Shortest Unique Palindromic Substring Queries in Optimal Time

Yuto Nakashima[1,2](✉), Hiroe Inoue[1], Takuya Mieno[1], Shunsuke Inenaga[1], Hideo Bannai[1], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University, Fukuoka, Japan
{yuto.nakashima,hiroe.inoue,inenaga,bannai,takeda}@inf.kyushu-u.ac.jp
[2] Japan Society for the Promotion of Science (JSPS), Tokyo, Japan

Abstract. A palindrome is a string that reads the same forward and backward. A palindromic substring P of a string S is called a shortest unique palindromic substring (*SUPS*) for an interval $[s, t]$ in S, if P occurs exactly once in S, this occurrence of P contains interval $[s, t]$, and every palindromic substring of S which contains interval $[s, t]$ and is shorter than P occurs at least twice in S. The *SUPS* problem is, given a string S, to preprocess S so that for any subsequent query interval $[s, t]$ all the *SUPS*s for interval $[s, t]$ can be answered quickly. We present an optimal solution to this problem. Namely, we show how to preprocess a given string S of length n in $O(n)$ time and space so that all *SUPS*s for any subsequent query interval can be answered in $O(\alpha + 1)$ time, where α is the number of outputs.

1 Introduction

A substring $S[i..j]$ of a string S is called a *shortest unique substring* (*SUS*) for a position p if $S[i..j]$ is the shortest substring s.t. $S[i..j]$ is unique in S (i.e., $S[i..j]$ occurs exactly once in S), and $[i..j]$ contains p (i.e., $i \leq p \leq j$). Recently, Pei et al. [13] proposed the *point SUS problem*, preprocessing a given string S of length n so that we can return a *SUS* for any given query position efficiently. This problem was considered for some applications in bioinformatics, e.g., polymerase chain reaction (PCR) primer design in molecular biology. Pei et al. [13] proposed an algorithm which returns a *SUS* for any given position in constant time after $O(n^2)$-time preprocessing. After that, Tsuruta et al. [15] and Ileri et al. [9] independently showed optimal $O(n)$-time preprocessing and constant query time algorithms. They also showed optimal $O(n)$-time preprocessing and $O(k)$ query time algorithms which return all *SUS*s for any given position where k is the number of outputs. Moreover, Hon et al. [6] proposed an in-place algorithm which returns a *SUS*. A more general problem called *interval SUS problem*, where a query is an interval, was considered by Hu et al. [7]. They proposed an optimal $O(n)$-time preprocessing and $O(k)$ query time algorithm which returns all *SUS*s containing a given query interval. Most recently, Mieno et al. [12] proposed an efficient algorithm for interval *SUS* problem when the input string is represented by *run-length encoding*.

© Springer International Publishing AG, part of Springer Nature 2018
L. Brankovic et al. (Eds.): IWOCA 2017, LNCS 10765, pp. 397–408, 2018.
https://doi.org/10.1007/978-3-319-78825-8_32

In this paper, we consider a new variant of interval *SUS* problems concerning palindromes. A substring $S[i..j]$ is called a palindromic substring of S if $S[i..j]$ and the reversed string of $S[i..j]$ is the same string. The study of combinatorial properties and structures on palindromes is still an important and well studied topic in stringology [1,3–5,8,14]. Droubay et al. [3] showed a string of length n can contain at most $n + 1$ distinct palindromes. Moreover, Groult et al. [5] proposed a linear time algorithm for computing all distinct palindromes in a string.

Our new problem can be described as follows. A substring $S[i..j]$ of a string S is called a *shortest unique palindromic substring* (*SUPS*) for an interval $[s, t]$ if $S[i..j]$ is the shortest substring s.t. $S[i..j]$ is unique in S, $[i..j]$ contains $[s, t]$, and $S[i..j]$ is a palindromic substring. The *interval SUPS problem* is to preprocess a given string S of length n so that we can return all *SUPS*s for any query interval efficiently. For this problem, we propose an optimal $O(n)$-time preprocessing and $O(\alpha+1)$-time query algorithm, where α is the number of outputs. Potential applications of our algorithm are in bioinformatics; It is known that the presence of particular (e.g., unique) palindromic sequences can affect immunostimulatory activities of oligonucleotides [10,16]. The size and the number of palindromes also influence the activity. Since any unique palindromic sequence can be obtained easily from a shorter unique palindromic sequences, we can focus on the shortest unique palindromic substrings.

