Faster Queries for Longest Substring Palindrome After Block Edit

Mitsuru Funakoshi
Department of Informatics, Kyushu University, Japan
mitsuru.funakoshi@inf.kyushu-u.ac.jp

Yuto Nakashima
Department of Informatics, Kyushu University, Japan
yuto.nakashima@inf.kyushu-u.ac.jp

Shunsuke Inenaga
Department of Informatics, Kyushu University, Japan
inenaga@inf.kyushu-u.ac.jp

Hideo Bannai
Department of Informatics, Kyushu University, Japan
bannai@inf.kyushu-u.ac.jp

Masayuki Takeda
Department of Informatics, Kyushu University, Japan
takeda@inf.kyushu-u.ac.jp

Abstract

Palindromes are important objects in strings which have been extensively studied from combinatorial, algorithmic, and bioinformatics points of views. Manacher [J. ACM 1975] proposed a seminal algorithm that computes the longest substring palindromes (LSPals) of a given string in $O(n)$ time, where n is the length of the string. In this paper, we consider the problem of finding the LSPal after the string is *edited*. We present an algorithm that uses $O(n)$ time and space for preprocessing, and answers the length of the LSPals in $O(\ell + \log\log n)$ time, after a substring in T is replaced by a string of arbitrary length ℓ. This outperforms the query algorithm proposed in our previous work [CPM 2018] that uses $O(\ell + \log n)$ time for each query.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases palindromes, string algorithm, periodicity

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.27

Related Version `https://arxiv.org/abs/1901.10722`

Funding *Yuto Nakashima*: Supported by JSPS KAKENHI Grant Number JP18K18002.
Shunsuke Inenaga: Supported by JSPS KAKENHI Grant Number JP17H01697.
Hideo Bannai: Supported by JSPS KAKENHI Grant Number JP16H02783.
Masayuki Takeda: Supported by JSPS KAKENHI Grant Number JP18H04098.

1 Introduction

Palindromes are strings that read the same forward and backward. Finding palindromic structures in strings has important applications in analysis of DNA, RNA, and protein sequences, and thus a variety of efficient algorithms for finding palindromic structures occurring in a given string have been proposed (e.g., see [3, 18, 12, 15, 19, 14, 10] and references therein).

In this paper, we consider the fundamental problem of finding the *longest substring palindrome* (*LSPal*) in a given string T. Observe that the longest substring palindrome is also a maximal (non-extensible) palindrome in the string, whose center is an integer position if its length is odd, or a half-integer position if its length is even. Hence, in order to compute the LSPal of a given string T, it suffices to compute all maximal palindromes in T. Manacher [16] gave an elegant $O(n)$-time algorithm to find all maximal palindromes in a given string of length n. Manacher's algorithm utilizes symmetry of palindromes and character equality comparisons only, and therefore works in $O(n)$ time for any alphabet. There is an alternative suffix tree [21] based algorithm which works in $O(n)$ time in the case of an integer alphabet of polynomial size in n [13]. Finding the longest substring palindrome in the streaming model has also been considered [6, 11].

Now we consider the following question: what happens to those palindromes if the string T is edited? It seems natural to ask this kind of question since a typical biological sequence can contain some uncertainties such that there are multiple character possibilities at some positions in the sequence. In our recent work [9], we initiated this line of research and showed the following results. Let n be the length of the input string T and σ the alphabet size.

1-ELSPal. We can preprocess T in $O(n)$ time and space such that later, we can answer in $O(\log \min\{\sigma, \log n\})$ time the longest substring palindrome after a single character edit operation (insertion, deletion, or substitution).

ℓ-ELSPal. We can preprocess T in $O(n)$ time and space such that later, we can answer in $O(\ell + \log n)$ time the longest substring palindrome after a block-wise edit operation, where ℓ is the length of the new block that substitutes the substring in T.

In this paper, we further pursue the second variant of the problem (ℓ-ELSPal) where an existing substring is replaced with a new string (block) of length ℓ. We remark that the length ℓ of a new block is arbitrary. The main result of this paper is an $O(\ell + \log \log n)$-time query algorithm that answers the longest substring palindrome after a block-wise edit operation, with $O(n)$-time and space preprocessing.

Note that ℓ-ELSPal is a generalization of 1-ELSPal, where $\ell = 1$ for insertion and substitution and $\ell = 0$ for deletion. Therefore, the result of this paper achieves $O(\log \log n)$-time query algorithm for 1-ELSPal. This is as efficient as the $O(\log \min\{\sigma, \log n\})$-time query of [9] when the alphabet size σ is at least $O(\log n)$ (e.g., in the case of an integer alphabet).

Related work

Amir et al. [1] proposed an algorithm to find the *longest common factor* (*LCF*) of two strings, after a single character edit operation is performed in one of the strings. Their data structure occupies $O(n \log^3 n)$ space and uses $O(\log^3 n)$ query time, where n is the length of the input strings. Their data structure can be constructed in $O(n \log^4 n)$ expected time. Urabe et al. [20] considered the problem of computing the longest *Lyndon word* after an edit operation. They showed $O(\log n)$-time queries for a single character edit operation and $O(\ell \log \sigma + \log n)$-time queries for a block-wise edit operation, both using $O(n)$ time and space for preprocessing. We note that in these results including ours in this current paper, edit operations are given as *queries* and thus the input string(s) remain static even after each query. This is due to the fact that changing the data structure dynamically can be too costly in many cases. It is noteworthy, however, that very recently Amir et al. [2] solved dynamic versions for the LCF problem and some of its variants. In particular, when n is the maximum length of the string that can be edited, they showed a data structure of $O(n \log n)$ space that can be dynamically maintained and can answer 1-ELSPal queries in $O(\sqrt{n} \log^{2.5} n)$ time, after $O(n \log^2 n)$ time preprocessing.

2 Preliminaries

Let Σ be the *alphabet*. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $T = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of T, respectively. For two strings X and Y, let $lcp(X, Y)$ denote the length of the longest common prefix of X and Y.

For a string T and an integer $1 \leq i \leq |T|$, $T[i]$ denotes the i-th character of T, and for two integers $1 \leq i \leq j \leq |T|$, $T[i..j]$ denotes the substring of T that begins at position i and ends at position j. For convenience, let $T[i..j] = \varepsilon$ when $i > j$. An integer $p \geq 1$ is said to be a *period* of a string T iff $T[i] = T[i + p]$ for all $1 \leq i \leq |T| - p$. If a string B is both a proper prefix and a proper suffix of another string T, then B is called a *border* of T.

