Smallest Suffixient Set Maintenance in Near-Real-Time

Dominik Köppl
University of Yamanashi, Kofu, Japan

Gregory Kucherov
LIGM, CNRS and Gustave Eiffel University, Marne-la-Vallée, France

Abstract

The size of the *smallest suffixient set* of positions of a string recently emerged as a new measure of string *repetitiveness* – a measure reflecting how much of repetitive content the string contains. We study how to maintain the smallest suffixient set online in near-real-time, that is with small (in our case, polyloglog) worst-case time for processing each letter. Two frameworks are considered: when the text is given letter-by-letter in either right-to-left or left-to-right order. Our central algorithmic tool is Weiner's suffix tree algorithm and associated algorithmic primitives for its efficient implementation.

2012 ACM Subject Classification Information systems → Information retrieval

Keywords and phrases online algorithms, string algorithms, suffix tree, real-time computation, smallest suffixient set, string attractor

Digital Object Identifier 10.4230/LIPIcs.MFCS.2026.86

Related Version *Full Version*: https://arxiv.org/abs/2604.27548

Funding *Dominik Köppl*: This work was supported in part by JSPS KAKENHI Grant Number 25K21150. We thank Fédération de Recherche Bézout for supporting the research visit of the first author to LIGM, where this work was initiated.

1 Introduction

Nowadays, it is not uncommon to deal with highly repetitive data in the sense that many data fragments are not locally consecutively repeated but rather spread across the entire data. Examples include genomic sequences, files of version control systems or archives, sensor data, and many others. This has motivated recent work on how indexing data structures can exploit this redundancy [19, 20]. A formal framework for such studies requires a definition of a measure of redundancy (repetitiveness) of data, and there are many ways to define such a measure [19]. These measures can be classified into two categories: those based on compressed representations of the data and those based on "extrinsic" properties of the data, unrelated to any compressed representation. For sequential data, the first category can be exemplified by dictionary-based compression algorithms, such as Lempel–Ziv or grammar-based compression. As for the second category, several new measures have been defined, such as *smallest string attractor* [13] or *normalized substring complexity* [22, 16].

A string attractor is a set of positions in the string such that every *distinct* substring has an occurrence that covers one of those positions. The size of the smallest attractor γ of a string of length n is a lower bound on the number of phrases of the Lempel–Ziv parsing or the size of a grammar producing this string, but, on the other hand, those can be bounded as $O(\gamma \log \frac{n}{\gamma})$, which shows that γ is a meaningful compressibility estimator. It has also been shown that space $O(\gamma \log \frac{n}{\gamma})$ is sufficient for a data structure capable of supporting string matching in a time linear in the pattern length [6].

Unfortunately, computing the smallest attractor is an NP-complete problem [13]. Recently, a related measure was proposed: the size of the *smallest suffixient set*, denoted χ [8]. The smallest suffixient set is an attractor that is not necessarily a smallest string attractor [8]. In contrast to γ, χ can be computed in linear time [4]. Testing whether a given set of positions is suffixient can also be done in linear time [5]. Several properties of χ have been established recently. These include relationships of χ with other repetitiveness measures and how it can be affected by transformations of the input string, such as edits or reversal [21]. Very recently, it has been shown that a string can be encoded into a lossless representation of size $O(\chi)$ [23].

In this paper, we study how to compute χ in the *online* mode in *near-real-time*, that is with small worst-case time guarantees per letter. An online algorithm for computing χ has already been proposed by Navarro et al. [10], which, however, provides only *amortized* time bounds, whereas our primary interest here is in worst-case bounds. In this work, we design algorithms for maintaining χ in $O(\log^2 \log n)$ worst-case time per letter for arbitrary integer polynomial-size alphabet. The time bound is reduced to $O(\log \log n)$ in the case of log-size alphabet.

Since the definition of suffixient set is "oriented", i.e. is not invariant under string reversal (as opposed, e.g., to string attractors), for online algorithms it is relevant whether the string is input left-to-right or right-to-left. Both cases were considered in [10]. For the left-to-right input, we propose a different computation based on Weiner's suffix tree algorithm, following the techniques that we recently applied to some other problems on strings [14]. For the right-to-left input, we propose a version of the algorithm of [10] admitting an efficient near-real-time implementation.

Since Weiner's algorithm and its efficient implementations are central to our work, Section 3 below is devoted to a short presentation of this algorithm, its improvements and associated algorithmic primitives. Our main results are then presented in Section 4.

2 Supermaximal extensions and suffixient sets

We consider an integer alphabet Σ and assume $\sigma = |\Sigma| = n^{O(1)}$ where n is the input size. Consider a text $w \in \Sigma^*$ and let $\mathcal{F}_w$ be the set of substrings of w. A substring u is called *right-maximal* if for two distinct letters $x \neq y$, we have $ux, uy \in \mathcal{F}_w$. Given a right-maximal substring u, a substring $ux \in \mathcal{F}_w$ is called a *right extension*. In what follows it will be convenient for us to represent a right extension as a pair (u, x) where $u \in \Sigma^*$ and $x \in \Sigma$. Let $E_r(w)$ denote the set of all right extensions of w.

Consider $E_r(w)$ and consider its minimal subset $S_r(w) \subseteq E_r(w)$ such that for every $(u, x) \in E_r(w)$, there exists $(v, x) \in S_r(w)$ such that u is a suffix of v. Due to minimality, for every pair $(u, x), (v, x) \in S_r(w)$ with $u \neq v$, neither u is a suffix of v nor v is a suffix of u. The subset $S_r(w)$ is uniquely defined and contains all those right extensions (u, x) for which (zu, x) is not a valid right extension for any letter $z \in \Sigma$. The latter means that either $zux \notin \mathcal{F}_w$ or $zux \in \mathcal{F}_w$ but zu is not right-maximal, i.e. $zuy \notin \mathcal{F}_w$ for any other letter $y \neq x$.

Elements of $S_r(w)$ are called *supermaximal (right-)extensions* (hereafter, SREs) [3, 21]. By definition, every right extension of w is a suffix of a right extension of $S_r(w)$. Since $S_r(w)$ is uniquely defined, it is also the smallest set of right extensions with this property.

Each SRE $(u, x) \in S_r(w)$ can be mapped to the end position of an arbitrary occurrence of ux in w. Distinct supermaximal extensions are necessarily mapped to distinct positions [4, Lemma 1]. Therefore, any such mapping defines a one-to-one correspondence between SREs and a certain subset of positions of w. Such a set of positions is called a *smallest suffixient*

set (hereafter, SSS), i.e. a smallest set of positions such that every right extension (u, x) has an occurrence of ux ending at one of those positions. Consequently, the set of SREs (based on substring distinctness) is uniquely defined whereas an SSS (based on the occurrences of SREs) is not.

▶ **Example 1.** For the string $w = \texttt{aabaababa\$}$, $S_r(w)$ consists of three SREs: $(\texttt{aaba}, \texttt{a})$, $(\texttt{aaba}, \texttt{b})$ and $(\texttt{aba}, \texttt{\$})$. Its unique SSS consists of positions shown underlined in the string: $\texttt{aab}\underline{\texttt{a}}\texttt{b}\underline{\texttt{a}}\texttt{b}\underline{\texttt{a}}\texttt{\$}$. For $v = \texttt{aabaabaab\$}$, $S_r(v) = \{(\texttt{a}, \texttt{a}), (\texttt{a}, \texttt{b}), (\texttt{aabaab}, \texttt{a}), (\texttt{aabaab}, \texttt{\$})\}$. It has several (nine) possible SSSs, such as $\texttt{a}\underline{\texttt{a}}\texttt{b}\underline{\texttt{aa}}\texttt{b}\underline{\texttt{aab}}\texttt{\$}$, $\texttt{a}\underline{\texttt{a}}\texttt{b}\underline{\texttt{aa}}\texttt{b}\underline{\texttt{aab}}\texttt{\$}$, etc.

