Shortest Unique Substrings Queries in Optimal Time

Kazuya Tsuruta, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda

Department of Informatics, Kyushu University, Japan
{inenaga,bannai,takeda}@inf.kyushu-u.ac.jp

Abstract. We present an optimal, linear time algorithm for the shortest unique substring problem, thus improving the algorithm by Pei et al. (ICDE 2013). Our implementation is simple and based on suffix arrays. Computational experiments show that our algorithm is much more efficient in practice, compared to that of Pei et al.

1 Introduction

The shortest unique substring problem was proposed by Pei et al. [4]. Given a string S and position p, the problem is to find a *shortest unique substring* (SUS) of S that contains position p, that is, a substring that only occurs once in S, and whose occurrence contains position p. They also consider a version of the problem where S may be preprocessed, and SUS queries for arbitrary positions may be answered efficiently.

For the first version of the problem, Pei et al. [4] presented an algorithm that computes the SUS for any given position p in $O(n)$ time and space, where n is the length of string S. For the second version, they present an $O(hn)$ time and $O(n)$ space preprocessing algorithm which allows queries to be answered in constant time, where h is a value depending on S. However, h is only bounded by $O(n)$, and in the worst case, this results in $O(n^2)$ time pre-processing.

The contributions of this paper is as follows: First, we give optimal time solutions for both problems and show that S can be preprocessed in $O(n)$ time so that a SUS for any query position can be answered in $O(1)$ time. This considerably improves the theoretical worst case running time compared to Pei et al. [4], allowing us to output a SUS for *all* positions in the string in $O(n)$ total time. Second, we consider the general problem of computing all SUSs that contain a given position. Although there can be multiple shortest substrings that contain a given query position, Pei et al. [4] only considered the problem of answering a single SUS that contains a position. We show that the same linear time preprocessing above also allows us to return *all* SUSs that contain a given query position in $O(k)$ time, where k is the size of the output. Finally, we implement our algorithm and show through computational experiments that our implementation is much more practical and scalable compared to an implemention of the algorithm by Pei et al. [4] made available by the authors.

2 Preliminaries

2.1 Strings

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

Given two distinct positions i, j $(i < j)$, we say that i is to the *left* and j the *right*. Two distinct intervals are *nested*, if one is a subset of the other. For two non-nested intervals $[i..j]$ and $[i'..j']$, we say that $[i..j]$ is to the left and $[i'..j']$ is to the right, if $i < j$. Since, for any interval $[i..j]$ $(1 \leq i \leq j \leq |S|)$ there is a corresponding substring $S[i..j]$ of S, we abuse the language and will many times call an interval a substring.

2.2 Unique Substrings

We say that a substring w of S is *unique*, if there is exactly one occurence of w in S. When w is unique, the interval $[i..i+|w|-1]$ such that $S[i..i+|w|-1] = w$ is called a unique interval of S. We say that a unique substring w of S contains position p, if $w = S[i..i+|w|-1]$ and $p \in [i..i+|w|-1]$. It is easy to see that any string that contains a substring that is unique, is also unique, and any interval that contains a sub-interval that is unique, is also unique.

Definition 1 (Shortest Unique Substring). *A substring w is a shortest unique substring (SUS) of S that contains position p, if $w = S[i..j]$ is unique in S, $i \leq p \leq j$, and no other substring $w' = S[i'..j']$ such that $i' \leq p \leq j'$ and $j' - i' < j - i$ is unique in S.*

Note that there can be more than one SUS that contains position p as shown in the following example. Let $SUS_S(p)$ denote the set of intervals corresponding to SUSs of S that contains position p. Note that $SUS_S(p) \neq \emptyset$ for any position $1 \leq p \leq |S|$.

Example 1 (SUS). Let $S =$ `aabaabcabababbaabdbab`. Then, $SUS_S(2) = \{[1..4], [2..5]\}$, $SUS_S(4) = \{[1..4], [2..5], [4..7]\}$, $SUS_S(9) = \{[7..9]\}$, $SUS_S(10) = \{[10..12]\}$. (See Fig. 1)

In this paper, we focus on the following problems.