The contents of our paper are as follows. In Sect. 2, we state some definitions and properties on strings. In Sect. 3, we explain properties on *SUPS* and our query algorithm. In Sect. 4, we show the main part of the preprocessing phase of our algorithm. Finally, we conclude.

2 Preliminaries

2.1 Strings

Let Σ be an integer *alphabet*. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. Let Σ^+ be the set of non-empty strings, i.e., $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $S = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of S, respectively. A prefix x and a suffix z of S are respectively called a *proper prefix* and *proper suffix* of S, if $x \neq S$ and $z \neq S$. The i-th character of a string S is denoted by $S[i]$, where $1 \leq i \leq |S|$. For a string S and two integers $1 \leq i \leq j \leq |S|$, let $S[i..j]$ denote the substring of S that begins at position i and ends at position j. For convenience, let $S[i..j] = \varepsilon$ when $i > j$.

2.2 Palindromes

Let S^R denote the reversed string of S, that is, $S^R = S[|S|] \cdots S[1]$. A string S is called a palindrome if $S = S^R$. Let $P \subset \Sigma^*$ be the set of palindromes. A substring $S[i..j]$ of S is said to be a palindromic substring of S, if $S[i..j] \in P$.

The center of a palindromic substring $S[i..j]$ of S is $\frac{i+j}{2}$. Thus a string S of length $n \geq 1$ has $2n-1$ centers $(1, 1.5, \ldots, n-0.5, n)$. The following lemma can be easily obtained by the definition of palindromes.

Lemma 1. *Let S be a palindrome. For any integers i, j s.t. $1 \leq i \leq j \leq |S|$, $S[|S|-j+1..|S|-i+1] = S[i..j]^R$ holds.*

2.3 *MUPS*s, *SUPS*s and Our Problem

For any non-empty strings S and w, let $occ_S(w)$ denote the set of occurrences of w in S, namely, $occ_S(w) = \{i \mid 1 \leq i \leq |S| - |w| + 1, w = S[i..i+|w|-1]\}$. A substring w of a string S is called a *unique substring* (resp. a *repeat*) of S if $|occ_S(w)| = 1$ (resp. $|occ_S(w)| \geq 2$). In the sequel, we will identify each unique substring w of S with its corresponding (unique) interval $[i, j]$ in S such that $w = S[i..j]$. A substring $S[i..j]$ is said to be *unique palindromic substring* if $S[i..j]$ is a unique substring in S and a palindromic substring. We will say that an interval $[i_1, j_1]$ contains an interval $[i_2, j_2]$ if $i_1 \leq i_2 \leq j_2 \leq j_1$ holds. The following notation is useful in our algorithm.

Definition 1 (Minimal Unique Palindromic Substring (*MUPS*)). *A string $S[i..j]$ is a MUPS in S if $S[i..j]$ satisfies all the following conditions;*

- *$S[i..j]$ is a unique palindromic substring in S,*
- *$S[i+1..j-1]$ is a repeat in S or $1 \leq |S[i..j]| \leq 2$.*

Let $\mathcal{M}_S$ denote the set of intervals of all *MUPS*s in S and let $mups_i = [b_i, e_i]$ denote the i-th *MUPS* in $\mathcal{M}_S$ where $1 \leq i \leq m$ and m is the number of *MUPS*s in S. We assume that *MUPS*s in $\mathcal{M}_S$ are sorted in increasing order of beginning positions. For convenience, we define $mups_0 = [-1, -1]$, $mups_{m+1} = [n+1, n+1]$.

Example 1 (MUPS). For $S =$ `acbaaabcbcbcbaab`, $\mathcal{M}_S = \{[4,6], [8,12], [13,16]\}$ (see also Fig. 1).

Definition 2 (Shortest Unique Palindromic Substring (*SUPS*)). *A string $S[i..j]$ is a SUPS for an interval $[s, t]$ in S if $S[i..j]$ satisfies all the following conditions;*

- *$S[i..j]$ is a unique palindromic substring in S,*
- *$[i, j]$ contains $[s, t]$,*
- *no unique palindromic substring $S[i'..j']$ containing $[s, t]$ with $j' - i' < j - i$ exists.*

Example 2 (SUPS). Let $S =$ `acbaaabcbcbcbaab`. *SUPS* for interval $[6, 7]$ is the $S[3..7] =$ `baaab`. *SUPS* for interval $[7, 8]$ are $S[2..8] =$ `cbaaabc` and $S[7..13] =$ `bcbcbcb`. *SUPS* for interval $[4, 13]$ does not exist. (see also Fig. 1).

In this paper, we tackle the following problem.

