On Shortest Unique Substring Queries

Jian Pei [1], Wush Chi-Hsuan Wu [*2], Mi-Yen Yeh [*3]

[1]*School of Computing Science, Simon Fraser University*
Burnaby, BC, Canada
jpei@cs.sfu.ca

[*]*Institute of Information Science, Academia Sinica*
Taipei, Taiwan
[2]wush978@gmail.com
[3]miyen@iis.sinica.edu.tw

***Abstract*— In this paper, we tackle a novel type of interesting queries — *shortest unique substring queries*. Given a (long) string S and a query point q in the string, can we find a shortest substring containing q that is unique in S? We illustrate that shortest unique substring queries have many potential applications, such as information retrieval, bioinformatics, and event context analysis. We develop efficient algorithms for online query answering. First, we present an algorithm to answer a shortest unique substring query in $O(n)$ time using a suffix tree index, where n is the length of string S. Second, we show that, using $O(n \cdot h)$ time and $O(n)$ space, we can compute a shortest unique substring for every position in a given string, where h is variable theoretically in $O(n)$ but on real data sets often much smaller than n and can be treated as a constant. Once the shortest unique substrings are pre-computed, shortest unique substring queries can be answered online in constant time. In addition to the solid algorithmic results, we empirically demonstrate the effectiveness and efficiency of shortest unique substring queries on real data sets.**

I. Introduction

You are searching the Complete Works of William Shakespeare using query term "king". The term "king" occurs $1,546$ times in $1,392$ speeches within 40 works, even without counting those related words like "king's" and "kings".[1] Using modern information retrieval techniques, such as an inverted index, one can find all occurrence positions of a query word easily. How can a search engine, however, present an informative list of the search results? Showing all occurrence positions, i.e., the line and page numbers of the occurrences, is likely not very helpful for a reader who is not a master of Shakespeare's works. As common practice, a modern search engine may show a snippet for each occurrence, where the length of snippets is predefined globally. On the one hand, if the length is short, some snippets may be identical and thus those occurrences still cannot be distinguished. On the other hand, if the length is long, then the snippets may overwhelm users. A smarter way is to present for each occurrence a shortest snippet that contains the query term and is different from all other snippets of the query term. In other words, we should list for each occurrence a shortest unique snippet. Now, the challenge is for each query position how we can quickly find a shortest unique snippet.

The above simple yet effective application in document search introduces an interesting novel problem to be tackled in this paper. Given a (long) string S and a query point q in S, we want to conduct a *shortest unique substring query* that finds a shortest unique substring containing q.

Shortest unique substring queries have many potential applications. In addition to the above document search example, shortest unique substring queries can be used in bioinformatics. For example, unique substrings can help to find the signature patterns and discover the distinctness between closely related organisms [1]. Moreover, finding shortest unique substrings on DNA sequences can help polymerase chain reaction (PCR) primer design in molecule biology. Also, it can help to identify unique DNA signatures of closely related species or organisms. As another example, in event analysis, to understand how an event is different from many events of the same kind in a long sequence of historical events, it is useful to extract the context of the event. The shortest unique substring of the event under investigation may serve as a concrete working base of the event context.

Answering shortest unique substring queries efficiently is far from trivial. A brute-force (heuristic) search may easily lead to cost in time quadratic to the length of the string, which is unacceptable in practice when the string is long and queries are expected to be answered online. In this paper, we address the problem of answering shortest unique substring queries from the algorithmic point of view and make several contributions.

First, we model shortest unique substring queries and explore their properties thoroughly. The properties clearly distinguish shortest unique substring queries from the existing related problems, such as computing global minimal substrings.

Second, we present an algorithm to answer a shortest unique substring query in $O(n)$ time using a suffix tree index, which

Authors are listed alphabetically. Pei's research in this paper was supported in part by an NSERC Discovery Grant, a BCFRST NRAS Endowment Research Team Program project, and a GRAND NCE project; and Wu and Yeh's by the National Science Council of Taiwan, R.O.C., under Contracts NSC100-2221-E-001-023 and NSC101-2221-E-001-013. Yeh was also supported by an Ebco/Epic Visiting Chair Fellowship of Simon Fraser University. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

[1]http://www.opensourceshakespeare.org/

can be constructed in $O(n)$ time and space, where n is the length of string S.

Third, we show that, using $O(n \cdot h)$ time and $O(n)$ space, we can compute a shortest unique substring for every position in a given string, where h is variable theoretically in $O(n)$ but on real data sets often much smaller than n and can be treated as a constant.

Last, in addition to the solid algorithmic results, we empirically demonstrate the effectiveness and efficiency of shortest unique substring queries on real data sets.

The rest of the paper is organized as follows. We define shortest unique substring queries in Section II, and review the related work briefly in Section III. We present a query answering algorithm using suffix tree in Section IV. In Section V, we give a constant time online query answering algorithm, which precomputes shortest unique substrings efficiently. We report the experimental results in Section VI, and conclude the paper in Section VII.

II. Shortest Unique Substring Queries

In this section, we formulate the shortest unique substring queries, and discuss the properties of several critical concepts.

A. Shortest Unique Substring Queries

Let S be a **string** of length n, that is, $|S| = n$. Denote by $S[i]$ the value at the i-th **position** of S $(1 \leq i \leq n)$, and $S[i, j] = S[i] \cdots S[j]$ $(1 \leq i \leq j \leq n)$ the **substring** starting at position i and ending at position j. The **length** of the substring is $|S[i, j]| = j - i + 1$. Substring $S[i, j]$ is said to **contain** position p if $i \leq p \leq j$. Moreover, substring $S[i, j]$ is said to **contain** substring $S[i', j']$ if $i \leq i' \leq j' \leq j$.

For two strings X and Y, X and Y are **identical**, denoted by $X = Y$, if $|X| = |Y|$ and for every $1 \leq i \leq |X|$, $X[i] = Y[i]$. X is called a **substring** of Y, denoted by $X \subseteq Y$, if $|X| \leq |Y|$ and there exists a number i such that $1 \leq i \leq |Y| - |X| + 1$ and $X = Y[i, i + |X| - 1]$. We call X a **proper substring** of Y, denoted by $X \subset Y$, if $X \subseteq Y$ but $X \neq Y$.

Definition 1 (Minimal unique substring (MUS)): A substring $S[i, j]$ is **unique** in S if there does not exist another substring $S[i', j']$ $(i \neq i', j \neq j')$ such that $S[i, j] = S[i', j']$. $S[i, j]$ is called a **minimal unique substring** (**MUS** for short) if $S[i, j]$ is unique and there is not any proper substring of $S[i, j]$ that is also unique, that is, every substring $S[i', j']$ is not unique, where $i \leq i' \leq j' \leq j$ and $j - i > j' - i'$. ■

Example 1 (MUS): Let $S = abbbcabb$. $|S| = 8$. $S[3, 6] = bbca$ is a unique substring, but not minimal, since some proper substrings of $S[3, 6]$, such as $S[4, 6] = bca$, $S[4, 5] = bc$ and $S[5, 6] = ca$, are also unique. $S[4, 5] = bc$ and $S[5, 6] = ca$ are minimal unique substrings. ■

Given a position p at string S, we are interested in a substring $S[i, j]$ containing position p, i.e., $i \leq p \leq j$, such that $S[i, j]$ is unique and as short as possible.

Definition 2 (Shortest unique substring (SUS)): Given a string S and a position p in S, a substring $S[i, j]$ is a **shortest unique substring** (**SUS** for short) at position p if $S[i, j]$ is unique and contains p, and there does not exists another unique substring $S[i', j']$ such that $S[i', j']$ also contains p and $j' - i' < j - i$. ■

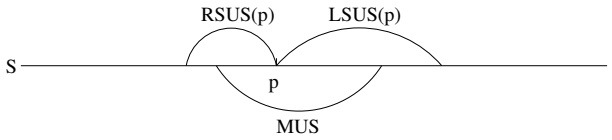


Fig. 1. The relationship among $LSUS(p)$, $RSUS(p)$ and the MUSs containing p. Each arc represents a substring.

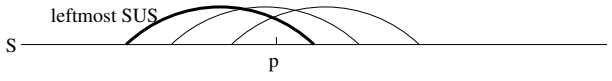


Fig. 2. The leftmost SUS at position p (the one in bold). Each arc represents a SUS at position p. All the three SUSs have the same length.

