Data structures for computing unique palindromes in static and non-static strings

Takuya Mieno[1] and Mitsuru Funakoshi[2,3]

[1]Department of Computer and Network Engineering, University of Electro-Communications, Japan
[2]Department of Informatics, Kyushu University, Japan
[3]Japan Society for the Promotion of Science

Abstract

A palindromic substring $T[i..j]$ of a string T is said to be a shortest unique palindromic substring (SUPS) in T for an interval $[p, q]$ if $T[i..j]$ is a shortest palindromic substring such that $T[i..j]$ occurs only once in T, and $[i, j]$ contains $[p, q]$. The SUPS problem is, given a string T of length n, to construct a data structure that can compute all the SUPSs for any given query interval. It is known that any SUPS query can be answered in $O(\alpha)$ time after $O(n)$-time preprocessing, where α is the number of SUPSs to output [Inoue et al., 2018]. In this paper, we first show that α is at most 4, and the upper bound is tight. We also show that the total sum of lengths of minimal unique palindromic substrings of string T, which is strongly related to SUPSs, is $O(n)$. Then, we present the first $O(n)$-bits data structures that can answer any SUPS query in constant time. Also, we present an algorithm to solve the SUPS problem for a sliding window that can answer any query in $O(\log \log W)$ time and update data structures in amortized $O(\log \sigma + \log \log W)$ time, where W is the size of the window, and σ is the alphabet size. Furthermore, we consider the SUPS problem in the after-edit model and present an efficient algorithm. Namely, we present an algorithm that uses $O(n)$ time for preprocessing and answers any k SUPS queries in $O(\log n \log \log n + k \log \log n)$ time after single character substitution. Finally, as a by-product, we propose a fully-dynamic data structure for range minimum queries (RmQs) with a constraint where the width of each query range is limited to poly-logarithmic. The constrained RmQ data structure can answer such a query in constant time and support a single-element edit operation in amortized constant time.

1 Introduction

A substring $T[i..j]$ of a string T is said to be a *shortest unique palindromic substring* (in short, *SUPS*) for an interval $[p, q]$ if $T[i..j]$ is the shortest substring such that $T[i..j]$ is a palindrome, $T[i..j]$ occurs only once in T, and the occurrence contains $[p, q]$, i.e., $[p, q] \subseteq [i, j]$. The notion of SUPS was introduced by Inoue et al. [22] in 2018, motivated by bioinformatics: for example, in DNA/RNA sequences, the presence of unique palindromic sequences can affect the immunostimulatory activities of oligonucleotides [25, 37]. Given a string T of length n, the SUPS problem is to construct a data structure that can compute all SUPSs for any given query interval. We call this general problem the interval SUPS problem because queries are intervals. When a query interval is restricted to a single position (i.e., $p = q$), the SUPS problem is called the point SUPS problem. The (interval) SUPS problem was formalized by Inoue et al. [22], and they showed that all SUPSs for a query interval can be enumerated in $O(\alpha)$ time after $O(n)$-time preprocessing, where α is the number of SUPSs to output. Watanabe et al. [36] considered the SUPS problem

1

on run-length encoded strings to reduce the space usage. They proposed an $O(r)$-space data structure that can enumerate all SUPSs for a query interval in $O(\sqrt{\log r / \log \log r} + \alpha)$ time where r is the size of the run-length encoded string, which satisfies $r \leq n$.

Both of the above results are for a static string. It is a natural question whether we can compute SUPSs efficiently in a *dynamic* string. In fact, since DNA sequences contain errors and change dynamically, it is worthwhile to consider them in a dynamic string setting. However, there is no research for solving the SUPS problem on a dynamic string to the best of our knowledge. Thus, in this paper, as a first step to designing dynamic algorithms, we consider the problem on two *semi-dynamic* models: the *sliding-window* model and the *after-edit* model. The sliding-window model aims to compute some objects (e.g., data structure, compressed string, statistics, and so on) w.r.t. the window sliding over the input string left to right. The after-edit model aims to compute some objects w.r.t. the string after applying an edit operation to the input string. Edit operations are given as queries, and they are discarded after processing the query. As related work, the set of *minimal unique palindromic substrings* (*MUPSs*) can be maintained efficiently in the sliding-window model [30]. Also, the set of MUPSs can be updated efficiently in the after-edit model [16]. Since MUPSs are strongly related to SUPSs, we utilize the above known results for MUPSs as black boxes.

Contributions of this paper are summarized as follows:

Section 4: Combinatorial properties on SUPSs and MUPSs.

- We show that the number α of SUPSs for any single interval is at most four, and the upper bound is tight even for binary strings,
- We show that the sum of lengths of MUPSs of a string of length n is $O(n)$.

Section 5: Compact SUPS data structures for static strings.

- We propose a compact data structure of size $3n + 2m + o(n)$ bits that can answer any *interval SUPS* query in constant time where n is the length of the input string and m is the number of MUPSs of the input string.
- We propose a compact data structure of size $3n + m + o(n)$ bits that can answer any *point SUPS* query in constant time.

Section 6: Algorithms for SUPS problem for semi-dynamic strings.

- We propose a data structure of size $O(W)$ for the sliding-window SUPS problem that supports SUPS query in $O(\log \log W)$ time and each window-shift in amortized $O(\log \sigma + \log \log W)$ time where W is the size of the window and σ is the alphabet size.
- We propose a data structure of size $O(n)$ for the after-substitution SUPS problem that can answer any after-substitution-SUPS query in amortized $O(\log n \log \log n)$ time after some character in the input string is substituted with another character.

Furthermore, as a by-product, we propose a fully-dynamic data structure for the range minimum query (RmQ) in which the width of each query range is in $O(\mathsf{polylog}(n))$. The data structure can answer such a query in constant time and update in (amortized) constant time for any single-element edit operation. Note that, for the original RmQ without any additional constraint, it is known that we need $\Omega(\log n / \log \log n)$ time for answering a query when $O(\mathsf{polylog}(n))$ updating time is allowed [2].

Related Work. A typical application to the sliding-window model is string compression such as LZ77 [38] and PPM [11]. The sliding-window LZ77 compression is based on the sliding-window suffix tree [13, 26, 33, 31]. Also, the sliding-window suffix tree can be applied to compute minimal absent words [12] and minimal unique substrings [29], which are significant concepts for bioinformatics, in the sliding-window model. Recently, the sliding-window palindromic tree was proposed, and it can be applied to compute MUPSs in the sliding-window model [30].

The after-edit model was formalized by Amir et al. [5] in 2017. They tackled the problem of computing the longest common substring for two strings in the after-edit model, and proposed

an algorithm running in poly-logarithmic time. Afterward, Abedin et al. [1] improved the complexities. Also, the problems of computing the longest Lyndon substring [34], the longest palindrome [17], and the set of MUPSs [16] were considered in the after-edit model.

As for more general settings, Amir et al. [6] proposed a fully-dynamic algorithm for computing the longest common substrings for two dynamic strings. They also developed a general (probabilistic) scheme for dynamic problems on strings and applied it to the computation of the longest Lyndon substring and the longest palindrome in a dynamic string. Besides that, there are several studies for dynamic settings (e.g., [19, 4, 9]). In particular, a fully-dynamic and deterministic algorithm for computing the longest palindrome was shown in [3].

Paper Organization. The rest of this paper is organized as follows: In Section 2, we give basic notations and algorithmic tools. In Section 3, we review a known static SUPS data structure proposed by Inoue at al. [22], which is the basis of most of our methods. In Section 4, we investigate combinatorial properties on SUPSs and MUPSs. We show the tight bounds on the maximum number of SUPSs for an interval and an upper bound of the total sum of the lengths of MUPSs. Further, we propose a simple algorithm for point SUPS queries based on the combinatorial results. In Section 5, we propose the first compact data structures that can answer any SUPS query in output-sensitive time. In Section 6, we consider how to update SUPS data structures in semi-dynamic settings and propose efficient algorithms. Finally, in Section 7, we conclude our paper and discuss future work.

2 Preliminaries

2.1 Strings

Let Σ be an alphabet. 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. For each i with $1 \leq i \leq |T|$, we denote by $T[i]$ the i-th character of T. If $T = xyz$, then x, y, and z are called a prefix, substring, and suffix of T, respectively. For each i, j with $1 \leq i \leq j \leq |T|$, we denote by $T[i..j]$ the substring of T starting at position i and ending at position j. For convenience, let $T[i'..j'] = \varepsilon$ for any i', j' with $i' > j'$. We say that string w is unique in T if w occurs only once in T. For convenience, we define that the empty string ε is not unique in any string. For a non-empty string T and a positive integer p with $p \leq |T|$, the integer p is a period of T if $T[i] = T[i + p]$ holds for every i with $1 \leq i \leq |T| - p$. We also say that T has a period p if p is a period of T. For convenience, the empty string ε is defined to have period 0.