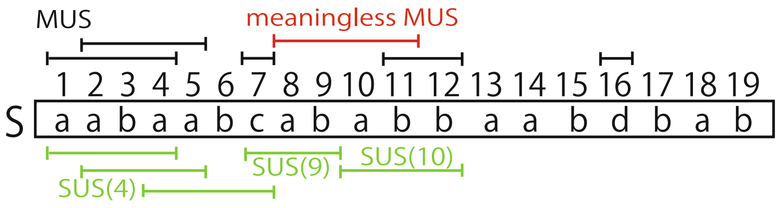

Fig. 1. Example of a string and its SUSs (see Definition 1) and MUSs (see Definition 2). Although all 6 MUSs are depicted, $SUS(p)$ is depicted only for positions 4, 9 and 10. $MUS_S = \{[1..4], [2..5], [7..7], [8..11], [11..12], [16..16]\}$. $SUS_S(4) = \{[1..4], [2..5], [4..7]\}$, $SUS_S(9) = \{[7..9]\}$, $SUS_S(10) = \{[10..12]\}$. The MUS [8..11] is meaningless since no SUS contains it, while the others are meaningful (see Definition 7).

Problem 1 (SUS query). Given string S of length n, compute for all positions p ($1 \leq p \leq n$), a shortest unique substring that contains position p.

Problem 2 (All SUS query). Given string S of length n, compute for all positions p ($1 \leq p \leq n$), all shortest unique substrings that contain position p.

Problem 1 was first considered by Pei et al. [4]. They first gave a simple $O(n)$ time algorithm for computing an SUS for a single p. However, this would result in $O(n^2)$ time for computing a SUS for each p. Thus, they further showed an improved algorithm which pre-process S in $O(hn)$ time, and allows queries for any p in $O(1)$ time, where h is a parameter that depends on S. This results in an $O(hn)$ time solution for computing the SUS for all positions $1 \leq p \leq n$. Although Pei et al. [4] gave empirical evidence that h is not very large in practice, they were not able to give a good theoretical bound on h, mentioning that h can be as large as $O(n)$, resulting in $O(n^2)$ time worst case pre-processing time.

In this paper, we give optimal time solutions for both problems, and show that S can be preprocessed in $O(n)$ time so that the queries can be answered in $O(k)$ time, for any query position p, where k is the size of the output. Noting that k is $O(1)$ for Problem 1, this results in an $O(n)$, i.e. a truly linear time solution for computing the SUS for all positions $1 \leq p \leq n$.

Like the algorithm by Pei et al. [4], our algorithm finds SUSs, based on the concept of Minimal Unique Substrings defined below.

Definition 2 (Minimal Unique Substring). *A substring w of S is a minimal unique substring (MUS) if w is unique in S, and no proper substring of w is unique in S.*

Let MUS_S denote the set of intervals corresponding to MUSs of string S. Notice that by definition, MUSs of S can overlap with each other, but cannot be nested. This implies that there can exist at most one MUS starting at a given position in S. Also, since there must exist at least one MUS, we have $0 < |MUS_S| \leq |S|$.

Example 2 (MUS). Let $S =$ `aabaabcababbaabdbab`, the same string as in Example 1. $MUS_S = \{[1..4], [2..5], [7..7], [8..11], [11..12], [16..16]\}$. (See Fig. 1)

2.3 Data Structures

We utilize the following data structures and algorithms. While the main data structure used by Pei et al. [4] was the suffix tree [5], we use the suffix array [3], which is theoretically almost equivalent to the suffix tree, but more time and space efficient in practice.

Definition 3 (Suffix Array [3]). *The suffix array SA of a string S of length n is a permutation of integers $\{1, \ldots, n\}$, such that $SA[i] = j$ represents the ith lexicographically smallest suffix $S[j..n]$ of S.*

Theorem 1 ([1]). *Assuming an integer alphabet, the suffix array of a string S of length n can be constructed in $O(n)$ time.*

Definition 4 (Rank array). *The rank array SA^{-1} of a string S of length n, is a permutation of integers $\{1, \ldots, n\}$, such that $SA^{-1}[SA[i]] = i$.*

Given SA, SA^{-1} can be computed in $O(n)$ time by a simple loop over SA.

