawaguchi, Saitama 332–0012, Japan
a) inenaga@inf.kyushu-u.ac.jp

*1 Usually, a string label is associated to a *node* in an XML tree structure. However, for such a tree, there is an obvious corresponding trie where each *edge* is labeled by a single character.
*2 We evaluate the space usage of algorithms and data structures by the number of machine words (not bits) unless otherwise stated.

© 2021 Information Processing Society of Japan

an $O(n \log n)$-time construction. Breslauer [11] showed how to build the suffix tree of a backward trie in $O(\sigma n)$ time and space, where σ is the alphabet size. Shibuya [40] presented an $O(n)$-time and space construction for the suffix tree of a backward trie over an integer alphabet of size $\sigma = O(n)$. This line of research has been followed by the invention of XBWTs [15], suffix arrays [15], enhanced suffix arrays [27], and position heaps [38] for backward tries.

1.1 Suffix Trees, DAWGs, and CDAWGs for Forward/ Backward Tries

In addition to the suffix trees, we also consider the *directed acyclic word graphs* (*DAWGs*) and the *compact DAWGs* (*CDAWGs*), which were introduced by Blumer et al. in 1985 [9], and in 1987 [10], respectively. Similar to suffix trees, DAWGs and CDAWGs support linear-time pattern matching queries. For any string of length m, the suffix tree, the DAWG, and the CDAWG contain linear $O(m)$ nodes and $O(m)$ edges [9], [10], [43]. These bounds have been generalized to a set of strings such that, for any set of strings of total length M, the (generalized) suffix tree, the DAWG, and the CDAWG contain linear $O(M)$ nodes and $O(M)$ edges [10], [22]. All these bounds are independent of the alphabet size σ.

The DAWG of a string w is the smallest DFA that accepts the suffixes of w [9]. DAWGs for strings have important applications including pattern matching with don't cares [29], online Lempel-Ziv factorization in compact space [46], finding minimal absent words in optimal time [18], and dynamic multiple pattern matching [24]. The CDAWG of a string w can be regarded as *grammar compression* for w and can be stored in $O(e)$ space, where e denotes the number of right-extensions of maximal repeats in w which can be much smaller than the string length in highly repetitive strings [4]. There is also an space-efficient suffix tree representation based on the CDAWG [4]. Hence, understanding DAWGs and CDAWGs for labeled trees is very important and will likely lead to further advances in efficient processing of labeled trees.

To this end, this paper initiates size analysis on these indexing structures for a forward (ordinary) trie and a backward trie. We show that, quite interestingly, some of the aforementioned size bounds *do not* generalize to the case of tries. We present *tight* lower and upper bounds on the sizes of all these indexing structures, as summarized in **Table 1**. Our size analysis is based on combinatorial properties that reside in these indexing structures, such as the *duality* of suffix trees and DAWGs, and *maximal repeats* on input tries, and is not straightforward from the known bounds for strings.

Let n denote the number of nodes in a given forward trie and in the corresponding backward trie. Our new bounds are summarized as follows:

- We first present a (folklore) result such that the number of nodes and the number of edges of the suffix tree for a forward trie are both $O(n^2)$ for any forward tries, and are $\Omega(n^2)$ for some tries. These bounds are independent of the alphabet size σ.
- As direct consequences to the aforementioned results on suffix trees, the sizes of the *suffix arrays* for a forward trie and a backward trie are $O(n^2)$ and $O(n)$, respectively. These upper bounds are also tight.
- The number of nodes in the DAWG for a forward trie is known to be $O(n)$ [36], however, it was left open how many edges the DAWG can contain. It is trivially upper bounded by $O(\sigma n)$, since any node can have at most σ children in DAWGs. We show that this upper bound is tight by presenting a worst-case instance that gives $\Omega(\sigma n)$ edges in the DAWG for a forward trie. Since this lower bound is valid for alphabet size σ from $\Theta(1)$ to $\Theta(n)$, we obtain an $\Omega(n^2)$ worst-case size bound for the DAWG of a forward trie.
- We show that the DAWG of a backward trie shares the same nodes with the suffix tree of the corresponding forward trie under reversal of substrings. This immediately leads to $O(n^2)$ and $\Omega(n^2)$ bounds for the numbers of nodes and edges of the DAWG for a backward trie, independently of the alphabet size.
- The CDAWG of a forward trie and the CDAWG of its corresponding backward trie also share the same nodes under reversal of substrings. This leads us to $O(n)$ bounds for the numbers of nodes in both of these CDAWGs. However, the number of edges can differ by at most a factor of n: The CDAWG of a forward trie contains $O(\sigma n)$ and $\Omega(\sigma n)$ edges in the worst case, but the CDAWG of a backward trie contains only $O(n)$ edges independently of the alphabet size. We remark that the $\Omega(\sigma n)$ lower bound for the CDAWG edges for the forward trie is valid for alphabet size σ ranging from $\Theta(1)$ to $\Theta(n)$, and hence, it can contain $\Omega(n^2)$ edges in the worst case.

Table 1 Summary of the numbers of nodes and edges of the suffix tree, DAWG, and CDAWG for a forward/backward trie with n nodes over an alphabet of size σ. The new bounds obtained in this paper are highlighted in bold. All the bounds here are valid with any alphabet size σ ranging from $\Theta(1)$ to $\Theta(n)$. Also, all these upper bounds are tight in the sense that there are matching lower bounds (see Section 5).

	forward trie		backward trie	
index structure	# of nodes	# of edges	# of nodes	# of edges
suffix tree	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n^2)}$	$O(n)$	$O(n)$
DAWG	$O(n)$	$\boldsymbol{O(\sigma n)}$	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n^2)}$
CDAWG	$\boldsymbol{O(n)}$	$\boldsymbol{O(\sigma n)}$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$
suffix array	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n^2)}$	$O(n)$	$O(n)$

1.2 Implicit $O(n)$-size Representation of the DAWG for Forward Trie

Probably the most interesting result in our size bounds is the $\Omega(n^2)$ lower bound for the size of the DAWG for a forward trie with n nodes over an alphabet of size $\Theta(n)$ (Theorem 6): Mohri et al. [36] proposed an algorithm that constructs the DAWG for a forward trie with n nodes in time linear in the *output* size. Following our $\Omega(n^2)$-size lower bound, Mohri et al.'s construction must take at least $\Omega(n^2)$ time and space in the worst case.

Now, one may wonder whether or not it is possible to store the DAWG for a forward trie in a compact manner, within linear space, in the size of the input trie, in case of large alphabets. Somewhat surprisingly, the answer to this challenging question is positive. In this paper, we propose an *implicit compact represen-*

© 2021 Information Processing Society of Japan

tation of the DAWG for a forward trie that occupies only $O(n)$ space independently of the alphabet size, and allows for simulating traversal of each DAWG edge in $O(\log \sigma)$ time. We emphasize that this is the *first* linear-space representation of *any* DFA that accepts all substrings of a given labeled tree, after 35 years from the seminal paper by Blumer et al. in 1985 for the DAWG of a string [9]. In addition, we present an algorithm that builds this implicit representation of the DAWG for a forward trie in $O(n)$ time and space for any integer alphabet of size $O(n)$. Our data structure of an implicit representation of the DAWG is carefully designed upon combinatorial properties of *Weiner links* that are defined on the suffix tree of the corresponding backward trie. Also, our algorithm does not require use of any additional complicated data structures.

1.3 Relation to Indexing Automata and Labeled DAGs

Mohri et al. [36] considered the problem of indexing a given acyclic DFA (or equivalently a labeled DAG). They proposed DAWGs (a.k.a. suffix automata) for acyclic DFAs, and claimed that the number of *nodes* in the DAWG for a given acyclic DFA with n states is $O(n)$. However, they did not consider the number of *edges* in the DAWG of an acyclic DFA.

A *sequence binary decision diagram* (*SDD*) [32] is a family of Zero-Suppressed BDD (ZDD) [35] that represents a set of strings. Roughly speaking, an SDD is a simple representation of a labeled DAG where the branches are implemented by a list that begins with the leftmost child and continues to the right siblings. SDDs are known to support a rich class of manipulations to the set of strings, which can quickly be performed in practice. Denzumi et al. [14] proposed an SDD-based indexing structure, named *PosFSDDdag*, for an input labeled DAG which is also given in an SDD form. PosFSDDdags can be regarded as a representation of DAWGs or suffix automata for acyclic DFAs, with the leftmost-child-and-right-siblings representation of branches. PosFSDDdags can be quite space-efficient in practice, but theoretical size bounds of PosFSDDdag were not well understood in the literature [14].

Notice that forward tries are the simplest kind of acyclic DFAs. Thus, our analysis also resolves the aforementioned open questions in terms of the worst case bounds. Namely, our $\Omega(n^2)$ size bound for the DAWG of a forward trie with n nodes over an alphabet of size $\sigma = \Theta(n)$ immediately leads to a worst-case quadratic size for DAWGs for input acyclic DFAs. Figure 10 illustrates a lower bound instance for DAWGs. The number of edges of the broom-like input trie in Fig. 10 must be preserved in its SDD representation. Moreover, even if we merge the sink nodes of the DAWG in Fig. 10 into a single sink, the number of edges in the DAWG still does not decrease. Thus the worst-case size of PosFSDDdags is at least $\Omega(n^2)$ for some input SDD of size n.

1.4 Bidirectional Pattern Search on Tries

In the bidirectional pattern search problem, the characters of a query pattern is given in a *bidirectional online manner*, added either to the left end or to the right end of the current pattern. This enables us to perform very flexible pattern searches so that given a current pattern, one can look for its left or right contexts in a dynamic manner. Bidirectional pattern searches in strings have other various applications, including: high-throughput short read alignments [31], discovery of secondary structures in RNA sequences [20], [34], [39], [42], multiple pattern matching [20], and approximate pattern matching [30].

Every existing bidirectional indexing structure for a string consists of a pair of two indexing structures, one built on a forward string (to be read from left to right) and the other on the corresponding backward string (to be read from right to left). *Affix trees*, proposed by Stoye in 2000 [41], are the first bidirectional indexing structure for strings, followed by the *affix arrays* [42] counterparts. *Bidirectional BWTs* [39] are the most widely used bidirectional search structure nowadays, since they can be stored in compact or succinct space [5], [6].

Because a huge set of biological sequences can be compactly stored as a tree in a practical database system [8], bidirectional pattern search algorithms that work on tries are of high significance. However, it seems that none of these existing bidirectional indexing structures can readily be generalized to tries. See also Table 1. Since affix trees (resp. affix arrays) contain at least the same number of nodes (resp. entries) as the corresponding suffix trees (resp. suffix arrays), the sizes of the affix tree/array built on a pair of froward and backward tries turn out to be $O(n^2)$ and $\Omega(n^2)$ in the worst case. It is not obvious if one can generalize BWT to a forward trie either, because the corresponding suffix array contains $O(n^2)$ entries.

We focus our attention to suffix trees and DAWGs. Using the duality of suffix trees and DAWGs [10], one can indeed perform bidirectional pattern searches. Therefore, in the case where σ is a constant (e.g., $\sigma = 4$ for DNA/RNA sequences), the combination of the DAWG for a forward trie and the suffix tree for a backward trie readily gives us a $O(n)$-space bidirectional index for a trie. However, when the underlying alphabet is very large, which is common in temporal sequences (time series) and Japanese/Chinese texts, the $O(\sigma n)$-space requirement is prohibitive. Still, our $O(n)$-space implicit representation of the DAWG for a forward trie enables efficient bidirectional pattern searches on a trie for a large alphabet of size $\sigma = O(n)$. To our knowledge, this is the *first* linear-space bidirectional indexing structure for labeled trees, after 20 years from the invention of affix trees for strings in 2000 [41].

