In-place algorithms for exact and approximate shortest unique substring problems

Wing-Kai Hon\textsuperscript{a}, Sharma V. Thankachan\textsuperscript{b}, Bojian Xu\textsuperscript{c,}\textsuperscript{*}

\textsuperscript{a} Department of Computer Science, National Tsing Hua University, Hsinchu 300, Taiwan
\textsuperscript{b} Department of Computer Science, University of Central Florida, FL 32816, USA
\textsuperscript{c} Department of Computer Science, Eastern Washington University, WA 99004, USA

\begin{abstract}
We revisit the exact shortest unique substring (SUS) finding problem, and propose its approximate version where mismatches are allowed, due to its applications in subfields such as computational biology. We design a generic in-place framework that fits to solve both the exact and approximate k-mismatch SUS finding, using the minimum 2n memory words, each of $\lceil \log_2(n) \rceil$ bits, plus $n$ bytes space, where $n$ is the input string size. By using the in-place framework, we can find the exact and approximate k-mismatch SUS for every string position using a total of $O(n)$ and $O(n^2)$ time, respectively, regardless of the value of $k$. Our framework does not involve any compressed or succinct data structures and thus is practical and easy to implement. Experimental study shows that the peak memory usage of our proposal is consistently 9n bytes for any string of size $n$, validating the claim that our solution is in-place. Further, our proposal uses much less memory and is much faster than the currently best work that has implementation for exact SUS finding.
\end{abstract}

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

We consider a string $S[1..n]$, where each character $S[i]$ is drawn from an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. We say the character $S[i]$ occupies the string position $i$. A substring $S[i..j]$ of $S$ represents $S[i]S[i+1]\ldots S[j]$ if $1 \leq i \leq j \leq n$, and is an empty string if $i > j$. We call $i$ the start position and $j$ the end position of $S[i..j]$. We say the substring $S[i..j]$ covers the kth position of $S$, if $i \leq k \leq j$. String $S[i'..j']$ is a proper substring of another string $S[i..j]$ if $i \leq i' \leq j' \leq j$ and $j' - i' < j - i$. The length of a non-empty substring $S[i..j]$, denoted as $|S[i..j]|$, is $j - i + 1$. We define the length of an empty string as zero.

The Hamming distance of two non-empty strings $A$ and $B$ of equal length, denoted as $H(A,B)$, is defined as the number of string positions where the characters differ. A substring $S[i..j]$ is $k$-mismatch unique, for some $k \geq 0$, if there does not exist another substring $S[i'..j']$, such that $i' \neq i$, $j - i = j' - i'$, and $H(S[i..j], S[i'..j']) \leq k$. A substring is a $k$-mismatch repeat if it is not $k$-mismatch unique.

\* Author names are listed in alphabetical order. A preliminary version of this article appeared at [1].
\* Corresponding author.
\textsuperscript{a} E-mail addresses: wkhon@cs.nthu.edu.tw (W.-K. Hon), sharma.thankachan@ucf.edu (S.V. Thankachan), bojianxu@ewu.edu (B. Xu).
Definition 1.1 (k-mismatch SUS). For a particular string position $p$ in $S$ and an integer $k$, $0 \leq k \leq n - 1$, the $k$-mismatch shortest unique substring (SUS) covering position $p$, denoted as $SUS_p^k$, is a $k$-mismatch unique substring $S[i..j]$, such that (1) $i \leq p \leq j$, and (2) there does not exist another $k$-mismatch unique substring $S[i’..j’]$, such that $i’ \leq p \leq j’$ and $j’ - i’ < j - i$.

We call 0-mismatch SUS as exact SUS, and the case $k > 0$ as approximate SUS. For any $k$ and $p$, $SUS_p^k$ must exist, because at least the string $S$ can be $SUS_p^0$, if none of its proper substrings is $SUS_p^0$. On the other hand, there might be multiple choices for $SUS_p^k$. For example, if $S = abcba$, $SUS_2^2$ can be either $S[1..2] = ab$ or $S[2..3] = bc$, and $SUS_1^3$ can be either $S[1..3] = abc$ or $S[2..4] = bca$. Note that in Definition 1.1, we require $k < n$, because finding $SUS_p^k$ is trivial: $SUS_p^0 \equiv S$ for any string position $p$.

Problem (k-mismatch SUS finding). Given the string $S$, the value of $k \geq 0$, and two empty integer arrays $A$ and $B$, we want to work in the place of $S$, $A$, and $B$, such that in the end of computation: (1) $S$ does not change. (2) Each $(A[i], B[i])$ pair stores the start and ending positions of the rightmost$^1$ $SUS_p^k$, i.e., $S[A[i]..B[i]] = SUS_p^k$, using a total of $O(n)$ time for $k = 0$ and $O(n^2)$ time for any $k \geq 1$.

1. Prior work and our contribution

Exact SUS finding was proposed and studied recently by Pei et al. [2], due to its application in locating snippets in document search, event analysis, and bioinformatics, such as finding the distinctness between closely related organisms [3], polymerase chain reaction (PCR) primer design in molecular biology, genome mapability [4], and next-generation short reads sequencing [5]. The algorithm in [2] can find all exact SUS in $O(n^2)$ time using a suffix tree of $O(n)$ space. Following their proposal, there has been a sequence of improvements [6,7] for exact SUS finding, reducing the time cost from $O(n^2)$ to $O(n)$ and alleviating the underlying data structure from suffix tree to suffix array of $O(n)$ space. Hu et al. [8] proposed an RMQ (range minimum query) technique based indexing structure, which can be constructed in $O(n)$ time and space, such that any future exact SUS covering any interval of string positions can be answered in $O(1)$ time. In this work, we make the following contributions:

- We revisit the exact SUS finding problem and also propose its approximate version where mismatches are allowed, which significantly increases the difficulty as well as the usage of SUS finding in subfields such as bioinformatics, where approximate string matching is unavoidable due to genetic mutation and errors in biological experiments.
- We propose a generic in-place algorithmic framework that fits to solve both the exact and approximate k-mismatch SUS finding, using $2n$ words, each of $[\log_2(n)]$ bits, plus $n$ bytes space. It is worth mentioning that $2n$ words plus $n$ bytes is the minimum memory space needed to store those $n$ calculated SUSes: (1) It needs 2 words to store each SUS by saving its start and ending positions (or one endpoint and its length) and there are $n$ SUSes. (2) It needs another $n$ bytes to store the original string $S$ in order to output the actual content of any SUS of interest from queries. Note that all prior work [2,6-8] use $O(n)$ space but there is big leading constant hidden within the big-oh notation (see the experimental study in [7]).
- After the suffix array is constructed, all the computation in our solution happens in the place of two integer arrays, using non-trivial techniques. It is worth noting that our solution does not involve any compressed or succinct data structures, making our solution practical and easy to implement. Our experimental study shows that our solution for exact SUS finding is much faster than the fastest one among [2,6,7],$^2$ in addition to a lot more space saving than them, enabling our solution to handle larger data sets.

2. Preparation

A prefix of $S$ is a substring $S[1..i]$, $1 \leq i \leq n$. A proper prefix $S[1..i]$ is a prefix of $S$ where $i < n$. A suffix of $S$ is a substring $S[i..n]$, denoted as $S_i$, $1 \leq i \leq n$. $S_i$ is a proper suffix of $S$, if $i > 1$.

For two strings $A$ and $B$, we write $A = B$ (and say $A$ is equal to $B$), if $|A| = |B|$ and $H(A, B) = 0$. We say $A$ is lexicographically smaller than $B$, denoted as $A < B$, if (1) $A$ is a proper prefix of $B$, or (2) there exists $k \geq 1$, such that $A[k] < B[k]$ and $A[i] = B[i]$ for every $i$, $1 \leq i \leq k - 1$, if $k > 1$.

