Fully-online Construction of Suffix Trees for Multiple Texts

Takuya Takagi[1], Shunsuke Inenaga[2], and Hiroki Arimura[3]

1 Graduate School of IST, Hokkaido University, Japan
tkg@ist.hokudai.ac.jp
2 Department of Informatics, Kyushu University, Japan
inenaga@inf.kyushu-u.ac.jp
3 Graduate School of IST, Hokkaido University, Japan
arim@ist.hokudai.ac.jp

Abstract

We consider *fully-online* construction of indexing data structures for multiple texts. Let $\mathcal{T} = \{T_1, \dots, T_K\}$ be a collection of texts. By fully-online, we mean that a new character can be appended to any text in $\mathcal{T}$ *at any time*. This is a natural generalization of *semi-online* construction of indexing data structures for multiple texts in which, after a new character is appended to the kth text T_k, then its previous texts $T_1, \dots, T_{k-1}$ will remain static. Our fully-online scenario arises when we maintain dynamic indexes for multi-sensor data. Let N and σ denote the total length of texts in $\mathcal{T}$ and the alphabet size, respectively. We first show that the algorithm by Blumer et al. [Theoretical Computer Science, 40:31-55, 1985] to construct the *directed acyclic word graph* (*DAWG*) for $\mathcal{T}$ can readily be extended to our fully-online setting, retaining $O(N \log \sigma)$-time and $O(N)$-space complexities. Then, we give a sophisticated fully-online algorithm which constructs the *suffix tree* for $\mathcal{T}$ in $O(N \log \sigma)$ time and $O(N)$ space. A key idea of this algorithm is synchronized maintenance of the DAWG and the suffix tree.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases suffix trees, DAWGs, multiple texts, online algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.22

1 Introduction

Text indexing is a fundamental problem in computer science, which plays important roles in many applications including text retrieval, molecular biology, signal processing, and sensor data analysis. In this paper, we focus on indexing a collection of multiple texts, so that subsequent pattern matching queries can be answered quickly. In particular, we study online indexing for a collection $\mathcal{T}$ of multiple texts, where a new character can be appended to each text at *any* time. Such fully-online indexing for multiple growing texts has potential applications to continuous processing of data streams, where a number of symbolic events or data items are produced from multiple, rapid, time-varying, and unbounded data streams [2, 11]. For example, motif mining system tries to discover characteristic or interesting collective behaviors, such as frequent path or anomalies, from data streams generated by a collection of moving objects or sensors [14, 11].

It is known that suffix trees [13] and DAWGs [4] can be efficiently constructed for a collection of growing texts in the *semi-online* setting, where only the last inserted text can be grown. However, these existing semi-online algorithms to maintain a suffix tree or a DAWG for multiple texts are not sufficient to construct indexing structures for multiple data streams which grow in a fully-online manner.

We propose how the DAWG and the suffix tree can be incrementally constructed for a fully-online text collection. First, we observe that Blumer et al.'s construction [4] for DAWGs and Weiner's right-to-left construction [15] for suffix trees can readily be adapted to solve this problem. Hence, at any moment during the fully-online growth of the texts, we can find all *occ* occurrences of a given pattern of length M in the current text collection in $O(M \log \sigma + occ)$ time.

Our next goal is to extend Ukkonen's construction [13] to fully-online left-to-right construction of suffix trees for multiple texts. A motivation of this goal is that a growing suffix tree can be enhanced with powerful semi-dynamic tree data structures such as those for *nearest marked ancestor (NMA) queries* [16], *lowest common ancestor (LCA) queries* [7], and *level ancestor (LA) queries* [1]. Note that these data structures cannot be applied to DAWGs, and that the same query results cannot be obtained on the suffix tree maintained in a Weiner-like right-to-left online manner since the suffix tree obtained in this manner inherently indexes the *reversed* texts in the collection. However, it turns out that this goal is a big algorithmic challenge, because: (A) In Ukkonen's algorithm, a pointer called the *active point* keeps track of the insertion points of suffixes in decreasing order of length. The efficiency of Ukkonen's algorithm is due to the monotonicity of the tracking path of the active point. However, unfortunately this monotonicity does not hold in the fully-online setting for multiple texts. (B) Due to the non-monotonicity mentioned above, Ukkonen's technique to amortize the cost to track the suffix insertion points does not work in our case. (C) Ukkonen's "open edge" technique to maintain the leaves does not work in our case, either. In Section 5 we will explain in more details why and how these problems arise in our fully-online setting. In this paper, we present a number of new novel techniques to overcome all the difficulties above. As a final result, we propose the first *optimal* $O(N \log \sigma)$-time $O(N)$-space fully-online left-to-right construction algorithm for a suffix tree of multiple texts over a general ordered alphabet of size σ, where N is the final total length of the texts.

1.1 Related work

We note that we can obtain fully-online text index for multiple texts using existing more general dynamic text indices as follows. To use the index of Ferragina and Grossi [8] which permits character-wise updates, we build a text $\$_1 \cdots \$_K$ which initially consists only of K delimiters. Then, appending a character a to the kth text in the collection reduces to prepending a to the kth delimiter $\$_k$. Using this approach, the index of Ferragina and Grossi [8] takes $O(N \log N)$ total time to be constructed, requires $O(N \log N)$ space, and allows pattern matching in $O(M + \log N + N \log M + occ)$ time. Using the compressed index for a dynamic text collection of Chan et al. [6], we can append a new character a to the kth text T_k by removing T_k and then adding $T_k a$ in $O(|T_k|)$ time. This yields a fully-online index with $O(N^2 \log N)$ construction time and $O(N)$ bits of space (or $O(N/\log N)$ words of space assuming $\Theta(\log N)$-bit machine word), supporting pattern matching in $O(M \log N + occ \log^2 N)$ time.