The size of an SSS, or equivalently, the size of $S_r(w)$, denoted χ, is a measure of repetitiveness [19] of a string, as presented in the introduction. Whereas χ is incomparable with common "copy-paste measures" such as measures based on Lempel–Ziv compression or context-free grammar representation, in the sense that it can happen to be asymptotically smaller or asymptotically larger than those [19], it is still a meaningful measure of repetitiveness. In particular, we have $\gamma \le \chi$ on the one hand, and $\chi \le 2r$ on the other hand, where r is the repetitiveness measure based on Burrows–Wheeler transform [21].

For example, some remarkable words studied in word combinatorics, such as Fibonacci words, have $\chi = O(1)$, which reflects their high repetitiveness [21]. On the other hand, for a de Bruijn sequence of length n, we have $\chi = \Theta(n)$, which shows a difference with copy-paste measures, which are all bounded by $O(n/\log n)$.

3 Algorithmic tools

3.1 Weiner's suffix tree algorithm: an outline

We assume the reader to be familiar with suffix trees, see e.g. [12]. Consider the suffix tree for a given text w. Edges are labeled by substrings of w, which we call *edge labels*. We call an edge an x-edge, for $x \in \Sigma$, if its edge label starts with x. Each substring $u \in \mathcal{F}_w$ corresponds to a *locus* in the suffix tree, denoted $\mathsf{locus}(u)$, which can be found by spelling u starting from the root of the tree. $\mathsf{locus}(u)$ is either an internal node or a leaf of the tree (hereafter called simply a *node*), or an *implicit node* specified by an offset from the parent node of some edge. The substring whose locus is a node α is called its *label* and is denoted $\mathsf{label}(\alpha)$.

Weiner's algorithm constructs the suffix tree for a text in a "reversed online" fashion, i.e. by processing the text right-to-left and after reading a letter of the text, updating the suffix tree by inserting a new suffix of the text. We will rely on Weiner's algorithm in this work. Here we focus only on the features of Weiner's algorithm relevant to the present work, for its full description we refer the reader to [1, 14].

Weiner's algorithm maintains a suffix tree augmented with Weiner's links, or *W-links* for short. W-links are pointers indexed by alphabet letters: a node α has a defined W-link by a letter $x \in \Sigma$ if $x \cdot \mathsf{label}(\alpha)$ is a substring of w. We call it an x-link and define $W_x(\alpha)$ to be the destination of the x-link from α. W-links are used by Weiner's algorithm to jump from α to $\gamma = \mathsf{locus}(x \cdot \mathsf{label}(\alpha))$. If γ is a node of the tree, then $W_x(\alpha) = \gamma$ and the link is called *hard*. If γ is an implicit node, the link is called *soft* and $W_x(\alpha)$ is defined to be the closest descendant node[1] of γ. Figure 1 shows the suffix trees for the two strings from Example 1. It also shows which W-links are defined for each node and the loci of the SREs.

[1] This definition of soft links does not match the original paper [24] and has been introduced later in work [1] that we comment on in the next section. We use this definition in our paper as well.

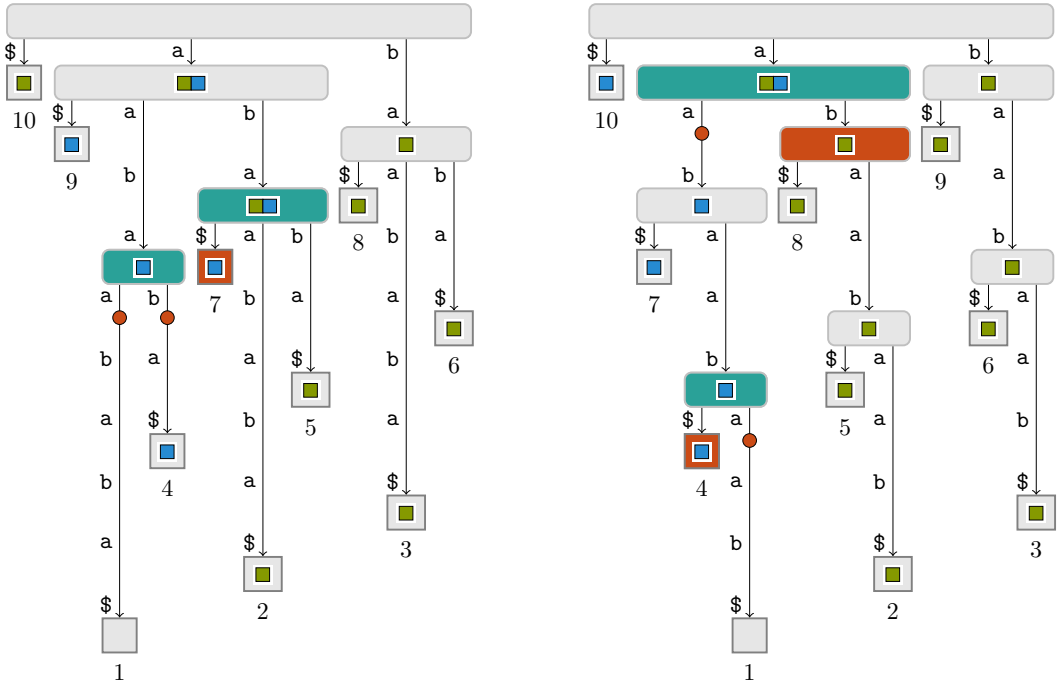

Figure 1 Suffix trees of the strings in Example 1. The locus ux for each SRE (u, x) marked by an orange dot (■) can be either a node or an implicit node; the node with label u is further filled in cyan (■). Leaves are labeled below by the starting position of the corresponding suffix in the string, e.g. the leaf labeled by 1 corresponds to the entire string. Nodes are augmented with the colored boxes similar to Fig. 3 to visualize the presence of an a-link with green (■) and a b-link with blue (■). **Left:** w has only one SSS since all orange marked loci are leaves or on edges connecting to leaves. This SSS is depicted on bottom by underlining the letters at the end positions of the SREs. **Right:** Since aa and ab each have three occurrences in v (by counting the number of leaves in the respective subtrees), v has nine SSSs. They are depicted separately in Figure 2. Each SSS can be obtained by picking, for each SRE (u, x), a suffix number of a leaf of the subtree of its locus (marked in orange) incremented with $|u|$.

Consider a string w ending with a unique sentinel symbol \$. The following lemma summarizes properties of W-links that will be useful to us.

▶ **Lemma 2.** *The following statements hold.*

(i) *Every node has at least one defined W-link except for the leaf node labeled by the entire string w.*

(ii) *For any $x \in \Sigma$, if an x-link is defined for a node ν, then it is defined for any ancestor of ν.*

(iii) *If a node ν has a defined soft x-link, there exists a descendant node of ν with a defined hard x-link.*

Proof.

(i) The label of every internal node has at least two occurrences in w and therefore at least one preceded by some letter. Every leaf is labeled by a suffix that is preceded by some letter as well, with the only exception of the leaf labeled by the entire string.

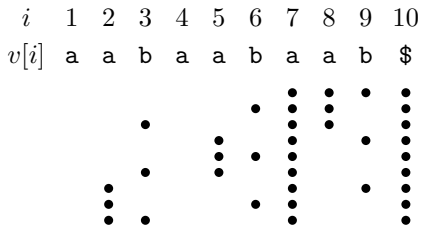

Figure 2 All SSSs of $v = \texttt{aabaabaab\$}$ depicted by rows of dots. By Figure 1, the positions 7 and 10 are present in all SSS since their loci are leaves or on edges leading to leaves (leaves with suffix numbers 1 and 4 in the figure).

(ii) The claim follows from the observation that if a substring has an occurrence preceded by a letter x, then this holds for any of its prefixes as well.

