Suffix Trees, DAWGs and CDAWGs for Forward and Backward Tries

Shunsuke Inenaga[1,2](✉)

[1] Department of Informatics, Kyushu University, Fukuoka, Japan
inenaga@inf.kyushu-u.ac.jp
[2] PRESTO, Japan Science and Technology Agency, Kawaguchi, Japan

Abstract. The suffix tree, DAWG, and CDAWG are fundamental indexing structures of a string, with a number of applications in bioinformatics, information retrieval, data mining, etc. An edge-labeled rooted tree (trie) is a natural generalization of a string, which can also be seen as a compact representation of a set of strings. Kosaraju [FOCS 1989] 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 a usual 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, and CDAWGs for forward and backward tries. 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 $O(n)$.

1 Introduction

Text indexing is a fundamental problem in theoretical computer science that dates back to 1970's when suffix trees were invented [26]. Here the task is to preprocess a given text string S so that subsequent patten matching queries on S can be answered efficiently. Suffix trees have numerous other applications e.g. sequence comparisons [26], lossless data compression [2], data mining [23], and bioinformatics [15,21].

A trie is a rooted tree where each edge is labeled with a single character. A *backward* trie is an edge-reversed trie. Kosaraju [19] was the first to consider the trie indexing problem, and he proposed the suffix tree of a backward trie that takes $O(n)$ space, where n is the number of nodes in the backward trie. Kosaraju also claimed an $O(n \log n)$-time construction. Breslauer [7] showed how to build the suffix tree of a backward trie in $O(\sigma n)$ time and space, where σ is the

Y. Kohayakawa and F. K. Miyazawa (Eds.): LATIN 2020, LNCS 12118, pp. 194–206, 2020.
https://doi.org/10.1007/978-3-030-61792-9_16

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 Sect. 5 of 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 Sect. 5).

	Forward trie		Backward trie	
Indexing structure	# of nodes	# of edges	# of nodes	# of edges
Suffix tree	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
DAWG	$O(n)$	$O(\sigma n)$	$O(n^2)$	$O(n^2)$
CDAWG	$O(n)$	$O(\sigma n)$	$O(n)$	$O(n)$

alphabet size. Shibuya [25] presented an $O(n)$-time and space construction for the suffix tree of a backward trie over an integer alphabet of size $O(n)$. This line of research has been followed by the invention of XBWTs [11], suffix arrays [11], enhanced suffix arrays [18], and position heaps [24] for backward tries.

This paper considers the suffix trees, the *directed acyclic word graphs* (*DAWGs*) [5,9], and the *compact DAWGs* (*CDAWGs*) [6] built on a backward trie or on a forward (ordinary) trie. While all these indexing structures support linear-time pattern matching queries on tries, their sizes can significantly differ. We present *tight* lower and upper bounds on the sizes of all these indexing structures, as summarized in Table 1. 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), since this reveals that Mohri et al.'s algorithm [22] that constructs the DAWG for a forward trie with n nodes must take at least $\Omega(n^2)$ time and space in the worst case. We show that, somewhat surprisingly, there exists an *implicit compact representation* 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 also 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 $O(n)$.

DAWGs for strings have important applications to pattern matching with don't cares [20], online Lempel-Ziv factorization in compact space [27], finding minimal absent words [13], etc. CDAWGs for strings can be regarded as *grammar compression* of input strings and can be stored in space linear in the number of right-extensions of maximal repeats [3]. It is known that the number of maximal repeats can be much smaller than the string length, particularly in highly repetitive strings. Hence, studying and understanding DAWGs/CDAWGs for tries are very important and are expected to lead to further research on efficient processing of tries.

Omitted proofs and supplemental figures can be found in a full version [16].

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 $1 \leq i \leq j \leq |S|$, 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 $1 \leq i \leq |S|$, 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, $\overline{\mathbf{S}} = \{\overline{S} \mid S \in \mathbf{S}\}$.

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 $(v, a, u)_\mathsf{b} \in \mathsf{E_b}$.

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

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

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) $|Suffix(\mathsf{T_b})| \leq n - 1$ 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. Then, for all nodes u in the path from r to v, the total number of strings in the set $\{str_{\mathsf{f}}(u,l) \mid l \in \mathsf{L}_{\mathsf{f}}\} \subset \mathit{Suffix}(\mathsf{T}_{\mathsf{f}})$ 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 $|\mathsf{V}_{\mathsf{f}}|$ of nodes in T_{f} equals n, we obtain Fact 1-(b). The lower bound is valid for alphabets of size σ ranging from 2 to $\Theta(k) = \Theta(n)$.