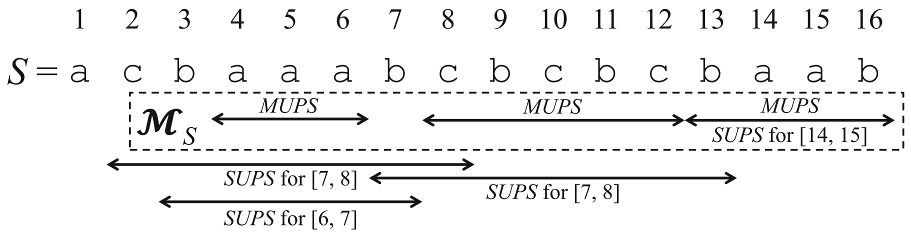

Fig. 1. This figure shows all *MUPS*s for S = `acbaaabcbcbcbaab` and some *SUPS* described in Example 2.

Problem 1 (SUPS problem).

- **Preprocess**: String S of length n.
- **Query**: An interval $[s, t](1 \leq s \leq t \leq n)$.
- **Return**: All the *SUPS*s for interval $[s, t]$.

2.4 Computation Model

Our model of computation is the word RAM: We shall assume that the computer word size is at least $\lceil \log_2 n \rceil$, and hence, standard operations on values representing lengths and positions of strings can be manipulated in constant time. Space complexities will be determined by the number of computer words (not bits).

3 Solution to the *SUPS* Problem

In this section, we show how to compute all *SUPS*s for any query interval $[s, t]$.

3.1 Properties on *SUPS* and *MUPS*

In our algorithm, we compute *SUPS*s by using *MUPS*s. Firstly, we show the following lemma. Lemma 2 states that *MUPS*s cannot nest in each other.

Lemma 2. *For any pair of distinct MUPSs, one cannot contain the other.*

Proof. Consider two *MUPS*s u, v such that u contains v. If u and v have the same center, then u is not a *MUPS*. On the other hand, if u and v have a different center, we have from Lemma 1 and that v is a palindromic substring, v occurs in u at least twice. This contradicts that v is unique.

From this lemma, we can see that no pair of distinct *MUPS*s begin nor end at the same position. This fact implies that the number of *MUPS*s is at most n for any string of length n. The following lemma states a characterization of *SUPS*s by *MUPS*s.

Lemma 3. *For any SUPS $S[i..j]$ for some interval, there exists exactly one MUPS that is contained in $[i, j]$. Furthermore, the MUPS has the same center as $S[i..j]$.*

Proof. Let $S[i..j]$ be a *SUPS* for some interval. $S[i..j]$ contains a *MUPS* $S[x_1..y_1]$ of the same center, i.e., $\frac{i+j}{2} = \frac{x_1+y_1}{2}$, s.t. $j - i \geq y_1 - x_1$. Suppose that there exists another *MUPS* $S[x_2..y_2]$ contained in $[i, j]$. From Lemma 2, $S[x_1..y_1]$ and $S[x_2..y_2]$ do not have the same center. On the other hand, if $S[x_1..y_1]$ and $S[x_2..y_2]$ have different centers, then $S[x_2..y_2]$ occurs at least two times in $S[i..j]$ by Lemma 1, since $S[x_2..y_2] = S[x_2..y_2]^R$. This contradicts that $S[x_2..y_2]$ is a *MUPS*.

From the above lemma, any *SUPS* contains exactly one *MUPS* which has the same center (see also Fig. 1). Below, we will describe the relationship between a query interval $[s, t]$ and the *MUPS* contained in a *SUPS* for $[s, t]$. Before explaining this, we define the following notations.

- $\mathcal{M}([s, t])$: the set of *MUPS*s containing $[s, t]$.
- $predMUPS[t] = i$ s.t. $i = \max\{k \mid e_k \leq t\}$.
- $succMUPS[s] = i$ s.t. $i = \min\{k \mid s \leq b_k\}$.

In other words, $mups_{predMUPS[t]}$ is the rightmost *MUPS* which ends before position $t+1$, and $mups_{succMUPS[s]}$ is the leftmost *MUPS* which begins after position $s - 1$.

