Shortest Unique Substring Query Revisited[⋆]

Atalay Mert İleri[1], M. Oğuzhan Külekci[2], and Bojian Xu[3,⋆⋆]

[1] Department of Computer Engineering, Bilkent University, Turkey
[2] TÜBİTAK National Research Institute of Electronics and Cryptology, Turkey
[3] Department of Computer Science, Eastern Washington University, WA 99004, USA
aileri@bilkent.edu.tr, oguzhan.kulekci@tubitak.gov.tr, bojianxu@ewu.edu

Abstract. We revisit the problem of finding shortest unique substring (SUS) proposed recently by Pei *et al.* (ICDE'13). 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 can also be 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 that shows our method is much faster and more space-saving.

Keywords: shortest unique substring, repetitiveness, regularity.

1 Introduction

Repetitive structure and regularity finding [1] has received much attention in stringology due to its comprehensive applications in different fields, including natural language processing, computational biology and bioinformatics, security, and data compression. However, finding the shortest unique substring (SUS) covering a given string location was not studied, until recently it was proposed by Pei *et al.* [5]. As pointed out in [5], SUS finding has its own important usage in search engines and bioinformatics. We refer readers to [5] 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 a 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 [5] is built upon the suffix tree data structure. Our

[⋆] Part of the work was done while the first and the third authors were with TÜBİTAK-BİLGEM-UEKAE Mathematical and Computational Sciences Labs in 2013 summer. All missed proofs and pseudocode can be found in the full version of this paper [2].
[⋆⋆] Corresponding author. Supported in part by EWU's Faculty Grants for Research and Creative Works.

© Springer International Publishing Switzerland 2014

algorithm also provides the functionality of finding all the SUSes covering every location, whereas the method of [5] searches for only one SUS for every location. Our method not only improves their results theoretically, the empirical study also shows that our method gains space saving by a factor of 20 and a speedup by a factor of four. The speedup gained by our method can become even more significant when the string becomes longer due to the quadratic time cost of [5]. Due to the very high memory consumption of [5], we were not able to run their method with massive data on our machine.

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 is 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 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 $S =$ `abcbb`, then SUS_2 can be either $S[1, 2] =$ `ab` or $S[2, 3] =$ `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 $LSUS_1 = SUS_1$, which always exists. However, if S is not suffixed by an artificial terminator character $\$ \notin \Sigma$, then for an arbitrary location $k \ge 2$, $LSUS_k$ may not exist. For example, if $S =$ `abcabc`, then none of $\{LSUS_4, LSUS_5, LSUS_6\}$ exists. An **up-to-j extension of $\mathbf{LSUS_k}$** is the substring $S[k \ldots j]$, where $k + |\, LSUS_k \,| \le j \le n$.

The **suffix array** $SA[1 \dots n]$ of the string S is a permutation of $\{1, 2, \dots, n\}$, such that for any i and j, $1 \le i < j \le n$, we have $S[SA[i] \dots n] < S[SA[j] \dots n]$. 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** $Rank[1 \dots n]$ is the inverse of the suffix array. That is, $Rank[i] = j$ iff $SA[j] = i$. The **longest common prefix (lcp) array** $LCP[1 \dots n+1]$ is an array of $n+1$ integers, such that for $i = 2, 3, \dots, n$, $LCP[i]$ is the length of the lcp of the two suffixes $S[SA[i-1] \dots n]$ and $S[SA[i] \dots n]$. 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 $LCP[n+1]$ is just to simplify the description of our upcoming algorithms. The next Lemma 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.

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

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

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

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 an extension to find all the SUSes covering location k with the same time and space complexities, if k has multiple SUSes.

Lemma 2. *Every SUS is either an LSUS or an extension of an LSUS.*

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

By Lemma 2, 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 $LSUS_1, \dots, LSUS_k$ if existing, using Lemma 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 LSUS or their up-to-k extensions as SUS_k. We resolve the tie by picking the leftmost candidate. 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. Due to the page limit, we give the pseudocode of this procedure in the full version of this paper [2].

Lemma 3. *Given a string location k and the rank and the lcp array of the string S, we can find SUS_k using $O(k)$ time. If multiple SUS_k exist, the leftmost one is returned.*

Adding the linear time cost for the construction of the suffixe array, the rank array, and the lcp array, we have the following theorem.

Theorem 1. *For any location k in the string S, we can find SUS_k using $O(n)$ time and space. If multiple SUS_k exist, the leftmost one is returned.*

It is trivial to extend the one-SUS finding algorithm to find all the SUSes covering a particular location k as follows. We will first find the leftmost SUS_k. Then we start over again to recheck $LSUS_1 \ldots LSUS_k$ or their up-to-k extensions, and return those whose length is equal to the length of SUS_k. The pseudocode of this new procedure is given in [2]. This procedure clearly costs an extra $O(k)$ time. Combining the claim in Theorem 1, we get the following theorem.

