Theoretical Computer Science 562 (2015) 621–633

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

A simple yet time-optimal and linear-space algorithm for shortest unique substring queries ☆

Atalay Mert İleri [a], M. Oğuzhan Külekci [b], Bojian Xu [c,*,1]

[a] Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, MA 02139, USA
[b] Department of Biomedical Engineering, Istanbul Medipol University, Turkey
[c] Department of Computer Science, Eastern Washington University, WA 99004, USA

ARTICLE INFO

Article history:
Received 30 March 2014
Accepted 7 November 2014
Available online 13 November 2014
Communicated by G. Ausiello

Keywords:
Unique substring
Shortest unique substring
Repetitiveness
Regularity

ABSTRACT

We revisit the problem of finding shortest unique substring (SUS) proposed recently by Pei et al. (2013) [12]. We propose an optimal $O(n)$ time and space algorithm that can find an SUS for every location of a string of size n and thus significantly improve their $O(n^2)$ time complexity. Our method also supports finding all the SUSes covering every location, whereas theirs can find only one SUS for every location. Further, our solution is simpler and easier to implement and is more space efficient in practice, since we only use the inverse suffix array and the longest common prefix array of the string, while their algorithm uses the suffix tree of the string and other auxiliary data structures. Our theoretical results are validated by an empirical study with real-world data that shows our method is at least 8 times faster and uses at least 20 times less memory. The speedup gained by our method against Pei et al.'s can become even more significant when the string size increases due to their quadratic time complexity. We also have compared our method with the recent Tsuruta et al.'s (2014) [14] proposal, another independent $O(n)$ time and space algorithm for SUS finding. The empirical study shows that both methods have nearly the same processing speed. However, ours uses at least 4 times less memory for finding one SUS and at least 2 times less memory for finding all SUSes, both covering every string location.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Repetitive structure and regularity finding [2,1,13] has received much attention in stringology due to its comprehensive applications in different fields, especially in computational biology and bioinformatics research [11,10]. Finding shortest unique substrings (SUS) can be an indirect way for finding repetitive structures of a string, because any proper substring of a shortest unique substring occurs multiple times in the string and thus is a repeat [7]. Shortest unique substrings have been previously used in comparing DNA sequences [3]. However, efficient method for finding the shortest unique substring covering a given string location was not studied, until recently it was proposed by Pei et al. [12]. As pointed out in [12],

☆ Author names are listed in alphabetical order. A preliminary version of this article appeared at [5]. Part of this work was done while all authors were with TÜBİTAK-BİLGEM-UEKAE of Turkey in Summer 2013.

* Corresponding author.
E-mail addresses: atalay@mit.edu (A.M. İleri), okulekci@medipol.edu.tr (M.O. Külekci), bojianxu@ewu.edu (B. Xu).

1 Supported in part by EWU's Faculty Grants for Research and Creative Works.

http://dx.doi.org/10.1016/j.tcs.2014.11.004
0304-3975/© 2014 Elsevier B.V. All rights reserved.

SUS finding also has its own other important usage in search engines and bioinformatics. We refer readers to [12] for its detailed discussion on the applications of SUS finding. Pei et al. proposed a solution that costs $O(n^2)$ time and $O(n)$ space to find an SUS for every location of a string of size n. In this paper, we propose an optimal $O(n)$ time and space algorithm for SUS finding. Our method uses simpler data structures that include the suffix array, the inverse suffix array, and the longest common prefix array of the given string, whereas the method in [12] is built upon the suffix tree data structure. Our algorithm also provides the functionality of finding all the SUSes covering every location, whereas the method of [12] searches for only one SUS for every location. Our method not only improves their results theoretically, the empirical study also shows that our method is more space saving by a factor of at least 20 and is faster by a factor of 4. The speedup gained by our method can become even more significant when the string becomes longer due to the quadratic time cost of [12]. Due to the very high memory consumption of [12], we were not able to run their method with massive data on our machine.

Independence of our work After we posted an initial version of this proposal at arXiv [6], we were contacted via emails by the coauthors of [14] and [4], both of which solved the SUS finding using $O(n)$ time and space. By the time we communicated, article [14] had been accepted but has not been published and [4] was still under review. We were also offered with their paper drafts and the source code of [14]. The methods for SUS finding in both papers are based on the search for *minimum unique substrings* (MUS), as what [12] did. Our algorithm takes a different approach and does not need to search for MUS. The problem studied by [4] is also more general, in that they want to find SUS covering a given chunk of locations in the string, instead of a single location considered by [12,14] and our work. So, by all means, our work is independent and presents a different optimal algorithm for SUS finding. We also have included the performance comparison with the algorithm of [14] in the empirical study. It shows that both methods have nearly the same processing speed, but our method uses at least 4 times less memory for finding one SUS for every string location and uses at least 2 times less memory for finding all SUSes for every string location. The algorithm from [4] cannot be empirically studied as the author did not prefer to release the code until their paper is accepted.

2. Preliminary

We consider a **string** $S[1\ldots n]$, where each character $S[i]$ is drawn from an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. A **substring** $S[i\ldots j]$ of S represents $S[i]S[i+1]\ldots S[j]$ if $1 \le i \le j \le n$, and is an empty string if $i > j$. String $S[i'\ldots j']$ is a **proper substring** of another string $S[i\ldots j]$ if $i \le i' \le j' \le j$ and $j' - i' < j - i$. The **length** of a non-empty substring $S[i\ldots j]$, denoted as $|S[i\ldots j]|$, is $j - i + 1$. We define the length of an empty string as zero. A **prefix** of S is a substring $S[1\ldots i]$ for some i, $1 \le i \le n$. A **proper prefix** $S[1\ldots i]$ is a prefix of S where $i < n$. A **suffix** of S is a substring $S[i\ldots n]$ for some i, $1 \le i \le n$. A **proper suffix** $S[i\ldots n]$ is a suffix of S where $i > 1$. We say the character $S[i]$ occupies the string **location** i. We say the substring $S[i\ldots j]$ **covers** the kth location of S, if $i \le k \le j$. For two strings A and B, we write $\mathbf{A} = \mathbf{B}$ (and say A is **equal** to B), if $|A| = |B|$ and $A[i] = B[i]$ for $i = 1, 2, \ldots, |A|$. We say A is lexicographically smaller than B, denoted as $\mathbf{A} < \mathbf{B}$, if (1) A is a proper prefix of B, or (2) $A[1] < B[1]$, or (3) there exists an integer $k > 1$ such that $A[i] = B[i]$ for all $1 \le i \le k - 1$ but $A[k] < B[k]$. A substring $S[i\ldots j]$ of S is **unique**, if there does not exist another substring $S[i'\ldots j']$ of S, such that $S[i\ldots j] = S[i'\ldots j']$ but $i \ne i'$. A substring is a **repeat** if it is not unique.

Definition 2.1. For a particular string location $k \in \{1, 2, \ldots, n\}$, the **shortest unique substring (SUS) covering location k**, denoted as $\mathbf{SUS_k}$, is a unique substring $S[i\ldots j]$, such that (1) $i \le k \le j$, and (2) there is no other unique substring $S[i'\ldots j']$ of S, such that $i' \le k \le j'$ and $j' - i' < j - i$.

For any string location k, SUS_k must exist, because the string S itself can be SUS_k if none of the proper substrings of S is SUS_k. Also there might be multiple candidates for SUS_k. For example, if $S = \texttt{abcbb}$, then SUS_2 can be either $S[1,2] = \texttt{ab}$ or $S[2,3] = \texttt{bc}$.

For a particular string location $k \in \{1, 2, \ldots, n\}$, the **left-bounded shortest unique substring (LSUS) starting at location k**, denoted as $\mathbf{LSUS_k}$, is a unique substring $S[k\ldots j]$, such that either $k = j$ or any proper prefix of $S[k\ldots j]$ is not unique. Note that $LSUS_1 = SUS_1$ always exists, because at least the whole string S is unique. However, for an arbitrary location $k \ge 2$, $LSUS_k$ may not exist. For example, if $S = \texttt{abcabc}$, then none of $LSUS_4$, $LSUS_5$, and $LSUS_6$ exists. An **up-to-j extension of** $\mathbf{LSUS_k}$, denoted as $\mathbf{LSUS_k^j}$, is the substring $S[k\ldots j]$, where $k + |LSUS_k| \le j \le n$.