(iii) Due to the unique terminal symbol \$, the locus of any suffix of w is a leaf, and therefore any leaf except for the one labeled by the entire string w has a single hard W-link pointing to another leaf. Assume now an internal node ν has a soft x-link. Its label $u = \mathsf{label}(\nu)$ must extend to a suffix preceded by x. The locus of this suffix is a leaf which is a descendant of ν and has a defined hard x-link. ◀

We now outline a round of Weiner's algorithm transforming the suffix tree for a text w to the one for xw for some $x \in \Sigma$. We refer to Figure 3 in our description. The algorithm maintains the leaf λ whose label is the entire string w. A key step of the algorithm is to locate the node α that is the lowest ancestor of λ with a defined x-link. That is, $\mathsf{label}(\alpha)$ is the longest prefix of w that has another occurrence in w preceded by x. In the original version [24], this is done by a straightforward upward traversal of the ancestors of λ until finding a node with a defined x-link. Although on an individual letter this work can take time up to $O(n)$, it takes $O(1)$ time per letter *amortized* over the entire run of the algorithm.

The goal of locating α is to use its x-link in order to locate the node $\gamma = W_x(\alpha)$ to which a new leaf λ' should be attached, such that λ' has the string label xw. Here two cases occur, depending on whether α's x-link is hard or soft, and, respectively, whether (a) node γ is an existing node or (b) an implicit one (i.e., a locus on an edge). In the latter case (b), γ is created by "splitting" the corresponding edge (see Figure 3). In both cases (a) and (b), a new leaf of γ is created labeled by xw and the new edge is a y-edge where y is the letter following the prefix occurrence of $\mathsf{label}(\alpha)$ in w.

On top of this, we have to update certain W-links. We create a *hard* x-link at λ and, if necessary, harden the x-link of α, spending constant time. Furthermore, we have to set or modify a non-constant number of *soft* W-links that are divided into two groups. One group consists of $O(\sigma)$ W-links that should be assigned to the newly created node γ by copying the W-links of its child node β. The other group consists of new x-links at nodes on the path between λ and α and modified existing x-links at nodes on the path between α and δ.

3.2 Fringe colored ancestor problem

In their seminal paper [1], Breslauer and Italiano proposed an implementation of Weiner's algorithm working in *near-real-time*, that is providing a small (double logarithmic) worst-case time bound on an individual round. Their main idea is to speed up the retrieval of α from λ by answering a query called FRINGE MARKED ANCESTOR on dynamic trees. By Lemma 2(ii), nodes with a defined x-link, for a given $x \in \Sigma$, form a connected subtree of the suffix tree

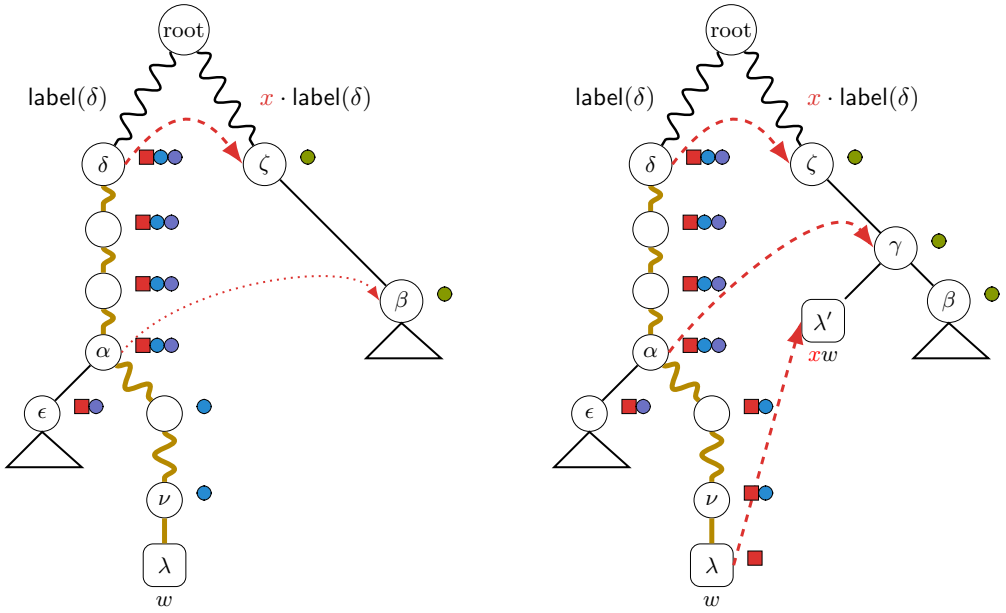

Figure 3 One round of Weiner's suffix tree construction algorithm: updating $\mathsf{ST}(w)$ (left) to $\mathsf{ST}(xw)$ (right) by inserting the new suffix xw. W-links are visualized by colored boxes and circles next to nodes. A red box (■) at a node represents an x-link. Blue (●), violet (●) and green (●) circles represent *sets* of W-links that can, in general, be growing towards the root. Note also that blue and violet sets are not necessarily disjoint. Explicitly drawn are some x-links: Dashed red arrows represent hard x-links, dotted red arrows represent soft x-links. Curly edges represent paths that may contain multiple nodes. Node δ is the lowest ancestor of λ with a *hard* x-link, and node α is the lowest ancestor of λ with a (not necessarily hard) x-link. α's x-link can be soft (as in the figure) or hard (in case $\alpha = \delta$). If this link is soft, the algorithm creates a new node γ that is the insertion point for the leaf λ'. If this link is hard, the insertion point is $W_x(\alpha)$. After creating γ, the W-links on the golden thick curly path from λ to δ need to be updated.

sharing the same root; the leaves of this subtree are called the *fringe*. Retrieving α amounts to computing the lowest ancestor of the fringe. We define a more general version of this problem where we simultaneously store information about W-links for all letters (expressed by *colors*[2]), as opposed to a separate data structure for each letter (as in [1]). We call this problem FRINGE COLORED ANCESTOR and define it as follows.

FRINGE COLORED ANCESTOR. Consider a set of colors $\mathcal{C}$ and a dynamic tree initially consisting of a single uncolored node. Maintain the tree under the following operations:[3]

- insert a new uncolored child of an existing node,
- assign a color $x \in \mathcal{C}$ to a node ν provided that either ν is the tree root or the parent of ν is colored with x,
- introduce a new node by splitting an edge and coloring this node by the colors assigned to its descendant,
- for a given node and a given color $x \in \mathcal{C}$, find its lowest ancestor colored with x.

[2] We introduce the term *colors* here as a means of abstraction since they will also play another role, besides being synonymous to letters, later in Section 4.2.2.

[3] The set of operations can be larger and include deletion and uncoloring of a node; here we only consider operations that we need for our purposes.

In particular, FRINGE MARKED ANCESTOR is FRINGE COLORED ANCESTOR with a single color. The updates preserve the *fringe property*, i.e., ancestors of a node colored by x are colored by x as well. The fringe property is important for modelling W-links since they also obey the fringe property by Lemma 2(ii).

Generalizing [1], FRINGE COLORED ANCESTOR can be solved by reducing it to COLORED PREDECESSOR in dynamic lists:

COLORED PREDECESSOR. Maintain a dynamic linked list $\mathcal{L}$ where each element is assigned a *color* from a set C of possible colors. Possible updates to $\mathcal{L}$ are insertions of an element after a given element. A colored predecessor query $\mathcal{L}.\mathsf{pred}(\nu, x)$, for an element $\nu \in \mathcal{L}$ of the list and a color $x \in C$, asks for the closest preceding element of ν in $\mathcal{L}$ colored by x. The colored successor $L.\mathsf{succ}(\nu, x)$ is defined symmetrically.

To explain the reduction from FRINGE COLORED ANCESTOR to COLORED PREDECESSOR, we need another operation: the *lowest common ancestor* of two nodes μ and ν of a tree, denoted $\mathsf{lca}(\mu, \nu)$. It is known that lowest common ancestor queries can be answered in constant time even on a dynamic tree undergoing insertion and deletion of nodes [7].

