Computing Palindromic Trees in the Sliding Window Model

Takuya Mieno[1,2,a)] Kiichi Watanabe[1,b)] Yuto Nakashima[1,c)] Shunsuke Inenaga[1,3,d)]
Hideo Bannai[4,e)] Masayuki Takeda[1,f)]

Abstract: The palindromic tree (a.k.a. eertree) for a string S of length n is a tree-like data structure that represents the set of all distinct palindromic substrings of S, using $O(n)$ space [Rubinchik and Shur, 2018]. It is known that, when S is over an alphabet of size σ and is given in an online manner, then the palindromic tree of S can be constructed in $O(n \log \sigma)$ time with $O(n)$ space. In this paper, we consider the sliding window version of the problem: For a sliding window of length at most d, we present two versions of an algorithm which maintains the palindromic tree of size $O(d)$ for every sliding window $S[i..j]$ over S, where $1 \leq j - i + 1 \leq d$. The first version works in $O(n \log \sigma')$ time with $O(d)$ space where $\sigma' \leq d$ is the maximum number of distinct characters in the windows, and the second one works in $O(n + d\sigma)$ time with $(d + 2)\sigma + O(d)$ space. We also show how our algorithms can be applied to efficient computation of minimal unique palindromic substrings (MUPS) and minimal absent palindromic words (MAPW) for a sliding window.

Keywords: string algorithms, palindromic trees, sliding window model

1. Introduction

Palindromes.

A *palindrome* is a string that reads the same forward and backward. Palindromic structures in strings have been heavily studied in the fields of string processing algorithms and combinatorics on strings [1], [8], [10], [13], [15], [19]. One of the most famous results related to palindromic structures is Manacher's algorithm [15], which computes all maximal palindromes in a given string S. Manacher's algorithm essentially computes all palindromes in S, since any palindromic substring of S is a substring of some maximal palindrome in S. Another interesting topic is enumeration of distinct palindromes in a string. It is known that any string of length n contains at most $n + 1$ distinct palindromes including the empty string [6]. Groult et al. [10] proposed an $O(n)$-time algorithm which enumerates all distinct palindromes in a given string of length n over an integer alphabet of size $\sigma = n^{O(1)}$. For the same problem in the *online* model, Kosolobov et al. [13] proposed an $O(n \log \sigma)$-time and $O(n)$-space algorithm for a general ordered alphabet. Kosolobov et al.'s algorithm is a combination of Manacher's algorithm and Ukkonen's online suffix tree construction algorithm [21]. Rubinchik and Shur [19] proposed a new data structure called *eertree*, which permits efficient access to distinct palindromes in a string *without* storing the string itself. Eertrees can be utilized for solving problems related to palindromic structures, such as the palindrome counting problem and the palindromic factorization problem [19]. The size of the eertree of S is linear in the number p_S of distinct palindromes in S [19]. It is known that p_S is at most $|S| + 1$, and that it can be much smaller than the length $|S|$ of the string, e.g., for $S = (\mathtt{abc})^{n/3}$, $p_S = 4$ since all distinct palindromes in S are $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$, and the empty string. Thus, the size of the eertree of S can be much smaller than that of the suffix tree of S which is $\Theta(n)$. Therefore, it is of significance if one can build eertrees without suffix trees. Rubinchik and Shur [19] indeed proposed an online eertree construction algorithm running in $O(n \log \sigma)$ time without suffix trees.

Recently, a concept of palindromic structures called *minimal unique palindromic substrings* (*MUPS*) is introduced by Inoue et al. [12]. A palindromic substring $w = S[i..j]$ of a string S is called a MUPS of S if w occurs in S exactly once, and $S[i+1..j-1]$ occurs at least twice in S. MUPSs are utilized for solving the *shortest unique palindromic substring* (*SUPS*) problem [12], which is motivated by an application in molecular biology. Watanabe et al. [22] proposed an algorithm to solve the SUPS problem based on the *run-length encoding* (*RLE*) version of eertrees, named $\mathtt{e^2rtre^2}$.

Sliding Window Model.

In this paper, we consider the problems of computing palindromic structures for the *sliding window* model. The sliding window model is a natural generalization of the online model, and the assumptions of this model are natural when we need to pro-