For any string P, let P^R denote the reversed string of P. A string P is called a *palindrome* if $P = P^R$. A non-empty substring palindrome $T[i..j]$ is said to be a *maximal palindrome* of T if $T[i - 1] \neq T[j + 1]$, $i = 1$, or $j = |T|$. For any non-empty substring palindrome $T[i..j]$ in T, $\frac{i+j}{2}$ is called its *center*. It is clear that for each center $q = 1, 1.5, \ldots, n - 0.5, n$, we can identify the maximal palindrome $T[i..j]$ whose center is q (namely, $q = \frac{i+j}{2}$). Thus, there are exactly $2n - 1$ maximal palindromes in a string of length n (including empty ones which occur at center $\frac{i+j}{2}$ when $T[i] \neq T[j]$).

A *rightward longest common extension* (*rightward LCE*) query on a string T is to compute $lcp(T[i..|T|], T[j..|T|])$ for given two positions $1 \leq i \neq j \leq |T|$. Similarly, a *leftward LCE* query is to compute $lcp(T[1..i]^R, T[1..j]^R)$. We denote by $\mathsf{RightLCE}_T(i, j)$ and $\mathsf{LeftLCE}_T(i, j)$ rightward and leftward LCE queries for positions $1 \leq i \neq j \leq |T|$, respectively. An *outward LCE* query is, given two positions $1 \leq i < j \leq |T|$, to compute $lcp((T[1..i])^R, T[j..|T|])$. We denote by $\mathsf{OutLCE}_T(i, j)$ an outward LCE query for positions $i < j$ in the string T.

Manacher [16] showed an elegant online algorithm which computes all maximal palindromes of a given string T of length n in $O(n)$ time. An alternative offline approach is to use outward LCE queries for $2n - 1$ pairs of positions in T. Using the suffix tree [21] for string $T\$T^R\#$ enhanced with a lowest common ancestor data structure [4], where $\$$ and $\#$ are special characters which do not appear in T, each outward LCE query can be answered in $O(1)$ time. For any integer alphabet of size polynomial in n, preprocessing for this approach takes $O(n)$ time and space [8, 13].

A palindromic substring P of a string T is called a *longest substring palindrome* (*LSPal*) if there are no palindromic substrings of T which are longer than P. Since any LSPal of T is always a maximal palindrome of T, we can find all LSPals and their lengths in $O(n)$ time.

In this paper, we consider the problem of finding an LSPal after a substring of T is replaced with another string. The problem is formally defined as follows:

▶ **Definition 1** (Longest substring palindrome query after block edit).
Preprocess: *A string T of length n.*
Query input: *An interval $[i, j] \subseteq [1, n]$ and a string X of length ℓ.*
Query output: *(The length of) a longest substring palindrome in the edited string $T' = T[1..i - 1]XT[j + 1..n]$.*

The query in the above problem is called an *ℓ-block edit longest substring palindrome* query (*ℓ-ELSPal* query in short). In the following section, we will propose an $O(n)$-time and space preprocessing scheme such that subsequent ℓ-ELSPal queries can be answered in $O(\ell + \log \log n)$ time. We remark that in this problem string edits are only given as *queries*, i.e., we do not explicitly rewrite the original string T into T' and T remains unchanged for further queries. We also remark that in our problem the length ℓ of a substring X that substitutes a given interval (substring) can be arbitrary.

Let ℓ' be the length of the substring to be replaced, i.e., $\ell' = j - i + 1$. Our block-wise edit operation generalizes character-wise substitution when $\ell' > 0$ and $\ell > 0$, character-wise insertion when $\ell' = 0$ and $\ell > 0$, and character-wise deletion when $\ell' > 0$ and $\ell = 0$.

The following properties of palindromes are useful in our algorithms.

▶ **Lemma 2.** *Any border B of a palindrome P is also a palindrome.*

Proof. Since P is a palindrome, for any $1 \leq m \leq |P|$, clearly $P[1..m] = (P[|P| - m + 1..|P|])^R$. Since B is a border of P, we have that $B = P[1..|B|] = (P[|P| - |B| + 1..|P|])^R = B^R$. ◀

Let T be a string of length n. For each $1 \leq i \leq n$, let $\mathit{MaxPalEnd}_T(i)$ denote the set of maximal palindromes of T that end at position i. Let $\mathbf{S}_i = s_1, \ldots, s_g$ be the sequence of lengths of maximal palindromes in $\mathit{MaxPalEnd}_T(i)$ sorted in increasing order, where $g = |\mathit{MaxPalEnd}_T(i)|$. Let d_j be the progression difference for s_j, i.e., $d_j = s_j - s_{j-1}$ for $2 \leq j \leq g$. For convenience, let $d_1 = 0$. We use the following lemma which is based on periodic properties of maximal palindromes ending at the same position.

▶ **Lemma 3** ([9]).

(i) *For any $1 \leq j < g$, $d_{j+1} \geq d_j$.*

(ii) *For any $1 < j < g$, if $d_{j+1} \neq d_j$, then $d_{j+1} \geq d_j + d_{j-1}$.*

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

(iv) *If $t \geq 2$, then the common difference d is a period of every maximal palindrome which ends at position i in T and whose length belongs to the arithmetic progression $\langle s, d, t \rangle$.*