The suffix array $SA[1..n]$ of the string $S$ is a permutation of $\{1, 2, \ldots, n\}$, such that for any $i$ and $j$, $1 \leq i < j \leq n$, we have $S[SA[i]..n] < S[SA[j]..n]$. That is, $SA[i]$ is the start position of the $i$th smallest suffix in the lexicographic order. The rank array $RA[1..n]$ is the inverse of the suffix array, i.e., $RA[i] = j$ iff $SA[j] = i$.

$^1$ Since any SUS may have multiple choices, it is our arbitrary decision to resolve the ties by picking the rightmost choice. However, our solution can also be easily modified to find the leftmost choice.

$^2$ Note that the work of [8] studies a different problem and its computation is of the query-answer model, and thus is not comparable with [2,6,7] and ours.
Definition 2.1. The \( k \)-mismatch longest common prefix (LCP) between two strings \( A \) and \( B \), \( k \geq 0 \), denoted as \( \text{LCP}^k(A, B) \), is the longest prefix of \( A \) and \( B \) within Hamming distance \( k \).

For example, if \( A = \text{abc} \) and \( B = \text{acb} \), then: \( \text{LCP}^0(A, B) = A[1] = B[1] = a \) and \( |\text{LCP}^0(A, B)| = 1 \); \( \text{LCP}^1(A, B) = A[1..2] = ab \) and \( B[1..2] = ac \) and \( |\text{LCP}^1(A, B)| = 2 \).

Definition 2.2 (\( k \)-mismatch LSUS). For a particular string position \( p \) in \( S \) and an integer \( k, 0 \leq k \leq n - 1 \), the \( k \)-mismatch left-bound shortest unique substring (LSUS) starting at position \( p \), denoted as \( \text{LSUS}^k_p \), is a \( k \)-mismatch unique substring \( S[p..j] \), such that either \( p = j \) or any proper prefix of \( S[p..j] \) is not \( k \)-mismatch unique.

We call 0-mismatch LSUS as exact LSUS, and the case \( k > 0 \) as approximate LSUS. Observe that for any \( k \), \( \text{LSUS}^k_1 = \text{SUS}_1^k \) always exists, because the whole string \( S \) is unique and \( \text{LSUS}^k_1 \) is the shortest prefix of \( S \) that is unique. However, for any \( k \geq 0 \) and \( p \geq 2 \), \( \text{LSUS}^k_p \) may not exist. For example, if \( S = \text{dabcabc} \), none of \( \text{LSUS}^0_p \) and \( \text{LSUS}^1_2 \) exists, for all \( i \geq 5 \), \( j \geq 4 \). It follows that some string positions may not be covered by any \( k \)-mismatch LSUS. For example, for the same string \( S = \text{dabcabc} \), positions 6 and 7 are not covered by any exact or 1-mismatch LSUS. On the other hand, if any \( \text{LSUS}^k_p \) does exist, there must be only one choice for \( \text{LSUS}^k_p \), because \( \text{LSUS}^k_p \) has its start position fixed on \( p \) and need to be as short as possible. Note that in Definition 2.2, we require \( k < n \), because finding \( \text{LSUS}^k_p \) is trivial as \( \text{LSUS}^1_1 = S \) and \( \text{LSUS}^k_0 \) does not exist for all \( p > 1 \).

Definition 2.3 (\( k \)-mismatch SLS). For a particular string position \( p \) in \( S \) and an integer \( k, 0 \leq k \leq n - 1 \), we use \( \text{SLS}^k_p \) to denote the shortest \( k \)-mismatch LSUS covering position \( p \).

We call 0-mismatch SLS as exact SLS, and the case \( k > 0 \) as approximate SLS. \( \text{SLS}^k_p \) may not exist, since position \( p \) may not be covered by any \( k \)-mismatch LSUS at all. For example, if \( S = \text{dabcabc} \), then none of \( \text{SLS}^0_1 \) and \( \text{SLS}^1_2 \) exists, for all \( p \geq 6 \). On the other hand, if \( \text{SLS}^k_p \) exists, there might be multiple choices for \( \text{SLS}^k_p \). For example, if \( S = \text{abcdabc} \), \( \text{SLS}^k_3 \) can be either \( \text{LSUS}^1_1 = S[1..2] \) or \( \text{LSUS}^2_2 = S[2..3] \), and \( \text{SLS}^k_1 \) can be any one of \( \text{LSUS}^1_1 = S[1..3] \), \( \text{LSUS}^1_1 = S[2..4] \), and \( \text{LSUS}^1_1 = S[3..5] \). Note that in Definition 2.3, we require \( k < n \), because finding \( \text{SLS}^k_p \) is trivial as \( \text{SLS}^1_1 = S \) for all \( p \).

Lemma 2.1. For any \( k \) and \( p \): (1) \( \text{LSUS}^k_1 \) always exists. (2) If \( \text{LSUS}^k_p \) exists, then \( \text{LSUS}^k_i \) exists, for all \( i \leq p \). (3) If \( \text{LSUS}^k_p \) does not exist, then none of \( \text{LSUS}^k_i \) exists, for all \( i \geq p \).

Proof. (1) \( \text{LSUS}^k_1 \) must exist, because at least the string \( S \) can be \( \text{LSUS}^k_1 \) if every proper prefix of \( S \) is a \( k \)-mismatch repeat. (2) If \( \text{LSUS}^k_p \) exists, say \( \text{LSUS}^k_p = S[p..q] \), \( q \geq p \), then \( \text{LSUS}^k_i \) exists for every \( i \leq p \), because at least \( S[i..q] \) is \( k \)-mismatch unique. (3) It is true, because otherwise we get a contradiction to the second statement in the lemma. □

Lemma 2.2. For any \( k \) and \( p \), \( |\text{LSUS}_p^k| \geq |\text{LSUS}_{p-1}^k| - 1 \), if \( \text{LSUS}^k_p \) exists.

Proof. Suppose the \( k \)-mismatch substring \( \text{LSUS}_p^k = S[p..q] \), for some \( q \geq p \). Then, \( S[p-1..q] \) is also \( k \)-mismatch unique. It follows immediately that, \( |\text{LSUS}_{p-1}^k| \leq |S[p-1..q]| = 1 + |\text{LSUS}_p^k| \). □

Lemma 2.3. For any \( k \) and \( p \), \( \text{SUS}^k_p \) is either \( \text{SLS}^k_p \) or \( S[i..p] \), for some \( i, i + |\text{SUS}^k_i| < p \). That is, \( \text{SUS}^k_p \) is either the shortest \( k \)-mismatch LSUS that covers position \( p \), or a right extension (through position \( p \)) of a \( k \)-mismatch LSUS.