A portion of the results reported in this paper appeared in Ref. [25].

2. Preliminaries

Let Σ be an ordered alphabet. Any element of Σ^* is called a *string*. For any string S, let $|S|$ denote its length. Let ε be the empty string, namely, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. If $S = XYZ$, then X, Y, and Z are called a *prefix*, a *substring*, and a *suffix* of S, respectively. For any $1 \le i \le j \le |S|$, let $S[i..j]$ denote the substring of S that begins at position i and ends at position j in S. For convenience, let $S[i..j] = \varepsilon$ if $i > j$. For any $1 \le i \le |S|$, let $S[i]$ denote the ith character of S. For any string S, let $\overline{S}$ denote the reversed string of S, i.e., $\overline{S} = S[|S|] \cdots S[1]$. Also, for any set $\mathbf{S}$ of strings, let $\overline{\mathbf{S}}$ denote the set of the reversed strings of $\mathbf{S}$, namely, $\overline{\mathbf{S}} = \{\overline{S} \mid S \in \mathbf{S}\}$.

© 2021 Information Processing Society of Japan

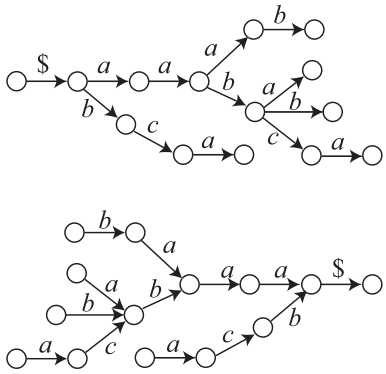

Fig. 1 A forward trie $\mathsf{T_f}$ (upper) and its corresponding backward trie $\mathsf{T_b}$ (lower).

A *trie* T is a rooted tree (V, E) such that (1) each edge in E is labeled by a single character from Σ and (2) the character labels of the out-going edges of each node begin with mutually distinct characters. In this paper, a *forward trie* refers to an (ordinary) trie as defined above. On the other hand, a *backward trie* refers to an edge-reversed trie where each path label is read in the leaf-to-root direction. We will denote by $\mathsf{T_f} = (\mathsf{V_f}, \mathsf{E_f})$ a forward trie and by $\mathsf{T_b} = (\mathsf{V_b}, \mathsf{E_b})$ the backward trie that is obtained by reversing the edges of $\mathsf{T_f}$. We denote by a triple $(u, a, v)_\mathsf{f}$ an edge in a forward trie $\mathsf{T_f}$, where $u, v \in \mathsf{V}$ and $a \in \Sigma$. Each reversed edge in $\mathsf{T_b}$ is denoted by a triple $(v, a, u)_\mathsf{b}$. Namely, there is a directed labeled edge $(u, a, v)_\mathsf{f} \in \mathsf{E_f}$ iff there is a reversed directed labeled edge $(v, a, u)_\mathsf{b} \in \mathsf{E_b}$. See **Fig. 1** for examples of $\mathsf{T_f}$ and $\mathsf{T_b}$.

For a node u of a forward trie $\mathsf{T_f}$, let $anc(u, j)$ denote the jth ancestor of u in $\mathsf{T_f}$ if it exists. Alternatively, for a node v of a backward $\mathsf{T_b}$, let $des(v, j)$ denote the jth descendant of v in $\mathsf{T_b}$ if it exists. We use a *level ancestor* data structure [7] on $\mathsf{T_f}$ (resp. $\mathsf{T_b}$) so that $anc(u, j)$ (resp. $des(v, j)$) can be found in $O(1)$ time for any node and integer j, with linear space.

For nodes u, v in a forward trie $\mathsf{T_f}$ s.t. u is an ancestor of v, let $str_\mathsf{f}(u, v)$ denote the string spelled out by the path from u to v in $\mathsf{T_f}$. Let r denote the root of $\mathsf{T_f}$ and $\mathsf{L_f}$ the set of leaves in $\mathsf{T_f}$. The sets of substrings and suffixes of the forward trie $\mathsf{T_f}$ are respectively defined by

$$Substr(\mathsf{T_f}) = \{str_\mathsf{f}(u, v) \mid u, v \in \mathsf{V_f},\ u \text{ is an ancestor of } v\},$$
$$Suffix(\mathsf{T_f}) = \{str_\mathsf{f}(u, l) \mid u \in \mathsf{V_f},\ l \in \mathsf{L_f}\}.$$

For nodes v, u in a backward trie $\mathsf{T_b}$ s.t. v is a descendant of u, let $str_\mathsf{b}(v, u)$ denote the string spelled out by the reversed path from v to u in $\mathsf{T_b}$. The sets of substrings and suffixes of the backward trie $\mathsf{T_b}$ are respectively defined by

$$Substr(\mathsf{T_b}) = \{str_\mathsf{b}(v, u) \mid v, u \in \mathsf{V_b},\ v \text{ is a descendant of } u\},$$
$$Suffix(\mathsf{T_b}) = \{str_\mathsf{b}(v, r) \mid v \in \mathsf{V_b},\ r \text{ is the root of } \mathsf{T_b}\}.$$

In what follows, let n be the number of nodes in $\mathsf{T_f}$ (or equivalently in $\mathsf{T_b}$).

Fact 1. (*a*) *For any* $\mathsf{T_f}$ *and* $\mathsf{T_b}$, $Substr(\mathsf{T_f}) = \overline{Substr(\mathsf{T_b})}$. (*b*) *For any forward trie* $\mathsf{T_f}$, $|Suffix(\mathsf{T_f})| = O(n^2)$. *For some forward trie* $\mathsf{T_f}$, $|Suffix(\mathsf{T_f})| = \Omega(n^2)$. (*c*) $|Suffix(\mathsf{T_b})| \le n - 1$ *for any backward trie* $\mathsf{T_b}$.

Fact 1-(a), Fact 1-(c) and the upper bound of Fact 1-(b) should be clear from the definitions. To see the lower bound of Fact 1-(b) in detail, consider a forward trie $\mathsf{T_f}$ with root r such that there is a single path of length k from r to a node v, and there is a complete binary tree rooted at v with k leaves (see also **Fig. 2**). Then, for all nodes u in the path from r to v, the total number of strings in the set $\{str_\mathsf{f}(u, l) \mid l \in \mathsf{L_f}\} \subset Suffix(\mathsf{T_f})$ is at least $k(k + 1)$, since each $str_\mathsf{f}(u, l)$ is distinct for each path (u, l). By setting $k \approx n/3$ so that the number $|\mathsf{V_f}|$ of nodes in $\mathsf{T_f}$ equals n, we obtain Fact 1-(b). The lower bound is valid for alphabets of size σ ranging from 2 to $\Theta(k) = \Theta(n)$.

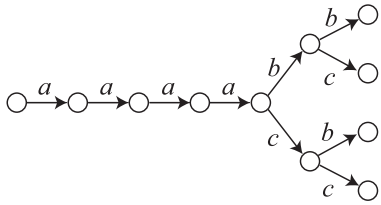

Fig. 2 Forward trie $\mathsf{T_f}$ containing distinct suffixes $a^i\{b, c\}^{\log_2(\frac{n+1}{3})}$ for all i $(0 \le i \le k = (n + 1)/3)$, which sums up to $k(k + 1) = \Omega(n^2)$ distinct suffixes. In this example $k = 4$.

Remark. In case the input labeled tree is given as a compact tree (a.k.a. Patricia tree) where each edge is labeled by a non-empty string and every internal node is branching, then the input size n should be translated to the total length of the string labels in the compact tree. Then all of the the following bounds also hold for input compact tries of size n.

3. Maximal Substrings in Forward/Backward Tries

Blumer et al. [10] introduced the notions of right-maximal, left-maximal, and maximal substrings in a set $\mathbf{S}$ of strings, and presented clean relationships between the right-maximal/left-maximal/maximal substrings and the suffix trees/DAWGs/CDAWGs for $\mathbf{S}$. Here we give natural extensions of these notions to substrings in our forward and backward tries $\mathsf{T_f}$ and $\mathsf{T_b}$, which will be the basis of our indexing structures for $\mathsf{T_f}$ and $\mathsf{T_b}$.

3.1 Maximal Substrings on Forward Tries

For any substring X in a forward trie $\mathsf{T_f}$, X is said to be *right-maximal* on $\mathsf{T_f}$ if (i) there are at least two distinct characters $a, b \in \Sigma$ such that $Xa, Xb \in Substr(\mathsf{T_f})$, or (ii) X has an occurrence ending at a leaf of $\mathsf{T_f}$. Also, X is said to be *left-maximal* on $\mathsf{T_f}$ if (i) there are at least two distinct characters $a, b \in \Sigma$ such that $aX, bX \in Substr(\mathsf{T_f})$, or (ii) X has an occurrence beginning at the root of $\mathsf{T_f}$. Finally, X is said to be *maximal* on $\mathsf{T_f}$ if X is both right-maximal and left-maximal in $\mathsf{T_f}$. In the example of Fig. 1 (upper), bc is left-maximal but is not right-maximal, ca is right-maximal but not left-maximal, and bca is maximal.

For any $X \in Substr(\mathsf{T_f})$, let $r\text{-}mxml_\mathsf{f}(X)$, $l\text{-}mxml_\mathsf{f}(X)$, and $mxml_\mathsf{f}(X)$ respectively denote the functions that map X to the shortest right-maximal substring $X\beta$, the shortest left-maximal substring αX, and the shortest maximal substring $\alpha X\beta$ that contain X in $\mathsf{T_f}$, where $\alpha, \beta \in \Sigma^*$.

3.2 Maximal Substrings on Backward Tries

For any substring Y in a backward trie $\mathsf{T_b}$, Y is said to be

© 2021 Information Processing Society of Japan

left-maximal on $\mathsf{T_b}$ if (i) there are at least two distinct characters $a, b \in \Sigma$ such that $aY, bY \in \mathit{Substr}(\mathsf{T_b})$, or (ii) Y has an occurrence beginning at a leaf of $\mathsf{T_b}$. Also, Y is said to be *right-maximal* on $\mathsf{T_b}$ if (i) there are at least two distinct characters $a, b \in \Sigma$ such that $Ya, Yb \in \mathit{Substr}(\mathsf{T_b})$, or (ii) Y has an occurrence ending at the root of $\mathsf{T_b}$. Finally, Y is said to be *maximal* on $\mathsf{T_b}$ if Y is both right-maximal and left-maximal in $\mathsf{T_b}$. In the example of Fig. 1 (lower), *baaa* is left-maximal but not right-maximal, *aaa*\$ is right-maximal but not left-maximal, and *baa* is maximal.

For any $Y \in \mathit{Substr}(\mathsf{T_b})$, let $\mathit{l\text{-}mxml}_{\mathsf{b}}(Y)$, $\mathit{r\text{-}mxml}_{\mathsf{b}}(Y)$, and $\mathit{mxml}_{\mathsf{b}}(Y)$ respectively denote the functions that map Y to the shortest left-maximal substring γY, the shortest right-maximal substring $Y\delta$, and the shortest maximal substring $\gamma Y \delta$ that contain Y in $\mathsf{T_b}$, where $\gamma, \delta \in \Sigma^*$.

Clearly, the afore-mentioned notions are symmetric over $\mathsf{T_f}$ and $\mathsf{T_b}$, namely:

Fact 2. *String X is right-maximal (resp. left-maximal) on $\mathsf{T_f}$ iff $\overline{X}$ is left-maximal (resp. right-maximal) on $\mathsf{T_b}$. Also, X is maximal on $\mathsf{T_f}$ iff $\overline{X}$ is maximal on $\mathsf{T_b}$.*

4. Indexing Structures for Forward/Backward Tries

In this section, we give formal definitions of indexing structures for forward/backward tries.