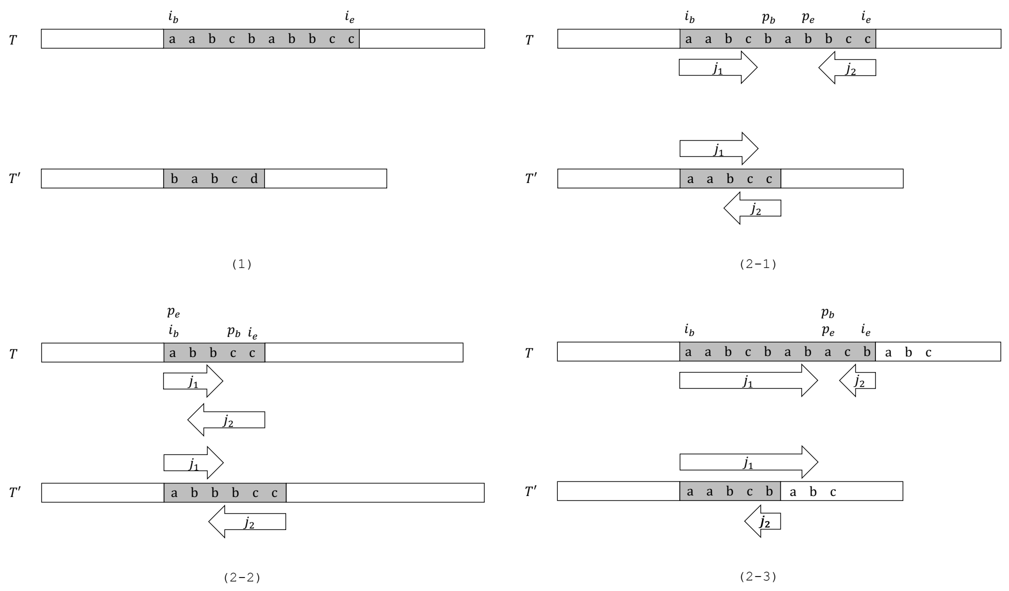
See also Fig. 3 in the next section. Each arithmetic progression $\langle s, d, t \rangle$ is called a *group* of maximal palindromes. Similar arguments hold for the set $\mathit{MaxPalBeg}_T(i)$ of maximal palindromes of T that begin at position i. For all $1 \leq i \leq n$ we can compute $\mathit{MaxPalEnd}_T(i)$ and $\mathit{MaxPalBeg}_T(i)$ in total $O(n)$ time: After computing all maximal palindromes of T in $O(n)$ time, we can bucket sort all the maximal palindromes with their ending positions and with their beginning positions in $O(n)$ time each.

3 Algorithm for ℓ-ELSPal

Consider to substitute a string X of length ℓ for the substring $T[i_b..i_e]$ beginning at position i_b and ending at position i_e, where $i_e - i_b + 1 = \ell'$ and $X \neq T[i_b..i_e]$. Let $T' = T[1..i_b - 1]XT[i_e + 1..n]$ be the string after the edit.

In order to compute (the lengths of) maximal palindromes that are affected by the block-wise edit operation, we need to know the first (leftmost) mismatching position between T and T', and that between T^R and T'^R. Let h and l be the smallest integers such that $T[ht] \neq T'[ht]$ and $T^R[l] \neq T'^R[l]$, respectively. If such h does not exist, then let $h = \min\{|T|, |T'|\} + 1$. Similarly, if such l does not exist, then let $l = \min\{|T|, |T'|\} + 1$. Let $j_1 = \mathit{lcp}(T[i_b..n], XT[i_e..n])$, $j_2 = \mathit{lcp}((T[1..i_e])^R, (T[1..i_b]X)^R)$, $p_b = i_b + j_1$, and $p_e = i_e - j_2$. There are two cases: (1) If $j_1 = j_2 = 0$, then the first and last characters of $T[i_b..i_e]$ differ from those of X. In this case, we have $i_b = h$ and $i_e = n - l + 1$. We use these positions i_b and i_e to compute maximal palindromes after the block-wise edit. (2) Otherwise, we have $p_b = i_b + j_1 = h$ and $p_e = i_e - j_2 = n - l + 1$. We use these positions p_b and p_e to compute maximal palindromes after the block-wise edit. See Figure 1 for illustration.

In the next subsection, we describe our algorithm for Case (1). Case (2) can be treated similarly, by replacing i_b and i_e with p_b and p_e, respectively. Our algorithm can handle the case where $p_e < p_b$. Remark that p_b and p_e can be computed in $O(\ell)$ time by naïve character comparisons and a single LCE query each.



■ **Figure 1** Illustration for the mismatching position between T and T', and that between T^R and T'^R. In particular, (2-2) is the sub-case of Case (2) with $p_e < p_b$, and (2-3) is the sub-case of Case (2) with $j_1 > \ell$.

We remark that the longest extension of the maximal palindromes in T which are unchanged or shortened after the block edit can be found $O(1)$ time upon query, after $O(n)$-time and space preprocessing, using similar techniques to the 1-ELSPal queries from our previous work [9]. Also, the longest maximal palindromes that have centers in the new block can be computed in $O(\ell)$ time after $O(n)$-time and space preprocessing, in a similar way to our previous algorithm for the 1-ELSPal [9]. Hence, in this paper we concentrate on the maximal palindromes of T which get extended after the block edit. The next observation describes such maximal palindromes.

▶ **Observation 4** ([9]). *For any $s \in \mathit{MaxPalEnd}_T(i_b - 1)$, the corresponding maximal palindrome $T[i_b - s..i_b - 1]$ centered at $\frac{2i_b - s - 1}{2}$ gets extended in T' iff $\mathsf{OutLCE}_{T'}(i_b - s - 1, i_b) \geq 1$. Similarly, for any $s \in \mathit{MaxPalBeg}_T(i_e + 1)$, the corresponding maximal palindrome $T[i_e + 1..i_e + s]$ centered at $\frac{2i_e + s + 1}{2}$ gets extended in T' iff $\mathsf{OutLCE}_{T'}(i_e, i_e + s + 1) \geq 1$.*

It follows from Observation 4 that it suffices to compute outward LCE queries efficiently for all maximal palindromes which end at position $i_b - 1$ or begin at position $i_e + 1$ in the edited string T'. The following lemma, which is a generalization of Lemma 21 from [9], shows how to efficiently compute the extensions of any given maximal palindromes that end at position $i_b - 1$. Those that begin at position $i_e + 1$ can be treated similarly.