The **suffix array** $SA[1\ldots n]$ of the string S is a permutation of $\{1, 2, \ldots, n\}$, such that for any i and j, $1 \le i < j \le n$, we have $S[SA[i]\ldots n] < S[SA[j]\ldots n]$. That is, $SA[i]$ is the starting location of the ith suffix in the sorted order of all the suffixes of S. The **rank array** $Rank[1\ldots n]$ is the inverse of the suffix array. That is, $Rank[i] = j$ iff $SA[j] = i$. The **longest common prefix (lcp) array** $LCP[1\ldots n+1]$ is an array of $n+1$ integers, such that for $i = 2, 3, \ldots, n$, $LCP[i]$ is the length of the lcp of the two suffixes $S[SA[i-1]\ldots n]$ and $S[SA[i]\ldots n]$. We set $LCP[1] = LCP[n+1] = 0$. In the literature, the lcp array is often defined as an array of n integers. We include an extra zero at $LCP[n+1]$ just to simplify the description of our upcoming algorithms. Table 1 shows the suffix array and the lcp array of the example string $\texttt{mississippi}$.

The next Lemma 2.1 shows that, by using the rank array and the lcp array of the string S, it is easy to calculate any $LSUS_i$ if it exists or to detect that it does not exist.

Table 1
The suffix array and the lcp array of an example string $S = \texttt{mississippi}$.

i	$LCP[i]$	$SA[i]$	suffixes
1	0	11	i
2	1	8	ippi
3	1	5	issippi
4	4	2	ississippi
5	0	1	mississippi
6	0	10	pi
7	1	9	ppi
8	0	7	sippi
9	2	4	sissippi
10	1	6	ssippi
11	3	3	ssissippi
12	0	–	–

Lemma 2.1. *For $i = 1, 2, \ldots, n$:*

$$LSUS_i = \begin{cases} S[i \ldots i + L_i], & \text{if } i + L_i \le n \\ \text{not existing}, & \text{otherwise} \end{cases}$$

where $L_i = \max\{LCP[Rank[i]], LCP[Rank[i] + 1]\}$.

Proof. Note that by the definition of the lcp array, L_i is the length of the longest common prefix between the suffix $S[i \ldots n]$ and any other suffix of S. The value of L_i can be any number from the set $\{0, 1, \ldots, n - i + 1\}$. If $i + L_i \le n$, i.e., $L_i < n - i + 1$, it means substring $S[i \ldots i + L_i]$ exists and is unique, while substring $S[i \ldots i + L_i - 1]$ is either empty or is a repeat. So, by the definition of LSUS, $S[i \ldots i + L_i]$ is $LSUS_i$. On the other hand, if $i + L_i > n$, i.e., $L_i = n - i + 1$, it means $S[i \ldots i + L_i - 1]$ is indeed the suffix $S[i \ldots n]$ and is a repeat, so $LSUS_i$ does not exist. □

3. SUS finding for one location

In this section, we want to find the SUS covering a given location k using $O(n)$ time and space. We start with finding the leftmost one if k has multiple SUSes. In the end, we will show a trivial extension to find all the SUSes covering location k with the same time and space complexities, if k has multiple SUSes.

Lemma 3.1. *Every SUS is either an LSUS or an extension of an LSUS.*

Proof. Let's say we are looking at SUS_k for any $k \in \{1 \ldots n\}$. We know SUS_k exists for any k, so let's say $SUS_k = S[i \ldots j]$, $1 \le i \le k \le j \le n$. If $S[i \ldots j]$ is neither $LSUS_i$ nor an extension of $LSUS_i$, it means $S[i \ldots j]$ is a proper prefix of $LSUS_i$ and thus is a repeat, which contradicts the fact that $S[i \ldots j] = SUS_k$ is unique. □

Example 1: $S = \texttt{abcbca}$, then $SUS_2 = S[1, 2] = \texttt{ab}$, which is $LSUS_1$. Example 2: $S = \texttt{abcbc}$, then $SUS_2 = S[1, 2] = \texttt{ab}$, which is an extension of $LSUS_1 = S[1]$ to location 2.

By Lemma 3.1, we know SUS_k is either an LSUS or an extension of an LSUS, and the starting location of that LSUS must be on or before location k. Then the algorithm for finding SUS_k for any given string location k is simply to calculate $LSUS_1, LSUS_2, \ldots, LSUS_k$ if existing, using Lemma 2.1. During this calculation, if any LSUS does not cover the location k, we simply extend that LSUS up to location k. We will pick the shortest one among all the LSUSes or their up-to-k extensions as SUS_k. We resolve the tie by picking the leftmost one. It is possible this procedure can early stop if it finds an LSUS does not exist, because that indicates all the other remaining LSUSes do not exist either. Algorithm 1 gives the pseudocode of this procedure, where we represent SUS_k by its two attributes: `start` and `length`, the starting location and the length of SUS_k, respectively.

Lemma 3.2. *Given a string location k and the rank and the lcp array of the string S, Algorithm 1 can find SUS_k using $O(k)$ time. If there are multiple candidates for SUS_k, the leftmost one is returned.*

Proof. The procedure starts with the candidate $S[1 \ldots n]$, which is indeed unique (Line 1). Then the `For` loop calculates the $LSUS_i$ for $i = 1, 2, \ldots, k$ (Lemma 2.1). If $LSUS_i$ exists (Line 4) and the length of $LSUS_i$ or its up-to-k extension is less than the length of the current best candidate (Line 5), then we will pick that $LSUS_i$ or its up-to-k extension as the new candidate for SUS_k. This also resolves the possible ties by picking the leftmost candidate. In the end of the procedure, we will have the shortest one among $LSUS_1 \ldots LSUS_k$ or their up-to-k extensions, and that is SUS_k. Early stop is made at Line 7 if the *LSUS* being calculated does not exist, because that means all the remaining *LSUS*es to be calculated do not exist either. Each step in the `For` loop costs $O(1)$ time and the loop executes no more than k steps, so the procedure takes a total of $O(k)$ time. □

Algorithm 1: Find SUS_k. Return the leftmost one if k has multiple SUSes.

```
Input: The location index k, and the rank array and the lcp array of the string S
Output: SUS_k. The leftmost one will be returned if k has multiple SUSes.
1  start ← 1; length ← n ;              // Start location and length of the best candidate for SUS_k.
2  for i = 1,...,k do
3      L ← max{LCP[Rank[i]], LCP[Rank[i]+1]};
4      if i + L ≤ n then                                                        // LSUS_i exists.
           /* Extend LSUS_i up to k if needed. Resolve the tie by picking the leftmost SUS.  */
5          if max{L+1, k−i+1} < length then
6              start ← i; length ← max{L+1, k−i+1};
7      else break;                                                                // Early stop.
8  Print SUS_k ← (start, length);
```

Algorithm 2: Find all the SUSes covering a given location k.

```
Input: The location index k, and the rank array and the lcp array of the string S
Output: All the SUSes covering location k.
1  start ← 1; length ← n ;              // Start location and length of the best candidate for SUS_k.
   /* Find the length of SUS_k.                                                            */
2  for i = 1,...,k do
3      L ← max{LCP[Rank[i]], LCP[Rank[i]+1]};
4      if i + L ≤ n then                                                        // LSUS_i exists.
5          if max{L+1, k−i+1} < length then          // Extend LSUS_i to location k if necessary.
6              start ← i; length ← max{L+1, k−i+1};
7      else break;                                                                // Early stop.
   /* Find all SUSes covering location k.                                                  */
8  for i = 1,...,k do
9      L ← max{LCP[Rank[i]], LCP[Rank[i]+1]};
10     if i + L ≤ n then                                                        // LSUS_i exists.
11         if max{L+1, k−i+1} = length then          // Extend LSUS_i to location k if necessary.
12             Print (i, max{L+1, k−i+1});
13     else break;                                                                // Early stop.
```