A compact tree for a set **S** of strings is a rooted tree such that (1) each edge is labeled by a non-empty substring of a string in **S**, (2) each internal node is branching, (3) the string labels of the out-going edges of each node begin with mutually distinct characters, and (4) there is a path from the root that spells out each string in **S**, which may end on an edge. Each edge of a compact tree is denoted by a triple (u, α, v) with $\alpha \in \Sigma^+$. We call internal nodes that are branching as *explicit nodes*, and we call loci that are on edges as *implicit nodes*. We will sometimes identify nodes with the substrings that the nodes represent.

In what follows, we will consider DAG or tree data structures built on a forward trie or backward trie. For any DAG or tree data structure D, let $|\mathsf{D}|_{\#Node}$ and $|\mathsf{D}|_{\#Edge}$ denote the numbers of nodes and edges in D, respectively.

4.1 Suffix Trees for Forward Tries

The *suffix tree* of a forward trie $\mathsf{T_f}$, denoted $\mathsf{STree}(\mathsf{T_f})$, is a compact tree which represents $\mathit{Suffix}(\mathsf{T_f})$. See **Fig. 3** for an example.

The nodes of $\mathsf{STree}(\mathsf{T_f})$ of Fig. 3 represent the right-maximal substrings in $\mathsf{T_f}$ of Fig. 1, e.g., *aab* is right-maximal since it is immediately followed by *a*, *b*, *c* and also it ends at a leaf in $\mathsf{T_f}$. Hence *aab* is a node in $\mathsf{STree}(\mathsf{T_f})$. On the other hand, *aabc* is not right-maximal since it is immediately followed only by *c* and hence it is not a node $\mathsf{STree}(\mathsf{T_f})$.

All non-root nodes in $\mathsf{STree}(\mathsf{T_f})$ represent right-maximal substrings on $\mathsf{T_f}$. Since now all internal nodes are branching, and since there are at most $|\mathit{Suffix}(\mathsf{T_f})|$ leaves, both the numbers of nodes and edges in $\mathsf{STree}(\mathsf{T_f})$ are proportional to the number of suffixes in $\mathit{Suffix}(\mathsf{T_f})$. The following (folklore) quadratic bounds hold due to Fact 1-(b).

Theorem 1. *For any forward trie $\mathsf{T_f}$ with n nodes, $|\mathsf{STree}(\mathsf{T_f})|_{\#Node} = O(n^2)$ and $|\mathsf{STree}(\mathsf{T_f})|_{\#Edge} = O(n^2)$. These upper bounds hold for any alphabet. For some forward trie $\mathsf{T_f}$ with n nodes, $|\mathsf{STree}(\mathsf{T_f})|_{\#Node} = \Omega(n^2)$ and $|\mathsf{STree}(\mathsf{T_f})|_{\#Edge} = \Omega(n^2)$. These lower bounds hold for a constant-size or larger alphabet.*

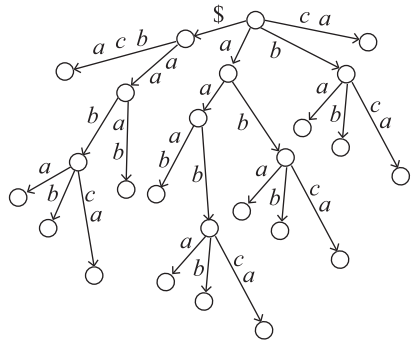

Fig. 3 $\mathsf{STree}(\mathsf{T_f})$ for the forward trie $\mathsf{T_f}$ of Fig. 1.

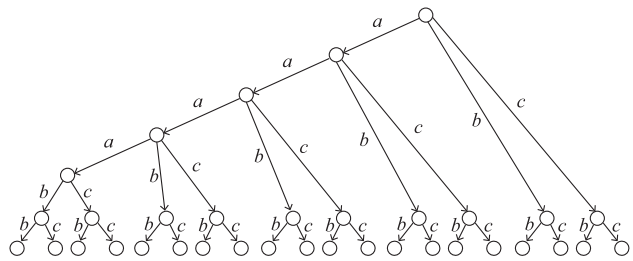

Fig. 4 $\mathsf{STree}(\mathsf{T_f})$ for the forward trie $\mathsf{T_f}$ of Fig. 2, which contains $k(k+1) = \Omega(n^2)$ nodes and edges where n is the size of this $\mathsf{T_f}$. In the example of Fig. 2, $k = 4$ and hence $\mathsf{STree}(\mathsf{T_f})$ here has $4 \cdot 5 = 20$ leaves. It is easy to modify the instance to a binary alphabet so that the suffix tree still has $\Omega(n^2)$ nodes.

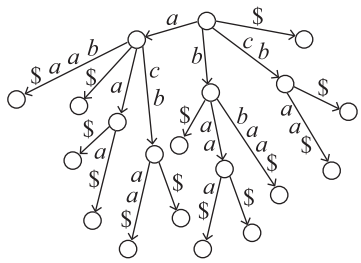

Fig. 5 $\mathsf{STree}(\mathsf{T_b})$ for the backward trie $\mathsf{T_b}$ of Fig. 1.

Figure 4 shows an example of the lower bounds of Theorem 1.

4.2 Suffix Trees for Backward Tries

The *suffix tree* of a backward trie $\mathsf{T_b}$, denoted $\mathsf{STree}(\mathsf{T_b})$, is a compact tree which represents $\mathit{Suffix}(\mathsf{T_b})$.

See **Fig. 5** for an example.

The nodes of $\mathsf{STree}(\mathsf{T_b})$ in Fig. 5 represent the right-maximal substrings in $\mathsf{T_b}$ of Fig. 1, e.g., *acb* is right-maximal since it is immediately followed by *a* and \$. Hence *acb* is a node in $\mathsf{STree}(\mathsf{T_b})$. On the other hand, *ac* is not right-maximal since it is immediately followed only by *c* and hence it is not a node $\mathsf{STree}(\mathsf{T_b})$.

Since $\mathsf{STree}(\mathsf{T_b})$ contains at most $n-1$ leaves by Fact 1-(c) and all internal nodes of $\mathit{Suffix}(\mathsf{T_b})$ are branching, the following precise bounds follow from Fact 1-(c), which were implicit in the literature [11], [28].

Theorem 2. *For any backward trie $\mathsf{T_b}$ with $n \geq 3$ nodes, $|\mathsf{STree}(\mathsf{T_b})|_{\#Node} \leq 2n-3$ and $|\mathsf{STree}(\mathsf{T_b})|_{\#Edge} \leq 2n-4$, independently of the alphabet size.*

The above bounds are tight since the theorem translates to the

© 2021 Information Processing Society of Japan

suffix tree with $2m-1$ nodes and $2m-2$ edges for a string of length m (e.g., $a^{m-1}b$), which can be represented as a path tree with $n = m+1$ nodes. By representing each edge label α by a pair $\langle v, u\rangle$ of nodes in $\mathsf{T_b}$ such that $\alpha = str_{\mathsf{b}}(u, v)$, $\mathsf{STree}(\mathsf{T_b})$ can be stored with $O(n)$ space.

Suffix Links and Weiner Links: For each explicit node aU of the suffix tree $\mathsf{STree}(\mathsf{T_b})$ of a backward trie $\mathsf{T_b}$ with $a \in \Sigma$ and $U \in \Sigma^*$, let $slink(aU) = U$. This is called the *suffix link* of node aU. For each explicit node V and $a \in \Sigma$, we also define the *reversed suffix link* $\mathcal{W}_a(V) = aVX$ where $X \in \Sigma^*$ is the shortest string such that aVX is an explicit node of $\mathsf{STree}(\mathsf{T_b})$. $\mathcal{W}_a(V)$ is undefined if $aV \notin Substr(\mathsf{T_b})$. These reversed suffix links are also called as *Weiner links* (or *W-link* in short) [12]. A W-link $\mathcal{W}_a(V) = aVX$ is said to be *hard* if $X = \varepsilon$, and *soft* if $X \in \Sigma^+$. The suffix links, hard and soft W-links of nodes in the suffix tree $\mathsf{STree}(\mathsf{T_f})$ of a forward trie $\mathsf{T_f}$ are defined analogously.

Suffix Arrays of Forward/Backward Tries

An array representation of the list of the leaves of $\mathsf{STree}(\mathsf{T_f})$ (resp. $\mathsf{STree}(\mathsf{T_b})$) sorted in the lexicographical order is the *suffix array* of the forward trie $\mathsf{T_f}$ (resp. backward trie $\mathsf{T_b}$). Hence, the following corollaries are immediate from Theorem 1 and Theorem 2, respectively:

Corollary 1. *For any forward trie $\mathsf{T_f}$ with n nodes, the size of the suffix array for $\mathsf{T_f}$ is $O(n^2)$. This upper bound holds for any alphabet. For some forward trie $\mathsf{T_f}$ with n nodes, the size of the suffix array for $\mathsf{T_f}$ is $\Omega(n^2)$. This lower bound holds for a constant-size or larger alphabet.*

Corollary 2 (Ref. [15])**.** *For any backward trie $\mathsf{T_b}$ with n nodes, the size of the suffix array is exactly $n-1$.*

4.3 DAWGs for Forward Tries

The *directed acyclic word graph* (*DAWG*) of a forward trie $\mathsf{T_f}$ is a (partial) DFA that recognizes all substrings in $Substr(\mathsf{T_f})$. Hence, the label of every edge of $\mathsf{DAWG}(\mathsf{T_f})$ is a single character from Σ. $\mathsf{DAWG}(\mathsf{T_f})$ is formally defined as follows: For any substring X from $Substr(\mathsf{T_f})$, let $[X]_{E,\mathsf{f}}$ denote the equivalence class w.r.t. $l\text{-}mxml_{\mathsf{f}}(X)$. There is a one-to-one correspondence between the nodes of $\mathsf{DAWG}(\mathsf{T_f})$ and the equivalence classes $[\cdot]_{E,\mathsf{f}}$, and hence we will identify the nodes of $\mathsf{DAWG}(\mathsf{T_f})$ with their corresponding equivalence classes $[\cdot]_{E,\mathsf{f}}$.

See **Fig. 6** for an example.

The nodes of $\mathsf{DAWG}(\mathsf{T_f})$ of Fig. 6 represent the equivalence classes w.r.t. the left-maximal substrings in $\mathsf{T_f}$ of Fig. 1, e.g., *aab* is left-maximal since it is immediately followed by *a* and \$ and hence it is the longest string in the node that represents *aab*. This node also represents the suffix *ab* of *aab*, since $l\text{-}mxml_{\mathsf{f}}(ab) = aab$.

By the definition of equivalence classes, every member of $[X]_{E,\mathsf{f}}$ is a suffix of $l\text{-}mxml_{\mathsf{f}}(X)$. If X, Xa are substrings in $Substr(\mathsf{T_f})$ and $a \in \Sigma$, then there exists an edge labeled with character $a \in \Sigma$ from node $[X]_{E,\mathsf{f}}$ to node $[Xa]_{E,\mathsf{f}}$ in $\mathsf{DAWG}(\mathsf{T_f})$. This edge is called *primary* if $|l\text{-}mxml_{\mathsf{f}}(X)| + 1 = |l\text{-}mxml_{\mathsf{f}}(Xa)|$, and is called *secondary* otherwise. For each node $[X]_{E,\mathsf{f}}$ of $\mathsf{DAWG}(\mathsf{T_f})$ with $|X| \geq 1$, let $slink([X]_{E,\mathsf{f}}) = Z$, where Z is the longest suffix of $l\text{-}mxml_{\mathsf{f}}(X)$ not belonging to $[X]_{E,\mathsf{f}}$. This is the *suffix link* of this node $[X]_{E,\mathsf{f}}$.