We further consider that a node colored by $x \in C$ can be either *explicitly x-colored* or *implicitly x-colored*, but we require that a node is implicitly x-colored if and only if at least one of its (not necessarily immediate) descendants is explicitly x-colored. We call a node *x-colored* if it is either explicitly or implicitly x-colored. Observe that x-colored nodes fulfill the fringe property and each x-colored node that has no x-colored descendants must be explicitly x-colored.

Consider the linked list $\mathcal{L}$ storing the *Euler tour* of the current tree, where each node of the tree has two copies in the list corresponding to its first and last visits. Thus, each element of the list is a node ν, which we augment by the set of the *explicit colorings* of ν. Then the following holds – see also Figure 4 for an illustration.

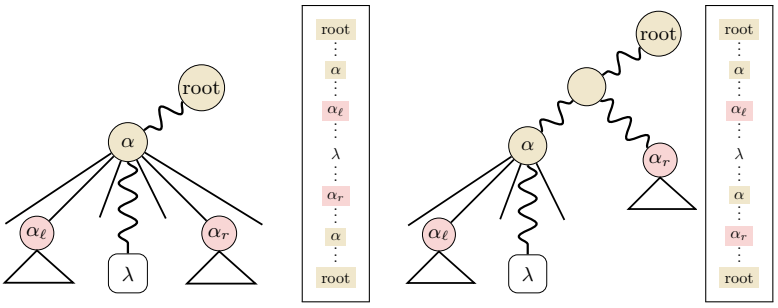

Figure 4 Illustration to Lemma 3. Explicitly colored nodes are shown in red ( ) and implicitly colored nodes are shown in yellow ( ). The Euler tour list $\mathcal{L}$ is shown next to the tree in top-down order. The tree on the left shows the case when λ is between two descendants α_ℓ and α_r of α in the Euler tour, and the tree on the right shows the case when all successors of λ inside the subtree rooted at α are not explicitly colored (the case that all predecessors of λ inside the subtree rooted at α are not explicitly colored is symmetric).

▶ **Lemma 3** ([15, Lemma 5]). *Let λ be a node that is not x-colored and let α be the lowest x-colored ancestor of λ. Consider the Euler tour list $\mathcal{L}$ in which only explicit colorings are represented, and let $\alpha_\ell = \mathcal{L}.\mathsf{pred}(\lambda, x)$ and $\alpha_r = \mathcal{L}.\mathsf{succ}(\lambda, x)$. Then α is the lowest node between $\mathsf{lca}(\alpha_\ell, \lambda)$ and $\mathsf{lca}(\lambda, \alpha_r)$.*

Proof. Observe that since λ is not x-colored, no descendant of λ is x-colored either, and therefore successor and predecessor operations are unambiguously defined. Since α is an x-colored ancestor of λ, it remains to show that it is the lowest one. This is immediate by

contradiction: if there was a lower x-colored ancestor, its first occurrence in the Euler tour would have been after the occurrence of α_ℓ and its second occurrence would have been earlier than the occurrence of α_r. ◀

Lemma 3 allows us to answer fringe colored ancestor queries by dealing only with a subset of colored nodes – explicitly colored ones – as long as the requirement on explicit coloring is satisfied. Lemma 3 is naturally applied to the suffix tree if we define explicitly and implicitly x-colored nodes to be nodes with a hard x-link and a soft x-link, respectively. Lemma 2(iii) ensures that the definition of implicit coloring is verified. It follows that to locate α, soft W-links are not needed, and corresponding colorings can be omitted in the Euler tour. This observation was used in [15] to avoid storing soft links altogether and retrieve α based on hard links only, even if the x-link of α is soft.

Moreover, the authors of [15, Thm. 1] showed that hard W-links are sufficient not only to locate α but also to recover the x-link $W_x(\alpha)$ needed to locate the insertion point γ (Section 3.1). This is possible due to a stronger version of Lemma 2(iii): not only a node ν with a soft x-link has a descendant with a hard x-link, but the unique closest such descendant has a hard x-link pointing to the same node as $W_x(\nu)$ (see also [9, Observation 15(c)]). As a consequence, maintaining soft W-links can be avoided altogether, which greatly simplifies the implementation of Weiner's algorithm. In particular, the work of setting or updating soft W-links (see Section 3.1) vanishes. We stick to this implementation in our work.

3.3 Colored predecessor queries on lists

As follows from the above, COLORED PREDECESSOR is the main tool for implementing Weiner's algorithm with worst-case guarantees. A data structure for solving this problem has been presented by Mortensen [18].

▶ **Theorem 4** ([18] Theorem 15). *For COLORED PREDECESSOR, there is an $O(n)$-space data structure supporting both updates and queries in worst-case time $O(\log^2 \log n)$ where n is the size of the list.*

The bound of Theorem 4 can be improved if the color space is log-size.

▶ **Theorem 5** ([11], Theorem 5.2). *Assuming $|C| = O(\log^{1/4} n)$, COLORED PREDECESSOR can be solved in worst-case time $O(\log \log n)$ for both updates and queries, using an $O(n)$-space data structure.*

We remark that the result of [11] is slightly more general in that it allows querying a subset of colors and not just a single color. On the other hand, in [11], the $O(\log \log n)$ time bound for insertions and deletions is only amortized. In our case, we do not use deletions, and the insertion operation can be deamortized using standard techniques [15]. A similar result is proved in [17, Theorem 4.1] without providing a specific exponent in the bound on the color space size.

Before concluding with the final complexity bound on a round of Weiner's algorithm, note that it also includes parent-to-child access in the suffix tree, however the complexity of this step is subsumed by COLORED PREDECESSOR (see [14] for more details).

▶ **Theorem 6.** *When the string is processed online right-to-left, the suffix tree can be updated in $O(\log^2 \log n)$ worst-case time per letter, for an integer alphabet. This bound can be reduced to $O(\log \log n)$ for an alphabet size $\sigma = O(\log^{1/4} n)$.*

4 Near-real-time computation of smallest suffixient set

Based on efficient implementations of Weiner's algorithm, we now present our near-real-time solutions to the problem of maintaining the set $S_r(w)$ of supermaximal right-extensions (SREs) and of a smallest suffixient set of positions (SSS). We consider two frameworks: when the text is provided online right-to-left and when it is provided left-to-right. Both frameworks have been previously considered in [10, Section 7] where the authors described online $O(n)$-time algorithms for both. Those algorithms, however, provide no worst-case time guarantee per individual letter, which is our primary goal in this work.

4.1 Right-to-left framework

In this section, we assume that the text is processed right-to-left, starting from the unique terminal letter \$. Weiner's algorithm maintains the suffix tree online in this framework. Below we will show how Weiner's algorithm can be modified to maintain the set $S_r(w)$ and an SSS.

First, let us clarify how SREs are identified in Weiner's suffix tree. A SRE (u, x) corresponds to a node $\mathsf{locus}(u)$ with a descending x-edge (i.e. an edge whose label starts with x). The following lemma characterizes those edges.

▶ **Lemma 7.** *Let $ux \in \mathcal{F}_w$ for $u \in \Sigma^*$ and $x \in \Sigma$. Then (u, x) is a SRE if and only if (i) $\mathsf{locus}(u)$ is a node β, (ii) one of β's descending edges is an x-edge, and (iii) for every hard W-link of β, say $W_z(\beta) = \nu$ for a $z \in \Sigma$, ν does not have a descending x-edge.*

Proof. By the definition of a SRE, u must be right-maximal and one of its occurrences is followed by x, which implies (i) and (ii). Furthermore, the definition requires that for every $z \in \Sigma$, either $W_z(\mathsf{locus}(u))$ does not have a descending x-edge, or $W_z(\mathsf{locus}(u))$ is an implicit node which means that $W_z(\mathsf{locus}(u))$ is soft – otherwise u is a suffix of a longer right-maximal repeat. That is, the excluded case is when $W_z(\mathsf{locus}(u))$ is hard and has a descending x-edge; (iii) follows. ◀

We now analyse how the set $S_r(w)$ of SREs can be modified when a letter $x \in \Sigma$ is prepended to w. We analyse this in terms of a round of Weiner's algorithm described in Section 3.1 and illustrated in Figure 3. Recall that $\mathsf{label}(\alpha)$ is the longest prefix of w that has an occurrence in w preceded by x, and then $\mathsf{label}(\gamma) = x\,\mathsf{label}(\alpha)$ is the longest prefix of xw that has another occurrence in xw. Note that $\mathsf{label}(\alpha)$ is right-maximal in w and $\mathsf{label}(\gamma)$ is right-maximal in xw. Let $y \in \Sigma$ such that w starts with $\mathsf{label}(\alpha) \cdot y$ (and therefore xw starts with $\mathsf{label}(\gamma) \cdot y$).