▶ **Lemma 5.** *Let T be a string of length n over an integer alphabet of size polynomially bounded in n. We can preprocess T in $O(n)$ time and space so that later, given a list of any f maximal palindromes from $\mathit{MaxPalEnd}_T(i_b - 1)$, we can compute in $O(\ell + f)$ time the extensions of those f maximal palindromes in the edited string T', where ℓ is the length of a new block.*

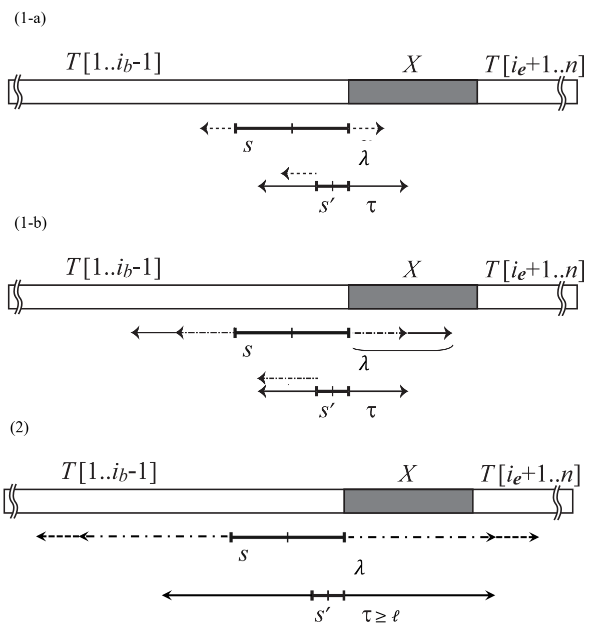


Figure 2 Illustration for Lemma 5, where solid arrows represent the matches obtained by naïve character comparisons, and broken arrows represent those obtained by LCE queries. This figure only shows the case where $s' < s$, but the other case where $s' > s$ can be treated similarly.

Proof. Let us remark that the maximal palindromes in the list can be given to our algorithm in any order. Firstly, we compute the extensions of given maximal palindromes from the list until finding the first maximal palindrome whose extension τ is at least one, and let s' be the length of this maximal palindrome. Namely, $s' + 2\tau$ is the length of the extended maximal palindrome for s', and the preceding maximal palindromes (if any) were not extended. Let s be the length of the next maximal palindrome from the list after s', and now we are to compute the extension λ for s. See also Figure 2. There are two cases: (1) If $0 < \tau < \ell$, then we first compute $\delta = \mathsf{LeftLCE}_T(i_b - s - 1, i_b - s' - 1)$. We have two sub-cases: (1-a) If $\delta < \tau$, then $\lambda = \delta$. (1-b) Otherwise ($\delta \geq \tau$), then we know that λ is at least as large as τ. We then compute the remainder of λ by naïve character comparisons. If the character comparison reaches the end of X, then the remainder of λ can be computed by $\mathsf{OutLCE}_T(i_b - s - \ell - 1, i_e + 1)$. Then we update τ with λ. (2) If $\tau \geq \ell$, then we can compute λ by $\mathsf{LeftLCE}_T(i_b - s - 1, i_b - s' - 1)$, and if this value is at least ℓ, then by $\mathsf{OutLCE}_T(i_b - s - \ell - 1, i_e + 1)$. The extensions of the following palindromes can also be computed similarly.

The following maximal palindromes from the list after s can be processed similarly. After processing all the f maximal palindromes in the given list, the total number of matching character comparisons is at most ℓ since each position of X is involved in at most one matching character comparison. Also, the total number of mismatching character comparisons is $O(f)$ since for each given maximal palindrome there is at most one mismatching character comparison. The total number of LCE queries on the original text T is $O(f)$, each of which can be answered in $O(1)$ time. Thus, it takes $O(\ell + f)$ time to compute the length of the f maximal palindromes of T' that are extended after the block edit. ◀

However, there can be $\Omega(n)$ maximal palindromes beginning or ending at each position of a string of length n. In what follows, we show how to reduce the number of maximal palindromes that need to be considered, by using periodic structures of maximal palindromes.

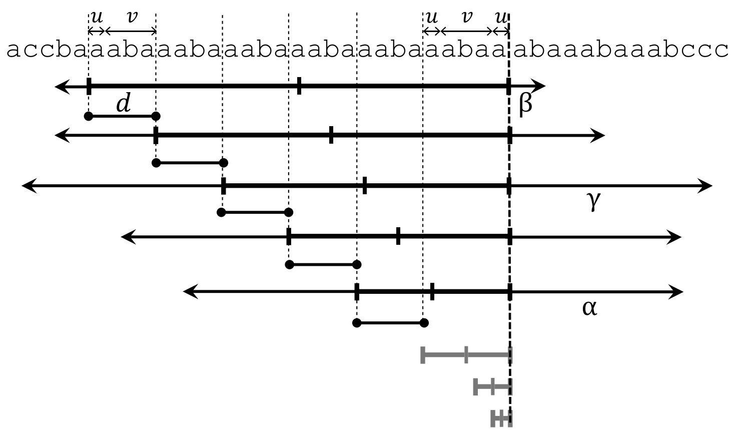


Figure 3 Example for Lemma 6, where Y = accbaaabaaabaaabaaabaaabaaabaa and Z = abaaabaaabccc. Here u = a and v = aba. The first five maximal palindromes $(uv)^k u$ = (aaba)^{k}a with $2 \leq k \leq 5$ belong to the same arithmetic progression (i.e. the same group) with common difference $|uv| = d = 4$. For this group of maximal palindromes, $\alpha = 10$, $\beta = 2$, and $\gamma = 12$. Notice that the sixth maximal palindrome uvu = aabaa belongs to another group since the length difference between it and the seventh one aa is 3.

Let $\langle s, d, t\rangle$ be an arithmetic progression representing a group of maximal palindromes ending at position $i_b - 1$. For each $1 \leq j \leq t$, we will use the convention that $s(j) = s + (j-1)d$, namely s denotes the jth shortest element for $\langle s, d, t\rangle$. For simplicity, let $Y = T[1..i_b - 1]$ and $Z = XT[i_e + 1..n]$. Let $Ext(s(j))$ denote the length of the maximal palindrome that is obtained by extending $s(j)$ in YZ.

▶ **Lemma 6** ([9]). *For any $\langle s, d, t\rangle \subseteq MaxPalEnd_T(i_b - 1)$, there exist palindromes u, v and a non-negative integer p, such that $(uv)^{t+p-1}u$ (resp. $(uv)^p u$) is the longest (resp. shortest) maximal palindrome represented by $\langle s, d, t\rangle$ with $|uv| = d$. Let $\alpha = lcp((Y[1..|Y| - s_1])^R, Z)$ and $\beta = lcp((Y[1..|Y| - s_t])^R, Z)$. If there exists $s_h \in \langle s, d, t\rangle$ such that $s_h + \alpha = s_t + \beta$, then let $\gamma = lcp((Y[1..|Y| - s_h])^R, Z)$. Then, for any $s(j) \in \langle s, d, t\rangle \setminus \{s_h\}$, $Ext(s(j)) = s(j) + 2\min\{\alpha, \beta + (t - j)d\}$. Also, if s_h exists, then $Ext(s_h) = s_h + 2\gamma \geq Ext(s(j))$ for any $j \neq h$.*

See Figure 3 for a concrete example of Lemma 6. It follows from Lemma 6 that it suffices to consider only three maximal palindromes from each group (i.e. each arithmetic progression). Then using Lemma 5, one can compute the longest maximal palindrome that gets extended in $O(\ell + \log n)$ time, and this is exactly how the algorithm from our previous paper [9] works.

