Algorithmica (2022) 84:670–693
https://doi.org/10.1007/s00453-021-00864-1

Computing Minimal Unique Substrings for a Sliding Window

Takuya Mieno[1,2] · **Yuta Fujishige**[1,2,3] · **Yuto Nakashima**[1] · **Shunsuke Inenaga**[1,4] · **Hideo Bannai**[1,5] · **Masayuki Takeda**[1]

Received: 29 August 2020 / Accepted: 1 August 2021 / Published online: 20 August 2021

Abstract

A substring u of a string T is called a *minimal unique substring* (*MUS*) of T if u occurs exactly once in T and any proper substring of u occurs at least twice in T. In this paper, we study the problem of computing MUSs for a sliding window over a given string T. We first show how the set of MUSs can change when the window slides over T. We then present an $O(n \log \sigma')$-time and $O(d)$-space algorithm to compute MUSs for a sliding window of size d over the input string T of length n, where $\sigma' \leq d$ is the maximum number of distinct characters in every window.

Keywords Minimal unique substring · Sliding window · Suffix tree

✉ Takuya Mieno
takuya.mieno@inf.kyushu-u.ac.jp

Yuto Nakashima
yuto.nakashima@inf.kyushu-u.ac.jp

Shunsuke Inenaga
inenaga@inf.kyushu-u.ac.jp

Hideo Bannai
hdbn.dsc@tmd.ac.jp

Masayuki Takeda
takeda@inf.kyushu-u.ac.jp

[1] Department of Informatics, Kyushu University, Fukuoka, Japan

[2] Japan Society for the Promotion of Science, Tokyo, Japan

[3] Present Address: Fujitsu Laboratories Ltd., Kawasaki, Japan

[4] PRESTO, Japan Science and Technology Agency, Kawaguchi, Japan

[5] Present Address: M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan

1 Introduction

1.1 Minimal Unique Substrings and Shortest Unique Substrings

A *unique substring* of string T is a substring of T which appears exactly once in T. Finding unique substrings of DNA sequences has gained attention in bioinformatics [8, 9, 15, 24]. For example, it can be applied in PCR primer design [24] and alignment-free genome comparison [9].

In the last decade, problems that relate to computing unique substrings in a given string have been studied in the field of string algorithmics. A unique substring u of T is said to be a *minimal unique substring* (*MUS*) of T if any proper substring of u is not a unique substring. Ilie and Smyth [13] formalized MUSs and proposed a linear time algorithm to compute all MUSs of a given string T.

MUSs has been heavily utilized for solving the *shortest unique substring* (*SUS*) problems: A unique substring $v = T[s \dots t]$ of T is said to be a *shortest unique substring* (*SUS*) of T for a text position p if v contains the position p (i.e., $p \in [s, t]$) and any proper substring of v which contains p is not a unique substring. The *single-SUS problem* is to preprocess a given string T of length n so that for any subsequent query position p, a SUS for p can be answered quickly. Pei et al. [20] introduced the single-SUS problem and gave an $O(n^2)$-time preprocessing scheme which can answer single-SUS queries in constant time. Tsuruta et al. [22], Ileri et al. [12], and Hu et al. [11] independently showed $O(n)$-time preprocessing schemes which can answer single-SUS queries in constant time. Also, Hon et al. [10] proposed an in-place algorithm for computing SUSs for all positions in linear time. The *all-SUS problem* is a generalization of the single-SUS problem which requires to output *all* SUSs for given position p. The methods of Tsuruta et al. [22] and Hu et al. [11] can answer all-SUS queries in $O(occ)$ time, where occ is the number of SUSs to output. Note that the SUS problem studied by [11] is more general, that is, they find SUSs covering a given *interval* in the string, instead of a text position. Moreover, Mieno et al. [16] considered the all-SUS problem on a run-length encoded string, and proposed an $O(r)$-space data structure which can answer all-SUS (interval) queries in $O(\sqrt{\log r / \log\log r} + occ)$ time, where r is the size of a given run-length encoded string. Although not mentioned explicitly in [16], the size of their data structure (except for the input string) and query time can be respectively written as $O(m)$ space and $O(\sqrt{\log m / \log\log m} + occ)$ time with respect to the number m of MUSs of the input string T. Note that all the above algorithms for the SUS problems compute all MUSs of the given string (or some data structure which is essentially equivalent to MUSs) in the preprocessing. We also refer to [1, 3, 7, 17] for related results on the SUS problems.

1.2 Sliding Window Model

In this paper, we tackle the problem of computing MUSs in the *sliding window model*. In the sliding window model, the input string is given in an *online* fashion,

one character at a time from left to right, and the memory usage is limited to some pre-determined space. The task of the sliding window model is to process all substrings $T[i \dots i+d-1]$ of pre-fixed length d in a string T of length n in an incremental fashion, for increasing $i = 0, \dots, n-d$. Usually the window size d is set to be much smaller than the string length n, and thus the challenge here is to design efficient algorithms that process all such substrings using only $O(d)$ working space.

A typical application to the sliding window model is data compression; examples are the famous Lempel-Ziv 77 (the original version) [25] and PPM [4]. Recently, Crochemore et al. [5] introduced the problem of computing *Minimal Absent Words* for a sliding window, and proposed an $O(n\sigma)$-time and $O(d\sigma)$-space algorithm using suffix trees for a sliding window where σ is the alphabet size. This paper deals with the problem of computing MUSs in a sliding window. This problem can be directly applied to compute *uniqueness score* of oligonucleotides for designing tilling arrays [8].

1.3 Our Contributions

We begin with combinatorial results on MUSs for a sliding window. Namely, we show that the number of MUSs that are added or deleted by one slide of the window is always constant (Sect. 3). We then present the first efficient algorithm that maintains the set of MUSs for a sliding window of length d over a string of length n in a total of $O(n \log \sigma')$ time and $O(d)$ working space where $\sigma' \leq d$ is the maximum number of distinct characters in every window (Sect. 4). Our main algorithmic tool is the suffix tree for a sliding window that requires $O(d)$ space and can be maintained in $O(n \log \sigma')$ time [6, 14, 21]. Our algorithm for computing MUSs for a sliding window is built on our combinatorial results, and it keeps track of three different loci over the suffix tree, all of which can be maintained in $O(\log \sigma')$ amortized time per each sliding step.

A part of the results reported in this article appeared in a preliminary version of this paper, [18]. The preliminary paper [18] consists of two parts: (1) efficient computation and combinatorial properties of MUSs for a sliding window, and (2) combinatorial properties of minimal absent words (MAWs) [19] for a sliding window. This current article is a full version of the former part (1) which contains complete proofs and supplemental figures which were omitted in the preliminary version [18]. We remark that an extended version of the latter part (2) can be found as an independent article [2].

2 Preliminaries

2.1 Strings

Let Σ be an *alphabet* of size σ. An element of Σ is called a *character*. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. The *empty string* ε is the string of length 0. If $T = xyz$, then x, y, and z are called a *prefix*, *substring*,

and *suffix* of T, respectively. They are called a *proper* prefix, *proper* substring, and *proper* suffix of T if $x \neq T$, $y \neq T$, and $z \neq T$, respectively. If a string b is a proper prefix of T and is a proper suffix of T, b is called a *border* of T.

For any $0 \leq i \leq |T| - 1$, $T[i]$ denotes the ith character of T. For any $0 \leq i \leq j \leq |T| - 1$, $T[i \dots j]$ denotes the substring of T starting at position i and ending at position j, i.e., $T[i \dots j] = T[i]T[i+1] \cdots T[j]$. For convenience, let $T[i' \dots j'] = \varepsilon$ for any $i' > j'$. For any $0 \leq i \leq |T| - 1$, $T[i \dots]$ denotes the suffix starting at position i, i.e., $T[i \dots] = T[i \dots |T| - 1]$.

For a non-empty string w, the set of beginning positions of occurrences of w in T is denoted by $occ_T(w) = \{i \mid T[i \dots i + |w| - 1] = w\}$. Let $\#occ_T(w) = |occ_T(w)|$. For any substring w of T, w is called *unique* in T if $\#occ_T(w) = 1$, *quasi-unique* in T if $1 \leq \#occ_T(w) \leq 2$, and *repeating* in T if $\#occ_T(w) \geq 2$. For convenience, let $\#occ_T(\varepsilon) = |T| + 1$, and thus, ε is always repeating in any non-empty string. For any $0 \leq i \leq j \leq |T| - 1$, $lrSuf_{i,j}$ denotes the longest repeating suffix of $T[i \dots j]$, $sqSuf_{i,j}$ denotes the shortest quasi-unique suffix of $T[i \dots j]$, and $sqPref_{i,j}$ denotes the shortest quasi-unique prefix of $T[i \dots j]$. While $lrSuf_{i,j}$ can be the empty string, both $sqSuf_{i,j}$ and $sqPref_{i,j}$ are always non-empty strings for any i, j with $0 \leq i \leq j \leq |T| - 1$. See Fig. 1 for examples.

In what follows, we consider an arbitrarily fixed string T of length $n \geq 1$ over an alphabet Σ of size $\sigma \geq 2$.

2.2 Minimal Unique Substrings

A unique substring $u = T[s \dots t]$ of T is called *a minimal unique substring* (*MUS*) of T if and only if both $T[s+1 \dots t]$ and $T[s \dots t-1]$ are repeating in T. Since a unique substring u of T has exactly one occurrence in T, it can be identified with a unique interval $[s, t]$ such that $0 \leq s \leq t \leq n-1$ and $u = T[s \dots t]$. We denote by $\mathsf{MUS}(T) = \{[s,t] \mid T[s \dots t] \text{ is a MUS of } T\}$ the set of intervals corresponding to the MUSs of T. See Fig. 1 for examples of MUSs.

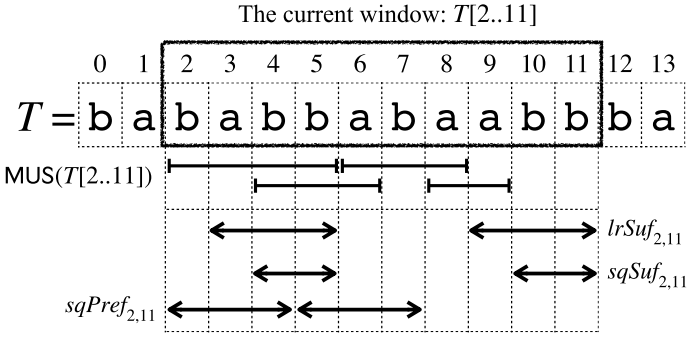


Fig. 1 String $T = \texttt{bababbabaabbba}$ of length 14 and its substrings $lrSuf_{2,11}$, $sqSuf_{2,11}$, and $sqPref_{2,11}$ for the current window $T[2 \dots 11]$

This paper deals with the problem of computing MUSs for a sliding window of fixed length d over a given string T, formalized as follows:

Input String T of length n and positive integer d ($< n$).
Output $\mathsf{MUS}(T[i \dots i+d-1])$ for all $0 \le i \le n-d$.