Definition 5 (LCP array). *The longest common prefix (lcp) array LCP of a string S of length n, is an array of integers where $LCP[1] = 0$ and $LCP[i]$ for $1 < i \leq n$ holds the length of the longest common prefix between suffix $S[SA[i-1]..n]$ and $S[SA[i]..n]$, where SA is the suffix array of S.*

Theorem 2 ([2]). *Given string S of length n and its suffix array SA, the lcp array LCP of S can be computed in $O(n)$ time.*

3 Algorithm

3.1 Finding All MUSs

Here, we describe how to find all MUSs of a string S in linear time, using the suffix and lcp arrays of S.

Lemma 1. *All MUSs of a string S of length n can be found in $O(n)$ time and space.*

Proof. Let SA and LCP respectively be the suffix array and lcp array of S. For any suffix $S[j..n]$ where $SA[i] = j$ (or $SA^{-1}[j] = i$), the shortest prefix of $S[j..n]$ that is unique is given by $S[j..j + \ell_j]$ where

$$\ell_j = \begin{cases} \max\{LCP[i], LCP[i+1]\} & 1 \leq i < n \\ LCP[i] & i = n. \end{cases}$$

The definition of ℓ_j implies that $S[j..j + \ell_j - 1]$ is not unique. Thus, $S[j..j + \ell_j]$ is the only candidate for a MUS starting at position j, and is a MUS if and only if $S[j+1..j+\ell_j]$ is not unique. Since the definition of ℓ_{j+1} implies that $S[j+1..j+\ell_{j+1}]$ is not unique, $S[j..j+\ell_j]$ is a MUS if and only if $j+\ell_j \leq j+\ell_{j+1}$. Once SA, SA^{-1}, and LCP are computed in $O(n)$ time, this can be checked in $O(1)$ time for each j. Therefore, the lemma follows since ℓ_j for all $1 \leq j \leq n$ can be computed in a total of $O(n)$ time. □

3.2 SUSs from MUSs

Next, we consider the relation between MUSs and SUSs.

Definition 6. *For an interval $[i..j]$ and position p, let $cover([i..j], p)$ denote the smallest interval that contains $[i..j]$ and p, i.e. $cover([i..j], p) = [\min(i, p)..\max(j, p)]$. We say that $cover([i..j], p)$ is derived from interval $[i..j]$ and position p.*

We first show that any $SUS_S(p)$ is derived from an element in MUS_S. The following Lemma is essentially the same as Theorem 2 in [4], but the statement has been reworded for our purposes.

Lemma 2. *For any position p and interval $[i..j] \in SUS_S(p)$, there exists exactly one sub-interval $[i'..j'] \in MUS_S$ of $[i..j]$ such that $[i..j] = cover([i'..j'], p)$.*

Proof. Since $[i..j]$ is unique, it must contain at least one minimal unique sub-interval. Let $[i'..j']$ be any MUS contained in the SUS $[i..j]$. Since $i \leq p \leq j$, $cover([i'..j'], p)$ is unique, contains position p, and is a sub-interval of $[i..j]$. Thus, $[i..j] = cover([i'..j'], p)$ must hold, since otherwise, $cover([i'..j'], p)$ would be an interval shorter than $[i..j]$ containing position p, contradicting the assumption that $[i..j]$ is an SUS.

Next we show that there is exactly one MUS contained in a SUS. Suppose there are two distinct minimal unique sub-intervals $[i_1..j_1]$ and $[i_2..j_2]$ of $[i..j]$. From the above arguments, $[i..j] = cover([i_1..j_1], p) = cover([i_2..j_2], p)$ must hold. Since MUSs cannot be nested, both must be proper sub-intervals of $[i..j]$, and we assume without loss of generality that $i \leq i_1 < i_2$ and $j_1 < j_2 \leq j$. However, if $i \leq p < j$, then $cover([i_1..j_1], p) \neq [i..j]$ since $\max\{p, j_1\} < j$, and if $i < p \leq j$, then $cover([i_2..j_2], p) \neq [i..j]$ since $\min\{p, i_2\} > i$. Thus, there can only be one MUS that is contained in a given SUS. □