To further speed up computation, we take deeper insights into combinatorial properties of maximal palindromes in $MaxPalEnd_T(i_b - 1)$. Let $G_1, \ldots, G_m$ be the list of all groups for the maximal palindromes from $MaxPalEnd_T(i_b - 1)$, which are *sorted in increasing order of their common difference*. When $m = O(\log\log n)$, $O(\ell + \log\log n)$-time queries immediately follow from Lemmas 5 and 6. In what follows we consider the more difficult case where $m = \omega(\log\log n)$. Recall also that $m = O(\log n)$ always holds.

For each $G_r = \langle s_r, d_r, t_r\rangle$ with $1 \leq r \leq m$, let α_r, β_r, γ_r, u_r, and v_r be the corresponding variables used in Lemma 6. If there is only a single element in G_r, let β_r be the length of extension of the palindrome and $\alpha_r = \beta_{r-1}$. For each G_r, let S_r (resp. L_r) denote the shortest (resp. longest) maximal palindrome in G_r, namely, $|S_r| = s_r(1)$ and $|L_r| = s_r(t_r)$.

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 $(1 \leq k \leq m)$ be the unique integer such that G_k is the type-2 group where d_k is the largest common difference among all the type-2 groups. Additionally, let $G'_k = G_k \cup \{u_k v_k u_k, u_k\}$. Note that u_k belongs to one of $G_1, \ldots, G_{k-1}$, and $u_k v_k u_k$ belongs to either G_k or one of $G_1, \ldots, G_{k-1}$. In the special case where $\alpha_k = \beta_k + td_k$, the extensions of u_k and $u_k v_k u_k$ can be longer than the extension of the shortest maximal palindrome in G_k (see Figure 4 for a concrete example). Thus, it is convenient for us to treat $G'_k = G_k \cup \{u_k v_k u_k, u_k\}$ as if it is a single group. We also remark that this set G'_k is defined only for this specific type-2 group G_k.

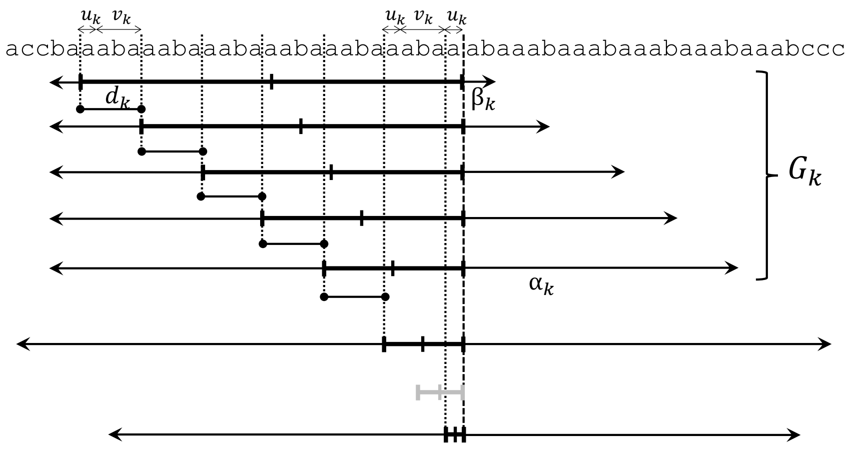


Figure 4 Example for $G'_k = G_k \cup \{u_k v_k u_k, u_k\}$, where the extensions of $u_k v_k u_k$ and u_k are longer than the extensions of any maximal palindromes in G_k.

▶ **Lemma 7.** *There is a longest substring palindrome in the edited string T' that is obtained by extending the maximal palindromes in G_m, G_{m-1}, or G'_k.*

Proof. The lemma holds if the two following claims are true:

▷ Claim (1). The extensions of the maximal palindromes in $G_1, \ldots, G_{k-1}$, except for $u_k v_k u_k$ and u_k, cannot be longer than the extension of the shortest maximal palindrome in G_k.

▷ Claim (2). Suppose both G_m and G_{m-1} are of type-1. Then, the extensions of the maximal palindromes from $G_{k+1}, \ldots, G_{m-2}$, which are also of type-1, cannot be longer than the extensions of the maximal palindromes from G_m or G_{m-1}.

Proof for Claim (1). Here we consider the case where the maximal palindrome $u_k v_k u_k$ does not belong to G_k, which implies that the shortest maximal palindrome S_k in G_k is $(u_k v_k)^2 u_k$ (The other case where $u_k v_k u_k$ belongs to G_k can be treated similarly). Now, $u_k v_k u_k$ belongs to one of $G_1, \ldots, G_{k-1}$. Consider the prefix $P = T[1..i_b - |u_k v_k u_k| - 1]$ of T that immediately precedes $u_k v_k u_k$. The extension of $u_k v_k u_k$ is obtained by $lcp(P^R, Z)$. Consider the prefix $P' = T[1..i_b - |(u_k v_k)^2 u_k|]$ of T that immediately precedes. It is clear that P is a concatenation of P' and $u_k v_k$. Similarly, the prefix $T[1..i_b - |u_k| - 1]$ of T that immediately precedes u_k is a concatenation of P and $u_k v_k$. From Lemma 6 and the definition of G'_k, it suffices for us to consider only the three maximal palindromes from G'_k. For any other maximal palindrome Q from $G_1, \ldots, G_{k-1}$, assume on the contrary that Q gets extended by at least d_k to the left and to the right. If $|u_k v_k| = d_k < |Q| < |u_k v_k u_k|$, then there is an internal occurrence of $u_k v_k$ inside the prefix $(u_k v_k)^2$ of $(u_k v_k)^2 u_k$. Otherwise ($|u_k| < |Q| < |u_k v_k| = d_k$ or $|Q| < |u_k|$), there is an internal occurrence of $u_k v_k$ inside $u_k v_k u_k$. Here we only consider the first case