Let T^R denote the reversal of a string T, i.e., $T[i] = T^R[n - i + 1]$ for every i with $1 \leq i \leq n$. A string P is called a palindrome if $P = P^R$ holds. A palindrome P is called an even-palindrome (resp., odd-palindrome) if $|P|$ is even (resp., odd). The length-$\lceil |P|/2 \rceil$ prefix (resp., suffix) of a palindrome P is called the left arm (resp., right arm) of P. Let $w = T[i..j]$ be a palindromic substring of T. The center of w is $(i + j)/2$ and is denoted by $\mathrm{center}(w)$. For a non-negative integer ℓ, $x = T[i - \ell..j + \ell]$ is said to be an expansion of w if $1 \leq i - \ell \leq j + \ell \leq n$ and x is a palindrome. Also, $T[i + \ell..j - \ell]$ is said to be a contraction of w. Further, if $i = 1$, $j = n$, or $T[i - 1] \neq T[j + 1]$, then w is said to be a maximal palindrome.

A palindromic substring $u = T[i..j]$ of a string T is said to be a minimal unique palindromic substring (MUPS) in T if u is unique in T and $T[i + 1..j - 1]$ is not unique in T. A palindromic substring $v = T[i..j]$ of a string T is said to be a shortest unique palindromic substring (SUPS) for an interval $[p, q]$ in T if v is unique in T, the occurrence contains interval $[p, q]$, and any shorter palindromic substring of T that contains $[p, q]$ is not unique in T. We denote by $\mathsf{SUPS}_T([p, q])$ the set of SUPSs for $[p, q]$. Note that all palindromes in $\mathsf{SUPS}_T([p, q])$ have equal lengths. See also Fig. 1 for examples. The interval SUPS problem is, given a string T for preprocessing and an interval $[p, q]$ as a query, to compute $\mathsf{SUPS}_T([p, q])$. We sometimes simply refer to the interval SUPS problem as the SUPS problem. When every query interval is restricted to a single position (i.e., $p = q$), the SUPS problem is called the point SUPS problem.
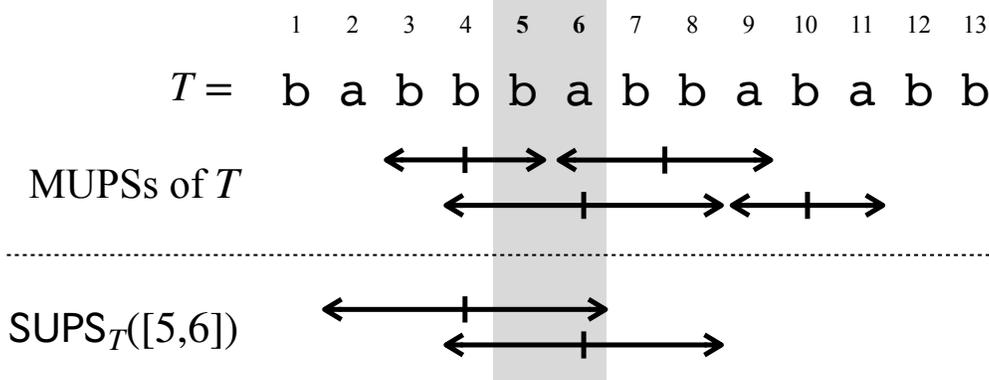
Figure 1: MUPSs of string $T = \mathtt{babbbabbababb}$ are $\mathtt{bbb}$, $\mathtt{bbabb}$, $\mathtt{abba}$, and $\mathtt{aba}$. SUPSs for interval $[5, 6]$ in T are $T[2..6] = \mathtt{abbba}$ and $T[4..8] = \mathtt{bbabb}$. The first SUPS $T[2..6]$ is an expansion of MUPS $T[3..5] = \mathtt{bbb}$, and the second SUPS $T[4..8]$ itself is a MUPS.

In what follows, we fix a string T of arbitrary length $n > 0$ over an integer alphabet of size $\sigma = O(\mathsf{poly}(n))$. Also, our computational model is a standard word RAM model of word size $\Omega(\log n)$.

2.2 Periodicity of Palindromic Suffixes

In this subsection, we recall some properties regarding the periodicity of palindromic suffixes of $T[1..i]$ that we use. Let $\mathbf{S}_i = \{w_1, \ldots, w_g\}$ be the set of lengths of palindromic suffixes of $T[1..i]$, where g is the number of palindromic suffixes of $T[1..i]$ and $w_{k-1} < w_k$ for $2 \leq k \leq g$. Let d_k be the progression difference for w_k, i.e., $d_k = w_k - w_{k-1}$ for $2 \leq k \leq g$. For convenience, let $d_1 = 0$.

Then, the following results are known:

Lemma 1 ([7, 18, 28]).

(A) *For any $1 \leq k < g$, $d_{k+1} \geq d_k$.*

(B) *For any $1 < k < g$, if $d_{k+1} \neq d_k$, then $d_{k+1} \geq d_k + d_{k-1}$.*

(C) $\mathbf{S}_i$ *can be represented by $O(\log i)$ arithmetic progressions, where each arithmetic progression is a tuple $\langle s, d, f \rangle$ representing the sequence $s, s + d, \ldots, s + (f - 1)d$ of lengths of f palindromic suffixes with common difference d.*

(D) *The common difference d is the smallest period of all palindromic suffixes of $T[1..i]$ whose length belongs to the arithmetic progression $\langle s, d, f \rangle$.*

Lemma 2 ([28]). *For any $\langle s, d, f \rangle$ from the representation of palindromic suffixes of $T[1..i]$, there exist palindromes u, v and a non-negative integer q, such that $(uv)^{f+q-1}u$ (resp. $(uv)^q u$) is the longest (resp. shortest) palindromic suffix represented by $\langle s, d, f \rangle$ with $|uv| = d$.*

For a position i, divide the set of palindromic suffixes of $T[1..i]$ into groups $G_1, G_2, \ldots, G_\pi$ w.r.t. their smallest periods in increasing order. For each $G_r = \langle s_r, d_r, f_r \rangle$ with $1 \leq r \leq \pi$, let u_r and v_r be the corresponding variables used in Lemma 2.

Let $lcp(x, y)$ for strings x and y denote the length of the longest common prefix of x and y. Also, let $\alpha_r = lcp((T[1..i - s_r])^R, T[i+1..n])$ and $\beta_r = lcp((T[1..i - s_r - (f_r - 1)d_r])^R, T[i+1..n])$ if $f_r \geq 2$. Namely, $s_r + 2\alpha_r$ (resp. $s_r + (f_r - 1)d_r + 2\beta_r$) is the length of the maximal expansion of the shortest (resp. longest) palindrome of $\langle s_r, d_r, f_r \rangle$. See also Figure 2 for a concrete example of G_r.

If $f_r = 1$, i.e., if G_r is a singleton, let β_r be the length of the maximal expansion of the palindrome. In addition, if $r \geq 2$, let $\alpha_r = \beta_{r-1}$. Note that $G_1 = \{\varepsilon\}$ is a singleton, i.e.,
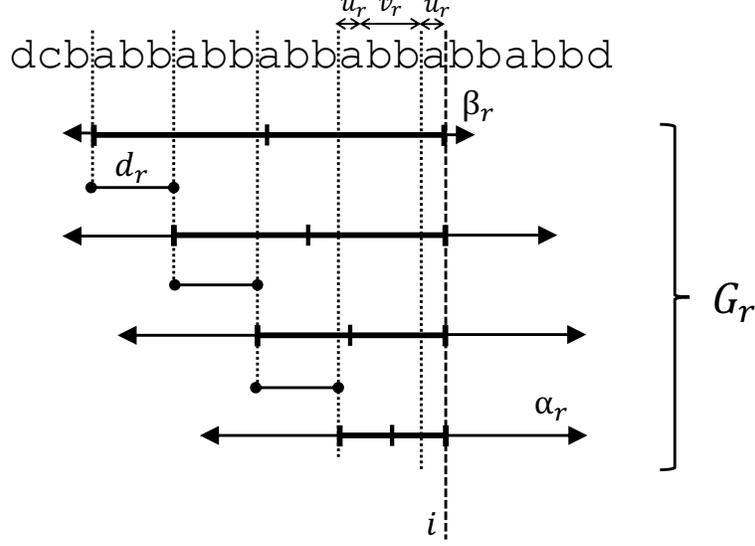
4

Figure 2: Example for a group G_r, with string `dcbabbabbabbabbabbabbd`. Here palindromic suffixes $(\text{abb})^j\text{a}$ with $1 \leq j \leq 4$ ending at i belong to $G_r = \langle 4, 3, 4 \rangle$. Also, $u_r = \text{a}$, $v_r = \text{bb}$, $\alpha_r = 5$, and $\beta_r = 1$ hold. Note that this G_r is of type 2.