Example 2 (SUS): Consider string $S = aabccabbcb$. For position $p = 3$, there are 3 shortest unique substrings containing position p, $S[1, 3] = aab$, $S[2, 4] = abc$, and $S[3, 5] = bcc$.

Interestingly, $S[2, 4] = abc$ is a MUS, but $S[1, 3] = aab$ and $S[3, 5] = bcc$ are not, since $S[1, 2] = aa$ and $S[4, 5] = cc$, respectively, are proper substrings that are unique. ■

As shown in Example 2, for a position p, there may exist more than one SUS. We denote by $SUS(p)$ the set of SUSs at position p. Similarly, for a position p, we denote by $MUS(p)$ the set of MUSs containing p. Example 2 clearly shows that MUSs and SUSs are different.

Definition 3 (Problem definition): Given a string S and a position p $(1 \leq p \leq |S|)$, the **shortest unique substring query** (**SUSQ** for short) is to find a SUS at position p. Any member in $SUS(p)$ is a valid answer. ■

In our algorithm design, we often consider two types of unique substrings that may be candidates of SUSs. We give the definitions here and will pursue further discussion later.

Definition 4: Given a string S and a position p in S, a substring $S[p, j]$ is called the **left-bound SUS** (**LSUS** for short) for position p, denoted by $LSUS(p)$, if $S[p, j]$ is unique and no other substring $S[p, j']$ is also unique for $p \leq j' < j$. Symmetrically, $S[i, p]$ is called the **right-bound SUS** (**RSUS** for short) for position p, denoted by $RSUS(p)$, if $S[i, p]$ is unique and no other substring $S[i', p]$ is also unique for $i < i' \leq p$.

Moreover, we define the **leftmost SUS** be the SUS whose starting point is smallest, denoted by leftmost-SUS(p) $= \arg\min_{S[i,j] \in SUS(p)}\{i\}$. ■

Figure 1 shows the relationship among $LSUS(p)$, $RSUS(p)$ and the MUSs containing p. Figure 2 illustrates the concept of leftmost SUS.

It is easy to see the following property.

Property 1 (LSUS and RSUS): Given a string S, for every position p in S, $LSUS(p)$ and $RSUS(p)$, if exist, are unique, respectively. ■

In some cases, LSUSs or RSUSs may not exist. Moreover, in some cases, LSUSs or RSUSs may not be SUSs.

Example 3 (LSUS and RSUS): In string $S = aaaaa$, neither $LSUS(3)$ nor $RSUS(3)$ exist, since a, aa, and aaa are

not unique.

Consider string $S = abbbbc$. $LSUS(2) = S[2,5] = bbbb$ is the shortest unique substring starting from position 2. This unique substring, however, is longer than $S[1,2] = ab$ containing position 2, which is a SUS at position 2. Therefore, $LSUS(2)$ in this example is not a SUS. ■

B. Properties

In this section, we explore a series of interesting properties related to shortest unique substring queries. We start with an essential monotonicity of MUSs.

Lemma 1 (Monotonicity): In a string S, if $S[i,j]$ is a MUS, then any $S[i',j']$ containing $S[i,j]$ is also unique.

Proof: Suppose $S[i',j']$ $(i' \leq i, j \leq j')$ is not unique, that is, there exists $S[i',j'] = S[i'',j'']$ such that $i' \neq i''$. Then, $S[i,j] = S[i''+i-i'+1, j''+j-i'+1]$. This contradicts the assumption that $S[i,j]$ is a MUS, which is unique. ■

For a string S, do the MUSs cover the whole string? Specifically, for a position p, denote by $MUS(p)$ the set of MUSs containing p. We have the following property.

Theorem 1: Given a string S, there exists at least one MUS. For a position p in S (i.e., $1 \leq p \leq |S|$),

$$0 \leq |MUS(p)| \leq \lceil \frac{|S|}{2} \rceil \tag{1}$$

and the lower and upper bounds are reachable.

Proof: S itself is unique. Therefore, either at least one substring of S or S itself is a MUS.

Consider string $S = aabccabc$. Every unique substring containing position 3, such as aab, bcc, $aabc$ and $abcc$, must contain at least one proper substring that is unique in S but does not contain position 3. Therefore, $MUS(3) = \emptyset$ and $|MUS(3)| = 0$. Thus, the lower bound is reachable.

Apparently, for any two MUSs $S[i,j]$ and $S[i',j']$, $i \neq i'$ and $j \neq j'$. Otherwise, one is a proper substring of the other and thus violates the minimality requirement. A MUS $S[i,j]$ containing position p must satisfy $i \leq p \leq j$. Therefore, position p can be contained in at most $\min\{p, |S| - p + 1\}$ MUSs. Thus,

$$|MUS(p)| \leq \max_{l=1}^{|S|}\{\min\{l, |S| - l + 1\}\} = \lceil \frac{|S|}{2} \rceil$$

Now, we show an example where the upper bound can be reached. Consider string $S = abba$. Position 2 is contained in MUSs $S[1,2] = ab$ and $S[2,3] = bb$. That is, $|MUS(2)| = 2$ and reaches the upper bound. ■

Theorem 1 indicates that MUSs may overlap. Please note that it is very likely that the upper bound in Equation 1 can be improved. A strong hint is that we cannot construct an example to reach the upper bound in Equation 1 for a string of non-trivial length, say 10 or longer. Limited by space, we omit the details here, and leave the improvement of the upper bound to future work.

Interestingly, not every position is contained by at least one MUS. Then, what is the relationship between MUSs and SUSs?

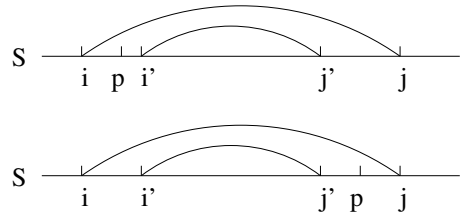


Fig. 3. SUS $S[i,j]$ and MUS $S[i',j']$ in the proof of Theorem 2.

Theorem 2 (SUS): Given a string S, for every position p in S, $SUS(p) \neq \emptyset$. Moreover, for every SUS $S[i,j]$ at position p, $S[i,j]$ contains a MUS $S[i',j']$. For every MUS $S[i',j']$ contained by $S[i,j]$, $i = i'$ or $j = j'$.

Proof: Assume $SUS(p) = \emptyset$. Apparently, S itself is unique and contains position p. A contradiction. Thus, $SUS(p) \neq \emptyset$.

Suppose we have a SUS $S[i,j]$ at position p, that is, $i \leq p \leq j$. $S[i,j]$ must contain at least one MUS. Otherwise, $S[i,j]$ itself is a MUS.

Assume that $S[i,j]$ is a SUS at position p and contains a MUS $S[i',j']$ such that $i < i'$ and $j' < j$ (Figure 3). Then, $|S[i,j]| \geq 4$, and either $p \leq j'$ or $p > j'$. If $p \leq j'$ (the upper case in Figure 3), then $S[i,j']$ is unique since it is a superstring of MUS $S[i',j']$ (Lemma 1), and is shorter than $S[i,j]$. Similarly, if $p > j'$ (the lower case in Figure 3), then $S[i',j]$ is unique and shorter than $S[i,j]$. This contradicts the assumption that $S[i,j]$ is a SUS at position p. ■

III. Related Work

String processing methods have drawn wide attention for a long time as they have been extensively applied in many applications, such as keyword search and text parsing in documents, sequence analysis in bioinformatics, and event stream processing in information systems. Some well studied string finding problems include locating all or the n-th occurrence(s) of some given patterns, finding longest/shortest common substrings, unique or repeated substrings.

To deal with the string operations efficiently, suffix trees [2] and suffix arrays [3] are two powerful and frequently adapted tools in existing methods. Both data structures can be constructed in linear time using linear space [4], [5], [6], [7]. In this paper, we choose to use the suffix tree structure that can obtain $LSUS(p)$ in O(1) time for any position p (details in Section IV).

Finding unique substrings is an important task in biological applications as it can help to find the signature patterns or discover the distinctness between closely related organisms. For example, Haubold *et al.* [1] proposed to find the unique shortest substrings to complete a number of sequence comparison tasks without doing time-consuming sequence alignments. For each position i of a string S, they determine a substring $S[i, ...i+x-1]$ such that it is unique while $S[i..i+x-2]$ is not, essentially the $LSUS(i)$ in our notation.