Lemma 4. *Let $S[i..j]$ be a SUPS for an interval $[s, t]$. Then, the unique MUPS $S[x..y]$ contained in $[i, j]$ is in $\{predMUPS[t]\} \cup \mathcal{M}([s, t]) \cup \{succMUPS[s]\}$.*

Proof. Assume to the contrary that there exists a *SUPS* $S[i..j]$ that contains a *MUPS* $S[x..y] \notin \{predMUPS[t]\} \cup \mathcal{M}([s, t]) \cup \{succMUPS[s]\}$. Since $S[x..y] \notin \mathcal{M}([s, t])$, $[x, y]$ does not contain $[s, t]$. Thus, there can be the following two cases:

- If $y < t$, there must exist *MUPS* $[x', y']$ s.t. $y < y' \leq t$, since $S[x, y] \neq predMUPS[t]$. By Lemma 2, $x < x'$. Thus $i \leq x < x' \leq y' \leq t \leq j$ holds. However, this contradicts Lemma 3.
- If $s < x$, there must exist *MUPS* $[x', y']$ s.t. $s \leq x' < x$, since $S[x, y] \neq succMUPS[s]$. By Lemma 2, $y' \leq y$. Thus $i \leq s \leq x' \leq y' \leq y \leq j$ holds, However, this contradicts Lemma 3.

Therefore the lemma holds.

Next, we want to explain how *SUPS*s are related to *MUPS*s. It is easy to see that there may not be a *SUPS* for some query interval. We first show a case where there are no *SUPS*s for a given query. The following corollary is obtained from Lemma 3.

Corollary 1. *Let $S[x_1..y_1]$ and $S[x_2..y_2]$ be MUPSs contained in a query interval $[s, t]$. There is no SUPS for an interval $[s, t]$.*

From this corollary, a *SUPS* for an interval $[s,t]$ can exist if the number of *MUPS*s contained in $[s,t]$ is less than or equal to 1. The following two lemmas show what the *SUPS* for $[s,t]$ is, when $[s,t]$ contains only one *MUPS*, and when $[s,t]$ does not contain any *MUPS*s.

Lemma 5. *Let $S[x..y]$ be the only MUPS contained in the query interval $[s,t]$. If $S[x-z, y+z]$ is a palindromic substring where $z = \max\{x-s, t-y\}$, then $S[x-z, y+z]$ is the SUPS for $[s,t]$. Otherwise, there is no SUPS for $[s,t]$.*

Proof. Assume that there exists a *SUPS* u for $[s,t]$ which has the same center with a *MUPS* other than $S[x..y]$. By the definition of *SUPS*, u should contain $[s,t]$. Since $[s,t]$ contains $[x,y]$, u contains two *MUPS*s, a contradiction. Thus, there can be no *SUPS* s.t. the center is not $\frac{x+y}{2}$. It is clear that $S[x-z, y+z]$ is a unique palindromic substring if $S[x-z, y+z]$ is a palindromic substring where $z = \max\{x-s, t-y\}$. Therefore the lemma holds.

Lemma 6. *Let $[s,t]$ be the query interval. Then SUPSs for $[s,t]$ are the shortest of the following candidates.*

1. $S[x..y]$ s.t. $[x,y] \in \mathcal{M}([s,t])$,
2. $S[x-t+y..t]$ s.t. $[x,y] = predMUPS([s,t])$, if it is a palindromic substring,
3. $S[s..y+x-s]$ s.t. $[x,y] = succMUPS([s,t])$, if it is a palindromic substring.

Proof. It is clear that $S[x..y]$ is a unique palindromic substring containing $[s,t]$ if $[x,y] \in \mathcal{M}([s,t])$ exists. It is also clear that if $[x,y] = predMUPS([s,t])$ or $[x,y] = succMUPS([s,t])$, then $S[x-t+y..t]$ or $[s..y+x-s]$, respectively, are unique palindromic substrings, if they are palindromic substrings. By Lemma 4, we do not need to consider palindromic substrings which have the same center as *MUPS*s other than the candidates considered above. Thus the shortest of the candidates is *SUPS* for $[s,t]$ (see also Fig. 2).

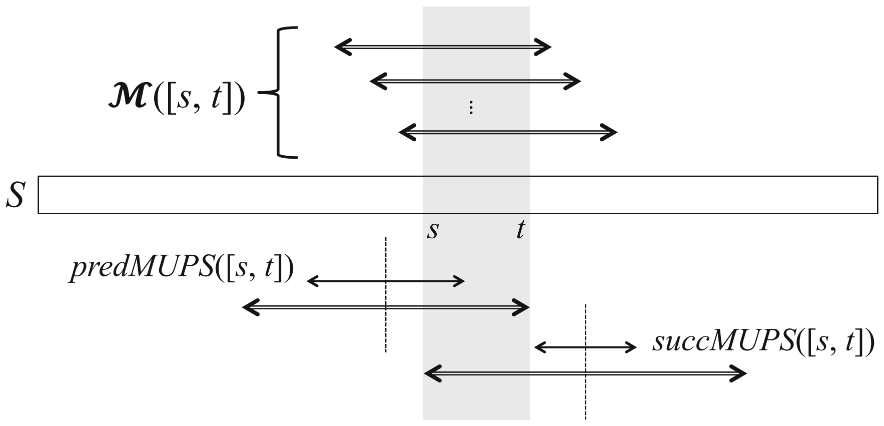

Fig. 2. Double arrows represent the candidates of *SUPS* for $[s,t]$. The shortest of the candidates is *SUPS* for $[s,t]$.