$f_1 = 1$. For convenience, let $\alpha_1 = 0$. Each group G_r is said to be of *type 1* (resp. *type 2*) if $\alpha_r < d_r$ (resp. $\alpha_r \geq d_r$). Let k be the largest index of groups such that G_k is of type 2. Since $\alpha_1 = d_1 = 0$ always holds, G_1 is of type 2 and k is well-defined. In the proof of Claim (1) and (2) of [17], the following statements are also proven:

Corollary 1. *The following two statements hold:*

(A) Any expansion of a palindrome Q in group G_r for every r with $1 \leq r \leq k-1$ except for $u_k v_k u_k$ and u_k cannot be longer than $|Q| + 2d_k$.

(B) For every r with $k+1 \leq r \leq \pi - 2$ and every palindrome P in group G_r, the length of any expansion of P is at most $|P| + 2\beta_r < |P| + 2d_{r+1}$.

From Corollary 1, the following lemmas can be obtained.

Lemma 3. *Any expansion of a palindrome Q in group G_r for every r with $1 \leq r \leq k-1$ except for $u_k v_k u_k$ and u_k cannot be unique in T.*

Proof. Since G_k is of type 2, the length of the maximal expansion of s_k is $|s_k| + 2\alpha_k \geq |s_k| + 2d_k$. From statement (A) of Corollary 1, any expansion of a palindrome Q, whose center is differ from the center of s_k, is contained by the maximal expansion of s_k. Namely, any expansion of Q occurs at least twice in s_k. □

Lemma 4. *For every r with $k + 1 \leq r \leq \pi - 2$ and every palindrome P in group G_r, any expansion of P is not longer than s_{r+2}, that is, the length of the shortest palindrome in G_{r+2}.*

Proof. From the formula of statement (B) of Corollary 1 and the definition of the progression differences d_{r+1} and d_{r+2}, $|P| + 2\beta_r < |P| + 2d_{r+1} < |P| + d_{r+1} + d_{r+2} \leq s_{r+1} + d_{r+2} \leq s_{r+2}$ holds. □

2.3 Tools

Longest Common Extension. A longest common extension (in short, LCE) query on string T is, given two integers i, j with $1 \leq i, j \leq n$, to compute the length of the longest common prefix (LCP) of two suffixes $T[i..n]$ and $T[j..n]$. It is known (e.g., [21]) that any LCE

5

query can be answered in constant time using the suffix tree of $T\$$ enhanced with a lowest common ancestor data structure, where $\$$ is a special character that is not in Σ. Once we build an LCE data structure on string $T\#T^R\$$, we can answer any LCE query in any direction on T in constant time, where $\# \notin \Sigma$ is another special character. Namely, we can compute in constant time the length of (1) the LCP length of any two suffixes of T, (2) the LCP length of the reverses of any two prefixes of T, and (3) the LCP length of any suffix of T and the reverse of any prefix of T. We call such a data structure a bidirectional LCE data structure.

RmQ, Predecessor and Successor. A range minimum query (RmQ) on integer array A is, given two indices i, j on A with $i \le j$, to compute an arbitrary index k such that $A[k]$ is the minimum value from $A[i..j]$.

A predecessor (resp., successor) query on non-decreasing integer array B is, given an integer x, to compute the maximum (resp., minimum) value that is smaller (resp., greater) than x. We use the famous van Emde Boas tree data structure [35] to answer predecessor/successor queries. Namely, we can answer a query and update the data structure in $O(\log \log U)$ time on a dynamic array, where U is the universe size. Also, the space complexity is $O(U)$. Throughout this paper, we will only apply this result to the case of $U = n$.

2.4 Our Problems

This paper handles SUPS problems under two variants of semi-dynamic models: the sliding-window model and the after-edit model. The sliding-window SUPS problem is to support any sequence of queries that consists of the following:

- pushback(c): append a character c to the right end of the string.
- pop(): remove the first character from the string.
- sups($[p, q]$): output all SUPSs of the string for an interval $[p, q]$.

The after-substitution SUPS problem on a string T is, given a substitution operation and a sequence of intervals, to compute SUPSs of T' for each interval where T' is the string after applying the substitution *to the original string* T. Note that each substitution is discarded after the corresponding SUPS queries are answered.

From the point of view of how the string changes, there are differences between the above two problems. On the one hand, in the sliding-window SUPS problem, the string can be changed dynamically under the constraints of positions to be edited. On the other hand, in the after-substitution SUPS problem, any position of the string can be changed, however, the string returns to the original one after the SUPS queries are answered.

3 Inoue et al.'s Static SUPS Data Structure

The SUPS data structure proposed in [22] consists of the following:

- the set of MUPSs of T,
- the set of maximal palindromes of T
  (or a bidirectional LCE data structure on T, instead),
- a successor data structure on the starting positions of MUPSs,
- a predecessor data structure on the ending positions of MUPSs, and
- an RmQ data structure on the array MUPSlen of lengths of MUPSs sorted by their starting positions[1].

---

[1]Since MUPSs cannot be nested [22], they are also sorted by their ending positions.
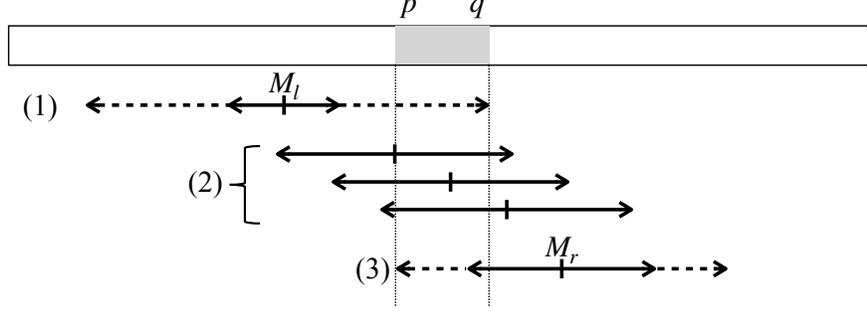
Figure 3: Illustration for candidates for SUPSs for interval $[p, q]$. Solid arrows represent MUPSs, and dashed arrows represent expansions of MUPSs. Note that dashed arrows may not be palindromes in general.

Given a query interval $[p, q]$, we can compute all SUPSs for $[p, q]$ as follows: First, we determine whether the interval $[p, q]$ covers some MUPS or not by querying the predecessor of q on the ending positions of MUPSs and the successor of p on the starting positions of MUPSs. If $[p, q]$ covers only one MUPS, the shortest expansion of the MUPS that covers $[p, q]$ is the only SUPS for $[p, q]$ if such a palindrome exists, and there are no SUPSs for $[p, q]$ otherwise. If $[p, q]$ covers more than one MUPS, then there are no SUPSs for $[p, q]$ since any SUPS covers exactly one MUPS [22]. Otherwise, i.e., if $[p, q]$ covers no MUPSs, all SUPSs are categorized into following three types (see also Fig. 3):

(1) an expansion of the rightmost MUPS M_l which ends before q,

(2) a MUPS which covers $[p, q]$, or

(3) an expansion of the leftmost MUPS M_r which begins after p.

We call M_l and M_r the left-neighbor MUPS and the right-neighbor MUPS of the interval $[p, q]$, respectively. We can find M_l by querying the predecessor of q. Also, we can determine whether there is an expansion of M_l, which covers $[p, q]$ by looking at the maximal palindrome centered at $c_l = \text{center}(M_l)$. Precisely, if the maximal palindrome centered at c_l covers $[p, q]$, its shortest contraction covering $[p, q]$ is the only candidate of type (1). Otherwise, there is no SUPS of type (1). We emphasize that we can also compute the maximal palindrome centered at c_l by querying bidirectional LCE once, without the precomputed maximal palindromes. The candidate of type (3) can be treated similarly. Finally, all SUPSs of type (2) can be computed by querying RmQ recursively on the array MUPSlen. Let $[b_i, e_i], \ldots, [b_j, e_j] \in \text{MUPS}(T)$ be all MUPSs covering $[p, q]$. Recall that such range $[i, j]$ of MUPSs can be detected by querying predecessor and successor (see above). We query the RmQ on MUPSlen for the range $[i, j]$, and obtain the index k that is the answer of the RmQ. Namely, $[b_k, e_k]$ is a shortest one within $[b_i, e_i], \ldots, [b_j, e_j]$. Then, we further query the RmQ on MUPSlen for the ranges $[i, k-1]$ and $[k+1, j]$, and repeat it recursively while obtained MUPS is a SUPS for $[p, q]$. The above operations can be done in time linear in the number of SUPSs to output by using linear size data structures of predecessor, successor, and RmQ.