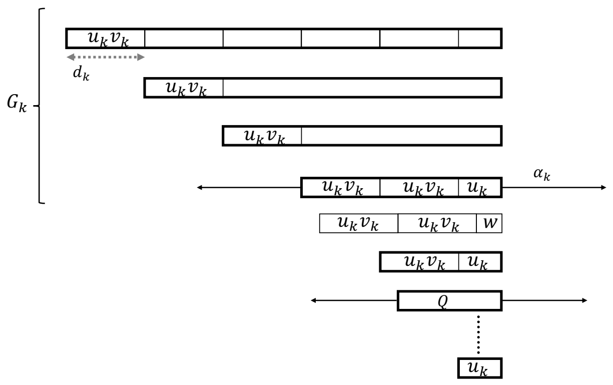


Figure 5 Illustration for the proof for Claim (1) of Lemma 7.

but other cases can be treated similarly. See also Figure 5. This internal occurrence of u_kv_k is immediately followed by u_kv_kw, where w is a proper prefix of u_k with $1 \leq |w| < |u_k|$. Namely, $(u_kv_k)^2w$ is a proper suffix of $(u_kv_k)^2u_k$. On the other hand, $(u_kv_k)^2w$ is also a proper prefix of $(u_kv_k)^2u_k$. Since $(u_kv_k)^2u$ is a palindrome, it now follows from Lemma 2 that $(u_kv_k)^2w$ is also a palindrome. Since $1 \leq |w| < |u_k|$, we have $|(u_kv_k)^2w| > |u_kv_ku_k|$ (note that this inequality holds also when v_k is the empty string). Then, $(u_kv_k)^2w$ is also immediately preceded by u_kv_k because of periodicity and is extended by at least d_k to the left and to the right. Since $T'[i_b] = T[i_b - |(u_kv_k)^2w| - 1]$ and $T'[i_b] \neq T[i_b]$, $(u_kv_k)^2w$ is a maximal palindrome. However this contradicts that $(u_kv_k)^2u_k$ belongs to G_k with common difference $d_k = |u_kv_k|$. Thus Q cannot be extended by d_k nor more to the left and to the right. Since G_k is of type-2, $\alpha_k \geq d_k$. Since $|Q| < |(u_kv_k)^2u_k|$, the extension of Q cannot be longer than the extension for $(u_kv_k)^2u_k$. This completes the proof for Claim (1). ◁

Proof for Claim (2). Consider each group $G_r = \langle s_r, d_r, t_r \rangle$ with $k+1 \leq r \leq m-2$. By Lemma 6, $s_r(t_r) + 2\beta_r$ and $s_r(t_r - 1) + 2\alpha_r$ are the candidates for the longest extensions of the maximal palindromes from G_r. Recall that both G_{m-1} and G_m are of type-1, and that if G_r is of type-1 then G_{r+1} is also of type-1. Now the following sub-claim holds:

▶ **Lemma 8.** *$\beta_r = \alpha_{r+1}$ for any $k+1 \leq r \leq m-2$.*

Proof. If G_{r+1} is a singleton, then by definition $\beta_r = \alpha_{r+1}$ holds. Now suppose $|G_{r+1}| \geq 2$. Since the shortest maximal palindrome S_{r+1} from G_{r+1} is either $(u_{r+1}v_{r+1})^2u_{r+1}$ or $u_{r+1}v_{r+1}u_{r+1}$, the longest maximal palindrome L_r from G_r is either $u_{r+1}v_{r+1}u_{r+1}$ or u_{r+1}. The prefix $T[1..i_b - |L_r| - 1]$ of T that immediately precedes L_r contains $u_{r+1}v_{r+1}$ as a suffix, which alternatively means $(u_{r+1}v_{r+1})^R$ is a prefix of $(T[1..i_b - |L_r| - 1])^R$. Moreover, it is clear that the prefix $T[1..i_b - |S_{r+1}| - 1]$ of T that immediately precedes S_{r+1} contains $u_{r+1}v_{r+1}$ as a suffix since $|G_{r+1}| \geq 2$. In addition, $\alpha_{r+1} < d_{r+1} = |u_{r+1}v_{r+1}|$ since G_{r+1} is of type-1. From the above arguments, we get $\beta_r = \alpha_{r+1}$. ◀

Since $\beta_r = \alpha_{r+1}$ and $\alpha_{r+1} < d_{r+1}$, we have $s_r(t_r) + 2\beta_r < s_r(t_r) + 2d_{r+1}$. In addition, $s_r(t_r - 1) + 2\alpha_r < s_r(t_r - 1) + 2d_r = s_r(t_r) + d_r$. It now follows from $d_r < d_{r+1}$ that $s_r(t_r) + d_r < s_r(t_r) + 2d_{r+1}$. Since the lengths of the maximal palindromes and their common differences are arranged in increasing order in the groups $G_{k+1}, \ldots .G_{m-2}$, we have that the longest extension from $G_{k+1}, \ldots .G_{m-2}$ is shorter than $s_{m-2}(t_{m-2}) + 2d_{m-1}$. Since $d_{m-1} < d_m$, we have

$$s_{m-2}(t_{m-2}) + 2d_{m-1} < s_{m-2}(t_{m-2}) + d_{m-1} + d_m \leq s_{m-1}(t_{m-1}) + d_m \leq s_m = s_m(1).$$

This means that the longest extended maximal palindrome from the type-1 groups G_{k+1}, …, G_{m-2} cannot be longer than the original length of the maximal palindrome from G_m before the extension. This completes the proof for Claim (2). ◁

◀

It follows from Lemmas 5, 6 and 7 that given G_k, we can compute in $O(\ell)$ time the length of the LSPal of T' after the block edit. What remains is how to quickly find G_k, that has the largest common difference among all the type-2 groups. Note that a simple linear search from G_m or G_1 takes $O(\log n)$ time, which is prohibited when $\ell = o(\log n)$. In what follows, we show how to find G_k in $O(\ell + \log \log n)$ time.

Recall that $T[1..i_b - |L_{r-1}| - 1]$ which immediately precedes S_r contains $u_r v_r$ as a suffix. Thus, $(u_r v_r)^R$ is a prefix of $(T[1..i_b - |L_{r-1}| - 1])^R$. We have the following observation.