Theorem 3.1. *For any location k in the string S, we can find SUS_k using $O(n)$ time and space. If there are multiple candidates for SUS_k, the leftmost one is returned.*

Proof. The suffix array of S can be constructed by existing algorithms using $O(n)$ time and space (for example, [9]). After the suffix array is constructed, the rank array (the inverse suffix array) can be trivially created using another $O(n)$ time and space. We can then use the suffix array and the rank array to construct the lcp array using another $O(n)$ time and space [8]. Combining the time cost of Algorithm 1 (Lemma 3.2), the total time cost for finding SUS_k for any location k in the string S of size n is $O(n)$ with a total of $O(n)$ space usage. If multiple candidates for SUS_k exist, the leftmost candidate will be returned as is provided by Algorithm 1 (Lemma 3.2). □

3.1. Extension: finding all SUSes for one location

It is trivial to extend Algorithm 1 to find all the SUSes covering a particular location k as follows. We can first use Algorithm 1 to find the leftmost SUS_k. Then we start over again to re-calculate $LSUS_1 \dots LSUS_k$ or their up-to-k extensions, and return all of those whose length is equal to the length of SUS_k. Algorithm 2 shows the pseudocode. This procedure clearly costs an extra $O(k)$ time. Combining the results from Theorem 3.1, we get the following theorem.

Theorem 3.2. *For any location k in the string S, we can find all the SUSes covering location k using $O(n)$ time and space.*

4. SUS finding for every location

In this section, we want to find SUS_k for every location $k = 1, 2, \dots, n$. If k has multiple SUSes, the leftmost one will be returned. In the end, we will show an extension to find all SUSes for every location.

A natural solution is to iteratively use Algorithm 1 as a subroutine to find every SUS_k, for $k = 1, 2, \dots, n$. However, the total time cost of this solution will be $O(n) + \sum_{k=1}^{n} O(k) = O(n^2)$, where $O(n)$ captures the time cost for the construction of the rank array and the lcp array and $\sum_{k=1}^{n} O(k)$ is the total time cost for the n instances of Algorithm 1. We want to have a solution that costs a total of $O(n)$ time and space, which implies that the amortized cost for finding each SUS is $O(1)$.

By Lemma 3.1, we know that every SUS must be an LSUS or an extension of an LSUS. The next Lemma 4.1 further says if SUS_k is an extension of an LSUS, it has some special properties and can be quickly obtained from SUS_{k-1}.

Lemma 4.1. *For any $k \in \{2, 3, \ldots, n\}$, if SUS_k is an extension of an LSUS, then (1) SUS_{k-1} must be a substring whose right boundary is the character $S[k-1]$, and (2) SUS_k is the substring SUS_{k-1} appended by the character $S[k]$.*

Proof. Because SUS_k is an extension of an LSUS, we have $SUS_k = S[i \ldots k]$ for some $i < k$ and $LSUS_i = S[i \ldots j]$ for some $j < k$. We also know $S[i \ldots k-1]$ is unique, because the unique substring $S[i \ldots j]$ is a prefix of $S[i \ldots k-1]$. Note that any substring starting from a location before i and covering location $k-1$ is longer than the unique substring $S[i \ldots k-1]$, so SUS_{k-1} must be starting from a location between i and $k-1$, inclusive. Next, we show SUS_{k-1} actually must start at location i. The fact $SUS_k = S[i \ldots k]$ tells us that $|LSUS_t| \geq |SUS_k| = k - i + 1$ for every $t = i+1, i+2, \ldots, k$; otherwise, any $LSUS_t$ that is shorter than $k - i + 1$ would be a better candidate than $S[i \ldots k]$ as SUS_k. That means, any unique substring starting from $t = i+1, i+2, \ldots, k-1$ has a length at least $k-i+1$. However, $|S[i \ldots k-1]| = k-i < k-i+1$ and $S[i \ldots k-1]$ is unique already and covers location $k-1$ as well, so $S[i \ldots k-1]$ is the only candidate for SUS_{k-1}. This also means SUS_k is indeed the substring SUS_{k-1} appended by $S[k]$. □

4.1. The overall strategy

We are ready to present the overall strategy for finding SUS of every location, by using Lemmas 3.1 and 4.1. We will calculate all the SUS in the order of $SUS_1, SUS_2, \ldots, SUS_n$. That means when we want to calculate SUS_k, $k \geq 2$, we have had SUS_{k-1} calculated already. Note that $SUS_1 = LSUS_1$, which is easy to calculate using Lemma 2.1. Now let's look at the calculation of a particular SUS_k, $k \geq 2$. By Lemma 3.1, we know SUS_k is either an LSUS or an extension of an LSUS. By Lemma 4.1, we also know if SUS_k is an extension of an LSUS, then the right boundary of SUS_{k-1} must be $S[k-1]$ and SUS_k is just SUS_{k-1} appended by the character $S[k]$. Suppose when we want to calculate SUS_k, we have already calculated the shortest LSUS covering location k or have known the fact that no LSUS covers location k. Then, by using SUS_{k-1}, which has been calculated by then, and the shortest LSUS covering location k, we will be able to calculate SUS_k as follows:

Case 1: If the right boundary of SUS_{k-1} is not $S[k-1]$, then we know SUS_k cannot be an extension of an LSUS (the contrapositive of Lemma 4.1). Thus, SUS_k is just the shortest LSUS covering location k, which must be existing in this case.

Case 2: If the right boundary of SUS_{k-1} is $S[k-1]$, then SUS_k may or may not be an extension of an LSUS. We will consider two possibilities: (1) If the shortest LSUS covering location k exists, we will compare its length with $|SUS_{k-1}| + 1$, and pick the shorter one as SUS_k. If both have the same length, we resolve the tie by picking the one whose starting location index is smaller. (2) If no LSUS covers location k, SUS_k will just be SUS_{k-1} appended by $S[k]$.

Therefore, the real challenge here, by the time we want to calculate SUS_k, $k \geq 2$, is to ensure that we would have already calculated the shortest LSUS covering location k or we would have already known the fact that no LSUS covers location k. If there exist multiple shortest LSUSes covering location k, we would like to know the leftmost one.

4.2. Preparation

We now focus on the calculation of the *leftmost* shortest LSUS covering every string location k, denoted by $\mathbf{SLS_k}$. Let $\mathbf{Candidate_i^k}$ denote the *leftmost* shortest one among those of $LSUS_1, \ldots, LSUS_k$ that exist and cover location i. For an arbitrary k, $1 \leq k \leq n$, SLS_k may not exist, because the location k may not be covered by any LSUS at all. For example, if $S = \texttt{abcabc}$, then locations 5 and 6 are not covered by any LSUS, and thus SLS_5 and SLS_6 do not exist. However, if SLS_k exists, by the definition of *SLS* and *Candidate*, we have the following fact.

Fact 4.1. *$SLS_k = Candidate_k^k = Candidate_k^{k+1} = \cdots = Candidate_k^n$, if SLS_k exists.*

Our goal is to ensure SLS_k will have been known when we want to calculate SUS_k, so we calculate every SLS_k following the same order $k = 1, 2, \ldots, n$, at which we calculate all SUSes. Because we need to know every $LSUS_i$, $i \leq k$ in order to calculate SLS_k (Fact 4.1), we will walk through the string locations $k = 1, 2, \ldots, n$: at each walk step k, we calculate $LSUS_k$ and maintain $Candidate_i^k$ for every string location i that has been covered by at least one of $LSUS_1, LSUS_2, \ldots, LSUS_k$. Note that $Candidate_i^k = SLS_i$ for every $i \leq k$ (Fact 4.1). Those $Candidate_i^k$ with $i \leq k$ would have already been used as SLS_i in the calculation of SUS_i. So, after each walk step k, we will only need to maintain the candidates for locations after k.