Proof. We know \( \text{SUS}^k_p \) must exist, because at least the string \( S \) can be \( \text{SUS}^k_p \). Let’s say \( \text{SUS}^k_p = S[i..j] \), \( i \leq p \leq j \). If \( S[i..j] \) is neither \( \text{SUS}^k_p \) nor a right extension of \( \text{SUS}^k_j \), it means \( S[i..j] \) is a proper prefix of \( \text{SUS}^k_j \) and thus is a \( k \)-mismatch repeat, which is a contradiction to the fact that \( S[i..j] = \text{SUS}^k_p \) is \( k \)-mismatch unique. Therefore, \( \text{SUS}^k_p = S[i..j] \) is either \( \text{SUS}^k_p \), or a right extension of \( \text{SUS}^k_p \) (clearly, \( j = p \) in this case). Further, if \( \text{SUS}^k_p = S[i..j] = \text{SUS}^k_1 \), it is obvious that \( \text{SUS}^k_1 \) must be the shortest \( k \)-mismatch LSUS covering position \( p \), i.e., \( \text{SUS}^k_p = \text{SLS}^k_p \). □

For example, let \( S = \text{dabcabc} \), then: (1) \( \text{SUS}^0_3 \) can be either \( S[3..5] = \text{SUS}^0_5 \), or \( S[1..3] \), which is a right extension of \( \text{SUS}^0_1 = S[1] \). (2) \( \text{SUS}^0_5 = S[4..5] = \text{SUS}^0_6 \). (3) \( \text{SUS}^0_0 = S[4..6] \), which is a right extension of \( \text{SUS}^0_4 = S[4..5] \). (4) \( \text{SUS}^1_4 = S[3..5] = \text{SUS}^1_5 \). (5) \( \text{SUS}^1_6 = S[3..6] \), which is a right extension of \( \text{SUS}^1_3 \).

The next lemma further says that if \( \text{SUS}^k_p \) is an extension of an \( k \)-mismatch LSUS, \( \text{SUS}^k_p \) can be quickly obtained from \( \text{SUS}^k_{p-1} \).
Lemma 2.4. For any \( k \) and \( p \), if \( \text{SUS}_k^p = S[i..p] \) and \( i + |\text{LSUS}_k^p| - 1 < p \), i.e., \( \text{SUS}_k^p \) is a right extension (through position \( p \)) of \( \text{LSUS}_k^p \), then the following must be true: (1) \( p > 2 \); (2) the rightmost character of \( \text{SUS}_k^{p-1} \) is \( S[p-1] \); (3) \( \text{SUS}_k^p = \text{SUS}_k^{p-1} S[p] \), the substring \( \text{SUS}_k^{p-1} \) appended by the character \( S[p] \).

Proof. If \( \text{SUS}_k^p \) is a right extension (through position \( p \)) of a \( k \)-mismatch \( \text{LSUS}_k \), it is certain that \( p > 1 \), because \( \text{SUS}_k^1 = \text{LSUS}_k^1 \), which already exists (Lemma 2.1).

Because \( \text{SUS}_k^p \) is a right extension (through position \( p \)) of a \( k \)-mismatch \( \text{LSUS} \), we have \( \text{SUS}_k^p = S[i..p] \) for some \( i < p \), and \( \text{LSUS}_k^j = S[i..j] \) for some \( j < p \). We also know \( S[i..p-1] \) is \( k \)-mismatch unique, because the \( k \)-mismatch unique substring \( S[i..j] \) is a prefix of \( S[i..p-1] \). Note that any substring starting from a position before \( i \) and covering position \( p - 1 \) is longer than the \( k \)-mismatch unique substring \( S[i..p-1] \), so \( \text{SUS}_k^{p-1} \) must be starting from a position between \( i \) and \( p - 1 \), inclusive. Next, we show \( \text{SUS}_k^{p-1} \) actually must start at position \( i \).

The fact that \( \text{SUS}_k^p = S[i..p] \) implies \( |\text{LSUS}_k^i| \geq |\text{SUS}_k^p| = p - i + 1 \) for every \( t = i + 1, i + 2, \ldots, p \); otherwise, rather than \( S[i..p] \), any one of these \( \text{LSUS}_k \) whose size is smaller than \( p - i + 1 \) would be a better choice for \( \text{SUS}_k^p \). That means, any \( k \)-mismatch unique substring starting from \( t = i + 1, i + 2, \ldots, p - 1 \) has a length at least \( p - i + 1 \). However, \( |S[i..p-1]| = p - i < p - i + 1 \) and \( S[i..p-1] \) is \( k \)-mismatch unique already and covers position \( p - 1 \) as well, so \( S[i..p-1] \) is the only choice for \( \text{SUS}_k^{p-1} \). This also means \( \text{SUS}_k^p \) is indeed the substring \( \text{SUS}_k^{p-1} \) appended by the character \( S[p] \). □

3. The high-level picture

In this section, we present an overview of our in-place framework for finding both the exact and approximate SUS. The framework is composed of three stages, where all computation happens in the place of three arrays, \( A \), \( B \), and \( C \), each of size \( n \). \( A \) and \( B \) are arrays of \( \lceil \log_2(n) \rceil \)-bit integers, whereas \( S \) contains the input string. The following table summarizes the roles of \( A \) and \( B \) at different stages by showing the end of each stage.

<table>
<thead>
<tr>
<th>Stages</th>
<th>( A[i] )</th>
<th>( B[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Used as a temporary workspace for stage 1.</td>
<td>Ending position of ( \text{LSUS}_k^i ), if ( \text{LSUS}_k^i ) exists; otherwise, ( \text{NIL} ).</td>
</tr>
<tr>
<td>2</td>
<td>The largest ( j ), such that ( \text{LSUS}_k^j ) is an ( \text{LSUS}_k ), if ( \text{LSUS}_k^j ) exists; otherwise, ( \text{NIL} ).</td>
<td>Ending position of ( \text{LSUS}_k^i ), if ( \text{LSUS}_k^i ) exists; otherwise, ( \text{NIL} ).</td>
</tr>
<tr>
<td>3</td>
<td>Start position of the rightmost ( \text{SUS}_k^i ).</td>
<td>Ending position of the rightmost ( \text{SUS}_k^i ).</td>
</tr>
</tbody>
</table>

**Stage 1** (Section 4). We take the array \( S \) that stores the input string as input to compute \( \text{LSUS}_k^i \) for all \( i \), in the place of \( A \) and \( B \). At the end of the stage, each \( B[i] \) stores the ending position of \( \text{LSUS}_k^i \), if \( \text{LSUS}_k^i \) exists. Since each existing \( \text{LSUS}_k^i \) has its start position fixed at \( i \), at the end of stage 1, each existing \( \text{LSUS}_k^i = S[i..B[i]] \). For those non-existing \( k \)-mismatch LSUSes, we assign \( \text{NIL} \) to the corresponding \( B \) array elements. The time cost of this stage is \( O(n) \) exact SUS finding (\( k = 0 \)), and is \( O(n^2) \) for approximate SUS finding, for any \( k \geq 1 \).

**Stage 2** (Section 5). Given the array \( B \) (i.e., the \( k \)-mismatch \( \text{LSUS}_k \) array of \( S \)) from stage 1, we compute the rightmost \( \text{SLS}_k^i \), the rightmost shortest \( \text{LSUS}_k \) covering position \( i \), for all \( i \), in the place of \( A \) and \( B \). At the end of stage 2, each \( A[i] \) stores the largest \( j \), such that \( \text{LSUS}_k^j \) is an \( \text{LSUS}_k \), i.e., the rightmost \( \text{SLS}_k^i = S[A[i]..B[A[i]]] \), if \( \text{SLS}_k^i \) exists; otherwise, we assign \( A[i] = \text{NIL} \). Array \( B \) does not change during stage 2. The time cost of this stage is \( O(n) \), for any \( k \geq 0 \).

**Stage 3** (Section 6). Given \( A \) and \( B \) from stage 2, we compute \( \text{SUS}_k^i \), for all \( i \), in the place of \( A \) and \( B \). At the end of stage 3, each \( (A[i], B[i]) \) pair stores the start and ending positions of the rightmost \( \text{SUS}_k^i \), i.e., \( \text{SUS}_k^i = S[A[i]..B[i]] \). The time cost of this stage is \( O(n) \), for any \( k \geq 0 \).

