Longest substring palindrome after edit

Mitsuru Funakoshi
Department of Physics, 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
https://orcid.org/0000-0002-6856-5185

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

Abstract

It is known that the length of the longest substring palindromes (LSPals) of a given string T of length n can be computed in $O(n)$ time by Manacher's algorithm [J. ACM '75]. 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 $O(\log(\min\{\sigma, \log n\}))$ time after single character substitution, insertion, or deletion, where σ denotes the number of distinct characters appearing in T. We also propose an algorithm that uses $O(n)$ time and space for preprocessing, and answers the length of the LSPals in $O(\ell + \log n)$ time, after an existing substring in T is replaced by a string of arbitrary length ℓ.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases maximal palindromes, edit operations, periodicity, suffix trees

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.12

1 Introduction

Palindromes are strings that read the same forward and backward. The problems of finding palindromes or palindrome-like structures in a given string are fundamental tasks in string processing, and thus have been extensively studied (e.g., see [2, 14, 8, 12, 16, 11, 15, 6] and references therein).

One of the earliest problems regarding palindromes is the *longest substring palindrome* (*LSPal*) problem, which asks to find (the length) of the longest palindromes that appear in a given string. This problem dates back to 1970's [13], and since then it has been popular as a good algorithmic exercise. 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. Since one can compute the maximal palindromes for all such centers in $O(n^2)$ total time by naïve character comparisons, the LSPal problem can also be easily solved in $O(n^2)$ time.

Manacher [13] gave an elegant $O(n)$-time solution to the LSPal problem. Manacher's algorithm uses symmetry of palindromes and character equality comparisons only, and therefore works in $O(n)$ time for any alphabet. It was pointed out in [2] that Manacher's algorithm actually computes all the maximal palindromes in the string. In case where the input string is drawn from a constant size alphabet or an integer alphabet of size polynomial in n, there is an alternative suffix tree [19] based algorithm which takes $O(n)$ time [9]. This algorithm also computes all maximal palindromes.

There is a simple $O(n)$-space data structure representing all of these computed maximal palindromes; simply store their lengths in an array of length $2n-1$ together with the input string T. However, this data structure is apparently not flexible for string edits, since even a single character substitution, insertion, or deletion can significantly break palindromic structures of the string. Indeed, $\Omega(n^2)$ substring palindromes and $\Omega(n)$ maximal palindromes can be affected by a single edit operation (E.g., consider to replace the middle character of string a^n with another character b). Hence, an intriguing question is whether there exists a space-efficient data structure for the input string T which can quickly answer the following query: What is the length of the longest substring palindrome(s), if single character substitution, insertion, or deletion is performed? We call this as a *1-ELSPal query.*

In this paper, we present an algorithm which uses $O(n)$ time and space for preprocessing and $O(\log(\min\{\sigma, \log n\}))$ time for 1-ELSPal queries, where σ is the number of distinct characters appearing in T. We also consider a more general variant of 1-ELSPal queries, where an existing substring in the input string T can be replaced with a string of arbitrary length ℓ, called an *ℓ-ELSPal queries.* We present an algorithm which uses $O(n)$ time and space for preprocessing and $O(\ell + \log n)$ time for ℓ-ELSPal queries. Our results are valid for string of length n over an integer alphabet of size polynomial in n. All bounds in this paper are in the worst case unless otherwise stated.

Related work

This line of research was recently initiated by Amir et al. [1] for the *longest common factor* (*LCF*) of two strings. For two strings S and T of length at most n, they proposed a data structure of $O(n \log^3 n)$ space which answers in $O(\log^3 n)$ time the length of the LCF of S and the string T' obtained by a single character edit operation on T. Their data structure can be constructed in $O(n \log^4 n)$ expected time.

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 $1 \leq i \leq |T| - p$.

The *run length* (*RL*) factorization of a string T is a sequence $f_1, \ldots, f_m$ of maximal runs of the same characters such that $T = f_1 \cdots f_m$ (namely, each RL factor f_j is a repetition of the same character a_j with $a_j \neq a_{j+1}$). For each position $1 \leq i \leq n$ in T, let $\mathit{RLFBeg}(i)$ and $\mathit{RLFEnd}(i)$ denote the beginning and ending positions of the RL factor that contains the position i, respectively. One can easily compute in $O(n)$ time the RL factorization of string T of length n together with $\mathit{RLFBeg}(i)$ and $\mathit{RLFEnd}(i)$ for all positions $1 \leq i \leq n$.

Let T^R denote the reversed string of T, i.e., $T^R = T[|T|] \cdots T[1]$. A string T is called a *palindrome* if $T = T^R$. 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.

Let $\mathit{PrePals}(T)$ and $\mathit{SufPals}(T)$ denote the sets of prefix palindromes and suffix palindromes of T, respectively. 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 $j = |T|$. Clearly, prefix palindromes and suffix palindromes of T are maximal palindromes of T.

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

Manacher [13] 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 [19] for string $T\$T^R\#$ enhanced with a lowest common ancestor data structure [10, 17, 3], 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 [5, 9]. Let $\mathcal{M}$ be an array of length $2n - 1$ storing the lengths of maximal palindromes in increasing order of centers. For convenience, we allow the index for $\mathcal{M}$ to be an integer or a half-integer from 1 to n, so that $\mathcal{M}[i]$ stores the length of the maximal palindrome of T centered at i.

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 three standard edit operations, i.e., insertion, deletion, and substitution of a character in the input string T of length n. Let T' denote the string after one of the above edit position was performed at a given position. A *1-edit longest substring palindrome* query (*1-ELSPal* query) is to answer (the length of) a longest palindromic substring of T'. In the next section, we will present an $O(n)$-time and space preprocessing scheme such that subsequent *1-ELSPal* queries can be answered in $O(\log(\min\{\sigma, \log n\}))$ time. For any integer $\ell \geq 0$, an *ℓ-block edit longest substring palindrome* query (*ℓ-ELSPal* query), which is a generalization of the 1-ELSPal query, asks (the length of) a longest palindromic substring of T'', where T'' denotes the string after an interval (substring) of T is replaced by a string of length ℓ. 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 $O(\ell + \log n)$ time. We remark that in both problems string edits are only given as *queries*, i.e., we do not explicitly rewrite the original string T into T' nor T'' and T remains unchanged for further queries.