2 Preliminaries

2.1 Strings

Let Σ be a general ordered alphabet. Any element of Σ^* is called a *string*. For any string T, let $|T|$ denote its length. Let ε be the empty string, namely, $|\varepsilon| = 0$. If $T = XYZ$, then X, Y, and Z are called a *prefix*, a *substring*, and a *suffix* of T, respectively. For any

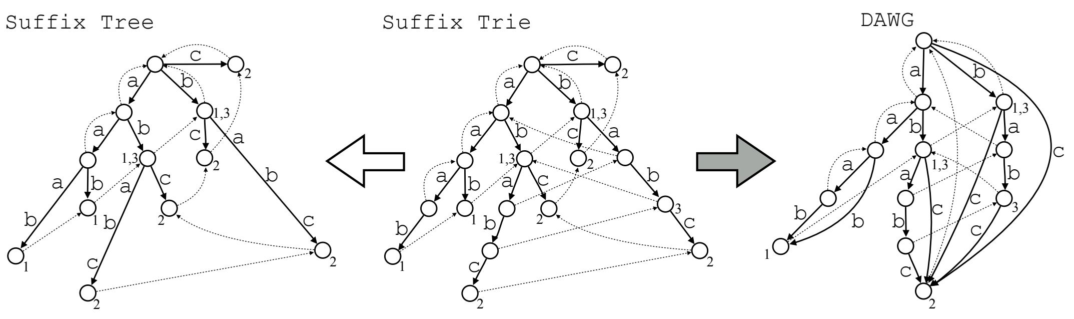

Figure 1 Illustration for *STrie*($\mathcal{T}$), *STree*($\mathcal{T}$), and *DAWG*($\mathcal{T}$) with $\mathcal{T} = \{T_1 = \texttt{aaab}, T_2 = \texttt{ababc}, T_3 = \texttt{bab}\}$. The solid arrows and broken arrows represent the edges and the suffix links of each data structure, respectively. The number k ($k = 1, 2, 3$) beside each node indicates that the node represents a suffix of T_k. The nodes $[\texttt{ab}]_{\mathcal{T}}$ and $[\texttt{b}]_{\mathcal{T}}$ are separated in *DAWG*($\mathcal{T}$) since the node bab in *STrie*($\mathcal{T}$) is represents a suffix of T_3, while the node abab does not (see also the subtrees rooted at nodes ab and b in *STrie*($\mathcal{T}$)).

$1 \leq i \leq j \leq |T|$, let $T[i..j]$ denote the substring of T that begins at position i and ends at position j in T. For any $1 \leq i \leq |T|$, let $T[i]$ denote the ith character of T. For any string T, let *Suffix*(T) denote the set of suffixes of T, and for any set $\mathcal{T}$ of strings, let *Suffix*($\mathcal{T}$) denote the set of suffixes of all strings in $\mathcal{T}$. Namely, $\mathit{Suffix}(\mathcal{T}) = \bigcup_{T \in \mathcal{T}} \mathit{Suffix}(T)$. For any string T, let $\overline{T}$ denote the reversed string of T, i.e., $\overline{T} = T[|T|] \cdots T[1]$.

Let $\mathcal{T} = \{T_1, \ldots, T_K\}$ be a collection of K texts. For any $1 \leq k \leq K$, let $\mathit{lrs}_{\mathcal{T}}(T_k)$ be the longest repeating suffix of T_k that occurs at least twice in $\mathcal{T}$.

2.2 Suffix trees and DAWGs for multiple texts

The suffix trie for a text collection $\mathcal{T} = \{T_1, \ldots, T_K\}$, denoted *STrie*($\mathcal{T}$), is a trie which represents *Suffix*($\mathcal{T}$). The size of *STrie*($\mathcal{T}$) is $O(N^2)$, where N is the total length of texts in $\mathcal{T}$. We identify each node v of *STrie*($\mathcal{T}$) with the string that v represents. A substring x of a text in $\mathcal{T}$ is said to be *branching* in $\mathcal{T}$, if there exist two distinct characters $a, b \in \Sigma$ such that both xa and xb are substrings of some texts in $\mathcal{T}$. Clearly, node x of *STrie*($\mathcal{T}$) is branching iff x is branching in $\mathcal{T}$. For each node av of *STrie*($\mathcal{T}$) with $a \in \Sigma$ and $v \in \Sigma^*$, let $\mathit{slink}(av) = v$. This auxiliary edge $\mathit{slink}(av) = v$ from av to v is called a *suffix link*.

The *suffix tree* [15] for a text collection $\mathcal{T}$, denoted *STree*($\mathcal{T}$), is a "compacted trie" which represents *Suffix*($\mathcal{T}$). *STree*($\mathcal{T}$) is obtained by compacting every path of *STrie*($\mathcal{T}$) which consists of non-branching internal nodes (see Fig. 1). Since every internal node of *STree*($\mathcal{T}$) is branching, and since there are at most N leaves in *STree*($\mathcal{T}$), the numbers of edges and nodes are $O(N)$. The edge labels of *STree*($\mathcal{T}$) are non-empty substrings of some text in $\mathcal{T}$. By representing each edge label x with a triple $\langle k, i, j \rangle$ of integers s.t. $x = T_k[i..j]$, *STree*($\mathcal{T}$) can be stored with $O(N)$ space. We say that any branching (resp. non-branching) substring of $\mathcal{T}$ is an *explicit node* (resp. *implicit node*) of *STree*($\mathcal{T}$). An implicit node x is represented by a triple (v, a, ℓ), called a *reference* to x, such that v is an explicit ancestor of x, a is the first character of the path from v to x, and ℓ is the length of the path from v to x. A reference (v, a, ℓ) to node x is called *canonical* if v is the lowest explicit ancestor of x. For each node av of *STree*($\mathcal{T}$) with $a \in \Sigma$ and $v \in \Sigma^*$, let $\mathit{slink}(av) = v$.

The *directed acyclic word graph* [3, 4] of a text collection $\mathcal{T}$, denoted *DAWG*($\mathcal{T}$), is a smallest DAG which represents *Suffix*($\mathcal{T}$). *DAWG*($\mathcal{T}$) is obtained by merging identical subtrees of *STrie*($\mathcal{T}$) connected by the suffix links (see Fig. 1). Hence, the label of every edge of *DAWG*($\mathcal{T}$) is a single character. The numbers of nodes and edges of *DAWG*($\mathcal{T}$) are