We now use Lemma 7 to determine the changes in $S_r(w)$ caused by a round of Weiner's algorithm. Based on Lemma 7, we identify a SRE (u, x) with the pair $(\mathsf{locus}(u), x)$. By definition, $\mathsf{locus}(u)$ is a node that has a descending x-edge. The following lemma specifies the modifications depending on whether $W_x(\alpha)$ is hard or soft.

▶ **Lemma 8.**

(a) *If $W_x(\alpha)$ is hard, i.e. γ is an existing node, then (γ, y) is a new SRE of xw that has to be added when transforming $S_r(w)$ to $S_r(xw)$. Furthermore, if (α, y) was a SRE in $S_r(w)$, by Lemma 7 it is no longer a SRE in $S_r(xw)$ and should be omitted.*

(b) *If $W_x(\alpha)$ is soft, then (γ, y) and (γ, z) are two new SREs of xw associated with the new node γ and its two descending edges (a y-edge and a z-edge). Furthermore, if any of (α, y) or (α, z) is a SRE in $S_r(w)$, it is not a member of $S_r(xw)$.*

Proof.

(a) First observe that any hard W-link of γ (if there is any) points to a node without descending y-edge. Indeed, by definition of α, $\mathsf{label}(\gamma) = x \cdot \mathsf{label}(\alpha)$ does not have right extension y before the round, and the only occurrence of $\mathsf{label}(\gamma) \cdot y$ after the round is a prefix occurrence. Therefore, for any $z \in \Sigma$, $z \cdot \mathsf{label}(\gamma) \cdot y$ does not occur in xw which means that $W_z(\gamma)$ cannot have a descending y-edge.
From the above, by Lemma 7, a new descending y-edge created for γ defines a new SRE (γ, y). On the other hand, if (α, y) was a SRE, it is no longer one because $\gamma = W_x(\alpha)$ has now a descending y-edge.

(b) Recall that the y-edge of γ connects γ to the newly created leaf (λ' in Figure 3) and then the z-edge connects γ to the previously existing descendant (β in Figure 3). In terms of the text w, y is the letter following the prefix occurrence of $\mathsf{label}(\gamma)$ in xw and z is the unique letter following its other occurrences. Also recall that γ inherits the W-links of its already existing child (β in the figure), but all these W-links become soft. Indeed, if $W_t(\gamma)$ were hard for some $t \in \Sigma$, then $t \cdot \mathsf{label}(\gamma)$ would be the label of either an internal node or a leaf before the round. In the former case, $t \cdot \mathsf{label}(\gamma)$ would have two distinct right extensions, and hence $\mathsf{label}(\gamma)$ would be right-maximal. In the latter case, $\mathsf{label}(\gamma)$ would be a suffix of the current text. In both cases, γ would already be explicit before the split, a contradiction. Hence, (γ, y) and (γ, z) are new SREs. Furthermore, similar to the previous case, if any of (α, y) and (α, z) was a SRE in $S_r(w)$, it is no longer a SRE in xw, i.e. it is not a member of $S_r(xw)$. ◀

Thus, a round of Weiner's algorithm either introduces one new SRE and possibly deletes one, or introduces two new SREs and possibly deletes one or two.

We now turn to the implementation and show how to maintain the set of SREs and, at the same time, an SSS corresponding to this set. As in [10, before Theorem 3], it is convenient to think of positions as decrementing negative integers referring to the distance from the end of the string. At each node μ of the suffix tree, we store the *end* position of the rightmost occurrence of $\mathsf{label}(\mu)$. For leaves, this position will be 0. For internal nodes, we maintain these positions in the standard way: when a new internal node is created by splitting an edge (node γ in Fig. 3), the rightmost end position of $\mathsf{label}(\gamma)$ is computed relative to the descendant node (β), by subtracting the label length of the corresponding edge ($|\mathsf{label}(\beta)| - |\mathsf{label}(\gamma)|$).

Furthermore, each SRE (u, x) is uniquely associated with an edge of the suffix tree (see Lemma 7) and therefore can be associated with a unique rightmost end position of ux. The uniqueness is supported by the fact that no two SREs can end at the same position [4, Lemma 1]. We compute this rightmost end position of a SRE (u, x) from the position stored at the descendant node of the corresponding edge. We then associate the current set of SREs to an SSS consisting of the rightmost end positions of each SRE. To maintain this set, we use a left-growable bit vector storing 1 at each bit position corresponding to the rightmost end position of one of the current SREs. The growable vector can be implemented e.g. using the *resizable array* of [2]. When a SRE is inserted into or deleted from the current set, its rightmost end position is computed and the corresponding bit of the array is flipped. The size of the SSS is the number of 1's in the array.

We conclude with the final result of this section and summarize the algorithmic steps in Algorithm 1. In the pseudocode, InsertSRE and DeleteSRE update both the represented SRE set and the corresponding bit vector.

▶ **Theorem 9.** *In the right-to-left framework, the set of SREs and an SSS consisting of the rightmost positions of each SRE can be maintained online in worst-case $O(\log^2 \log n)$ time per letter. This bound can be reduced to $O(\log \log n)$ for alphabet size $\sigma = O(\log^{1/4} n)$.*

Algorithm 1 Right-to-left update of $S_r(w)$ and of the rightmost SSS.

Require: Current suffix tree $\mathsf{ST}(w)$ with the leaf λ labeled w, current SRE set $S = S_r(w)$, bit vector B, and new letter x
Ensure: Updated structures for xw

$\alpha \leftarrow$ lowest ancestor of λ with a defined x-link
$\ell \leftarrow W_x(\alpha)$ recovered from the hard-link representation
$y \leftarrow$ the letter following the prefix occurrence of $\mathsf{label}(\alpha)$ in w
if ℓ is a node **then**
    $\gamma \leftarrow \ell$ and create a new leaf λ' below γ labeled xw by a y-edge
    InsertSRE(γ, y) ▷ *flip the bit for the rightmost end position*
    if IsSRE(α, y) **then** DeleteSRE(α, y)
else
    split the edge at the implicit locus ℓ and let γ be the new node
    let β be the old child of γ and let z be the first letter of the edge from γ to β
    create a new leaf λ' below γ labeled xw by a y-edge
    InsertSRE(γ, y) and InsertSRE(γ, z)
    for $c \in \{y, z\}$ **do**
        if IsSRE(α, c) **then** DeleteSRE(α, c)
perform the ordinary hard-link updates of the Weiner round
$\lambda \leftarrow \lambda'$ and regard the current text as xw

4.2 Left-to-right framework

If we receive the letters of the text from left to right, we can use Ukkonen's algorithm to maintain the suffix tree online. This algorithm, in turn, can be used to maintain online the set of SREs and an SSS [10, Section 7.1]. Ukkonen's algorithm runs in linear time for constant-sized alphabets, however it provides no worst-case guarantee on the time spent on an individual letter. Here we apply the modified Weiner's algorithm from Section 3 to obtain an efficient near-real-time algorithm for this problem.

4.2.1 Modifications to the set of SREs

We will be maintaining the suffix tree $\mathsf{ST}(\overleftarrow{w})$ of the reversal $\overleftarrow{w}$ for an input text w received online from left to right. The reversal $\overleftarrow{w}$ of $w[1..n]$ is the string $w[n] \cdot w[n-1] \cdots w[1]$. Consequently, $\mathsf{ST}(\overleftarrow{w})$ indexes the reverse $\overleftarrow{u}$ of each substring $u \in \mathcal{F}_w$.