3 Algorithm for 1-ELSPal

In this section, we will show the following result:

▶ **Theorem 1.** *There is an algorithm for the 1-ELSPal problem which uses $O(n)$ time and space for preprocessing, and answers each query in $O(\log(\min\{\sigma, \log n\}))$ time for single character substitution and insertion, and in $O(1)$ time for single character deletion.*

3.1 Periodic structures of maximal palindromes

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 $S_i = s_1, \ldots, s_k$ be the sequence of lengths of maximal palindromes in $\mathit{MaxPalEnd}_T(i)$ sorted in increasing order, where $k = |\mathit{MaxPalEnd}_T(i)|$. Let d_j be the progression difference for s_j, i.e., $d_j = s_{j+1} - s_j$ for $1 \leq j < k$. We use the following lemma which is based on periodic properties of maximal palindromes ending at the same position.

▶ **Lemma 2.**
(i) *For any $1 \leq j < k$, $d_{j+1} \geq d_j$.*
(ii) *For any $1 < j < k$, if $d_{j+1} \neq d_j$, then $d_{j+1} \geq d_j + d_{j-1}$.*
(iii) *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 end at position i in T and whose length belongs to the arithmetic progression $\langle s, d, t\rangle$.*

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.

To prove Lemma 2, we use arguments from the literature [2, 7, 14]. Let us for now consider any string W of length m. In what follows we will focus on suffix palindromes in $\mathit{SufPals}(W)$ and discuss their useful properties. We remark that symmetric arguments hold for prefix palindromes in $\mathit{PrePals}(W)$ as well. Let $S' = s'_1, \ldots, s'_{k'}$ be the sequence of lengths of suffix palindromes of S' sorted in increasing order, where $k' = |\mathit{SufPals}(W)|$. Let d'_j be the progression difference for s'_j, i.e., $d'_j = s'_{j+1} - s'_j$ for $1 \leq j < k'$. Then, the following results are known:

▶ **Lemma 3** ([2, 7, 14])**.**
(A) *For any $1 \leq j' < k'$, $d'_{j'+1} \geq d'_{j'}$.*
(B) *For any $1 < j' < k'$, if $d'_{j'+1} \neq d'_{j'}$, then $d'_{j'+1} \geq d'_{j'} + d'_{j'-1}$.*
(C) *S' can be represented by $O(\log m)$ arithmetic progressions, where each arithmetic progression is a tuple $\langle s', d', t'\rangle$ representing the sequence $s', s'+d', \ldots, s'+(t'-1)d'$ of lengths of t' suffix palindromes with common difference d'.*
(D) *If $t' \geq 2$, then the common difference d' is a period of every suffix palindrome of W whose length belongs to the arithmetic progression $\langle s', d', t'\rangle$.*

The set of suffix palindromes of W whose lengths belong to the same arithmetic progression $\langle s', d', t'\rangle$ is also called a *group* of suffix palindromes. Clearly, every suffix palindrome in the same group has period d', and this periodicity will play a central role in our algorithms.

We are ready to prove Lemma 2.

Proof. It is clear that $\mathit{MaxPalEnd}_T(i) \subseteq \mathit{SufPals}(T[1..i])$, namely,

$$\mathit{MaxPalEnd}_T(i) = \{s' \in \mathit{SufPals}(T[1..i]) \mid T[i-s'] \neq T[i+1], i-s' = 1, \text{ or } i = n\}.$$

The case where $i = n$ is trivial, and hence in what follows suppose that $i < n$. Let $c = T[i+1]$, and for a group $\langle s', d', t'\rangle$ of suffix palindromes let $a = T[i-s']$ and $b = T[i-s'-(t'-1)d']$, namely, a (resp. b) is the character that immediately precedes the shortest (resp. longest) palindrome in the group (notice that $a = b$ when $t' = 1$). Then, it follows from Lemma 3 (D) that $s', s'+d', \ldots, s'+(t'-2)d' \in \mathit{MaxPalEnd}_T(i)$ iff $a \neq c$. Also,

$s' + (t' - 1)d' \in \mathit{MaxPalEnd}_T(i)$ iff $b \neq c$. Therefore, for each group of suffix palindromes of $T[1..i]$, there are only four possible cases: (1) all members of the group are in $\mathit{MaxPalEnd}_T(i)$, (2) all members but the longest one are in $\mathit{MaxPalEnd}_T(i)$, (3) only the longest member is in $\mathit{MaxPalEnd}_T(i)$, or (4) none of the members is in $\mathit{MaxPalEnd}_T(i)$.

Now, it immediately follows from Lemma 3 that (i) $d_{j+1} \geq d_j$ for $1 \leq j < k$ and (ii) $d_{j+1} \geq d_j + d_{j-1}$ holds for $1 < j < k$. Properties (iii) and (iv) also follow from the above arguments and Lemma 3. ◀

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.

3.2 Algorithm for substitutions

In what follows, we will present our algorithm to compute the length of the LSPals after single character substitution. Our algorithm can also return the occurrence of an LSPal.

Let i be any position in the string T of length n and let $c = T[i]$. Also, let $T' = T[1..i-1]c'T[i+1..n]$, i.e., T' is the string obtained by substituting character c' for the original character $c = T[i]$ at position i. To compute the length of the LSPals of T', it suffices to consider maximal palindromes of T'. Those maximal palindromes of T' will be computed from the maximal palindromes of T.

The following observation shows that some maximal palindromes of T remain unchanged after character substitution at position i.