4 Combinatorial Properties on SUPSs

4.1 Tight Bounds on Maximum Number of SUPSs for Single Query

In this subsection, we prove the following theorem:

Theorem 1. *For any interval $[p, q]$ over T, the inequality $|\text{SUPS}_T([p, q])| \leq 4$ holds. Also, this upper bound is tight even for binary strings.*
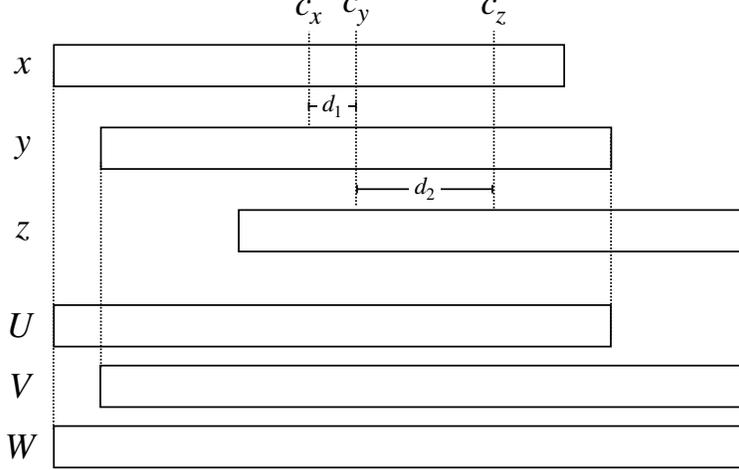
Figure 4: Illustration for three overlapped palindromes x, y, and z.

To begin with, we prove Lemma 5. Roughly speaking, Lemma 5 states that a periodic structure occurs when two palindromes overlap enough. Essentially, Lemma 5 has been proven in Lemma 3.3 of [7]. However, we restate the proposition in a form that is convenient for us and show it for completeness.

Lemma 5. *Let $x = T[i..i + \ell - 1]$ and $y = T[j..j + \ell - 1]$ be palindromic substrings of length ℓ of string T with $i < j$. If x and y overlap, then $z = T[i..j + \ell - 1]$ has period $2d$ where d is the distance between their center positions.*

Proof. Firstly, $d = (2j + \ell - 1)/2 - (2i + \ell - 1)/2 = j - i$ holds. Let $z = rst$ where $r = T[i..j - 1]$, $s = T[j..i + \ell - 1]$, and $t = T[i + \ell..j]$. Since x and y are palindromes, s^R is a prefix of x and a suffix of y. Namely, s^R is both a prefix and a suffix of z, and thus, z has period $|z| - |s^R| = (j - i + \ell) - (i - j + \ell) = 2(j - i) = 2d$. $\square$

Now we are ready to prove Theorem 1.

Proof of Theorem 1. Let us focus on the SUPSs whose center is at most p. We assume we have at least three such SUPSs for a single query interval $[p, q]$ and show that this leads to a contradiction. Let ℓ be the length of the SUPSs. Let x, y, and z be the SUPSs from left to right, and let c_x, c_y, and c_z be their center positions (see also Fig. 4). Further let $d_1 = c_y - c_x$ and $d_2 = c_z - c_y$. Since the center positions of the three SUPSs are at most p, and they cover the position p, they overlap at least $\ell/2$ each other. Namely, $d_1 \leq \ell/2$, $d_2 \leq \ell/2$, and $d_1 + d_2 \leq \ell/2$ hold. Next, let $U = T[\lceil c_x - \ell/2 \rceil..\lfloor c_y + \ell/2 \rfloor]$, $V = T[\lceil c_y - \ell/2 \rceil..\lfloor c_z + \ell/2 \rfloor]$, and $W = T[\lceil c_x - \ell/2 \rceil..\lfloor c_z + \ell/2 \rfloor]$. By Lemma 5, U has period $2d_1$ and V has period $2d_2$. Thus, y has periods both $2d_1$ and $2d_2$. Also, since $2d_1 + 2d_2 \leq \ell$ holds, y has a period $g = \gcd(2d_1, 2d_2)$ by the periodicity lemma [14] where $\gcd(a, b)$ denotes the greatest common divisor of a and b. Then W also has period g since $g < |y| = \ell$ and g divides both period $2d_1$ of U and period $2d_2$ of V. Furthermore, since $g \leq \min(2d_1, 2d_2) \leq d_1 + d_2$ and $d_1 + d_2 + \ell = |W|$, the inequality $g + \ell \leq |W|$ holds, and thus, $x = W[1..\ell] = W[g + 1..g + \ell]$ holds by the periodicity. This contradicts the uniqueness of x.

We have shown that the maximum number of SUPSs whose center is at most p, is two. Symmetrically, the maximum number of SUPSs whose center is at least p is also two. Thus, the maximum number of SUPSs for a single query interval is four.

Finally, we show that the upper bound is tight. Let us consider the following string $S \in$

8

$\{\mathtt{a},\mathtt{b}\}^*$ of length 87:

$$S =\texttt{aababaaababaaababa}\textcolor{blue}{\texttt{a}}\texttt{aabaaabaaabaaabaaa} \qquad \backslash\backslash \text{ length } 36$$

$$+ \texttt{ababaaababaaababa} \qquad\qquad\qquad\qquad \backslash\backslash \text{ length } 17$$

$$+ \texttt{baaababaaababaaab} \qquad\qquad\qquad\qquad \backslash\backslash \text{ length } 17$$

$$+ \texttt{baaabaaabaaabaaab} \qquad\qquad\qquad\qquad \backslash\backslash \text{ length } 17.$$

The operator $+$ denotes the concatenation of strings. For this string and query interval $[18, 18]$ (highlighted in blue in the figure), $\mathsf{SUPS}_S([18, 18]) = \{[1, 19], [4, 22], [16, 34], [18, 36]\}$ holds. Note that palindromes $S[2..18]$, $S[5..21]$, and $S[17..33]$, which are shorter than 19 and cover the interval $[18, 18]$, are not unique since each of them has another occurrence in the artificial gadgets concatenated by $+$ operators. Also, it can be easily checked that all palindromes of length at most 18 that cover the interval $[18, 18]$ are not unique. $\qquad\qquad\square$

The above example having four SUPSs is of length 87, and the length of each SUPS is 19. The smallest period of the former two SUPSs is 6, and that of the latter two SUPSs is 4. We do not know if the example is the shortest one.

4.2 Sum of Lengths of All MUPSs

In this subsection, we show an upper bound of the total sum of the lengths of MUPSs in a string T. It was shown that the number of MUPSs stabbed by a single position is $O(\log|T|)$ in [16]. This immediately implies that the sum of the lengths of MUPSs of a string T is $O(|T|\log|T|)$. Here, we improve the upper bound as follows:

Theorem 2. *The total sum of the lengths of MUPSs of a string T is $O(|T|)$.*

In order to show Theorem 2, we first analyze the sum of the lengths of MUPSs covering a single position. Let p be an arbitrary position in a string T. Also, let $\mathsf{LMUPS}_T(p)$ be the set of MUPSs that cover p and whose centers are at most p. Namely, $\mathsf{LMUPS}_T(p) = \{[s,t] \in \mathsf{MUPS}(T) \mid s \leq p \leq t \text{ and } (s+t)/2 \leq p\}$. We show the following lemma:

Lemma 6. *For any position p in a string T,*

$$\sum_{[s,t]\in\mathsf{LMUPS}_T(p)} (t - s + 1) \in O(L_p)$$

holds where L_p is the maximum length of MUPSs covering position p.

Proof. Each MUPS in $\mathsf{LMUPS}_T(p)$ is an expansion of some palindromic suffix of $T[1..p]$. For a set G of palindromic suffixes of $T[1..p]$ and a MUPS μ of T, we say that μ is *obtained from* G if μ is an expansion of some palindrome in G. Here we use some notations used in Section 2.2. Namely, $G_1, G_2, \ldots, G_\pi$ are the groups of palindromic suffixes of $T[1..p]$. Also, k is the largest index of groups such that G_k is of type 2. Since no MUPS can be obtained from $\bigcup_{i\in[1,k-1]} G_i \setminus \{u_k, u_k v_k u_k\}$ by Lemma 3, we only consider the set $\bigcup_{j\in[k+1,\pi]} G_j \cup G_k \cup \{u_k, u_k v_k u_k\}$. If there are no MUPSs obtained from $\bigcup_{j\in[k+1,\pi]} G_j$, the number of MUPSs obtained from $\bigcup_{j\in[k+1,\pi]} G_j \cup G_k \cup \{u_k, u_k v_k u_k\}$ is $O(1)$ and the lemma holds. Thus, in the following, we consider the case that there exists some MUPSs obtained from $\bigcup_{j\in[k+1,\pi]} G_j$.