From the above arguments, the number of *MUPS*s is useful to compute *SUPS*s for a query interval. The following lemma shows how to compute the number of *MUPS*s contained in a given interval.

Lemma 7. *For any interval* $[s, t]$*,*

- *if* $succMUPS[s] > predMUPS[t]$*,* $[s, t]$ *contains no MUPS,*
- *if* $succMUPS[s] = predMUPS[t]$*,* $[s, t]$ *contains only one MUPS,* $mups_{succMUPS[s]} = mups_{predMUPS[t]}$*, and*
- *if* $succMUPS[s] < predMUPS[t]$*,* $[s, t]$ *contains at least two MUPSs.*

Proof.
- Let $j = succMUPS[s] > predMUPS[t] = i$. Then $b_i < s \leq b_j$ and $e_i \leq t < e_j$ hold, and thus neither of $mups_i$ and $mups_j$ are contained in $[s, t]$. If we assume that $[s, t]$ contains a *MUPS* $mups_k$ for some k, it should be that $i < k < j$, $b_i < s \leq b_k < b_j$. However, this contradicts that $j = succMUPS[s]$ (see also the top in Fig. 3).
- Let $succMUPS[s] = predMUPS[t] = i$. Since $succMUPS[s] = i$, b_{i-1} should be less than s, and b_i at least s. Since $predMUPS[t] = i$, e_{i+1} should be larger than t, and e_i at most t. Thus $[s, t]$ only contains $mups_i$ (see also the middle in Fig. 3).
- Let $i = succMUPS[s] < predMUPS[t] = j$. Then $s \leq b_i < b_j$ and $e_i < e_j \leq t$ hold, which implies $s \leq b_i \leq e_i < t$ and $s < b_j \leq e_j \leq t$. Thus, both $mups_i$ and $mups_j$ are contained in $[s, t]$ (see also the bottom in Fig. 3).

3.2 Tools

Here, we show some tools for our algorithm.

Lemma 8 (e.g., [14]). *For any interval* $[i, j]$ *in* S *of length* n*, we can check whether* $S[i..j]$ *is a palindromic substring or not in* $O(n)$ *preprocessing time and constant query time with* $O(n)$ *space.*

Manacher's algorithm [11] can compute all maximal palindromic substrings in linear time. If we have the array of radiuses of maximal palindromic substrings for all $2n-1$ centers, we can check whether a given substring $S[i..j]$ is a palindromic substring or not in constant time.

Range Minimum Queries (RmQ). Let A be an integer array of size n. A *range minimum query* $RmQ_A(i, j)$ returns the index of a minimum element in the subarray $A[i, j]$ for given a query interval $[i, j] (1 \leq i \leq j \leq n)$, i.e., it returns one of $\arg\min_{i \leq k \leq j}\{A[k]\}$. It is well-known (see e.g., [2]) that after an $O(n)$-time preprocessing over the input array A, $RmQ_A(i, j)$ can be answered in $O(1)$ time for any query interval $[i, j]$, using $O(n)$ space.