2.3 Suffix Trees

The *suffix tree* of string T, denoted $\mathsf{STree}(T)$, is a *compacted trie* that represents all suffixes of T. We consider a version of suffix trees a.k.a. *Ukkonen trees* [23]: Namely, $\mathsf{STree}(T)$ is a rooted tree such that

1. each edge is labeled by a non-empty substring of T,
2. each internal node has at least two children,
3. the out-going edges of each node begin with mutually distinct characters, and
4. the suffixes of T that are unique in T are represented by paths from the root to the leaves, and the other suffixes of T that are repeating in T are represented by paths from the root that end either on internal nodes or on edges.

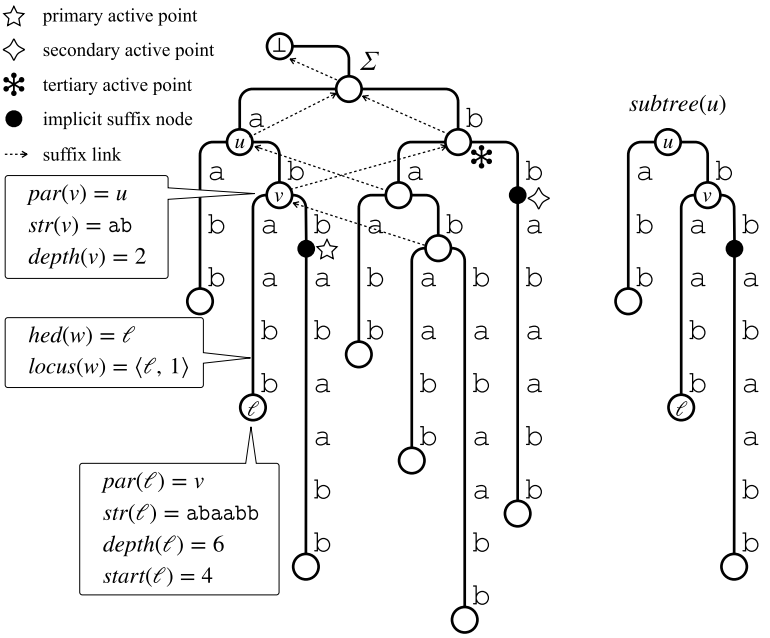


Fig. 2 The suffix tree of string $T = \texttt{babbabaabb}$, where the suffix links are depicted by broken arrows, the implicit suffix nodes are depicted by black circles, as well as the three kinds of active points are marked. For example of other notions on the suffix tree, substring $w = \texttt{abaab}$ of T is considered here

To simplify the description of our algorithm, we assume that there is an auxiliary node $\perp$ which is the parent of only the root node. The out-going edge of $\perp$ is labeled with Σ; This means that we can go down from $\perp$ by reading any character in Σ. See Fig. 2 for an example of $\mathsf{STree}(T)$.

For each node v in $\mathsf{STree}(T)$, $parent(v)$ denotes the parent of v, $str(v)$ denotes the path string from the root to v, $depth(v)$ denotes the *string depth* of v (i.e., $depth(v) = |str(v)|$), and $subtree(v)$ denotes the subtree of $\mathsf{STree}(T)$ rooted at v. For each leaf ℓ in $\mathsf{STree}(T)$, $start(\ell)$ denotes the starting position of $str(\ell)$ in T. For each non-empty substring w of T, $hed(w) = v$ denotes the *highest explicit descendant* where w is a prefix of $str(v)$ and $depth(parent(v)) < |w| \leq depth(v)$. For each substring w of T, $locus(w) = \langle u, h \rangle$ represents the locus in $\mathsf{STree}(T)$ where the path that spells out w from the root terminates, such that $u = hed(w)$ and $h = depth(u) - |w| \geq 0$. Namely, h is the off-set length from the child u of the locus for w when w is on an edge, and $h = 0$ when w is on a node (namely u). We say that a substring w of T with $locus(w) = \langle u, h \rangle$ is represented by an *explicit node* if $h = 0$, and by an *implicit node* if $h \geq 1$. We remark that in the Ukkonen tree $\mathsf{STree}(T)$ of a string T, some repeating suffixes may be represented by implicit nodes. An implicit node which represents a suffix of T is called an *implicit suffix node*. For any internal node v except for the root, the *suffix link* of v is a reversed edge from v to the explicit node that represents $str(v)[1 \ldots]$. The suffix link of the root that represents ϵ points to $\perp$.

3 Combinatorial Results on MUSs for a Sliding Window

Throughout this section, we consider positions i and j with $0 \leq i \leq j \leq n-1$ such that $T[i \ldots j]$ denotes the sliding window for the ith position over the input string T. The following arguments hold for *any* values of i and j, and hence, they will be useful for sliding windows of any length d. The next lemmas are useful for analyzing combinatorial properties of MUSs and for designing an efficient algorithm for computing MUSs for a sliding window.

Lemma 1 *The following three statements are equivalent*:

(1) $|lrSuf_{i,j}| \geq |sqSuf_{i,j}|$,
(2) $\#occ_{T[i\ldots j]}(lrSuf_{i,j}) = 2$, *and*
(3) $\#occ_{T[i\ldots j]}(sqSuf_{i,j}) = 2$.

Proof **(1) ⇒ (2) and (3):** Since $|lrSuf_{i,j}| \geq |sqSuf_{i,j}|$, $sqSuf_{i,j}$ is a suffix of $lrSuf_{i,j}$ and thus $\#occ_{T[i\ldots j]}(sqSuf_{i,j}) \geq \#occ_{T[i\ldots j]}(lrSuf_{i,j})$. By the definitions of $sqSuf_{i,j}$ and $lrSuf_{i,j}$, $\#occ_{T[i\ldots j]}(sqSuf_{i,j}) \leq 2$ and $\#occ_{T[i\ldots j]}(lrSuf_{i,j}) \geq 2$. Thus $\#occ_{T[i\ldots j]}(lrSuf_{i,j}) = \#occ_{T[i\ldots j]}(sqSuf_{i,j}) = 2$.

(2) ⇒ (1): Since $\#occ_{T[i\ldots j]}(lrSuf_{i,j}) = 2$, the shortest suffix $sqSuf_{i,j}$ of $T[i \ldots j]$ that occurs at most twice in $T[i \ldots j]$ cannot be longer than $lrSuf_{i,j}$, i.e., $|lrSuf_{i,j}| \geq |sqSuf_{i,j}|$.

(3) ⇒ (1): Since $\#occ_{T[i\ldots j]}(sqSuf_{i,j}) = 2$, the longest suffix $lrSuf_{i,j}$ of $T[i\ldots j]$ that occurs at least twice in $T[i\ldots j]$ is at least as long as $sqSuf_{i,j}$, i.e., $|lrSuf_{i,j}| \geq |sqSuf_{i,j}|$.

Figure 1 shows a concrete example where (1) of Lemma 1 holds (and hence both (2) and (3) also hold.)

Lemma 2 $|lrSuf_{i,j+1}| \leq |lrSuf_{i,j}| + 1$.

Proof Assume on the contrary that $|lrSuf_{i,j+1}| > |lrSuf_{i,j}| + 1$. By the definition of $lrSuf_{i,j+1}$, $lrSuf_{i,j+1} = T[j+2-|lrSuf_{i,j+1}|\ldots j+1]$ occurs at least twice in $T[i\ldots j+1]$. Hence, $T[j+2-|lrSuf_{i,j+1}|\ldots j]$ which is a proper prefix of $lrSuf_{i,j+1}$ also occurs at least twice in $T[i\ldots j]$. In addition, $lrSuf_{i,j} = T[j+2-|lrSuf_{i,j}|\ldots j]$ is a proper suffix of $T[j+2-|lrSuf_{i,j+1}|\ldots j]$ since $|lrSuf_{i,j+1}| > |lrSuf_{i,j}| + 1$. However, this contradicts the definition of $lrSuf_{i,j}$. Therefore, $|lrSuf_{i,j+1}| \leq |lrSuf_{i,j}| + 1$.

3.1 Changes to MUSs When Appending a Character to the Right

In this subsection, we consider an operation that slides the right-end of the current window $T[i\ldots j]$ with one character by appending the next character $T[j+1]$ to $T[i\ldots j]$. We use the following observation.

Observation 1 For any non-empty substring s of $T[i\ldots j]$,

$$\#occ_{T[i\ldots j+1]}(s) \leq \#occ_{T[i\ldots j]}(s) + 1.$$

Also, the equality holds if and only if s is a suffix of $T[i\ldots j+1]$.

3.1.1 MUSs to be Deleted When Appending a Character to the Right

Due to Observation 1, we obtain Lemma 3 which describes MUSs to be deleted when a new character $T[j+1]$ is appended to the current window $T[i\ldots j]$.

Lemma 3 *For any* $[s,t]$ *with* $i \leq s < t \leq j$, $[s,t] \in \mathsf{MUS}(T[i\ldots j])$ *and* $[s,t] \notin \mathsf{MUS}(T[i\ldots j+1])$ *if and only if* $T[s\ldots t] = sqSuf_{i,j+1}$ *and* $\#occ_{T[i\ldots j+1]}(sqSuf_{i,j+1}) = 2$.

Proof ($\Rightarrow$) Let $w = T[s\ldots t]$. Since $[s,t] \in \mathsf{MUS}(T[i\ldots j])$ and $[s,t] \notin \mathsf{MUS}(T[i\ldots j+1])$, $\#occ_{T[i\ldots j]}(w) = 1$ and $\#occ_{T[i\ldots j+1]}(w) \geq 2$. It follows from Observation 1 that $\#occ_{T[i\ldots j+1]}(w) = 2$ and w is a suffix of $T[i\ldots j+1]$. If we assume that w is a proper suffix of $sqSuf_{i,j+1}$, then $\#occ_{T[i\ldots j+1]}(w) \geq 3$ by the definition of $sqSuf_{i,j+1}$, but this contradicts with $\#occ_{T[i\ldots j+1]}(w) = 2$. If we assume that $sqSuf_{i,j+1}$ is a proper suffix of w, then $\#occ_{T[i\ldots j]}(sqSuf_{i,j+1}) \geq \#occ_{T[i\ldots j]}(T[s+1\ldots t]) \geq 2$.