There are some critical differences between our work and [1]. First, as shown in Example 3, the shortest unique

substring in [1], i.e., $LSUS(p)$ in our notation, may not even be a SUS. In our study, we find SUSs. LSUSs are only used as possible candidates of SUSs. Second, Haubold *et al.* [1] did not develop a new detection method but just relied on the suffix tree method in [8]. They focused on the utility of the unique shortest substrings in some biological applications, such as unique genome region discovery, and the probability distribution of those shortest unique substrings on different biosequences. We expect that our new technique in shortest unique substring finding may also help those biological applications.

Ilie and Smyth [9] found minimum unique substrings and maximum repeats in a string in linear time using suffix arrays. The notion of minimum unique substring in [9] is the same as our definition of MUS in this paper, while a maximum repeat is a substring $S[i, j]$ of S such that it occurs at least twice in S but the strings $S[i-1, j]$ and $S[i, j+1]$, if defined, do not. They found there was a duality relationship between the minimum unique substrings and maximum repeats, which means an algorithm for finding minimum unique substrings can be easily transformed into one that can compute maximum repeats, and vice versa. Our work does not find minimum unique substrings. Instead, we find a shortest unique substring at any position p in a sequence.

Recently, Ye *et al.* [10] proposed to find unique-m substrings, which are patterns that each has only one single exact match on one strand of the entire genome while all other approximate matches must have more than m mismatches. Clearly, they addressed a different problem from ours.

Chan *et al.* [11] and Ji *et al.* [12] proposed to find emerging substrings and minimal distinguishing subsequences, respectively, for sequence classification. Both the emerging substrings and minimal distinguishing subsequences are those frequent in one class but infrequent in the other, where each class contains many strings/sequences. Therefore, the problem is fundamentally different from ours.

Last but not least, our work of finding shortest unique subsequence is just the opposite of those finding frequent patterns in a given long string, such as mining sequence motifs [12], frequent substring mining in a long string [13], and finding maximal frequent word sequences in documents [14].

In summary, to the best of our knowledge, the problem of shortest unique substring queries has not been identified and tackled systematically in literature. The existing methods cannot be straightforwardly adapted to tackle the problem.

IV. Query Answering Using Suffix Trees

In this section, we first review suffix trees and the construction. Then, we discuss how to use a suffix tree as an index to answer shortest unique substring queries.

A. Suffix Trees and Construction

A suffix tree is a data structure that concisely records all possible suffixes of a given string and allows fast string search operations. A string S of length n has n suffixes $S[i, n]$, $i = 1, \ldots, n$. In the suffix tree of S, each edge represents a substring of S, and a path from the root to a leaf node represents exactly one suffix of S.

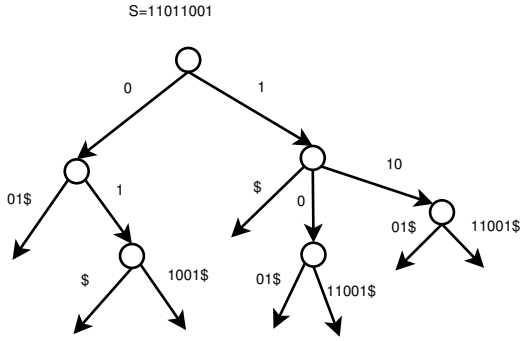


Fig. 4. The suffix tree of $S = 11011001$.

Ukkonen [4] proposed a well-known suffix tree construction method that requires only linear time and space when the alphabet size of a string is a constant. Taking $S = 11011001$ as an example, we briefly show how to construct its suffix tree, as shown in Figure 4, using Ukkonen's algorithm. The construction procedure is illustrated in Figure 5. Generally, the suffix tree is built in $|S|$ phases, 8 phases in this example. At the end of phase i, we have the suffix tree of the prefix $S[1, i]$. To extend the suffix tree of $S[1, i]$ to $S[1, i+1]$, i.e., to ensure that $S[j, i+1]$ is in the tree, we need to extend $S[j, i]$ for $1 \leq j \leq i$, with $S[i+1]$. There are three possible cases.

1) $S[j, i]$ ends at a leaf node. Then, we pad $S[i+1]$ to the corresponding leaf edge.
2) $S[j, i]$ does not end at a leaf node and is not followed by $S[i+1]$. Then, we split the edge and create a new node.
3) $S[j, i]$ does not end at a leaf node but followed by $S[i+1]$, i.e., $S[j, i+1]$ already exists in the tree. In this case, we do not need to do anything.

When we expand the tree from $j = 1$ to $j = i$ during phase $i+1$, the occurrences of this three phases follow some properties. First, after case 2 or case 3 happen, then case 1 will never happen again. Moreover, case 2 will never happen again after case 3 happens. With these properties, once we meet case 3 at step j of phase i, we can immediately finish the current phase and start the phase $i+1$ at step j.

To ensure $O(n)$ construction time, Ukkonne's algorithm uses the *suffix links* and the skip/count technique during the tree construction. A suffix link is a directed path from an internal node associated with substring $S[i, j]$ to another internal node associated with substring $S[i+1, j]$, which allows fast jump to the next extension point in the tree. The skip/count technique enables us to add the new character $S[i+1]$ at phase $i+1$ quickly. Instead of examining through each character, the traversal from the root to a specific leaf node is linear in time to the number of nodes in this path instead of the length of the substrings represented by this path. To save more space, instead of storing copies of substrings, we label edges using start and end indexes. The end index of a leaf edge is omitted and denoted by $-$. Finally, an end symbol \$ is padded at each

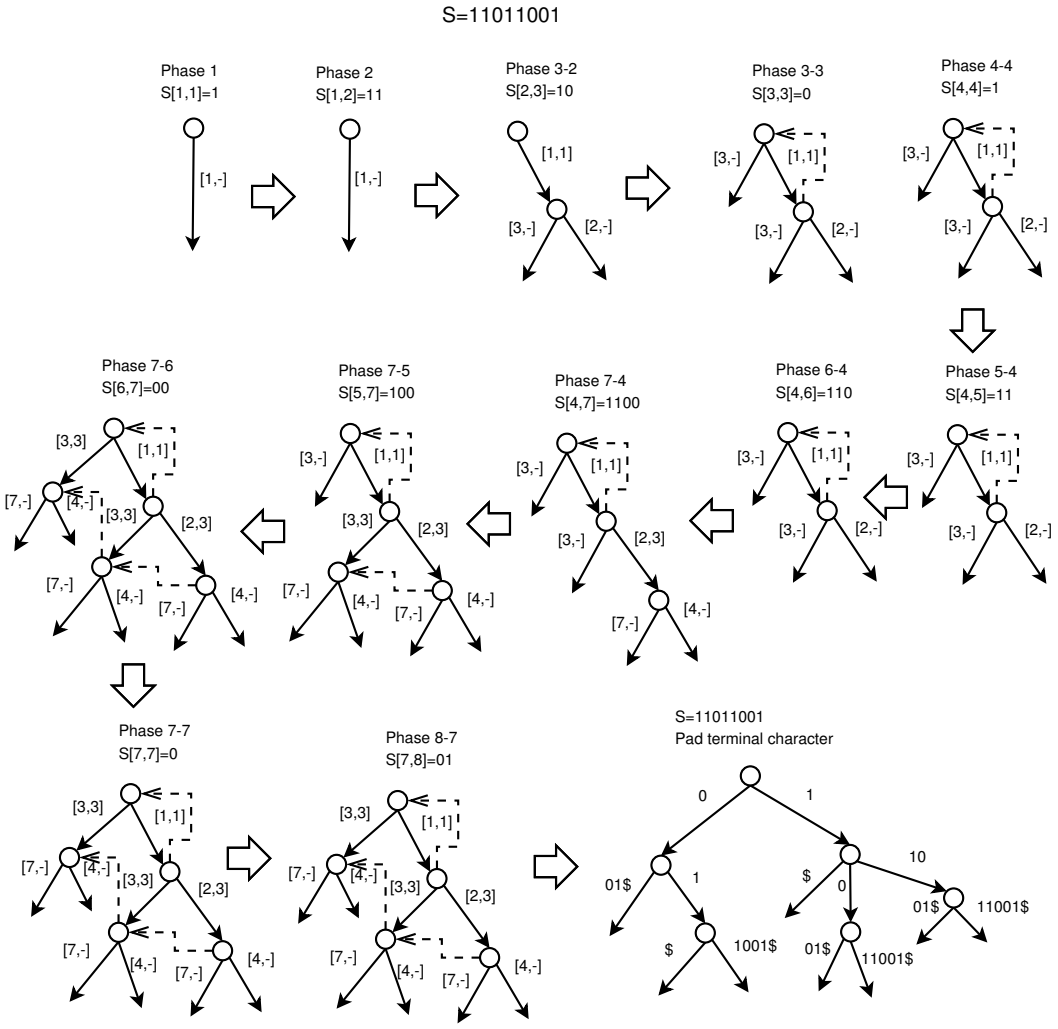


Fig. 5. The construction of the suffix tree of $S = 11011001$. Only the steps and phases that require new actions are shown.

path as a leaf node. As a result, the space used for a suffix tree is reduced to $O(n)$. Please refer to [4] for more details about the suffix tree construction.