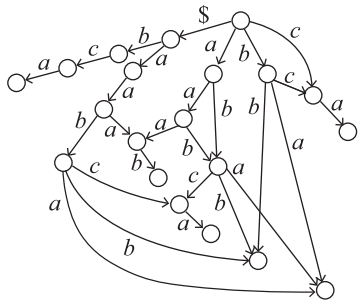

Fig. 6 $\mathsf{DAWG}(\mathsf{T_f})$ for the forward trie $\mathsf{T_f}$ of Fig. 1.

Mohri et al. [36] introduced the *suffix automaton* for an acyclic DFA G, which is a small DFA that represents all suffixes of strings accepted by G. They considered equivalence relation $\equiv$ of substrings X and Y in an acyclic DFA G such that $X \equiv Y$ iff the following paths of the occurrences of X and Y in G are equal. Mohri et al.'s equivalence class is identical to our equivalence class $[X]_{E,\mathsf{f}}$ when $\mathsf{G} = \mathsf{T_f}$. To see why, recall that $l\text{-}mxml_{\mathsf{f}}(X) = \alpha X$ is the shortest substring of $\mathsf{T_f}$ such that αX is left-maximal, where $\alpha \in \Sigma^*$. Therefore, X is a suffix of $l\text{-}mxml_{\mathsf{f}}(X)$ and the following paths of the occurrences of X in $\mathsf{T_f}$ are identical to the following paths of the occurrences of $l\text{-}mxml_{\mathsf{f}}(X)$ in $\mathsf{T_f}$. Hence, in the case where the input DFA G is in form of a forward trie $\mathsf{T_f}$ such that its leaves are the accepting states, then Mohri et al.'s suffix automaton is identical to our DAWG for $\mathsf{T_f}$. Mohri et al. [36] showed the following:

Theorem 3 (Corollary 2 of Ref. [36])**.** *For any forward trie $\mathsf{T_f}$ with $n \geq 3$ nodes, $|\mathsf{DAWG}(\mathsf{T_f})|_{\#Node} \leq 2n-3$, independently of the alphabet size.*

We remark that Theorem 3 is immediate from Theorem 2 and Fact 2. This is because there is a one-to-one correspondence between the nodes of $\mathsf{DAWG}(\mathsf{T_f})$ and the nodes of $\mathsf{STree}(\mathsf{T_b})$, which means that $|\mathsf{DAWG}(\mathsf{T_f})|_{\#Node} = |\mathsf{STree}(\mathsf{T_b})|_{\#Node}$. Recall that the bound in Theorem 3 is only on the number of *nodes* in $\mathsf{DAWG}(\mathsf{T_f})$. We shall show later that the number of *edges* in $\mathsf{DAWG}(\mathsf{T_f})$ is $\Omega(\sigma n)$ in the worst case, which can be $\Omega(n^2)$ for a large alphabet.

4.4 DAWGs for Backward Tries

The DAWG of a backward trie $\mathsf{T_b}$, denoted $\mathsf{DAWG}(\mathsf{T_b})$, is a (partial) DFA that recognizes all strings in $Substr(\mathsf{T_b})$. The label of every edge of $\mathsf{DAWG}(\mathsf{T_b})$ is a single character from Σ. $\mathsf{DAWG}(\mathsf{T_b})$ is formally defined as follows: For any substring Y from $Substr(\mathsf{T_b})$, let $[Y]_{E,\mathsf{b}}$ denote the equivalence class w.r.t. $l\text{-}mxml_{\mathsf{b}}(Y)$. There is a one-to-one correspondence between the nodes of $\mathsf{DAWG}(\mathsf{T_b})$ and the equivalence classes $[\cdot]_{E,\mathsf{b}}$, and hence we will identify the nodes of $\mathsf{DAWG}(\mathsf{T_b})$ with their corresponding equivalence classes $[\cdot]_{E,\mathsf{b}}$.

See **Fig. 7** for an example.

The nodes of $\mathsf{DAWG}(\mathsf{T_b})$ in Fig. 7 represent the equivalence classes w.r.t. the left-maximal substrings in $\mathsf{T_b}$ of Fig. 1, e.g., *ac* is left-maximal since it begins at a leaf in $\mathsf{T_b}$, and hence it is the longest string in the node that represents *ac*. This node also represents the suffix *c* of *ac*, since $l\text{-}mxml_{\mathsf{b}}(c) = ac$.

The notions of primary edges, secondary edges, and the suffix links of $\mathsf{DAWG}(\mathsf{T_b})$ are defined in a similar manner to $\mathsf{DAWG}(\mathsf{T_f})$, by using the equivalence classes $[Y]_{E,\mathsf{b}}$ for substrings Y in the

© 2021 Information Processing Society of Japan

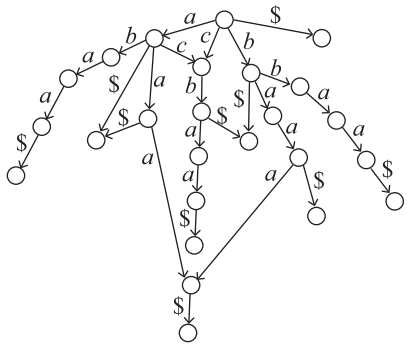

Fig. 7 DAWG($\mathsf{T_b}$) for the backward trie $\mathsf{T_b}$ of Fig. 1.

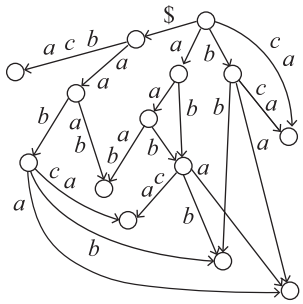

Fig. 8 CDAWG($\mathsf{T_f}$) for the forward trie $\mathsf{T_f}$ of Fig. 1.

backward trie $\mathsf{T_b}$.

Symmetries between Suffix Trees and DAWGs: The well-known *symmetry* between the suffix trees and the DAWGs (Refs. [9], [10], [13]) also holds in our case of forward and backward tries. Namely, the suffix links of DAWG($\mathsf{T_f}$) (resp. DAWG($\mathsf{T_b}$)) are the (reversed) edges of STree($\mathsf{T_b}$) (resp. STree($\mathsf{T_f}$)). Also, the hard W-links of STree($\mathsf{T_f}$) (resp. STree($\mathsf{T_b}$)) are the primary edges of DAWG($\mathsf{T_b}$) (resp. DAWG($\mathsf{T_f}$)), and the soft W-links of STree($\mathsf{T_f}$) (resp. STree($\mathsf{T_b}$)) are the secondary edges of DAWG($\mathsf{T_b}$) (resp. DAWG($\mathsf{T_f}$)).

4.5 CDAWGs for Forward Tries

The *compact directed acyclic word graph* (*CDAWG*) of a forward trie $\mathsf{T_f}$, denoted CDAWG($\mathsf{T_f}$), is the edge-labeled DAG where the nodes correspond to the equivalence class of *Substr*($\mathsf{T_f}$) w.r.t. $mxml_{\mathsf{f}}(\cdot)$. In other words, CDAWG($\mathsf{T_f}$) can be obtained by merging isomorphic subtrees of STree($\mathsf{T_f}$) rooted at internal nodes and merging leaves that are equivalent under $mxml_{\mathsf{f}}(\cdot)$, or by contracting non-branching paths of DAWG($\mathsf{T_f}$).

See **Fig. 8** for an example.

The nodes of CDAWG($\mathsf{T_f}$) of Fig. 8 represent the equivalence classes w.r.t. the maximal substrings in $\mathsf{T_f}$ of Fig. 1, e.g., *aab* is maximal since it is both left- and right-maximal as described above and hence it is the longest string in the node that represents *aab*. This node also represents the suffix *ab* of *aab*, since $mxml_{\mathsf{f}}(ab) = aab$.

Theorem 4 (Ref. [26]). *For any forward trie* $\mathsf{T_f}$ *with n nodes over a constant-size alphabet,* $|\mathsf{CDAWG}(\mathsf{T_f})|_{\#Node} = O(n)$ *and* $|\mathsf{CDAWG}(\mathsf{T_f})|_{\#Edge} = O(n)$.

We emphasize that the above result by Inenaga et al. [26] states size bounds of CDAWG($\mathsf{T_f}$) only in the case where $\sigma = O(1)$. We will later show that this bound does not hold for the number of edges, in the case of a large alphabet.

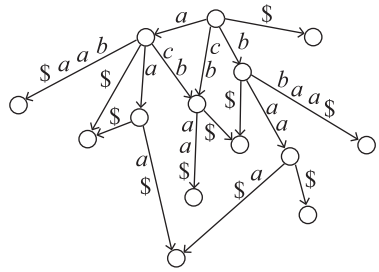

Fig. 9 CDAWG($\mathsf{T_b}$) for the backward trie $\mathsf{T_b}$ of Fig. 1.

4.6 CDAWGs for Backward Tries

The *compact directed acyclic word graph* (*CDAWG*) of a backward trie $\mathsf{T_b}$, denoted CDAWG($\mathsf{T_b}$), is the edge-labeled DAG where the nodes correspond to the equivalence class of *Substr*($\mathsf{T_b}$) w.r.t. $mxml_{\mathsf{b}}(\cdot)$. Similarly to its forward trie counterpart, CDAWG($\mathsf{T_b}$) can be obtained by merging isomorphic subtrees of STree($\mathsf{T_b}$) rooted at internal nodes and merging leaves that are equivalent under $mxml_{\mathsf{f}}(\cdot)$, or by contracting non-branching paths of DAWG($\mathsf{T_b}$).

See **Fig. 9** for an example.

The nodes of CDAWG($\mathsf{T_b}$) in Fig. 9 represent the equivalence classes w.r.t. the maximal substrings in $\mathsf{T_f}$ of Fig. 1, e.g., *acb* is maximal since it is both left- and right-maximal in $\mathsf{T_b}$ and hence it is the longest string in the node that represents *acb*. This node also represents the suffix *cb* of *acb*, since $mxml_{\mathsf{f}}(cb) = acb$. Notice that there is a one-to-one correspondence between the nodes of CDAWG($\mathsf{T_f}$) in Fig. 9 and the nodes of CDAWG($\mathsf{T_b}$) in Fig. 8. In other words, X is the longest string represented by a node in CDAWG($\mathsf{T_f}$) iff $Y = \overline{X}$ is the longest string represented by a node in CDAWG($\mathsf{T_b}$). For instance, *aab* is the longest string represented by a node of CDAWG($\mathsf{T_f}$) and *baa* is the longest string represented by a node of CDAWG($\mathsf{T_b}$), and so on. Hence the numbers of nodes in CDAWG($\mathsf{T_f}$) and CDAWG($\mathsf{T_b}$) are equal.

5. New Size Bounds on Indexing Forward/ Backward Tries

To make the analysis simpler, we assume each of the roots, the one of $\mathsf{T_f}$ and the corresponding one of $\mathsf{T_b}$, is connected to an auxiliary node $\perp$ with an edge labeled by a unique character \$ that does not appear elsewhere in $\mathsf{T_f}$ or in $\mathsf{T_b}$.

5.1 Size Bounds for DAWGs for Backward Tries

We begin with the size bounds for the DAWG for a backward trie.