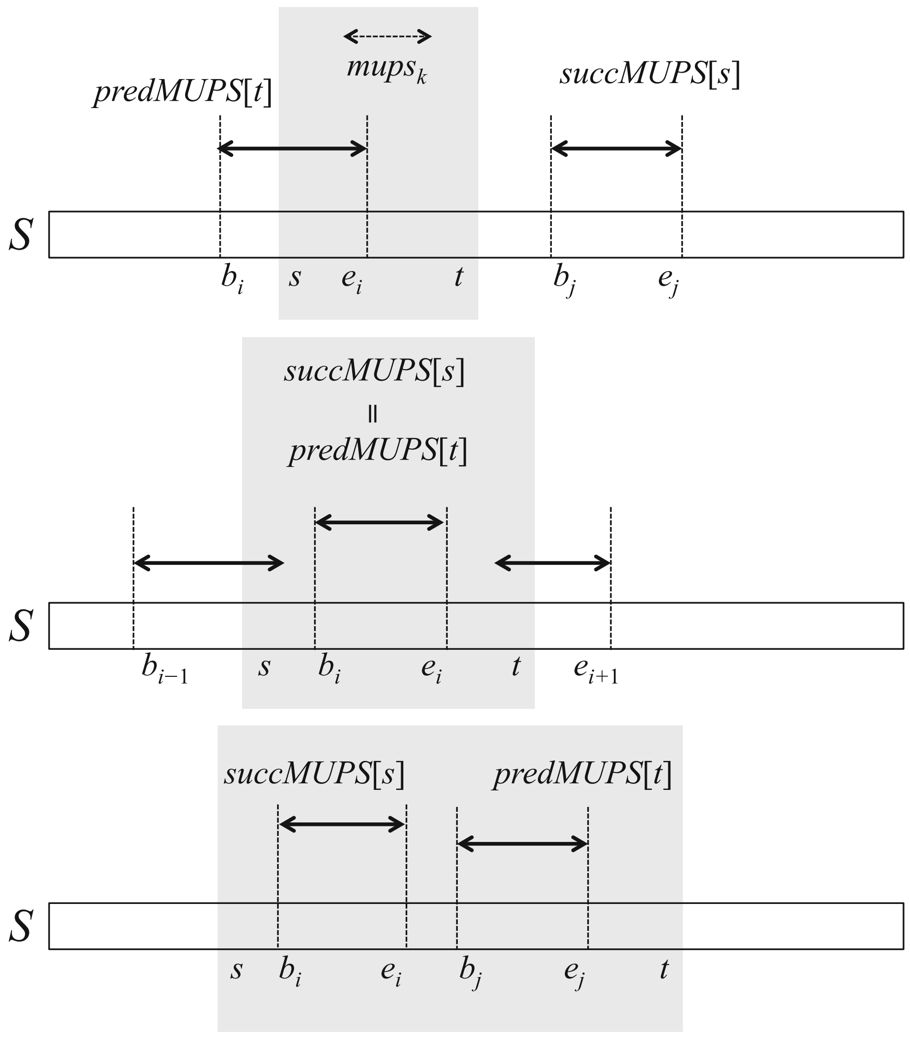

Fig. 3. Illustrations for proof of Lemma 7.

3.3 Algorithm

Due to the arguments in Sect. 3.1, if we can compute *predMUPS*, the shortest *MUPS*s in $\mathcal{M}([s,t])$ and *succMUPS* for a query interval $[s,t]$, then, we can compute *SUPS*s for $[s,t]$. Below, we will describe our solution to the *SUPS* problem.

Preprocessing Phase. First, we compute $\mathcal{M}_S$ for a given string S of length n in increasing order of beginning positions. We show, in the next section, that this can be done in $O(n)$ time and space. After computing $\mathcal{M}_S$, we compute the arrays *predMUPS* and *succMUPS*. It is easy to see that we can also compute these arrays in $O(n)$ time by using $\mathcal{M}_S$. In the query phase, we are required to compute the shortest *MUPS*s that contain the query interval $[s,t]$. To do so efficiently, we prepare the following array. Let *Mlen* be an array of length $m = |\mathcal{M}_S|$, and the i-th entry $Mlen[i]$ holds the length of $mups_i$, i.e., $Mlen[i] = |mups_i| = e_i - b_i + 1$. We also preprocess *Mlen* for *RmQ* queries. This can be done in $O(m)$ time and space as noted in Sect. 3.2. Thus, since $m = O(n)$, the total preprocessing is $O(n)$ time and space.

Query Phase. First, we compute how many *MUPS*s are contained in a query interval $[s,t]$ by using Lemma 7, which we denote by *num*. This can be done in $O(1)$ time given arrays *predMUPS* and *succMUPS*.