1 Department of Informatics, Kyushu University
2 Japan Society for the Promotion of Science
3 PRESTO, Japan Science and Technology Agency
4 M&D Data Science Center, Tokyo Medical and Dental University
a) takuya.mieno@inf.kyushu-u.ac.jp
b) kiichi.watanabe@inf.kyushu-u.ac.jp
c) yuto.nakashima@inf.kyushu-u.ac.jp
d) inenaga@inf.kyushu-u.ac.jp
e) hdbn.dsc@tmd.ac.jp
f) takeda@inf.kyushu-u.ac.jp

cess a massive or a streaming string data with a limited memory space. A typical and classical application to the sliding window model is data compression, such as Lempel-Ziv 77 (the original version) [23] and PPM [3]. Note that sliding-window Lempel-Ziv 77 is an immediate application of suffix trees for a sliding window, which can be maintained in $O(n \log \sigma')$ time using $O(d)$ space [7], [14], [20] where d is the size of the window and $\sigma' \leq d$ is the maximum number of distinct characters in every window. Recently, several algorithms for computing substrings for a sliding window with certain interesting properties are proposed: For instance, Crochemore et al. [4] introduced the problem of computing *minimal absent words* (*MAWs*) for a sliding window, and proposed an $O(n\sigma)$-time and $O(d\sigma)$-space algorithm using suffix trees for a sliding window. Mieno et al. [16] proposed an algorithm for computing *minimal unique substrings* (*MUSs*) [11] for a sliding window, in $O(n \log \sigma')$-time and $O(d)$ space, again based on suffix trees for a sliding window.

Our Contributions.

In this paper, we consider the problem of maintaining eertrees for the sliding window model, that is, given a string S of length n and a *window* of a fixed size d, we maintain eertrees of substrings $S[i..i+d-1]$ for incremental $i = 0, 1, \ldots, n-d$. Also, we consider the problem of maintaining MUPSs for a sliding window. In addition, we introduce a new concept of palindromic structures called *minimal absent palindromic words* (*MAPW*), and consider the problem of maintaining MAPWs for a sliding window. A string w is called a MAPW of string S if w is a palindrome, w does not occur in S, and $w[1..|w|-2]$ occurs in S. MAPWs can be seen as a palindromic version of the notion of MAWs, which are extensively studied in the fields of string processing and bioinformatics [2], [5], [9], [17], [18].

In this paper, we propose an algorithm which maintains eertrees for a sliding window in a total of $O(n \log \sigma')$ time using $O(d)$ space. We then give an alternative eertree construction algorithm for a sliding window which runs in $O(n + d\sigma)$ time with $(d+2)\sigma + O(d)$ space. As applications to the aforementioned result, we propose an algorithm which maintains MUPSs for a sliding window in a total of $O(n \log \sigma')$ time using $O(d)$ space, and an algorithm which maintains MAPWs for a sliding window in a total of $O(n + d\sigma)$ time using $O(d\sigma)$ space. We emphasize that our algorithms are stand-alone in the sense that they do not use suffix trees, while the majority of existing efficient sliding window algorithms (see above) make heavy use of suffix trees.

All proofs of lemmas and theorems are omitted due to lack of space.

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 S is denoted by $|S|$. The *empty string* ε is the string of length 0. If $S = xyz$, then x, y, and z are called a *prefix*, *substring*, and *suffix* of S, respectively. They are called a *proper* prefix, *proper* substring, and *proper* suffix of S if $x \neq S$, $y \neq S$, and $z \neq S$, respectively. If a non-empty string x is both a proper prefix and a proper suffix of S, then x is called a *border* of S. For any $0 \leq i \leq |S|-1$, $S[i]$ denotes the i-th character of S. For any $0 \leq i \leq j \leq |S|-1$, $S[i..j]$ denotes the substring of S starting at position i and ending at position j, i.e., $S[i..j] = S[i]S[i+1]\cdots S[j]$. For convenience, $S[i..j] = \varepsilon$ for any $i > j$. A string S is called a *palindrome* if $S[i] = S[|S|-i-1]$ for every $0 \leq i \leq |S|-1$. Note that the empty string is a palindrome. A substring $S[i..j]$ of S is said to be a *palindromic substring* of S if $S[i..j]$ is a palindrome. The *center* of a palindromic substring $S[i..j]$ of S is $\frac{i+j}{2}$. A palindromic substring $S[i..j]$ of S is *maximal* if $i = 0$, $j = |S|-1$, or $S[i-1..j+1]$ is not a palindrome. We denote by $lpp(S)$ (resp. $lps(S)$) the longest palindromic prefix (resp. suffix) of S. We denote by $\mathsf{DPal}(S)$ the set of all distinct palindromes in S. It is known that $|\mathsf{DPal}(S)| \leq |S|+1$ [6]. For any non-empty strings S and w, w is said to be *unique* in S if w occurs in S exactly once. Also, w is said to be *repeating* in S if w occurs in S at least twice. For convenience, we define that the empty string ε is repeating in any string. In what follows, we consider an arbitrary fixed string S of length $n > 0$.