For the purpose of describing our algorithm, we define a generalization of SUSs with respect to a subset of MUSs, namely, MUSs that begin at or before a certain position. Let $MUS_S^k = \{[i..j] \in MUS_S \mid i \leq k\}$. We define $SUS_S^k(p)$ to be the subset of intervals which are shortest, of the intervals that can be derived from intervals in MUS_S^k and position p, i.e., $[i..j] \in SUS_S^k(p)$ if $[i..j] = cover([i'..j'], p)$ for some $[i'..j'] \in MUS_S^k$, and $|[i..j]| \leq |cover([i''..j''], p)|$ for any $[i''..j''] \in MUS_k$. Let $lmSUS_S^k(p)$ denote the leftmost interval of $SUS_S^k(p)$, and $lmMUS_S^k(p)$ the interval in MUS_k that derives $lmSUS_S^k(p)$.

Note that $MUS_S = MUS_S^n$, and $SUS_S(p) = SUS_S^n(p)$. Also note that although for any $k < k'$, $MUS_S^k \subseteq MUS_S^{k'}$, it is not necessarily the case that $SUS_S^k(p) \subseteq SUS_S^{k'}(p)$.

Next, we define the concept of *meaningful* and *meaningless* MUSs, which is the main difference of our algorithm with [4].

Definition 7 (Meaningful Minimal Unique Substring). *We say that an interval $[i..j] \in MUS_S^k$ is* meaningful *with respect to MUS_S^k, if, for some position p, $cover([i..j], p) \in SUS_S^k(p)$. Otherwise, we say that a minimal unique substring is meaningless with respect to MUS_S^k.*

Example 3 (Meaningful MUS). Let $S =$ `aabaabcababbaabdbab`, the same string as in Example 1. Then, the set of MUSs $\{[1..4], [2..5], [7..7], [11..12], [16..16]\}$ are meaningful, since they respectively derive SUSs corresponding to positions 4, 9 and 10. However, MUS $[8..11]$ is meaningless, it does not derive any SUS. (See Fig. 1)

Observation 1. *For any $k < k'$, if an interval $[i..j] \in MUS_S^k$ is meaningless with respect to MUS_S^k, then it is meaningless with respect to $MUS_S^{k'}$.*

Let $MMUS_S^k$ denote the set of all meaningful MUSs with respect to MUS_S^k. We first show that if we have an array $MMUS_S = MMUS_S^n$ of meaningful MUSs with respect to MUS_S, in order of their occurrence, and for each position p we hold an index $L[p]$ such that $MMUS_S[L[p]] = lmMUS_S^n(p)$, we can answer $SUS_S(p)$ in $O(|SUS_S(p)|)$ time.

To prove this, we give a more specific characterization of which MUSs can derive elements of $SUS_S(p)$. Let $MUS_S(p)$ denote the set of MUSs that contain position p, i.e.,

$$MUS_S(p) = \{S[i..j] \in MUS_S \mid i \leq p \leq j\}.$$

$MUS_S(p)$ can be empty. For any position p, let $pred_S(p) = [i..j]$ represent the rightmost MUS that occurs before position p if one exists, that is, $[i..j] \in MUS_S$, $j < p$, and there exists no $[i'..j'] \in MUS_S$ such that $j < j' < p$. Similarly, let $succ_S(p) = [i..j]$ represent the leftmost MUS that occurs after position p if one exists, that is, $[i..j] \in MUS_S$, $i > p$, and there exists no $[i'..j'] \in MUS_S$ such that $p < i' < i$. We say that the set $\{pred_S(p), succ_S(p)\} \cup MUS_S(p)$ is the MUSs in the *neighborhood* of position p.

The following lemma shows that $|cover([i..j], p)|$ for MUSs $[i..j]$ in the neighborhood of position p which are meaningful with respect to MUS_S^k and are to the right of $lmMUS_S^k(p)$ (including $lmMUS_S^k(p)$), form a monotonically increasing sequence.