3 Maximal Substrings in Forward/Backward Tries

Blumer et al. [6] 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 T_{f} and T_{b}, which will be the basis of our indexing structures for T_{f} and T_{b}.

Maximal Substrings on Forward Tries: For any substring X in a forward trie T_{f}, X is said to be *right-maximal* on T_{f} if (i) there are at least two distinct characters $a, b \in \Sigma$ such that $Xa, Xb \in \mathit{Substr}(\mathsf{T}_{\mathsf{f}})$, or (ii) X has an occurrence ending at a leaf of T_{f}. Also, X is said to be *left-maximal* on T_{f} if (i) there are at least two distinct characters $a, b \in \Sigma$ such that $aX, bX \in \mathit{Substr}(\mathsf{T}_{\mathsf{f}})$, or (ii) X has an occurrence beginning at the root of T_{f}. Finally, X is said to be *maximal* on T_{f} if X is both right-maximal and left-maximal in T_{f}. For any $X \in \mathit{Substr}(\mathsf{T}_{\mathsf{f}})$, let $\mathit{r\text{-}mxml}_{\mathsf{f}}(X)$, $\mathit{l\text{-}mxml}_{\mathsf{f}}(X)$, and $\mathit{mxml}_{\mathsf{f}}(X)$ 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 T_{f}, where $\alpha, \beta \in \Sigma^*$.

Maximal Substrings on Backward Tries: For any substring Y in a backward trie T_{b}, Y is said to be *left-maximal* on T_{b} if (i) there are at least two distinct characters $a, b \in \Sigma$ such that $aY, bY \in \mathit{Substr}(\mathsf{T}_{\mathsf{b}})$, or (ii) Y has an occurrence beginning at a leaf of T_{b}. Also, Y is said to be *right-maximal* on T_{b} if (i) there are at least two distinct characters $a, b \in \Sigma$ such that $Ya, Yb \in \mathit{Substr}(\mathsf{T}_{\mathsf{b}})$, or (ii) Y has an occurrence ending at the root of T_{b}. Finally, Y is said to be *maximal* on T_{b} if Y is both right-maximal and left-maximal in T_{b}. For any $Y \in \mathit{Substr}(\mathsf{T}_{\mathsf{b}})$, 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 T_{b}, where $\gamma, \delta \in \Sigma^*$.

Clearly, the afore-mentioned notions are symmetric over T_{f} and T_{b}, namely:

Fact 2. *String X is right-maximal (resp. left-maximal) on T_{f} iff $\overline{X}$ is left-maximal (resp. right-maximal) on T_{b}. Also, X is maximal on T_{f} iff $\overline{X}$ is maximal on T_{b}.*

4 Indexing Forward/Backward Tries and Known Bounds

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 $\mathit{Suffix}(\mathsf{T_f})$. 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, 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.*

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 $\mathit{Suffix}(\mathsf{T_b})$. 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 [7,19].

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 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 $\alpha = \mathit{str}_\mathsf{b}(u, v)$, $\mathsf{STree}(\mathsf{T_b})$ 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) [8]. 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.

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 $[\cdot]_{E,\mathsf{f}}$. 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 $[X]_{E,\mathsf{f}}$.

Mohri et al. [22] 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 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. [22] showed the following:

Theorem 3 (Corollary 2 of [22]**).** *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 $\mathit{Substr}(\mathsf{T_b})$. 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 $\mathit{Substr}(\mathsf{T_b})$, let $[Y]_{E,\mathsf{b}}$ denote the equivalence class w.r.t. $\mathit{l\text{-}mxml}_{\mathsf{b}}(Y)$. 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 $[\cdot]_{E,\mathsf{b}}$. The notions of primary edges, secondary edges, and the suffix links of $\mathsf{DAWG}(\mathsf{T_b})$ are defined in similar manners to $\mathsf{DAWG}(\mathsf{T_f})$, but using the equivalence classes $[Y]_{E,\mathsf{b}}$ for substrings Y in the backward trie $\mathsf{T_b}$.