2.2 Eertrees (Palindromic Trees)

The *eertree* of S denoted by $\mathsf{eertree}(S)$ is a tree-like data structure that enables us to efficiently access each of the distinct palindromes in S [19]. The $\mathsf{eertree}(S)$ consists of m *ordinary nodes* and two auxiliary nodes, denoted $\mathtt{0}$-*node* and -1-*node*, where $m = |\mathsf{DPal}(S)| - 1$. Each ordinary node corresponds to each element of $\mathsf{DPal}(S) \setminus \{\varepsilon\}$. For each ordinary node v, we denote by $pal(v)$ the palindrome which corresponds to v, and by $len(v)$ its length. Conversely, for each non-empty palindromic substring p of S, we denote by $node(p)$ the node which corresponds to the palindrome p. Namely, $node(pal(v)) = v$ for each ordinary node v. For convenience, we define $pal(\mathtt{0}\text{-node}) = pal(-1\text{-node}) = \varepsilon$, $len(\mathtt{0}\text{-node}) = 0$, and $len(-1\text{-node}) = -1$. For any nodes u, v in $\mathsf{eertree}(S)$, there is an edge (u, v) if and only if $len(u) + 2 = len(v)$ and $pal(u) = pal(v)[1..len(v)-2]$. Each edge (u, v) is labeled by a character $pal(v)[0]$. Also, each node v in $\mathsf{eertree}(S)$ has a *suffix link* denoted by $slink(v)$. For each node v in $\mathsf{eertree}(S)$ with $len(v) \geq 2$, $slink(v)$ points to the node corresponding to the longest palindromic proper suffix of $pal(v)$. For each node v in $\mathsf{eertree}(S)$ with $len(v) = 1$, $slink(v)$ points to the $\mathtt{0}$-node. Also, $slink(\mathtt{0}\text{-node}) = -1\text{-node}$ and $slink(-1\text{-node}) = -1\text{-node}$. For each node v in $\mathsf{eertree}(S)$, $inSL(v) = |\{u \mid slink(u) = v\}|$ denotes the number of incoming suffix links of v. See Fig. 1 for an example of $\mathsf{eertree}(S)$.

Note that each node v does not store the string $pal(v)$ explicitly. Instead, we can obtain $pal(v)$ by traversing edges backward, from v to the root, since $pal(u) = c\,pal(u')c$ for each node u with $|pal(u)| \geq 2$ where u' is the parent of u and c is the label of the edge (u', u). Each node only stores pointers to its children and a constant number of integers. Thus, the size of $\mathsf{eertree}(S)$ is linear in the number of nodes, i.e., $O(|\mathsf{DPal}(S)|)$. It is known that $\mathsf{eertree}(S)$ can be constructed in $O(n \log \sigma)$ time for any string S given in an online manner [19].

2.3 Sliding Window

We formalize sliding windows over string S. For each time $t = 0, 1, \ldots,$ we consider the substring $S[i_t..j_t]$ called *the win-*

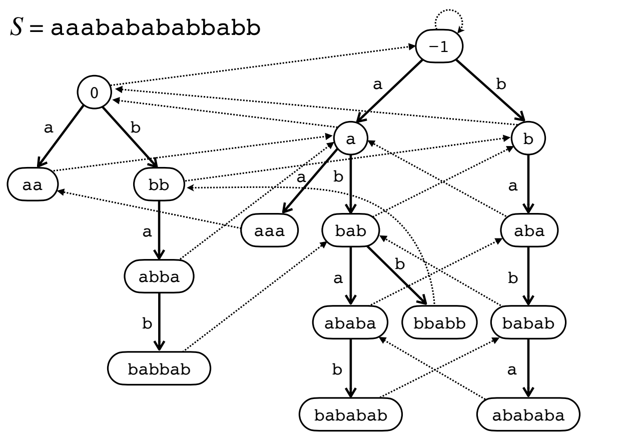

Fig. 1 The eertree of S = aaabababababbabb. The solid and broken arrows represent edges and suffix links, respectively. Note that $pal(v)$ is written inside each node v in this figure, however, it is for only explanation. Namely, each node does not explicitly store the corresponding string.

dow at time t. The windows must satisfy the following conditions: (1) $i_0 = j_0 = 0$ for the initial window at time 0; and (2) $0 \leq i_t \leq j_t \leq n-1$ and either $(i_t, j_t) = (i_{t-1}+1, j_{t-1})$ or $(i_t, j_t) = (i_{t-1}, j_{t-1}+1)$ for every time $t > 0$. In other words, the second condition means that we can either *delete* the leftmost character from the current window, or *append* a character to the right end of the current window at each time.

Given a sequence of windows (or equivalently, a sequence of delete / append operations), the aim of our sliding window model is processing the windows in space proportional to the size of each window. This paper mainly deals with the problem of maintaining eertrees with respect to a sequence of windows over a given string S.

3. Combinatorial Properties on Palindromes for a Sliding Window

In this section, we show some combinatorial properties on palindromes for a sliding window, which is helpful for designing efficient algorithms to maintain eertrees for a sliding window. Since the nodes of the eertree of a string represent all distinct palindromes in the string, we obtain the next lemma.