We use the *libstree* library (`http://www.icir.org/christian/libstree/`) to implement a suffix tree.

B. Query Answering Using Suffix Trees

Given a string S, we first build its suffix tree in $O(n)$ space and $O(n)$ time using Ukkonen's algorithm [4]. We further store all leaf nodes into an array so that we can access a specific leaf node $Leaf(i)$, its edge $Edge(Leaf(i))$ and its associated string $S_{edge}(Leaf(i))$ in constant time.

Given a position p, the basic idea of using the suffix tree to get a SUS containing position p is as follows.

We can use the suffix tree to get $LSUS(p)$ in constant time, as shown in Algorithm 1. We first target at the corresponding leaf node of p in the suffix tree. Backtracking along the leaf edge to this leaf node, we meet an internal node. Base on the property of the suffix tree, the represented string from the root to this internal node is a common prefix of different suffixes. As a result, only one more character (the first character of the leaf edge, say $S[k]$) should be added and make the substring $S[p, k]$ $LSUS(p)$. If the leaf edge is \$, meaning that no more character can make the common prefix unique, the algorithm then returns null meaning no $LSUS(p)$ can be found.

Algorithm 1 The LSUS finding algorithm

Input: string $S[1, n]$, a position p, and the suffix tree T of S
Output: $LSUS(p)$

1: find the leaf node of $S[p, n]$ in T; ▷ the leaf node can be indexed during the construction of the suffix tree, so the access to the leaf node costs $O(1)$ time.
2: **if** the label of the leaf edge is \$ **then return** $null$;
3: **end if**
4: $l \leftarrow$ the length of the label of the leaf edge; ▷ the padded ternimal character is not counted into the length of the leaf edge.
5: **return** $S[p, n - l + 1]$;

With $LSUS(p)$, we can now find a SUS containing position p as shown in Algorithm 2. Let $S[i, j] = LSUS(p)$. If

Algorithm 2 The baseline SUS finding algorithm

Input: string S, a position p, and the suffix tree T of S
Output: the leftmost SUS containing position p

```
1: find LSUS(p);
2: if LSUS(p) ≠ null then
3:     let S[i, j] be the LSUS(p);
4: else
5:     let S[i, j] be S[1, n];
6: end if
7: for k ← p − 1; k > 0 and p − k ≤ j − i; k ← k − 1 do
8:     if LSUS(k) is null then continue;
9:     end if
10:    Let S[k, r] be the LSUS(k);
11:    if r < p then r ← p;
12:    end if
13:    if r − k ≤ j − i then i = r; j = k;
14:    end if
15: end for
16: return S[i, j];
```

$LSUS(p)$ does not exist, we set $LSUS(p)$ to the whole string $S[1, n]$. $S[i, j]$ is the current candidate of a SUS containing p. Then, we keep looking up $LSUS(k)$ for $k < p$ in k descending order. We stop finding $LSUS(k)$ for no later than $k = j - i$, because any $LSUS(k)$ $(k < j - i)$ containing p has a length longer than $LSUS(p)$, and thus cannot be a SUS containing p. For each $LSUS(k) = S[k, r]$, if $r < p$, i.e., it does not contain position p, we extend it to $S[k, p]$ by making $r = p$. Now, if $|S[k, r]| \leq |S[i, j]|$, it means that we find a new shorter unique substring containing position p, and we update $S[i, j] = S[k, r]$. Finally, the shortest $S[i, j]$ with the smallest i is output as the answer that is a SUS containing p. In fact, our algorithm outputs the leftmost SUS containing p.

Obviously, in the worst case, the algorithm has to test $LSUS(k)$ for every position in the string. Thus, the time complexity to obtain a SUS containing a position p is $O(n)$.

Example 4: Suppose we want to find a SUS containing position 5 in the string S shown in Figure 4. First, we find $LSUS(5) = S[5, 7]$. Then, we check $LSUS(4) = S[4, 7]$, which is longer than $S[5, 7]$, so we try $LSUS[3] = S[3, 5]$. Although $|S[3, 5]| = |S[5, 7]|$, we return $S[3, 5]$ as $S[3, 5]$ is more left. We do not have to check $LSUS(2)$ because the length of $S[2, 5]$ is already larger than $|S[3, 5]|$. Therefore, the final answer is $S[3, 5]$. ■

V. A Constant Time Online Query Answering Algorithm

In this section, we develop a method that precomputes the leftmost SUS for every position using linear space. Then, online query answering can be conducted in constant time. Let us start with some critical ideas.

A. Ideas

We first observe that a SUS must fall into one of the following three cases: a MUS, a LSUS, or a RSUS.

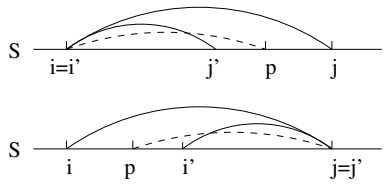


Fig. 6. SUS $S[i, j]$ and MUS $S[i', j']$ in the proof of Theorem 3.

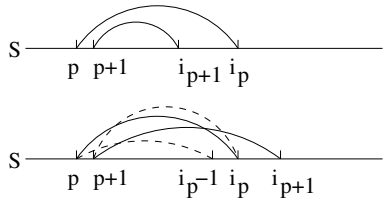


Fig. 7. $LSUS(p)$ and $LSUS(p + 1)$ in the proof of Theorem 4.

Theorem 3: Given a string S and a position p in S, if $S[i, j]$ is a SUS at position p but not a MUS, then either $i = p$ or $j = p$.

Proof: We prove by contradiction. Assume $i < p < j$. Since $S[i, j]$ is a SUS but not a MUS, according to Theorem 2, there must be a MUS $S[i', j']$ contained in $S[i, j]$ such that either $i = i'$ or $j = j'$.

We first consider the case $i = i'$ (the upper case in Figure 6). Since MUS $S[i', j']$ is a proper substring of $S[i, j]$, $S[i', j']$ cannot contain p. Otherwise, $S[i, j]$ cannot be a SUS. Thus, $j' < p$. Then, $S[i, p]$ is a superstring of MUS $S[i', j']$, and thus is unique according to Lemma 1. Since $p < j$, $S[i, p]$ is a proper substring of $S[i, j]$. This contradicts the assumption that $S[i, j]$ is a SUS at position p.

Similarly, we can show that in the case $j = j'$ (the lower case in Figure 6), $S[p, j]$ is a unique proper substring of $S[i, j]$, and is containing position p. This contradicts the assumption that $S[i, j]$ is a SUS. ■

Now, the question is how we can quickly determine whether a SUS is a MUS. Recall that in Section IV we introduce a method to find LSUS for any position in constant time using a suffix tree. We have the following interesting result.

Theorem 4 (MUS determination): Given a string S and positions p and $p + 1$ $(1 \leq p < |S|)$, let $LSUS(p) = S[p, i_p]$ and $LSUS(p + 1) = S[p + 1, i_{p+1}]$. Then, $S[p, i_p]$ is a MUS if and only if $i_p < i_{p+1}$.

Proof: (Necessity) We prove by contradiction. If $i_p \geq i_{p+1}$ (the upper case in Figure 7), then $S[p, i_p]$ is a proper superstring of $S[p + 1, i_{p+1}]$. Since $S[p + 1, i_{p+1}]$ is unique, $S[p, i_p]$ cannot be a MUS.