Analogous to Section 4.1, we first characterize how SREs of w are represented in $\mathsf{ST}(\overleftarrow{w})$.

▶ **Lemma 10.** *Let $ux \in \mathcal{F}_w$ for $u \in \Sigma^*$ and $x \in \Sigma$. Then (u, x) is a SRE in w if and only if (i) $\alpha = \mathsf{locus}(\overleftarrow{u})$ is a node in $\mathsf{ST}(\overleftarrow{w})$ with an x-link and at least another defined W-link, and (ii) for each child node β of α, either $W_x(\beta)$ is not defined, or $W_x(\beta)$ is the only defined W-link of β.*

Proof. Since u is right-maximal in w, it has at least two right extensions in w, one of which is x, and therefore $\overleftarrow{u}$ is preceded by at least two distinct letters in $\overleftarrow{w}$, one of which is x. Thus, $\alpha = \mathsf{locus}(\overleftarrow{u})$ has at least two defined W-links in $\mathsf{ST}(\overleftarrow{w})$, one of which is an x-link. By definition of SREs, for each $z \in \Sigma$, either $zux \notin \mathcal{F}_w$ or zu is followed only by x in w. This means that, for each $z \in \Sigma$ such that α has a z-edge, the child of α connected by this z-edge either has no x-link, or has an x-link as its sole W-link. This also implies that α must be a node, as assuming that it has only one child contradicts Lemma 2(ii) which implies that the W-links of an implicit node must be the same as those of its closest descendant. ◀

Note that according to Lemma 10, here we are only concerned with what W-links are defined at a node and not with the destination of those W-links or with the distinction between hard and soft W-links. Therefore, for the purpose of this section, it is sufficient for us to assume that each node is associated with a subset of letters $z \in \Sigma$ with a defined z-link. We now analyse how the set of SREs is modified when a new letter x is appended to the current string w. Similar to Section 4.1, we infer these modifications from a round of Weiner's algorithm maintaining the suffix tree for the reversed string $\overleftarrow{w}$.

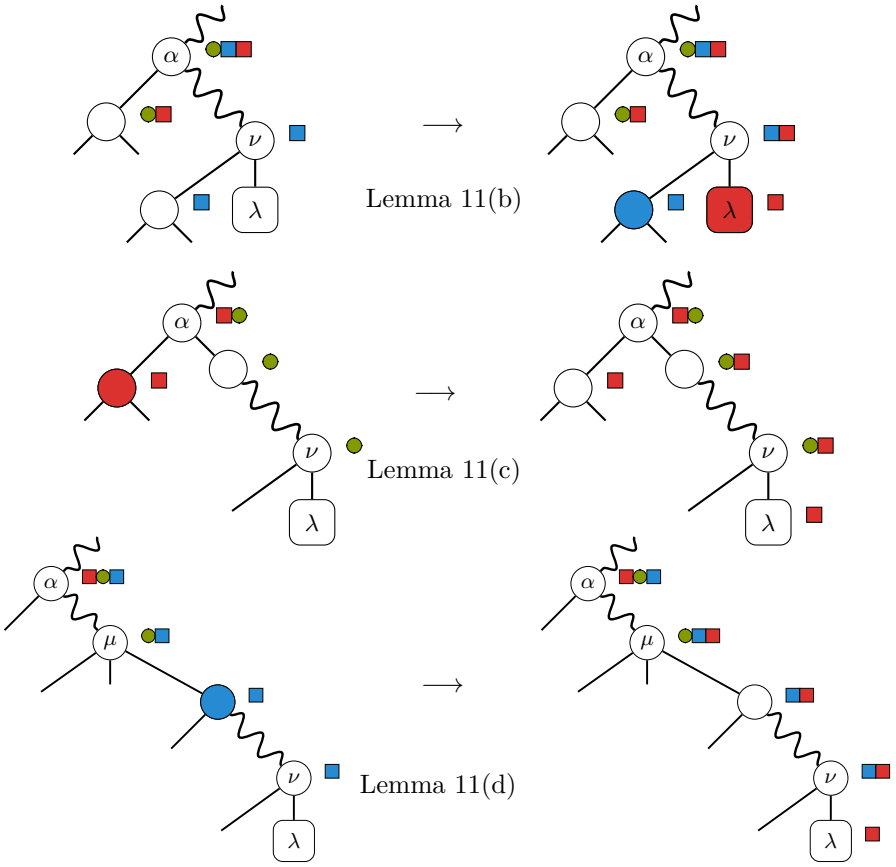

Figure 5 Illustration to the modifications of the set of SREs specified in Lemma 11(b),(c),(d) of Section 4.2.1. Each subfigure depicts a transformation resulting from a round of Weiner's algorithm. Like in Figure 3, squares represent individual letters and circles denote sets of letters. Color-filled nodes represent SREs: a node χ colored y witnesses a SRE (ζ, y), where χ is a child of ζ having only a y-link in its set of W-links. Lemma 11(b): the subfigure depicts the case when ν has a single W-link (■) before the round, in which case ν induces a new blue SRE in addition to the new red SRE. If ν has two or more W-links before the round, only the red SRE is created. Lemma 11(c): the subfigure depicts the case when α formed a red SRE before the round, which ceases to be a SRE after the round. Lemma 11(d): ν has a single color and μ is the lowest ancestor of ν with two or more colors. The blue SRE ceases to be a SRE.

According to Lemma 10, we have to keep track of pairs (α, x), where α is a node and $x \in \Sigma$, such that α has an x-link as well as at least one W-link by another letter, and any child node of α either has only an x-link or does not have an x-link at all. Like in the previous section, we will say that such a pair (α, x) is a SRE.

Appending x to w amounts to prepending x to $\overleftarrow{w}$, and the corresponding updates to the suffix tree made by Weiner's algorithm have been described in Section 3.1. In particular, recall that each new W-link belongs to one of the two types:

(i) W-links defined for the newly created node γ which are copied from the descendant node β,

(ii) x-links defined for nodes that are on the path to the leaf λ with $\mathsf{label}(\lambda) = \overleftarrow{w}$. In Figure 3, these are nodes on the path between α (excluded) and λ (included).

W-links of Type (i) are not relevant to us because the W-links of γ are the same as the W-links of its descendant β, and therefore the conditions of Lemma 10 cannot be satisfied. Therefore, only W-links of Type (ii) are of interest. This implies that if (μ, z) is a SRE to be added to or deleted from the current set of SREs, then μ is one of the nodes on the path between α (included) and λ (excluded). For the other nodes of the tree, their set of W-links as well as the set of W-links of their children is not modified.

Based on Lemma 10, we now specify the possible modifications to the set of SREs that can occur at a round of Weiner's algorithm. As before, we assume that x is the letter to be prepended to $\overleftarrow{w}$, λ is the leaf labeled $\overleftarrow{w}$, and α is the node labeled by the longest prefix of $\overleftarrow{w}$ that has an occurrence in $\overleftarrow{w}$ preceded by x. Let ν be the parent of λ. The following lemma specifies all the updates that can occur.

▶ **Lemma 11.**

(a) *If $\nu = \alpha$, that is, the parent of λ has an x-link, then no modification has to be made to the set of SREs.*

The following cases assume that $\nu \neq \alpha$, i.e. ν is a proper descendant of α.

(b) *(ν, x) is a new SRE. Furthermore, if ν had a single W-link by some letter $y \neq x$, then (ν, y) is a new SRE as well.*

(c) *If (α, x) is a SRE, then it has to be omitted in $S_r(wx)$.*

(d) *Let μ be the lowest proper ancestor of ν with two or more W-links such that its child on the path from μ to λ has only one W-link by some letter $y \neq x$. If (μ, y) is a SRE, it has to be omitted in $S_r(wx)$.*

Proof. Cases (b), (c) and (d) are illustrated in Figure 5.