4. Finding \( k \)-mismatch SUS

The goal of this section is that, given the input string \( S \) and two integer arrays \( A \) and \( B \), we want to work in the place of \( A \) and \( B \), such that \( B[i] \) stores the ending position of \( \text{LSUS}_k^i \) for all existing \( \text{LSUS}_k^i \); otherwise, \( B[i] \) is assigned \( \text{NIL} \). We take different approaches in finding the exact SUS (\( k = 0 \)) and approximate SUS (\( k \geq 1 \)).

4.1. Finding exact SUS (\( k = 0 \))

Lemma 4.1. (Lemma 7.1 in [9]). Given the string \( S \) of size \( n \), drawn from an alphabet of size \( \sigma \), we can construct the suffix array \( \text{SA} \) of \( S \) in \( O(n) \) time, using \( n + \sigma \) words plus \( n \) bytes, where the space of \( n \) bytes stores \( S \), the space of \( n \) words stores \( \text{SA} \), and the extra space of \( \sigma \) words is used as the workspace for the run of the SA construction algorithm.

Given the input string \( S \), we first use the \( O(n) \)-time suffix array construction algorithm from [9] to create the \( \text{SA} \) of \( S \), where the array \( A \) is used to store the \( \text{SA} \) and the array \( B \) is used as the workspace. Note that \( \sigma \leq n \) is always true, because
otherwise we will discard from the alphabet those characters that do not appear in the string. After SA (stored in A) is constructed, we can easily spend another $O(n)$ time to create the rank array RA of S (stored in B): $RA[SA[i]] \leftarrow i$ (i.e., $B[A[i]] \leftarrow i$), for all $i$. Next, we use and work in the place of $A$ (i.e., SA) and $B$ (i.e., RA) to compute the ending position of each existing $LSUS_i^0$ and store the result in $B[i]$, using another $O(n)$ time.

**Definition 4.1.**

\[ x_i = \begin{cases} \lfloor LCP^0(S[i...n], S[SA[RA[i] - 1]...n]) \rfloor, & \text{if } RA[i] > 1 \\ 0, & \text{otherwise} \end{cases} \]

\[ y_i = \begin{cases} \lfloor LCP^0(S[i...n], S[SA[RA[i] + 1]...n]) \rfloor, & \text{if } RA[i] < n \\ 0, & \text{otherwise} \end{cases} \]

That is, $x_i$ ($y_i$, resp.) is the length of the longest common prefix of $S[i...n]$ and its lexicographically preceding (succeeding, resp.) suffix, if the preceding (succeeding, resp.) suffix exists.

**Fact 4.1.** For every string position $i$, $1 \leq i \leq n$:

$$LSUS_i^0 = \begin{cases} S[i...i + \max(x_i, y_i)], & \text{if } i + \max(x_i, y_i) \leq n \\ \text{not existing}, & \text{otherwise.} \end{cases}$$

First, observe that in the sequence of $x_i$’s, if $x_i > 0$, then $x_{i+1} \geq x_i - 1$ must be true, because at least $S[SA[RA[i] - 1] + 1...n]$ can be the lexicographically preceding suffix of $S[i + 1...n]$, and they share the leading $x_i - 1$ characters. That means, when we compute $x_{i+1}$, we can skip over the comparisons of the first $x_i - 1$ pair of characters between $S[i + 1...n]$ and its lexicographically preceding suffix. It follows that, given the SA and RA of S and using the above observation, we can compute the sequence of $x_i$’s in $O(n)$ time. Notice that this is also the idea behind the linear-time longest common prefix array construction [10], which was later further explored in the permuted LCP array construction [11]. Using the similar observation, we can compute the sequence of $y_i$’s in $O(n)$ time, provided that S and its SA and RA are given.

Second, since we can compute the sequences of $x_i$’s and $y_i$’s in parallel (i.e., compute the sequence of $(x_i, y_i)$ pairs), we can use **Fact 4.1** to compute the sequence of $LSUS_i^0$ in $O(n)$ time. Further, since RA[j] is used only for retrieving the lexicographically preceding and succeeding suffixes of $S[i...n]$ when we compute the pair $(x_i, y_i)$, we can store each computed $LSUS_i^0$ (indeed, $i + \max(x_i, y_i)$, the ending position of $LSUS_i^0$) in the place of RA[i] (i.e., $B[i]$). In the case $i + \max(x_i, y_i) > n$, meaning $LSUS_i^0$ does not exist, we will assign $\text{NIL}$ to RA[j] (i.e., $B[j]$) for all $j \geq i$ (**Lemma 2.1**). The overall time cost for computing the sequence of $LSUS_i^0$ is thus $O(n)$, yielding the following lemma.

**Lemma 4.2.** Given the character array $S$ of size $n$ that stores the input string, and the integer arrays $A$ and $B$, each of size $n$, we can work in the place of $S$, $A$, and $B$, using $O(n)$ time, such that at the end of the computation, $S$ does not change, $B[i]$ stores the ending position of $LSUS_i^0$, if $LSUS_i^0$ exists (otherwise, $B[i] = \text{NIL}$).

**Algorithm 1** shows the pseudocode of the procedure described in this subsection.

### 4.2. Finding approximate $LSUS_i^0$ ($k \geq 1$)

**Definition 4.2.** For a particular string position $p$ in $S$ and an integer $k$, $0 \leq k \leq n - 1$, the $k$-mismatch left-bounded longest repeat (LLR) starting at position $p$, denoted as $LLR_p^k$, is a $k$-mismatch repeat $S[p...j]$, such that either $j = n$ or $S[p...j + 1]$ is $k$-mismatch unique.

**Fact 4.2.** (1) If $|LLR_p^k| < n - p + 1$, i.e., the ending position of $LLR_p^k$ is less than $n$, then $LSUS_p^k = S[p...p + |LLR_p^k|]$, the substring of $LLR_p^k$ appended by the character following $LLR_p^k$. (2) Otherwise, $LSUS_p^k$ does not exist.

Our high-level strategy for finding $LSUS_i^0$ for all $i$ is as follows. We first find $LLR_i^k$ for all $i$. Then we use **Fact 4.2** to find each $LSUS_i^0$ from $LLR_i^k$: If $LLR_i^k$ does not end on position $n$, we will extend it for one more character on its right side and make the extension to be $LSUS_i^0$; otherwise, $LSUS_i^0$ does not exist. Next, we explain how to find $LLR_i^k$, for all $i$.

Clearly, $|LLR_i^k| = \max(|LCP^k(S_i, S_j)|, j \neq i)$, for all $i$. The way we calculate $|LLR_i^k|$ for all $i$ is simply to let every pair of two distinct suffixes to be compared with each other. In order to do so, we work over $n - 1$ phases, named as $\mathcal{P}_i$ through $\mathcal{P}_{n-1}$. On a particular phase $\mathcal{P}_i$, we compare suffixes $S_i$ and $S_{i-\delta}$ for all $i = n, n - 1, \ldots, \delta + 1$. Obviously, over these $n - 1$ phases, every pair of distinct suffixes have been compared with each other exactly once. Over these $n - 1$ phases, we simply record in $B[i]$, which is initialized to be 0, the length of the longest $k$-mismatch LCP that each suffix $S_i$ has seen when compared with any other suffixes. Next, we explain the details of a particular phase $\mathcal{P}_\delta$.

On a particular phase $\mathcal{P}_\delta$, $1 \leq \delta \leq n - 1$, we compare suffixes $S_i$ and $S_{i-\delta}$ for all $i = n, n - 1, \ldots, \delta + 1$. When we compare $S_i$ and $S_{i-\delta}$, we store in $A[1..k+1]$, which is initialized to be empty at the beginning of each phase, the leftmost
Algorithm 1: Finding exact LSUS.