▶ **Observation 4** (Unchanged maximal palindromes after single character substitution). *For any position $1 \leq j < i$, $\mathit{MaxPalEnd}_{T'}(j) = \mathit{MaxPalEnd}_T(j)$. For any position $i < j \leq n$, $\mathit{MaxPalBeg}_{T'}(j) = \mathit{MaxPalBeg}_T(j)$.*

By Observation 4, for each position i $(1 \leq i \leq n)$ of T, we precompute the largest element of $\bigcup_{1 \leq j < i} \mathit{MaxPalEnd}_T(j)$ and that of $\bigcup_{i < j \leq n} \mathit{MaxPalBeg}_T(j)$, and store the larger one in the ith position of an array $\mathcal{U}$ of length n. $\mathcal{U}[i]$ is a candidate for the solution after the substitution at position i. For each position i, $\bigcup_{1 \leq j < i} \mathit{MaxPalEnd}_T(j)$ contains the lengths of all maximal palindromes which end to the left of i, and $\bigcup_{i < j \leq n} \mathit{MaxPalBeg}_T(j)$ contains the lengths of all maximal palindromes which begin to the right of i. Thus, by simply scanning $\mathit{MaxPalEnd}_T(j)$ for increasing $j = 1, \ldots, n$ and $\mathit{MaxPalBeg}_T(j)$ for decreasing $j = n, \ldots, 1$, we can compute $\mathcal{U}[i]$ for every position $1 \leq i \leq n$. Since there are only $2n - 1$ maximal palindromes in string T, it takes $O(n)$ time to compute the whole array $\mathcal{U}$.

Next, we consider maximal palindromes of the original string T whose lengths are extended in the edited string T'. As above, let i be the position where a new character c' is substituted for the original character $c = T[i]$. In what follows, let σ denote the number of distinct characters appearing in T.

▶ **Observation 5** (Extended maximal palindromes after single character substitution). *For any $s \in \mathit{MaxPalEnd}_T(i-1)$, the corresponding maximal palindrome $T[i-s..i-1]$ centered at $\frac{2i-s-1}{2}$ gets extended in T' iff $T[i-s-1] = c'$. Similarly, for any $p \in \mathit{MaxPalBeg}_T(i+1)$, the corresponding maximal palindrome $T[i+1..i+p]$ centered at $\frac{2i+p+1}{2}$ gets extended in T' iff $T[i+p+1] = c'$.*

▶ **Lemma 6.** *Let T be a string of length n over an integer alphabet of size polynomial in n. It is possible to preprocess T in $O(n)$ time and space so that later we can compute in $O(\log(\min\{\sigma, \log n\}))$ time the length of the longest maximal palindromes in T' that are extended after substitution of a character.*

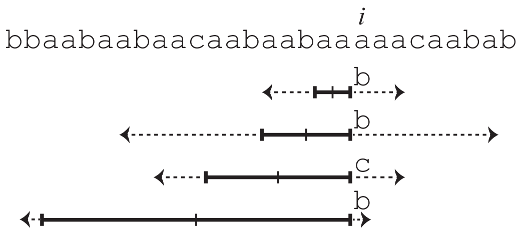

Figure 1 Example for Lemma 6, with string `bbaabaabaacaabaabaaaaacaabab` where the character `a` at position $i = 20$ is to be substituted. There are four maximal palindromes ending at position 19, whose lengths are represented by two groups $\langle 2, 3, 3\rangle$ and $\langle 17, 9, 1\rangle$. For the first group, `c` precedes the longest maximal palindrome and `b` precedes all the other maximal palindromes. The second group contains only one maximal palindrome and `b` precedes it. The largest extended lengths are 21 for `b`, and 14 for `c`. Thus we have $\mathcal{E}_i = [(\mathtt{b}, 21), (\mathtt{c}, 14), (\hat{c}, 17)]$, where 17 is the length of the longest maximal palindrome ending at position 19 in the original string.

Proof. By Observation 5, we consider maximal palindromes corresponding to $\mathit{MaxPalEnd}_T(i-1)$. Those corresponding to $\mathit{MaxPalBeg}_T(i+1)$ can be treated similarly. Let $\langle s, d, t\rangle$ be an arithmetic progression representing a group of maximal palindromes in $\mathit{MaxPalEnd}_T(i-1)$. Let us assume that the group contains more than 1 member (i.e., $t \geq 2$) and that $i-s \geq 2$, since the case where $t = 1$ or $i-s = 1$ is easier to deal with. Let P_j denote the jth shortest member of the group, i.e., $P_1 = T[i-s..i-1]$ and $P_t = T[i-s-(t-1)d..i-1]$. Then, it follows from Lemma 2 (iv) that if a is the character immediately preceding the occurrence of P_1 (i.e., $a = T[i-s-1]$), then a also immediately precedes the occurrences of $P_2, \ldots, P_{t-1}$. Hence, by Observation 5, P_j $(2 \leq j < t)$ gets extended in the edited text T' iff $c' = a$. Similarly, P_t gets extended iff $c' = b$, where b is the character immediately preceding the occurrence of P_t. For each $1 \leq j \leq t$ the final length of the extended maximal palindrome can be computed in $O(1)$ time by a single outward LCE query $\mathsf{OutLCE}(i-s-(j-1)d-2, i+1)$. Let P'_j denote the extended maximal palindrome for each $1 \leq j \leq t$.

The above arguments suggest that for each group of maximal palindromes, there are at most two distinct characters that can extend those palindromes after single character substitution. For each position i in T, let Σ_i denote the set of characters which can extend maximal palindromes w.r.t. $\mathit{MaxPalEnd}_T(i-1)$ after character substitution at position i. It now follows from Lemma 2 and from the above arguments that $|\Sigma_i| = O(\min\{\sigma, \log i\})$. Also, when any character in $\Sigma \setminus \Sigma_i$ is given for character substitution at position i, then no maximal palindromes w.r.t. $\mathit{MaxPalEnd}_T(i-1)$ are extended.