$O(N)$ [4], and hence $DAWG(\mathcal{T})$ can be stored with $O(N)$ space. $DAWG(\mathcal{T})$ can be defined formally as follows: For any string x, let $Epos_{\mathcal{T}}(x)$ be the set of ending positions of x in the texts in $\mathcal{T}$, i.e., $Epos_{\mathcal{T}}(x) = \{(k, j) \mid x = T_k[j - |x| + 1..j], 1 \leq j \leq |T_k|, 1 \leq k \leq K\}$. Consider an equivalence relation $\equiv_{\mathcal{T}}$ on substrings x, y of texts in $\mathcal{T}$ such that $x \equiv_{\mathcal{T}} y$ iff $Epos_{\mathcal{T}}(x) = Epos_{\mathcal{T}}(y)$. For any substring x of texts of $\mathcal{T}$, let $[x]_{\mathcal{T}}$ denote the equivalence class w.r.t. $\equiv_{\mathcal{T}}$. There is a one-to-one correspondence between each node v of $DAWG(\mathcal{T})$ and each equivalence class $[x]_{\mathcal{T}}$, and hence we will identify each node v of $DAWG(\mathcal{T})$ with its corresponding equivalence class $[x]_{\mathcal{T}}$. Let $long([x]_{\mathcal{T}})$ denote the longest member of $[x]_{\mathcal{T}}$. By the definition of equivalence classes, $long([x]_{\mathcal{T}})$ is unique for each $[x]_{\mathcal{T}}$ and every member of $[x]_{\mathcal{T}}$ is a suffix of $long([x]_{\mathcal{T}})$. If x, xa are substrings of some text in $\mathcal{T}$ with $x \in \Sigma^*$ and $a \in \Sigma$, then there exists an edge labeled with character $a \in \Sigma$ from node $[x]_{\mathcal{T}}$ to node $[xa]_{\mathcal{T}}$. This edge is called *primary* if $|long([x]_{\mathcal{T}})| + 1 = |long([xa]_{\mathcal{T}})|$, and is called *secondary* otherwise. For each node $[x]_{\mathcal{T}}$ of $DAWG(\mathcal{T})$ with $|x| \geq 1$, let $slink([x]_{\mathcal{T}}) = y$, where y is the longest suffix of $long([x]_{\mathcal{T}})$ which does not belong to $[x]_{\mathcal{T}}$. In the example of Fig. 1, $[\texttt{aaab}]_{\mathcal{T}} = \{\texttt{aaab}, \texttt{aab}\}$. The edge labeled with $\texttt{b}$ from node $[\texttt{aaa}]_{\mathcal{T}}$ to node $[\texttt{aaab}]_{\mathcal{T}}$ is primary, while the edge labeled with $\texttt{b}$ from $[\texttt{aa}]_{\mathcal{T}}$ to node $[\texttt{aaab}]_{\mathcal{T}}$ is secondary. $slink([\texttt{aaab}]_{\mathcal{T}}) = [\texttt{ab}]_{\mathcal{T}}$.

The following fact follows from the definition of branching substrings:

▶ **Fact 1.** *For any substring x of texts in $\mathcal{T}$, node x is branching (explicit) in $STree(\mathcal{T})$ iff node $[x]_{\mathcal{T}}$ is branching in $DAWG(\mathcal{T})$.*

2.3 Fully-online text collection

We consider a collection $\{T_1, \ldots, T_K\}$ of K growing texts, where each text T_k $(1 \leq k \leq K)$ is initially the empty string ε. Given a pair (k, a) of a text id k and a character $a \in \Sigma$ which we call an *update operator*, the character a is appended to the k-th text of the collection. For a sequence U of update operators, let $U[1..i]$ denote the sequence of the first i update operators in U with $0 \leq i \leq |U|$. Also, for $0 \leq i \leq |U|$ let $\mathcal{T}_{U[1..i]}$ denote the collection of texts which have been updated according to the first i update operators of U. For instance, consider a text collection of three texts which grow according to the following sequence $U = (1, \texttt{a}), (2, \texttt{b}), (2, \texttt{a}), (3, \texttt{a}), (1, \texttt{a}), (3, \texttt{c}), (3, \texttt{b}), (2, \texttt{b}), (1, \texttt{a}), (1, \texttt{b}), (3, \texttt{c}), (3, \texttt{b}), (1, \texttt{c}), (3, \texttt{b}),$ $(2, \texttt{c})$ of 15 update operators. Then,

$$\mathcal{T}_{U[1..0]} = \left\{ \begin{array}{c} \varepsilon \\ \varepsilon \\ \varepsilon \end{array} \right\}, \ \ldots, \ \mathcal{T}_{U[1..14]} = \left\{ \begin{array}{llllll} \overset{1}{\texttt{a}} & \overset{5}{\texttt{a}} & \overset{9}{\texttt{a}} & \overset{10}{\texttt{b}} & \overset{13}{\texttt{c}} & \\ \overset{2}{\texttt{b}} & \overset{3}{\texttt{a}} & \overset{8}{\texttt{b}} & & & \\ \overset{4}{\texttt{a}} & \overset{6}{\texttt{c}} & \overset{7}{\texttt{b}} & \overset{11}{\texttt{c}} & \overset{12}{\texttt{b}} & \overset{14}{\texttt{b}} \end{array} \right\}, \ \mathcal{T}_{U[1..15]} = \left\{ \begin{array}{llllll} \overset{1}{\texttt{a}} & \overset{5}{\texttt{a}} & \overset{9}{\texttt{a}} & \overset{10}{\texttt{b}} & \overset{13}{\texttt{c}} & \\ \overset{2}{\texttt{b}} & \overset{3}{\texttt{a}} & \overset{8}{\texttt{b}} & \overset{15}{\texttt{c}} & & \\ \overset{4}{\texttt{a}} & \overset{6}{\texttt{c}} & \overset{7}{\texttt{b}} & \overset{11}{\texttt{c}} & \overset{12}{\texttt{b}} & \overset{14}{\texttt{b}} \end{array} \right\}$$

where the superscript i over each character a in the k-th text implies that $U[i] = (k, a)$. For instance, $U[15] = (2, \texttt{c})$ and hence $\texttt{c}$ was appended to the 2nd text $T_2 = \texttt{bab}$ in $\mathcal{T}_{U[1..14]}$, yielding $T_2 = \texttt{babc}$ in $\mathcal{T}_{U[1..15]}$.

If there is no restriction on U like the one in the example above, then U is called *fully-online*. If there is a restriction on U such that once a new character is appended to the k-th text, then no characters will be appended to its previous $k - 1$ texts, then U is called *semi-online*. Hence, any semi-online sequence of update operators is of form $(1, T_1[1]), \ldots, (1, T_1[|T_1|]), \ldots, (K, T_K[1]), \ldots, (K, T_K[|T_K|])$.

Section 3 reviews previous algorithms which incrementally construct the DAWG and the suffix tree for a growing text collection in the semi-online setting. Sections 4 and 5 propose our new algorithms which incrementally construct the DAWG and the suffix tree for a text collection in the fully-online setting, respectively.

3 Semi-online construction algorithms

3.1 Blumer et al.'s semi-online DAWG construction algorithm

We recall Blumer et al.'s algorithm [4] which incrementally builds $DAWG(\mathcal{T}_U)$ for a given semi-online sequence U of update operators of length N. Since U is semi-online, at each step i $(0 \leq i \leq N)$ of the semi-online update, there exists a unique k $(1 \leq k < K)$ such that $T_1, \ldots, T_{k-1}$ will be static for all the following i'th steps $(i \leq i' \leq N)$, T_k is now growing from left to right, and $T_{k+1}, \ldots, T_K$ are still the empty strings. Assume that $U[i] = (k, a)$, and hence a new character a is appended to the kth text in the collection at the ith step. For ease of notation, let $\mathcal{T}' = \mathcal{T}_{U[1..i-1]}$ and $\mathcal{T} = \mathcal{T}_{U[1..i]}$. Also, assume that $DAWG(\mathcal{T}')$ has already been constructed. In updating $DAWG(\mathcal{T}')$ to $DAWG(\mathcal{T})$, we have to assure that all suffixes of the extended text $T_k a$ will be represented by $DAWG(\mathcal{T})$. These suffixes are categorized to three different types:

Type-1 The suffixes of $T_k a$ that are longer than $lrs_{\mathcal{T}'}(T_k)a$.