Now, let $\tilde{\mathcal{G}} = \tilde{G}_1, \ldots, \tilde{G}_\ell$ be the subsequence of the sequence $G_{k+1}, \ldots, G_\pi$ such that for each $\tilde{G}_i \in \tilde{\mathcal{G}}$, there exists a MUPS obtained from $\tilde{G}_i$. Further let $\tilde{s}_i$ be the shortest palindrome in $\tilde{G}_i$, and let $\tilde{d}_i$ be the smallest period for $\tilde{G}_i$. Then, by Lemma 4, for every i with $1 \leq i \leq \ell - 2$, any MUPS obtained from group $\tilde{G}_i$ is not longer than $\tilde{s}_{i+2}$. Since there are at most two MUPSs obtained from a single group [16], the sum of lengths of MUPSs obtained from $\tilde{G}_1 \cup \ldots \cup \tilde{G}_{\ell-2}$ is bounded by $2\tilde{s}_3 + 2\tilde{s}_4 + \cdots + 2\tilde{s}_\ell$. In general, $\tilde{s}_r < 2\tilde{d}_r$ holds from the
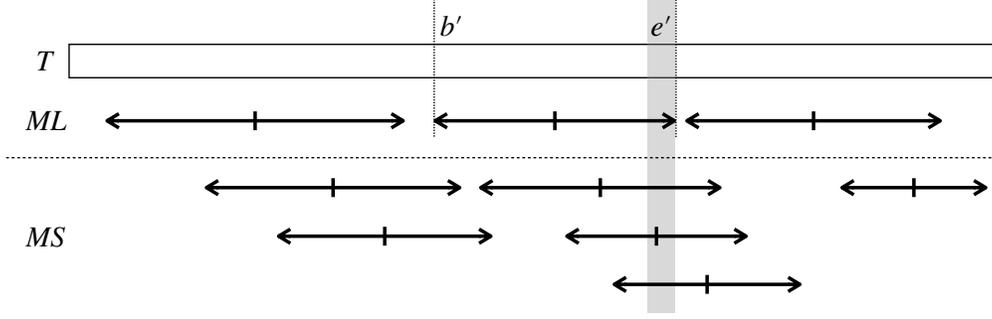
9

Figure 5: Illustration for the proof of Theorem 2. There are three MUPSs in *ML* and six MUPSs in *MS*. Also, there are four MUPSs that cover the ending position e' of MUPS $[b', e'] \in ML$. Since $[b', e']$ is a longest one among them, the sum of their lengths is bounded by $O(e' - b' + 1)$ by Corollary 2.

periodicity. Thus, $2\tilde{s}_3 + 2\tilde{s}_4 + \cdots + 2\tilde{s}_\ell < 4\sum_{r=3}^{\ell} \tilde{d}_r \in O(\tilde{d}_\ell)$ since $\tilde{d}_{r+2} \geq \tilde{d}_{r+1} + \tilde{d}_r$ holds for any $3 \leq r \leq \ell - 2$ by Lemma 1. Let L'_p be the maximum length of MUPSs obtained from $H = \{u_k, u_k v_k u_k\} \cup G_k \cup \tilde{G}_{\ell-1} \cup \tilde{G}_\ell$. Then, the sum of lengths of MUPSs obtained from H is $O(L'_p)$. Therefore, the total sum of the lengths of MUPSs in $\mathsf{LMUPS}_T(p)$ is in $O(\tilde{d}_\ell + L'_p) \subseteq O(L_p)$ since $\tilde{d}_\ell \leq \tilde{s}_\ell \leq L'_p \leq L_p$. □

By symmetry, Lemma 6 immediately leads to the next Corollary 2.

Corollary 2. *For any position p in a string T, the sum of the lengths of MUPSs covering position p is $O(L_p)$ where L_p is the maximum length of MUPSs covering position p.*

We are now ready to prove Theorem 2.

Proof of Theorem 2. We divide the set of MUPSs into two sets. First, let *ML* and *MS* be the empty sets initially. We then perform the following operations for each MUPS of T in descending order of their lengths (the order of two elements of the same length is arbitrary): If $[b, e] \in \mathsf{MUPS}(T)$ does not share the same position for any $[b', e'] \in ML$, then update $ML = ML \cup \{[b, e]\}$. Otherwise, add $[b, e]$ to *MS*. Then, $\sum_{[b,e] \in ML}(e - b + 1) \leq |T|$ holds since all elements in *ML* do not overlap each other. Also, any MUPS $[b, e] \in MS$ contains some position p such that $b' \leq p \leq e'$ with $[b', e'] \in ML$. Moreover, since MUPSs cannot be nested, each MUPS in *MS* contains either or both of the ending position of MUPS $[b'_i, e'_i] \in ML$ and the beginning position of MUPS $[b'_{i+1}, e'_{i+1}] \in ML$ for some i. By Corollary 2, the sum of the length of MUPSs covering position b'_i or e'_i is $O(e'_i - b'_i + 1)$. Therefore, $\sum_{[b,e] \in MS}(e - b + 1) \in O(\sum_{[b',e'] \in ML}(e' - b' + 1)) \subseteq O(|T|)$ holds (see also Figure 5). This completes the proof. □

Using Theorem 2, we design a simple algorithm for computing all point SUPSs. As in Inoue et al.'s method described in Section 3, we use the MUPSs and maximal palindromes to compute point SUPSs. However, we can avoid using an RmQ data structure.

Proposition 1. *Given the set of MUPSs of a string T of length n and the set of maximal palindromes of T one can compute all point SUPSs for all positions in $O(n)$ time without using RmQs.*

Proof. First, for each position, we record the shortest MUPS(s) covering the position. If there are no such MUPSs for a position, we record ∞ for the position. This can be done in $O(n)$ time by scanning all MUPSs naively since Theorem 2 holds. Next, for each position p, compare the three following values and find the shortest one(s): (1) the shortest expansion of the left-neighbor MUPS of p, (2) the recorded value (i.e., the length of the shortest MUPS covering p), and (3) the shortest expansion of the right-neighbor MUPS of p. Note that (1) and (3) may not

exist. This can be done in a total of linear time by using an $O(n)$-space and $O(1)$-query time predecessor/successor data structure. $\qquad\square$

5 Compact SUPS Data Structures

In this section, we propose space-efficient SUPS data structures. To our knowledge, the only SUPS data structure that can be sublinear size, i.e., $o(n \log n)$ bits, is that of Watanabe et al. [36]. Watanabe et al. [36] proposed a SUPS data structure of size $O(r \log n)$ bits where $r \leq n$ is the size of the run-length encoded string. Their data structure will be small when the input string is highly compressible with run-length encoding. On the other hand, $O(r \log n)$ bits can be large as much as $O(n \log n)$ bits in the worst case. In this section, we propose $O(n)$-bits data structures which can answer SUPS queries in optimal time. Namely, our data structure is always space-efficient regardless of the compression scheme or characteristic structures of the input string. In the rest of this paper, let m be the number of MUPSs of string T. The sizes of our data structures are $3n + 2m + o(n)$ bits for interval SUPS queries, and $3n + m' + o(n)$ bits for point SUPS queries where $m' \leq m$ is the number of *meaningful MUPSs* that we will define later.

5.1 Data Structures for Interval SUPS Queries

First, we show a compact representation of the data structure of Inoue et al. Our data structure consists of compact representations of (1) the set of MUPSs, (2) the set $\mathcal{M}$ of maximal palindromes each of which is an expansion of some MUPS, and (3) an RmQ data structure over the sequence of the lengths of MUPSs.

(1) We represent the set of MUPSs as two length-n bit-arrays B and E that indicate the beginning and the ending positions of MUPSs. Namely, $B[i] = 1$ iff some MUPS begins at i, and $E[j] = 1$ iff some MUPS ends at j for each $1 \leq i, j \leq n$. Since MUPSs cannot be nested, the number of the set-bits in B is exactly m, and in E as well (see Fig. 6 for examples).

(2) We represent $\mathcal{M}$ as the length-n bit-array L that indicates the beginning positions of maximal palindromes in $\mathcal{M}$. Namely, $L[i] = 1$ iff some palindrome in $\mathcal{M}$ begins at i for each $1 \leq i \leq n$. Since all palindromes in $\mathcal{M}$ are unique by the definition, they cannot be nested, and thus, the number of the set-bits in L is exactly m. Namely, the i-th set-bit in L corresponds to the i-th MUPS. Note that, for any k, we can restore the ending position of the k-th palindrome in $\mathcal{M}$ from three arrays B, E, and L, that is, $b + e - \ell$ where b, e, and ℓ are the positions of the k-th set-bits in B, E, and L, respectively.

(3) We build the succinct RmQ data structure of [15] on the sequence of the lengths of MUPSs. The size of the data structure is $2m + o(m)$ bits.

Also, we enhance three bit-arrays B, E, and L with rank/select dictionaries. Then, we can completely simulate the algorithm of Inoue et al., i.e., any SUPS query can be answered in constant time. This data structure requires $3n + 2m + o(n)$ bits of space.

The construction time is linear: All the maximal palindromes in T and all the MUPSs of T can be computed in $O(n)$ time [27, 22], and hence, three bit-arrays B, E, and L can be computed in $O(n)$ time. Also, the rank/select dictionaries and the succinct RmQ data structure for a bit-array of length n can be constructed in $O(n)$ time [23, 10, 15].