Input: String $S$ and integer arrays $A$ and $B$, each of size $n$.
Output: $S$ does not change. $B[i] = \text{ending position of } LSUS_k$, if $LSUS_k$ exists; otherwise, $B[i] = \text{NIL}$.

1. Create the SA of $S$ using the suffix array construction algorithm from [9], where array $A$ is used to save the resulting SA and $B$ is used as the workspace for the run of the algorithm.

2. for $i = 1$ to $n$ do
   3. \[ RA[SA[i]] \leftarrow i; \]
   4. \[ /* i.e., SA = A, RA = B, and B[SA[i]] \leftarrow i */ \]
5. \[ /* From here on, A and SA are the same physical array. B, RA, and LSUS are the same physical array. */ \]
6. $x \leftarrow 0$; $y \leftarrow 0$;
7. for $i = 1, 2, \ldots, n$ do
   8. if $RA[i] > i$ then
      9. \[ j \leftarrow SA[RA[i] - 1]; \]
      10. /* Calculate the length of the $0$-mismatched LCP between $S[j..n]$ and its lexicographically preceding suffix. */
      11. while $S[i + x] = S[j + x]$ do $x \leftarrow x + 1$;
      12. \[ */ \]
   13. else $x \leftarrow 0$;
14. if $RA[i] < n$ then
   15. \[ j \leftarrow SA[RA[i] + 1]; \]
   16. /* Calculate the length of the $0$-mismatched LCP between $S[i..n]$ and its lexicographically succeeding suffix. */
   17. while $S[i + y] = S[j + y]$ do $y \leftarrow y + 1$;
   18. \[ */ \]
   19. else $y \leftarrow 0$;
20. if $i + \max(x, y) \leq n$ then
   21. \[ LSUS[i] \leftarrow i + \max(x, y); \]
   22. /* ending position of $|LSUS|$ */
   23. \[ */ \]
24. else /* LSUS does not exist. Early stop. */
25. \[ for j = i \ldots n \text{ do } LSUS[j] \leftarrow \text{NIL}; \]
26. \[ Break; \]
27. if $x > 0$ then $x \leftarrow x - 1$;
28. if $y > 0$ then $y \leftarrow y - 1$;
29. \[ /* */ \]

mismatched $k + 1$ positions in $S_i$. We will see later how to update $A[1..k + 1]$ efficiently over the progress of a particular phase and use it to update the $B$ array.

We treat $A[1..k + 1]$ as a circular array, i.e., $i - 1 = k + 1$ when $i = 1$, and $i + 1 = 1$ when $i = k + 1$. Let $\text{size}$, which is initialized to be $0$ at the beginning of each phase, denote the number of mismatched positions being stored in $A[1..k + 1]$ so far in $P_k$. We can describe the work of phase $P_k$, inductively, as follows.

1. We compare $S_n$ and $S_{n-k}$ by only comparing $S[n]$ and $S[n - \delta]$, since $S_n = S[n]$.
   (a) If $S[n] \neq S[n - \delta]$: Store $n$ in any position in $A[1..k + 1]$; $\text{size} \leftarrow 1$.
   (b) \[ B[n] \leftarrow \max(B[n], 1); B[n - \delta] \leftarrow \max(B[n - \delta], 1). \]
2. Suppose we have finished the comparison between the suffixes $S_{i+1}$ and $S_{i+1-\delta}$, for some $i$, $\delta + 1 \leq i \leq n - 1$. The leftmost $k + 1$ mismatched positions (if existing) between them have been stored in the circular array $A[1..k + 1]$. Let $A[\text{cursor}]$ be the element that is saving the first mismatched position (if existing) between the two suffixes.
3. Next, we compare the suffixes $S_i$ and $S_{i-\delta}$ by only comparing $S[i]$ and $S[i - \delta]$, since $S_{i+1}$ and $S_{i+1-\delta}$ have been compared. Remind that $\text{cursor} - 1$ below is in its cyclic manner.
   (a) If $S[i] \neq S[i - \delta]$: $\text{cursor} \leftarrow \text{cursor} - 1$; Store $i$ in $A[\text{cursor}]$ and overwrite the old content in $A[\text{cursor}]$ if there is; $\text{size} \leftarrow \min(\text{size} + 1, k + 1)$.
   (b) \[ B[i] \leftarrow \max(B[i], n - i + 1); B[i - \delta] \leftarrow \max(B[i - \delta], n - i + 1). \]
   (c) Else: $B[i] \leftarrow \max(B[i], A[\text{cursor} - 1] - i)$; $B[i - \delta] \leftarrow \max(B[i - \delta], A[\text{cursor} - 1] - i)$. Note that $A[\text{cursor} - 1]$ is saving the $(k + 1)$th mismatched position between $S_i$ and $S_{i-\delta}$.

After the computation of all $LLR_k^L$ is finished, using the above $n - 1$ phases, each $B[i]$ is saving $\|LLR_k^L\|$. Next, we can use Fact 4.2 to convert each $LLR_k^L$ to $LSUS_k^L$ by simply checking each $B[i]$: if $i + B[i] - 1 < n$, i.e., $LLR_k^L$ does not end on position $n$, then we assign $B[i] = i + B[i]$, the ending position of $LSUS_k^L$; otherwise, we assign $B[i] = \text{NIL}$, meaning $LSUS_k^L$ does not exist.

The computation of all $LLR_k^L$ takes $n - 1$ phases and each phase clearly has no more than $n$ comparisons, giving a total of $O(n^2)$ time cost. The procedure of converting each $LLR_k^L$ to $LSUS_k^L$ spends another $O(n)$ time. Altogether, we get an $O(n^2)$-time in-place procedure for finding approximate LSUS, for any $k \geq 1$. 

Algorithm 2: Finding approximate LSUS.

Input: String $S$ and integer arrays $A$ and $B$, each of size $n$, the value of $k \geq 1$.
Output: $S$ does not change. $B[i]$ = ending position of LSUS$_i^k$, if LSUS$_i^k$ exists; otherwise, $B[i] = \text{NIL}$.

1 for $i = 1 \ldots n$ do  
  2 $B[i] \leftarrow 0$;  
  3 /* Initialization */
  4 /* We use $A[1 \ldots k + 1]$ as a circular array to save the $k+1$ most recently found mismatched positions. */
  5 capacity $\leftarrow k+1$;  
  6 /* The capacity of the circular array that records at most $k+1$ mismatched positions. */
  7 $\text{cursor} \leftarrow 1$;  
  8 /* The index of the circular array position that is saving the most recently founded mismatched position. It can be initialized to be any value from $\{1, 2, \ldots, \text{capacity}\}$. */
  9 for $\delta = 1 \ldots n - 1$ do /* $n-1$ phases */
    10 size $\leftarrow 0$;  
    11 /* The number of recorded mismatched positions in the circular array in the current phase. */
    12 for $i = n$ down to $\delta + 1$ do  
      13 /* Comparing suffixes $S_i$ and $S_{i+1}$ by comparing their leading characters, as their remaining characters have been compared in previous steps of this phase. */
      14 if $S[i] \neq S[i - \delta]$ then
        15 /* We use 1-based indexing. */
        16 $\text{cursor} \leftarrow ((\text{cursor} - 2 + \text{capacity}) \mod \text{capacity}) + 1$;  
        17 $A[\text{cursor}] \leftarrow i$;
      18 size $\leftarrow \min(\text{size} + 1, \text{capacity})$;
      19 else
        20 $B[i] \leftarrow \max(B[i], n - i + 1)$;  
        21 /* = size - i + 1 */
      22 $B[i - \delta] \leftarrow \max(B[i - \delta], n - i + 1)$;
    23 $B[i] \leftarrow \max(B[i], A[\{(\text{cursor} - 1 + k) \mod \text{capacity}) + 1\} - i)$;  
    24 /* We use 1-based indexing. */
      25 $B[i - \delta] \leftarrow \max(B[i - \delta], A[\{(\text{cursor} - 1 + k) \mod \text{capacity}) + 1\} - i)$;
      26 for $i = 1 \ldots n$ do
      27 if $B[i] = \text{size} - i + 1$ then
        28 /* LSUS$_i^k$ does not exist. */
        29 $B[i] \leftarrow \text{NIL}$;
      30 else
        31 $B[i] \leftarrow i + 1$;
      32 /* The ending position of LSUS$_i^k$. */
    33 end for
  18 end for