Type-2 The suffixes of $T_k a$ that are not longer than $lrs_{\mathcal{T}'}(T_k)a$ and are longer than $lrs_{\mathcal{T}}(T_k a)$.

Type-3 The suffixes of $T_k a$ that are not longer than $lrs_{\mathcal{T}}(T_k a)$.

Blumer et al.'s algorithm inserts the suffixes of $T_k a$ in decreasing order of length, from the Type-1 ones to the Type-2 ones. By definition, the Type-3 ones are already represented by $DAWG(\mathcal{T}')$, and hence, we need not insert them explicitly.

Their algorithm maintains an invariant v which indicates node $[T_k]_{\mathcal{T}'}$, called the *active point*, from which the update starts. There are two cases to happen:

1. If there is an out-going edge labeled with a from v, then $T_k a = lrs_{\mathcal{T}}(T_k a)$, which implies all suffixes of $T_k a$ are of Type-3. There are two subcases:
   - **a.** If the edge labeled with a is primary, then no updates to the graph topology are needed. The new active point for the next step is on $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$.
   - **b.** If the edge labeled with a is secondary, then the graph topology needs to be updated. Since the edge is secondary, every member Xa of $u = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}'}$ that is longer than $T_k a$ is not a suffix of $T_k a$, while every member Ya of $u = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}'}$ that is not longer than $T_k a$ is a Type-3 suffix of $T_k a$. This implies that $Epos_{\mathcal{T}}(lrs_{\mathcal{T}}(T_k a)) \supset Epos_{\mathcal{T}}(Xa)$. By the definition of the nodes of DAWGs (recall Section 2.2), the node u is split into two nodes $z = [Xa]_{\mathcal{T}}$ and $w = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$: First, a new node w is created. All secondary in-coming edges of u corresponding to Type-3 suffixes Ya are redirected to w. This can be done by traversing the chain of the suffix links starting from v. All the out-going edges of u are copied to w. Now, node w is complete, and the node u with its remaining in-coming edges is the other new node z. The suffix link of u is inherited by w, and the suffix link of z is set to w. The new active point for the next step is on node w.
2. If there is no out-going edge labeled with a from the active point v, then a new sink s is created. The Type-1 suffixes are inserted by making a new edge labeled by a from $v = [T_k]_{\mathcal{T}'}$ to s. To insert the Type-2 suffixes, the active point v moves by updating $v \leftarrow slink(v)$. Then the following procedure is repeated until an out-going edge labeled with a from the active point is found: (i) A new edge labeled with a from v to s is created. (ii) The active point v moves by updating $v \leftarrow slink(v)$. The node u where the above procedure ends is $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}'}$, and the new sink s is exactly $[T_k a]_{\mathcal{T}}$ which represent all Type-1 and Type-2 suffixes of $T_k a$. There are two cases:

a. If the edge labeled with a from the last locus v of the active point to u is primary, then $u = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$. Thus no updates to the graph topology are needed. The suffix link of the new sink $s = [T_k a]_{\mathcal{T}}$ is set to u.

b. If the edge labeled with a from the last locus v of the active point to u is secondary, then as in Case 1b, u is split into two nodes w and z where w represents the members of u that are longer than the longest repeating suffix $lrs_{\mathcal{T}}(T_k a)$ (none of these members is a suffix of $T_k a$), and z represents the members of u which are Type-3 suffixes of $T_k a$. The suffix link of the new sink s is set to z.

In both subcases above, the new active point is on the new sink $s = [T_k a]_{\mathcal{T}}$.

It is not difficult to see that if the total number of new nodes, edges, and suffix links is q, then the above update takes $O(q \log \sigma)$ time, where the $\log \sigma$ term is due to searching for an out-going edge labeled by a. Since no existing nodes, edges, or suffix links are deleted during the updates, and since the size of $DAWG(\mathcal{T}_U)$ is $O(N)$, the amortized time for the update is $O(\log \sigma)$. Hence, $DAWG(\mathcal{T}_U)$ can be constructed in $O(N \log \sigma)$ time and $O(N)$ space in the semi-online setting.

3.2 Ukkonen's semi-online suffix tree construction algorithm

Ukkonen [13] proposed an algorithm to incrementally construct the suffix tree of a single text. His algorithm can easily be extended to incrementally construct the suffix tree for multiple texts in the semi-online setting.

Let U be a semi-online sequence of N update operators such that the last update operator for each k $(1 \leq k \leq K)$ is $(k, \$_k)$, where $\$_k$ is a special end-marker for the kth text in the collection. For ease of notation, $\mathcal{T}' = \mathcal{T}_{u[1..i-1]}$ and $\mathcal{T} = \mathcal{T}_{u[1..i]}$, Also, assume that we have already constructed $STree(\mathcal{T}')$ and that the next update operator is $U[i] = (k, a)$. Thus a new character a is appended to the kth text T_k of $\mathcal{T}'$, and the kth text of $\mathcal{T}$ becomes $T_k a$.

As in the case of semi-online DAWG construction, the suffixes of $T_k a$ are inserted in decreasing order of length. The Type-1 suffixes are maintained as follows. Let s be any suffix of T_k which is represented by a leaf of $STree(\mathcal{T}')$. Since s is a non-repeating suffix of T_k in $\mathcal{T}'$, sa is a non-repeating suffix of $T_k a$ in $\mathcal{T}$, which implies that sa will also be a leaf of $STree(\mathcal{T})$. Based on this observation, the label of the in-coming edge of s is represented by a triple $\langle k, b, \infty \rangle$ called an *open edge*, where b is the beginning position of the label of the in-coming edge in the kth text. This way, every existing leaf will then be automatically extended. Hence, updating $STree(\mathcal{T}')$ to $STree(\mathcal{T})$ reduces to inserting the Type-2 suffixes of $T_k a$. For this sake, the algorithm maintains an invariant which indicates the locus of $x = lrs_{\mathcal{T}'}(T_k)$ on $STree(\mathcal{T}')$ called the *active point*. Since x can be an implicit node, the algorithm maintains the canonical reference (v, c, ℓ) to x. For convenience, if x is an explicit node, then let its canonical reference be $(x, \varepsilon, 0)$. The update starts from the current active point x represented by its canonical reference pair, and the Type-2 suffixes of $T_k a$ are inserted in decreasing order of length, by using the chain of (virtual) suffix links. There are two cases:

I. If it is possible to go down from x with character a, then no updates to the tree topology are needed. The new active point is xa, and the reference to xa is made canonical if necessary. The update ends.