Lemma 4.2. *(1) $LSUS_1$ always exists. (2) If $LSUS_k$ exists, then $LSUS_1, LSUS_2, \ldots, LSUS_k$ all exist. (3) If $LSUS_k$ does not exist, then none of $LSUS_k, LSUS_{k+1}, \ldots, LSUS_n$ exist.*

Proof. (1) $LSUS_1$ must exist, because the string S can be $LSUS_1$ if every proper prefix of S is a repeat. (2) If $LSUS_k$ exists, say $LSUS_k = S[k \ldots \gamma_k]$, then $LSUS_i$ exists for every $i \leq k$, because at least $S[i \ldots \gamma_k]$ is unique due to the fact that $S[k \ldots \gamma_k]$ is unique and also is a suffix of $S[i \ldots \gamma_k]$. (3) If $LSUS_k$ does not exist, it means $S[k \ldots n]$ is a repeat, and thus every suffix $S[i \ldots n]$ of $S[k \ldots n]$ for $i \leq k$ is also a repeat, i.e., $LSUS_i$ does not exist for every $i \geq k$. □

The next lemma shows that the right boundary of $LSUS_i$ will be on or after the right boundary of $LSUS_{i-1}$, if $LSUS_i$ exists.

Lemma 4.3. *For each $i = 2, 3, \ldots, n$: $|LSUS_i| \geq |LSUS_{i-1}| - 1$.*

Proof. We prove the lemma by contradiction. Suppose $LSUS_{i-1} = S[i-1 \ldots j]$ for some j, $i-1 \leq j \leq n$. If $|LSUS_i| < |LSUS_{i-1}| - 1$, it means $LSUS_i = S[i \ldots k]$, where $i \leq k < j$. Because $S[i \ldots k]$ is unique, $S[i-1 \ldots k]$ is also unique, whose length however is shorter than $S[i-1 \ldots j]$. This is a contradiction because $S[i-1 \ldots j]$ is already $LSUS_{i-1}$. Thus, the claim in the lemma is true. □

Now let's look at the situation at the end of the kth walk step. By then, we have calculated $LSUS_1, LSUS_2, \ldots, LSUS_k$. By Lemma 4.2, we know that there exists some ℓ_k, $1 \leq \ell_k \leq k$, such that $LSUS_1, \ldots, LSUS_{\ell_k}$ all exist, but $LSUS_{\ell_k+1} \ldots LSUS_k$ do not exist. If $\ell_k = k$, that means $LSUS_1, \ldots, LSUS_k$ all exist. Let γ_k denote the right boundary of $LSUS_{\ell_k}$, i.e., $LSUS_{\ell_k} = S[\ell_k \ldots \gamma_k]$. By Lemma 4.3, we know γ_k is also the right boundary of the string locations covered by $LSUS_1, \ldots, LSUS_{\ell_k}$. So, every location $1, 2, \ldots, \gamma_k$ is covered by at least one LSUS from $LSUS_1, \ldots, LSUS_{\ell_k}$. That is, at the end of the kth walk step: (1) every location $j = 1, \ldots, \gamma_k$ has its candidate $Candidate_j^k$ calculated already. (2) If $\gamma_k < n$, every location $j = \gamma_k + 1, \ldots, n$ still does not have its candidate calculated, because every such location j has not been covered by any LSUS from $LSUS_1, \ldots, LSUS_{\ell_k}$ that we have calculated at the end of the kth walk step.

Lemma 4.4. *At the end of the kth walk step, if $\gamma_k > k$, then for any i and j, $k \leq i < j \leq \gamma_k$, $Candidate_j^k$ also covers location i.*

Proof. $Candidate_j^k$ is a substring starting somewhere on or before k and going through the location j. Because $k \leq i < j$, it is obvious that $Candidate_j^k$ goes through location i. □

Lemma 4.5. *At the end of the kth walk step, if $\gamma_k > k$, then*

$$\left|Candidate_k^k\right| \leq \left|Candidate_{k+1}^k\right| \leq \cdots \leq \left|Candidate_{\gamma_k}^k\right|$$

Proof. By Lemma 4.4, we know $Candidate_j^k$ also covers location i, for any i and j, $k \leq i < j \leq \gamma_k$. Thus, if $|Candidate_j^k| < |Candidate_i^k|$, location i's current candidate should be replaced by location j's candidate, because that gives location i a shorter candidate. However, the current candidate for location i is already the shortest candidate. It is a contradiction. So, $|Candidate_i^k| \leq |Candidate_j^k|$, which proves the lemma. □

4.3. Finding SLS for every location

Invariant. We calculate SLS_k for $k = 1, 2, \ldots, n$ by maintaining the following invariant at the end of every walk step k: (A) If $\gamma_k > k$, locations $\{k+1, k+2, \ldots, \gamma_k\}$ will be cut into chunks, such that: (A.1) All locations in one chunk have the same candidate. (A.2) Locations belonging to different chunks have different candidates. (A.3) Each chunk will be represented by a linked list node of four fields: `ChunkStart, ChunkEnd, start, length`, respectively representing the start and end location of the chunk and the start and length of the candidate shared by all locations of the chunk. (A.4) All nodes representing different chunks will be connected into a linked list, ordered by the string positions of the corresponding chunks. The linked list has a `head` and a `tail`, referring to the two nodes that represent the lowest positioned chunk and the highest positioned chunk. (B) If $\gamma_k \leq k$, the linked list is empty.

Maintenance of the invariant. We describe in an inductive manner the procedure that maintains the invariant. Algorithm 3 shows the pseudocode of the procedure. We start with an empty linked list.

Base step: $k = 1$ We are walking the first step. We first calculate $LSUS_1$ using Lemma 2.1. We know $LSUS_1$ must exist. Let's say $LSUS_1 = S[1 \ldots \gamma_1]$ for some $\gamma_1 \leq n$. Then, $Candidate_i^1 = LSUS_1$ for every $i = 1, 2, \ldots, \gamma_1$. We record all these candidates by using a single node $(1, \gamma_1, 1, \gamma_1)$. This is the only node in the linked list and is pointed by both `head` and `tail`. We know $SLS_1 = Candidate_1^1$ (Fact 4.1), so we return SLS_1 by returning $(\texttt{head.start}, \texttt{head.length}) = (1, \gamma_1)$. We then change `head.ChunkStart` from 1 to 2. If it turns out $\texttt{head.ChunkEnd} = \gamma_1 < 2$, meaning $LSUS_1$ really covers location 1 only, we delete the `head` node from the linked list, which will then become empty.

Inductive step: $k \geq 2$ We are walking the kth step. We first calculate $LSUS_k$ using Lemma 2.1.

- Case 1: $LSUS_k$ does not exist. (1) If `head` does not exist. It means that location k is covered neither by any of $LSUS_1, \ldots, LSUS_{k-1}$ nor by $LSUS_k$, so SLS_k simply does not exist and we will return $(\texttt{null}, \texttt{null})$. (2) If `head` exists, we will return $(\texttt{head.start}, \texttt{head.length})$ as SLS_k, because $Candidate_k^k = SLS_k$ (Fact 4.1). Then we will