(Sufficiency) We prove by contradiction, too. Assume $i_p < i_{p+1}$ but $S[p, i_p]$ is not a MUS. Then, $S[p, i_p]$ must have a proper substring that is unique. There are only two possible cases (the lower case in Figure 7). In the first case, $S[p+1, i_p]$ is unique. This contradicts the assumption that $S[p + 1, i_{p+1}]$ is the LSUS at position $p + 1$ and $i_p < i_{p+1}$. In the second case, $S[p, i_p - 1]$ is unique. This contradicts that $S[p, i_p]$ is the LSUS at position p. Thus, $S[p, i_p]$ is a MUS. ■

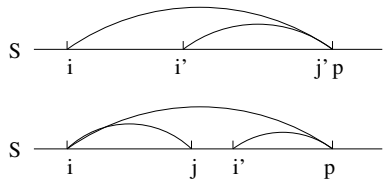


Fig. 8. $RSUS(p) = S[i,p]$ and MUS $S[i',j']$ in the proof of Theorem 5.

Theorem 4 indicates that, by one scan of S and obtain $LSUS$ at each position, we can find all MUSs and LSUSs that are not MUSs. To determine whether a SUS is a RSUS, we have the following result.

Theorem 5 (RSUS): Given a string S and a position p, a substring $S[i,p]$ is $RSUS(p)$ if and only if $S[i,p]$ contains only one MUS $S[i,j]$ $(j \leq p)$.

Proof: (Necessity) We prove by contradiction. Since $RSUS(p) = S[i,p]$, according to Theorem 2, $S[i,p]$ contains at least one MUS. If $S[i,p]$ contains two or more MUSs, then at least one of them $S[i',j']$ must satisfy $i < i'$ (the upper case in Figure 8). Then, $S[i',p]$ is a shorter unique substring than $S[i,p]$ containing p. This contradicts the assumption that $S[i,p]$ is the RSUS. Thus, $S[i,p]$ contains only one MUS.

Denote by $S[i',j']$ the only MUS contained in $S[i,p]$. According to Theorem 2, if $i \neq i'$ then $j' = p$. Then, $S[i',j']$ is a shorter unique substring containing p. A contradiction. Thus, $i = i'$.

(Sufficiency) Since $S[i,p]$ contains only one MUS $S[i,j]$ $(j \leq p)$, $S[i,p]$ is unique (the lower case in Figure 8). Moreover, every proper substring $S[i',p]$ $(i' > i)$ cannot be unique, otherwise $S[i',p]$ contains a MUS different from $S[i,j]$. Thus, $S[i,p]$ is $RSUS(p)$. ■

Last, as special cases, we have the results below following the related definitions immediately.

Corollary 1 ($LSUS(1)$ and $RSUS(n)$): Given a string S, $LSUS(1)$ is the only SUS containing position 1, and $RSUS(|S|)$ is the only SUS containing position $|S|$. ■

B. The Framework

Given a string S, our algorithm first constructs a suffix tree. This takes $O(|S|)$ time and $O(|S|)$ space. For each position p, our algorithm maintains a currently shortest MUS that contains position p, denoted by $p.cand$. It also takes $O(|S|)$ space to store the information for all positions. Moreover, we keep track of the SUS obtained at the last position. Therefore, our algorithm needs only $O(|S|)$ space overall.

Algorithm 3 shows the pesudo-code of our method.

At the beginning, we initialize $p.cand$ to null for all positions p. Our algorithm scans string S from the beginning to the end. At position 1, $LSUS(p)$ is the only SUS containing position 1 (Corollary 1).

At each position p $(p > 1)$, we compute $LSUS(p)$ using the suffix tree in constant time, as explained in Section IV. Let $S[i_{prev}, j_{prev}]$ be the SUS obtained at position $p-1$. Since a SUS must fall into one of the following three cases: a MUS, a LSUS, or a RSUS (Theorem 3). We only need to compare three candidates as follows.

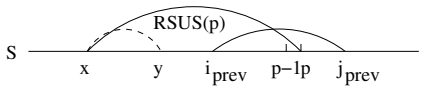


Fig. 9. $RSUS(p)$ when $p \leq j_{prev}$.

- $LSUS(p)$.
- $p.cand$ if it is not null, which records the shortest MUS containing p.
- $RSUS(p)$, which can be divided into two sub-cases.
  - $S[i_{prev}, p]$ if $p > j_{prev}$, which is $RSUS(p)$; and
  - $S[i_{prev}, j_{prev}]$ if $p \leq j_{prev}$. In this case, if $p = j_{prev}$, $S[i_{prev}, j_{prev}]$ is $RSUS(p)$ since it is unique and any substring $S[i', j_{prev}]$ is not unique (due to the fact that $S[i_{prev}, j_{prev}]$ is a SUS at position $p-1$).
    If $p > j_{prev}$, as illustrated in Figure 9, $RSUS(p)$ must contain a MUS $S[x,y]$ $(x < p)$ and be $S[x,p]$. $RSUS(p-1) = S[x,p-1]$. Since $S[x,p-1]$ is not picked as the SUS at position $p-1$, $p-1-x \geq j_{prev} - i_{prev}$. Since $p - x > p - 1 - x$, the length of $RSUS(p)$ must be longer than $S[i_{prev}, j_{prev}]$. Thus, we do not need to consider $RSUS(p)$. Instead, we should consider $S[i_{prev}, j_{prev}]$. The reason is as follows.
    As a SUS at position $p-1$, $S[i_{prev}, j_{prev}]$ is a MUS if $i_{prec} < p-1$, since $S[i_{prev}, j_{prev}]$ is neither $RSUS(p)$ nor $LSUS(p)$. If $i_{prec} = p-1$, then by comparing the length of $S[i_{prev}, j_{prev}]$ and $LSUS(p)$, $S[i_{prev}, j_{prev}]$ will be eliminated if it is longer and thus is not a MUS (Theorem 4).

We pick the shortest one as the SUS at position p. If there are more than one substring having the shortest length, we pick the leftmost one.

Before we move to the next position $p+1$, we need to update the currently shortest MUSs for some positions $p' > p$ using the MUS found at the current position. Specifically, let $S[i,j]$ be the SUS computed as above at position p. There are two types of updates.

- If $p.cand = S[x,y]$ is not chosen as the SUS at position p and $y > j$, then $p.cand$ may still be a candidate MUS for position $(j+1).cand$. We need to propagate $S[x,y]$ to position $j+1$.
- $LSUS(p)$ is not chosen as the SUS at position p, using $LSUS(p)$ and $LSUS(p+1)$, which can be extracted in constant time, we can detect whether $LSUS(p)$ is a MUS based on Theorem 4. If it is a MUS, then it should be considered as a candidate for position $j+1$, and should be propagated.

C. MUS Propagation

Although we reserve space to record the shortest MUS for each position, we do not need to explicitly store one MUS at each $p.cand$. Instead, we only need to store at those positions

Algorithm 3 The pre-computation algorithm.

Input: string S
Output: a SUS for each position p $(1 \leq p \leq |S|)$

1: build a suffix tree for string S;
2: initialize $p.cand \leftarrow null$ for $1 \leq p \leq |S|$;
3: output $LSUS(1)$, denoted by $S[1, j]$ as the SUS at position 1;
4: $i_{prev} \leftarrow 1, j_{prev} \leftarrow j$; ▷ use $LSUS(1)$ to initialize the SUS at position $p-1$
5: **for** $p = 2$ to $|S|$ **do**
6: let $S[l, r]$ be $LSUS(p)$ obtained from the suffix tree;
7: let $S[i, j]$ be the shortest substring among the following 4 strings: (1) $S[l, r]$, (2) $p.cand$ if $p.cand \neq null$, (3) $S[i_{prev}, p]$ if $j_{prev} < p$, and (4) $S[i_{prev}, j_{prev}]$ if $j_{prev} \geq p$. If there are more than one substring having the shortest length, pick the leftmost one. Output $S[i, j]$ as a SUS at position p;
8: suppose $p.cand = S[x, y]$ if $p.cand \neq null$;
9: **if** $p.cand \neq null$ and $y > j$ **then**
10: call PROPAGATE$(S[x, y], j + 1)$
11: **end if**
12: **if** $LSUS(p) = S[l, r]$ is a MUS and is not $S[i, j]$, and $r > j$ **then** call PROPAGATE$(S[l, r], j + 1)$
13: **end if**
14: $i_{prev} \leftarrow i$, $j_{prev} \leftarrow j$;
15: **end for**

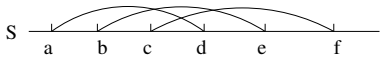


Fig. 10. MUS propagation.