Theorem 5. *For any backward trie* $\mathsf{T_b}$ *with n nodes,* $|\mathsf{DAWG}(\mathsf{T_b})|_{\#Node} = O(n^2)$ *and* $|\mathsf{DAWG}(\mathsf{T_b})|_{\#Edge} = O(n^2)$. *These upper bounds hold for any alphabet. For some backward trie* $\mathsf{T_b}$ *with n nodes,* $|\mathsf{DAWG}(\mathsf{T_b})|_{\#Node} = \Omega(n^2)$ *and* $|\mathsf{DAWG}(\mathsf{T_b})|_{\#Edge} = \Omega(n^2)$. *These lower bounds hold for a constant-size or larger alphabet.*

Proof. The bounds $|\mathsf{DAWG}(\mathsf{T_b})|_{\#Node} = O(n^2)$ and $|\mathsf{DAWG}(\mathsf{T_b})|_{\#Node} = \Omega(n^2)$ for the number of nodes immediately follow from Fact 2 and Theorem 1.

Since each internal node in DAWG($\mathsf{T_b}$) has at least one outgoing edge and since $|\mathsf{DAWG}(\mathsf{T_b})|_{\#Node} = \Omega(n^2)$, the lower bound $|\mathsf{DAWG}(\mathsf{T_b})|_{\#Edge} = \Omega(n^2)$ for the number of edges is immediate.

© 2021 Information Processing Society of Japan

To show the upper bound for the number of edges, we consider the *suffix trie* of $\mathsf{T_b}$. Since there are $O(n^2)$ pairs of nodes in $\mathsf{T_b}$, the number of substrings in $\mathsf{T_b}$ is clearly $O(n^2)$. Thus, the numbers of nodes and edges in the suffix trie of $\mathsf{T_b}$ are $O(n^2)$. Hence $|\mathsf{DAWG}(\mathsf{T_b})|_{\#Edge} = O(n^2)$. □

5.2 Size Bounds for DAWGs for Forward Tries

In this subsection, we consider the size bounds for the DAWG of a forward trie.

Theorem 6. *For any forward trie* $\mathsf{T_f}$ *with n nodes,* $|\mathsf{DAWG}(\mathsf{T_f})|_{\#Edge} = O(\sigma n)$. *For some forward trie* $\mathsf{T_f}$ *with n nodes,* $|\mathsf{DAWG}(\mathsf{T_f})|_{\#Edge} = \Omega(\sigma n)$ *which is* $\Omega(n^2)$ *for a large alphabet of size* $\sigma = \Theta(n)$.

Proof. Since each node of $\mathsf{DAWG}(\mathsf{T_f})$ can have at most σ outgoing edges, the upper bound $|\mathsf{DAWG}(\mathsf{T_f})|_{\#Edge} = O(\sigma n)$ follows from Theorem 3.

To obtain the lower bound $|\mathsf{DAWG}(\mathsf{T_f})|_{\#Edge} = \Omega(\sigma n)$, we consider $\mathsf{T_f}$ which has a broom-like shape such that there is a single path of length $n - \sigma - 1$ from the root to a node v which has outgoing edges with σ distinct characters $b_1, \ldots, b_\sigma$ (see **Fig. 10** for illustration.) Since the root of $\mathsf{T_f}$ is connected with the auxiliary node $\perp$ with an edge labeled \$, each root-to-leaf path in $\mathsf{T_f}$ represents $\$a^{n-\sigma+1}b_i$ for $1 \le i \le \sigma$. Now a^k for each $1 \le k \le n - \sigma - 2$ is left-maximal since it is immediately preceded by a and \$. Thus $\mathsf{DAWG}(\mathsf{T_f})$ has at least $n - \sigma - 2$ internal nodes, each representing a^k for $1 \le k \le n - \sigma - 2$. On the other hand, each $a^k \in \mathit{Substr}(\mathsf{T_f})$ is immediately followed by b_i with all $1 \le i \le \sigma$. Hence, $\mathsf{DAWG}(\mathsf{T_f})$ contains $\sigma(n - \sigma - 2) = \Omega(\sigma n)$ edges when $n - \sigma - 2 = \Omega(n)$. By choosing e.g., $\sigma \approx n/2$, we obtain $\mathsf{DAWG}(\mathsf{T_f})$ that contains $\Omega(n^2)$ edges. □

5.2.1 On Construction of DAWGs for Forward Tries

Mohri et al. (Proposition 4 of Ref. [36]) claimed that one can construct $\mathsf{DAWG}(\mathsf{T_f})$ in time proportional to its size. The following corollary is immediate from Theorem 6:

Corollary 3. *The DAWG construction algorithm of Ref. [36] applied to a forward trie with n nodes must take at least* $\Omega(n^2)$ *time in the worst case for an alphabet of size* $\sigma = \Theta(n)$.

Mohri et al.'s proof for Proposition 4 in Ref. [36] contains yet another issue: They claimed that the number of redirections of secondary edges during the construction of $\mathsf{DAWG}(\mathsf{T_f})$ can be bounded by the number n of nodes in $\mathsf{T_f}$, but this is not true. Breslauer [11] already pointed out this issue in his construction for $\mathsf{STree}(\mathsf{T_b})$ that is based on Weiner's algorithm (recall that Weiner's suffix tree construction algorithm inherently builds the DAWG for the reversed strings), and he overcame this difficulty by using σ nearest marked ancestor data structures for all σ characters, instead of explicitly maintaining soft W-links. This leads to $O(\sigma n)$-time and space construction for $\mathsf{STree}(\mathsf{T_b})$ that works in $O(n)$ time and space for constant-size alphabets. In Section 6 we will present how to build an $O(n)$-space *implicit* representation of $\mathsf{DAWG}(\mathsf{T_f})$ in $O(n)$ time and working space for larger alphabets of size $\sigma = O(n)$.

5.2.2 Size Bounds of Smallest Automaton for Forward Tries

Consider the smallest DFA that accepts the set $\mathit{Suffix}(\mathsf{T_f})$ of suffixes of forward trie $\mathsf{T_f}$. It is apparent that $\mathsf{DAWG}(\mathsf{T_f})$ is not the smallest such DFA, since it can have multiple sinks as in the example of Fig. 6. However, our lower bound instance for $\mathsf{DAWG}(\mathsf{T_f})$ also gives $\Omega(\sigma n)$ transitions in the smallest DFA. See Fig. 10. The smallest DFA that accepts the same language as the DAWG in Fig. 10 is the one obtained by merging all the sink nodes which correspond to the final states. However, this does not reduce the number of edges (transitions) at all. Hence, this smallest DFA still has $\Omega(\sigma n)$ transitions.

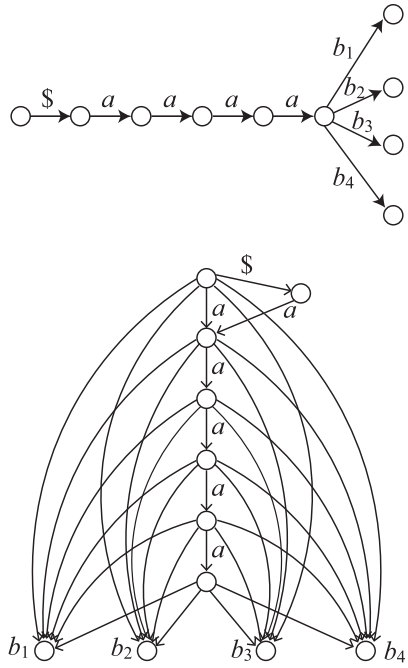

Fig. 10 Upper: The broom-like $\mathsf{T_f}$ for the lower bound of Theorem 6, where $n = 10$ and $\sigma = (n-2)/2 = 4$. Lower: $\mathsf{DAWG}(\mathsf{T_f})$ for this $\mathsf{T_f}$ has $\Omega(n^2)$ edges. The labels $b_1, \ldots, b_4$ of the in-coming edges to the sinks are omitted for better visualization.

Theorem 7. *For any forward trie* $\mathsf{T_f}$ *with n nodes, the number of transitions in the smallest DFA that accepts* $\mathit{Suffix}(\mathsf{T_f})$ *is* $O(\sigma n)$. *For some forward trie* $\mathsf{T_f}$ *with n nodes, the number of transitions in the smallest DFA that accepts* $\mathit{Suffix}(\mathsf{T_f})$ *is* $\Omega(\sigma n)$, *which is* $\Omega(n^2)$ *for a large alphabet of size* $\sigma = \Theta(n)$.

The same bounds hold for the smallest DFA that accepts the set $\mathit{Substr}(\mathsf{T_f})$ of substrings in forward trie $\mathsf{T_f}$.

5.3 Size Bounds for CDAWGs for Backward Tries

We begin with the size bounds of the CDAWG for a backward trie.

Theorem 8. *For any backward trie* $\mathsf{T_b}$ *with n nodes,* $|\mathsf{CDAWG}(\mathsf{T_b})|_{\#Node} \le 2n - 3$ *and* $|\mathsf{CDAWG}(\mathsf{T_b})|_{\#Edge} \le 2n - 4$. *These bounds are independent of the alphabet size.*

Proof. Since any maximal substring in $\mathit{Substr}(\mathsf{T_b})$ is right-maximal in $\mathit{Substr}(\mathsf{T_b})$, by Theorem 2 we have $|\mathsf{CDAWG}(\mathsf{T_b})|_{\#Node} \le |\mathsf{STree}(\mathsf{T_b})|_{\#Node} \le 2n - 3$ and $|\mathsf{CDAWG}(\mathsf{T_b})|_{\#Edge} \le |\mathsf{STree}(\mathsf{T_b})|_{\#Edge} \le 2n - 4$. □

The bounds in Theorem 8 are tight: Consider an alphabet $\{a_1, \ldots, a_{\lceil \log_2 n \rceil}, b_1, \ldots, b_{\lceil \log_2 n \rceil}, \$\}$ of size $2\lceil \log_2 n \rceil + 1$ and a binary backward trie $\mathsf{T_b}$ with n nodes where the binary edges at each depth $d \ge 2$ are labeled by the sub-alphabet $\{a_d, b_d\}$ of size 2 (see also **Fig. 11**). Because every suffix $S \in \mathit{Suffix}(\mathsf{T_b})$ is maximal in $\mathsf{T_b}$, $\mathsf{CDAWG}(\mathsf{T_b})$ for this $\mathsf{T_b}$ contains $n - 1$ sinks. Also, since for each suffix S in $\mathsf{T_b}$ there is a unique suffix $S' \ne S$ that shares

© 2021 Information Processing Society of Japan

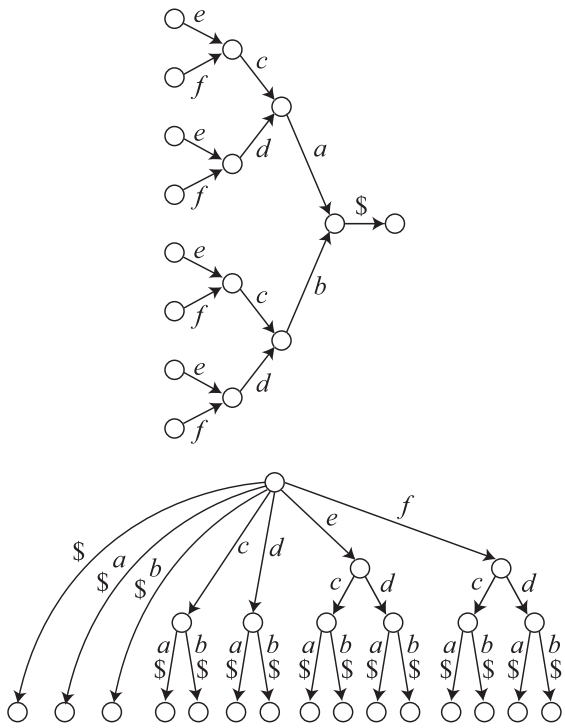

Fig. 11 Upper: A backward trie which gives the largest number of nodes and edges in the CDAWG for backward tries. Here, the sub-alphabets are $\{a, b\}$ for depth 2, $\{c, d\}$ for depth 3, and $\{e, f\}$ for depth 4. Lower: The CDAWG for the backward trie. Notice that no isomorphic subtrees are merged under our definition of equivalence classes. For instance, consider substrings c and d. Since $mxml_{\mathsf{b}}(c) = r\text{-}mxml_{\mathsf{b}}(l\text{-}mxml_{\mathsf{b}}(c)) = r\text{-}mxml_{\mathsf{b}}(c) = c \neq d = r\text{-}mxml_{\mathsf{b}}(l\text{-}mxml_{\mathsf{b}}(d)) = r\text{-}mxml_{\mathsf{b}}(d) = mxml_{\mathsf{b}}(d)$, the isomorphic subtrees rooted at c and d are not merged. By the same reasoning, isomorphic subtrees (including sink nodes) are not merged in this CDAWG.

the longest common prefix with S, $\mathsf{CDAWG}(\mathsf{T_b})$ for this $\mathsf{T_b}$ contains $n-2$ internal nodes (including the source). This also means $\mathsf{CDAWG}(\mathsf{T_b})$ is identical to $\mathsf{STree}(\mathsf{T_b})$ for this backward trie $\mathsf{T_b}$.