Algorithm 3: The sequence of function calls *FindSLS(1)*, *FindSLS(2)*, ..., *FindSLS(n)* returns SLS_1, $SLS_2, \dots, SLS_n$, if the corresponding *SLS* exists; otherwise, `null` will be returned.

```
1  Construct Rank[1...n] and LCP[1...n] of the string S;
2  Initialize an empty List;          // Each node's 4 fields: ChunkStart, ChunkEnd, start, length.
3  head ← 0; tail ← 0 ;                         // Reference to the head and tail node of the List

4  FindSLS(k)
     /* Process LSUS_k, if it exists.                                                        */
5    L ← max{LCP[Rank[k]], LCP[Rank[k] + 1]};
6    if k + L ≤ n then                                                        // LSUS_k exists.
       // Add a new list element at the tail, if necessary.
7      if head = 0 then  List[1] ← (k, k + L, k, L + 1); head ← 1; tail ← 1 ;  // List was empty.
8      else if k + L > List[tail].ChunkEnd then
9        tail++; List[tail] ← (List[tail − 1].ChunkEnd + 1, k + L, k, L + 1);

       /* Update candidates and merge the nodes whose candidates can be shorter. Resolve the tie by
          picking the leftmost one.                                                           */
10     j ← tail;
11     while j ≥ head and List[j].length > L + 1 do j−−;
12     List[j + 1] ← (List[j + 1].ChunkStart, List[tail].ChunkEnd, k, L + 1); tail ← j + 1;
13   if head ≠ 0 then  SLS_k ← (head.start, head.length) ;              // The list is not empty.
14   else  SLS_k ← (null, null) ;                                       // SLS_k does not exist.

     /* Discard the information about location k from the List.                              */
15   if head > 0 then                                                       // List is not empty
16     if List[head].ChunkEnd ≤ k then
17       head++;                                                  // Delete the current head node
18       if head > tail then  head ← 0; tail ← 0;                           // List becomes empty
19     else List[head].ChunkStart ← k + 1;
20   return SLS_k
```

remove the information about location k from the head by setting $head.ChunkStart = k + 1$. If it turns out that $\texttt{head.ChunkEnd} < \texttt{head.ChunkStart}$, we will remove the `head` node.

- Case 2: $LSUS_k$ exists and $LSUS_k = S[k \dots \gamma_k]$, $\gamma_k \le n$. By Lemma 4.2, we know $LSUS_1$, $LSUS_2, \dots, LSUS_{k-1}$ all exist. Let γ_{k-1} denote the right boundary of $LSUS_1, LSUS_2, \dots, LSUS_{k-1}$. By Lemma 4.3, we know $\gamma_k \ge \gamma_{k-1}$ and γ_{k-1} is also the right boundary of $LSUS_{k-1}$, i.e., $LSUS_{k-1} = S[k-1 \dots \gamma_{k-1}]$. Note that both $\gamma_{k-1} < k$ and $\gamma_{k-1} \ge k$ are possible.
  1. If `head` does not exist, it means $\gamma_{k-1} < k$ and none of the locations $\{k \dots \gamma_k\}$ is covered by any of $LSUS_1, LSUS_2, \dots,$ $LSUS_{k-1}$. We will insert a new node $(\mathtt{k}, \gamma_\mathtt{k}, \mathtt{k}, \gamma_\mathtt{k} - \mathtt{k} + 1)$, which will be the only node in the linked list.
  2. If `head` exists, it means $\gamma_{k-1} \ge k$. If $\gamma_\mathtt{k} > \texttt{tail.ChunkEnd} = \gamma_{\mathtt{k}-1}$, we first insert at the tail side a new linked list node $(\texttt{tail.ChunkEnd} + 1, \gamma_\mathtt{k}, \mathtt{k}, \gamma_\mathtt{k} - \mathtt{k} + 1)$ to record the candidate information for locations in the chunk after γ_{k-1} through γ_k.

  Then, we will travel through the nodes in the linked list from the tail side toward the head. We stop when we meet a node whose candidate is shorter than or equal to $LSUS_k$ or when we reach the head end of the linked list. We will merge all the nodes whose candidates are longer than $LSUS_k$ into a new linked list node. The chunk covered by the new node is the union of the chunks covered by the merged nodes, and the candidate of the new node is $LSUS_k$.

  This travel and merge process is valid because of Lemma 4.5. This merge process ensures every location maintains its best (shortest) candidate by the end of every walk step. It also resolves the possible ties of multiple shortest LSUSes covering a particular location by picking the leftmost one as that location's candidate, because the merge process does not merge nodes whose candidates are of the same length.

  We will return (`head.start`, `head.length`) as SLS_k, since $Candidate_k^k = SLS_k$ (Fact 4.1). Finally, we will remove the information about location k from the head by setting $head.ChunkStart = k + 1$. We will remove the `head` node if it turns out that $\texttt{head.ChunkEnd} > \texttt{head.ChunkStart}$.

Lemma 4.6. *Given the lcp array and the rank array of S, the sequence of FindSLS(1), FindSLS(2), ..., FindSLS(n) function calls will return $SLS_1, SLS_2, \dots, SLS_n$ if existing. The amortized time cost of one FindSLS() function call is $O(1)$.*

Proof. The correctness of Algorithm 3 is already given in the description of the procedure that maintains the invariance. All operations in an instance of *FindSLS*() function call clearly take $O(1)$ time, except the `while` loop at Line 11, which is to merge linked list nodes whose candidates can be shorter. Thus, the lemma will be proved, if we can prove the amortized number of linked nodes that will be merged via that `while` loop is also bounded by a constant. Note that no node in the linked list ever splits due to Lemma 4.3. In the sequence of function calls *FindSLS*(1), *FindSLS*(2), ..., *FindSLS*(n), there are at

Algorithm 4: Finding the leftmost SUS_k, $k = 1, \ldots, n$.

```
1 for k ← 1...n do
2     (start, length) ← FindSLS(k) ;          // SLS_k; It is (null,null) if SLS_k does not exist.
3     if k = 1 then  Print SUS_k ← (start, length);
4     else if SUS_{k-1}.start + SUS_{k-1}.length − 1 > k − 1 then  Print SUS_k ← (start, length);
5     else if (start, length) = (null, null) then  Print SUS_k ← (SUS_{k-1}.start, SUS_{k-1}.length + 1);
6     else if length < SUS_{k-1}.length + 1 then  Print SUS_k ← (start, length);
7     else                                     // Resolve the tie by picking the leftmost one.
8         Print SUS_k ← (SUS_{k-1}.start, SUS_{k-1}.length + 1)
9
```

most n linked list nodes to be merged. We know the number of merge operations in merging n nodes into one node (in the worst case) is no more than $O(n)$. Therefore, the amortized time cost on merging the linked list nodes in one *FindSLS*() function call over the sequence of n function calls *FindSLS*(1), *FindSLS*(2), ..., *FindSLS*(n) is $O(1)$. This finishes the proof of the lemma. □

4.4. Finding the leftmost SUS for every location

Once we are able to sequentially calculate every SLS_k or detect it does not exist, we are ready to calculate every SUS_k by using the strategy described in Section 4.1. Algorithm 4 gives the pseudocode of the procedure. It calculates SUSes in the order of $SUS_1, SUS_2, \ldots, SUS_n$ (Line 1). For each location k, the function call at Line 2 is to calculate SLS_k or to find SLS_k does not exist. Line 3 handles the special case where $SUS_1 = LSUS_1 = SLS_1$. The condition at Line 4 shows that SUS_i cannot be an extension of an LSUS (Lemma 4.1), so $SUS_k = SLS_k$, which must be existing in this case. Line 5 handles the case where SLS_k does not exist, so SUS_k must be SUS_{k-1} appended by $S[k]$. Line 6 handles the case where SLS_k is shorter than the one-character extension of SUS_{k-1}, so SUS_k is SLS_k. Lines 7–8 handle the case where SLS_k is longer than or equal to the one-character extension of SUS_{k-1}, so SUS_k is SUS_{k-1} appended by $S[k]$. This also resolves the tie by picking the leftmost one if k is covered by multiple SUSes.

Theorem 4.1. *Algorithm 4 finds $SUS_1, SUS_2, \ldots, SUS_n$ of string S using a total of $O(n)$ time and space. If any string location is covered by multiple SUS, Algorithm 4 finds the leftmost one.*

Proof. We can construct the suffix array of the string S in a total of $O(n)$ time and space using existing algorithms (for example, [9]). The rank array is just the inverse suffix array and can be directly obtained from SA using $O(n)$ time and space. Then we can obtain the lcp array from the suffix array and rank array using another $O(n)$ time and space [8]. So the total time and space costs for preparing these auxiliary data structures are $O(n)$.