Theorem 2. *We can find all the SUSes covering any given 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 $k = 1, 2, \ldots, n$. If k has multiple SUSes, the leftmost one will be returned. In the end, we will show an extension to return all SUSes for every location.

A natural solution is to iteratively use the algorithm for finding the SUS covering a particular location as a subroutine to find every SUS_k, for $k = 1, 2, \ldots, n$. 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 suffix array, the rank array, and the lcp array, and $\sum_{k=1}^{n} O(k)$ is the total time cost for the n instances of the one-SUS finding algorithm. 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 2, we know that every SUS must be an LSUS or an extension of an LSUS. The next Lemma 4 further says if SUS_k is an extension of an LSUS, it has special properties and can be quickly obtained from SUS_{k-1}.

Lemma 4. *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 $S[k]$.*

4.1 The Overall Strategy

We are ready to present the overall strategy for finding SUS of every location, by using Lemma 2 and 4. 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 1. Now let's look at the calculation of a particular SUS_k, $k \geq 2$. By Lemma 2, we know SUS_k is either an LSUS or an extension of an LSUS. By Lemma 4, 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 SUS_k cannot be an extension of an LSUS (the contrapositive of Lemma 4). 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, by the time we want to calculate SUS_k, $k \geq 2$, is to ensure that we would already have calculated the shortest LSUS covering location k or we would already have known that no LSUS covers location k.

4.2 Preparation

We now focus on the calculation of the shortest LSUS covering every string location k, denoted by $\mathbf{SLS_k}$. Let $\mathbf{Candidate_i^k}$ denote the shortest one among those of $\{LSUS_1, \ldots, LSUS_k\}$ that exist and cover location i. The leftmost one will be picked if multiple choices exist for both SLS_k and $Candidate_i^k$. 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. However, if SLS_k exists, by the definition of SLS and $Candidate$, we have:

Fact 1. *If SLS_k exists: $SLS_k = Candidate_k^k = Candidate_k^{k+1} = \cdots = Candidate_k^n$*

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 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 1). Those $Candidate_i^k$ with $i \leq k$ would have 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 5. *(1) $LSUS_1$ always exists. (2) If $LSUS_k$ exists, then $\{LSUS_1$, $LSUS_2$, …, $LSUS_k\}$ all exist. (3) If $LSUS_k$ does not exist, then none of $\{LSUS_k$, $LSUS_{k+1}$, …, $LSUS_n\}$ exists.*

We know after k walk steps, we have calculated $LSUS_1, LSUS_2, \ldots, LSUS_k$. By Lemma 5, 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., $LSUS_{\ell_k} = S[\ell_k \ldots \gamma_k]$. We know every location $j = 1, \ldots, \gamma_k$ has its candidate $Candidate_j^k$ calculated already, because every such location j has been covered by at least one of the LSUSes among $LSUS_1, \ldots, LSUS_{\ell_k}$. We also know 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 6. *At the end of the kth walk step, if $\gamma_k > k$, then for any i and j, $k \le i < j \le \gamma_k$, $Candidate_j^k$ also covers location i.*

Lemma 7. *At the end of the kth walk step, if $\gamma_k > k$, then $|\, Candidate_k^k \,| \le |\, Candidate_{k+1}^k \,| \le \ldots \le |\, Candidate_{\gamma_k}^k \,|$.*

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

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) 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.3) All nodes representing different chunks will be connected into a linked list, which has a `head` and a `tail`, referring to the two nodes that represent the lowest positioned chunk and the highest positioned chunk. (B) If $\gamma_k \le k$, the linked list is empty.

Maintenance of the Invariant. We describe in an inductive manner the procedure that maintains the invariant. Algorithm 1 shows the pseudocode. We start with an empty linked list.

Base Step: $k = 1$. We are walking the first step. We first calculate $LSUS_1$ using Lemma 1. We know $LSUS_1$ must exist. Let's say $LSUS_1 = S[1 \ldots \gamma_1]$ for some $\gamma_1 \le n$. 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 1), so we return SLS_1 by returning (`head.start`, `head.length`) $= (1, \gamma_1)$. We then update `head.ChunkStart` from 1 to be 2. If it turns out `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.