For each maximal palindrome P of T, let (i, c, l) be a tuple such that i is the ending position of P, and l is the length of the extended maximal palindrome P' after the immediately following character $T[i+1]$ is substituted for the character $c = T[i-|P|-1]$ which immediately precedes the occurrence of P in T. We then radix-sort the tuples (i, c, l) for all maximal palindromes in T as 3-digit numbers. This can be done in $O(n)$ time since T is over an integer alphabet of size polynomial in n. Then, for each position i, we compute the maximum value l_c for each character c. Since we have sorted the tuples (i, c, l), this can also be done in total $O(n)$ time for all positions and characters. See Figure 1 for a concrete example.

Let $\hat{c}$ be a special character which represents any character in $\Sigma \setminus \Sigma_i$ (if $\Sigma \setminus \Sigma_i \neq \emptyset$). Since no maximal palindromes w.r.t. $\mathit{MaxPalEnd}_T(i-1)$ are extended by $\hat{c}$, we associate $\hat{c}$ with the length $\ell_{\hat{c}}$ of the longest maximal palindrome w.r.t. $\mathit{MaxPalEnd}_T(i-1)$. We assume that $\hat{c}$ is lexicographically larger than any characters in Σ_i. For each position i we store pairs (c, l_c) in an array $\mathcal{E}_i$ of size $|\Sigma_i| + 1 = O(\min\{\sigma, \log i\})$ in lexicographical order of c. Then, given a character c' to substitute for the character at position i ($1 \leq i \leq n$), we can binary search $\mathcal{E}_i$ for $(c', l_{c'})$ in $O(\log(\min\{\sigma, \log n\}))$ time. If c' is not found in the array, then we take the pair $(\hat{c}, l_{\hat{c}})$ from the last entry of $\mathcal{E}_i$. We remark that $\sum_{i=1}^{n} |\mathcal{E}_i| = O(n)$ since there are $2n - 1$ maximal palindromes in T and for each of them at most two distinct characters contribute to $\sum_{i=1}^{n} |\mathcal{E}_i|$. ◀

Finally, we consider maximal palindromes of the original string T whose lengths are shortened in the edited string T' after substituting a character c' for the original character at position i.

▶ **Observation 7** (Shortened maximal palindromes after single character substitution). *A maximal palindrome $T[b..e]$ of T gets shortened in T' iff $b \leq i \leq e$, $T[b + e - i] \neq c'$, and $i \neq \frac{b+e}{2}$.*

▶ **Lemma 8.** *It is possible to preprocess a string T of length n in $O(n)$ time and space so that later we can compute in $O(1)$ time the length of the longest maximal palindromes of T' that are shortened after substitution of a character.*

Proof. Let $\mathcal{S}$ be an array of length n such that $\mathcal{S}[i]$ stores the length of the longest maximal palindrome that is shortened by the character substitution at position i. To compute $\mathcal{S}$, we preprocess T by scanning it from left to right. Suppose that we have computed $\mathcal{S}[i]$. By Observation 7, we have that $\mathcal{S}[i] = 2(i - \frac{b+e+1}{2})$ where $T[b..e]$ is the longest maximal palindrome of T satisfying the conditions of Observation 7. In other words, $T[b..e]$ is the maximal palindrome of T of which the center $\frac{b+e}{2}$ is the smallest possible under the conditions.

For any position $i < i' \leq e$, we have that $\mathcal{S}[i'] = \mathcal{S}[i]$. For the next position $e + 1$, we can compute $\mathcal{S}[e + 1]$ in amortized $O(1)$ time by simply scanning the array $\mathcal{M}$ from position $\frac{b+e+1}{2}$ to the right until finding the first (i.e., leftmost) entry of $\mathcal{M}$ which stores the length of a maximal palindrome whose ending position is at least $e + 1$. Hence, we can compute $\mathcal{S}$ in $O(n)$ total time and space. ◀

Remark that maximal palindromes of T which do not satisfy the conditions of Observations 5 and 7 are also unchanged in T'.The following lemma summarizes this subsection:

▶ **Lemma 9.** *Let T be a string of length n over an integer alphabet of size polynomial in n. It is possible to preprocess T of length n in $O(n)$ time and space so that later we can compute in $O(\log(\min\{\sigma, \log n\}))$ time the length of the LSPals of the edited string T' after substitution of a character.*

3.3 Algorithm for deletions

Suppose the character at position i is deleted from the string T, and let T'_i denote the resulting string, namely $T'_i = T[1..i - 1]T[i + 1..n]$. Now the RL factorization of T comes into play: Observe that for any $1 \leq i \leq n$, $T'_i = T'_{RLFBeg(i)} = T'_{RLFEnd(i)}$. Thus, it suffices for us to consider only the boundaries of the RL factors for T.

It is easy to see that an analogue of Observation 4 for unchanged maximal palindromes holds, as follows.

▶ **Observation 10** (Unchanged maximal palindromes after single character deletion). *For any position $1 \leq j < RLFEnd(i)$, $MaxPalEnd_{T'}(j) = MaxPalEnd_T(j)$. For any position $RLFBeg(i) < j \leq n$, $MaxPalBeg_{T'}(j) = MaxPalBeg_T(j)$.*

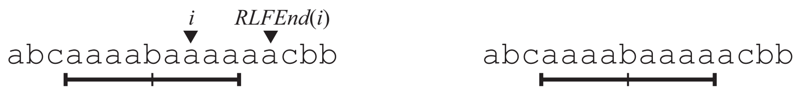

Figure 2 Example for Observation 10. The maximal palindrome aaaabaaaa do not change if the character a at position i is deleted. The result is the same if the character a at position $RLFEnd(i)$ is deleted.