Also, $\#occ_{T[i\dots j+1]}(sqSuf_{i,j+1}) = \#occ_{T[i\dots j]}(sqSuf_{i,j+1}) + 1 \geq 3$ by Observation 1, but this contradicts the definition of $sqSuf_{i,j+1}$. Therefore, $w = sqSuf_{i,j+1}$. Moreover, $\#occ_{T[i\dots j+1]}(sqSuf_{i,j+1}) = 2$ since $w = sqSuf_{i,j+1}$ is a substring of $T[i \dots j]$.

($\Leftarrow$) Since $w = T[s \dots t]$ is a suffix of $T[i \dots j+1]$ and $\#occ_{T[i\dots j+1]}(w) = 2$, w is unique in $T[i \dots j]$. By the definition of $sqSuf_{i,j+1}$, a proper suffix $w[1 \dots] = T[s+1 \dots t]$ of $w = sqSuf_{i,j+1}$ occurs at least three times in $T[i \dots j+1]$, i.e., $T[s+1 \dots t]$ is repeating in $T[i \dots j]$ (see also Fig. 3 for illustration).

Also, a prefix $w[0 \dots |w|-2] = T[s \dots t-1]$ of $w = sqSuf_{i,j+1}$ is clearly repeating in $T[i \dots j]$. Therefore, $w = T[s \dots t]$ is a MUS of $T[i \dots j]$ and is not a MUS of $T[i \dots j+1]$.

By Lemma 3, at most one MUS can be deleted when appending $T[j+1]$ to the current window $T[i \dots j]$, and such a deleted MUS must be $sqSuf_{i,j+1}$.

3.1.2 MUSs to be Added When Appending a Character to the Right

First, we consider a MUS to be added when appending $T[j+1]$ to $T[i \dots j]$, which is a suffix of $T[i \dots j+1]$. The next observation follows from the definition of $lrSuf_{i,j}$:

Observation 2 If $[s,j] \in \mathsf{MUS}(T[i \dots j])$, then $s = j - |lrSuf_{i,j}|$. Namely, if there is a MUS of $T[i \dots j]$ that is a suffix of $T[i \dots j]$, then it must be the suffix of $T[i \dots j]$ that is exactly one character longer than $lrSuf_{i,j}$.

Lemma 4 *The interval* $[j+1-\ell, j+1] \in \mathsf{MUS}(T[i \dots j+1])$ *if and only if* $T[j+1-\ell \dots j+1] = \alpha^{\ell+1}$ *or* $\ell \leq |lrSuf_{i,j}|$, *where* $\ell = |lrSuf_{i,j+1}|$ *and* $\alpha = T[j+1]$.

Proof ($\Rightarrow$) Assume on the contrary that $T[j+1-\ell \dots j+1] \neq \alpha^{\ell+1}$ and $\ell > |lrSuf_{i,j}|$. By the assumptions and Lemma 2, $|lrSuf_{i,j}| = \ell - 1$, and thus, $T[j-|lrSuf_{i,j}| \dots j] = T[j+1-\ell \dots j]$. Since $T[j+1-\ell \dots j+1]$ is a MUS of $T[i \dots j+1]$, $T[j+1-\ell \dots j] = T[j-|lrSuf_{i,j}| \dots j]$ occurs at least twice in $T[i \dots j+1]$. On the other hand, $T[j-|lrSuf_{i,j}| \dots j]$ is unique in $T[i \dots j]$ by the definition of $lrSuf_{i,j}$, hence $T[j-|lrSuf_{i,j}| \dots j]$ occurs in $T[i \dots j+1]$ as a suffix of $T[i \dots j+1]$. Consequently, we have $T[j-|lrSuf_{i,j}| \dots j] = T[j+1-|lrSuf_{i,j}| \dots j+1]$, i.e., $T[j-\ell \dots j] = T[j+1-\ell \dots j+1] = \alpha^{\ell+1}$ with $\alpha = T[j+1]$, a contradiction.

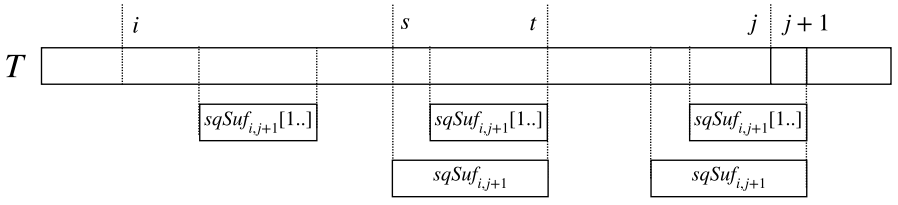


Fig. 3 Illustration for the case where $\#occ_{T[i\dots j+1]}(sqSuf_{i,j+1}) = 2$. In this case, $T[s \dots t] = sqSuf_{i,j+1}$ is unique in $T[i \dots j]$ and $T[s+1 \dots t]$ is repeating in $T[i \dots j]$

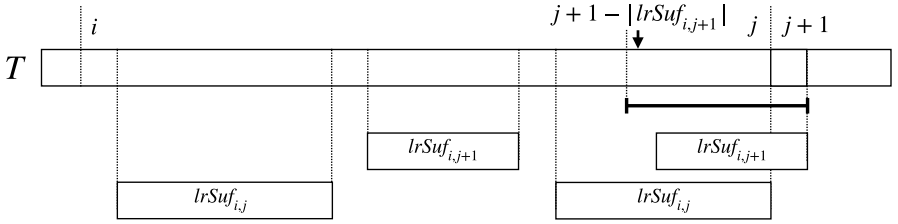


Fig. 4 Illustration for the case where $|lrSuf_{i,j+1}| \leq |lrSuf_{i,j}|$. In this case, $T[j+1-|lrSuf_{i,j+1}| \dots j+1]$ is a MUS of $T[i \dots j+1]$

($\Leftarrow$) By the definition of $lrSuf_{i,j+1}$, $T[j+2-\ell \dots j+1] = lrSuf_{i,j+1}$ is repeating in $T[i \dots j+1]$, and $T[j+1-\ell \dots j+1]$ is unique in $T[i \dots j+1]$. Now it suffices to show $T[j+1-\ell \dots j]$ is repeating in $T[i \dots j+1]$. If $T[j+1-\ell \dots j+1] = \alpha^{\ell+1}$, then clearly $T[j+1-\ell \dots j] = \alpha^{\ell}$ is repeating in $T[i \dots j+1]$. If $\ell \leq |lrSuf_{i,j}|$, then $T[j+1-\ell \dots j]$ is a suffix of $T[j+1-|lrSuf_{i,j}| \dots j]$ (see Fig. 4).

Thus $\#occ_{T[i\dots j+1]}(T[j+1-\ell \dots j]) \geq \#occ_{T[i\dots j]}(T[j+1-\ell \dots j]) \geq \#occ_{T[i\dots j]}(T[j+1-|lrSuf_{i,j}| \dots j]) \geq 2$.

Next, we consider MUSs to be added when appending $T[j+1]$ to $T[i \dots j]$, which are *not* suffixes of $T[i \dots j+1]$.

Lemma 5 *For each* $[s,t] \in \mathsf{MUS}(T[i \dots j+1])$ *with* $t \neq j+1$, *if* $[s,t] \notin \mathsf{MUS}(T[i \dots j])$ *then* $\#occ_{T[i\dots j+1]}(sqSuf_{i,j+1}) = 2$ *and* $sqSuf_{i,j+1}$ *is a proper substring of* $T[s \dots t]$.

Proof Since $[s,t] \in \mathsf{MUS}(T[i \dots j+1])$ and $t \neq j+1$, $T[s \dots t]$ is unique in $T[i \dots j]$. Moreover, since $T[s \dots t]$ is not a MUS of $T[i \dots j]$, there exists a MUS u of $T[i \dots j]$ which is a proper substring of $T[s \dots t]$. Since $T[s \dots t]$ is a MUS of $T[i \dots j+1]$, u is repeating in $T[i \dots j+1]$. Then, it follows from Lemma 3 that $u = sqSuf_{i,j+1}$ and u occurs exactly twice in $T[i \dots j+1]$.

Namely, a MUS which is not a suffix is added by appending one character only if there is a MUS to be deleted by the same operation. Moreover, such added MUSs must contain the deleted MUS.

Lemma 6 *If* $\#occ_{T[i\dots j+1]}(sqSuf_{i,j+1}) = 2$, *then there are three integers* p_l, p_s, q *such that* $i \leq p_l \leq p_s \leq q < j+1$, $T[p_s \dots q] = sqSuf_{i,j+1}$ *and* $T[p_l \dots q] = lrSuf_{i,j+1}$. *Also, the following propositions hold*:

(a) *If there is no MUS of* $T[i \dots j]$ *ending at* $q+1$, *then* $[p_s, q+1] \in \mathsf{MUS}(T[i \dots j+1])$.

(b) *If there is no MUS of* $T[i \dots j]$ *starting at* $p_l - 1$ *and* $p_l \geq i+1$, *then* $[p_l - 1, q] \in \mathsf{MUS}(T[i \dots j+1])$.

Proof Since $\#occ_{T[i\dots j+1]}(sqSuf_{i,j+1}) = 2$, it follows from Lemma 1 that $\#occ_{T[i\dots j+1]}(lrSuf_{i,j+1}) = 2$ and $sqSuf_{i,j+1}$ is a suffix of $lrSuf_{i,j+1}$. Hence, the ending positions of the

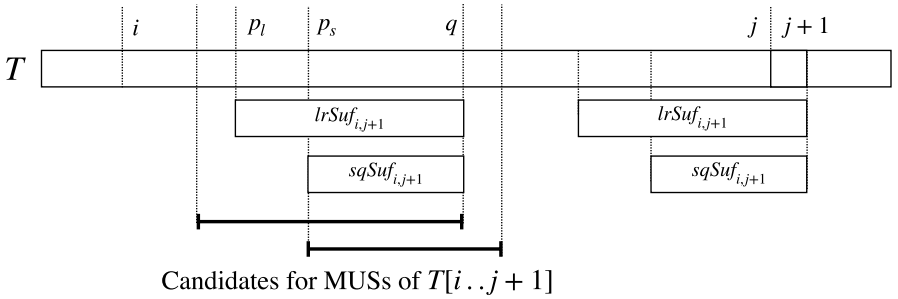


Fig. 5 Illustration of the situation when $sqSuf_{i,j+1}$ is repeating in $T[i \dots j+1]$. In this situation, $[p_l - 1, q]$ and $[p_s, q+1]$ are the only candidates for MUSs in $\mathsf{MUS}(T[i \dots j+1]) \backslash \mathsf{MUS}(T[i \dots j])$ each of which is not a suffix of $T[i \dots j+1]$

occurrences of $sqSuf_{i,j+1}$ in $T[i \dots j]$ and that of $lrSuf_{i,j+1}$ in $T[i \dots j]$ are the same (see Fig. 5). Hence, there exist indices p_s, p_l, and q such that $T[p_s \dots q] = sqSuf_{i,j+1}$ and $T[p_l \dots q] = lrSuf_{i,j+1}$.