▶ **Observation 9.** *Let $W_1 = (T[1..i_b - 1])^R$, and $W_r = (T[1..i_b - |L_{r-1}| - 1])^R$ for $2 \leq r \leq m$. Let W be the string such that $lcp(W_r, Z)$ is the largest for all $1 \leq r \leq m$ (i.e. for all groups $G_1, \ldots, G_m$), namely, $W = \arg\max_{1 \leq r \leq m} lcp(W_r, Z)$. Then $G_k = G_x$ such that*

(a) *$(u_x v_x)^R$ is a prefix of W,*
(b) *$d_x \leq lcp(W, Z)$, and*
(c) *d_x is the largest among all groups that satisfy Conditions (a) and (b).*

Due to Observation 9, the first task is to find W.

▶ **Lemma 10.** *W can be found in $O(\ell + \log \log n)$ time after $O(n)$-time and space preprocessing.*

Proof. We preprocess T as follows. For each $1 \leq i \leq n$, let $G_1, \ldots, G_m$ be the list of groups that represent the maximal palindromes ending at position i in T. Let $W_1 = (T[1..i])^R$ and $W_r = (T[1..i - |L_{r-1}|])^R$ for $2 \leq r \leq m$. Let $\mathcal{A}_i$ be the *sparse suffix array* of size $m = O(\log i)$ such that $\mathcal{A}_i[j]$ stores the jth lexicographically smallest string in $\{W_1, \ldots, W_m\}$. We build $\mathcal{A}_i$ with the LCP array $\mathcal{L}_i$. Since there are only $2n - 1$ maximal palindromes in T, $\mathcal{A}_i$ for all positions $1 \leq i \leq n$ can easily be constructed in a total of $O(n)$ time from the full suffix array of T. The LCP array $\mathcal{L}_i$ for all $1 \leq i \leq n$ can also be computed in $O(n)$ total time from the LCP array of T enhanced with a range minimum query (RMQ) data structure [4].

To find W, we binary search $\mathcal{A}_{i_b - 1}$ for $Z[1..\ell] = X$ in a similar way to pattern matching on the suffix array with the LCP array [17]. This gives us the range of $\mathcal{A}_{i_b - 1}$ such that the corresponding strings have the longest common prefix with X. Since $|\mathcal{A}_{i_b - 1}| = O(\log n)$, this range can be found in $O(\ell + \log \log n)$ time. If the longest prefix found above is shorter than ℓ, then this prefix is W. Otherwise, we perform another binary search on this range for $Z[\ell + 1..|Z|] = T[i_e + 1..n]$, and this gives us W. Here each comparison can be done in $O(1)$ time by an outward LCE query on T. Hence, the longest match for $Z[\ell + 1..|Z|]$ in this range can also be found in $O(\log \log n)$ time. Overall, W can be found in $O(\ell + \log \log n)$ time. ◀

▶ **Lemma 11.** *We can preprocess T in $O(n)$ time and space so that later, given W for a position in T, we can find G_k for that position in $O(\log \log n)$ time.*

Proof. Let $\mathcal{D}_i$ be an array of size $|\mathcal{A}_i|$ such that $\mathcal{D}_i[j]$ stores the value of $d_r = |u_r v_r|$, where W_r is the lexicographically jth smallest string in $\{W_1, \ldots, W_m\}$. Let $\mathcal{R}_i$ be an array of size $|\mathcal{A}_i|$ where $\mathcal{R}_i[j]$ stores a sorted list of common differences $d_r = |u_r v_r|$ of groups G_r, such that G_r stores maximal palindromes ending at position i and $(u_r v_r)^R$ is a prefix of the string $\mathcal{A}_i[j]$. Clearly, for any j, $\mathcal{D}_i[j] \subseteq \mathcal{R}_i$. See also Figure 6 for an example of $\mathcal{R}_i$.

Suppose that we have found W by Lemma 10, and let j be the entry of $\mathcal{A}_{i_b-1}$ where the binary search for X terminated. We then find the largest d_x that satisfies Condition (b) of Observation 9, by binary search on the sorted list of common differences stored at $\mathcal{R}_{i_b-1}[j]$. This takes $O(\log\log n)$ time since the list stored at each entry of $\mathcal{R}_{i_b-1}$ contains at most $|A_{i_b-1}| = O(\log n)$ elements.

We remark however that the total number of elements in $\mathcal{R}_i$ is $O(\log^2 i)$ since each entry $\mathcal{R}_i[j]$ can contain $O(\log i)$ elements. Thus, computing and storing $\mathcal{R}_i$ explicitly for all text positions $1 \le i \le n$ can take superlinear time and space.

Instead of explicitly storing $\mathcal{R}_i$, we use a tree representation of $\mathcal{R}_i$, defined as follows: The tree consists of exactly $m = |\mathcal{A}_i|$ leaves and exactly m non-leaf nodes. Each leaf corresponds to a distinct entry $j = 1, \ldots, m$, and each non-leaf node corresponds to a value from $\mathcal{D}_i$. Each leaf j is contained in a (sub)tree rooted at a node with $d \in \mathcal{D}_i$, iff there is a maximal interval $[j'..j'']$ such that $j' \le j \le j''$ and $\mathcal{L}_i[j+1] \ge \mathcal{D}_i[j]$. We associate each node with this maximal interval. Since we have assumed $d_1 = 0$, the root stores 0 and it has at most σ children. See Figure 6 that illustrates a concrete example for $T[1..i_b-1] = dddF_7^4F_6^2F_5^2F_4F_3^3F_2^2F_1^3$ with $i_b = 3451$, where

$$
\begin{aligned}
F_1 &= a\\
F_2 &= {F_1^3}^R b\\
F_3 &= {F_1^3}^R {F_2^2}^R\\
F_4 &= {F_1^3}^R {F_2^2}^R {F_3^2}^R F_2\\
F_5 &= {F_1^3}^R {F_2^2}^R {F_3^3}^R {F_4}^R c\\
F_6 &= {F_1^3}^R {F_2^2}^R {F_3^3}^R {F_4}^R {F_5}^R c F_4 F_3^2\\
F_7 &= {F_1^3}^R {F_2^2}^R {F_3^3}^R {F_4}^R {F_5^2}^R {F_6}^R {F_3^2}^R {F_4}^R c F_5 F_4 F_3^3 F_2^2 F_1^2.
\end{aligned}
$$

We remark that $F_7^4F_6^2F_5^2F_4F_3^3F_2^2F_1^3$, $F_7^3F_6^2F_5^2F_4F_3^3F_2^2F_1^3$, $\ldots$, F_1 are suffix palindromes of $T[1..i_b-1]$.