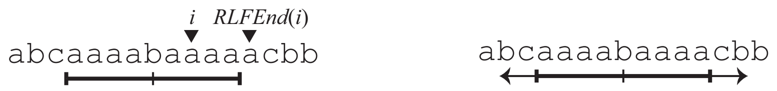

Figure 3 Example for Observation 11. The maximal palindrome aaaabaaaa gets extended to bcaaaabaaaacb if the character a at position i is deleted. The result is the same if the character a at position $RLFEnd(i)$ is deleted.

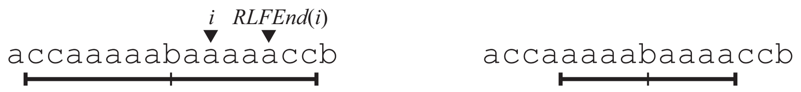

Figure 4 Example for Observation 12. The maximal palindrome ccaaaaabaaaaacc gets shortened to aaaabaaaa if the character a at position i is deleted. The result is the same if the character a at position $RLFEnd(i)$ is deleted.

See Figure 2 for a concrete example of Observation 10.

By the above observation, we can compute the lengths of the longest unchanged maximal palindromes for the boundaries of all RL factors in $O(n)$ time, in a similar way to the case of substitution.

Clearly the new character at position $RLFEnd(i)$ in the string T' after deletion is always $T[RLFEnd(i)+1]$, and a similar argument holds for $RLFBeg(i)$. Thus, we have the following observation for extended maximal palindromes after deletion, which is an analogue of Observation 5.

▶ **Observation 11** (Extended maximal palindromes after single character deletion). *For any $s \in MaxPalEnd_T(RLFEnd(i)-1)$, the corresponding maximal palindrome $T[RLFEnd(i)-s..RLFEnd(i)-1]$ centered at $\frac{2RLFEnd(i)-s-1}{2}$ gets extended in T' iff $T[RLFEnd(i)-s-1] = T[RLFEnd(i)+1]$. Similarly, for any $p \in MaxPalBeg_T(RLFBeg(i)+1)$, the corresponding maximal palindrome $T[RLFBeg(i)+1..RLFBeg(i)+p]$ centered at $\frac{2RLFBeg(i)+p+1}{2}$ gets extended in T' iff $T[RLFBeg(i)+p+1] = T[RLFBeg(i)-1]$.*

See Figure 3 for a concrete example for Observation 11.

Since the new characters that come from the left and the right of each deleted position are always unique, for each $RLFEnd(i)$ and $RLFBeg(i)$, the longest maximal palindrome that gets extended after deletion is also unique. Overall, we can precompute their lengths for all positions $1 \le i \le n$ in $O(n)$ total time by using $O(n)$ outward LCE queries in the original string T.

Next, we consider those maximal palindromes which get shortened after single character deletion. We have the following observation which is analogue to Observation 7.

▶ **Observation 12** (Shortened maximal palindromes after deletion). *A maximal palindrome $T[b..e]$ of T gets shortened in T' iff $b \le RLFBeg(i)$ and $RLFEnd(i) \le e$.*

See Figure 4 for a concrete example for Observation 12.

By Observation 12, we can precompute the length of the longest maximal palindrome after deleting the characters at the beginning and ending positions of each RL factors in $O(n)$ total time, using an analogous way to Lemma 8.

Summing up all the above discussions, we obtain the following lemma:

▶ **Lemma 13.** *It is possible to preprocess a string T of length n in $O(n)$ time and space so that later we can compute in $O(1)$ time the length of the LSPals of the edited string T' after deletion of a character.*

3.4 Algorithm for insertion

Consider to insert a new character c' between the ith and $(i+1)$th positions in T, and let $T' = T[1..i]c'T[i+1..n]$. If $c' \neq T[i]$ and $c' \neq T[i+1]$, we can find the length of the LSPals in T' in a similar way to substitution. Otherwise (if $c' = T[i]$ or $c' = T[i+1]$), then we can find the length of the LSPals in T' in a similar way to deletion since c' is merged to an adjacent RL factor. Thus, we have the following.

▶ **Lemma 14.** *Let T be a string of length n over an integer alphabet of size polynomial in n. It is possible to preprocess in $O(n)$ time and space string T so that later we can compute in $O(\log(\min\{\sigma, \log n\}))$ time the length of the LSPals of the edited string T' after insertion of a character.*

3.5 Hashing

By using hashing instead of binary searches on arrays, the following corollary is immediately obtained from Theorem 1.

▶ **Corollary 15.** *There is an algorithm for the 1-ELSPal problem which uses $O(n)$* expected *time and $O(n)$ space for preprocessing, and answers each query in $O(1)$ time for single character substitution, insertion, and deletion.*

4 Algorithm for ℓ-ELSPal

In this section, we consider the ℓ-ELSPal problem where an existing block of length ℓ' in the string T is replaced with a new block of length ℓ. This generalizes substitution when $\ell' > 0$ and $\ell > 0$, insertion when $\ell' = 0$ and $\ell > 0$, and deletion when $\ell' > 0$ and $\ell = 0$.

This section presents the following result:

▶ **Theorem 16.** *There is an $O(n)$-time and space preprocessing for the ℓ-ELSPal problem such that each query can be answered in $O(\ell + \log n)$ time, where ℓ denotes the length of the block after edit.*

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

Consider to substitute a substring 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 edit. For ease of explanation, we assume that there exist two positions $j_1 < j_2$ in X such that j_1 is the smallest position with $T[i_b + j_1 - 1] \neq X[j_1]$ and j_2 is the greatest position with $T[i_e - \ell + j_2] \neq X[j_2]$. The other cases (e.g., X or $T[i_b..i_e]$ is the empty string, j_1 and j_2 do not exist, or $j_1 = j_2$) can be treated similarly. Given the above assumption, we can restrict ourselves to the case where the first and last characters