(a) In this case, Weiner's algorithm assigns a single new x-link to λ, which does not modify the set of SREs.

(b) By Lemma 2(i), ν had at least one W-link and could not have an x-link. Weiner's algorithm assigns an x-link to λ and ν and therefore, by Lemma 10, (ν, x) becomes a SRE. If, before the update, ν had exactly one W-link, say by letter y, then (ν, y) becomes a SRE as well, as all descendants of ν except for λ must have a single W-link by y and λ has only the one by x.

(c) Since α had an x-link and at least one of its children (the one on the path to λ) had no x-link but had one or more W-links by other letter(s), it may happen that (α, x) was a SRE. After the round of Weiner's algorithm, (α, x) is then no longer a SRE, as it has now a child with an x-link and at least one other W-link by another letter.

(d) Observe that such node μ always exists whenever ν has a single W-link, and in this case, μ is uniquely defined independently of y as the lowest proper ancestor of ν with at least two W-links such that its child on the path to λ has a single W-link. Then, if (μ, y) was a SRE, it ceases to be a SRE in $S_r(wx)$, as it has now a child with both a y-link and an x-link. ◀

Before proceeding to algorithmic details, we show that the above cases are exhaustive.

Algorithm 2 Left-to-right update of $S_r(w)$ and of the leftmost SSS. $\textsc{Child}(\mu, z)$ returns the child of μ via the edge labeled by z, and $\textsc{UniqueWLinkLetter}(\nu)$ returns the letter corresponding to the unique W-link of ν.

Require: Current suffix tree $\mathsf{ST}(\overleftarrow{w})$ with the leaf λ labeled $\overleftarrow{w}$, current SRE set $S = S_r(w)$, bit vector B, and new letter x
Ensure: Updated structures for wx

$\nu \leftarrow \textsc{Parent}(\lambda)$
$\alpha \leftarrow$ lowest ancestor of λ with a defined x-link ▷ *cf. Operation (A) in Section 4.2.2*
$\mu \leftarrow$ lowest ancestor of λ with at least two W-links ▷ *cf. Operation (B) in Section 4.2.2*
$Ins \leftarrow \emptyset$; $Del \leftarrow \emptyset$
if $\nu \neq \alpha$ **then**
    $Ins \leftarrow Ins \cup \{(\nu, x)\}$
    if $\nu \neq \mu$ **then** ▷ *ν has a single W-link if $\nu \neq \mu$*
        $y \leftarrow \textsc{UniqueWLinkLetter}(\nu)$
        $Ins \leftarrow Ins \cup \{(\nu, y)\}$, $d \leftarrow \textsc{StringDepth}(\mu)$, and $z \leftarrow \overleftarrow{w}[d+1]$
        $\eta \leftarrow \textsc{Child}(\mu, z)$ ▷ *μ is lowest ancestor of λ with ≥ 2 W-links and*
        $y' \leftarrow \textsc{UniqueWLinkLetter}(\eta)$ ▷ *ν has single W-link $\Rightarrow$ η has single W-link*
        if $\textsc{IsSRE}(\mu, y')$ **then**
            $Del \leftarrow Del \cup \{(\mu, y')\}$
    if $\textsc{IsSRE}(\alpha, x)$ **then**
        $Del \leftarrow Del \cup \{(\alpha, x)\}$
update the suffix tree via Weiner's algorithm for prepending x to $\overleftarrow{w}$
assign $\textsc{UniqueWLinkLetter}(\lambda)$ and, if needed, $\textsc{UniqueWLinkLetter}(\gamma)$
for all $(\mu, c) \in Del$ **do** $\textsc{DeleteSRE}(\mu, c)$ ▷ *flip the bit for its leftmost end position*
for all $(\mu, c) \in Ins$ **do** $\textsc{InsertSRE}(\mu, c)$ ▷ *flip the bit for its leftmost end position*
if $\nu \neq \alpha \wedge \nu \neq \mu$ **then** explicitly color ν for Operation (B)
update λ to the new full-text leaf and regard the current text as wx

▶ **Lemma 12.** *Lemma 11 specifies all possible modifications to the set of SREs.*

Proof. As noted earlier (W-links of Type (ii)), only nodes μ between α (included) and λ (excluded) can form SREs to be added or deleted. For the letter x, a new SRE (μ, x) can only occur if $\mu = \nu$, and is covered by Lemma 11(b) – for each other node ξ on this path, one of ξ's children has a W-link by another letter (due to Lemma 2(i)). A SRE (μ, x) to be deleted can only occur if $\mu = \alpha$, and is covered by Lemma 11(c). A letter $y \neq x$ can cause a modification in two cases as well: A new SRE (μ, y) can only appear for $\mu = \nu$ (Lemma 11(b)), and a SRE (μ, y) can cease to be a SRE only if μ is the parent of a node having a single y-link, as specified in Lemma 11(d). ◀

Unsurprisingly, modifications described in Lemma 11(b),(c),(d) correspond to those described in [10, Section 7.1]. The difference is that in [10], modifications are described in terms of the suffix tree for w maintained by Ukkonen's algorithm, whereas we describe them in terms of the suffix tree for $\overleftarrow{w}$ maintained by Weiner's algorithm. Below we show how to leverage our version in order to maintain the set of SREs in near-real-time.

Observe also that if the conditions of Lemma 11(d) are satisfied, then the parent of λ must have a single W-link, and therefore two new SREs are added according to Lemma 11(b). This implies that possible modifications amount to adding between zero and two new SREs [10, Lemma 2].

4.2.2 Maintaining the set of SREs in near-real-time

Based on the results of Section 4.2.1, we now describe how we can maintain the set of SREs in near-real-time relying on a near-real-time implementation of Weiner's algorithm from Section 3.

To compute the SREs to be added or deleted as specified by Lemma 11(b),(c),(d), we have to support the following two operations on the suffix tree.

(A) given a leaf node λ and a letter $x \in \Sigma$, find the lowest ancestor α of λ with a defined x-link,

(B) given a leaf node λ, find the lowest ancestor μ of λ with at least two defined W-links such that its child on the path to λ has only one defined W-link.

Operation (A) implements the update described in Lemma 11(c), whereas Operation (B) implements the update described in Lemma 11(d). Checking Lemma 11(a) and implementing Lemma 11(b) is trivial because ν is directly accessed from its child λ. Note also that if ν has two or more W-links, Lemma 11(d) does not apply.

Operation (A) is a part of Weiner's algorithm and has been discussed in Section 3 (see Fig. 3). We implement it using the reduction to COLORED PREDECESSOR, as explained in Sections 3.2 and 3.3.

Operation (B) is implemented similarly. We use Lemma 3 with a single color: the colored nodes are nodes with two or more defined W-links. Let M be the set of these nodes. By Lemma 2(ii), M is closed under ancestry relation, i.e. the coloring has the fringe property. Our goal is to maintain some subset $D \subseteq M$ such that every node from M has a descendant node from D. Nodes of D will be considered explicitly colored in the sense of Lemma 3.

We reason by induction on the rounds of Weiner's algorithm. Assume we have correctly computed the set D, that is every node with two or more W-links has an explicitly colored descendant. We show how to maintain D through a round of Weiner's algorithm. We start by observing that the W-links assigned to the newly created node γ (see Figure 3) do not cause any modifications to the set D because the set of W-links of γ is the same as that of its child β. Then, only the new W-links on the path between α (excluded) and λ (included) may cause modification. For the nodes on this path, Weiner's algorithm adds one new x-link (x letter prepended at this round) and therefore by Lemma 2(i), all these nodes except for λ belong to M after this round. Thus, it is sufficient to explicitly color the immediate parent of λ unless it was already in D before the round. In other words, if before the round the immediate parent of λ had only one defined W-link, it has to be explicitly colored. All other nodes that are not on the path between α and λ do not require any updates of D. We summarize the arguments in the following statement.