II. If it is impossible to go down from x with character a, then we create a new leaf. Let j be the beginning position of the suffix of $T_k a$ which corresponds to this new leaf. The following procedure is repeated until Case I happens.

1. If the active point x is on an explicit node, then a new leaf node s is created as a new child of x, with its incoming edge labeled by $\langle k, b, \infty \rangle$, where $b = |T_k a| - |x| + 1$. The active point x is updated to $\mathit{slink}(x)$.
2. If the active point x is on an implicit node, then x becomes explicit in this step. A new leaf node s is created as a new child of x with its incoming edge labeled by $\langle k, b, \infty \rangle$. Since the suffix link of the new explicit node x does not yet exist, we simulate the suffix link traversal as follows: Let (v_j, c_j, ℓ_j) be the canonical reference to x. First, we follow the suffix link $\mathit{slink}(v_j)$ of v_j, and then go down along the path of length ℓ_j from $\mathit{slink}(v_j)$ starting with character c_j. Let this locus be x'. Let v_{j+1} be the longest explicit node in this path.
   - **(i)** If $|v_{j+1}| = |x'|$, then we firstly create the new suffix link $\mathit{slink}(x) = v_{j+1}$ for the new explicit node x. The active point x is updated to x' and is represented by canonical reference $(v_{j+1}, \varepsilon, 0)$.
   - **(ii)** If $|v_{j+1}| < |x'|$, then the next active point is implicit. The active point x is updated to x' and is represented by canonical reference $(v_{j+1}, c_{j+1}, \ell_{j+1})$. The suffix link of x will be set to x' when x' becomes explicit in the next step.

The most expensive case is II-b-(ii). Since the path from v_{j+1} to x' contains at most $\ell_j - \ell_{j+1}$ explicit nodes, it takes $O((\ell_j - \ell_{j+1} + 1)\log\sigma)$ time to locate the next active point x' (note $\ell_j - \ell_{j+1} \geq 0$ holds). All the other operations take $O(\log\sigma)$ time. Hence, the total cost to insert all leaves (suffixes) for the kth text is $O(\sum_{j=1}^{N_k}(\ell_j - \ell_{j+1} + 1)\log\sigma) = O(N_k \log\sigma)$, where N_k is the final length of the kth text. Thus the amortized time cost for each leaf (suffix) for the kth text is $O(\log\sigma)$. Overall, it takes a total of $O(N\log\sigma)$ time to construct $\mathit{STree}(\mathcal{T}_U)$ for a semi-online sequence U of update operators. The space requirement is $O(N)$.

4 Fully-online DAWG construction algorithm

We can easily extend Blumer et al.'s semi-online DAWG construction algorithm to the fully-online setting. Let U be a fully-online sequence of N update operators. Our fully-online algorithm maintains the active point v_k for *every* growing text T_k in the collection, at any step of the algorithm. Now, assume that we have already constructed $\mathit{DAWG}(\mathcal{T}')$, where $\mathcal{T}' = \mathcal{T}_{U[1..i-1]}$ for $1 \leq i \leq N$. Let $U[i] = (k, a)$, and we are updating $\mathit{DAWG}(\mathcal{T}')$ to $\mathit{DAWG}(\mathcal{T})$, where $\mathcal{T} = \mathcal{T}_{U[1..i]}$. The update starts from the active point $v_k = [T_k]_{\mathcal{T}'}$, exactly in the same way as was described in Section 3. The total cost to update $\mathit{DAWG}(\mathcal{T}')$ to $\mathit{DAWG}(\mathcal{T})$ is again $O(q\log\sigma)$, where q is the total number of nodes, edges, and suffix links which were introduced in this update. Since the total size of $\mathit{DAWG}(\mathcal{T})$ is $O(N)$, the amortized cost for this update is again $O(\log\sigma)$. By the above arguments, we obtain the following theorem.

▶ **Theorem 1.** *Given a fully-online sequence U of N update operators for a collection of K texts, we can update $\mathit{DAWG}(\mathcal{T}_{U[1..i]})$ for $i = 1, \ldots, N$ in a total of $O(N\log\sigma)$ time and $O(N)$ space.*

Assume for now that each text T_k in a collection $\mathcal{T}$ begins with a special character $\#_k$ which does not appear elsewhere in $\mathcal{T}$. Then, the tree of the (reversed) suffix links of $\mathit{DAWG}(\mathcal{T})$ forms the suffix tree $\mathit{STree}(\overline{\mathcal{T}})$ for the collection $\overline{\mathcal{T}} = \{\overline{T_1}, \ldots, \overline{T_K}\}$ of the reversed texts of $\mathcal{T}$ [4]. Hence, the next corollary follows from Theorem 1, which gives *right-to-left* fully-online suffix tree construction.

▶ **Corollary 2.** *Given a fully-online sequence U of N update operators for a collection of K texts, we can update $STree(\overline{\mathcal{T}_{U[1..i]}})$ for $i = 1, \ldots, N$ in a total of $O(N \log \sigma)$ time and $O(N)$ space.*

5 Fully-online suffix tree construction algorithm

5.1 Difficulties in fully-online construction of suffix trees

Unlike the case with DAWGs, it is not easy to extend Ukkonen's semi-online suffix tree construction algorithm to our left-to-right fully-online setting, because:

(A) Let $U[i] = (k, a)$ which updates the current kth text T_k to $T_k a$, and assume that we have just constructed $STree(\mathcal{T}_{U[1..i]})$. Recall that we defined the initial locus of the active point for $T_k a$ on $STree(\mathcal{T}_{U[1..i]})$ to be the longest repeating suffix of $T_k a$ in $\mathcal{T}_{U[1..i]}$. However, since U is fully-online, any other text T_h $(h \neq k)$ in the collection would be updated by following update operators $U[r]$ with $r > i$. Then, the longest repeating suffix of $T_k a$ in $\mathcal{T}_{U[1..r]}$ can be much longer than that of $T_k a$ in $\mathcal{T}_{U[1..i]}$. In other words, some Type-1 suffixes of $T_k a$ in $\mathcal{T}_{U[1..i]}$ can become of Type-2 in $\mathcal{T}_{U[1..r]}$. What is worse, updating T_h can affect the longest repeating suffix of any other text in the collection as well. If we maintain all these active points naïvely, it takes $O(KN \log \sigma)$ time.