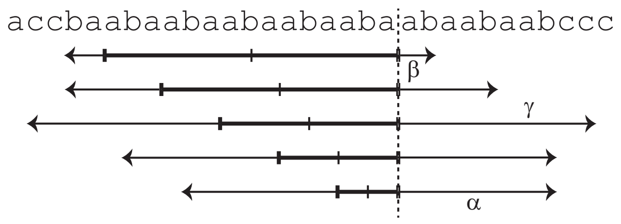

Figure 5 Example for Lemma 19, where Y = accbaabaabaabaabaaba and Z = abaabaabccc. Here we have $\alpha = 8$, $\beta = 2$, and $\gamma = 10$.

of $T[i_b..i_e]$ differ from those of X: Otherwise, then let $p_b = lcp(T[i_b..i_e], X) = j_1 - 1$ and $p_e = lcp((T[i_b..i_e])^R, X^R) = \ell - j_2$. We can compute p_b and p_e in $O(\ell - \hat{\ell} + 1)$ time by naïve character comparisons, where $\hat{\ell} = \ell - p_b - p_e = j_2 - j_1 + 1$. Then, the above ℓ-ELSPal query reduces to an $\hat{\ell}$-ELSPal query with edited string $T[1..i_b + p_b]X[p_b + 1..\ell - p_e]T[i_e - p_e..n]$.

We have the following observation for those of maximal palindromes in T whose lengths do not change, which is a generalization of Observation 4.

▶ **Observation 17** (Unchanged maximal palindromes after block edit). *For any position $1 \leq j < i_b$, $MaxPalEnd_{T''}(j) = MaxPalEnd_T(j)$. For any position $i_e < j \leq n$, $MaxPalBeg_{T''}(j) = MaxPalBeg_T(j)$.*

Hence, we can use the same $O(n)$-time preprocessing and $O(1)$ queries as the 1-ELSPal problem: When we consider substitution for an existing block $T[i_b..i_e]$, we take the length of the longest maximal palindrome ending before i_b and that of the longest maximal palindrome beginning after i_e as candidates for a solution to the ℓ-ELSPal query.

Next, we consider the maximal palindromes of T that get extended after block edit.

▶ **Observation 18** (Extended maximal palindromes after block edit). *For any $s \in MaxPalEnd_T(i_b-1)$, 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 $p \in MaxPalBeg_T(i_e+1)$, the corresponding maximal palindrome $T[i_e + 1..i_e + p]$ centered at $\frac{2i_e+p+1}{2}$ gets extended in T'' iff $\mathsf{OutLCE}_{T''}(i_e, i_e + p + 1) \geq 1$.*

It follows from Observation 18 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''. However, there can be $\Omega(n)$ maximal palindromes beginning or ending at each position of a string of length n. Yet, we can compute the length of the longest maximal palindromes that get extended after edit using periodic structures of maximal palindromes.

Let $\langle s, d, t\rangle$ be an arithmetic progression representing a group of maximal palindromes ending at position $i_b - 1$. For each $1 \leq j \leq t$, let s_j denote the jth shortest element for $\langle s, d, t\rangle$, namely, $s_j = s + (j-1)d$. 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 19.** *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 5 for a concrete example of Lemma 19.

Lemma 19 can be proven immediately from Lemma 12 of [14]. However, for the sake of completeness we here provide a proof. We use the following known result:

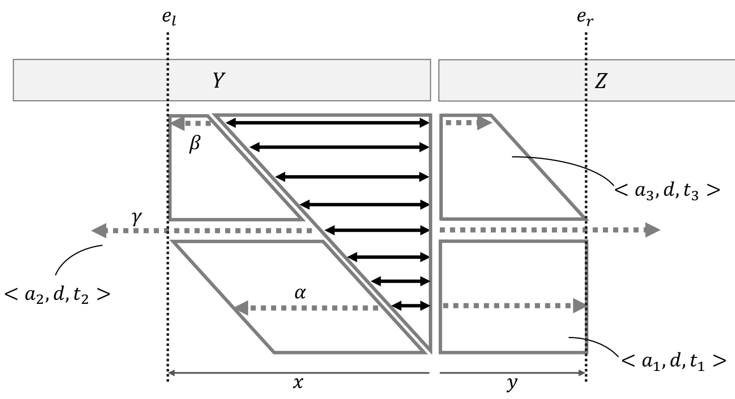

Figure 6 Illustration for the proof of Lemma 19, where $a_1 = s$, $a_2 = s{+}t_1d$, and $a_3 = s{+}(t_1{+}t_2)d$.

▶ **Lemma 20** ([14]). *For any string Y and $\{s_j \mid s_j \in \langle s, d, t\rangle\} \subseteq \mathit{SufPals}(Y)$, there exist palindromes u, v and a non-negative integer k, such that $(uv)^{t+k-1}u$ is a suffix of Y, $|uv| = d$ and $|(uv)^k u| = s$.*

Now we are ready to prove Lemma 19 (see also Figure 6).

Proof. Let us consider $\mathit{Ext}(s_j)$, such that $s_j \in \langle s, d, t\rangle$. By Lemma 20, $Y[|Y| - s_1 - (t - 1)d + 1] = (uv)^{t+k-1}u$, where $|uv| = d$ and $|(uv)^k u| = s$.

Let x be the largest integer such that $(Y[|Y| - x + 1..|Y|])^R$ has a period $|uv|$. Namely, $(Y[|Y| - x + 1..|Y|])^R$ is the longest prefix of Y^R that has a period $|uv|$. Then x is given as $x = \mathit{lcp}(Y^R, (Y[1..|Y| - d])^R + d$. Let y be largest integer such that $(uv)^{y/d}$ is a prefix of Z. Then y is given as $y = \mathit{lcp}(Y^R, Z)$.