Symmetries Between Suffix Trees and DAWGs: The well-known *symmetry* between the suffix trees and the DAWGs (refer to [5,6,10]) also holds in our case of forward and backward tries. Namely, the suffix links of $\mathsf{DAWG}(\mathsf{T_f})$ (resp. $\mathsf{DAWG}(\mathsf{T_b})$) are the (reversed) edges of $\mathsf{STree}(\mathsf{T_b})$ (resp. $\mathsf{STree}(\mathsf{T_f})$). Also, the hard W-links of $\mathsf{STree}(\mathsf{T_f})$ (resp. $\mathsf{STree}(\mathsf{T_b})$) are the primary edges of $\mathsf{DAWG}(\mathsf{T_b})$ (resp. $\mathsf{DAWG}(\mathsf{T_f})$), and the soft W-links of $\mathsf{STree}(\mathsf{T_f})$ (resp. $\mathsf{STree}(\mathsf{T_b})$) are the secondary edges of $\mathsf{DAWG}(\mathsf{T_b})$ (resp. $\mathsf{DAWG}(\mathsf{T_f})$).

4.5 CDAWGs for Forward Tries

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

Theorem 4 ([17]). *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. [17] states size bounds of $\mathsf{CDAWG}(\mathsf{T_f})$ 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.

4.6 CDAWGs for Backward Tries

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

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 Forward/Backward Tries

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.*

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 out-going 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 out-going edges with σ distinct characters $b_1, \ldots, b_\sigma$. 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 $1 \leq i \leq \sigma$. Now a^k for each $1 \leq k \leq n - \sigma - 2$ 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 $1 \leq k \leq n - \sigma - 2$. On the other hand, each $a^k \in \mathit{Substr}(\mathsf{T_f})$ is immediately followed by b_i with all $1 \leq i \leq \sigma$. 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. □

Mohri et al. (Proposition 4 of [22]) 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 1. *The DAWG construction algorithm of [22] 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* $\sigma = \Theta(n)$*.*

5.2 Size Bounds for CDAWGs for Forward/Backward Tries

Theorem 7. *For any backward trie* $\mathsf{T_b}$ *with* n *nodes,* $|\mathsf{CDAWG}(\mathsf{T_b})|_{\#Node} \leq 2n - 3$ *and* $|\mathsf{CDAWG}(\mathsf{T_b})|_{\#Edge} \leq 2n - 4$*. 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 $|\mathsf{CDAWG}(\mathsf{T_b})|_{\#Node} \leq |\mathsf{STree}(\mathsf{T_b})|_{\#Node} \leq 2n-3$ and $|\mathsf{CDAWG}(\mathsf{T_b})|_{\#Edge} \leq |\mathsf{STree}(\mathsf{T_b})|_{\#Edge} \leq 2n-4$. □

The bounds in Theorem 7 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 $d \geq 2$ are labeled by the sub-alphabet $\{a_d, b_d\}$ of size 2. 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 $S' \neq S$ that shares 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 $\mathsf{T_b}$.

Theorem 8. *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 7 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 to Theorem 6, $\mathsf{CDAWG}(\mathsf{T_f})$ for this $\mathsf{T_f}$ contains at least $\sigma(n-\sigma-2) = \Omega(\sigma n)$ edges, which accounts to $\Omega(n^2)$ for a large alphabet of size e.g. $\sigma \approx n/2$. □

The upper bound of Theorem 8 generalizes the bound of Theorem 4 for constant-size alphabets. Remark that $\mathsf{CDAWG}(\mathsf{T_f})$ for the broom-like $\mathsf{T_f}$ 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.

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 Sect. 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 [12]:

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 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 $\mathsf{STree}(\mathsf{T_b})$*.*

We construct a micro-macro tree decomposition [1] of $\mathsf{STree}(\mathsf{T_b})$ in a similar manner to [14], 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. Such a decomposition always exists and can be computed in $O(n)$ time. The *macro tree* is the induced tree from the 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 $O(n)$.

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 that V belongs to. 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 has 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 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.
  (ii) Otherwise, then there is no W-link from V for a.
(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 $Z = \mathit{slink}(U)$ is a descendant of V, then U is the destination of the soft W-link $\mathcal{W}_a(V)$ from V for a.
  (ii) Otherwise, then there is no W-link from V for a.

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.