Lemma 1. *There is a node ℓ in* $\mathsf{eertree}(S[i-1..j-1])$ *to be removed when the leftmost character $S[i-1]$ is deleted from $S[i-1..j-1]$ if and only if (A) $pal(\ell)$ is unique in $S[i-1..j-1]$, (B) $pal(\ell) = lpp(S[i-1..j-1])$, and (C) ℓ is a leaf node.*

Namely, when the leftmost character of the window is deleted, at most one leaf will be removed from the eertree. Also, in order to detect such a leaf, we need to compute the longest palindromic prefix of each window and to determine its uniqueness. In the following, we show some combinatorial properties on unique palindromes and the longest palindromic prefix for a sliding window.

3.1 Unique Palindromes for a Sliding Window

A palindromic substring w of string S is said to be *left-maximal* in S if there is no palindromic substring of S which contains w as a proper suffix. See Fig. 2 for examples. If a palindrome w is not left-maximal in S, then for some palindrome w', w is a proper suffix and prefix of w', i.e., w is not unique in S. In other words, any unique palindromic substring must be left-maximal.

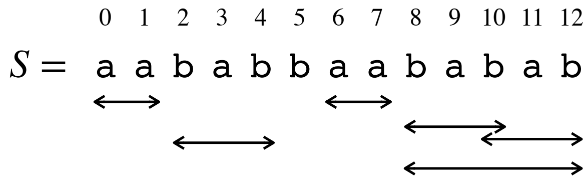

Fig. 2 For string S = aababbaababab, its palindromic substring aa is left-maximal in S. On the other hand, bab is not left-maximal in S since there is a palindromic substring $S[8..12]$ = babab of S which contains bab as a proper suffix.

Lemma 2. *For any time t and any left-maximal palindromic substring w of $S[i_t..j_t]$, there exists time $t' < t$ which satisfies one of the followings:*
(1) the longest palindromic suffix of $S[i_{t'}..j_{t'}]$ is w, or
(2) the longest palindromic suffix of $lpp(S[i_{t'}..j_{t'}])$ is w.

As mentioned before, any unique palindrome is left-maximal. Thus, Lemma 2 is useful for maintaining uniqueness of palindromes for a sliding window.

3.2 Longest Prefix Palindrome for a Sliding Window

Next, we consider the longest palindromic prefixes for sliding windows.

Lemma 3. *Let w be the longest palindromic prefix of the window $S[i_t..j_t]$ at time t. There exists time $t' < t$ which satisfies one of the followings:*
(1) the longest palindromic suffix of $S[i_{t'}..j_{t'}]$ is w, or
(2) the longest palindromic suffix of $lpp(S[i_{t'}..j_{t'}])$ is w.

4. Eertree for a Sliding Window

In this section, we show how to update a given eertree when we shift the sliding window to the right by one character. Sliding a given window consists of two operations: *deleting* the leftmost character and *appending* a character to the right end. Namely, when the eertree of $S[i-1..j-1]$ is given, we first compute the eertree of $S[i..j-1]$ (deleting the leftmost character $S[i-1]$), and then, compute the eertree of $S[i..j]$ (appending a character $S[j]$). To update the eertree when a character is appended, we can apply Rubinchik and Shur's algorithm [19] which constructs the eertree of a given string in an online manner. In this section, we propose new additional data structures and algorithms which update the eertree when the leftmost character is deleted.

We emphasize that our algorithms work for any valid*1 sequence of windows of arbitrary lengths. However, for simplicity, we consider the case where a fix-sized window of length d shifts to the right one-by-one throughout this section.

4.1 Auxiliary Data Structures for Detecting the Node to be Deleted

We introduce auxiliary data structures for computing the longest palindromic prefixes and for determining uniqueness of palindromes.

*1 C.f., the definition of sliding windows in Section 2.3.

For Computing the Longest Palindromic Prefix.

Let $prefPal[0..d-1]$ be a cyclic array of size d such that $prefPal[i_t \bmod d]$ stores the node which corresponds to the longest palindromic prefix of the window $S[i_t..j_t]$ at each time t. Namely, for every time t, $prefPal[i_t \bmod d] = node(lpp(S[i_t..j_t]))$ holds.

For Determining Uniqueness of a Palindrome.

For each ordinary node v in $\mathsf{eertree}(S[i..j])$, let $rm_{i,j}(v)$ be the starting position of the rightmost occurrence of $pal(v)$ in $S[i..j]$. Further let $srm_{i,j}(v)$ be the starting position of the second rightmost occurrence of $pal(v)$ in $S[i..j]$ if such a position exists, and otherwise, $srm_{i,j}(v) = -1$. Throughout the computation of the eertree for a sliding window, for each node v of $\mathsf{eertree}(S[i..j])$ we keep the following invariant $BegPair_{i,j}(v)$ which consists of two fields *first* and *second* such that:

$$BegPair_{i,j}(v).first = \begin{cases} rm_{i,j}(v) & \\ \quad \text{if } inSL(v) = 0, & \\ \text{An occurrence of } pal(v) \text{ in } S[0..j] & \\ \quad \text{otherwise.} & \end{cases}$$

$$BegPair_{i,j}(v).second = \begin{cases} srm_{i,j}(v) & \\ \quad \text{if } inSL(v) = 0 \text{ and } srm_{i,j}(v) \neq -1, & \\ \text{An occurrence of } pal(v) \text{ in } S[0..j] & \\ \quad \text{otherwise.} & \end{cases}$$