(B) Even if we somehow manage to efficiently maintain the active point for each text in the collection, there remains another difficulty. Let j be the beginning position of the longest repeating suffix of $T_k a$ in $\mathcal{T}_{U[1..i]}$, and let (v_j, c_j, ℓ_j) be the canonical reference to this suffix. Let $U[i'] = (k, a')$ be the first update operator in U which updates the kth text after $U[i] = (k, a)$. Let (v'_j, c'_j, ℓ'_j) be the canonical reference to the longest repeating suffix of $T_k a$ in $\mathcal{T}_{U[1..i']}$, which is the "real" initial active point where insertion of the Type-2 suffixes should start at this i'th step. By the property of suffix trees $\ell'_j \geq \ell_j$ holds, and what is worse, this length ℓ'_j is unbounded by the number of Type-2 suffixes inserted at this i'th step. Thus, the amortization technique we used for the semi-online construction does not work in the fully-online setting.

(C) The phenomenon mentioned in Difficulty A also causes a problem of how to represent the labels of the in-coming edges to the leaves. Assume that we created a new leaf w.r.t. an update operator (k, a), and let $\langle k, b_k, \infty \rangle$ be the triple representing the label of the in-coming edge to the leaf, where b_k is the beginning position of the edge label in the kth text. It corresponds to a Type-1 suffix of the kth text, but the leaf can later be extended by another growing text T_h. Then, the triple $\langle k, b_k, \infty \rangle$ has to be updated to $\langle h, b_h, \infty \rangle$, where b_h is the beginning position of the edge label in the hth text. Notice that this update may happen repeatedly.

5.2 Constructing suffix trees with the aid of DAWGs

We utilize DAWGs to overcome Difficulties A, B and C in fully-online construction of suffix trees. Namely, we construct $STree(\mathcal{T})$ in tandem with $DAWG(\mathcal{T})$.

A high-level description of our algorithm is as follows. We insert the Type-2 suffixes of $T_k a$ in *increasing order* of length, starting from the locus of the longest Type-3 suffix of $T_k a$. The idea of inserting the Type-2 suffixes in increasing order of length was also used by Breslauer and Italiano [5], for quasi real-time left-to-right construction of the suffix tree for a single text. To efficiently find the locus where the next longer Type-2 suffix should be inserted in the tree from the locus where the last Type-2 suffix was inserted, we introduce a simpler amortized variant of the *suffix tree oracle* of Fischer and Gawrychowski [10, 9].

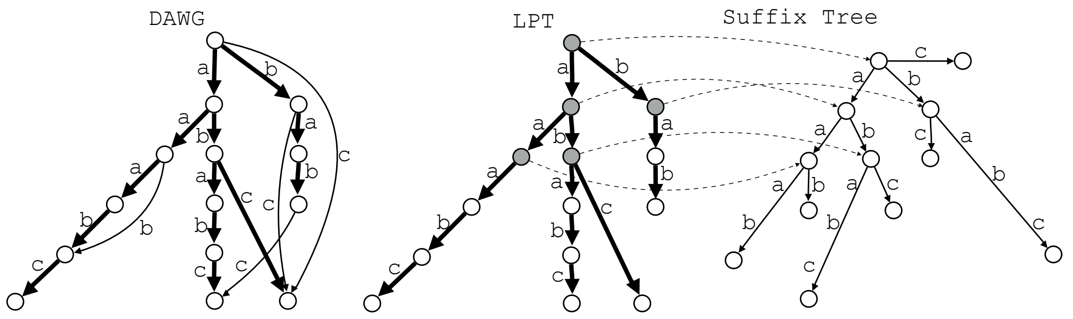

Figure 2 Illustration for $DAWG(\mathcal{T})$, $LPT(\mathcal{T})$, and $STree(\mathcal{T}')$, where $\mathcal{T}' = \{T_1 = \texttt{aaab}, T_2 = \texttt{ababc}, T_3 = \texttt{bab}\}$ and $\mathcal{T} = \{T_1\texttt{c}, T_2, T_3\}$. The bold solid arrows represent the primary edges of $DAWG(\mathcal{T})$, the gray nodes are the marked nodes of $LPT(\mathcal{T})$, and the dashed arrows represent the links between the marked nodes of $LPT(\mathcal{T})$ and the corresponding branching nodes of $STree(\mathcal{T}')$. $lrs_{\mathcal{T}}(T_1\texttt{c}) = \texttt{abc}$, and hence we perform an NMA query from node `abc` on $LPT(\mathcal{T})$, obtaining node `ab`. We then access the suffix tree node `ab` using the pointer from $LPT(\mathcal{T})$, and obtain the locus of `abc` on $STree(\mathcal{T}')$.

These will overcome Difficulties A and B. To overcome Difficulty C, we introduce new *lazy representation* of the labels of edges leading to the leaves. The next is a key lemma.

▶ **Lemma 3.** *We can compute, in amortized $O(\log \sigma)$ time, the canonical reference to the longest Type-3 suffix $lrs_{\mathcal{T}}(T_k a)$ of $T_k a$ on $STree(\mathcal{T}')$, using a data structure which requires space linear in the total length of the texts in $\mathcal{T}$.*

Proof. We introduce the *longest path tree* of $\mathcal{T}'$, denoted $LPT(\mathcal{T}')$, which is the spanning tree of $DAWG(\mathcal{T}')$ consisting only of the primary edges of $DAWG(\mathcal{T}')$. Every node of $LPT(\mathcal{T}')$ is marked iff its corresponding node on $DAWG(\mathcal{T}')$ is branching. Every marked node of $LPT(\mathcal{T})$ is linked to its corresponding node of $STree(\mathcal{T}')$ which is also branching by Fact 1 (see also Fig. 2). $LPT(\mathcal{T}')$ is enhanced with the *nearest marked ancestor* (*NMA*) data structure of Westbrook [16], which supports the following operations in amortized $O(1)$ time using linear space: 1) find the NMA of any node; 2) insert an unmarked node; 3) mark an unmarked node.

When $DAWG(\mathcal{T}')$ is updated to $DAWG(\mathcal{T})$, at most two new primary edges are introduced to $DAWG(\mathcal{T})$, one for the new sink and one for the split node. We insert these new edges to $LPT(\mathcal{T}')$ and obtain $LPT(\mathcal{T})$. Because of these new edges, at most two non-branching nodes of $DAWG(\mathcal{T}')$ can become branching in $DAWG(\mathcal{T})$. We mark their corresponding nodes in $LPT(\mathcal{T})$, and link them to the corresponding suffix tree nodes after we have constructed $STree(\mathcal{T})$. This is because the corresponding nodes of $STree(\mathcal{T}')$ are still non-branching.