Let $e_l = |Y| - x + 1$ and $e_r = |Y| + y$. Then, clearly string $T''[e_l..e_r]$ has a period d. We divide $\langle s, d, t\rangle$ into three disjoint subsets as

$$\langle s, d, t\rangle = \langle s, d, t_1\rangle \cup \langle s + t_1d, d, t_2\rangle \cup \langle s + (t_1 + t_2)d, d, t_3\rangle,$$

such that
$|Y| - e_l - s_j + 1 > e_r - |Y|$ for any $s_j \in \langle s, d, t_1\rangle$,
$|Y| - e_l - s_j + 1 = e_r - |Y|$ for any $s_j \in \langle s + t_1d, d, t_2\rangle$,
$|Y| - e_l - s_j + 1 < e_r - |Y|$ for any $s_j \in \langle s + (t_1 + t_2)d, d, t_3\rangle$,
and $t_1 + t_2 + t_3 = t$.

Then, for any s_j in the first sub-group $\langle s, d, t_1\rangle$, $\mathit{Ext}(s_j) = s_j + 2(e_r - |Y|) = s_j + 2y$. Also, for any s_j in the third sub-group $\langle s + (t_1 + t_2)d, d, t_3\rangle$, $\mathit{Ext}(s_j) = s_j + 2(|Y| - e_l - s_j + 1) = s_j + 2(x - s_j)$. Now let us consider $s_j \in \langle a_2, d, t_2\rangle$, in which case $s_j = s_h$ (see the statement of Lemma 19). Note that $0 \le t_2 \le 1$, and here we consider the interesting case where $t_2 = 1$. Since the palindrome s_h can be extended beyond the periodicity w.r.t. uv, we have $\mathit{Ext}(s_h) = s_h + 2\gamma$, where $\gamma = \mathit{lcp}((Y[1..|Y| - s_h])^R, Z)$.

Additionally, we have that $y = \mathit{lcp}(Y^R, Z) = \mathit{lcp}((Y[1..|Y| - s_1])^R, Z) = \alpha$ where the second equality comes from the periodicity w.r.t. uv, and that $x - s_j = \mathit{lcp}((Y[1..|Y| - s_t])^R, Z) + (t - j)d = \beta + (t - j)d$. Therefore, for any $s_j \in \langle s, d, t\rangle$, $\mathit{Ext}(s_j)$ can be represented as follows:

$$\mathit{Ext}(s_j) = \begin{cases} s_j + 2\alpha & (\alpha < \beta + (t - j)d) \\ s_j + 2(\beta + (t - j)d) & (\alpha > \beta + (t - j)d) \\ s_j + 2\gamma & (\alpha = \beta + (t - j)d) \end{cases}$$

This completes the proof. ◀

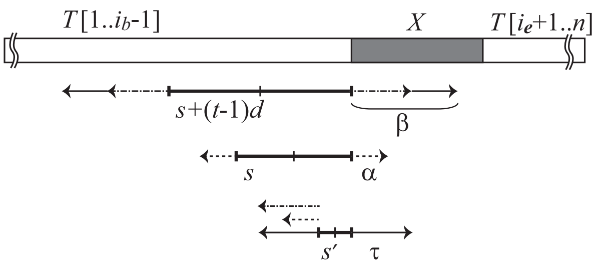

Figure 7 Illustration for Lemma 21, where solid arrows represent the matches obtained by naïve character comparisons, and broken arrows represent those obtained by LCE queries. Here we consider the case where $0 < \tau < \ell$. To compute α, we first perform a leftward LCE query. Here, the LCE value is less than τ and thus it is α. To compute β, we also perform a leftward LCE query. Here, the LCE value is at least τ, and thus we perform naïve character comparisons to determine the remainder of β. Other cases can be treated similarly.

Due to Lemma 19, provided that α, β, and γ (if s_h exists) are already computed, then it is a simple arithmetic to calculate the length of the longest extended maximal palindrome from $\langle s, d, t\rangle$ in $T'' = YZ$.

▶ **Lemma 21.** *Let T be a string of length n over an integer alphabet of size polynomially bounded in n. It is possible to preprocess T in $O(n)$ time and space so that later we can compute in $O(\ell + \log n)$ time the length of the longest maximal palindromes of T'' that are extended after replacing an existing block with a new block of length ℓ.*

Proof. Let $\langle s, d, t\rangle$ be any arithmetic progression representing a group of $MaxPalEnd_T(i_b - 1)$, and α, β, and γ be the lcp values for this group as defined in Lemma 19. Suppose that we have already processed all groups of shorter maximal palindromes. Let s' be one of the already processed maximal palindromes which has the longest extension of length τ (i.e., $s' + 2\tau$ is the length of the extended maximal palindrome for s'). See also Figure 7. There are three cases: (1) If $\tau = 0$, then we compute α by naïve character comparisons between $(T[1..i_b - s - 1])^R$ and X. (2) If $0 < \tau < \ell$, then we first compute $\delta = \mathsf{LeftLCE}_T(i_b - s - 1, i_b - s' - 1)$. (2-a) If $\delta < \tau$, then $\alpha = \delta$. (2-b) Otherwise ($\delta \geq \tau$), then we know that α is at least as large as τ. We then compute the remainder of α by naïve character comparisons. If the character comparison reaches the end of X, then the remainder of α can be computed by $\mathsf{OutLCE}_T(i_b - s - \ell - 1, i_e + 1)$. Then we update τ with α. (3) If $\tau \geq \ell$, then we can compute α by $\mathsf{LeftLCE}_T(i_b - s - 1, i_b - s' - 1)$, and if this value is at least ℓ, then by $\mathsf{OutLCE}_T(i_b - s - \ell - 1, i_e + 1)$. β and γ (if it exists) can also be computed similarly.