- If $num = 0$, let $mups_i = predMUPS([s,t])$ and $mups_j = succMUPS([s,t])$, i.e., $i = predMUPS[s]$ and $j = succMUPS[t]$. We check whether $S[b_i - t + e_i..t]$ and $S[s..e_j + b_j - s]$ are palindromic substrings or not. If so, then they are candidates of *SUPS*s for $[s,t]$ by Lemma 6. Let q be the length of the shortest candidates which can be found in the above. Second, we compute the shortest *MUPS* in $\mathcal{M}([s,t])$, if their lengths are at most q. In other words, we compute the smallest values in $Mlen[i+1..j-1]$, if they are at most q. We can compute all such *MUPS*s in linear time w.r.t. the number of such *MUPS*s by using *RmQ* queries on $Mlen[i+1, j-1]$; if $k = RmQ_{Mlen}(i+1, j-1)$ and $Mlen[k] \leq q$, then we consider the range $Mlen[i+1..k-1]$ and $Mlen[k+1, j-1]$ and recurse. Otherwise, we stop the recursion. Finally, we return the shortest candidates as *SUPS*.
- If $num = 1$, let $mups_i$ be the *MUPS* contained in $[s,t]$. First, we check whether $S[b_i - z, e_i + z]$ is a palindromic substring or not by using Lemma 8 where $z = \max\{b_i - s, t - e_i\}$. If so, then return $[b_i - z, e_i + z]$, otherwise *SUPS* for $[s,t]$ does not exist.
- If $num \geq 2$, then, from Corollary 1, *SUPS* for $[s,t]$ does not exist.

Therefore, we obtain the following.

Theorem 1. *After constructing an $O(n)$-space data structure of a given string of length n in $O(n)$ time, we can compute all SUPSs for a given query interval $[s,t]$ in $O(\alpha+1)$ time where α is the number of outputs.*

4 Computing *MUPS*s

In this section, we show how to compute $\mathcal{M}_S$ in $O(n)$ time and space. Let DP_S be the set of *distinct palindromic substrings* in S, and $strM_S = \{S[i,j] \mid [i,j] \in \mathcal{M}_S\}$. Our idea of computing $\mathcal{M}_S$ is based on the following lemma.

Lemma 9. $strM_S \subseteq DP_S$.

Proof. It is clear that any string in $strM_S$ is a palindromic substring of S.

An algorithm for computing all distinct palindromic substrings in string in linear time and space was proposed by Groult et al. [5]. We show a linear time and space algorithm which computes $\mathcal{M}_S$ by modifying Groult et al.'s algorithm.

4.1 Tools

We show some tools for computing $\mathcal{M}_S$ below.

- **Longest previous factor array (LPF).** We denote the longest previous factor array of S by LPF_S. The i-th entry $(1 \leq i \leq n)$ is the length of the longest prefix of $S[i..n]$ which occurs at a position less than i.
- **Inverse suffix array (ISA).** We denote the inverse suffix array of S by ISA_S. The i-th entry $(1 \leq i \leq n)$ is the lexicographic order of $S[i..n]$ in all suffixes of S.
- **Longest common prefix array (LCP).** We denote the longest common prefix array of S by LCP_S. The i-th entry $(2 \leq i \leq n)$ is the length of the longest common prefix of the lexicographically i-th suffix of S and the $(i-1)$-th suffix of S.

4.2 Computing Distinct Palindromes

Here, we show a summary of Groult et al.'s algorithm. The following lemma states the main idea.

Lemma 10 ([3]). *The number of distinct palindromic substrings in S is equal to the number of prefixes of S s.t. its longest palindromic suffix is unique in the prefix.*

Since counting suffixes that uniquely occur in a prefix implies that only the leftmost occurrences of substrings, and thus distinct substrings are counted, their algorithm finds all the distinct palindromic substrings by:

- computing the longest palindromic suffix of each prefix of S, and
- checking whether each longest palindromic suffix occurs uniquely in the prefix or not.

They first propose an algorithm which computes all the longest palindromic suffixes in linear time. They then check, in constant time, the uniqueness of the occurrence in the prefix by using the LPF array, thus computing DP_S in linear time and space.

4.3 Computing All *MUPS*s

Finally, we show how to modify Groult et al.'s algorithm. As mentioned, they compute the leftmost occurrence of each distinct palindromic substring. We call such a palindromic substring, the leftmost palindromic substring. It is clear that if a leftmost palindromic substring w is unique in S and is a minimal palindromic substring, then w is a $MUPS$. Thus, we add operations to check the uniqueness and minimality of each leftmost palindromic substring. We can do these operations by using ISA and LCP array.

Let $S[i..j]$ be a leftmost palindromic substring in S. First, we check whether $S[i..j]$ is unique or not in S. If $ISA[i] = k$, $S[i..n]$ is the lexicographically k-th suffix of S. $S[i..j]$ is unique in S iff $LCP[k] < j - i + 1$ and $LCP[k + 1] < j - i + 1$. Thus we can check whether $S[i..j]$ is unique or not in constant time. Finally, we check whether $S[i..j]$ is a minimal palindromic substring or not. By definition, $S[i..j]$ is minimal palindromic substring if $j - i + 1 \leq 2$, i.e., $S[i..j]$ has no shorter unique palindromic substring. If $j - i + 1 > 2$, then we check whether $S[i + 1..j - 1]$ is unique or not by using ISA and LCP in a similar way. Thus we can also check whether $S[i..j]$ is minimal or not in constant time. By the above arguments, we can compute all $MUPS$s in linear time and space.