We use $LPT(\mathcal{T})$ to quickly move from the DAWG to the suffix tree (see Fig. 2 for a concrete example). Since $lrs_{\mathcal{T}}(T_k a)$ is the longest in $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$, there always exists a node y of $LPT(\mathcal{T})$ which represents $lrs_{\mathcal{T}}(T_k a)$. We conduct an NMA query from y on $LPT(\mathcal{T})$, and let v be the NMA of y. Let $\ell = |y| - |v|$, and let c be the label of the first edge in the path from v to y. We move from v to its corresponding node x in $STree(\mathcal{T}')$. Then, (x, c, ℓ) is a reference to $lrs_{\mathcal{T}}(T_k a)$ in $STree(\mathcal{T}')$. Since v is the NMA of y in $LPT(\mathcal{T})$, and since updating T_k to $T_k a$ does not explicitly insert any suffix shorter than $lrs_{\mathcal{T}}(T_k a)$, this reference is canonical by Fact 1.

Clearly the total size of the above data structures is linear in the total length of the texts in $\mathcal{T}$. We analyze the time complexity. Recall Case 2 when updating $DAWG(\mathcal{T}')$ to

$DAWG(\mathcal{T})$. At the end of the update, we find (or create) in amortized $O(\log \sigma)$ time the node of $DAWG(\mathcal{T})$ which represents $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$. Hence we can find node $y = lrs_{\mathcal{T}}(T_k a)$ in amortized $O(\log \sigma)$ time. Updating $LPT(\mathcal{T}')$ to $LPT(\mathcal{T})$ takes $O(\log \sigma)$ time. Inserting a new node and querying an NMA from a given node takes amortized $O(1)$ time. We can link a new marked node of $LPT(\mathcal{T})$ to the corresponding new branching node of $STree(\mathcal{T})$ in $O(1)$ time, since we can remember this new branching node when updating $STree(\mathcal{T}')$ to $STree(\mathcal{T})$. Hence, the amortized bound is $O(\log \sigma)$. ◀

To find the insertion point of the shortest Type-2 suffix from the longest Type-3 suffix $lrs_{\mathcal{T}}(T_k a)$, and to insert the Type-2 suffixes of $T_k a$ in increasing order of length, we maintain the labeled reversed suffix links for each explicit node of the suffix tree. Namely, if $slink(bv) = v$ for two nodes bv, v with $v \in \Sigma^*$ and $b \in \Sigma$, let $rslink_b(v) = bv$. We leave $rslink_b(v)$ undefined if bv is not a substring of any text in the collection, or if node bv is implicit in the suffix tree.

A *suffix tree oracle* for a suffix tree S is a data structure which efficiently answers the following query: given a pair (v, b) of a node v of S and a character $b \in \Sigma$, return the nearest ancestor u of v for which $rslink_b(u)$ is defined. The state-of-the-art suffix tree oracle by Fischer and Gawrychowski [10, 9] answers queries and supports updates in worst-case $O(\log\log n + (\log\log\sigma)^2/\log\log\log\sigma)$ time each, using $O(n)$ space, where n is the number of leaves in S. We present a simpler suffix tree oracle with amortized $O(\log \sigma)$ bound.

▶ **Lemma 4.** *For a suffix tree with n leaves, there is a suffix tree oracle of size $O(n)$ which answers each query in amortized $O(\log \sigma)$ time. It takes amortized $O(\log \sigma)$ time to update this suffix tree oracle, per insertion of a new leaf or a new suffix link to the suffix tree.*

Proof. (Sketch) We follow the approach by Fischer and Gawrychowski [10, 9]. The $\log\log n$ term in the running time of their suffix tree oracle is due to the fringe nearest marked ancestor data structure by Breslauer and Italiano [5], which answers each NMA query in a special case in worst case $O(\log\log n)$ time. It is possible to replace the fringe nearest marked ancestor data structures with the NMA data structures of Westbrook [16], so the time cost for each NMA query is amortized to $O(1)$. The other $(\log\log\sigma)^2/\log\log\log\sigma$ term is due to fast predecessor data structures for integer alphabets. Since our alphabet is more general, we use balanced search trees with $O(\log \sigma)$-time operations. Hence our bound is $O(\log \sigma)$ amortized. A complete proof can be found in a full version of this paper [12]. ◀

To overcome Difficulty C, we employ lazy maintenance for leaves, namely, we maintain only the *first character* of the label of every edge leading to a leaf. On the other hand, we eagerly maintain the whole label of every edge leading to an internal explicit node.

▶ **Lemma 5.** *The lazy representation of the in-coming edges of leaves allows for updating the suffix tree in amortized $O(\log \sigma)$ time per insertion of a new leaf.*

Proof. Let $U[i] = (k, a)$ and $\mathcal{T} = \mathcal{T}_{U[1..i]}$ as previously. Let xa be a Type-2 suffix of the extended text $T_k a$ that will be inserted to the suffix tree. Using the suffix tree oracle of Lemma 4, we obtain the canonical reference (v, c, ℓ) to x from which a new leaf for the suffix xa is to be inserted.

The difficult case is when x is on the edge e from v to a leaf and $\ell \geq 2$, since we only know the first character c of the label of e. We create a new internal node x on e, and create a new leaf as a child of x and its in-coming edge labeled with the first character a. We can determine the label of the in-coming edge of the new internal explicit node x as follows. Let y be the node of $LPT(\mathcal{T})$ which corresponds to the node $[v]_{\mathcal{T}}$ of $DAWG(\mathcal{T})$,