5.4 Size Bounds for CDAWGs for Forward Tries

Next, we turn our attention to the size bounds of the CDAWG for a forward trie.

Theorem 9. *For any forward trie* $\mathsf{T_f}$ *with* n *nodes,* $|\mathsf{CDAWG}(\mathsf{T_f})|_{\#Node} \leq 2n-3$ *and* $|\mathsf{CDAWG}(\mathsf{T_f})|_{\#Edge} = O(\sigma n)$. *For some forward trie* $\mathsf{T_f}$ *with* n *nodes,* $|\mathsf{CDAWG}(\mathsf{T_f})|_{\#Edge} = \Omega(\sigma n)$ *which is* $\Omega(n^2)$ *for a large alphabet of size* $\sigma = \Theta(n)$.

Proof. It immediately follows from Fact 1-(a), Fact 2, and Theorem 8 that $|\mathsf{CDAWG}(\mathsf{T_f})|_{\#Node} = |\mathsf{CDAWG}(\mathsf{T_b})|_{\#Node} \leq 2n-3$. Since a node in $\mathsf{CDAWG}(\mathsf{T_f})$ can have at most σ out-going edges, the upper bound $|\mathsf{CDAWG}(\mathsf{T_f})|_{\#Edge} = O(\sigma n)$ of the number of edges trivially holds. To obtain the lower bound, we consider the same broom-like forward trie $\mathsf{T_f}$ as in Theorem 6. In this $\mathsf{T_f}$, a^k for each $1 \leq k \leq n-\sigma-2$ is maximal and thus $\mathsf{CDAWG}(\mathsf{T_f})$ has at least $n-\sigma-2$ internal nodes each representing a^k for $1 \leq k \leq n-\sigma-2$. By the same argument as Theorem 6, $\mathsf{CDAWG}(\mathsf{T_f})$ for this $\mathsf{T_f}$ contains at least $\sigma(n-\sigma-2) = \Omega(\sigma n)$ edges, which amounts to $\Omega(n^2)$ for a large alphabet of size e.g., $\sigma \approx n/2$. □

The upper bound of Theorem 9 generalizes the bound of Theorem 4 for constant-size alphabets. Remark that $\mathsf{CDAWG}(\mathsf{T_f})$ for the broom-like $\mathsf{T_f}$ of Fig. 10 is almost identical to $\mathsf{DAWG}(\mathsf{T_f})$, except for the unary path $\$a$ that is compacted in $\mathsf{CDAWG}(\mathsf{T_f})$.

6. Constructing $O(n)$-size Representation of $\mathsf{DAWG}(\mathsf{T_f})$ in $O(n)$ time

We have seen that $\mathsf{DAWG}(\mathsf{T_f})$ for any forward trie $\mathsf{T_f}$ with n nodes contains only $O(n)$ nodes, but can have $\Omega(\sigma n)$ edges for some $\mathsf{T_f}$ over an alphabet of size σ ranging from $\Theta(1)$ to $\Theta(n)$. Thus some $\mathsf{DAWG}(\mathsf{T_f})$ can have $\Theta(n^2)$ edges for $\sigma = \Theta(n)$ (Theorem 3 and Theorem 6). Hence, in general it is impossible to build an *explicit* representation of $\mathsf{DAWG}(\mathsf{T_f})$ within linear $O(n)$-space. By an explicit representation we mean an implementation of $\mathsf{DAWG}(\mathsf{T_f})$ where each edge is represented by a pointer between two nodes.

Still, we show that there exists an $O(n)$-space *implicit* representation of $\mathsf{DAWG}(\mathsf{T_f})$ for any alphabet of size σ ranging from $\Theta(1)$ to $\Theta(n)$, that allows us $O(\log \sigma)$-time access to each edge of $\mathsf{DAWG}(\mathsf{T_f})$. This is trivial in case $\sigma = O(1)$, and hence in what follows we consider an alphabet of size σ such that σ ranges from $\omega(1)$ to $\Theta(n)$. Also, we suppose that our alphabet is an integer alphabet $\Sigma = [1..\sigma]$ of size σ. Then, we show that such an implicit representation of $\mathsf{DAWG}(\mathsf{T_f})$ can be built in $O(n)$ time and working space.

Based on the property stated in Section 4, constructing $\mathsf{DAWG}(\mathsf{T_f})$ reduces to maintaining hard and soft W-links over $\mathsf{STree}(\mathsf{T_b})$. Our data structure explicitly stores all $O(n)$ hard W-links, while it only stores carefully selected $O(n)$ soft W-links. The other soft W-links can be simulated by these explicitly stored W-links, in $O(\log \sigma)$ time each.

Our algorithm is built upon the following facts which are adapted from Refs. [16], [17]:

Fact 3. *Let* a *be any character from* Σ.

(a) *If there is a (hard or soft) W-link* $\mathcal{W}_a(V)$ *for a node* V *in* $\mathsf{STree}(\mathsf{T_b})$, *then there is a (hard or soft) W-link* $\mathcal{W}_a(U)$ *for any ancestor* U *of* V *in* $\mathsf{STree}(\mathsf{T_b})$.

(b) *If two nodes* U *and* V *have hard W-links* $\mathcal{W}_a(U)$ *and* $\mathcal{W}_a(V)$, *then the lowest common ancestor (LCA)* Z *of* U *and* V *also has a hard W-link* $\mathcal{W}_a(Z)$.

In the following statements (c), (d), *and* (e), *let* V *be any node of* $\mathsf{STree}(\mathsf{T_b})$ *such that* V *has a soft W-link* $\mathcal{W}_a(V)$ *for* $a \in \Sigma$.

(c) *There is a descendant* U *of* V *s.t.* $U \neq V$ *and* U *has a hard W-link* $\mathcal{W}_a(V)$.

(d) *The highest descendant of* V *that has a hard W-link for character* a *is unique. This fact follows from* (b).

(e) *Let* U *be the unique highest descendant of* V *that has a hard W-link* $\mathcal{W}_a(U)$. *For every node* Z *in the path from* V *to* U, $\mathcal{W}_a(Z) = \mathcal{W}_a(U)$, *i.e., the W-links of all nodes in this path for character* a *point to the same node in* $\mathsf{STree}(\mathsf{T_b})$.

6.1 Compact Representation of Weiner Links

We construct a micro-macro tree decomposition [2] of $\mathsf{STree}(\mathsf{T_b})$ in a similar manner to Ref. [19], such that the nodes of $\mathsf{STree}(\mathsf{T_b})$ are partitioned into $O(n/\sigma)$ connected components (called *micro-trees*), each of which contains $O(\sigma)$ nodes. (see **Fig. 12**). Such a decomposition always exists and can be computed in $O(n)$ time. The *macro tree* is the induced tree from the

© 2021 Information Processing Society of Japan

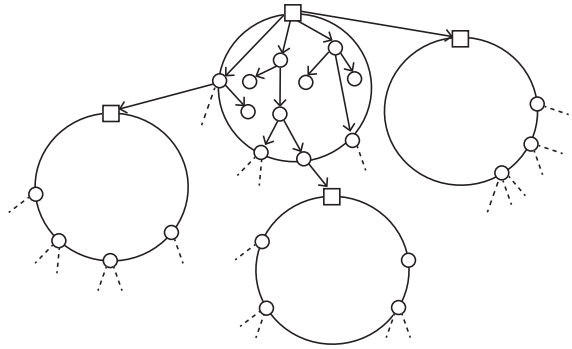

Fig. 12 Illustration for our micro-macro tree decomposition of $\mathsf{STree}(\mathsf{T_b})$. The large circles represent micro tree of size $O(\sigma)$ each, and the square nodes are the roots of the micro trees. The macro tree is the induced tree from the square nodes.

roots of the micro trees, and thus the macro tree contains $O(n/\sigma)$ nodes.

In every node V of the macro tree, we explicitly store all soft and hard W-links from V. Since there can be at most σ W-links from V, this requires $O(n)$ total space for all nodes in the macro tree. Let mt denote any micro tree. We compute the ranks of all nodes in a pre-order traversal in mt. Let $a \in \Sigma$ be any character such that there is a node V in mt that has a hard W-link $\mathcal{W}_a(V)$. Let $\mathsf{P}_a^{\mathsf{mt}}$ denote an array that stores a sorted list of pre-order ranks of nodes V in mt that have hard W-links for character a. Hence the size of $\mathsf{P}_a^{\mathsf{mt}}$ is equal to the number of nodes in mt that have hard W-links for character a. For all such characters a, we store $\mathsf{P}_a^{\mathsf{mt}}$ in mt. The total size of these arrays for all the micro trees is clearly $O(n)$.

Let $a \in \Sigma$ be any character, and V any node in $\mathsf{STree}(\mathsf{T_b})$ which does not have a hard W-link for a. We wish to know if V has a soft W-link for a, and if so, we want to retrieve the target node of this link. Let mt denote the micro-tree to which V belongs. Consider the case where V is not the root R of mt, since otherwise $\mathcal{W}_a(V)$ is explicitly stored. If $\mathcal{W}_a(R)$ is nil, then by Fact 3-(a) no nodes in the micro tree have W-links for character a. Otherwise (if $\mathcal{W}_a(R)$ exists), then we can find $\mathcal{W}_a(W)$ as follows:

(A) If the predecessor P of V exists in $\mathsf{P}_a^{\mathsf{mt}}$ and P is an ancestor of V, then we follow the hard W-link $\mathcal{W}_a(P)$ from P. Let $Q = \mathcal{W}_a(P)$, and let c be the first character in the path from P to V.
  (i) If Q has an out-going edge whose label begins with c, the child of Q below this edge is the destination of the soft W-link $\mathcal{W}_a(V)$ from V for a (see **Fig. 13**).
  (ii) Otherwise, then there is no W-link from V for a (see **Fig. 14**).

(B) Otherwise, $\mathcal{W}_a(R)$ from the root R of mt is a soft W-link, which is explicitly stored. We follow it and let $U = \mathcal{W}_a(R)$.
  (i) If $Z = slink(U)$ is a descendant of V, then U is the destination of the soft W-link $\mathcal{W}_a(V)$ from V for a (see **Fig. 15**).
  (ii) Otherwise, then there is no W-link from V for a (see **Fig. 16**).