Lemma 3. *Consider any position p and integer k, and let $[i..j] = lmMUS_S^k(p)$. Any two distinct intervals $[i_1..j_1], [i_2..j_2] \in \{\{pred_S(p), succ_S(p)\} \cup MUS_S(p)\} \cap MMUS_S^k$ such that $i \leq i_1 < i_2$, satisfy $|cover([i_1..j_1], p)| \leq |cover([i_2..j_2], p)|$.*

Proof. Suppose to the contrary that $|cover([i_1..j_1], p)| > |cover([i_2..j_2], p)|$. Since $cover([i..j], p) \in SUS_S^k(p)$, it holds that $|cover([i..j], p)| \leq |cover([i_2..j_2], p)| < |cover([i_1..j_1], p)|$. For all positions $i \leq p' < p$, it holds that $|cover([i..j], p')| \leq |cover([i..j], p)| < |cover([i_1..j_1], p)|$. Since $[i..j] = lmMUS_s^k(p)$ and $i < i_1$, it holds that $[i_1..j_1] \neq pred_s(p)$ and $p' < p \leq j_1$. Since $p' < p \leq j_1$, it holds that $|cover([i_1..j_1], p)| \leq |cover([i_1..j_1], p')|$. Similarly, for all positions $p < p' < j_2$, it holds that $|cover([i_2..j_2], p')| = |cover([i_2..j_2], p)| < |cover([i_1..j_1], p)|$. Since $|cover([i_1..j_1], p)| > |cover([i_2..j_2], p)|$, it holds that $[i_1..j_1] \neq succ_s(p)$ and $i_1 \leq p < p'$. It holds that $|cover([i_1..j_1], p)| \leq |cover([i_1..j_1], p')|$.

Also, for any position $p' < i$, $|cover([i..j], p)| < |cover([i_1..j_1], p)|$, and for any position $p' > j_2$, $|cover([i_2..j_2], p)| < |cover([i_1..j_1], p)|$. This implies that $[i_1..j_1]$

cannot be meaningful for all positions $1 \le p' \le n$, and must be meaningless with respect to MUS^k_S, contradicting the assumption that $[i_1..j_1] \in MMUS^k_S$. Thus, it must be that $|cover([i_1..j_1], p)| \le |cover([i_2..j_2], p)|$. □

The next lemma shows that intervals in $SUS^k_S(p)$ are the shortest ones derived from MUSs in the neighborhood of position p which are meaningful with respect to MUS^k_S.

Lemma 4. *Consider position p, integer k, interval $[i..j] \in MUS^k_S$, and let $Y = \{\{pred_S(p), succ_S(p)\} \cup MUS_S(p)\} \cap MMUS^k_S$. If $cover([i..j], p) \in SUS^k_S(p)$, then $[i..j] \in Y$ and $|cover([i..j], p)| \le |cover([i'..j'], p)|$ for all intervals $[i'..j'] \in Y$.*

Proof. Assume $cover([i..j], p) \in SUS^k_S(p)$ holds. Since $Y \subseteq MUS^k_S$ and by the defintion of $SUS^k_S(p)$, $|cover([i..j], p)| \le |cover([i'..j'], p)|$ holds for all $[i'..j'] \in Y$.

It is easy to see that $[i..j]$ cannot be to the left of $pred_S(p)$, since then, $|cover([i..j], p)| > |cover(pred_S(p), p)|$ and $[i..j]$ could not be in $SUS^k_S(p)$. Similarly, $[i..j]$ cannot be to the right of $succ_S(p)$, since then, $|cover([i..j], p)| > |cover(succ_S(p), p)|$ and again, $[i..j]$ could not be in $SUS^k_S(p)$.

Finally, by the definition of meaningful, $[i..j] \in MMUS^k_S$. □

Algorithm 1. $SUS_S(p)$ from L and $MMUS_S$.

```
Input: position $p$, $MMUS_S$, $L$
Output: $SUS_S(p)$
1 $t \leftarrow L[p]$;
2 $l \leftarrow |cover(MMUS_S[t], p)|$ ;                         // length of SUS
3 while $|cover(MMUS_S[t], p)| = l$ do
4     output $cover(MMUS_S[t], p)$;
5     $t \leftarrow t + 1$;
6 end
```

Theorem 3. *Given an array $MMUS_S$ of all meaningful MUSs with respect to MUS_S in order of occurrence, and an array L of size n, where, for each position $1 \le p \le n$, $MMUS_S[L[p]] = lmMUS^n_S(p)$, we can compute $SUS_S(p)$ in $O(|SUS_S(p)|)$ time.*

Proof. The pseudo code of the algorithm is shown in Algorithm 1. By definition of $MMUS_S$ and L, it is clear that the first output is $lmSUS^n_S(p)$, i.e., the leftmost SUS that contains position p. From Lemma 2 and by the definition of a meaningful interval, it is easy to see that all MUSs that derive elements in $SUS_S(p)$ must be in $MMUS_S$.