Algorithm 4 The Propagation procedure.

1: **procedure** PROPAGATE(MUS $S[i, j]$, the position k to propagate)
2: **if** $k < i$ or $k > j$ **then**
3: ▷ k is not in the range of $[i, j]$
4: **return**
5: **else if** $k.cand$ is null **then**
6: $k.cand \leftarrow S[i, j]$; **return**
7: **end if**
8: suppose $k.cand = S[i', j']$;
9: **if** $j' - i' > j - i$ **then** ▷ $k.cand$ is longer than $S[i, j]$
10: $k.cand \leftarrow S[i, j]$;
11: **if** $j' > j$ **then** ▷ $S[i, j]$ ends before $k.cand$
12: call PROPAGATE$(S[i', j'], j + 1)$;
13: **end if**
14: **else if** $j' - i' < j - i$ and $j' < j$ **then**
15: ▷ $k.cand$ is shorter than $S[i, j]$ and ends before $S[i, j]$
16: call PROPAGATE$(S[i, j], j' + 1)$;
17: **else** ▷ $S[i, j]$ and $k.cand$ have the same length
18: **if** $i < i'$ **then**
19: $k.cand \leftarrow S[i, j]$
20: call PROPAGATE$(S[i', j'], j + 1)$
21: **else** ▷ $i > i'$
22: call PROPAGATE$(S[i, j], j' + 1)$
23: **end if**
24: **end if**
25: **return**
26: **end procedure**

p where the shortest MUS may not be obtained by $LSUS(p)$ and $RSUS(p)$. We use an example to explain the idea.

Example 5 (MUS propagation): Consider Figure 10, where $S[a, d]$, $S[b, e]$ and $S[c, f]$ are MUSs such that $|S[a, d]| \leq |S[b, e]| \leq |S[c, f]|$. After MUS $S[a, d]$ is detected, it is propagated from a position p to the next position $p + 1$ using the item $S[i_{prev}, j_{prev}]$ in the SUS selection step (Line 7 in Algorithm 3).

At position b, $S[b, e]$ is identified as a MUS. However, since $S[b, e]$ is longer than $S[a, d]$, it cannot be used as the candidate of SUS for positions before $S[a, d]$ ends at d. We propagate $S[b, e]$ to $(d + 1).cand$ so that at position $(d + 1)$, this MUS is not lost, as it cannot be obtained by $RSUS(d + 1)$ or $LSUS(d + 1)$.

At position c, MUS $S[c, f]$ is found. Since $S[c, f]$ is longer than $S[a, d]$, it is propagated to $(d+1).cand$, too. At this time, $(d+1).cand$ stores $S[b, e]$. Since $S[b, e]$ is shorter than $S[c, f]$, $S[c, f]$ has to be further propagated to $(e + 1).cand$. ■

Algorithm 4 gives the pseudo-code of the propagation procedure.

D. Analysis

As claimed at the beginning of Section V-B, our algorithm takes $O(n)$ space overall. On average, the space cost for each position is $O(1)$.

Except for the propagation procedure, the algorithm takes $O(1)$ time to process each position. For a position p, a MUS found at the position or $p.cand$ may be propagated and cause cascading propagations up to $h = \max_{p=1}^{|S|}\{|MUS(p)| - 1\}$ times. For instance, in Example 5, position c is contained by 3 MUSs including $S[c, f]$. As shown in the example, $S[c, f]$ is propagated twice, first to position $d+1$ and then to position $e + 1$.

Equation 1 provides an upper bound for $|MUS(p)|$. Accordingly, the time complexity of the algorithm is $O(n^2)$, the same as querying all positions using the baseline suffix tree algorithm directly. As discussed, it is very likely that the upper bound in Equation 1 can be improved. As shown in Section VI, in practice, h is often a very small number and can be empirically treated as a constant.

VI. Experiments

We conducted extensive experiments on three real data sets and a group of synthetic data sets to evaluate our methods. All experiments were run on a PC with an Intel(R) Core(TM) i7-2600 CPU 3.40GHz and 8G RAM using R and C++.

TABLE I

THE STATISTICS OF THE THREE REAL DATA SETS.

	sequence length	alphabet set	distinct alphabet count	word count
R-sequence	2,152	a-p, r-z, R, space	27	359
Mycoplasma	580,076	A, T, C, G	4	N/A
Bible	4,015,410	a-z, space	27	792,206

A. Settings

Three real data sets were used in our experiments. The statistics are listed in Table I.

The first data set is an introduction of the R language, appeared in section 2.1 of the FAQs on the R project website (`http://www.r-project.org/`). We replaced all returns, line feeds and special symbols with the space character. If there are consecutive spaces after the replacement, we just left one. All the characters are case insensitive, we only distinguished the R's that referring to the language name from the normal character "r". The resulting sequence is called *R-sequence* hereafter.

The second real data set is the genome sequence of *Mycoplasma genitalium*, the pathogenic bacterium that has one of the smallest genomes known for any free-living organism [15]. The same data set was also used by Haubold *et al.* [1] to analyze the distribution of *shustring*, which is equivalent to LSUS in this paper. We downloaded the sequence from the NCBI website (`http://www.ncbi.nlm.nih.gov/nuccore/NC_000908.2`).

The third data is the Bible (King James Version) (`http://atschool.eduweb.co.uk/sbs777/bible/text/`). We concatenated the content of 66 chapters as one long string. Then we preprocessed the sequence by replacing the special character symbols with spaces, replacing consecutive spaces with only one space, and making the sequence case-insensitive.

B. SUS Queries

For the R-sequence, we chose the positions where the language name R appeared as the query index. The corresponding SUSs are shown in Figure 11. We can see that the length of these SUSs are very short, from 2 to 3 words. This shows that using SUSs as the context to distinguish different occurrences of R's is feasible and effective.

For a given sequence, we can compute one SUS at each position. We wanted to observe the distribution of the SUS counts over different SUS lengths on the R-sequence. In addition to the original R-sequence itself, we further generated three mutations with the same string length using the same alphabet set. The first one was composed of characters that were independently and uniformly drawn from the alphabet set at each position, while the characters of the second one were sampled according to their occurrence frequency in the original R-sequence. The third one was generated by simulating the 1-lag correlation of the original article. We sampled the first character according to the empirical character distribution of the real article. Then, we sampled the next character according

R is a system for statistical computation and graphics it consists of a language plus a run time environment with graphics a debugger access to certain system functions and the ability to run programs stored in script files the design of R has been heavily influenced by two existing languages becker chambers wilks s see what is s and sussman s scheme whereas the resulting language is very similar in appearance to s the underlying implementation and semantics are derived from scheme see what are the differences between R and s for further details the core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions most of the user visible functions in R are written in R it is possible for the user to interface to procedures written in the c c or fortran languages for efficiency the R distribution contains functionality for a large number of statistical procedures among these are linear and generalized linear models nonlinear regression models time series analysis classical parametric and nonparametric tests clustering and smoothing there is also a large set of functions which provide a flexible graphical environment for creating various kinds of data presentations additional modules add on packages are available for a variety of specific purposes see R add on packages R was initially written by ross ihaka and robert gentleman at the department of statistics of the university of auckland in auckland new zealand in addition a large group of individuals has contributed to R by sending code and bug reports since mid there has been a core group the R core team who can modify the R source code archive the group currently consists of doug bates john chambers peter dalgaard seth falcon robert gentleman kurt hornik stefano iacus ross ihaka friedrich leisch uwe ligges thomas lumley martin maechler duncan murdoch paul murrell martyn plummer brian ripley deepayan sarkar duncan temple lang luke tierney and simon urbanek R has a home page at http www R project org it is free software distributed under a gnu style copyleft and an official part of the gnu project gnu s

Fig. 11. The SUSs of the queries at the positions of "R".