Algorithm 1. Function calls *FindSLS(1)*, ..., *FindSLS(n)* return SLS_1, ..., SLS_n, if the corresponding *SLS* exists; otherwise, `null` is returned

```
1  Construct $Rank[1...n]$ and $LCP[1...n]$ of the string $S$;
2  Initialize an empty $List$;      // Each node has four fields: {ChunkStart, ChunkEnd, start,
   length}.
3  $head \leftarrow 0$; $tail \leftarrow 0$ ;              // Reference to the head and tail node of the $List$

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

           /* Update candidates and merge the nodes whose candidates can be shorter.
              Resolve the tie by picking the leftmost one.                    */
10         $j \leftarrow tail$;
11         while $j \geq head$ and $List[j].length > L + 1$ do $j--$;
12         ;
13         $List[j+1] \leftarrow (List[j+1].ChunkStart, List[tail].ChunkEnd, k, L+1)$;
           $tail \leftarrow j + 1$;
14     if $head \neq 0$ then $SLS_k \leftarrow (head.start, head.length)$ ;     // The list is not empty.
15     else $SLS_k \leftarrow$ (null, null) ;                            // $SLS_k$ does not exist.
16     ;

       /* Discard the information about location $k$ from the $List$.        */
17     if $head > 0$ then                                          // $List$ is not empty
18         if $List[head].ChunkEnd \leq k$ then
19             $head++$;                                    // Delete the current head node
20             if $head > tail$ then $head \leftarrow 0$; $tail \leftarrow 0$; ;          // $List$ becomes empty
21         else $List[head].ChunkStart \leftarrow k + 1$;
22         ;
23     return $SLS_k$
```

Inductive Step: $k \geq 2$. We are walking the kth step. We first calculate $LSUS_k$. Case 1: $LSUS_k$ does not exist. (1) If `head` does not exist. It means that location k is covered neither by any of $LSUS_1, \dots, LSUS_{k-1}$ nor by $LSUS_k$, so SLS_k simply does not exist, and we will simply return $(\texttt{null}, \texttt{null})$ to indicate that SLS_k does not exist. (2) If `head` exists, $(\texttt{head.start}, \texttt{head.length})$ will be returned as SLS_k, because $Candidate_k^k = SLS_k$ (Fact 1). Then we will remove the information about location k from the head by setting $head.ChunkStart = k + 1$. After that, we will remove the `head` node if $\texttt{head.ChunkEnd} < \texttt{head.ChunkStart}$.

Case 2: $LSUS_k$ exists and $LSUS_k = S[k \dots \gamma_k]$, $\gamma_k \leq n$. By Lemma 5, we know $LSUS_1, \dots, LSUS_{k-1}$ all exist. Let γ_{k-1} denote the right boundary of $LSUS_1, \dots, LSUS_{k-1}$. By Lemma 8, we know $\gamma_k \geq \gamma_{k-1}$ 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 $\gamma_{k-1} \geq k$ are possible. (1) If `head` does not exist, it means $\gamma_{k-1} < k$ and none of locations $\{k \dots \gamma_k\}$ is covered by any of $LSUS_1, \dots, LSUS_{k-1}$. 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 $\gamma_{k-1} \geq k$. If $\gamma_\mathtt{k} > \mathtt{tail.ChunkEnd} = \gamma_\mathtt{k-1}$, we will first insert a new node $(\mathtt{tail.ChunkEnd} + \mathtt{1}, \gamma_\mathtt{k}, \mathtt{k}, \gamma_\mathtt{k} - \mathtt{k} + \mathtt{1})$ at the tail side of the linked list to record the candidate information for locations in the chunk after γ_{k-1} through γ_k. After the work in either (1) or (2) is finished, we then 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. This travel is valid because of Lemma 7. We will merge all the nodes whose candidates are longer than $LSUS_k$ into one 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 obtained from merging is $LSUS_k$. This merge process ensures every location maintains its best (shortest) candidate by the end of each walk step, and also resolves ties of multiple candidates by picking the leftmost one. We will return (head.start, head.length) as SLS_k, because $Candidate_k^k = SLS_k$ (Fact 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 head.ChunkEnd > head.ChunkStart.

Lemma 9. *Given the lcp array and the rank array of S, the amortized time cost of FindSLS() is $O(1)$ over the sequence of function calls FindSLS(1), FindSLS(2), ..., FindSLS(n).*

4.4 Finding *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 at the order of $k = 1, 2, \ldots, n$, by using the strategy described in Section 4.1. Due to the page limit, the pseudocode describing this procedure is given in [2].

Theorem 3. *We can find $SUS_1, SUS_2, \ldots, SUS_n$ of string S using a total of $O(n)$ time and space.*

4.5 Extension: Finding all the SUSes for every Location

It is possible that a particular location can have multiple SUSes. For example, if $S = \mathtt{abcbb}$, then SUS_2 can be either $S[1, 2] = \mathtt{ab}$ or $S[2, 3] = \mathtt{bc}$. The algorithm of Theorem 3 only returns one of them. However, we can easily modify the algorithm to return all the SUSes of every location, without changing Algorithm 1.