Next, we consider MUSs to be added.

(a) First, for the sake of contradiction, assume that $T[p_s \dots q+1]$ is repeating in $T[i \dots j+1]$. By the definition, $T[p_s \dots q] = sqSuf_{i,j+1}$ occurs in $T[i \dots j+1]$ as a suffix. Also, $T[p_s \dots q]$ occurs at least twice in $T[i \dots j+1]$ as a proper prefix of $T[p_s \dots q+1]$. These implies that $\#occ_{T[i \dots j+1]}(T[p_s \dots q]) \geq 3$, however, this contradicts the definition of $sqSuf_{i,j+1}$ $(= T[p_s \dots q])$. Hence, $T[p_s \dots q+1]$ is unique in $T[i \dots j+1]$. Next, $T[p_s \dots q]$ is repeating in $T[i \dots j+1]$ by the assumption. Further, by Lemma 3, $T[p_s \dots q] = sqSuf_{i,j+1}$ is a MUS of $T[i \dots j]$ since $\#occ_{T[i \dots j+1]}(sqSuf_{i,j+1}) = 2$. Thus, $T[p_s + 1 \dots q]$ is repeating in $T[i \dots j]$. Finally, for the sake of contradiction, assume that $T[p_s + 1 \dots q+1]$ is unique in $T[i \dots j]$. Let u be a MUS of $T[i \dots j]$ which is a substring of $T[p_s + 1 \dots q+1]$. Since $T[p_s + 1 \dots q]$ is repeating in $T[i \dots j]$, the ending position of u must be $q+1$. This contradicts the assumption that there is no MUS of $T[i \dots j]$ ending at $q+1$. Thus, $T[p_s + 1 \dots q+1]$ is repeating in $T[i \dots j]$, as well as in $T[i \dots j+1]$. Therefore, $T[p_s \dots q+1]$ is a MUS of $T[i \dots j+1]$.

(b) First, for the sake of contradiction, assume that $T[p_l - 1 \dots q]$ is repeating in $T[i \dots j+1]$. From the discussion at the beginning of the proof, the starting positions of the occurrences of $lrSuf_{i,j+1}$ are p_l and $j + 2 - |lrSuf_{i,j+1}|$ (see also Fig. 5). Since $lrSuf_{i,j+1}$ is a proper suffix of $T[p_l - 1 \dots q]$ and $T[p_l - 1 \dots q]$ is repeating, the starting positions of the occurrences of $T[p_l - 1 \dots q]$ are $p_l - 1$ and $j + 1 - |lrSuf_{i,j+1}|$. Then, $T[j + 1 - |lrSuf_{i,j+1} \dots j+1]$ of length $|lrSuf_{i,j+1}| + 1$ is a repeating suffix of $T[i \dots j+1]$, however, it contradicts the definition of $lrSuf_{i,j+1}$. Thus, $T[p_l - 1 \dots q]$ is unique in $T[i \dots j+1]$. Also, by the definition, $T[p_l \dots q] = lrSuf_{i,j+1}$ is repeating in $T[i \dots j+1]$. Finally, for the sake of contradiction, assume that $T[p_l - 1 \dots q-1]$ is unique in $T[i \dots j]$. Let v be a MUS of $T[i \dots j]$ which is a substring of $T[p_l - 1 \dots q-1]$. Since $T[p_l \dots q-1]$

is repeating in $T[i \dots j]$, the starting position of v must be $p_l - 1$. This contradicts the assumption that there is no MUS of $T[i \dots j]$ starting at $p_l - 1$. Thus, $T[p_l - 1 \dots q - 1]$ is repeating in $T[i \dots j]$, as well as in $T[i \dots j + 1]$. Therefore, $T[p_l - 1 \dots q]$ is a MUS of $T[i \dots j + 1]$.

Now we have the main result of this subsection:

Theorem 1 *For any* $0 \le i \le j < n - 1$, $|\mathsf{MUS}(T[i \dots j + 1]) \bigtriangleup \mathsf{MUS}(T[i \dots j])| \le 4$ *and* $-1 \le |\mathsf{MUS}(T[i \dots j + 1])| - |\mathsf{MUS}(T[i \dots j])| \le 2$. *Furthermore, these bounds are tight for any* σ, i, j *with* $\sigma \ge 3$, $0 \le i \le j < n - 1$, *and* $j - i + 1 \ge 5$.

Proof First, we show that $|\mathsf{MUS}(T[i \dots j + 1]) \bigtriangleup \mathsf{MUS}(T[i \dots j])| \le 4$. By Lemma 3, $|\mathsf{MUS}(T[i \dots j]) \backslash \mathsf{MUS}(T[i \dots j + 1])| \le 1$. By Observation 2 and Lemma 6, $|\mathsf{MUS}(T[i \dots j + 1]) \backslash \mathsf{MUS}(T[i \dots j])| \le 3$. Thus, $|\mathsf{MUS}(T[i \dots j + 1]) \bigtriangleup \mathsf{MUS}(T[i \dots j])| = |\mathsf{MUS}(T[i \dots j + 1]) \backslash \mathsf{MUS}(T[i \dots j])| + |\mathsf{MUS}(T[i \dots j]) \backslash \mathsf{MUS}(T[i \dots j + 1])| \le 4$. Also, we show that the upper bound is tight if $\sigma \ge 3$. For an integer $k \ge 2$, we consider two strings u and u' such that $u = \mathtt{a}^k\mathtt{bcc}$ of length $k + 3 \ge 5$ and $u' = u\mathtt{b} = \mathtt{a}^k\mathtt{bccb}$ of length $k + 4 \ge 6$. Then, $\mathsf{MUS}(u) = \{[0, k - 1], [k, k], [k + 1, k + 2]\}$ and $\mathsf{MUS}(u') = \{[0, k - 1], [k - 1, k], [k, k + 1], [k + 1, k + 2], [k + 2, k + 3]\}$. Therefore, $|\mathsf{MUS}(u') \bigtriangleup \mathsf{MUS}(u)| = 4$.

Next, we show that $-1 \le |\mathsf{MUS}(T[i \dots j + 1])| - |\mathsf{MUS}(T[i \dots j])| \le 2$. By Lemma 3, it is clear that $-1 \le |\mathsf{MUS}(T[i \dots j + 1])| - |\mathsf{MUS}(T[i \dots j])|$. By Observation 2, the number of added MUSs which are suffixes of $T[i \dots j + 1]$ is at most one. Also, by Lemma 6, the number of added MUSs which are not suffixes of $T[i \dots j + 1]$ is at most two, however, if such an added MUS exists, exactly one MUS ($= sqSuf_{i,j+1}$) must be deleted (cf. Lemmas 3, 5). Therefore, $|\mathsf{MUS}(T[i \dots j + 1])| - |\mathsf{MUS}(T[i \dots j])| \le 2$. Also, we show that each bound is tight when $\sigma \ge 3$. We consider strings u and u' that are described in the case (a), and we then obtain $|\mathsf{MUS}(u')| - |\mathsf{MUS}(u)| = 2$. On the other hand, for any integer ℓ with $\ell \ge 1$, we consider two strings v and v'; $v = \mathtt{a}^\ell\mathtt{bcac}$ of length $\ell + 4 \ge 5$ and $v' = v\mathtt{a} = \mathtt{a}^\ell\mathtt{bcaca}$ of length $\ell + 5 \ge 6$. If $\ell = 1$, then $\mathsf{MUS}(v) = \{[1, 1], [2, 3], [3, 4]\}$, and $\mathsf{MUS}(v') = \{[1, 1], [3, 4]\}$. If $\ell \ge 2$, then $\mathsf{MUS}(v) = \{[0, \ell - 1], [\ell, \ell], [\ell + 1, \ell + 2], [\ell + 2, \ell + 3]\}$, and $\mathsf{MUS}(v') = \{[0, \ell - 1], [\ell, \ell], [\ell + 2, \ell + 3]\}$. Therefore, $|\mathsf{MUS}(v')| - |\mathsf{MUS}(v)| = -1$.

3.2 Changes to MUSs When Deleting the Leftmost Character

In this subsection, we consider an operation that deletes the leftmost character $T[i - 1]$ from $T[i - 1 \dots j]$. Basically, we can use symmetric arguments to the previous subsection where we considered appending a character to the right of the window.

Observation 3 For each non-empty substring s of $T[i - 1 \dots j]$, $\#occ_{T[i-1\dots j]}(s) \le \#occ_{T[i\dots j]}(s) + 1$. Also, $\#occ_{T[i-1\dots i]}(s) = \#occ_{T[i\dots j]}(s) + 1$ if and only if s is a prefix of $T[i - 1 \dots j]$.

3.2.1 MUSs to be Added When Deleting the Leftmost Character

Lemma 7 *For any* $i \le s \le t \le j, [s,t] \notin \mathsf{MUS}(T[i-1 \dots j])$ *and* $[s,t] \in \mathsf{MUS}(T[i \dots j])$ *if and only if* $T[s \dots t] = sqPref_{i-1,j}$ *and* $\#occ_{T[i-1\dots j]}(sqPref_{i-1,j}) = 2$.

Proof Symmetric to the proof of Lemma 3.

3.2.2 MUSs to be Deleted When Deleting the Leftmost Character

Next, we consider MUSs to be deleted by removing $T[i-1]$ from $T[i-1 \dots j]$. If there is a MUS w of $T[i-1 \dots j]$ which is a prefix of $T[i-1 \dots j]$, clearly, w is not a MUS of $T[i \dots j]$. Then, we consider MUSs to be deleted which are *not* prefixes of $T[i-1 \dots j]$.

Lemma 8 *For each* $[s,t] \in \mathsf{MUS}(T[i-1 \dots j])$ *with* $s \neq i-1$, *if* $[s,t] \notin \mathsf{MUS}(T[i \dots j])$ *then* $\#occ_{T[i-1\dots j]}(sqPref_{i-1,j}) = 2$ *and* $sqPref_{i-1,j}$ *is a proper substring of* $T[s \dots t]$.

Proof Symmetric to the proof of Lemma 5.