To summarize, we obtain the next theorem:

Theorem 3. *There is a data structure of size $3n + 2m + o(n)$ that can answer any interval SUPS query in $O(1)$ time where m is the number of MUPSs of T. Also, given T, we can construct the data structure in $O(n)$ time.*
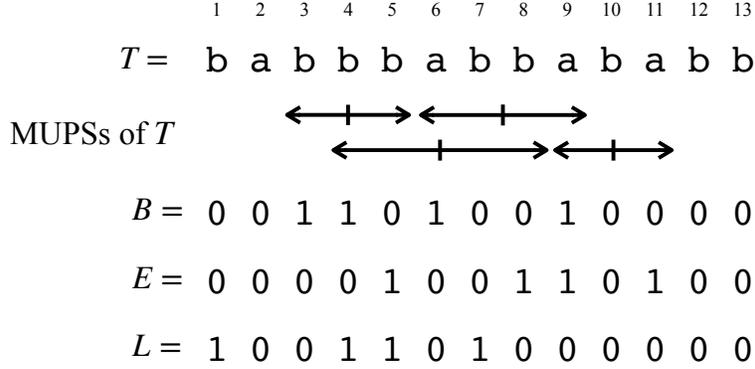
11

	1	2	3	4	5	6	7	8	9	10	11	12	13
$T =$	b	a	b	b	b	a	b	b	a	b	a	b	b

MUPSs of T

$B =$	0	0	1	1	0	1	0	0	1	0	0	0	0
$E =$	0	0	0	0	1	0	0	1	1	0	1	0	0
$L =$	1	0	0	1	1	0	1	0	0	0	0	0	0

Figure 6: Three arrays B, E, and L for string $T = $ babbbabbababb. The first two arrays B and E indicate the beginning and ending positions of MUPSs of T. The third array L indicates the beginning positions of the maximal expansions of MUPSs of T. For instance, $L[1] = 1$ holds since $T[1..7] = $ babbbab is the maximal palindrome centered at position 4, which is the center of MUPS $T[3..5]$.

5.2   Data Structures for Point SUPS Queries

As for the point SUPS queries, we can further reduce the space usage with a new algorithm specialized to point queries. Firstly, we define a new[2] notion for MUPSs.

Definition 1. A MUPS is said to be meaningful if there is a SUPS which is an expansion of the respective MUPS for some position. MUPSs which are not meaningful are said to be meaningless.

For example, in Fig. 6, MUPS $T[4..8] = $ bbabb of length 5 is meaningless since the lengths of SUPSs for positions 4 and 5 are $3 < 5$ and the lengths of SUPSs for positions 6, 7, and 8 are $4 < 5$. Given the set of MUPSs, we can compute the set of meaningful MUPSs in $O(n)$ time by computing all SUPSs for all positions (e.g., Proposition 1) and removing MUPSs unused.

Let $\mathsf{MLen} = (x_1, \ldots, x_{m'})$ be the sequence of the lengths of meaningful MUPSs sorted in increasing order on their starting positions. Also, let $\mathsf{MLen}_p \subseteq \mathsf{MLen}$ be the sequence of the lengths of meaningful MUPSs stabbed by a position p. The next lemma states that MLen_p has a sort of monotonicity.

Lemma 7. There are no three elements x_i, x_j, and x_k in MLen_p such that $i < j < k$ and $x_i < x_j > x_k$.

Proof. Assume on the contrary that there exist x_i, x_j, and x_k satisfying the conditions above. Let s_i, s_j, and s_k be the starting positions of the MUPSs, respectively. Since MUPSs cannot be nested and three the MUPSs cover the same position p, every position inside the second MUPS is covered by the first MUPS or the third MUPS. Namely, $s_i < s_j$, $s_j + x_j - 1 < s_k + x_k - 1$, and $s_k \leq p \leq s_i + x_i - 1$ hold (see Fig. 7). For each position $q \in [1, s_i]$, there is no expansion of the second MUPS $T[s_j..s_j + x_j - 1]$ starting at q because if such a palindrome exists, it contradicts the uniqueness of the first MUPS $T[s_i..s_i + x_i - 1]$. Symmetrically, for each position $q' \in [s_k + x_k - 1, n]$, there is no expansion of the second MUPS $T[s_j..s_j + x_j - 1]$ ending at q'. Finally, for each position $q'' \in [s_i + 1, s_k + x_k - 2]$, any palindrome covering both q'' and $[s_j, s_j + x_j - 1]$ cannot be a SUPS for q'' since the second MUPS is longer than another MUPS covering q''. Thus, the second MUPS is meaningless, a contradiction. □

From this lemma, MLen_p can be regarded as a concatenation of a (possibly empty) non-increasing sequence and a non-decreasing sequence. Thus, if we find the leftmost value x_t such

---

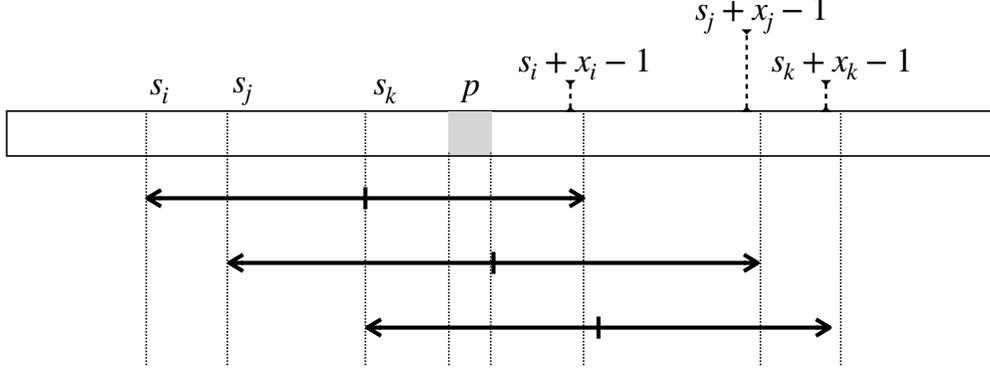[2]This is inspired by a similar notion defined for minimal unique substrings in [32].

Figure 7: Illustration for a contradiction in the proof of Lemma 7. The second MUPS which is the longest among three MUPSs cannot be a meaningful MUPS.

that $x_t > x_{t-1}$, then x_{t-1} is the smallest in MLen_p and $x_{t'}$ is not for each $t' \geq t$. Also, from the monotonicity of MLen_p before x_t, we can find MUPSs of length equal to x_{t-1} by (backward) linear search starting at x_t on MLen_p. In order to find such x_t, we precompute the bit-array inc of length $|\mathsf{MLen}|$ such that $\mathsf{inc}[0] = 0$, and $\mathsf{inc}[i] = 1$ iff $x_i > x_{i-1}$. Also, we enhance the array inc with rank/select dictionaries so that we can answer any successor query on inc in constant time.

Our data structure includes the bit-arrays B, E, and L as in the previous subsection. On the other hand, instead of an RmQ data structure of size $2m + o(m)$ bits, we use bit-array inc of length m' and rank/select dictionaries of size $o(m')$ where $m' = |\mathsf{MLen}| \leq m$.

Our algorithm is almost the same as Inoue et al.'s one except that we use the bit-array inc instead of an RmQ data structure. Given a query position p, we first compute the left-neighbor and right-neighbor MUPSs and check whether their expansion can cover p or not. Simultaneously, we obtain the range of meaningful MUPSs stabbed by p. Let i (resp., j) be the index of the leftmost (resp., rightmost) meaningful MUPS stabbed by p. We then find the position k of the leftmost set-bit in $\mathsf{inc}[i + 1..j]$. If such a set-bit does not exist (i.e., $\mathsf{inc}[i + 1..j] = \mathbf{0}$), let $k = j + 1$ for convenience. By the definition of inc and k, the prefix $(x_i, \ldots, x_{k-1})$ of MLen_p is non-increasing. Especially, if $k = j + 1$, then $\mathsf{MLen}_p = (x_i, \ldots, x_j)$ is non-increasing, and hence, x_j is the smallest within MLen_p. Since there are at most four SUPSs (Theorem 1), it suffices to compare x_{j+1-t} for $t = 1, 2, 3, 4$. If $k \leq j$, then the suffix $(x_k, \ldots, x_j)$ of MLen_p is non-decreasing by Lemma 7. In this case, x_{k-1} is the smallest since $x_{k-1} < x_k$. Again, since there are at most four SUPSs (Theorem 1), it suffices to compare x_{k-t} for $t = 1, 2, 3, 4$. Therefore, we can find all MUPSs that are candidates for SUPSs for p without using any RmQ data structure.

All the above operations can be performed in constant time by using rank/select dictionary on inc, B, E, and L. We obtain the next theorem:

Theorem 4. *There is a data structure of size $3n + m' + o(n)$ that can answer any point SUPS query in $O(1)$ time where m' is the number of meaningful MUPSs of T. Also, given T, we can construct the data structure in $O(n)$ time.*

6 Semi-dynamic SUPS Data Structures

In this section, we introduce SUPS data structures under two semi-dynamic models: the sliding-window model and the after-edit model. Our results are based on the static method proposed by Inoue et al. [22], which we reviewed in Section 3.

6.1 Sliding-window Data Structures

We make some modifications to the static data structures from Section 3 to answer any SUPS queries for a sliding window.