Time cost. The amortized time cost for each *FindSLS* function call at Line 2 in the sequence of function calls *FindSLS*(1), ..., *FindSLS*(n) is $O(1)$ (Lemma 4.6). The time cost for Lines 3–8 is also $O(1)$. There are a total of n steps in the `For` loop, yielding a total of $O(n)$ time cost.

Space usage. The only space usage (in addition to the auxiliary data structures such as suffix array, rank array, and the lcp array, which cost a total of $O(n)$ space) in our algorithm is the dynamic linked list, which however has no more than n nodes at any time. Each node costs $O(1)$ space. Therefore, the linked list costs $O(n)$ space. Adding the space usage of the auxiliary data structures, we get the total space usage of finding every SUS is $O(n)$.

Finding the leftmost SUS. For any particular location k, if one SUS covering location k is an extension of an LSUS, we know by Lemma 4.1, that SUS must be the substring SUS_{k-1} appended by the letter $S[k]$. Clearly this SUS is the leftmost one among all the SUSes covering location k and is guaranteed to be returned by Lines 7–8 in Algorithm 4. If all SUSes covering location k are LSUSes, the leftmost one of those LSUSes is already guaranteed to be returned by Algorithm 3 (Lemma 4.6). □

4.5. Extension: finding all SUSes for every location

It is possible that a particular location can have multiple SUSes. For example, if $S = \texttt{abcbb}$, then SUS_2 can be either $S[1,2] = \texttt{ab}$ or $S[2,3] = \texttt{bc}$. Algorithm 4 only returns one of them and resolve the tie by picking the leftmost one. However, it is easy to modify Algorithm 4 to return all the SUSes of every location, without changing Algorithm 3.

Suppose a particular location k is covered by multiple SUSes. We know, at the end of the kth walk step but before the linked list update (at the end of Line 14 in Algorithm 3), SLS_k returned by Algorithm 3 is recorded by the `head` node and is the leftmost one among all the SUSes that are LSUS and cover location k. Because every string location maintains its shortest candidate and due to Lemma 4.5, all the other SUSes that are LSUS and cover location k are being recorded by other linked list nodes that are immediately following the `head` node. This is because if those other SUSes are not being recorded, that means the location right after the head node's chunk has a candidate longer than SUS_k or does not have a

Algorithm 5: Finding all choices of each SUS_k, for $k = 1, \ldots, n$.

```
1  for k ← 1...n do
2      flag ← 0; (start, length) ← FindSLS(k) ;          // SLS_k; (null,null) if SLS_k does not exist.
3      if k = 1 then
4          Print SUS_k ← (start, length);
5      else if SUS_{k−1}.start + SUS_{k−1}.length − 1 > k − 1 then
6          Print SUS_k ← (start, length); flag ← 1;
7      else if (start, length) = (null, null) then
8          Print SUS_k ← (SUS_{k−1}.start, SUS_{k−1}.length + 1);
9      else if length ≤ SUS_{k−1}.length + 1 then
10         Print SUS_k ← (start, length); flag ← 1;
11     else
12         Print SUS_k ← (SUS_{k−1}.start, SUS_{k−1}.length + 1);
       /* Print out other SUSes that cover location k.                                        */
13     if flag = 1 then
14         if SUS_{k−1}.length + 1 = SUS_k.length then
15             Print (SUS_{k−1}.start, SUS_{k−1}.length + 1);
16         j ← head;
17         while j > 0 and j ≤ tail do
               /* List[j].start ≠ SUS_k.start condition checking is because the SUS from head node may have been printed.
                  */
18             if List[j].length = SUS_k.length and List[j].start ≠ SUS_k.start then
19                 Print (List[j].start, List[j].length); j ← j + 1;
20             else if List[j].start ≠ SUS_k.start then
21                 Break;
```

candidate calculated yet, but that location is indeed covered by an SUS_k at the end of the kth walk step. It's a contradiction. Same argument can be made to the other next neighboring locations that are covered by SUS_k.

Therefore, finding all the SUSes covering location k becomes easy—simply go through the linked list nodes from the `head` node toward the `tail` node and report all the candidates whose lengths are equal to the length of SUS_k that we have found. If the rightmost character of SUS_{k-1} is $S[k-1]$ and the substring SUS_{k-1} appended by $S[k]$ has the same length, that substring will be reported too. Algorithm 5 gives the pseudocode, where the `flag` is used to note in what cases it is possible to have multiple SUSes.

If `flag` is on, we will need to check the linked list nodes (Lines 17–21) as well as the one letter extension of SUS_{k-1} (Lines 14–15). The overall time and space cost of maintaining the linked list data structure (the sequence of function calls $FindSLS(1), FindSLS(2), \ldots, FindSLS(n)$) is still $O(n)$. The time cost of reporting the SUSes covering a particular location becomes $O(occ)$, where occ is the number of SUSes that cover that location. That gives us the following theorem.

Theorem 4.2. *Algorithm 5 finds all SUSes covering every location of a string of size n using $O(n)$ space and $O(N)$ time, where $N = \sum_{k=1}^{n} occ_k$ and $occ_k \geq 1$ is the number of SUSes covering location k.*

5. Experiments

We have implemented our proposal named IKXSUS in C++,[2] using the `libdivsufsort`[3] library for the suffix array construction and Kasai et al.'s method [8] to compute the lcp array. We have compared our work against Pei et al.'s RSUS [12] and Tsurata et al.'s [14] OSUS implementations, on both one-SUS and all-SUS finding for every string location. Notice that OSUS also computes the suffix array using the same `libdivsufsort` package and computes the lcp array using Kasai et al.'s method.

RSUS was originally prepared with an R interface. We stripped off that R interface and built a standalone C++ executable for the sake of fair benchmarking. OSUS was originally developed in C++. We run OSUS both with and without the `-l` option to compute a single leftmost SUS and all SUSes for every string location. In all three implementations, we also commented out the sections that print the results onto the screen and/or the disk as output, in order to measure the algorithmic performance better.

We run the tests on a machine that has Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz processor with 8192 KB cache size and 16 GB memory. The operating system was Linux Mint 14. We used the Pizza&Chili corpus in the experiments by taking

[2] Source code can be downloaded at: http://penguin.ewu.edu/~bojianxu/publications.

[3] Available at: https://code.google.com/p/libdivsufsort.