Namely, when deleting the leftmost character, a MUS which is not a prefix is deleted only if an added MUS exists. Moreover, such deleted MUSs must contains the added MUS.

Lemma 9 *If* $\#occ_{T[i-1\dots j]}(sqPref_{i-1,j}) = 2$, *then following propositions hold*:

(a) *If there is a MUS* w *starting at* s *in* $T[i-1 \dots j]$, w *is not a MUS of* $T[i \dots j]$,
(b) *If there is a MUS* w' *ending at* t *in* $T[i-1 \dots j]$, w' *is not a MUS of* $T[i \dots j]$,

where $T[s \dots t] = sqPref_{i-1,j}$ and $s \neq i-1$.
Proof Symmetric to the proof of Lemma 6. See also Fig. 6 for illustration.

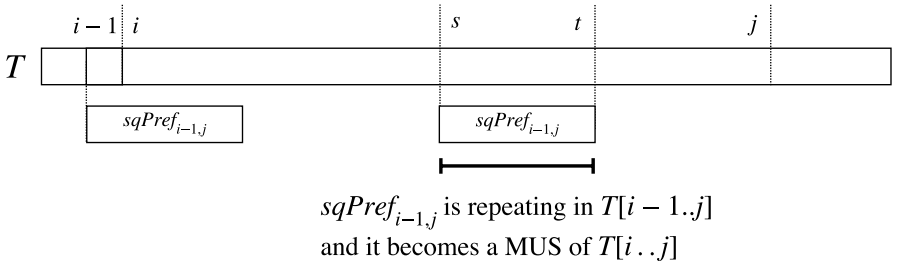


Fig. 6 Illustration of the situation when $sqPref_{i-1,j}$ is repeating in $T[i-1 \dots j]$. In this situation, $T[s \dots t] = sqPref_{i-1,j}$ is a new MUS of $T[i \dots j]$ by Lemma 7

The main result of this subsection is the following:

Theorem 2 *For any* $0 < i \leq j \leq n-1$, $|\mathsf{MUS}(T[i-1 \dots j]) \bigtriangleup \mathsf{MUS}(T[i \dots j])| \leq 4$ *and* $-1 \leq |\mathsf{MUS}(T[i-1 \dots j])| - |\mathsf{MUS}(T[i \dots j])| \leq 2$. *Furthermore, these bounds are tight for any* σ, i, j *with* $\sigma \geq 3$, $0 < i \leq j \leq n-1$, *and* $j - i + 1 \geq 5$.

Proof Symmetric to the proof of Theorem 1.

The next corollary is immediate from Theorems 1 and 2.

Corollary 1 *Let* $0 < d < n$. *For every* i *with* $0 \leq i \leq n-d-1$, $|\mathsf{MUS}(T[i \dots i+d-1]) \bigtriangleup \mathsf{MUS}(T[i+1 \dots i+d])| \in O(1)$.

4 Algorithm for Computing MUSs for a Sliding Window

This section presents our algorithm for computing MUSs for a sliding window.

4.1 Updating Suffix Tree and Its Three Loci

First, we introduce some additional notions. Since we use Ukkonen's algorithm [23] for updating the suffix tree when a new character $T[j+1]$ is appended to the right end of the window $T[i \dots j]$, we maintain the locus for $lrSuf_{i,j}$ as in [23]. Also, in order to compute the changes of MUSs, we use $sqSuf_{i,j}$ (c.f. Lemma 3, 6). Thus, we also maintain the locus for $sqSuf_{i,j}$.

The locus for $lrSuf_{i,j}$ (resp. $sqSuf_{i,j}$) in $\mathsf{STree}(T[i \dots j])$ is called the *primary active point* (resp. the *secondary active point*) and is denoted by $\mathsf{pp}_{i,j}$ (resp. $\mathsf{sp}_{i,j}$). Additionally, in order to maintain $\mathsf{sp}_{i,j}$ efficiently, we also maintain the locus for the longest suffix of $T[i \dots j]$ which occurs at least three times in $T[i \dots j]$. We call this locus the *tertiary active point* that is denoted by $\mathsf{tp}_{i,j}$. See Fig. 2 for concrete examples of these three loci in a suffix tree.

4.1.1 Appending One Character

When $T[i \dots j]$ is the empty string (the base case, where $i = 0$ and $j = -1$), we set all the three active points $\langle root, 0 \rangle$. Then we increase j, and the suffix tree grows in an online manner until $j = d-1$ using Ukkonen's algorithm. Then, for each $j > d-1$, we also increase i each time j increases, so that the sliding window is shifted to the right, by using sliding window algorithm for the suffix tree [21].

When $T[j+1]$ is appended to the right end of $T[i \dots j]$, we first update the suffix tree to $\mathsf{STree}(T[i \dots j+1])$ and compute $\mathsf{pp}_{i,j+1}$. Since $\mathsf{pp}_{i,j+1}$ coincides with the *active point*, $\mathsf{pp}_{i,j+1}$ can be found in amortized $O(\log \sigma')$ time [21].

After updating the suffix tree, we can compute $\mathsf{tp}_{i,j+1}$ and $\mathsf{sp}_{i,j+1}$ as follows:

- Traverse character $T[j+1]$ from $\mathsf{tp}_{i,j}$, and set $w \leftarrow str(\mathsf{tp}_{i,j})T[i+1]$.
- While $\#occ_{T[i\dots j+1]}(w) < 3$, set $w \leftarrow w[1\dots]$ and search for the locus p for w by using suffix links in $\mathsf{STree}(T[i\dots j+1])$.
- After breaking from the while-loop, obtain $\mathsf{tp}_{i,j+1} = \mathsf{p}$.
- $\mathsf{sp}_{i,j+1}$ equals the locus stored in p at the penultimate iteration of the while-loop.

Let us show the correctness of the above algorithm. After the first step, w is the longest suffix which possibly corresponds to $\mathsf{tp}_{i,j+1}$. In the while loop of the second step, we search for the suffix corresponding to $\mathsf{tp}_{i,j+1}$ by deleting the first characters from w one-by-one. After breaking from the while-loop, we store in w the longest suffix of $T[i\dots j+1]$ which occurs more than twice in $T[i\dots j+1]$, i.e., $\mathsf{tp}_{i,j+1} = locus(w)$. Also, by the definitions of sp and tp, $\mathsf{sp}_{i,j+1}$ is the locus for the suffix of $T[i\dots j+1]$ which is one character longer than $w = str(\mathsf{tp}_{i,j+1})$.

As is described in the above algorithm, we can locate $\mathsf{tp}_{i,j+1}$ using suffix links, in a similar manner to the active point $\mathsf{pp}_{i,j+1}$. Thus, the time cost for locating $\mathsf{tp}_{i,j+1}$ for each increasing j is amortized $O(\log \sigma')$, again by a similar argument to the active point $\mathsf{pp}_{i,j+1}$. What remains is, for each candidate w for $\mathsf{tp}_{i,j+1}$, how to quickly determine whether $\#occ_{T[i\dots j+1]}(w) < 3$ or not. In what follows, we show that it can be checked in $O(1)$ time for each candidate.

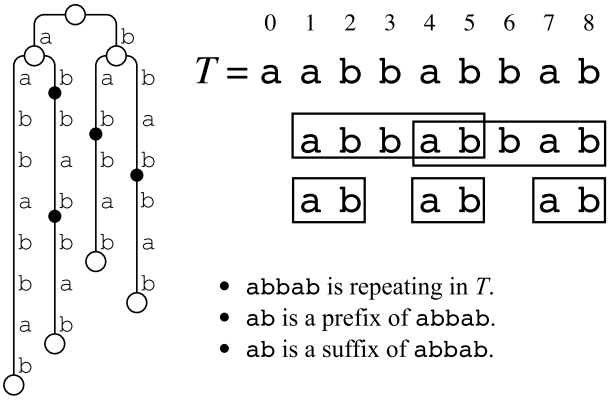
Observation 4 For each suffix s of string $T[i\dots j+1]$, let $locus(s) = \langle u, h\rangle$.

Case 1 If u is an internal node, s occurs at least three times in $T[i\dots j+1]$.
Case 2 If u is a leaf and $h = 0$, s occurs exactly once in $T[i\dots j+1]$.
Case 3 If u is a leaf and $h \neq 0$,

Case 3.1 if there is a suffix s' of $T[i\dots j+1]$ with $hed(s') = hed(s)$ which is longer than s, s occurs at least three times in $T[i\dots j+1]$ (see Fig. 7 for examples).
Case 3.2 otherwise, s occurs exactly twice in $T[i\dots j+1]$.

For any suffix s of $T[i\dots j+1]$, if we are given $locus(s) = \langle u, h\rangle$, then we can obviously determine in constant time whether s occurs at least three times in $T[i\dots j+1]$ or not, except Case 3. The next lemma allows us to determine it in constant time in Case 3 as well.

Lemma 10 *Suppose the locus* $\mathsf{pp}_{i,j+1}$ *in* $\mathsf{STree}(T[i\dots j+1])$ *is already computed. Given a leaf* ℓ *of* $\mathsf{STree}(T[i\dots j+1])$, *it can be determined in* $O(1)$ *time whether there is an implicit suffix node on the edge* $(parent(\ell), \ell)$ *and if so, the locus of the lowest implicit suffix node on* $(parent(\ell), \ell)$ *can be computed in* $O(1)$ *time.*



- abbab is repeating in T.
- ab is a prefix of abbab.
- ab is a suffix of abbab.

Fig. 7 The suffix tree of string $T = \texttt{aabbabbab}$ as an example of the Case 3.1 in Observation 4. Black circles represent implicit suffix nodes. For two suffixes $s = \texttt{ab}$ and $s' = \texttt{abbab}$ of T, $hed(s') = hed(s)$ and s occurs three times in T

Proof By Observation 4, for each leaf ℓ, the suffix corresponding to the lowest implicit suffix node on $(parent(\ell), \ell)$ occurs exactly twice in $T[i \dots j+1]$ if such an implicit suffix node exists. Let $x = lrSuf_{i,j+1}$ and $\mathsf{pp}_{i,j+1} = \langle u, h \rangle$.

If u is not a leaf, there is no implicit suffix node on the edge $(parent(\ell), \ell)$ for any leaf ℓ, since every suffix of $T[i \dots j+1]$ which is shorter than $|x|$ occurs more than twice in $T[i \dots j+1]$.