Lemma 4.3. Given the character array $S$ of size $n$ that stores the input string, the integer arrays $A$ and $B$, each of size $n$, and the value of integer $k \geq 1$, we can work in the place of $S$, $A$, and $B$, using $O(n^2)$ time, such that at the end of the computation, $S$ does not change, $B[i]$ stores the ending position of LSUS$_i^k$, i.e., LSUS$_i^k = S[i..B[i]]$, if LSUS$_i^k$ exists; otherwise, $B[i] = \text{NIL}$.

Algorithm 2 shows the pseudocode of the procedure described in this subsection.

5. Finding $k$-mismatch SLS

Now we are given the array $B$, where each $B[i]$ stores the ending position of LSUS$_i^k$ if LSUS$_i^k$ exists and \text{NIL} otherwise. In this section, we want to work in the place of $A$ and $B$, such that in the end of computation: $A[i]$ stores $j$, such that LSUS$_j^k$ is the rightmost SLS$_j^k$, if such $j$ exists; otherwise, $A[i] = \text{NIL}$. That means, in the end of this section, the rightmost SLS$_j^k = S[A[i]..B[A[i]]]$, if SLS$_j^k$ exists; otherwise, $A[i] = B[i] = \text{NIL}$.

Recall that some $k$-mismatch LSUS may not exist and some positions may not be covered by any $k$-mismatch LSUS (see the examples after Definition 2.2). Further, due to Lemmas 2.1 and 2.2, we know such positions that are not covered by any $k$-mismatch LSUS must comprise a continuous chunk on the right end of string $S$.

Definition 5.1. Let LSUS$_i^k$, $1 \leq r \leq n$, be the rightmost existing $k$-mismatch LSUS of the input string $S$. Let $z$, $1 \leq z \leq n$, be the rightmost string position that is covered by any $k$-mismatch LSUS of the string $S$.

Again, due to Lemmas 2.1 and 2.2, it is trivial to find the values of $r$ and $z$ in $O(n)$ time: scan array $B$ (i.e. LSUS array) from right to left, and stop when seeing the first non-\text{NIL} $B$ array element, which is exactly $B[r]$, then $z = B[r]$. If $z < n$, we can then simply set $A[i] = \text{NIL}$ for all $i > z$. Recall that $B[i] = \text{NIL}$ already for all $i > r$ from stage 1. In the rest of this section, we only need to work with the two subarrays $A[1..z]$ and $B[1..z]$, wanting to make $A[i]$ to be the start position of the rightmost SLS$_j^k$, for all $i \leq z$.
Let $B[1..z]$ and an integer $r$, $1 \leq r \leq z$, be the input, where (1) $B[1..r]$ is of monotonically nondecreasing integers (Lemma 2.2), with $i \leq B[i]$, (2) $B[r+1..z]$ are all NILs, if $r < z$, and (3) $B[r] = z$.

We can use each $B[i]$, $i \leq r$, as a compact representation of the interval $I_i = (i, B[i])$. Let $I = \{ I_i \mid i \in [1..r] \}$, and $\ell_i = |B[i] - i + 1|$ be the length of $I_i$. Let $A[1..z]$ be an output array such that $A[j] = i$, where $I_i$ is the rightmost shortest interval in $I$ that covers $j$.

To illustrate the ideas and concepts that we will present in the rest of this section, let us use the following as a running example, where $r = 9$, $z = 15$, and $n = 17$ (we add $(0, B[0]) = (0, 0)$ as a sentinel).

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B[i]$</td>
<td>0 3 4 7 8 10 10 10 11 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\ell_i$</td>
<td>0 3 5 5 6 5 4 4 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{pre}(i)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_i$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4 2 2 2</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\max t_i^{-1}$</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A[i]$</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

**Definition 5.2.** For an interval $I_i$, we define the effective covering region with respect to the previous intervals $I_{<i} = \{ I_k \mid k < i \}$ to be $[t_i, B[i]]$ where

$$t_i = \max \left\{ i, \max \{ B[k] + 1 \mid I_k \text{ is shorter than } I_i, k < i \} \right\}.$$  

We call $t_i$ the starting point of the effective covering region of $I_i$.

The effective covering region of $I_i$ is exactly those regions that would set $I_i$ as the answer, provided that all the intervals $I_{<i}$ before $I_i$ are present, and all the intervals $I_{>i} = \{ I_k \mid k > i \}$ are absent.

We next define $t_i^{-1}$ as a list\footnote{In actual run, $t_i^{-1}$ stores the largest number in that list, as we will see more clearly later.}, such that $j \in t_i^{-1}$ if and only if $I_j = I_i$. Observe that since $t_i \geq i$ by definition, any value $j$ in $t_i^{-1}$ must have $j \leq i$, and the effective region of $I_j$ must cover $i$.

**Lemma 5.1.** For $i = 1, 2, \ldots, z$:

$$A[i] = \max \bigcup_{k=1}^{i} t_k^{-1} = \max \{ A[i-1], \max t_i^{-1} \}.$$  

**Proof.** Let $j = \max \bigcup_{k=1}^{i} t_k^{-1}$. This means that for the effective region of any $I_h$, with $h > j$, none of them covers $i$. Next, observe that $I_j$ must cover $i$; otherwise, for all the intervals $I_h$ with $h < j$, we have $B[h] \leq B[j] < i$, so that none of them can cover $i$, and thus a contradiction occurs. Finally, we show that for those $h < j$, $I_h$ can be pruned\footnote{In the computation of each array element $A[i]$ in Section 5, when we say an interval $I_h$ can be pruned, it means that we are certain that $h$ will not be the answer for $A[i]$.} by $I_j$, thus implying that $A[i] = j$ is a correct answer.

Consider all those $h$ with $h < j$:

1. If $I_h$ is longer than $I_j$, $I_h$ can be pruned away directly.
2. Else, if $I_h$ and $I_j$ have equal length, $I_h$ can be pruned away also, regardless of its coverage on $i$, since we pick the rightmost shortest interval that covers $i$.
3. Else, $h$ must appear in $\bigcup_{k=1}^{i} t_k^{-1}$. By the definition of $t_j$, we have $B[h] < t_j \leq i$; thus, $I_h$ does not cover $i$, and can be pruned away.

Thus, the first equality in the lemma follows, while the second equality in the lemma is trivial once we have the first equality. □

**Lemma 5.2.** Suppose that all $t_i$, $1 \leq i \leq r$, can be generated incrementally in $O(n)$ time. Then, we can obtain all $\max t_i^{-1}$, $1 \leq i \leq z$, in $O(n)$ time.