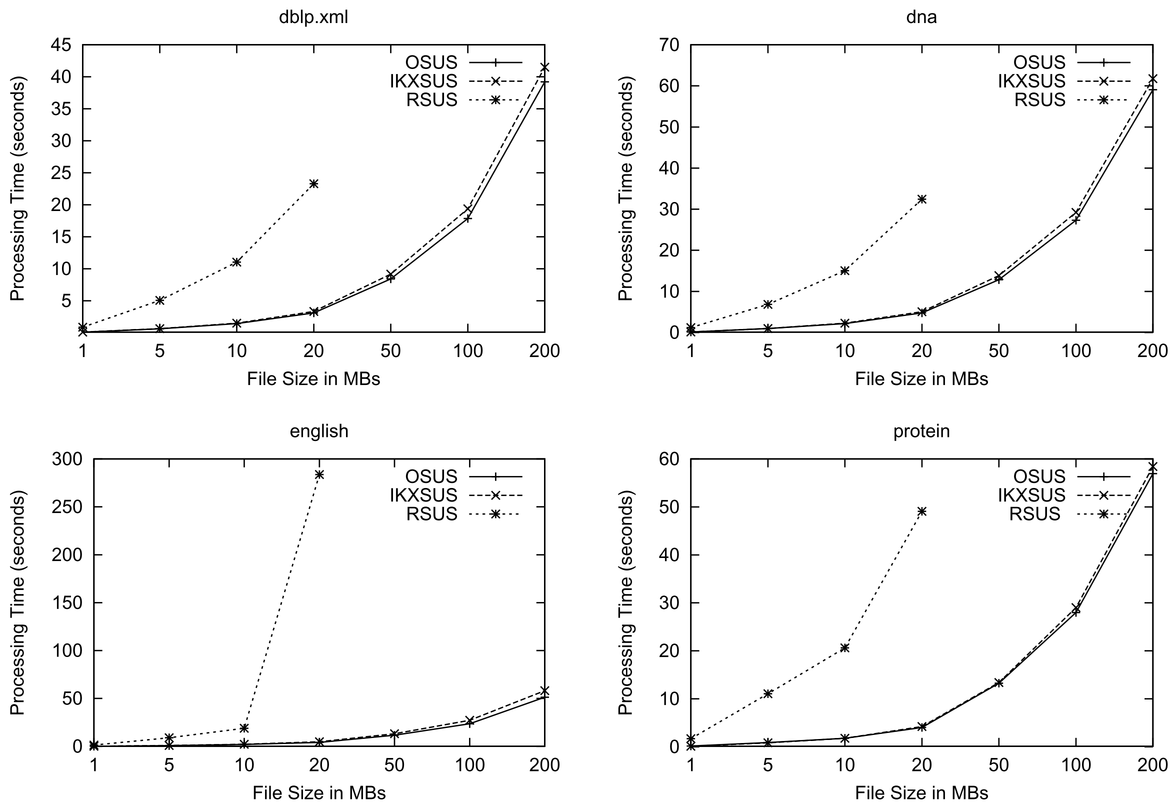

Fig. 1. The processing speed of RSUS, OSUS, and our proposal in finding the leftmost SUS of every location on several strings of different sizes.

the first 1, 5, 10, 20, 50, 100, and 200 MBs of the largest `dblp.xml`, `dna`, `English`, and `protein` files. The results are shown in Figs. 1, 2, 3, and 4.

Finding the leftmost SUS of every location, Figs. 1 and 2 It was not possible to run RSUS on longer strings, since RSUS requires more memory than what our machine has, and thus, only up to 20 MB files were included in the RSUS benchmark. Compared to RSUS, we have observed that IKXSUS is in average more than **8** times faster and uses **20** times less memory. The experimental results also revealed that difference of the processing speeds of OSUS and IKXSUS is negligible, but in average OSUS uses **4** times more memory than IKXSUS.

Finding all SUSes of every location, Figs. 3 and 4 In the experiments of all-SUS finding for every string location, RSUS was not included as it does not have this functionality. We have observed that OSUS uses less memory in the all-SUS finding than what it needs for one-SUS finding, while IKXSUS's memory cost does not change between the one-SUS and all-SUS finding. Overall, IKXSUS uses at least **2** times less memory space than OSUS and also marginally beats OSUS in terms of their processing speeds.

Although all three works have linear space complexity in both theory and experiments (note that the `X` axis in all figures uses log scale), IKXSUS and OSUS use significantly less memory space, due to the fact that these two works use simpler data structures rather than the suffix tree used by RSUS. On the other hand, although both IKXSUS and OSUS use the same set of data structures, such as suffix array, rank array (inverse suffix array), and the lcp array, and computing these arrays are done via the same library (`libdivsufsort` for suffix array construction) and the same algorithm (Kasai et al.'s method [8] for lcp array construction), the peak memory usage by OSUS is much higher than IKXSUS. The difference stems from different mechanisms these studies follow to compute the SUS. OSUS computes the SUS by using an additional array, which is named as the meaningful minimal unique substring array. Thus, the space used for that additional data structure makes OSUS require more memory.

With respect to the processing speed, both IKXSUS and OSUS present stable running times on all `dblp`, `dna`, `protein`, and `English` texts and scale well on increasing sizes of the target data conforming to their linear time complexity. On the other hand, RSUS exhibits its quadratic time complexity on all texts, and especially its running time on `English` text is much longer when compared to other text types. The speed-up of IKXSUS and OSUS against RSUS can be even more significant, when the string size becomes larger, due to the quadratic time complexity of RSUS.

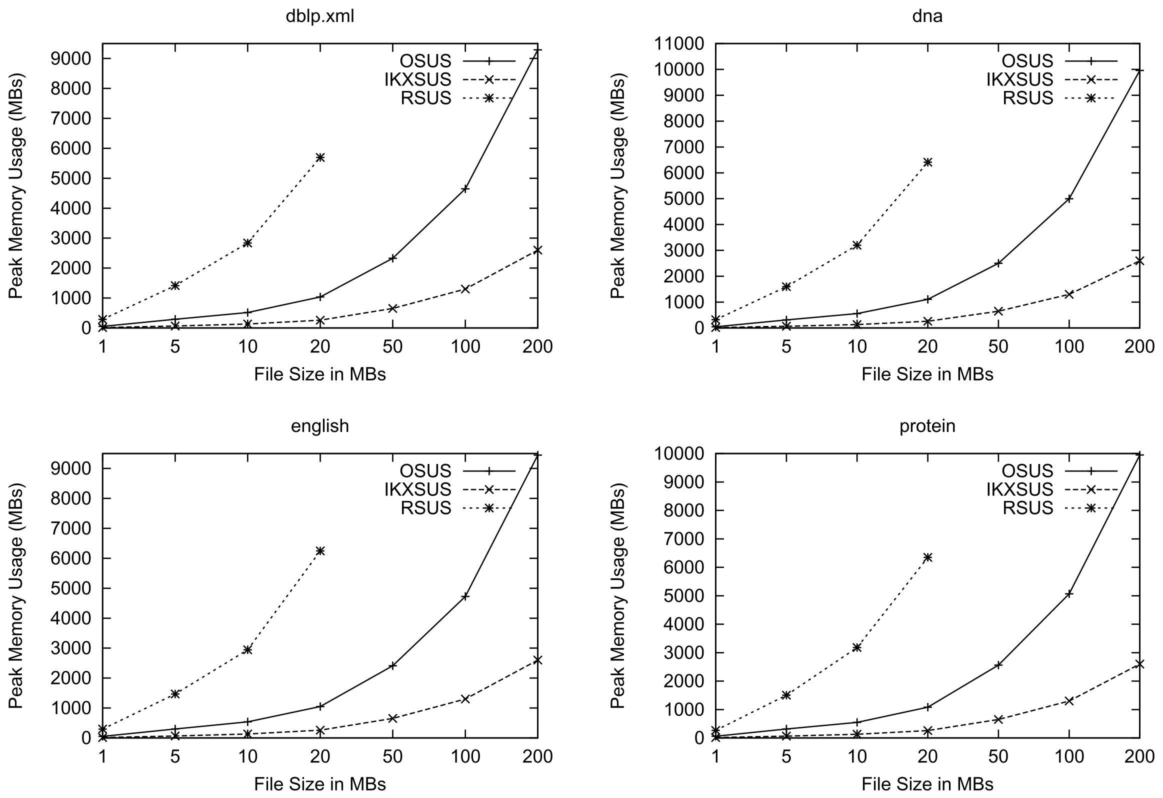

Fig. 2. The peak memory consumptions of RSUS, OSUS, and our proposal in finding the leftmost SUS of every location on several strings of different sizes.