It remains to prove that each element in $SUS_S(p)$ is derived from MUSs in a contiguous range in $MMUS_S$. This can be seen from Lemmas 3 and 4, which claim that all MUSs in $SUS_S(p)$ are in the neighborhood of position p that are meaningful with respect to MUS_S, and for all such meaningful MUSs $[i..j]$ to the right of $lmMUS^n_S(p)$ (including $lmMUS^n_S(p)$), $cover([i..j], p)$ forms a monotonically increasing sequence. □

Algorithm 2. Create array $MMUS_S$ of meaningful MUSs and an array of pointers L to $lmMUS$ for each position of string S

```
Input: LCP and RANK array for string S.
Output: $MMUS[1..|MMUS.size()|]$: array of meaningful MUSs; $L[1..n]$: index
        in $MMUS$ of leftmost SUS for each position.
1  for $p \leftarrow 1$ to $n$ do
2      $\ell \leftarrow MMUS.size()$;
       // lmMUS for position $p$ wrt $MUS_S^{p-1}$ is either the same as $p-1$,
          or the next one.
3      if $p = 1$ then
4          $L[1] \leftarrow 1$;            // Core MUS of position 1 is leftmost MUS.
5      else if $L[p-1] < \ell$ and
       $|cover(MMUS[L[p-1]+1], p)| < |cover(MMUS[L[p-1]], p)|$ then
6          $L[p] \leftarrow L[p-1]+1$;
7      else
8          $L[p] \leftarrow L[p-1]$;
       // update $MMUS$ and $L$ to values wrt $MUS_S^p$
9      if exists MUS: $newMUS = [p, e]$ for some $e \geq p$. then // $O(1)$ time using
       LCP and RANK array
10         if $\ell > 0$ then
               // $j$: rightmost position that doesn't need update
11             $j \leftarrow \max\{i \leq p \mid |cover(MMUS[L[i]], i)| \leq |cover(newMUS, i)|\}$;
12             if $j = p$ then  // No updates for $L$. Remove meaningless MUSs
               from $MMUS$
13                 $MMUS \leftarrow MMUS[1..k]$ where
                   $k = \max\{k' \leq \ell \mid |cover(MMUS[k'], p)| \leq |cover(newMUS, p)|\}$;
14             else  // remove meaningless MUSs after the one pointed by $j$
               and $newMUS$
15                 $MMUS \leftarrow MMUS[1..k]$ where $k = \max\{k' \leq \ell \mid$
                   $|cover(MMUS[k'], j)| \leq |cover(MMUS[L[j]], j)|\}$;
16                 for $j+1 \leq i \leq p$ do $L[i] \leftarrow k+1$;   // update L to new MUS
17         $MMUS.push\_back(newMUS)$;
```

Next we show that $MMUS_S$ and L can be constructed in linear time, by incrementally updating $MMUS_S^k$ and L. Let L^k denote an array of indices where $MMUS_S^k[L^k[p]] = lmMUS_S^k(p)$.

Lemma 5. *$L^{p-1}[p]$ is either the MUS $[i..j]$ pointed to by $L^{p-1}[p-1]$, or the next MUS $[i'..j']$ in $MMUS_S^{p-1}$, i.e., the one pointed to by $L^{p-1}[p-1]+1$.*

Proof. By definition, $[i..j] = lmMUS_S^{p-1}(p-1)$. Let $[i''..j'']$ be an arbitary interval in $MMUS_S^{p-1}$ to the right of $[i..j]$. Then, $[i''..j''] \in MUS_S^{p-1}(p-1) \cap MUS_S^{p-1}(p)$, since we have that $i < i'' \leq p-1$, and if $j < j'' < p$, then $|cover([i..j], p-1)| = |[i..p-1]| > |[i''..p-1]| = |cover([i''..j''], p-1)|$