If u is a leaf, then $\#occ_{T[i \dots j+1]}(x) = 2$. Let $s = start(u)$ and $t_\ell = start(\ell)$ for each leaf ℓ. Notice that x is a border of $T[s \dots j+1]$. There are two sub-cases:

- First, we consider the case where $t_\ell < s$. Suppose that there is an implicit suffix node on $(parent(\ell), \ell)$ for the sake of contradiction. Let w be a string corresponding to the lowest implicit suffix node on $(parent(\ell), \ell)$. Then, w is a proper suffix of x, and occurs exactly twice in $T[i \dots j+1]$. Furthermore, w occurs exactly twice in $T[s \dots j+1]$ since x is a border of $T[s \dots j+1]$. However, w is also a prefix of $T[t_\ell \dots j+1]$, hence w occurs at least three times in $T[i \dots j+1]$, it is a contradiction. Thus, if $t_\ell < s$, there is no implicit suffix node on $(parent(\ell), \ell)$.
- Second, we consider the case where $t_\ell \geq s$ (see Fig. 8). In this case, $T[t_\ell \dots s + |x| - 1]$ which is a prefix of $T[t_\ell \dots j+1]$ matches the suffix of x which is $t_\ell - s$ characters shorter than x, i.e., $x[t_\ell - s \dots]$. Thus, there is an implicit suffix node on $(parent(\ell), \ell)$ if and only if $|T[t_\ell \dots s + |x| - 1]| = |x| - (t_\ell - s) > depth(parent(\ell))$. Also, if there is an implicit suffix node on $(parent(\ell), \ell)$, the locus of the lowest one is $\langle \ell, h \rangle$.

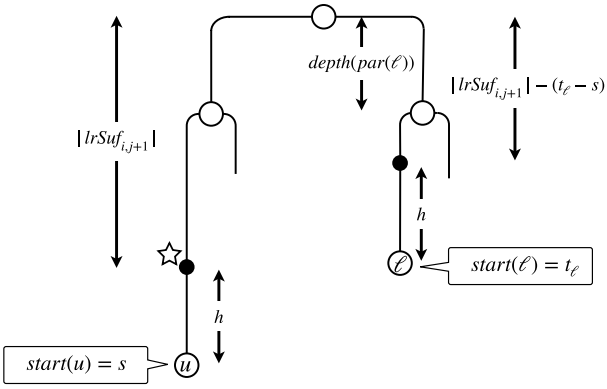


Fig. 8 For an example of Lemma 10. The situation of this figure is that each of u and ℓ is a leaf with $t_\ell = start(\ell) \geq s = start(u)$ and $|lrSuf_{i,j+1}| - (t_\ell - s) > depth(parent(\ell))$ where $\langle u, h\rangle$ represents the primary active point. Also, black nodes represent implicit suffix nodes

4.1.2 Deleting the Leftmost Character

When the leftmost character $T[i-1]$ is deleted from $T[i-1 \ldots j]$, we first update the suffix tree and compute $\mathsf{pp}_{i,j}$ by using the sliding window algorithm for the suffix tree [21]. Each pair of position pointers for the edge-labels of the suffix tree can be maintained in amortized $O(1)$ time so that these pointers always refer to positions within the current sliding window, by a simple *batch update* technique (see [21] for details). After that, we compute $\mathsf{tp}_{i,j}$ and $\mathsf{sp}_{i,j}$ in a similar way to the case of appending a new character shown previously.

It follows from the above arguments in this subsection that we can update the suffix tree and the three active points in amortized $O(\log \sigma')$ time, each time the window is shifted by one character.

4.2 Computing $sqPref_{i-1,j}$

In order to compute the changes of MUSs when the leftmost character $T[i-1]$ is deleted from $T[i-1 \ldots j]$, we use $sqPref_{i-1,j}$ (c.f. Lemmas 7 and 9) before updating the suffix tree. In this subsection, we present an efficient algorithm for computing $sqPref_{i-1,j}$. First, we consider the following cases (see Fig. 9), where ℓ is the leaf corresponding to $T[i-1 \ldots j]$:

Case A $hed(lrSuf_{i-1,j}) = \ell$.
Case B $hed(lrSuf_{i-1,j}) \neq \ell$ and $subtree(parent(\ell))$ has more than two leaves.
Case C $hed(lrSuf_{i-1,j}) \neq \ell$ and $subtree(parent(\ell))$ has exactly two leaves.

For Case A, the next lemma holds:

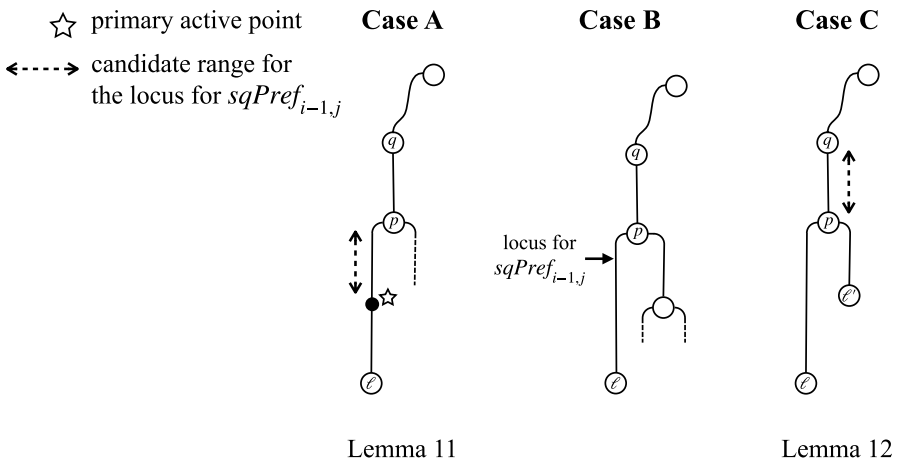


Fig. 9 Illustration for the three cases that are described in Sect. 4.2

Lemma 11 *Given* $\mathsf{STree}(T[i-1\dots j])$ *and* $\mathsf{pp}_{i-1,j}$. *Let* ℓ *be the leaf corresponding to* $T[i-1\dots j]$. *If* $\mathsf{pp}_{i-1,j}$ *is on the edge* $(parent(\ell),\ell)$, *the following propositions hold*:

(a) $occ_{T[i-1\dots j]}(sqPref_{i-1,j}) = \{i-1, j-|lrSuf_{i-1,j}|+1\}$.
(b) *If there is exactly one implicit suffix node on* $(parent(\ell),\ell)$, $sqPref_{i-1,j} = T[i-1\dots i-1+depth(parent(\ell))]$.
(c) *If there are more than one implicit suffix node on* $(parent(\ell),\ell)$, *then* $|lrSuf_{i-1,j}| > \lfloor (j-i+2)/2 \rfloor$ *and* $sqPref_{i-1,j} = T[i-1\dots j-2h+1]$, *where* $\mathsf{pp}_{i-1,j} = \langle \ell, h\rangle$.

Proof Let $\mathsf{pp}_{i-1,j} = \langle \ell, h\rangle$ and $m = |lrSuf_{i-1,j}|$.

(a) Since $\mathsf{pp}_{i-1,j}$ is on the edge $(parent(\ell),\ell)$, $sqPref_{i-1,j}$ is a prefix of $lrSuf_{i-1,j}$, and $\#occ_{T[i-1\dots j]}(lrSuf_{i-1,j}) = \#occ_{T[i-1\dots j]}(sqPref_{i-1,j}) = 2$. Therefore, we obtain that $occ_{T[i-1\dots j]}(sqPref_{i-1,j}) = occ_{T[i-1\dots j]}(lrSuf_{i-1,j}) = \{i-1, j-m+1\}$.
(b) In this case, it is clear that $sqPref_{i-1,j} = T[i-1\dots i-1+depth(parent(\ell))]$.
(c) Let $\langle \ell, h'\rangle$ be the locus of the implicit suffix node which is the lowest on the edge $(parent(\ell),\ell)$ except $\mathsf{pp}_{i-1,j}$. Also, let x be the string corresponding to the locus $\langle \ell, h'\rangle$. In this case, x occurs exactly three times in $T[i-1\dots j]$. Also, x is the longest border of $lrSuf_{i-1,j}$. Assume on the contrary that $m \le \lfloor (j-i+2)/2 \rfloor$. Then, two occurrences of $lrSuf_{i-1,j}$ in $T[i-1\dots j]$ are not overlapping, and thus $\#occ_{T[i-1\dots j]}(x) \ge 2 \times \#occ_{T[i-1\dots j]}(lrSuf_{i-1,j}) = 4$, it is a contradiction. Therefore, $m > \lfloor (j-i+2)/2 \rfloor$ (see Fig. 10). Next, we consider a relation between h and h'. By the definition, $h = |T[i-1\dots j]| - m = j-i+2-m$. Since $m > \lfloor (j-i+2)/2 \rfloor$, x matches the intersection of two occur-

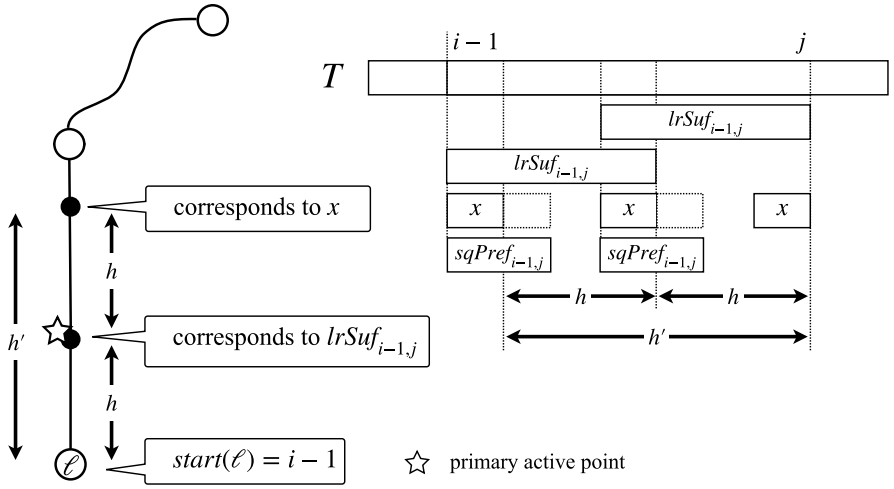


Fig. 10 Illustration for the proposition (c) in Lemma 11. For the sake of simplicity, this figure shows a simple case where there are only two implicit suffix nodes on the edge $(parent(\ell), \ell)$. However, the lemma also holds for the other cases

rences of $lrSuf_{i-1,j}$, i.e., $x = T[j-m+1 \dots i+m-2]$. Thus, $h' = |T[i-1 \dots j]| - |x| = j - i + 2 - (2m - j + i - 2) = 2(j - i + 2 - m) = 2h$. Therefore $sqPref_{i-1,j} = T[i-1 \dots j-h'+1] = T[i-1 \dots j-2h+1]$.