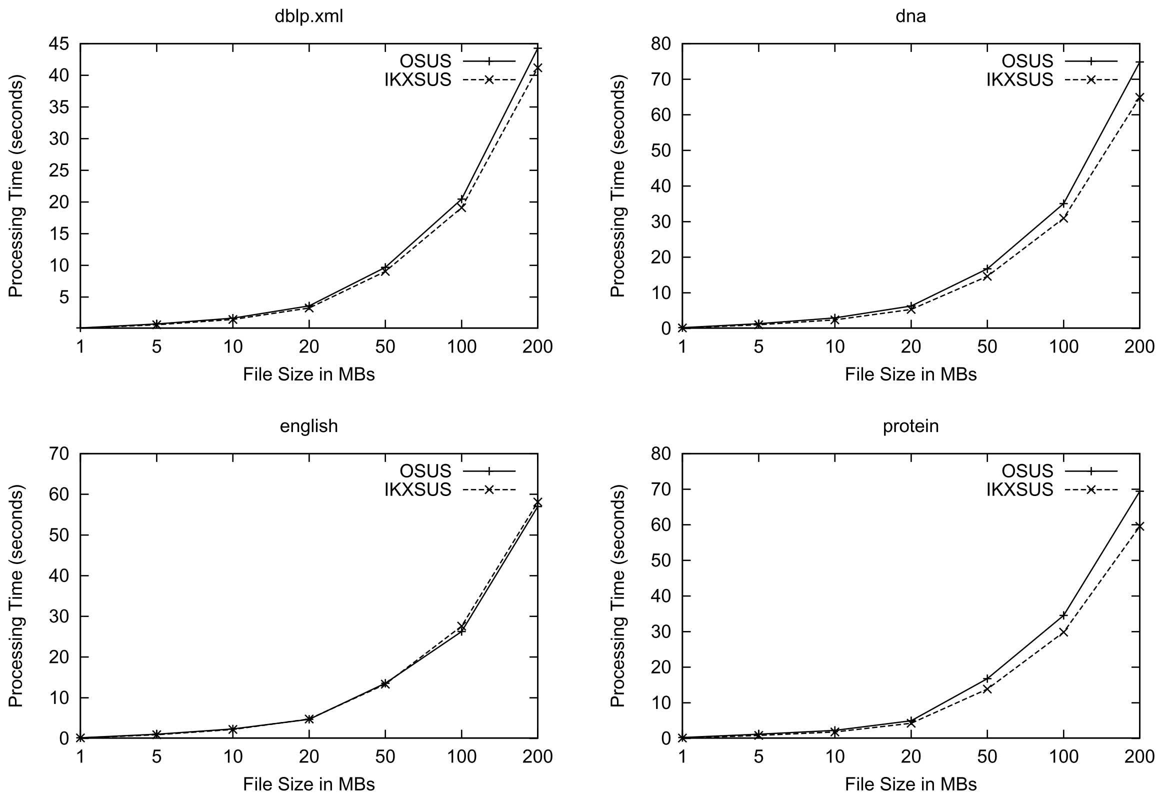

Fig. 3. The processing speed of OSUS and our proposal in finding all SUSes of every location on several strings of different sizes.

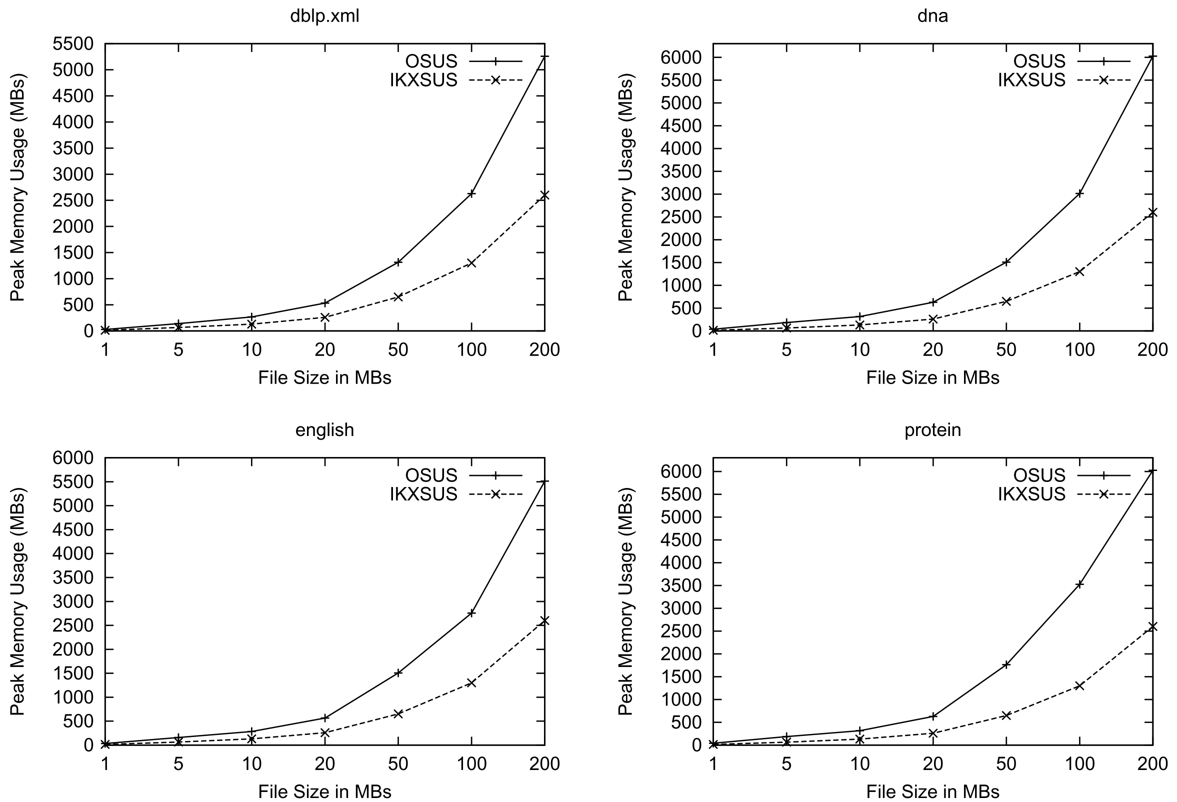

Fig. 4. The peak memory consumptions of OSUS and our proposal in finding all SUSes of every location on several strings of different sizes.

6. Conclusion

We proposed IKXSUS, an optimal linear-time and linear-space algorithm for shortest unique substring query. Our algorithm significantly improved RSUS, the original work on shortest unique substring query proposed recently [12], both theoretically and empirically in both the space and the time costs. Our work is independently discovered without knowing OSUS, another recent linear-time and linear-space solution [14] for SUS finding, and uses a different approach. In practice, IKXSUS uses significantly less memory than OSUS while maintaining nearly the same processing speed.

Acknowledgements

We acknowledge the authors of [12,14] for providing their source code.

References

[1] M. Crochemore, W. Rytter, Jewels of Stringology: Text Algorithms, World Scientific, 2003.
[2] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
[3] B. Haubold, N. Pierstorff, F. Möller, T. Wiehe, Genome comparison without alignment using shortest unique substrings, BMC Bioinform. 6 (1) (2005) 123.
[4] X. Hu, J. Pei, Y. Tao, Shortest unique queries on strings, in: Proceedings of the 21st International Symposium on String Processing and Information Retrieval (SPIRE), 2014, pp. 161–172.
[5] A.M. Ileri, M.O. Külekci, B. Xu, Shortest unique substring query revisited, in: Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM), 2014, pp. 172–181.
[6] A.M. İleri, M.O. Külekci, B. Xu, Shortest unique substring query revisited, http://arxiv.org/abs/1312.2738.
[7] L. Ilie, W.F. Smyth, Minimum unique substrings and maximum repeats, Fund. Inform. 110 (1–4) (2011) 183–195.
[8] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Symposium on Combinatorial Pattern Matching, 2001, pp. 181–192.
[9] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, J. Discrete Algorithms 3 (2–4) (2005) 143–156.
[10] M.O. Külekci, J.S. Vitter, B. Xu, Efficient maximal repeat finding using the Burrows–Wheeler transform and wavelet tree, IEEE/ACM Trans. Comput. Biol. Bioinform. 9 (2) (2012) 421–429.
[11] S. Kurtz, J.V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, R. Giegerich, Reputer: the manifold applications of repeat analysis on a genomic scale, Nucleic Acids Res. 29 (22) (2001) 4633–4642.

[12] J. Pei, W.C.H. Wu, M.Y. Yeh, On shortest unique substring queries, in: Proceedings of IEEE International Conference on Data Engineering (ICDE), 2013, pp. 937–948.
[13] W.F. Smyth, Computing regularities in strings: a survey, European J. Combin. 34 (1) (2013) 3–14.
[14] K. Tsuruta, S. Inenaga, H. Bannai, M. Takeda, Shortest unique substrings queries in optimal time, in: Proceedings of International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), 2014, pp. 503–513.