It is shown in [30] that the number of changes of MUPSs is constant when we append a character or delete the first character, and we can detect the changes in amortized $O(\log \sigma)$ time. Further, predecessor and successor data structures on the MUPSs can be updated dynamically in $O(\log \log n)$ time using van Emde Boas trees [35].

For a dynamic RmQ data structure, we can use the one proposed by Brodal et al. [8]. However, if we directly apply their data structure to our problem, the updating time is in $\Omega(\log n / \log \log n)$, and it becomes a bottleneck. In order to avoid such a situation, we use another dynamic data structure with some constraints which suffices for our problem.

As in the algorithm described in Section 3, we will use RmQ on the sequence of the lengths of MUPSs. The width of a query range of RmQ is bounded by the number of MUPSs covering query interval $[p, q]$ (see also Fig. 3). It is known that the number of MUPSs covering any interval is $O(\log n)$ [16], hence the width of a query range of RmQ is also $O(\log n)$. We call the range minimum query such that the width of any query is constrained in $O(\mathsf{polylog}(n))$ LogRmQ. Later, we show the following lemma:

Lemma 8. *There exists a linear size data structure for a dynamic array A that supports any* **LogRmQ** *on A in constant time. We can maintain the data structure in constant time when an element of A is substituted by another value. Also, we can maintain the data structure in amortized constant time when some element is inserted to (or deleted from) A.*

Finally, we show that the set of maximal palindromes for a sliding window can be maintained efficiently. We generalize Manacher's algorithm [27] to the sliding-window model.

6.1.1 Manacher's Algorithm for Sliding Window.

Manacher's algorithm is an online algorithm that computes the set of maximal palindromes in a string. In this subsection, we apply Manacher's algorithm to the sliding-window model. The problem was solved in [20], however, we will describe a sliding-window algorithm for completeness.

Important invariants of Manacher's algorithm before reading the i-th character are (1) we know the center position c of the longest palindromic suffix of $T[1..i-1]$, and (2) we know all the maximal palindromes, each of whose center is at most c. Note that for the SUPS query, we are interested in maximal palindromes, which are *unique* in the string. Since any palindrome whose center is greater than c is not unique, the second invariant is sufficient for our purpose.

When a character is appended to the current window, we update the set of maximal palindromes in the online manner of the original Manacher's algorithm. When the first character of the window is deleted, we do not need to do anything if the window $T[b..e]$ itself is not a palindrome. Instead, when we refer to the arm-length of the maximal palindrome centered at a specified position, we need to consider that the left-end of the palindrome may exceed the left-end of the window. Namely, if the stored arm-length for center x is ℓ_x, the actual arm-length is $\min\{\ell_x, \lceil x - b \rceil\}$.

If the window $T[b..e]$ itself is a palindrome, we need to update the longest palindromic suffix to keep the first invariant. This can be done in amortized $O(1)$ time as in Manacher's algorithm. More precisely, for every (half) integers $j = 0.5, 1, 1.5 \ldots$, the arm-length of the maximal palindrome of center $\frac{b+e}{2} + j$ is equal to that of center $\frac{b+e}{2} - j$. Thus, we copy them for incremental j's until we find a palindromic suffix of $T[b+1..e]$. Then, we set c to the center position of the suffix palindrome we found. Since the sequence of center positions of the longest palindromic suffixes of the windows is non-decreasing while running the algorithm, the total processing time is $O(n)$.

Therefore, we obtain the following:

Theorem 5. *There exists a data structure of size $O(W)$ for the sliding-window SUPS problem that supports* sups($[p, q]$) *in* $O(\log \log W)$ *time and* pushback(c) *and* pop() *in amortized* $O(\log \sigma + \log \log W)$ *time, where W is the size of the window.*

6.2 After-edit Data Structure

In this subsection, we design a SUPS data structure for the after-edit model. Basically, the idea is the same as the previous one. The only difference is that we do not maintain maximal palindromes in the after-substitution SUPS problem. Instead, we use a bidirectional LCE on the original string T.

Theorem 6. *There exists a data structure of size $O(n)$ for the after-substitution SUPS problem that can be updated in amortized $O(\log \sigma + (\log \log n)^2 + d \log \log n)$ time for a single substitution and can answer any subsequent SUPS queries in $O(k \log \log n)$ time, where d is the number of changes of MUPSs when the substitution is applied to T, and k is the number of the SUPS queries after the substitution. Also, given a string T, the data structure can be constructed in $O(n)$ time.*

Proof. Given a substitution operation, we can detect all the changes of MUPSs in $O(\log \sigma + (\log \log n)^2 + d)$ time [16]. Then, the set of MUPSs can be updated in $O(d)$ time, the predecessor/successor data structures can be updated in $O(d \log \log n)$ time, and the LogRmQ data structure can be updated in amortized $O(d)$ time by Lemma 8. Finally, we can compute the maximal palindromes in T' that are expansions of the left-neighbor and the right-neighbor MUPSs by answering a constant number of bidirectional LCE queries on T while skipping the edited position (so-called kangaroo jumps). Also, it is known that the set of MUPSs of T, the predecessor/successor data structures, and the LCE data structure can be computed in $O(n)$ time. Further, the LogRmQ data structure can be computed in $O(n)$ time by Lemma 8. ☐

6.3 Dynamic LogRmQ

In this subsection, we give a proof of Lemma 8. We assume that the width of the query range is constrained in $O(\log^c n)$ for a fixed constant c. We first consider dividing the input array A into blocks of size $\log^c n$. We call each of the blocks large block. Then, we build a linear size dynamic RmQ data structure on each large block. Given a query range of width $O(\log^c n)$, we get range minima from a constant number of large blocks and then naively compare them.

We update the RmQ data structure on the large block containing the edited position when the input array A is edited. If the size of a large block becomes far from $\log^c n$ by insertions or deletions, then we split a block or merge continuous blocks to keep the size in $\Theta(\log^c n)$. For example, we split a block into two blocks when the block size exceeds $2 \log^c n$ and merge two adjacent blocks when the block size falls below $\frac{1}{2} \log^c n$. If each large block can be updated in amortized constant time, the whole data structure can also be updated in amortized constant time. In the next subsection, we consider how to treat a large block.

Recursive Structure of Large Block. In order to update large blocks efficiently, we apply the path minima data structure proposed by Brodal et al. [8]. They treated the problem of path minima queries on a tree, a generalization of range minimum queries on an array.

First, we divide a large block B of length $\Theta(\log^c n)$ into small blocks each of length $L = \Theta(\log^\varepsilon n)$ where $\varepsilon < 1$ is an arbitrary small constant.

Let B_1 be the array of length $\Theta(\log^{c-\varepsilon} n)$ that stores the minima of small blocks on B. A query on large block B can be reduced to at most two queries on small blocks and at most one query on B_1 (see Fig. 8). Similarly, for every $i \geq 2$, we divide B_{i-1} into small blocks of the fixed-length L and let B_i be the array of size $\Theta(\log^{c-i\varepsilon} n)$ that stores the minima of small blocks on B_{i-1}. A query on B_i can be reduced to at most two queries on small blocks and at most one query on B_{i+1} at the next level. We recursively apply such division until the size of B_i becomes a constant. The recursion depth is $O(c/\varepsilon)$, i.e., a constant. Notice that recursion
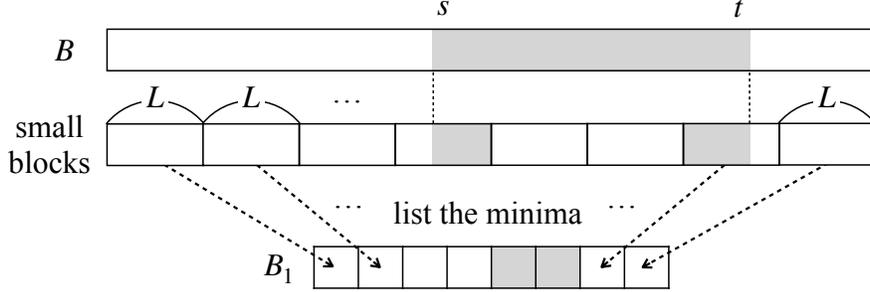
15

Figure 8: Illustration for dividing a large block B into small blocks. Query $[s, t]$ on B can be reduced to queries inside the fourth and the seventh small blocks, and query $[5, 6]$ on B_1.

occurs at most once at each level. Thus, if we answer RmQ inside a small block in constant time, then the total query time is also a constant.