Namely, $BegPair_{i,j}(v)$ stores the rightmost and second rightmost occurrences of $pal(v)$ in $S[i..j]$ when $inSL(v) = 0$, if such occurrences exist. Otherwise, it temporarily stores some pair of integers, however, it will never be referred in our algorithms. In other words, we employ a kind of lazy maintenance of the rightmost and second rightmost occurrences of $pal(v)$ in $S[i..j]$ that suffices for our purpose. See Fig. 3 for an example of $BegPair_{i,j}(v)$.

The next lemma states that given a node v, we can determine if $pal(v)$ is unique or not by checking the incoming suffix links of v and $BegPair_{i,j}(v)$.

Lemma 4. *Let v be any node in $\mathsf{eertree}(S[i..j])$. Then, $pal(v)$ is unique in $S[i..j]$ if and only if $inSL(v) = 0$ and $BegPair_{i,j}(v).second < i$.*

Next, we introduce our algorithms to maintain *prefPal* and *BegPair* for a sliding window which utilizes combinatorial properties shown in Section 3.

4.2 Maintaining the Auxiliary Data Structures

First, in Algorithm 1, we show subroutine *update_bp* which updates the member variable $v.bp$ of a given node v where $v.bp$ must be kept equal to $BegPair_{i_t,j_t}(v)$ at each time t. It will be called in the algorithms that we show later.

Next, we show our algorithms for updating data structures when we slide the given window. When the leftmost character $S[i-1]$ is deleted from $S[i-1..j-1]$, our data structures are updated by Algorithm 2. Also, when a character $S[j]$ is appended to $S[i..j-1]$, our data structures are updated by Algorithm 3.

Time Complexities.

Clearly, Algorithm 1 runs in constant time. In Algorithm 2, all lines except for Line 13 can be processed in constant time. Thus,

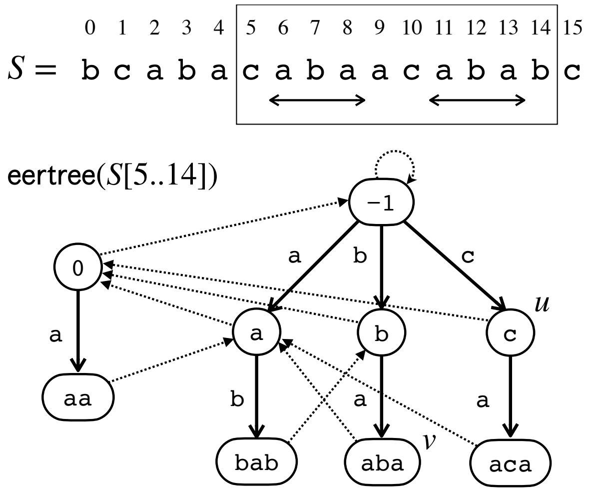

Fig. 3 Examples for $BegPair_{i,j}(v)$. For string S = bcabacabaacababc and window $[5, 14]$, the $\mathsf{eertree}(S[5..14])$ is depicted. Consider node v in $\mathsf{eertree}(S[5..14])$ with $pal(v)$ = aba. The rightmost and the second rightmost occurrences of aba in the window $S[5..14]$ are 11 and 6. Namely, $rm_{5,14}(v) = 11$ and $srm_{5,14}(v) = 6$. Further, $inSL(v) = 0$, and thus, $BegPair_{5,14}(v) = (11, 6)$. Also, for node u in $\mathsf{eertree}(S[5..14])$ with $pal(u)$ = c, $BegPair_{5,14}(u) = (10, 5)$ since $rm_{5,14}(u) = 10$, $srm_{5,14}(u) = 5$, and $inSL(u) = 0$. When the leftmost character $S[5]$ = c is deleted from the window $S[5..14]$, $srm_{5,14}(u)$ changes to -1. However, $BegPair_{6,14}(u) = BegPair_{5,14}(u) = (10, 5)$ is allowed since $BegPair_{6,14}(u).second = 5 < 6$ is a valid value for our invariant. Namely, we do not have to update $BegPair_{6,14}(u)$ explicitly when deleting the leftmost character $S[5]$ from $S[5..14]$.

Algorithm 1 *update_bp*(v, x).

Require: Node v, and a starting position x of $pal(v)$.
Ensure: Update $v.bp$ appropriately with respect to the position x.
if $x > v.bp.first$ **then**
  $v.bp.second \leftarrow v.bp.first$
  $v.bp.first \leftarrow x$
else if $x > v.bp.second$ **then**
  $v.bp.second \leftarrow x$
end if

Algorithm 2 Update *BegPair* and *prefPal* when the leftmost character is deleted.