After processing all arithmetic progressions representing the groups for $MaxPalEnd_T(i_b - 1)$, 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 $O(\log n)$ since for each arithmetic progression there are at most three mismatching character comparisons (those for α, β, and γ). The total number of LCE queries in the original text T is $O(\log n)$, each of which can be answered in $O(1)$ time. Thus, together with Lemma 19, it takes $O(\ell + \log n)$ time to compute the length of the longest maximal palindromes of T'' that are extended after block edit. ◀

▶ Remark. An alternative method to Lemma 21 would be to first build the suffix tree of $T\#T^R\$$ enhanced with a dynamic lowest common ancestor data structure [4] using $O(n)$ time and space [5], and then to update the suffix tree with string $T\#T^R\$X\#'X^R\$'$ using Ukkonen's online algorithm [18], where $\#'$ and $\$'$ are special characters not appearing in T

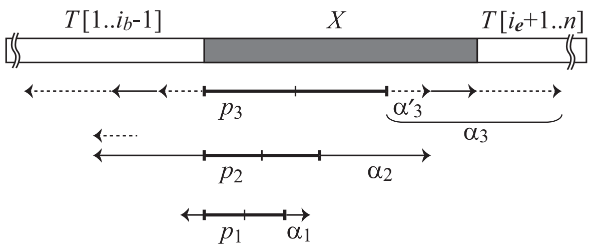

Figure 8 Illustration for Lemma 24, where solid arrows represent the matches obtained by naïve character comparisons, and broken arrows represent those obtained by LCE queries. Here are three prefix palindromes of X of length p_1, p_2, and p_3. We compute α_1 naïvely. Here, since $p_1+\alpha_1 < p_2$, we compute p_2 naïvely. Since $p_2+\alpha_2 > p_3$, we compute $\mathsf{LeftLCE}_T(i_b-1, i_b-\alpha_2+\alpha'_3-1)$. Here, since its value reached α'_3, we perform naïve character comparison for $X[p_3+\alpha'_3+1..\ell]$ and $(T[1..i_b-\alpha'_3-1])^R$. Here, since there was no mismatch, we perform $\mathsf{OutLCE}_T(i_b-\ell+p_3-1, i_e+1)$ and finally obtain α_3. Other cases can be treated similarly.

nor X. This way, one can answer LCE queries between any position of the original string T and any position of the new block X in $O(1)$ time. Since we need $O(\log n)$ LCE queries, it takes $O(\log n)$ total time for all LCE queries. However, Ukkonen's algorithm requires $O(\ell \log \sigma)$ time to insert $X\#'X^R\$'$ into the existing suffix tree, where $\ell = |X|$. Thus, this method requires us $O(\ell \log \sigma + \log n)$ time and thus is slower by a factor of $\log \sigma$ than the method of Lemma 21.

Finally, we consider the maximal palindromes that get shortened after block edit.

▶ **Observation 22** (Shortened maximal palindromes after block edit)**.** *A maximal palindrome $T[b..e]$ of T gets shortened in T'' iff $b \le i_b \le e$ and $i_b \ne \frac{b+e}{2}$, or $b \le i_e \le e$ and $i_e \ne \frac{b+e}{2}$.*

The difference between Observation 7 and this one is only in that here we need to consider two positions i_b and i_e. Hence, we obtain the next lemma using a similar method to Lemma 8:

▶ **Lemma 23.** *We can preprocess a string T of length n in $O(n)$ time and space so that later we can compute in $O(1)$ time the length of the longest maximal palindromes of T'' that are shortened after block edit.*

Finally, we consider those maximal palindromes whose centers exist in the new block X of length ℓ. By symmetric arguments to Observation 18, we only need to consider the prefix palindromes and suffix palindromes of X. Using a similar technique to Lemma 21, we obtain:

▶ **Lemma 24.** *We can compute the length of the longest maximal palindromes whose centers are inside X in $O(\ell)$ time and space.*

Proof. First, we compute all maximal palindromes in X in $O(\ell)$ time. Let $p_1, \ldots, p_u$ be a sequence of the lengths of the prefix palindromes of X sorted in increasing order. For each $1 \le j \le u$, let $\alpha_j = lcp(X[p_j+1..\ell], (T[1..i_b-1])^R)$, namely, $p_j + 2\alpha_j$ is the length of the extended maximal palindrome for each p_j. Suppose we have computed α_{j-1}, and we are to compute α_j. See also Figure 8. If $p_{j-1} + \alpha_{j-1} \le p_j$, then we compute p_j by naïve character comparisons. Otherwise, then let $\alpha'_j = p_{j-1} + \alpha_{j-1} - p_j$. Then, we can compute $lcp(X[p_j+1..p_j+\alpha'_j], (T[1..i_b-1])^R)$ by a leftward LCE query in the original string T. If this value is less than α'_j, then it equals to α_j. Otherwise, then we compute $lcp(X[p_j+\alpha'_j+1..\ell], (T[1..i_b-1])^R)$ by naïve character comparisons. The total number of matching character comparisons is at most ℓ since each position in X can be involved in at most one matching character comparison. The total number of mismatching character comparisons is also ℓ, since there are at most ℓ prefix palindromes of X and for each of them there is at most one mismatching character comparison. Hence, it takes $O(\ell)$ time to compute the length of the longest maximal palindromes whose centers are inside X. ◀

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

3 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000*, pages 88–94, 2000.

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

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

6 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.

7 Leszek Gąsieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding. In *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT1996)*, volume 1097 of *LNCS*, pages 392–403. Springer, 1996.

8 Richard Groult, Élise Prieur, and Gwénaël Richomme. Counting distinct palindromes in a word in linear time. *Inf. Process. Lett.*, 110(20):908–912, 2010.

9 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

10 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

11 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theor. Comput. Sci.*, 410(51):5365–5373, 2009.

12 Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Finding distinct subpalindromes online. In *Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013*, pages 63–69, 2013.

13 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.

14 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.

15 Shintaro Narisada, Diptarama, Kazuyuki Narisawa, Shunsuke Inenaga, and Ayumi Shinohara. Computing longest single-arm-gapped palindromes in a string. In *SOFSEM 2017*, pages 375–386, 2017.

16 Alexandre H. L. Porto and Valmir C. Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 35:2581–2591, 2002.

17 Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

18 E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

19 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.