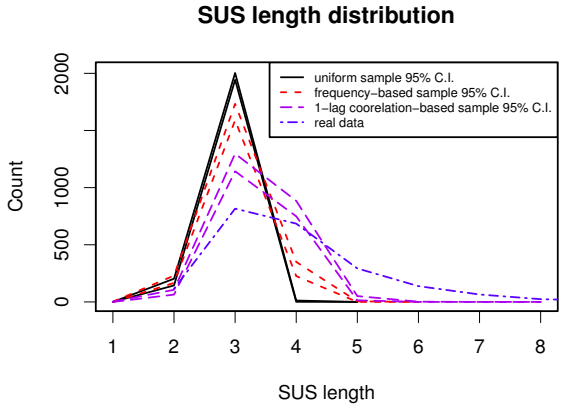


Fig. 12. The distribution of SUS length on the R introduction article.

to the empirical distribution of the character following the first one and so on. For all three types of synthetic strings, we sampled each of them $1,000$ times and plotted the 95% confidence interval of the SUS counts over the corresponding SUS lengths. The results are shown in Figure 12.

For the original R-sequence, most SUSs are of length of 3. As the length increases, the corresponding counts decreases. For the synthetic sequences, we did not see any SUS longer than 5. Importantly, the distribution of the real R-sequence does not lie in the confidence interval of the synthetic sequences. The uniform random sequence is most dissimilar

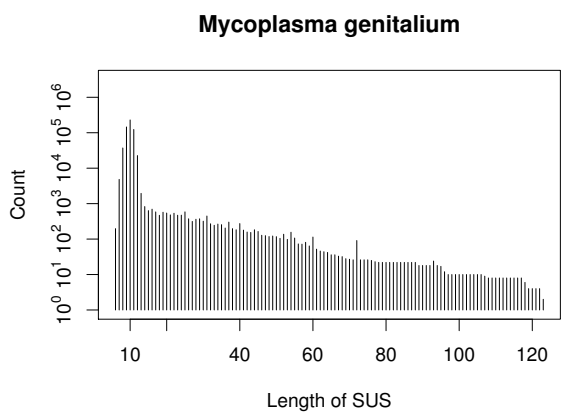


Fig. 13. The distribution of SUS length on the Mycoplasma genitalium sequence.

TABLE II
EXAMPLES OF LONG SUSS.

keyword	example of query result	chapter name
god	think not to say within yourselves we have abraham to our father for i say unto you that god	Matthew
lord	that hated me for they were too strong for me they prevented me in the day of my calamity but the lord	2 Samuel
father	kingdom for ever i will be his father	2 Samuel
israel	not let the children of israel go out	Exodus

to the real R-sequence, while the 1-lag correlation sequence, where the character distribution is closest to the original R-sequence, is most similar to it. The original R-sequence has much more relatively longer SUSs than the synthetic text sequences. This shows that the distribution of SUSs can be used to identify the real article with semantics and the random generated synthetic text.

For the genome sequence of Mycoplasma genitalium, we also analyzed the characteristics of its SUSs. We plotted the count distribution of different SUS lengths in Figure 13. Most SUSs were very short (length=10 has the highest count) and only a few of them has a length up to 123.

For the Bible data, we tested by finding the SUSs for the occurrences of some frequent keywords. As our SUS queries are a kind of point queries, we first located all the occurrences of the query keyword, and used the positions as the query positions. Then, after finding the SUSs of those query positions, we reported the complete words that each SUS has spanned, and counted the word number as the SUS length.[2]

Table II shows some long SUSs found using four different keywords, and the chapters they appear in. To further understand the SUSs of those four keywords, we show the SUS length distribution in Figure 14. The term frequencies of father, israel, god, lord are 1127, 2577, 4471, and 7965, respectively. For all four keywords, their length distributions have long tails. That is, most SUSs are of small lengths while only a small number of SUSs have long lengths. The larger the term frequency, the longer the tail. For example, the maximum length of the SUSs of the keyword "lord", which has the highest term frequency among the four, is 44, which is the largest among the four keywords. The phenomenon makes sense because if a word appears frequently, there is a high chance that we need more words to distinguish it from others at different positions.

[2]In practice, alternatively, we can treat each word as a character. Since we wanted to test the performance on long strings, we did not take this alternative.

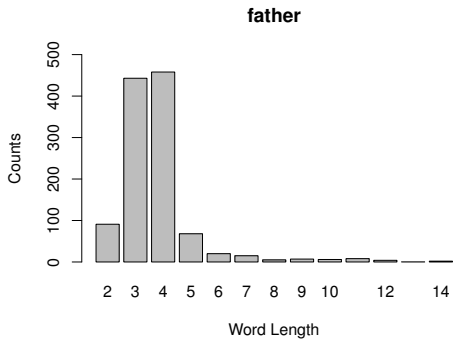
TABLE III
PROCESSING TIME ON THREE REAL DATA SETS

	Bible	Rintro	Mycoplasma
baseline offline (sec)	4.493	0.001	0.527
CTOQA offline (sec)	4.799	0.001	0.587
baseline online (sec)	< 0.001	< 0.001	< 0.001
CTOQA online (sec)	< 0.001	< 0.001	< 0.001
baseline total (sec)	5.051	0.001	0.618
CTOQA total (sec)	4.969	0.001	0.601
max h	8	4	11
avg h	2.337	1.488	5.525

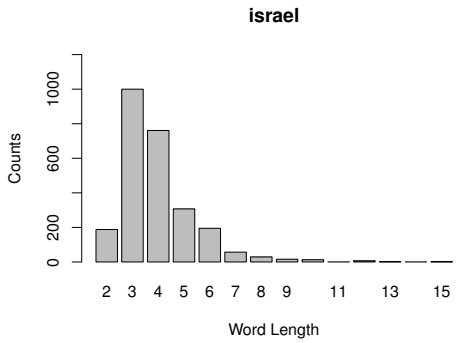
C. Scalability

In this section, we study the scalability of the baseline method and the constant time online query answering algorithm, which we denoted by CTOQA hereafter. We report three types of processing time for each method: the offiline index construction time, the average query time per query position, and the total execution time. The offline index construction time for the baseline method includes only the suffix tree construction time while that for CTOQA includes both the suffix tree construction time and the time of computing all SUSs at each position. The online query time for the baseline method includes the time to search the suffix tree, while that of CTOQA is always constant. The total processing time includes both the offline index construction time and computing SUSs for all positions in the given string.

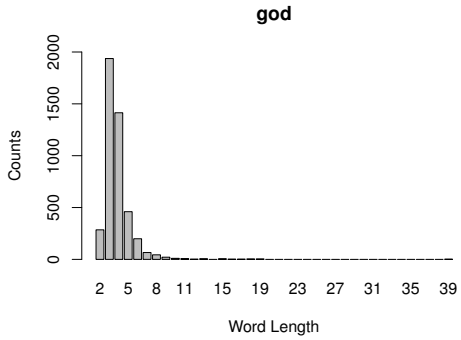
First, we report the processing time of the baseline method and the CTOQA method on the three real data sets. The results are shown in Table III. The online processing time of both methods on the three real data sets was too small to be recorded accurately. The offline processing time was proportional to the string length. We also discovered that the offline processing time of CTOQA was only slightly bigger compared to the baseline method. This shows that precomputing SUSs was very efficient in practice. We also report the maximum and average values of h, which is the overlap of MUSs at the same position, when computing SUSs on those three data sets. The values of h are very small on those data sets. Finally, the total processing time was dominated by the offline index construction time.



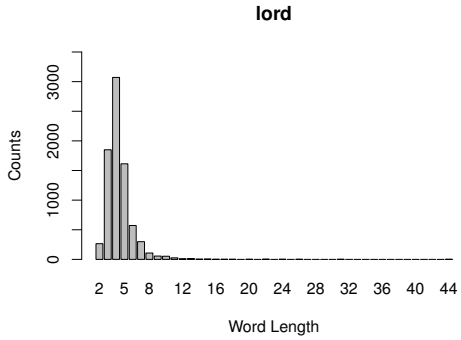
(a) Lengths of SUSs at each position of keyword "father".



(b) Lengths of SUSs at each position of keyword "israel".



(c) Lengths of SUSs at each position of keyword "god".



(d) Lengths of SUSs at each position of keyword "lord".