We answer a query on each small block by using a lookup-table, where the index is a pair of a small block and a query range, and the value is the answer (the position of a minimum). Since RmQ returns the *position* corresponding to a range minimum, we can convert each small block to a sequence of its *local ranks*. Then, the number of possible variants of such small blocks is at most $O((\log^\varepsilon n)^{\log^\varepsilon n}) \subset o(n)$. Also, the total variations with all possible query intervals are still $O((\log^\varepsilon n)^2) \subset o(n)$, i.e., the number of elements in the lookup-table is $O((\log^\varepsilon n)^{\log^\varepsilon n+2}) \subset o(n)$. Furthermore, a small block (i.e., an element in the lookup-table) can be represented in $o(\log n)$ bits: the length, the pointers to each element, and the local ranks. Thus, table lookup can be done in constant time. Namely, the time complexity of an RmQ on a small block is constant. For substitutions (resp., insertions and deletions), updating small blocks can be done in worst-case (resp., amortized) constant time by combining another lookup-table and Q-heap (cf. [8]). Therefore, we have proven Lemma 8.

7 Conclusions and Discussions

In this paper, we studied SUPS problems of static and non-static strings. Firstly, we showed combinatorial properties on the problems; the tight upper bound on the maximum number of SUPSs for a single interval, and the sum of lengths of MUPSs of a string is linear to the length of the string. Secondly, we improved Inoue et al.'s time-optimal algorithm on space usage, i.e., we designed a compact data structure of size $3n + 2m + o(n)$ bits that can answer any interval SUPS query in constant time where n is the length of the input string and m is the number of MUPSs of the input string. Also, we proposed a new method specialized for the point SUPS problem, and based on the method, we designed a more space-efficient compact data structure of size $3n + m' + o(n)$ bits that can answer any point SUPS query in constant time where m' is the number of meaningful MUPSs of the input string. Finally, we considered SUPS problems in two semi-dynamic models. We proposed a data structure of size $O(W)$ for the sliding-window SUPS problem that supports any SUPS query and window-shift operation in $\tilde{O}(1)$ time where W is the size of the window. Further, we propose a data structure of size $O(n)$ for the after-substitution SUPS problem that can answer any SUPS query after a single character substitution in amortized $\tilde{O}(1)$ time. As a by-product, we proposed a fully-dynamic data structure for the range minimum queries in which the width of each query range is in $\tilde{O}(1)$.

Designing an efficient SUPS algorithm for a fully dynamic setting is future work. All the known algorithms for SUPS queries for static/non-static strings basically precompute the set of MUPSs of the input string. If we try to extend such algorithms to a fully-dynamic one, maintaining the set of MUPSs may be a bottleneck. To the best of our knowledge, there is no study that deals with unique substrings in a dynamic string while maintaining palindromic structures in a fully dynamic string has been studied in some literature [3, 6]. In general,

occurrences of substrings can change dramatically when a string is edited. The algorithm of [16] can capture the changes of MUPSs for a single edit after linear-time preprocessing, however, it cannot be applied directly to multiple edits. Another possible way is to compute SUPSs for a query interval without using MUPSs, namely, to determine the uniqueness of palindromes covering the query interval in a dynamic string. For an implementation of this idea, a dynamic suffix array proposed by Kempa and Kociumaka [24] might be useful.

Acknowledgements

References

[1] Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The heaviest induced ancestors problem: Better data structures and applications. *Algorithmica*, 84(7):2088–2105, 2022. `doi:10.1007/s00453-022-00955-7`.

[2] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 534–544. IEEE Computer Society, 1998. `doi:10.1109/SFCS.1998.743504`.

[3] Amihood Amir and Itai Boneh. Dynamic palindrome detection. *CoRR*, abs/1906.09732, 2019. URL: `http://arxiv.org/abs/1906.09732`.

[4] Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition detection in a dynamic string. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ESA.2019.5`.

[5] Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2017. `doi:10.1007/978-3-319-67428-5\_2`.

[6] Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. `doi:10.1007/s00453-020-00744-0`.

[7] Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. *Theor. Comput. Sci.*, 141(1&2):163–173, 1995. `doi:10.1016/0304-3975(94)00083-U`.

[8] Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. Path minima queries in dynamic weighted trees. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings*, volume 6844 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2011. `doi:10.1007/978-3-642-22300-6\_25`.

[9] Panagiotis Charalampopoulos, Pawel Gawrychowski, and Karol Pokorski. Dynamic longest common substring in polylogarithmic time. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 27:1–27:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.ICALP.2020.27`.

[10] David Clark. *Compact pat trees*. PhD thesis, University of Waterloo, 1997.

[11] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, 32(4):396–402, 1984. `doi:10.1109/TCOM.1984.1096090`.

[12] Maxime Crochemore, Alice Héliou, Gregory Kucherov, Laurent Mouchard, Solon P. Pissis, and Yann Ramusat. Absent words in a sliding window with applications. *Inf. Comput.*, 270, 2020. `doi:10.1016/j.ic.2019.104461`.

[13] Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989. `doi:10.1145/63334.63341`.

[14] Nathan J Fine and Herbert S Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. `doi:10.1090/S0002-9939-1965-0174934-9`.

[15] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. `doi:10.1137/090779759`.

[16] Mitsuru Funakoshi and Takuya Mieno. Minimal unique palindromic substrings after single-character substitution. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings*, volume 12944 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 2021. `doi:10.1007/978-3-030-86692-1\_4`.

[17] Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing longest palindromic substring after single-character or block-wise edits. *Theor. Comput. Sci.*, 859:116–133, 2021. `doi:10.1016/j.tcs.2021.01.014`.

[18] Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding. In Rolf Karlsson and Andrzej Lingas, editors, *Algorithm Theory — SWAT'96*, pages 392–403, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[19] Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1509–1528. SIAM, 2018. `doi:10.1137/1.9781611975031.99`.

[20] Pawel Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemyslaw Uznanski. Tight tradeoffs for real-time approximation of longest palindromes in streams. *Algorithmica*, 81(9):3630–3654, 2019. `doi:10.1007/s00453-019-00591-8`.

[21] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. `doi:10.1017/cbo9780511574931`.

[22] Hiroe Inoue, Yuto Nakashima, Takuya Mieno, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Algorithms and combinatorial properties on shortest unique palindromic substrings. *J. Discrete Algorithms*, 52-53:122–132, 2018. `doi:10.1016/j.jda.2018.11.009`.

[23] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554. IEEE Computer Society, 1989. `doi:10.1109/SFCS.1989.63533`.

[24] Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, page 1657–1670, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3519935.3520061`.

[25] Etsuro Kuramoto, Osamu Yano, Yoshimitsu Kimura, Makoto Baba, Tadashi Makino, Saburo Y, Toshiko Yamamoto, Tetsuro Kataoka, and Tohru Tokunaga. Oligonucleotide sequences required for natural killer cell activation. *Japanese Journal of Cancer Research*, 83(11):1128–1131, 1992. `doi:10.1111/j.1349-7006.1992.tb02734.x`.

[26] N. Jesper Larsson. Extended application of suffix trees to data compression. In James A. Storer and Martin Cohn, editors, *Proceedings of the 6th Data Compression Conference (DCC '96), Snowbird, Utah, USA, March 31 - April 3, 1996*, pages 190–199. IEEE Computer Society, 1996. `doi:10.1109/DCC.1996.488324`.

[27] Glenn K. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975. `doi:10.1145/321892.321896`.

[28] Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8-10):900–913, 2009. `doi:10.1016/j.tcs.2008.12.016`.

[29] Takuya Mieno, Yuta Fujishige, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing minimal unique substrings for a sliding window. *Algorithmica*, 84(3):670–693, 2022. `doi:10.1007/s00453-021-00864-1`.

[30] Takuya Mieno, Kiichi Watanabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Palindromic trees for a sliding window and its applications. *Inf. Process. Lett.*, 173, 106174, 2022. `doi:10.1016/j.ipl.2021.106174`.

[31] M Senft. Suffix tree for a sliding window: An overview. In *WDS 2005*, pages 41–46, 2005.

[32] Kazuya Tsuruta, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Shortest unique substrings queries in optimal time. In *SOFSEM 2014: Theory and Practice of Computer Science - 40th International Conference on Current Trends in Theory and Practice of Computer Science*, volume 8327 of *Lecture Notes in Computer Science*, pages 503–513. Springer, 2014. `doi:10.1007/978-3-319-04298-5\_44`.

[33] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. `doi:10.1007/BF01206331`.

[34] Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest Lyndon substring after edit. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, volume 105 of *LIPIcs*, pages 19:1–19:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.CPM.2018.19`.

[35] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977. `doi:10.1016/0020-0190(77)90031-X`.

[36] Kiichi Watanabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fast algorithms for the shortest unique palindromic substring problem on run-length encoded strings. *Theory Comput. Syst.*, 64(7):1273–1291, 2020. `doi:10.1007/s00224-020-09980-x`.

[37] S Yamamoto, T Yamamoto, T Kataoka, E Kuramoto, O Yano, and T Tokunaga. Unique palindromic sequences in synthetic oligonucleotides are required to induce IFN [correction of INF] and augment IFN-mediated [correction of INF] natural killer activity. *The Journal of Immunology*, 148(12):4072–4076, 1992. URL: `https://www.jimmunol.org/content/148/12/4072`.

[38] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. `doi:10.1109/TIT.1977.1055714`.