▶ **Lemma 13.** *Explicitly coloring the immediate parent of λ unless it was explicitly colored before correctly maintains the set D of explicitly colored nodes.*

Since the coloring of Lemma 13 is done by two updates of an Euler tour list with only one color, Lemma 3 implies that Operation (B) can be implemented via COLORED PREDECESSOR query in the case of one color. Thus, Operation (B) can be executed in $O(\log \log n)$ worst-case time (cf. comment after Theorem 5).

Operation (B) retrieves node μ. To implement the case of Lemma 11(d), we first need to access the child η of μ on the path to λ. To determine the edge leading to η, we retrieve the string depth[4] d of μ and then retrieve $z := \overleftarrow{w}[d+1]$. η is then accessed through the edge whose label starts with z.

[4] String depths of nodes can be determined and maintained along Weiner's algorithm at the creation time of the nodes in constant time.

We then have to determine the letter y that corresponds to the only W-link of η, however, since we do not store soft W-links explicitly, it may happen that the desired W-link is not stored at η. To cope with this, we explicitly store an additional letter at some nodes ensuring the following invariant: if a node has exactly one defined W-link, the stored letter corresponds to this W-link. We maintain this information through a round of Weiner's algorithm as follows: we only have to assign letter x to leaf λ and to possibly assign a letter to the newly created node γ by copying it from its ancestor β in case β had an assigned letter (see Figure 3). The assigned letter does not allow, in general, testing if a node has a single W-link, but will allow us to retrieve the corresponding letter for nodes known to have exactly one W-link. Thus, after accessing η, we retrieve the letter y corresponding to the unique W-link of η.

We now specify how we keep track of the current set of SREs and of an SSS. Before executing the W-link updates of the round, we determine which SREs have to be added or deleted, as specified in Lemma 11(b),(c), and (d). The corresponding bit-vector updates are then performed once the insertion locus and the relevant new W-link destinations are known. Similarly to Section 4.1, we keep track of the leftmost end positions of the current SREs in the current string w, exploiting the distinctness of this position for each SRE. For that, we maintain a growable (to the right) vector of $O(n)$ bits marking those positions. Since we work with the suffix tree of the reversed string $\overleftarrow{w}$, we maintain, at each node ν, the *rightmost start* position of $\mathsf{label}(\nu)$ in the current string $\overleftarrow{w}$. This is done similarly to how it is described in Section 4.1. It remains to specify how we compute the leftmost end position of a given SRE that we have to add to or delete from the current set. That is, if (ν, x) is a SRE, we need to compute the rightmost start position of $x \cdot \mathsf{label}(\nu)$ in $\overleftarrow{w}$. If $W_x(\nu)$ is hard, this position is retrieved directly from the destination node. If $W_x(\nu)$ is soft, we use the technique of [15] described at the end of Section 3.2 that enables the recovery of soft W-links without explicitly storing them. Then, again, we retrieve the start position from the destination node. Algorithm 2 summarizes the whole algorithm. The results of this section are thus summarized as follows.

▶ **Theorem 14.** *In the left-to-right framework, the set of SREs and the SSS consisting of the leftmost positions of each SRE can be maintained online in worst-case $O(\log^2 \log n)$ time per letter. This bound can be reduced to $O(\log \log n)$ for alphabet size $\sigma = O(\log^{1/4} n)$.*

5 Conclusion

We have shown how to maintain the set of SREs and an SSS in near-real-time when the text is received online from either right to left or left to right. In both cases, we used the near-real-time implementation of Weiner's algorithm to achieve the same time complexity bounds, i.e. worst-case $O(\log^2 \log n)$ time per letter, or $O(\log \log n)$ for alphabet size $\sigma = O(\log^{1/4} n)$. Further improvements of the time complexity bounds for maintaining the suffix tree with Weiner's algorithm in near-real-time would directly imply improvements for maintaining the set of SREs and an SSS. While our space usage is $O(n)$, it would be interesting to study whether we can use techniques based on the online construction of the Burrows–Wheeler transform to maintain the set of SREs online using compact space or compressed space in terms of the number of letter runs in the BWT.

References

1 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. doi:10.1016/j.jda.2012.07.003.

2 Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proc. WADS*, volume 1663 of *LNCS*, pages 37–48, 1999. doi:10.1007/3-540-48447-7_4.

3 Davide Cenzato, Lore Depuydt, Travis Gagie, Sung-Hwan Kim, Giovanni Manzini, Francisco Olivares, and Nicola Prezza. Suffixient arrays: a new efficient suffix array compression technique. *CoRR*, abs/2407.18753, 2025. doi:10.48550/arXiv.2407.18753.

4 Davide Cenzato, Francisco Olivares, and Nicola Prezza. On computing the smallest suffixient set. In Zsuzsanna Lipták, Edleno Silva de Moura, Karina Figueroa, and Ricardo Baeza-Yates, editors, *String Processing and Information Retrieval - 31st International Symposium, SPIRE 2024, Puerto Vallarta, Mexico, September 23-25, 2024, Proceedings*, volume 14899 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2024. doi:10.1007/978-3-031-72200-4_6.

5 Davide Cenzato, Francisco Olivares, and Nicola Prezza. Testing suffixient sets. *CoRR*, abs/2506.08225, 2025. doi:10.48550/arXiv.2506.08225.

6 Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Trans. Algorithms*, 17(1), December 2021. doi:10.1145/3426473.

7 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi:10.1137/S0097539700370539.

8 Lore Depuydt, Travis Gagie, Ben Langmead, Giovanni Manzini, and Nicola Prezza. Suffixient sets. *CoRR*, abs/2312.01359, 2023. doi:10.48550/arXiv.2312.01359.

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

10 Hiroto Fujimaru, Gonzalo Navarro, Giuseppe Romana, and Cristian Urbina. Smallest suffixient sets: Effectiveness, resilience, and calculation. *CoRR*, abs/2506.05638, 2025. doi:10.48550/arXiv.2506.05638.

11 Yoav Giyora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Trans. Algorithms*, 5(3):28:1–28:51, 2009. doi:10.1145/1541885.1541889.

12 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

13 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proc. STOC*, pages 827–840, 2018. doi:10.1145/3188745.3188814.

14 Dominik Köppl and Gregory Kucherov. Near-real-time solutions for online string problems. *CoRR*, abs/2602.15311, 2026. to appear in CPM 2026. doi:10.48550/arXiv.2602.15311.

15 Gregory Kucherov and Yakov Nekrich. Full-fledged real-time indexing for constant size alphabets. *Algorithmica*, 79(2):387–400, 2017. doi:10.1007/s00453-016-0199-7.

16 Gregory Kucherov and Yakov Nekrich. Online computation of normalized substring complexity. *CoRR*, abs/2510.16454, 2025. appeared in LATIN 2026. doi:10.48550/arXiv.2510.16454.

17 Christian Worm Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 618–627. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644210.

18 Christian Worm Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.*, 35(6):1494–1525, 2006. doi:10.1137/S0097539703436722.

19 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2021. doi:10.1145/3434399.

20 Gonzalo Navarro. Indexing highly repetitive string collections, part II: compressed indexes. *ACM Comput. Surv.*, 54(2):26:1–26:32, 2021. doi:10.1145/3432999.

21 Gonzalo Navarro, Giuseppe Romana, and Cristian Urbina. Smallest suffixient sets as a repetitiveness measure. In Golnaz Badkobeh, Jakub Radoszewski, Nicola Tonellotto, and Ricardo Baeza-Yates, editors, *String Processing and Information Retrieval - 32nd International Symposium, SPIRE 2025, London, UK, September 8-11, 2025, Proceedings*, volume 16073 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2025. doi:10.1007/978-3-032-05228-5_18.

22 Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013. doi:10.1007/s00453-012-9618-6.

23 Hiroki Shibata and Hideo Bannai. String representation in suffixient set size space. *CoRR*, abs/2604.04377, 2026. doi:10.48550/arXiv.2604.04377.

24 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Symposium on Switching and Automata Theory (SWAT)*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.