Fig. 14. SUSs lengths of the keywords in Bible

Next, we synthetically generated strings to show the processing time versus the string length and the alphabet size.

The results of the online query time is in Figure 15. We varied the length of the strings from $100,000$ to $500,000$ under 4 different alphabet sizes. The average query time for each SUS was shown in logarithmic scale. Compared to the baseline method, the query time of CTOQA was 10 times faster (1 order smaller as shown in the figure). As CTOQA is a constant time query method, its time cost is independent from both the string length and the alphabet size. The time cost of the baseline method increases as the string length increases. However, as the alphabet size increases, the time cost of the baseline method decreases. This was because in the baseline method the cost of finding a SUS is linear to the string length. A larger alphabet size increases the chance of shorter SUSs.

Figure 16 shows the offline index construction time of both methods when the alphabet size is 10. The processing time for both methods was linear to the string length. CTQOA costs more time due to the pre-computation of all SUSs. As the offline index construction is related with the value of h, we list the corresponding max and average values of h under different alphabet sizes and string lengths in Table IV. The h values are independent from the string length but become lower when the alphabet size increases. In general, the h value was very small compared to the total sting length n, and can be treated as a constant in practice.

Finally, the total processing time is reported in Figure 17.

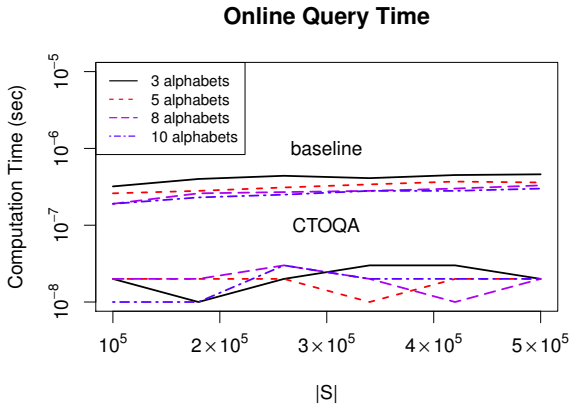


Fig. 15. Average online query time per SUS.

The two methods have very similar performance.

VII. Conclusions

In this paper, we formulated a novel type of interesting queries — shortest unique substring queries, which have many potential applications. We developed efficient algorithms and verified the effectiveness and efficiency of our approach on real data sets.

We believe that our study leads to a new direction on string queries. As future work, it is interesting to extend

TABLE IV

THE h VALUES ON THE SYNTHETIC DATA

	|S|=100000	|S|=180000	|S|=260000	|S|=340000	|S|=420000	|S|=500000
alphabet size: 3	max: 11 avg: 5.8	max: 11 avg: 6.1	max: 12 avg: 6.28	max: 12 avg: 6.42	max: 12 avg: 6.51	max: 12 avg: 6.6
alphabet size: 5	max: 8 avg: 4.8	max: 9 avg: 5.11	max: 9 avg: 5.26	max: 9 avg: 5.32	max: 9 avg: 5.36	max: 9 avg: 5.42
alphabet size: 8	max: 7 avg: 4.34	max: 7 avg: 4.34	max: 7 avg: 4.39	max: 7 avg: 4.52	max: 7 avg: 4.67	max: 8 avg: 4.82
alphabet size: 10	max: 6 avg: 3.82	max: 6 avg: 4.2	max: 6 avg: 4.45	max: 7 avg: 4.53	max: 7 avg: 4.51	max: 7 avg: 4.49

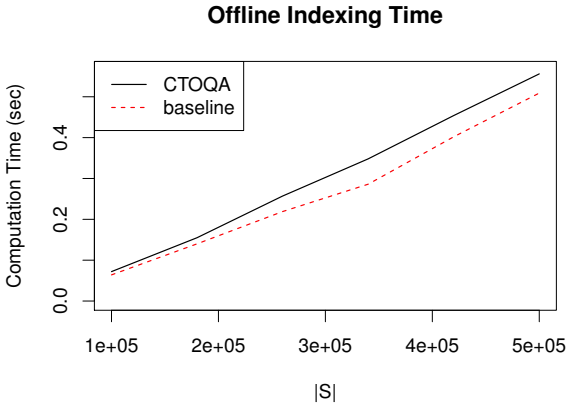


Fig. 16. Offlne index building time.

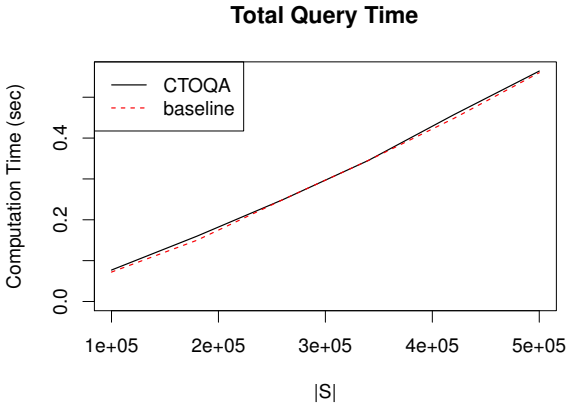


Fig. 17. Total time of computing all SUSs.

and generalize shortest unique substring queries. For example, given a string S and a position p in S, it is interesting to find a shortest substring that contains p and appears at most k times in S. Moreover, it is important to explore applications of shortest unique substring queries, such as bioinformatics applications.

REFERENCES

[1] B. Haubold, N. Pierstorff, F. Möller, and T. Wiehe, "Genome comparison without alignment using shortest unique substrings," *BMC Bioinformatics*, vol. 6, no. 123, May 2005.

[2] P. Weiner, "Linear pattern matching algorithms," in *Proc. of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, 1973, pp. 1–11.

[3] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," in *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, 1990, pp. 319–327.

[4] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, pp. 249–260, 1995.

[5] M. Farach, "Optimal suffix tree construction with large alphabets," in *Proc. of the 38th Annual Symposium on Foundations of Computer Science (FOCS'97)*, 1997, pp. 137–.

[6] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Computer Survey*, vol. 39, no. 2, July 2007.

[7] G. Nong, S. Zhang, and W. H. Chan, "Linear time suffix array construction using d-critical substrings," in *Proc. 20th Annual Symp. Combinatorial Pattern Matching*, 2009, pp. 54–67.

[8] D. Cusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[9] L. Ilie and W. F. Smyth, "Minimum unique substrings and maximum repeats," *Fundamenta Informaticae*, vol. 110, no. 1-4, pp. 183–195, 2011.

[10] K. Ye, Z. Jia, Y. Wang, P. Flicek, and R. Apweiler, "Mining unique-m substrings from genomes," *Journal of Proteomics and Bioinformatics*, vol. 3, no. 3, pp. 99–100, 2010.

[11] S. Chan, B. Kao, C. L. Yip, and M. Tang, "Mining emerging substrings," in *Proc. of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)*, 2003, pp. 119–.

[12] X. Ji, J. Bailey, and G. Dong, "Mining minimal distinguishing subsequence patterns with gap constraints," *KNOWLEDGE AND INFORMATION SYSTEMS*, vol. 11, no. 3, pp. 259–286, 2007.

[13] K. Iwanuma, R. Ishihara, Y. Takano, and H. Nabeshima, "Extracting frequent subsequences from a single long data sequence: A novel anti-monotonic measure and a simple on-line algorithm," in *IEEE International Conference on Data Mining*, 2005, pp. 186–193.

[14] H. Ahonen-Myka, "Discovery of frequent word sequences in text," in *Proceedings of the ESF Exploratory Workshop on Pattern Detection and Discovery*, 2002, pp. 180–189.

[15] C. M. Fraser, J. D. Gocayne, O. White, M. D. Adams, R. A. Clayton, R. D. Fleischmann, C. J. Bult, A. R. Kerlavage, G. Sutton, J. M. Kelley, J. L. Fritchman, J. F. Weidman, K. V. Small, M. Sandusky, J. Fuhrmann, D. Nguyen, T. R. Utterback, D. M. Saudek, C. A. Phillips, J. M. Merrick, J.-F. Tomb, B. A. Dougherty, K. F. Bott, P.-C. Hu, T. S. Lucier, S. N. Peterson, H. O. Smith, C. A. H. III, and J. C. Venter, "The minimal gene complement of mycoplasma genitalium," *Science*, vol. 270, no. 5235, 1995.