Lemma 1. *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.*

Lemma 2. *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 1). For this sake, 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. □

Theorem 9. *Given a forward trie* $\mathsf{T_f}$ *of size* n *over an integer alphabet* $\Sigma = [1..\sigma]$ *with* $\sigma = O(n)$*, we can construct an* $O(n)$*-space representation of* $\mathsf{DAWG}(\mathsf{T_f})$ *in* $O(n)$ *time and working space.*

References

1. Alstrup, S., Secher, J.P., Spork, M.: Optimal on-line decremental connectivity in trees. Inf. Process. Lett. **64**(4), 161–164 (1997)
2. Apostolico, A., Lonardi, S.: Off-line compression by greedy textual substitution. Proc. IEEE **88**(11), 1733–1744 (2000)
3. Belazzougui, D., Cunial, F.: Fast label extraction in the CDAWG. In: Fici, G., Sciortino, M., Venturini, R. (eds.) SPIRE 2017. LNCS, vol. 10508, pp. 161–175. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67428-5_14
4. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. **321**(1), 5–12 (2004)
5. Blumer, A., et al.: The smallest automaton recognizing the subwords of a text. Theor. Comput. Sci. **40**, 31–55 (1985)
6. Blumer, A., Blumer, J., Haussler, D., Mcconnell, R., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. J. ACM **34**(3), 578–595 (1987)
7. Breslauer, D.: The suffix tree of a tree and minimizing sequential transducers. Theor. Comput. Sci. **191**(1–2), 131–144 (1998)
8. Breslauer, D., Italiano, G.F.: Near real-time suffix tree construction via the fringe marked ancestor problem. J. Discrete Algorithms **18**, 32–48 (2013)
9. Crochemore, M.: Transducers and repetitions. Theor. Comput. Sci. **45**(1), 63–86 (1986)
10. Crochemore, M., Rytter, W.: Text Algorithms. Oxford University Press, Cambridge (1994)
11. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM **57**(1), 4:1–4:33 (2009)
12. Fischer, J., Gawrychowski, P.: Alphabet-dependent string searching with wexponential search trees. In: Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching, CPM, pp. 160–171 (2015). Full version: https://arxiv.org/abs/1302.3347
13. Fujishige, Y., Tsujimaru, Y., Inenaga, S., Bannai, H., Takeda, M.: Computing DAWGs and minimal absent words in linear time for integer alphabets. In: Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science, MFCS, pp. 38:1–38:14 (2016)
14. Gawrychowski, P.: Simple and efficient LZW-compressed multiple pattern matching. J. Discrete Algorithms **25**, 34–41 (2014)

15. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
16. Inenaga, S.: Suffix trees, DAWGs and CDAWGs for forward and backward tries. arXiv e-prints p. 1904.04513 (2019). http://arxiv.org/abs/1904.04513
17. Inenaga, S., Hoshino, H., Shinohara, A., Takeda, M., Arikawa, S.: Construction of the CDAWG for a trie. In: Proceedings of the Prague Stringology Conference, PSC 2001, pp. 37–48 (2001)
18. Kimura, D., Kashima, H.: Fast computation of subpath kernel for trees. In: Proceedings of the 29th International Conference on Machine Learning, ICML (2012)
19. Kosaraju, S.R.: Efficient tree pattern matching (preliminary version). In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science, FOCS, pp. 178–183 (1989)
20. Kucherov, G., Rusinowitch, M.: Matching a set of strings with variable length don't cares. Theor. Comput. Sci. **178**(1–2), 129–154 (1997)
21. Mäkinen, V., Belazzougui, D., Cunial, F., Tomescu, A.I.: Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing. Cambridge University Press, Cambridge (2015)
22. Mohri, M., Moreno, P.J., Weinstein, E.: General suffix automaton construction algorithm and space bounds. Theor. Comput. Sci. **410**(37), 3553–3562 (2009)
23. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, pp. 657–666 (2002)
24. Nakashima, Y., I, T., Inenaga, S., Bannai, H., Takeda, M.: The position heap of a Trie. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 360–371. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34109-0_38
25. Shibuya, T.: Constructing the suffix tree of a tree with a large alphabet. IEICE Trans. **E86-A(5)**, 1061–1066 (2003)
26. Weiner, P.: Linear pattern-matching algorithms. In: Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory, pp. 1–11 (1973)
27. Yamamoto, J., Tomohiro, I., Bannai, H., Inenaga, S., Takeda, M.: Faster compact on-line Lempel-Ziv factorization. In: Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science, STACS, pp. 675–686 (2014)