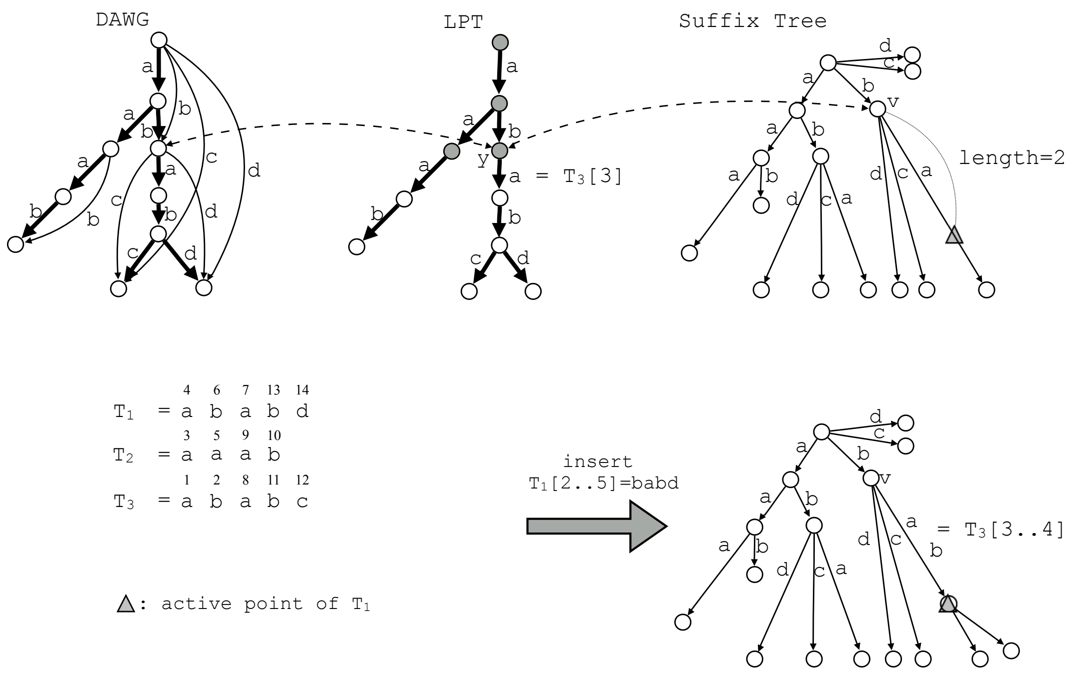

Figure 3 Illustration of how to determine the label of the in-coming edge of a new internal explicit node which is created on an edge leading to an existing leaf. Let $\mathcal{T}' = \{T_1 = \mathtt{abab}, T_2 = \mathtt{aaab}, T_3 = \mathtt{ababc}\}$, and $\mathcal{T} = \{T_1\mathtt{d}, T_2, T_3\}$. Now we are inserting a new leaf w.r.t. Type-2 suffix $\mathtt{babd}$ of $T_1\mathtt{d}$. The canonical reference to the insertion point of this suffix is $(\mathtt{b}, \mathtt{a}, 2)$, and hence we create a new internal node on the middle of the out-going edge of node $\mathtt{b}$ whose edge label begins with $\mathtt{a}$. Now, since $long([\mathtt{b}]_{\mathcal{T}}) = \mathtt{ab}$, we access the LPT node $y = \mathtt{ab}$. Since the label $\mathtt{a}$ of the out-going edge of y in $LPT(\mathcal{T})$ is now represented by pair $\langle 3, 3\rangle$, we can label the new suffix tree edge leading to the new internal node by $\langle 3, 3, 3+2-1\rangle = \langle 3, 3, 4\rangle$.

namely $y = long([v]_{\mathcal{T}})$. We represent the label of each edge of $LPT(\mathcal{T})$ by a pair of the text id and the position of the character in the text of that id. Let $\langle h, j\rangle$ be the label of the out-going edge of node y of $LPT(\mathcal{T})$ such that $T_h[j] = c$. Since we insert the Type-2 suffixes of $T_k a$ in increasing order of length, the path in $LPT(\mathcal{T})$ of length ℓ starting with this edge from y is non-branching. Thus, we can label the in-coming edge of the suffix tree by triple $\langle h, j, j+\ell-1\rangle$. See also Fig. 3.

While updating $DAWG(\mathcal{T}')$ to $DAWG(\mathcal{T})$, we have visited the node $[x]_{\mathcal{T}}$. We can obtain node y on $LPT(\mathcal{T})$ by an NMA query from node $long([x]_{\mathcal{T}})$, and associate to y each Type-2 suffix xa of $T_k a$ whose length is in range $[s+1, l+1]$, where s and l are the lengths of the shortest and longest members of $[x]_{\mathcal{T}}$, respectively. As we insert the Type-2 suffixes of $T_k a$ to the suffix tree in increasing order of length, for each Type-2 suffix xa the time cost to access its corresponding node y on $LPT(\mathcal{T})$ is $O(\log\sigma)$ amortized. It takes amortized $O(\log\sigma)$ time to query the suffix tree oracle by Lemma 4. All the other operations take $O(1)$ time each. ◀

Assume we are searching a growing text collection $\mathcal{T}$ for a given pattern P. If we stuck on the parent node u of a leaf in $STree(\mathcal{T})$ due to our lazy leaf representation, then we can move to the DAWG node which corresponds to the parent node u via $LPT(\mathcal{T})$, and continue searching for P on $DAWG(\mathcal{T})$. This way we can find the locus of P on $STree(\mathcal{T})$ in optimal $O(M\log\sigma)$ time, where $M = |P|$. Also, since the tree topology is correctly maintained with our lazy leaf representation, semi-dynamic NMA [16], LCA [7], and LA [1] queries can be correctly supported in $O(1)$ time on our suffix tree representation.

Finally, we obtain the main result of this section.

▶ **Theorem 6.** *Given a fully-online sequence U of N update operators for a collection of K texts, we can update $STree(\mathcal{T}_{U[1..i]})$ for $i = 1, \ldots, N$ in a total of $O(N \log \sigma)$ time and $O(N)$ space.*

After the whole U has been processed, we determine the triples representing the entire labels of the in-coming edges of all leaves of $STree(\mathcal{T}_U)$ in a total of $O(N)$ time. We can then discard $DAWG(\mathcal{T}_U)$ and $LPT(\mathcal{T}_U)$.

6 Conclusions and open problems

The main contribution of this paper is an $O(N \log \sigma)$-time algorithm to maintain the suffix tree for a text collection in the left-to-right fully-online setting, where N and σ are the total text length and the alphabet size, respectively. The key was a non-trivial use of the DAWG.

There are interesting open problems for the left-to-right fully-online suffix tree construction:

1. Is it possible to efficiently maintain *complete* labels of the edges leading to the leaves?
2. Our bound is amortized, namely, for each new character our algorithm takes $O(\log \sigma)$ amortized time. Is it possible to de-amortize it, e.g. by using techniques in [5, 9, 10]?

References

1 Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *ICALP 2000*, pages 73–84, 2000.

2 Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS 2002*, pages 1–16, 2002.

3 Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel Seiferas. The smallest automaton recognizing the subwords of a text. *TCS*, 40:31–55, 1985.

4 Anselm Blumer, Janet Blumer, David Haussler, Ross Mcconnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987.

5 Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013.

6 Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), 2007.

7 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. In *SODA 1999*, pages 235–244, 1999.

8 Paolo Ferragina and Roberto Grossi. Improved dynamic text indexing. *J. Algorithms*, 31(2):291–319, 1999.

9 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. *CoRR*, abs/1302.3347, 2013.

10 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *CPM 2015*, pages 160–171, 2015.

11 Eamonn J. Keogh, Stefano Lonardi, and Bill Yuan-chi Chiu. Finding surprising patterns in a time series database in linear time and space. In *KDD 2002*, pages 550–556, 2002.

12 Takuya Takagi, Shunsuke Inenaga, and Hiroki Arimura. Fully-online construction of suffix trees and dawgs for multiple texts. *CoRR*, abs/1507.07622, 2015.

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

14 Yilun Wang, Yu Zheng, and Yexiang Xue. Travel time estimation of a path using sparse trajectories. In *KDD 2014*, pages 25–34, 2014.

15 Peter Weiner. Linear pattern-matching algorithms. In *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*, pages 1–11, 1973.

16 Jeffery Westbrook. Fast incremental planarity testing. In *ICALP 1992*, pages 342–353, 1992.