5 Conclusions

We consider a new problem called the shortest unique palindromic substring problem. We proposed an optimal linear time preprocessing algorithm so that all $SUPS$s for any given query interval can be answered in linear time w.r.t. the number of outputs. The key idea was to use palindromic properties in order to obtain a characterization of $SUPS$, more precisely, that a palindromic substring cannot contain a unique palindromic substring with a different center.

References

1. Bannai, H., Gagie, T., Inenaga, S., Kärkkäinen, J., Kempa, D., Piątkowski, M., Puglisi, S.J., Sugimoto, S.: Diverse palindromic factorization is NP-complete. In: Potapov, I. (ed.) DLT 2015. LNCS, vol. 9168, pp. 85–96. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21500-6_6
2. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000). https://doi.org/10.1007/10719839_9
3. Droubay, X., Justin, J., Pirillo, G.: Episturmian words and some constructions of de Luca and Rauzy. Theor. Comput. Sci. **255**(1–2), 539–553 (2001)
4. Fici, G., Gagie, T., Kärkkäinen, J., Kempa, D.: A subquadratic algorithm for minimum palindromic factorization. J. Discrete Algorithms **28**, 41–48 (2014)
5. Groult, R., Prieur, É., Richomme, G.: Counting distinct palindromes in a word in linear time. Inf. Process. Lett. **110**(20), 908–912 (2010)
6. Hon, W.-K., Thankachan, S.V., Xu, B.: An in-place framework for exact and approximate shortest unique substring queries. In: Elbassioni, K., Makino, K. (eds.) ISAAC 2015. LNCS, vol. 9472, pp. 755–767. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48971-0_63
7. Hu, X., Pei, J., Tao, Y.: Shortest unique queries on strings. In: Moura, E., Crochemore, M. (eds.) SPIRE 2014. LNCS, vol. 8799, pp. 161–172. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11918-2_16
8. I, T., Sugimoto, S., Inenaga, S., Bannai, H., Takeda, M.: Computing palindromic factorizations and palindromic covers on-line. In: Kulikov, A.S., Kuznetsov, S.O., Pevzner, P. (eds.) CPM 2014. LNCS, vol. 8486, pp. 150–161. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07566-2_16
9. İleri, A.M., Külekci, M.O., Xu, B.: Shortest unique substring query revisited. In: Kulikov, A.S., Kuznetsov, S.O., Pevzner, P. (eds.) CPM 2014. LNCS, vol. 8486, pp. 172–181. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07566-2_18

10. Kuramoto, E., Yano, O., Kimura, Y., Baba, M., Makino, T., Yamamoto, S., Yamamoto, T., Kataoka, T., Tokunaga, T.: Oligonucleotide sequences required for natural killer cell activation. Jpn. J. Cancer Res. **83**(11), 1128–1131 (1992)
11. Manacher, G.: A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. J. ACM **22**, 346–351 (1975)
12. Mieno, T., Inenaga, S., Bannai, H., Takeda, M.: Shortest unique substring queries on run-length encoded strings. In: Proceedings of MFCS 2016, pp. 69:1–69:11 (2016)
13. Pei, J., Wu, W.C.H., Yeh, M.Y.: On shortest unique substring queries. In: Proceedings of ICDE 2013, pp. 937–948 (2013)
14. Rubinchik, M., Shur, A.M.: EERTREE: an efficient data structure for processing palindromes in strings. In: Lipták, Z., Smyth, W.F. (eds.) IWOCA 2015. LNCS, vol. 9538, pp. 321–333. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29516-9_27
15. Tsuruta, K., Inenaga, S., Bannai, H., Takeda, M.: Shortest unique substrings queries in optimal time. In: Geffert, V., Preneel, B., Rovan, B., Štuller, J., Tjoa, A.M. (eds.) SOFSEM 2014. LNCS, vol. 8327, pp. 503–513. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-04298-5_44
16. Yamamoto, S., Yamamoto, T., Kataoka, T., Kuramoto, E., Yano, O., Tokunaga, T.: Unique palindromic sequences in synthetic oligonucleotides are required to induce IFN [correction of INF] and augment IFN-mediated [correction of INF] natural killer activity. J. Immunol. **148**(12), 4072–4076 (1992)