Require: $lpsuf = node(lps(S[i-1..j-1]))$, and
  $v.bp = BegPair_{i-1,j-1}(v)$ for each node v in $\mathsf{eertree}(S[i-1..j-1])$.
Ensure: $lpsuf = node(lps(S[i..j-1]))$, and
  $v.bp = BegPair_{i,j-1}(v)$ for each node v in $\mathsf{eertree}(S[i..j-1])$.
1: $lppref \leftarrow prefPal[i-1]$ \\ $pal(lppref) = lpp(S[i-1..j-1])$
2: **if** $lppref = lpsuf$ **then**
3:   $lpsuf \leftarrow slink(lpsuf)$ \\ `For the case the window is a palindrome`
4: **end if**
5: $q \leftarrow slink(lppref)$
6: $inSL(q) \leftarrow inSL(q) - 1$
7: $x \leftarrow i - 1 + len(lppref) - len(q)$ \\ x `is a starting position of` $pal(q)$
8: $update\_bp(q, x)$
9: **if** $len(q) > len(prefPal[x])$ **then**
10:   $prefPal[x] = q$
11: **end if**
12: **if** $inSL(lppref) = 0$ and $lppref.bp.second < i - 1$. **then**
13:   Remove node *lppref* from the eertree
14: **end if**

Algorithm 3 Update *BegPair* and *prefPal* when a character is appended.

Require: $lpsuf = node(lps(S[i..j-1]))$, $S[j]$, and
$v.bp = BegPair_{i,j-1}(v)$ for each node v in $\mathsf{eertree}(S[i..j-1])$.
Ensure: $lpsuf = node(lps(S[i..j]))$, and
$v.bp = BegPair_{i,j}(v)$ for each node v in $\mathsf{eertree}(S[i..j])$.

1: Compute $lps(S[i..j])$ and overwrite $lpsuf \leftarrow node(lps(S[i..j]))$
2: **if** $lpsuf$ does not exist in $\mathsf{eertree}(S[i..j-1])$ **then**
3:     Add new node $lpsuf$ to the eertree
4: **end if**
5: $y \leftarrow j - len(lpsuf) + 1$ \\ y is a starting position of $lps(S[i..j])$
6: $update\_bp(lpsuf, y)$
7: $prefPal[y] \leftarrow lpsuf$

the total running time of Algorithm 2 is dominated by Line 13, i.e., $O(\log\sigma')$. In Algorithm 3, the first four lines can be processed in amortized $O(\log\sigma')$ time by using the online construction algorithm [19]. Also, the remaining lines can be processed in constant time, and thus, the total running time of Algorithm 3 is amortized $O(\log\sigma')$.

Correctness.

First, it is clear that Algorithm 1 runs correctly.

Next, let us consider the correctness of Algorithm 2. Let us first consider a special case when the window $S[i-1..j-1]$ itself is a palindrome. Then, we need to update *lpsuf*, which will be used in Algorithm 3. Lines 2–3 of Algorithm 2 captures such a case. Next, we show that *BegPair* for all nodes are updated correctly. By Lemma 2, it is suffice to update $v.bp$ for every node v where $pal(v)$ is left-maximal. Let q be is the node corresponding to the longest palindromic suffix of $lppref = lpp(S[i-1..j-1])$. Then, it is suffice to update $q.bp$ since the node q is the only candidate for a node whose corresponding palindrome to be left-maximal *just* in this step. Thus, we update only $q.bp$ in Lines 5–8, if it is needed. Further, we show that *prefPal* is also updated correctly. By Lemma 3, the longest palindromic prefix of a window must be the longest palindromic suffix of either some window or the longest palindromic prefix of some window. The palindrome $pal(q)$ is the only one which is to be such a palindrome *just* in this step. Thus, $prefPal[x]$ is the only candidate which may be updated in this step where x is the starting position of the occurrence of $pal(q)$ which is the longest palindromic suffix of $lpp(S[i-1, j-1])$. Therefore, it is suffice to update $prefPal[x]$ and update it if necessary (Lines 9–11). Line 12 determines the uniqueness of $lpp(S[i-1..j-1])$ correctly by using Lemma 4, and if it is unique, then the corresponding node *lppref* is removed (in Line 13).

Finally, consider the correctness of Algorithm 3. When a character is appended, we first check the new longest palindromic suffix, and create a new node corresponding to the palindrome if necessary. These procedures in Lines 1–4 are correctly performed by running the online construction algorithm [19]. Let y be the starting position of the longest palindromic suffix of the window $S[i..j]$. The palindrome $lps(S[i..j])$ is the only candidate for a palindrome to be left-maximal *just* in this step, and thus, by Lemma 2, it is suffice to update $lpsuf.bp$ in this step. Also, by Lemma 3, *lpsuf* is the only candidate for the node that we need to newly store into *prefPal* in this step. At this moment, *lpsuf* is clearly the longest palindrome starting at position y. Thus, we set $prefPal[y] \leftarrow lpsuf$ (Line 7).