**Proof.** We examine each $t_i$, $i = 1, 2, \ldots, r$, and write $i$ at entry $t_i = j$ of the $t^{-1}$ array; if such an entry contains a value $i'$ already, we simply overwrite $i'$ with the latter $i$. □
Indeed, we may scan \( t_i \) from right to left, i.e., \( i = r, r - 1, \ldots, 1 \), and update max \( t_i^{-1} \) as we proceed. Firstly, if \( t_i > i \), we set \( t_i^{-1} = \text{NIL} \). Next, let \( j = t_i \) (whose value is at least \( i \)), and we check if \( t_j^{-1} \) is defined: If not, simply set \( t_j^{-1} = i \); otherwise, no update is needed.

The advantage of the ‘right-to-left’ approach is that we can construct \( t_i^{-1} \) in-place, by re-using the memory space of \( t_i \). To see why it is so, by the time we need to update a certain entry \( j = t_i \) at step \( i \), the information \( t_i \) has been used (and will never be used), so that we can safely overwrite the original entry, storing \( t_j \), to store \( t_j^{-1} \) instead. This gives the following corollary.

**Corollary 5.1.** Suppose that all \( t_i \)'s are generated, and are stored in a certain array \( A[1..z] \). Then, we can obtain max \( t_i^{-1} \) for all \( i \)'s, in-place, by storing the results in the same array \( A[1..z] \); the time cost is \( O(n) \).

Our goal is to make our algorithm in-place. Suppose that we can have in-place incremental generation of \( t_i \). Then, by the above lemma, we may store max \( t_i^{-1} \) temporarily at \( A[i] \); afterwards, by the second equality of Lemma 5.1, we can compute the correct output \( A \) by a simple scan of \( A \) from left to right.

Thus, to make the whole process in-place, it remains to show how \( t_j \) can be computed in \( O(n) \) time, in-place. For this, we define \( \text{pred}[i] \) to be the largest \( j \) (if it exists) such that \( j < i \) and length of \( I_j \) is shorter than \( I_i \). It is easy to check that if \( \text{pred}[i] = j \) is defined, then \( t_i = \max \{ B[j] + 1, i \} \) (and \( t_i = i \) otherwise).

Moreover, \( \text{pred}[i] \) for all \( i \)'s can be computed incrementally, with a way analogous to the construction of the failure function in KMP algorithm [12]: we check \( \text{pred}[i - 1], \text{pred}[\text{pred}[i - 1]], \text{pred}[\text{pred}[\text{pred}[i - 1]]], \) and so on, until we obtain \( j \) in the process such that \( I_j \) is shorter than \( I_i \), and set \( \text{pred}[i] := j \).

If such \( j \) does not exist, we set \( \text{pred}[i] = \text{NIL} \). The running time is bounded by \( O(n) \).

This gives the following \( O(n) \)-time in-place algorithm (where \( B \) is read-only):

1. Compute \( \text{pred}[i], i = 1, 2, \ldots, r \), and store this in \( A[i] \). Note that this step requires the length information of the intervals of \( I_i \), which can be obtained in \( O(1) \) time, on the fly, from \( B[i] \).
2. Scan \( A[1..r] \) (i.e., \( \text{pred} \)) incrementally, and obtain \( t_i \) from the above discussion. The value of \( t_i \) is stored in \( A[i] \). Note that this step requires the access to the original \( B \).
3. Scan \( A[1..r] \) (i.e., \( t_j \)) from right to left, and obtain max \( t_j^{-1} \) decrementally (stored in \( A[i] \)) by Corollary 5.1.
4. Scan \( A[1..z] \) (i.e., max \( t_i^{-1} \)) incrementally (\( i = 1, 2, \ldots, z \)), and obtain the desired \( A[i] \) by the second equality in Lemma 5.1.

**Algorithm 3** shows the pseudocode described in this section.

### 6. Finding \( k \)-mismatch SUS

Now we have array \( A \), where \( A[i] = j \), such that \( \text{LSUS}_i^k \) is the rightmost \( \text{SLS}_i^k \), if position \( i \) is covered by any \( k \)-mismatch SUS; otherwise, \( A[i] = \text{NIL} \). Note that \( A[i] = j \) is recording the start position of the rightmost \( \text{SLS}_i^k \) already, because \( \text{LSUS}_i^k \) starts on position \( j \). We also have array \( B \), where \( B[i] = i + | \text{LSUS}_i^k | - 1 \), the ending position of \( \text{SUS}_i^k \), if \( \text{LSUS}_i^k \) exists; otherwise, \( B[i] = \text{NIL} \).

#### Step 1

We want to transform \( A \) and \( B \), such that each \( (A[i], B[i]) \) pair stores the start and ending positions of \( \text{SLS}_i^k \), if \( \text{SLS}_i^k \) exists; otherwise, we set \( (A[i], B[i]) = (\text{NIL}, \text{NIL}) \). Since each \( A[i] \) is already recording the start position of \( \text{SLS}_i^k \) already, as we have explained at the beginning of this section, we only need to make changes to array \( B \). We first set \( B[i] = \text{NIL} \) for all \( i > z \) (Definition 5.1). Then, we scan array \( B \) from right to left, starting from position \( z \) through 1, and set each \( B[i] = B[A[i]] \), the ending position of the rightmost \( \text{SLS}_i^k \). Because the leftmost position that any existing \( \text{LSUS}_i^k \) can cover is position \( i \), we know \( A[i] < i \) and we no longer need \( B[i] \) (i.e., the information of \( \text{LSUS}_i^k \)) after \( \text{SLS}_i^k \) is computed. Therefore, it is safe to record \( \text{SLS}_i^k \) by overwriting \( B[i] \) by \( B[A[i]] \) (i.e., the ending position of \( \text{SLS}_i^k \)), in this right-to-left scan.

---

5. For each \( j < j \), if \( I_j \) covers \( i \), \( I_j \) would also cover \( i \); in such a case, \( B[j] + 1 \geq B[j'] + 1 \). For each \( j' \in \text{pred}[i], i - 1 \), \( I_{j'} \) is longer than \( I_i \).

6. Intuitively, \( \text{pred} \) defines the shortcuts so that we can skip some intervals in \( I_{<j} \) to compute \( t_i \).
Algorithm 3: Finding SLS (exact or approximate).

Input: Integer arrays $A$ and $B$, each of size $n$. Each $B[i]$ stores the ending position of $LSUS_j^k$, if $LSUS_j^k$ exists; $NIL$, otherwise.

Output: Array $B$ does not change. Each $A[i] = j$, such that $LSUS_j^k$ is the rightmost $SLS_j^k$, if $SLS_j^k$ exists; otherwise, $A[i] = NIL$.

```c
/* Find the index of the rightmost existing k-mismatch LSUS. */
for r = n ... 1 do
  if B[r] ≠ NIL then break;
  /* Compute the pred array, using the memory space of A array. pred[i] is the largest j, such that j < i and |LSUS_j^k| < |LSUS_i^k|, if such j exists; otherwise pred[i] = NIL. If LSUS_j^k does not exist, pred[i] = NIL also. From here on, pred and A are the same physical array. */
  if r < n then
    for i = r ... n do
      pred[i] ← NIL;
      /* Positions that do not have k-mismatch LSUS. */
  pred[1] ← NIL;
  for i = 2 ... r do
    if pred[i] = NIL then t[i] ← i;
  else t[i] ← max(B[pred[i]] + 1, i);
  /* Compute the t array, using the memory space of A array. t[i] is the start position of the effective region of LSUS_j^k, if $LSUS_j^k$ exists; NIL, otherwise. From here on, t and A are the same physical array. */
  for i = r ... 1 do
    if t^{-1}[t[i]] = NIL then t^{-1}[t[i]] ← i
    if i < t[i] then t^{-1}[i] ← NIL;
    /* Enable us to update this place in the future when needed. */
  /* Compute SLS array using the memory space of array A. SLS[i] = j, such that LSUS_j^k is the rightmost SLS_j^k, if SLS_j^k exists; NIL, otherwise. From here on, SLS and A are the same physical array. */
  SLS[1] ← 1;
  for i = 2 ... |A| do
    SLS[i] ← max(SLS[i-1], t^{-1}[t[i]]);
```