In Figs. 13, 14, 15 and 16, the large circles show micro tree mt and the square nodes are the roots of mt. We query the soft W-link of V (gray nodes) for character a. The black nodes are the

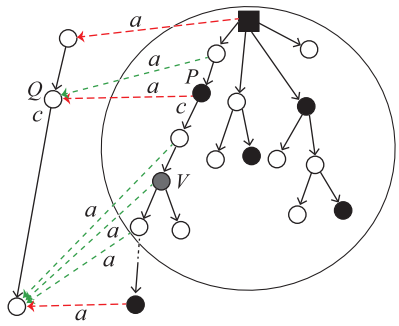

Fig. 13 Case (A)-(i) of our soft W-link query algorithm.

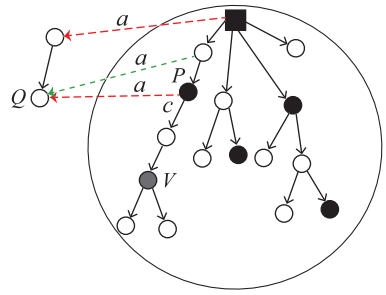

Fig. 14 Case (A)-(ii) of our soft W-link query algorithm.

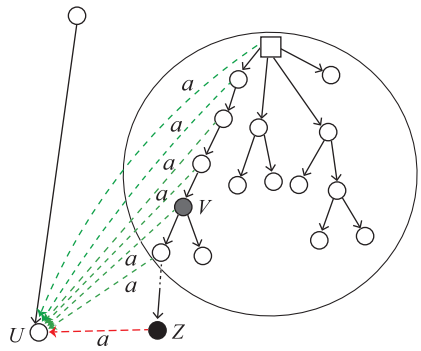

Fig. 15 Case (B)-(i) of our soft W-link query algorithm.

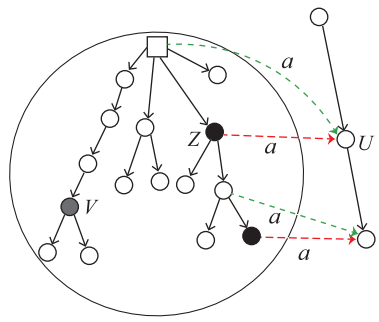

Fig. 16 Case (B)-(ii) of our soft W-link query algorithm.

nodes that have hard W-link for character a, and the red broken arrows represent hard W-links for a of our interest. The green broken arrows represent soft W-links for a of our interest.

Figures 13 and 14 respectively show the sub-cases of Case (A)-(i) and Case (A)-(ii) where the root of the micro tree mt has a hard W-link for a, but our algorithm works also in the sub-cases where the root has a soft W-link for a.

We remark that in Case (B) there can be at most one path in the micro tree mt containing nodes which have hard W-links for character a, as illustrated in Fig. 15 and in Fig. 16. This is because, if

© 2021 Information Processing Society of Japan

there are two distinct such paths in mt, then by Fact 3-(b) the root of mt must have a hard W-link for character a, which contradicts our assumption for Case (B).

The correctness of this algorithm follows from Fact 3-(e). Since each micro-tree contains $O(\sigma)$ nodes, the size of $\mathsf{P}_a^{\mathsf{mt}}$ is $O(\sigma)$ and thus the predecessor P of V in $\mathsf{P}_a^{\mathsf{mt}}$ can be found in $O(\log \sigma)$ time by binary search. We can check if one node is an ancestor of the other node (or vice versa) in $O(1)$ time, after standard $O(n)$-time preprocessing over the whole suffix tree. Hence, this algorithm simulates soft W-link $\mathcal{W}_a(V)$ in $O(\log \sigma)$ time. Since soft W-links correspond to secondary edges of DAWGs, we obtain the following:

Lemma 1. *Our data structure simulates secondary-edge traversal on* $\mathsf{DAWG}(\mathsf{T_f})$ *for a forward trie with n nodes in* $O(\log \sigma)$ *time each, and occupies* $O(n)$ *space.*

Note that our data structure for secondary-edge traversal on $\mathsf{DAWG}(\mathsf{T_f})$ is as time and space efficient as a standard representation of labeled graphs where branches are implemented by balanced binary search trees or arrays.

6.2 Construction Algorithm

What remains is how to preprocess the input trie to compute the above data structure.

Lemma 2. *Given a backward trie* $\mathsf{T_b}$ *with n nodes, we can compute* $\mathsf{STree}(\mathsf{T_b})$ *with all hard W-links in* $O(n)$ *time and space.*

Proof. We build $\mathsf{STree}(\mathsf{T_b})$ without suffix links in $O(n)$ time and space [40]. We then add the suffix links to $\mathsf{STree}(\mathsf{T_b})$ as follows. To each node v of $\mathsf{T_b}$ we allocate its rank in a breadth-first traversal so that for any reversed edge $str_{\mathsf{b}}(v, a, u)$, v has a smaller rank than u. We will identify each node with its rank.

Let SA be the suffix array for $\mathsf{T_b}$ that corresponds to the leaves of $\mathsf{STree}(\mathsf{T_b})$, where $\mathsf{SA}[i] = j$ iff the suffix in $\mathsf{T_b}$ beginning at node j is the ith lexicographically smallest suffix. We compute SA and its inverse array in $O(n)$ time via $\mathsf{STree}(\mathsf{T_b})$, or directly from $\mathsf{T_b}$ using the algorithm proposed by Ferragina et al. [15]. The suffix links of the leaves of $\mathsf{STree}(\mathsf{T_b})$ can easily be computed in $O(n)$ time and space, by using the inverse array of SA. Unlike the case of a single string where the suffix links of the leaves form a single chain, the suffix links of the leaves of $\mathsf{STree}(\mathsf{T_b})$ form a tree, but this does not incur any problem in our algorithm. To compute the suffix links of the internal nodes of $\mathsf{STree}(\mathsf{T_b})$, we use the following standard technique that was originally designed for the suffix tree of a single string (see e.g., Ref. [33]): For any internal node V in $\mathsf{STree}(\mathsf{T_b})$, let ℓ_V and r_V denote the smallest and largest indices in SA such that $\mathsf{SA}[\ell_V..r_V]$ is the maximal interval corresponding to the suffixes which have string V as a prefix. It then holds that $slink(V) = U$, where U is the lowest common ancestor (LCA) of $slink(\ell_V)$ and $slink(r_V)$. For all nodes V in $\mathsf{T_b}$, the LCA of $slink(\ell_V)$ and $slink(r_V)$ can be computed in $O(n)$ time and space. After computing the suffix links, we can easily compute the character labels of the corresponding hard W-links in $O(n)$ time. □

Lemma 3. *We can compute, in* $O(n)$ *time and space, all W-links of the macro tree nodes and the arrays* $\mathsf{P}_a^{\mathsf{mt}}$ *for all the micro trees* mt *and characters* $a \in \Sigma$.

Proof. We perform a pre-order traversal on each micro tree mt. At each node V visited during the traversal, we append the pre-order rank of V to array $\mathsf{P}_a^{\mathsf{mt}}$ iff V has a hard W-link $\mathcal{W}_a(V)$ for character a. Since the size of mt is $O(\sigma)$ and since we have assumed an integer alphabet $[1..\sigma]$, we can compute $\mathsf{P}_a^{\mathsf{mt}}$ for all characters a in $O(\sigma)$ time. It takes $O(\frac{n}{\sigma} \cdot \sigma) = O(n)$ time for all micro trees.

The preprocessing for the macro tree consists of two steps. Firstly, we need to compute soft W-links from the macro tree nodes (recall that we have already computed hard W-links from the macro tree nodes by Lemma 2). For this purpose, in the above preprocessing for micro trees, we additionally pre-compute the successor of the root R of each micro tree mt in each non-empty array $\mathsf{P}_a^{\mathsf{mt}}$. By Fact 3-(d), this successor corresponds to the unique descendant of R that has a hard W-link for character a. As above, this preprocessing also takes $O(\sigma)$ time for each micro tree, resulting in $O(n)$ total time. Secondly, we perform a bottom-up traversal on the macro tree. Our basic strategy is to "propagate" the soft W-links in a bottom up fashion from lower nodes to upper nodes in the macro tree (recall that these macro tree nodes are the roots of micro trees). In so doing, we first compute the soft W-links of the macro tree leaves. By Fact 3-(c) and -(e), this can be done in $O(\sigma)$ time for each leaf using the successors computed above. Then we propagate the soft W-links to the macro tree internal nodes. The existence of soft W-links of internal nodes computed in this way is justified by Fact 3-(a), however, the destinations of some soft W-links of some macro tree internal nodes may not be correct. This can happen when the corresponding micro trees contain hard W-links (due to Fact 3-(e)). These destinations can be modified by using the successors of the roots computed in the first step, again due to Fact 3-(e). Both of our propagation and modification steps take $O(\sigma)$ time for each macro tree node of size $O(\sigma)$, and hence, it takes a total of $O(n)$ time. □

We have shown the following:

Theorem 10. *Given a forward trie* $\mathsf{T_f}$ *of size n over an integer alphabet* $\Sigma = [1..\sigma]$ *with* $\sigma = O(n)$*, we can construct in* $O(n)$ *time and working space an* $O(n)$*-space representation of* $\mathsf{DAWG}(\mathsf{T_f})$ *that simulates edge traversals in* $O(\log \sigma)$ *time each.*

6.3 Bidirectional Searches on Tries

A consequence of Theorem 10 is a space-efficient data structure that allows for *bidirectional searches* of patterns on a given trie.

Theorem 11. *Our* $O(n)$*-size implicit representation of Weiner links of* $\mathsf{STree}(\mathsf{T_b})$ *allows bidirectional searches of a given pattern P of length m in* $O(m \log \sigma + occ)$ *time, where occ is the number of occurrences of P in the input trie.*

Proof. We initially set $P \leftarrow \varepsilon$ and start at the root of $\mathsf{STree}(\mathsf{T_b})$.

Suppose we have added k $(0 \le k < m)$ characters to P in a bidirectional manner, and that we know the locus for the current P over $\mathsf{STree}(\mathsf{T_b})$.

If a new character a is appended to P, then we perform a standard edge traversal on $\mathsf{STree}(\mathsf{T_b})$, in $O(\log \sigma)$ time. Then we set $P \leftarrow Pa$ and continue to the next character that will be either prepended or appended to P.

If a new character b is prepended to P, then there are two sub-

© 2021 Information Processing Society of Japan

cases. If the locus for P is an explicit node of $\mathsf{STree}(\mathsf{T_b})$, then we perform the aforementioned algorithm and find the corresponding Weiner link for b, in $O(\log \sigma)$ time. Otherwise (if the locus for P is an implicit node of $\mathsf{STree}(\mathsf{T_b})$), then let V be the nearest explicit ancestor of the locus for P. We perform the aforementioned algorithm on V and find the destination U of the Weiner link $\mathcal{W}_b(V)$. Let c be the first character of the edge from V to the locus for P. We find the out-going edge of U whose edge label begins with c in $O(\log \sigma)$ time. Then, the locus for bP is at the $|P| - |V|$th character of this edge. Since each edge label in $\mathsf{STree}(\mathsf{T_b})$ is implemented by a pointer to a corresponding path in $\mathsf{T_b}$, we can move to the locus for bP in $O(1)$ time (using the level ancestor query on the trie). Then we set $P \leftarrow bP$ and continue to the next character that will be either prepended or appended to P.

After adding all m characters, we know the locus for the final pattern P in $\mathsf{STree}(\mathsf{T_b})$. Then we use a standard technique of traversing the subtree rooted at the locus in $\mathsf{STree}(\mathsf{T_b})$, and takes all its leaves. Each of these leaves corresponds to the *occ* occurrences of P in the trie. This takes $O(occ)$ time. □

To the best of our knowledge, this is the *first* indexing structure on tries that permits efficient bidirectional pattern searches.