contradicting the definition of $[i..j]$. Thus, we have that $|cover([i''..j''], p-1)| = |cover([i''..j''], p)|$, and from Lemma 3, these values are monotonically increasing. Therefore, the first one, which is $[i'..j'] = MMUS_S^{p-1}[L^{p-1}[p-1]+1]$, gives the smallest value. Note that $[i_p..j_p] = lmMUS_S^{p-1}(p)$ cannot be to the left of $[i..j]$; If $p \le j_p$, then since $i_p < i < p \le j_p < j$ and from the definition of $[i_p..j_p]$, we have $|cover([i_p..j_p], p-1)| = |cover([i_p..j_p], p)| \le |cover([i..j], p)| = |cover([i..j], p-1)|$ which contradicts the definition of $[i..j]$ If $j_p \le p-1$, then $cover([i_p..j_p], p-1) + 1 = cover([i_p..j_p], p) \le cover([i..j], p) \le cover([i..j], p-1) + 1$, again contradicting the definition of $[i..j]$. Thus, $lmMUS_S^{p-1}(p)$ must be either $[i..j]$ or $[i'..j']$. $\square$

Theorem 4. *$MMUS_S$ and L can be constructed in linear time.*

Proof. The pseudo code of the algorithm is shown in Algorithm 2. The algorithm computes $MMUS_S$ and L for increasing positions. For each value of p, we assume that $MMUS_S^{p-1}$ and $L^{p-1}[1..p-1]$ are correctly computed, and we update them to correct values of $MMUS_S^p$ and $L^p[1..p]$.

Lines 3-8 in Algorithm 2 compute $L^{p-1}[p]$ from $L^{p-1}[p-1]$, and $MMUS_S^{p-1}$. The correctness can be seen from Lemma 5. The calculation for updating L can be done in constant time for each position.

Next, we show how to compute $MMUS_S^p$ and $L^p[1..p]$ given $MMUS_S^{p-1}$ and $L^{p-1}[1..p]$. The existence of an MUS starting at position p can be checked in constant time with Lemma 1. If there exists no such MUS, then, since $MUS_S^{p-1} = MUS_S^p$, $MMUS_S^p = MMUS_S^{p-1}$ and $L^p[p] = L^{p-1}[p]$, and no update is required. If there does exist $[p..e] \in MUS_S$ for some $e \ge p$, we check previous positions $i \le p$ to see if $L^{p-1}[i]$ needs to be updated to $L^p[i]$. Such positions i satisfy $|cover(MMUS_S^{p-1}[L^{p-1}[i]], i)| > |cover([p..e], i)|$, and if for some position j this does not hold, then it is easy to see that it does not hold for all $j' \le j$. Let j be the rightmost position such that the condition does not hold, i.e., $L^{p-1}[1..j]$ does not need to be updated.

If $j = p$, this means that no values in $L^{p-1}[1..p]$ need to be updated, and $L^p[p] = L^{p-1}[p]$. Concerning updating $MMUS_S^{p-1}$, we can easily see that $cover([p..e], n) \in SUS_S^p(n)$, and thus $[p..e]$ will be the last element in $MMUS_S^p$. However, MUSs in $MMUS_S^{p-1}$ may become meaningless with respect to MUS_S^p, because of the addition of $[p..e]$. These are the ones to the right of $[i'..j'] = MMUS_S^{p-1}[L^p[p]]$. They can be found and removed in line 13, whose correctness can be seen from Lemma 3.

If $j < p$, MUSs in $MMUS_S^{p-1}[L[j]+1..\ell]$ such that $|cover(MMUS_S[k'], j)| > |cover(MMUS_S[j], j)|$, i.e., those that do not derive an interval in $SUS_S^p(j)$ become meaningless with respect to MUS_S^p, so are removed in line 15. The correctness can also be seen from Lemma 3.

Although there may be more than a constant number of positions and MUSs that need to be updated with the addition of $[p..e]$, the cost can be amortized. Such operations correspond to lines 11, 13, 15, and 16 of Algorithm 2.