In Case B, it is clear that $sqPref_{i-1,j} = T[i-1 \dots i-1+depth(p)]$ since $str(p)$ occurs at least three times in $T[i-1 \dots j]$ (see Fig. 9).

For Case C, the next lemma holds:

Lemma 12 *Suppose that* $\mathsf{STree}(T[i-1 \dots j])$ *and* $\mathsf{pp}_{i-1,j}$ *have already been computed. Let* ℓ *be the leaf corresponding to* $T[i-1 \dots j]$, $p = parent(\ell)$, *and* $q = parent(p)$. *If subtree*(p) *has exactly two leaves and there are no implicit suffix nodes on any edges in subtree*(p), *then it can be determined in* $O(1)$ *time whether there is an implicit suffix node on* (q, p). *If such an implicit node exists, then the locus of the lowest implicit suffix node on* (q, p) *can be computed in* $O(1)$ *time.*

Proof Note that the suffix corresponding to the lowest implicit suffix node on (q, p) occurs exactly three times in $T[i-1 \dots j]$ from the assumptions. Let $\mathsf{pp}_{i-1,j} = \langle u, h \rangle$. If $h = 0$, the primary active point is an explicit node, and there is no implicit suffix node on every edge in $\mathsf{STree}(T[i-1 \dots j])$. If $h \neq 0$ and $u = p$, the lowest implicit suffix node on (q, p) is clearly the primary active point. Thus, in the following, we consider the situation with $u \neq p$ and $h \neq 0$.

If u is not a leaf and the number of leaves in $subtree(u)$ is greater than two, then the number of leaves in $subtree(hed(v))$ is also greater than two for each implicit suffix node v. Thus, there is no implicit suffix node on (q, p). If u is not a leaf and

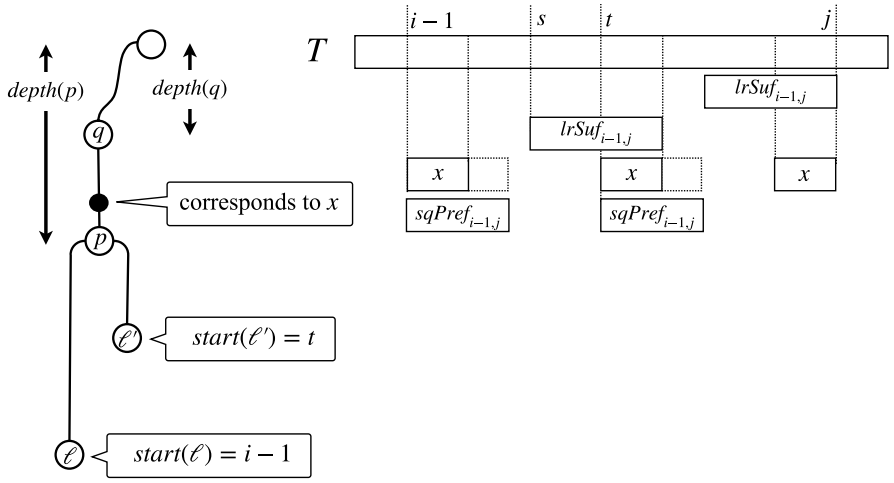


Fig. 11 Illustration for Lemma 12

the number of leaves in $subtree(u)$ is exactly two, then $lrSuf_{i-1,j}$ occurs at least three times in $T[i-1 \dots j]$ since $u \neq p$. Thus, if a suffix s of $T[i-1 \dots j]$ which is shorter than $lrSuf_{i-1,j}$ occurs as a prefix of $T[i-1 \dots j]$, $\#occ_{T[i-1\dots j]}(s) \geq 4$. Therefore, there is no implicit suffix node on (q, p).

If u is a leaf, as in the proof in Lemma 10, it can be proven that there is an implicit suffix node on (q, p) if and only if $t \geq s$ and $depth(p) > |lrSuf_{i-1,j}| - (t-s) > depth(q)$, where $s = start(u)$, $t = start(\ell')$ with ℓ' being the sibling of ℓ (see Fig. 11).

In addition, if there is an implicit suffix node on the edge (q, p), the length of the string x corresponding to the lowest implicit suffix node on the edge (q, p) is $|lrSuf_{i-1,j}| - (t-s)$, and thus, the implicit suffix node is $\langle p, depth(p) - |x| \rangle = \langle p, depth(p) - |lrSuf_{i-1,j}| + t - s \rangle$.

We can design an algorithm for computing $sqPref_{i-1,j}$ by using the above lemmas, as follows. Let ℓ be the leaf corresponding to $T[i-1 \dots j]$, $p = parent(\ell)$ and $q = parent(p)$.

In Case A. $sqPref_{i-1,j}$ is computed by Lemma 11.

In Case B. $sqPref_{i-1,j} = T[i-1 \dots i-1+depth(p)]$ and $\#occ_{T[i-1\dots j]}(sqPref_{i-1,j}) = 1$.

In Case C. We divide this case into some subcases by the existence of an implicit suffix node on edges (p, ℓ') and (q, p) where ℓ' is the sibling of ℓ. We first determine the existence of an implicit suffix node on (p, ℓ') (by Lemma 10).

- If there is an implicit suffix node on (p, ℓ'), then $sqPref_{i-1,j} = T[i-1 \dots i-1+depth(p)]$ and $\#occ_{T[i-1\dots j]}(sqPref_{i-1,j}) = 1$.
- If there is no implicit suffix node on both (p, ℓ) and (p, ℓ'), we can determine in constant time the existence of an implicit suffix node on (q, p) (by Lemma 12). If there is an implicit suffix node on (q, p), $sqPref_{i-1,j} = T[i-1 \dots depth(p)-h+1]$ and $occ_{T[i-1\dots j]}(sqPref_{i-1,j}) = \{i-1, start(\ell')\}$. Otherwise, $sqPref_{i-1,j} = T[i-1 \dots depth(q)+1]$ and $occ_{T[i.-1,j]}(sqPref_{i-1,j}) = \{i-1, start(\ell')\}$.

It follows from the above arguments in this subsection that $sqPref_{i-1,j}$ can be computed in $O(1)$ time by using the suffix tree and the (primary) active point.

4.3 Detecting MUSs to be Added/Deleted

By using the afore-mentioned lemmas in this section, we can design an efficient algorithm for detecting MUSs to be added / deleted.

4.3.1 Data Structure for Maintaining MUSs

First, we introduce a data structure for managing the set of MUSs for a sliding window. Our data structure for MUSs consists of two arrays S2E and E2S of length d each. Note that by the definition of MUSs, any MUSs cannot be nested each other. Thus, for any text position i, if a MUS starting (resp. ending) at i exists, then its ending (resp. starting) position is unique. From this fact, we can define S2E and E2S as follows:

Let $[p, p+d-1]$ be the current window. For every index i with $p \le i \le p+d-1$,

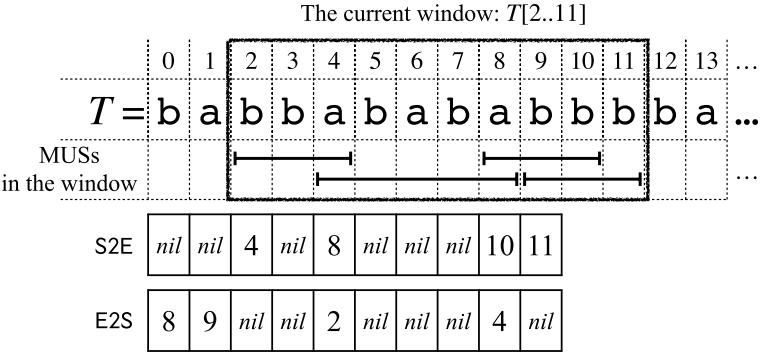


Fig. 12 A long string $T = \texttt{babbababbbba}\cdots$ and two arrays S2E and E2S. The current window is $T[2 \dots 11]$ of length $d = 10$, and the MUSs in the window are $T[2 \dots 4], T[4 \dots 8], T[8 \dots 10]$, and $T[9 \dots 11]$

$$\mathsf{S2E}[i \bmod d] = \begin{cases} e & \text{if } [i,e] \in \mathsf{MUS}(T[p \dots p+d-1]) \text{ exists,} \\ nil & \text{otherwise.} \end{cases}$$

$$\mathsf{E2S}[i \bmod d] = \begin{cases} s & \text{if } [s,i] \in \mathsf{MUS}(T[p \dots p+d-1]) \text{ exists,} \\ nil & \text{otherwise.} \end{cases}$$

Since MUSs cannot be nested each other, these arrays are uniquely defined (see Fig. 12). By using these two arrays, all the following operations for MUSs can be executed in $O(1)$ time; compute the ending/starting position of the MUS that starts/ends at a specified position, and add/remove a MUS into/from the set of MUSs. In particular, when a MUS $[s_r, e_r]$ is removed from the set of MUSs, we set $\mathsf{S2E}[s_r \bmod d] = \mathsf{E2S}[e_r \bmod d] = nil$. Also, when a MUS $[s_a, e_a]$ is added into the set of MUSs, we set $\mathsf{S2E}[s_a \bmod d] = e_a$ and $\mathsf{E2S}[e_a \bmod d] = s_a$.

4.3.2 Algorithm When Appending a Character to the Right

Assume that $\mathsf{S2E}$, $\mathsf{E2S}$ and the suffix tree of $T[i \dots j]$ are computed before reading $\gamma = T[j+1]$. Also, assume that the longest single character run β^e as a suffix of $T[i \dots j]$ is known, where $\beta = T[j]$ and $e \geq 1$.

- First, compute the length of $lrSuf_{i,j}$.
- Second, read γ, and update the suffix tree and the active points. Then, compute the lengths of $lrSuf_{i,j+1}$ and $sqSuf_{i,j+1}$. Also, update information about the run of the last character of $T[i \dots j+1]$. Specifically, if $\gamma = \beta$ then $\beta^e = \gamma^{e+1}$, and otherwise $\beta^e = \gamma^1$. If $|lrSuf_{i,j+1}| \leq |lrSuf_{i,j}|$ or $T[j+1-|lrSuf_{i,j+1}| \dots j+1] = \gamma^{|lrSuf_{i,j+1}|+1}$, add $[j+1-|lrSuf_{i,j+1}|, j+1]$ into the set of MUSs (by Lemma 4).
- If $|lrSuf_{i,j+1}| < |sqSuf_{i,j+1}|$, then terminate this step (by Lemma 5).
- Otherwise, compute p_s and q of Lemma 6 by using $\mathsf{STree}(T[i \dots j+1])$ and $\mathsf{sp}_{i,j+1}$. Then, remove $[p_s, q]$ from the set of MUSs (by Lemma 3).
- Next, if $\mathsf{E2S}[t' \bmod d] = nil$, then add $[p_s, t']$ into the set of MUSs, where $t' = q+1$. Also, if $s' \geq i$ and $\mathsf{S2E}[s' \bmod d] = nil$, then add $[s', q]$ into the set of MUSs, where $s' = q - |lrSuf_{i,j+1}|$ (by Lemma 6).
- Terminate this step.