7. Concluding Remarks

This paper dealt with the labeled tree indexing problem. We presented tight upper and lower bounds on the sizes of the indexing structures (i.e. suffix trees, DAWGs, CDAWGs, suffix arrays, affix trees, and affix arrays) built on forward and backward tries.

In addition, we proposed a non-trivial $O(n)$-space implicit representation of the DAWG for a forward trie for any alphabet of size $\sigma = O(n)$, of which a naïve explicit representation must use $\Omega(n^2)$ space in the worst case. We also showed how to construct such a representation in $O(n)$ time for an integer alphabet of size $\sigma = O(n)$. The proposed data structure can also solve the bidirectional pattern search problem over a trie.

We believe that all these results have revealed a much clearer view of the labeled tree indexing problem. In addition, some of the quadratic lower bounds were generalized to the labeled DAG indexing problem, which provided negative answers to the open questions from the literature [14], [36].

There remain intriguing open questions:

- Direct construction of $\mathsf{CDAWG}(\mathsf{T_b})$: It is not difficult to construct $\mathsf{CDAWG}(\mathsf{T_b})$ from $\mathsf{STree}(\mathsf{T_b})$, but whether or not there is an efficient algorithm that builds $\mathsf{CDAWG}(\mathsf{T_b})$ *directly* from a given backward trie $\mathsf{T_b}$ remains open.
- Implicit representation of $\mathsf{CDAWG}(\mathsf{T_f})$: Can we generalize the ideas from Section 6 to $\mathsf{CDAWG}(\mathsf{T_f})$, so that it can be stored in $O(n)$ space for a large alphabet of size $\sigma = O(n)$?
- Bidirectional BWT for tries: Our bidirectional indexing structure for a trie is not succinct or compressed. Is there a version of BWT for tries that is capable of bidirectional searches with succinct or compressed space?
- Tight size bound on indexing structures for labeled DAGs: In Section 1.3 we discussed that our quadratic lower bounds also hold for the DAWG or its SDD version built on a labeled DAG. We conjecture that a stronger lower bound holds for some labeled DAGs.

Acknowledgments The author thanks Dany Breslauer (deceased) for fruitful discussions at the initial stage of this work. The author is also grateful to an anonymous referee for pointing out an error in the previous version of this paper and for suggesting use of micro-macro tree decomposition. This research is supported by KAKENHI grant number JP17H01697 and by JST PRESTO Grant Number JPMJPR1922.

References

[1] Aho, A.V. and Corasick, M.: Efficient string matching: An Aid to Bibliographic Search, *Comm. ACM*, Vol.18, No.6, pp.333–340 (1975).
[2] Alstrup, S., Secher, J.P. and Spork, M.: Optimal On-Line Decremental Connectivity in Trees, *Inf. Process. Lett.*, Vol.64, No.4, pp.161–164 (1997).
[3] Apostolico, A. and Lonardi, S.: Off-line compression by greedy textual substitution, *Proc. IEEE*, Vol.88, No.11, pp.1733–1744 (2000).
[4] Belazzougui, D. and Cunial, F.: Fast Label Extraction in the CDAWG, *Proc. SPIRE 2017*, pp.161–175 (2017).
[5] Belazzougui, D. and Cunial, F.: Fully-Functional Bidirectional Burrows-Wheeler Indexes and Infinite-Order De Bruijn Graphs, *Proc. CPM 2019*, pp.10:1–10:15 (2019).
[6] Belazzougui, D., Cunial, F., Kärkkäinen, J. and Mäkinen, V.: Versatile Succinct Representations of the Bidirectional Burrows-Wheeler Transform, *Proc. ESA 2013*, pp.133–144 (2013).
[7] Bender, M.A. and Farach-Colton, M.: The Level Ancestor Problem simplified, *Theor. Comput. Sci.*, Vol.321, No.1, pp.5–12 (2004).
[8] Bieganski, P., Riedl, J., Carlis, J.V. and Retzel, E.F.: Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation, *Proc. HICSS 1994*, pp.35–44 (1994).
[9] Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T. and Seiferas, J.: The Smallest Automaton Recognizing the Subwords of a Text, *Theor. Comput. Sci.*, Vol.40, pp.31–55 (1985).
[10] Blumer, A., Blumer, J., Haussler, D., Mcconnell, R. and Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis, *J. ACM*, Vol.34, No.3, pp.578–595 (1987).
[11] Breslauer, D.: The Suffix Tree of a Tree and Minimizing Sequential Transducers, *Theoretical Computer Science*, Vol.191, No.1–2, pp.131–144 (1998).
[12] Breslauer, D. and Italiano, G.F.: Near real-time suffix tree construction via the fringe marked ancestor problem, *J. Discrete Algorithms*, Vol.18, pp.32–48 (2013).
[13] Crochemore, M. and Rytter, W.: *Text Algorithms*, Oxford University Press (1994).
[14] Denzumi, S., Tsuda, K., Arimura, H. and Minato, S.: Compact Complete Inverted Files for Texts and Directed Acyclic Graphs Based on Sequence Binary Decision Diagrams, *Proc. PSC 2013*, pp.157–167 (2013).
[15] Ferragina, P., Luccio, F., Manzini, G. and Muthukrishnan, S.: Compressing and indexing labeled trees, with applications, *J. ACM*, Vol.57, No.1, pp.4:1–4:33 (2009).
[16] Fischer, J. and Gawrychowski, P.: Alphabet-Dependent String Searching with Wexponential Search Trees (2013).
[17] Fischer, J. and Gawrychowski, P.: Alphabet-Dependent String Searching with Wexponential Search Trees, *CPM 2015*, pp.160–171 (2015).
[18] Fujishige, Y., Tsujimaru, Y., Inenaga, S., Bannai, H. and Takeda, M.: Computing DAWGs and Minimal Absent Words in Linear Time for Integer Alphabets, *Proc. MFCS 2016*, pp.38:1–38:14 (2016).
[19] Gawrychowski, P.: Simple and efficient LZW-compressed multiple pattern matching, *J. Discrete Algorithms*, Vol.25, pp.34–41 (2014).
[20] Gog, S., Karhu, K., Kärkkäinen, J., Mäkinen, V. and Välimäki, N.: Multi-pattern matching with bidirectional indexes, *J. Discrete Algorithms*, Vol.24, pp.26–39 (2014).
[21] Gusfield, D.: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press (1997).
[22] Gusfield, D. and Stoye, J.: Linear time algorithms for finding and representing all the tandem repeats in a string, *J. Comput. Syst. Sci.*, Vol.69, No.4, pp.525–546 (2004).
[23] Hammer, J. and Schneider, M.: Data Structures for Databases, *Handbook of Data Structures and Applications*, Chapman and Hall/CRC (2004).
[24] Hendrian, D., Inenaga, S., Yoshinaka, R. and Shinohara, A.: Efficient dynamic dictionary matching with DAWGs and AC-automata, *Theor. Comput. Sci.*, Vol.792, pp.161–172 (2019).
[25] Inenaga, S.: Suffix Trees, DAWGs and CDAWGs for Forward and Backward Tries., *Accepted for LATIN 2020* (2020).
[26] Inenaga, S., Hoshino, H., Shinohara, A., Takeda, M. and Arikawa, S.:

© 2021 Information Processing Society of Japan

Construction of the CDAWG for a Trie, *Proc. PSC 2001*, pp.37–48 (2001).
[27] Kimura, D. and Kashima, H.: Fast Computation of Subpath Kernel for Trees, *Proc. ICML 2012* (2012).
[28] Kosaraju, S.R.: Efficient Tree Pattern Matching (Preliminary Version), *Proc. FOCS 1989*, pp.178–183 (1989).
[29] Kucherov, G. and Rusinowitch, M.: Matching a Set of Strings with Variable Length don't Cares, *Theor. Comput. Sci.*, Vol.178, No.1-2, pp.129–154 (1997).
[30] Kucherov, G., Salikhov, K. and Tsur, D.: Approximate string matching using a bidirectional index, *Theor. Comput. Sci.*, Vol.638, pp.145–158 (2016).
[31] Lam, T.W., Li, R., Tam, A., Wong, S.C.K., Wu, E. and Yiu, S.: High Throughput Short Read Alignment via Bi-directional BWT, *Proc. BIBM 2009*, pp.31–36 (2009).
[32] Loekito, E., Bailey, J. and Pei, J.: A binary decision diagram based approach for mining frequent subsequences, *Knowl. Inf. Syst.*, Vol.24, No.2, pp.235–268 (2010).
[33] Mäkinen, V., Belazzougui, D., Cunial, F. and Tomescu, A.I.: *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*, Cambridge University Press (2015).
[34] Mauri, G. and Pavesi, G.: Algorithms for pattern matching and discovery in RNA secondary structure, *Theor. Comput. Sci.*, Vol.335, No.1, pp.29–51 (2005).
[35] Minato, S.: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems, *Proc. DAC 1993*, pp.272–277 (1993).
[36] Mohri, M., Moreno, P.J. and Weinstein, E.: General suffix automaton construction algorithm and space bounds, *Theor. Comput. Sci.*, Vol.410, No.37, pp.3553–3562 (2009).
[37] Muthukrishnan, S.: Efficient algorithms for document retrieval problems, *Proc. SODA 2002*, pp.657–666 (2002).
[38] Nakashima, Y., I, T., Inenaga, S., Bannai, H. and Takeda, M.: The Position Heap of a Trie, *Proc. SPIRE 2012*, pp.360–371 (2012).
[39] Schnattinger, T., Ohlebusch, E. and Gog, S.: Bidirectional search in a string with wavelet trees and bidirectional matching statistics, *Inf. Comput.*, Vol.213, pp.13–22 (2012).
[40] Shibuya, T.: Constructing the Suffix Tree of a Tree with a Large Alphabet, *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, Vol.E86-A, No.5, pp.1061–1066 (2003).
[41] Stoye, J.: Affix trees, Technical Report Report 2000-04, Universität Bielefeld (2000).
[42] Strothmann, D.: The affix array data structure and its applications to RNA secondary structure analysis, *Theor. Comput. Sci.*, Vol.389, No.1-2, pp.278–294 (2007).
[43] Weiner, P.: Linear pattern-matching algorithms, *Proc. 14th IEEE Ann. Symp. Switching and Automata Theory*, pp.1–11 (1973).
[44] Welch, T.A.: A Technique for High Performance Data Compression, *IEEE Computer*, Vol.17, pp.8–19 (1984).
[45] Witten, I.H., Moffat, A. and Bell, T.C.: *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*, Morgan Kaufmann (1999).
[46] Yamamoto, J., I, T., Bannai, H., Inenaga, S. and Takeda, M.: Faster Compact On-Line Lempel-Ziv Factorization, *Proc. STACS 2014*, pp.675–686 (2014).
[47] Ziv, J. and Lempel, A.: Compression of Individual Sequences via Variable-length Coding, *IEEE Trans. Information Theory*, Vol.24, No.5, pp.530–536 (1978).

Shunsuke Inenaga Shunsuke Inenaga received a master's degree in science at Kyushu University in 2002, and received a PhD in science at Kyushu University in 2003. From 2003, he worked as a posdoc researcher for Japan Science Technology Agency, University of Helsinki, Kyoto University, and Japan Society for the Promotion of Science. He became a tenure track research fellow at Kyushu University in 2006, and then became an associate professor at Kyushu University in 2011. Since 2019, he has also been a researcher for PRESTO, Japan Science and Technology Agency. His main research interests are algorithms and data structures for string processing, text compression, and applied word combinatorics.

© 2021 Information Processing Society of Japan