The time required for lines 11 and 16 is linear in the number of updates required for L. We show that $L[p]$ for each p is updated only a constant number of times. $L^{p-1}[p]$ is first determined at lines 3-8, with respect to MUS_S^{p-1}, pointing to $pred_S(p)$ or the leftmost shortest element in $MUS_S(p) \cap MUS_S^{p-1}$. It can be seen from Lemma 4 that for all $p' \geq p$, $L^{p'}[p]$ can only point to $pred_S(p)$, the leftmost shortest element in $MUS_S(p)$ $(= MUS_S(p) \cap MUS_S^{p'})$, or $succ_S(p)$. There are only two possibly remaining MUSs that will be added to $MUS_S(p) \cap MUS_S^{p-1}$ and update $L[p]$; an MUS in $MUS_S(p)$ beginning at position p, or $succ_S(p)$. Thus, the total time for this is linear in the number of positions.

The time required for lines 13 and 15, is linear in the number of intervals added or deleted from *MMUS*. Since each interval in *MMUS* is added or removed at most once, the total time for this update is linear in the total number of MUSs in S, which is $O(n)$. Thus, the total time of the algorithm is $O(n)$. □

From Theorems 3 and 4, we obtain the following main theorem.

Theorem 5. *A string S of length n can be preprocessed in $O(n)$ time and space so that shortest unique substring queries can be answered in $O(k)$ time, where k is the number of shortest substrings returned. Notably, outputting a single SUS can be done in $O(1)$ time.*

4 Computational Experiments

We implemented our algorithm using the C++ language. All computational experiments were conducted on a MacPro (Early 2008) with two 3.2GHz Quad Core Xeon processors and 18GB Memory (DDR2 FB-DIMM 800MHz). We use libdivsufsort (`http://code.google.com/p/libdivsufsort/`) for construction of the suffix array.

Table 1. Comparison of Computation Time

	english ($\|\Sigma\|$ =239)		dna ($\|\Sigma\|$ =16)		dblp.xml ($\|\Sigma\|$ =97)		protein ($\|\Sigma\|$ =27)	
n (MB)	time (sec)		time (sec)		time (sec)		time (sec)	
	TSUS	RSUS	TSUS	RSUS	TSUS	RSUS	TSUS	RSUS
10	4.21	122.31	4.79	18.63	3.42	14.34	4.01	28.28
20	9.16	324.58	10.54	40.46	7.44	29.98	9.04	66.74
30	14.13	445.84	16.45	61.80	11.43	46.51	14.57	108.00
40	20.14	500.19	23.06	84.75	16.17	62.76	21.68	151.85
50	25.62	580.00	29.31	107.34	20.35	78.73	28.90	197.99
60	31.20	667.16	36.08	131.38	24.62	95.55	35.61	242.55
70	38.26	N/A	43.90	N/A	30.14	728.71	43.96	N/A
80	44.00	N/A	50.83	N/A	34.67	N/A	51.01	N/A
90	50.37	N/A	57.88	N/A	39.03	N/A	58.13	N/A
100	56.71	N/A	65.17	N/A	43.30	N/A	64.22	N/A

We used data taken from the Pizza & Chile corpus (`http://pizzachili.dcc.uchile.cl/texts.html`), namely, english texts, DNA sequences, XML, and protein sequences. We compared our algorithm with the implementation RSUS of [4] available at `https://bitbucket.org/wush_iis/rsus`. RSUS is actually a combination of an interface for the R language (`http://www.r-project.org`) and core routines written in C++. For comparison in our experiments, we modified the RSUS C++ routines to be called from a C++ program so that all programs utilize only the C++ language.

The results of experiments for the 4 data are shown in Table 1. We take a prefix of length n for each data, and measure the running times of RSUS [4], and TSUS (the implementation of the algorithm in this paper). The entries marked N/A for RSUS was when the time exceeded 1 hour, at which time the execution of the program was stopped. The cause for the sudden increase in running times for RSUS was due to the fact that RSUS consumed all of the available physical memory. The results show that our algorithm is much faster (as fast as 20 times) in preprocessing time compared to RSUS.

References

1. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
2. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Amir, A. (ed.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
3. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Computing 22(5), 935–948 (1993)
4. Pei, J., Wu, W.C.H., Yeh, M.Y.: On shortest unique substring queries. In: Proc. ICDE, pp. 937–948 (2013)
5. Weiner, P.: Linear pattern-matching algorithms. In: Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, pp. 1–11. Institute of Electrical Electronics Engineers, New York (1973)