To summarize this section, we obtain the following theorem.

Theorem 1. *We can maintain eertrees for a sliding window in a total of $O(n\log\sigma')$ time using $O(d') + d$ space where $d' \le d$ be the maximum number of distinct palindromes in all windows.*

By applying a subtle modification to the above algorithm, we obtain another variant of the algorithm (Theorem 2 below) which is faster than Theorem 1 when $d'\sigma < n\log\sigma'$, but using additional $(d'+1)\sigma$ space.

Theorem 2. *We can maintain eertrees for a sliding window in a total of $O(n+d'\sigma)$ time using $(d'+1)\sigma + O(d') + d \in O(d\sigma)$ space.*

5. Applications of Eertrees for a Sliding Window

In this section, we apply our sliding-window eertree algorithm of Section 4 to computing minimal unique palindromic substrings and minimal absent palindromic words for a sliding window.

5.1 Computing Minimal Unique Palindromic Substrings for a Sliding Window

A substring $S[i..j]$ of S is called a *minimal unique palindromic substring (MUPS)* of S if and only if $S[i..j]$ is a palindrome, $S[i..j]$ is unique in S, and $S[i+1..j-1]$ is repeating in S. We denote $\mathsf{MUPS}(S)$ the set of intervals corresponding to MUPSs of S, i.e., $\mathsf{MUPS}(S) = \{[i, j] \mid S[i..j] \text{ is a MUPS of } S\}$. For example, palindromic substring $S[9..13] = \mathtt{bbabb}$ of string $S = \mathtt{aaababababbabb}$ is a MUPS of S since $S[9..13] = \mathtt{bbabb}$ is unique in S and $S[10..12] = \mathtt{bab}$ is repeating in S.

Now, we show Lemma 5 which states a relationship between eertrees and MUPSs. Then, in Lemma 6, we show that all MUPS can be computed using eertrees in an offline manner.

Lemma 5. *A string w is a MUPS of S if and only if there is a node v in $\mathsf{eertree}(S)$ such that $pal(v) = w$, $pal(v)$ is unique in S and $pal(u)$ is repeating in S, where u is the parent of v.*

Lemma 6. *Given $\mathsf{eertree}(S)$, we can compute $\mathsf{MUPS}(S)$ in $O(|\mathsf{DPal}(S)|)$ time.*

Moreover, we can efficiently maintain MUPSs for a sliding window.

Theorem 3. *We can maintain the set of MUPSs for a sliding window in a total of $O(n\log\sigma')$ time using $O(d)$ space.*

5.2 Computing Minimal Absent Palindromic Words for a Sliding Window

A string w is called a *minimal absent palindromic word* (*MAPW*) of string S if and only if w is a palindrome, w does not occur in S, and $w[1..|w|-2]$ occurs in S. For example, palindrome $w = \mathtt{aabbaa}$ is a MAPW of string $S = \mathtt{aaababababbabb}$ since w does not occur in S and $w[1..|w|-2] = \mathtt{abba}$ occurs in S at position 8. For a relation between MAPWs and eertrees, the next lemma holds.

Lemma 7. *For any non-empty string $w \in \Sigma^*$, w is a MAPW of a string S if and only if there is a node u in $\mathsf{eertree}(S)$ such that $pal(u) = w[1..|w|-2]$, $len(u) = |w|-2$, and u does not have an edge labeled by $w[0]$.*

In order to maintain the set of MAPWs on top of $\mathsf{eertree}(S)$, we store an array M_v of size σ for each node v in $\mathsf{eertree}(S)$ where $M_v[c] = 0$ if v has an edge labeled by c and $M_v[c] = 1$ otherwise. By Lemma 7, $M_v[c] = 1$ iff $c\,pal(v)c$ is a MAPW of S. It is easy to see that M_v for all nodes v (i.e., all MAPWs of S) can be computed by traversing $\mathsf{eertree}(S)$ only once. Thus, the next corollary holds.

Corollary 1. *The number of MAPWs of S is at most* $(|\mathsf{DPal}(S)| + 1)\sigma$. *Also, given* $\mathsf{eertree}(S)$*, the set of MAPWs of S can be computed in* $O(|\mathsf{DPal}(S)|\sigma)$ *time.*

Also, we can maintain MAPWs for a sliding window by applying Theorem 2.

Corollary 2. *We can maintain the set of MAPWs for a sliding window in a total of $O(n + d\sigma)$ time using $O(d\sigma)$ space.*

References

[1] Bannai, H., Gagie, T., Inenaga, S., Kärkkäinen, J., Kempa, D., Piatkowski, M. and Sugimoto, S.: Diverse Palindromic Factorization is NP-Complete, *Int. J. Found. Comput. Sci.*, Vol. 29, No. 2, pp. 143–164 (2018).