Suppose a particular location k has multiple SUSes. We know, at the end of the kth walk step but before the linked list update, SLS_k returned by Algorithm 1 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 7, 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

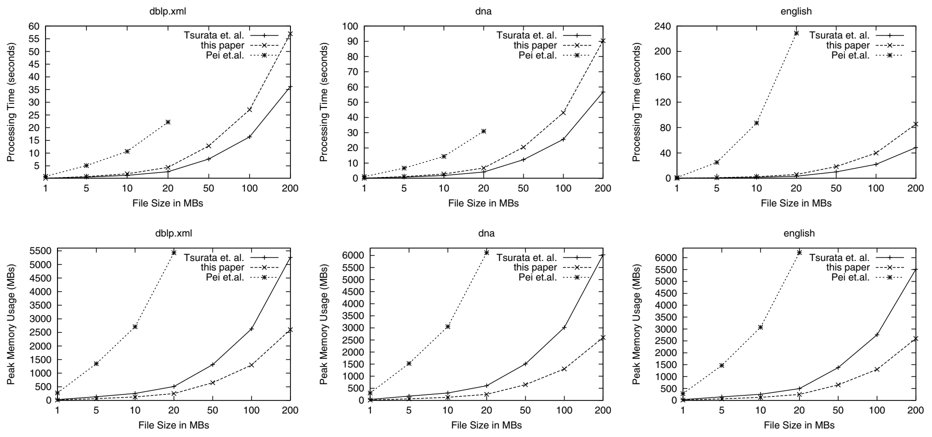

Fig. 1. Processing speed and peak memory consumption of RSUS, OSUS, and ours

has a candidate longer than SUS_k or does not have a candidate calculated yet, but that location is indeed covered by a 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 LSUSes, whose lengths are equal to the length of SUS_k, which 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. Due to the page limit, the pseudocode describing this procedure is given in [2]. The overall time cost of maintaining the linked list data structure (the sequence of function calls $\mathit{FindSLS}(1), \mathit{FindSLS}(2), \ldots, \mathit{FindSLS}(n)$) 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.

5 Experiments

We have implemented our proposal in `C++` without best engineering effort, using the `libdivsufsort`[1] library for the suffix array construction and Kasai *et al.*'s method [3] to compute the LCP array. We have compared our work against Pei *et al.*'s RSUS [5] and Tsurata *et al.*'s [6] OSUS implementations, a recent independent work obtained via personal communication after we posted our work at `arXiv`. Notice that OSUS also computes the suffix array with the same `libdivsufsort` package.

RSUS was 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

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

developed in `C++`. We run it with the `-l` option to compute a single leftmost SUS for a given position rather than its default configuration of reporting all SUSs. We also commented the sections that print the results to the screen on all three programs so as to measure the algorithmic performance better.

We run the tests on a machine that has Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz processor with 8192 KB cache size and 16GB memory. The operating system was Linux Mint 14. We used the Pizza&Chili corpus in the experiments by taking the first 1, 5, 10, 20, 50, 100, and 200 MBs of the largest *dblp.xml*, *dna*, and *English* files. The results are shown in Figure 1.

It was not possible to run the RSUS on large files, since RSUS requires more memory than that our machine has, and thus, only up to 20MB files were included in the RSUS benchmark. Compared to RSUS, we have observed that our proposal is more than 4 times faster and uses 20 times less memory. The experimental results revealed that OSUS is on the average 1.6 times faster than our work, but in contrast, uses 2.6 times more memory.

The asymptotic time and space complexities of both ours and OSUS are same as being linear (note that the x axis in both figures uses log scale). The peak memory usage of OSUS and ours are different although they both use suffix array, rank array (inverse suffix array), and the LCP array, and computing these arrays are done with the same library (libdivsufsort). The difference stems from different ways 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 in the corresponding study. Thus, the space used for that additional data structure makes OSUS require more memory.

Both OSUS and our scheme present stable running times on all dblp, dna, 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 $O(n^2)$ time complexity on all texts, and especially its running time on english text takes much longer when compared to other text types.

References

1. Crochemore, M., Rytter, W.: Jewels of stringology. World Scientific (2003)
2. İleri, A.M., Külekci, M.O., Xu, B.: Shortest unique substring query revisited, `http://arxiv.org/abs/1312.2738`
3. Kasai, T., Lee, G.H., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
4. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. Journal of Discrete Algorithms 3(2-4), 143–156 (2005)
5. Pei, J., Wu, W.C.H., Yeh, M.Y.: On shortest unique substring queries. In: Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE), pp. 937–948 (2013)
6. Tsuruta, K., Inenaga, S., Bannai, H., Takeda, M.: Shortest unique substrings queries in optimal time. In: Geffert, V., Preneel, B., Rovan, B., Štuller, J., Tjoa, A.M. (eds.) SOFSEM 2014. LNCS, vol. 8327, pp. 503–513. Springer, Heidelberg (2014)