We can easily construct this tree in time linear in its size $m = |\mathcal{A}_i|$, in a bottom up manner. First, we create leaves for all entries $j = 1, \ldots, m$. Next, we build the tree in a bottom-up manner, by performing the following operations in decreasing order of $\mathcal{D}_i[j]$.

(1) Create a new node with $\mathcal{D}_i[j]$, and connect this node with the highest ancestor of leaf j.

(2) We check $j' < j$ in decreasing order, and connect the new node with the highest ancestor of leaf j' iff $\mathcal{L}_i[j'+1] \ge \mathcal{D}_i[j]$. We skip the interval corresponding to this ancestor, and perform the same procedure until we find j' that does not meet the above condition. We do the same for $j'' > j$.

Since each node is associated with its corresponding interval in the LCP array, it suffices for us to check the conditions $\mathcal{L}_i[j'+1] \ge \mathcal{D}_i[j]$ and $\mathcal{L}_i[j''] \ge \mathcal{D}_i[j]$ only at either end of the intervals that we encounter. Clearly, in the path from the root to leaf j, the values in $\mathcal{R}_j[j]$ appear in increasing order. Thus, we can find the largest d_x that satisfies Condition (b) of Observation 9, by a binary search on the corresponding path in the tree. We augment the tree with a level ancestor data structure [7, 5], so that each binary search takes logarithmic time in the tree height, namely $O(\log\log n)$ time. The size of the tree for position i is clearly bounded by the number of maximal palindromes ending at position i. Thus, the total size and construction time for the trees for all positions in T is $O(n)$. ◀

j	$W_{\mathcal{A}_i[1]}, \ldots, W_{\mathcal{A}_i[m]}$	$\mathcal{D}_i$	$\mathcal{L}_i$	$\mathcal{R}_i$
1	`abaaabaaaaaabaaaba` $\cdots$	689	-	0,1,689
2	`aabaaabaaaaaabaaab` $\cdots$	1	1	0,1
3	`aaabaaabaaaaaabaaa` $\cdots$	223	2	0,1,11,223
4	`aaabaaabaaaaaabaaa` $\cdots$	0	22	0,1,11
5	`aaabaaabaaaaaabaaa` $\cdots$	11	33	0,1,11
6	`baaabaaaaaabaaabaa` $\cdots$	4	0	0,4
7	`baaaaaabaaabaaaaaa` $\cdots$	37	4	0,4,37
8	`caaabaaabaaaaaabaa` $\cdots$	82	0	0,82

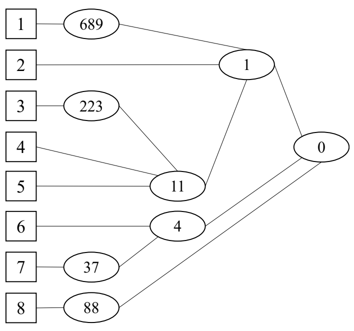


Figure 6 Example for $\mathcal{R}_i$ (left) and its corresponding tree (right). The remaining parts of the strings $W_{\mathcal{A}_i[1]}, \ldots, W_{\mathcal{A}_i[m]}$ are omitted due to lack of space.

By Lemma 5, 6, 7 and 11, we can compute in $O(\ell + \log\log n)$ time the length of the LSPal of T' that are extended after the block edit.

Consequently we obtain the following theorem:

▶ **Theorem 12.** *There is an $O(n)$-time and space preprocessing for the ℓ-ELSPal problem such that each query can be answered in $O(\ell + \log\log n)$ time, where ℓ denotes the length of the block after edit.*

Note that the time complexity for our algorithm is independent of the length of the original block to edit. Also, the length ℓ of a new block is arbitrary.

References

1 Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest Common Factor After One Edit Operation. In *SPIRE 2017*, pages 14–26, 2017.

2 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest Common Factor Made Fully Dynamic. *CoRR*, abs/1804.08731, 2018. `arXiv:1804.08731`.

3 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141:163–173, 1995.

4 Michael A. Bender and Martin Farach-Colton. The LCA Problem Revisited. In *LATIN 2000*, pages 88–94, 2000.

5 Michael A. Bender and Martin Farach-Colton. The Level Ancestor Problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

6 Petra Berenbrink, Funda Ergün, Frederik Mallmann-Trenn, and Erfan Sadeqi Azer. Palindrome Recognition In The Streaming Model. In *STACS 2014*, pages 149–161, 2014.

7 O. Berkman and U. Vishkin. Finding level-ancestors in trees. *J. Comput. System Sci.*, 48(2):214–230, 1994.

8 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

9 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest substring palindrome after edit. In *CPM 2018*, pages 12:1–12:14, 2018.

10 Pawel Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Tighter Bounds and Optimal Algorithms for All Maximal α-gapped Repeats and Palindromes – Finding All Maximal α-gapped Repeats and Palindromes in Optimal Worst Case Time on Integer Alphabets. *Theory Comput. Syst.*, 62(1):162–191, 2018.

11 Pawel Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemyslaw Uznanski. Tight Tradeoffs for Real-Time Approximation of Longest Palindromes in Streams. In *CPM 2016*, pages 18:1–18:13, 2016.
12 Richard Groult, Élise Prieur, and Gwénaël Richomme. Counting distinct palindromes in a word in linear time. *Inf. Process. Lett.*, 110(20):908–912, 2010.
13 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.
14 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theor. Comput. Sci.*, 410(51):5365–5373, 2009.
15 Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Finding Distinct Subpalindromes Online. In *PSC 2013*, pages 63–69, 2013.
16 Glenn Manacher. A New Linear-Time “On-Line” Algorithm for Finding the Smallest Initial Palindrome of a String. *Journal of the ACM*, 22:346–351, 1975.
17 U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
18 W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient Algorithms to Compute Compressed Longest Common Substrings and Compressed Palindromes. *Theor. Comput. Sci.*, 410(8–10):900–913, 2009.
19 Alexandre H. L. Porto and Valmir C. Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 35:2581–2591, 2002.
20 Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest Lyndon Substring After Edit. In *CPM 2018*, pages 19:1–19:10, 2018.
21 Peter Weiner. Linear Pattern Matching Algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.