[2] Chairungsee, S. and Crochemore, M.: Using minimal absent words to build phylogeny, *Theor. Comput. Sci.*, Vol. 450, pp. 109–116 (2012).

[3] Cleary, J. G. and Witten, I. H.: Data Compression Using Adaptive Coding and Partial String Matching, *IEEE Trans. Communications*, Vol. 32, No. 4, pp. 396–402 (1984).

[4] Crochemore, M., Héliou, A., Kucherov, G., Mouchard, L., Pissis, S. P. and Ramusat, Y.: Absent words in a sliding window with applications, *Inf. Comput.*, Vol. 270 (2020).

[5] Crochemore, M., Mignosi, F., Restivo, A. and Salemi, S.: Data compression using antidictionaries, *Proceedings of the IEEE*, Vol. 88, No. 11, pp. 1756–1768 (2000).

[6] Droubay, X., Justin, J. and Pirillo, G.: Episturmian words and some constructions of de Luca and Rauzy, *Theor. Comput. Sci.*, Vol. 255, No. 1-2, pp. 539–553 (2001).

[7] Fiala, E. R. and Greene, D. H.: Data Compression with Finite Windows, *Commun. ACM*, Vol. 32, No. 4, pp. 490–505 (1989).

[8] Fici, G., Gagie, T., Kärkkäinen, J. and Kempa, D.: A subquadratic algorithm for minimum palindromic factorization, *J. Discrete Algorithms*, Vol. 28, pp. 41–48 (2014).

[9] Fujishige, Y., Tsujimaru, Y., Inenaga, S., Bannai, H. and Takeda, M.: Computing DAWGs and Minimal Absent Words in Linear Time for Integer Alphabets, *Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science (MFCS '16)*, pp. 38:1–38:14 (2016).

[10] Groult, R., Prieur, É. and Richomme, G.: Counting distinct palindromes in a word in linear time, *Inf. Process. Lett.*, Vol. 110, No. 20, pp. 908–912 (2010).

[11] Ilie, L. and Smyth, W. F.: Minimum Unique Substrings and Maximum Repeats, *Fundam. Inform.*, Vol. 110, No. 1-4, pp. 183–195 (2011).

[12] Inoue, H., Nakashima, Y., Mieno, T., Inenaga, S., Bannai, H. and Takeda, M.: Algorithms and combinatorial properties on shortest unique palindromic substrings, *J. Discrete Algorithms*, Vol. 52-53, pp. 122–132 (2018).

[13] Kosolobov, D., Rubinchik, M. and Shur, A. M.: Finding Distinct Subpalindromes Online, *Proceedings of the Prague Stringology Conference 2013*, pp. 63–69 (2013).

[14] Larsson, N. J.: Extended Application of Suffix Trees to Data Compression, *Proceedings of the 6th Data Compression Conference (DCC '96)*, pp. 190–199 (1996).

[15] Manacher, G. K.: A New Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String, *J. ACM*, Vol. 22, No. 3, pp. 346–351 (1975).

[16] Mieno, T., Kuhara, Y., Akagi, T., Fujishige, Y., Nakashima, Y., Inenaga, S., Bannai, H. and Takeda, M.: Minimal Unique Substrings and Minimal Absent Words in a Sliding Window, *Proceedings of the 46th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM '20)*, pp. 148–160 (2020).

[17] Mignosi, F., Restivo, A. and Sciortino, M.: Words and forbidden factors, *Theor. Comput. Sci.*, Vol. 273, No. 1-2, pp. 99–117 (2002).

[18] Ota, T. and Morita, H.: On a universal antidictionary coding for stationary ergodic sources with finite alphabet, *Proceedings of the International Symposium on Information Theory and its Applications (ISITA '14)*, pp. 294–298 (2014).

[19] Rubinchik, M. and Shur, A. M.: EERTREE: An efficient data structure for processing palindromes in strings, *Eur. J. Comb.*, Vol. 68, pp. 249–265 (2018).

[20] Senft, M.: Suffix tree for a sliding window: An overview, *WDS '05*, pp. 41–46 (2005).

[21] Ukkonen, E.: On-Line Construction of Suffix Trees, *Algorithmica*, Vol. 14, No. 3, pp. 249–260 (1995).

[22] Watanabe, K., Nakashima, Y., Inenaga, S., Bannai, H. and Takeda, M.: Fast Algorithms for the Shortest Unique Palindromic Substring Problem on Run-Length Encoded Strings, *Theory Comput. Syst.* (2020).

[23] Ziv, J. and Lempel, A.: A universal algorithm for sequential data compression, *IEEE Trans. Inf. Theory*, Vol. 23, No. 3, pp. 337–343 (1977).