4.3.3 Algorithm When Deleting the Leftmost Character

Assume that $\mathsf{S2E}$, $\mathsf{E2S}$ and the suffix tree of $T[i-1 \dots j]$ are computed before deleting $\alpha = T[i-1]$.

- First, compute $\#occ_{T[i-1 \dots j]}(sqPref_{i-1,j})$. If $\#occ_{T[i-1 \dots j]}(sqPref_{i-1,j}) = 2$, compute two integers s and t with $T[s \dots t] = sqPref_{i-1,j}$ and $s \neq i-1$.

- Second, delete $T[i-1]$ and update the suffix tree and the active points. If $\mathsf{S2E}[(i-1) \bmod d] \neq nil$, remove the MUS starting at $i-1$ from the set of MUSs.
- If $\#occ_{T[i-1\ldots j]}(sqPref_{i-1,j}) = 1$, terminate this step (by Lemma 8).
- Otherwise, if $\mathsf{S2E}[s \bmod d] \neq nil$, then remove the MUS starting at s from the set of MUSs. Also, if $\mathsf{E2S}[t \bmod d] \neq nil$, then remove the MUS ending at t from the set of MUSs (by Lemma 9).
- Finally, add $[s, t]$ into the set of MUSs (by Lemma 7), and terminate this step.

The main result of this section is the following:

Theorem 3 *We can maintain the set of MUSs for a sliding window of length d on a string T of length n in a total of $O(n \log \sigma')$ time and $O(d)$ working space where $\sigma' \leq d$ is the maximum number of distinct characters in every window.*

Corollary 2 *There exists an online algorithm to compute all MUSs in a string T of length n in a total of $O(n \log \sigma)$ time with $O(n)$ working space where σ is the alphabet size.*

5 Conclusions and Future Work

In this paper, we studied the problem of computing MUSs for a sliding window over a given string T of length n. We first showed combinatorial properties on MUSs for a sliding window, i.e., changes of the set of MUSs are at most constant when appending a character to the right end of the window or deleting the first character from the window. Also, we proposed an $O(n \log \sigma')$-time and $O(d)$-space algorithm to compute MUSs for a sliding window of size d over T, where $\sigma' \leq d$ is the maximum number of distinct characters in every window.

As future work, we are interested in developing a data structure for the SUS problems for a sliding window. As we described in the introduction, MUSs are heavily utilized for solving the SUS problems. Our sliding window MUS algorithm could be used as a basis for an efficient SUS query data structure for a sliding window. Also, it would be interesting to extend or generalize MUSs for a sliding window, e.g., to computing MUSs with *k-mismatches* for a sliding window. A substring of T is said to be unique with k-mismatches in T, if it is unique in T even when substituting arbitrary k characters of the substring. To the best of our knowledge, only one *deterministic* algorithm to compute unique substrings with k-mismatches is known in [10], and their algorithm runs in $O(n^2)$ time for any $k \geq 1$ in an offline manner. An interesting open question is: Can we design an online deterministic algorithm which computes MUSs with k-mismatches in sub-quadratic time?.

Acknowledgements This work was supported by JSPS KAKENHI Grant Numbers JP20J11983 (TM), JP18J10967 (YF), JP18K18002 (YN), JP17H01697 (SI), JP16H02783 (HB), JP20H04141 (HB), JP18H04098 (MT), and by JST PRESTO Grant Number JPMJPR1922 (SI).

References

1. Abedin, P., Ganguly, A., Pissis, S.P., Thankachan, S.V.: Range shortest unique substring queries. In: Brisaboa, N.R., Puglisi, S.J. (eds.) String Processing and Information Retrieval—26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11811, pp. 258–266. Springer (2019). https://doi.org/10.1007/978-3-030-32686-9_18
2. Akagi, T., Kuhara, Y., Mieno, T., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Combinatorics of minimal absent words for a sliding window. **abs/2105.08496** (2021). https://arxiv.org/abs/2105.08496
3. Belazzougui, D., Cunial, F.: Indexed matching statistics and shortest unique substrings. In: de Moura, E.S., Crochemore, M. (eds.) String Processing and Information Retrieval—21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20–22, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8799, pp. 179–190. Springer (2014). https://doi.org/10.1007/978-3-319-11918-2_18
4. Cleary, J.G., Witten, I.H.: Data compression using adaptive coding and partial string matching. IEEE Trans. Commun. **32**(4), 396–402 (1984). https://doi.org/10.1109/TCOM.1984.1096090
5. Crochemore, M., Héliou, A., Kucherov, G., Mouchard, L., Pissis, S.P., Ramusat, Y.: Absent words in a sliding window with applications. Inf. Comput. (2020). https://doi.org/10.1016/j.ic.2019.104461
6. Fiala, E.R., Greene, D.H.: Data compression with finite windows. Commun. ACM **32**(4), 490–505 (1989). https://doi.org/10.1145/63334.63341
7. Ganguly, A., Hon, W., Shah, R., Thankachan, S.V.: Space-time trade-offs for finding shortest unique substrings and maximal unique matches. Theor. Comput. Sci. **700**, 75–88 (2017). https://doi.org/10.1016/j.tcs.2017.08.002
8. Gräf, S., Nielsen, F.G.G., Kurtz, S., Huynen, M.A., Birney, E., Stunnenberg, H., Flicek, P.: Optimized design and assessment of whole genome tiling arrays. In: Proceedings 15th International Conference on Intelligent Systems for Molecular Biology (ISMB) & 6th European Conference on Computational Biology (ECCB), Vienna, Austria, July 21–25, 2007, pp. 195–204 (2007). https://doi.org/10.1093/bioinformatics/btm200
9. Haubold, B., Pierstorff, N., Möller, F., Wiehe, T.: Genome comparison without alignment using shortest unique substrings. BMC Bioinform. **6**, 123 (2005). https://doi.org/10.1186/1471-2105-6-123
10. Hon, W., Thankachan, S.V., Xu, B.: In-place algorithms for exact and approximate shortest unique substring problems. Theor. Comput. Sci. **690**, 12–25 (2017). https://doi.org/10.1016/j.tcs.2017.05.032
11. Hu, X., Pei, J., Tao, Y.: Shortest unique queries on strings. In: de Moura, E.S., Crochemore, M. (eds.) String Processing and Information Retrieval—21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20–22, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8799, pp. 161–172. Springer (2014). https://doi.org/10.1007/978-3-319-11918-2_16

12. Ileri, A.M., Külekci, M.O., Xu, B.: A simple yet time-optimal and linear-space algorithm for shortest unique substring queries. Theor. Comput. Sci. **562**, 621–633 (2015). https://doi.org/10.1016/j.tcs.2014.11.004
13. Ilie, L., Smyth, W.F.: Minimum unique substrings and maximum repeats. Fundam. Inform. **110**(1–4), 183–195 (2011). https://doi.org/10.3233/FI-2011-536
14. Larsson, N.J.: Structures of string matching and data compression. Ph.D. thesis, Lund University, Sweden (1999). http://lup.lub.lu.se/record/19255
15. Li, F., Stormo, G.D.: Selection of optimal DNA oligos for gene expression arrays. Bioinformatics **17**(11), 1067–1076 (2001). https://doi.org/10.1093/bioinformatics/17.11.1067
16. Mieno, T., Inenaga, S., Bannai, H., Takeda, M.: Shortest unique substring queries on run-length encoded strings. In: Faliszewski, P., Muscholl, A., Niedermeier, R. (eds.) 41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22–26, 2016—Kraków, Poland, LIPIcs, vol. 58, pp. 69:1–69:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.MFCS.2016.69
17. Mieno, T., Köppl, D., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Compact data structures for shortest unique substring queries. In: Brisaboa, N.R., Puglisi, S.J. (eds.) String Processing and Information Retrieval—26th International Symposium, SPIRE 2019, Segovia, Spain, October 7–9, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11811, pp. 107–123. Springer (2019). https://doi.org/10.1007/978-3-030-32686-9_8
18. Mieno, T., Kuhara, Y., Akagi, T., Fujishige, Y., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Minimal unique substrings and minimal absent words in a sliding window. In: Chatzigeorgiou, A., Dondi, R., Herodotou, H., Kapoutsis, C.A., Manolopoulos, Y., Papadopoulos, G.A., Sikora, F. (eds.) SOFSEM 2020: Theory and Practice of Computer Science—46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20–24, 2020, Proceedings, Lecture Notes in Computer Science, vol. 12011, pp. 148–160. Springer (2020). https://doi.org/10.1007/978-3-030-38919-2_13
19. Mignosi, F., Restivo, A., Sciortino, M.: Words and forbidden factors. Theor. Comput. Sci. **273**(1), 99–117 (2002)
20. Pei, J., Wu, W.C., Yeh, M.: On shortest unique substring queries. In: Jensen, C.S., Jermaine, C.M., Zhou, X. (eds.) 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8–12, 2013, pp. 937–948. IEEE Computer Society (2013). https://doi.org/10.1109/ICDE.2013.6544887
21. Senft, M.: Suffix tree for a sliding window: An overview. In: WDS, vol. 5, pp. 41–46. Matfyzpress (2005)
22. Tsuruta, K., Inenaga, S., Bannai, H., Takeda, M.: Shortest unique substrings queries in optimal time. In: Geffert, V., Preneel, B., Rovan, B., Stuller, J., Tjoa, A.M. (eds.) SOFSEM 2014: Theory and Practice of Computer Science—40th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 26–29, 2014, Proceedings, Lecture Notes in Computer Science, vol. 8327, pp. 503–513. Springer (2014). https://doi.org/10.1007/978-3-319-04298-5_44
23. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995). https://doi.org/10.1007/BF01206331
24. Zheng, J., Close, T.J., Jiang, T., Lonardi, S.: Efficient selection of unique and popular oligos for large EST databases. Bioinformatics **20**(13), 2101–2112 (2004). https://doi.org/10.1093/bioinformatics/bth210
25. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977). https://doi.org/10.1109/TIT.1977.1055714