Step II We use arrays $A$ and $B$ to calculate $SUS_j^k$ for each $i$ and save the result in the place of $A$ and $B$, i.e., each $(A[i], B[i])$ pair stores the start and ending position of $SUS_j^k$. Because of Lemmas 2.3 and 2.4, we can use arrays $A$ and $B$ to compute each $SUS_j^k$ inductively, as follows:

1. $SUS_j^k = LSUS_j^k = S[A[i]..B[i]]$.
2. For $i = 3, 4, ...$, we compute $SUS_j^k$:
   (a) If $(A[i], B[i]) = (NIL, NIL)$, meaning $SLS_j^k$ does not exist, we set $SUS_j^k$ to be $SUS_{j-1}^k$ appended by the character $S[i]$, i.e., $SUS_j^k = S[A[i-1]..B[i-1]+1]$, and save $SUS_j^k$ by setting $(A[i], B[i]) = (A[i-1], B[i-1]+1)$;
   (b) Else, if $SUS_{j-1}^k$ ends at position $i-1$ and $SUS_{j-1}^k S[i] = S[A[i-1]..B[i-1]+1]$ is shorter than $SLS_j^k = S[A[i]..B[i]]$, we set $(A[i], B[i]) = (A[i-1], B[i-1]+1)$;
   (c) Else, $SUS_j^k = SLS_j^k$ and thus we leave $A[i]$ and $B[i]$ unchanged.

**Lemma 6.1.** Given arrays $A$ and $B$:
Algorithm 4: Finding SUS (exact or approximate).

**Input:** Integer arrays $A$ and $B$, each of size $n$. (1) $A[i] = j$, such that $LSUS^k_j$ is the rightmost $SLS^k_j$, if $SLS^k_j$ exists; otherwise, $A[i] = NIL$. (2) $B[i]$ is the ending position of $LSUS^k_j$, if $LSUS^k_j$ exists; otherwise, $B[i] = NIL$.

**Output:** Each $(A[i], B[i])$ pair represents the start and ending positions of $SUS^k_i$.

```
1 for i = n ... 1 do
2   if $B[i] \neq NIL$ then
3     $z \leftarrow i + B[i] - 1$;
4     break;
5   end if
6 for i = z + 1 ... n do
7     $B[i] \leftarrow NIL$;
8   end for
9 for i = z ... 1 do
10    $B[i] \leftarrow B[A[i]]$;
11   end for
12 end for
```

- $A[i] = j$, such that $LSUS^k_j$ is the rightmost $SLS^k_j$, if $SLS^k_j$ exists; otherwise, $A[i] = NIL$;
- $B[i] = i + |LSUS^k_j| - 1$, the ending position of $LSUS^k_j$, if $LSUS^k_j$ exists; otherwise, $B[i] = NIL$.

We can work in the place of $A$ and $B$, using $O(n)$ time, such that, in the end of computation, each $(A[i], B[i])$ stores the start and ending positions of $SUS^k_i$, i.e., $SUS^k_i = S[A[i]..B[i]]$, $i = 1, 2, \ldots, n$.

By concatenating the claims in Lemmas 4.2, 4.3, 5.3, and 6.1, we get the final result.

**Theorem 6.1.** Given the array $S$ of size $n$ that stores the input string, two empty integer arrays $A$ and $B$, each of size $n$, and the value of integer $k \geq 0$, we can work in the place of arrays $S$, $A$, and $B$, using a total of $O(n)$ time for $k = 0$ and $O(n^2)$ time for any $k \geq 1$, such that in the end of computation, $S$ does not change, each $(A[i], B[i])$ pair represents the start and ending positions of the rightmost $SUS^k_i$, i.e., $SUS^k_i = S[A[i]..B[i]]$.

Algorithm 4 shows the pseudocode described in this section.

7. Experiments

We implemented our proposal named IPSUS in C\(^7\) using the libdivsufsort\(^8\) library for the suffix array construction. Our experiments were conducted on a MacBook Pro computer that has 16 GB 1600 MHz DDR3 main memory and an Intel 2.2 GHz Core i7 CPU with 256 KB L2 cache (per core) and 6 MB L3 cache. We used the text collection from Pizza&Chili corpus\(^9\) as the strings by taking the first $n$ characters from the largest dblp.xml, dna, English, and protein files available there, where $n$ is the string size involved in the experiments.

We considered all other existing solutions for SUS finding that have implementations, including RSUS [2], OSUS [6], and IXSSUS [7]. According to the results reported in [7], RSUS is way slower and significantly more space consuming than OSUS and IXSSUS, due to its quadratic time complexity and its use of the suffix tree data structure. It is also observed in [7] that, in finding one (exact) SUS for every position, OSUS and IXSSUS have negligible differences in their processing speed, but OSUS uses 4 times more memory space than IXSSUS. Because IXSSUS is currently the best all-around solution for exact SUS finding, in this experimental study, we focused on the comparison between IXSSUS and our proposal, using the same set of data from the Pizza&Chili corpus. Note that IXSSUS was also implemented in C and uses libdivsufsort library for the suffix array construction.

\(^7\) http://penguin.ewu.edu/~bojianxu/publications.
\(^8\) https://code.google.com/p/libdivsufsort.
\(^9\) http://pizzachili.dcc.uchile.cl/texts.html.
Time efficiency (exact SUS finding)  Fig. 1 shows the comparison of the processing speeds of IKXSUS and our proposal in finding one exact SUS for every string position for four different types of strings. Both solutions are clearly showing a linear time complexity for all four types of data, but our proposal is consistently much faster than IKXSUS with an average ratio of 1:1.6, i.e., IKXSUS has to spend 1.6 seconds on the workload that can be finished by IPSUS using 1 second.

Time efficiency (approximate SUS finding)  Fig. 2 shows clearly the quadratic processing time cost of our proposal for finding SUS with mismatches from different types of strings. Note that the string sizes in these experiments are in kilobytes. Our proposal spends the same amount of time regardless the positive number of mismatches allowed. Because no prior work has considered the k-mismatch case, there is no baseline to compare in this setting.

Space efficiency  In our experiments, an integer is stored using four bytes in both IKXSUS and IPSUS. The comparison on peak memory space usage by both proposals is given in Fig. 3. Our proposal uses exactly 9n bytes peak memory for all string types and sizes, where n is the string size, validating the claim that our proposal is an in-place solution. However the peak memory usage of IKXSUS is consistently around 13.5n bytes. Note that our proposal’s 9n-byte peak memory usage is true for both exact and approximate SUS finding.

Fig. 1. The processing speed of IKXSUS and our proposal in finding one exact SUS of every location on several strings of different sizes.

Fig. 2. The processing speed of our proposal in finding one approximate SUS of every location on several strings of different sizes.
8. Note

Seemingly, throughout the presentation of the paper, each word in both the A and B arrays needs to have \(\lceil \log_2(n+1) \rceil\) bits, because we use NIL in the algorithm. However, the use of NIL is only for illustration purpose. Precisely, the use of NIL in various parts of the algorithm can be avoided as follows.

- Stage 1: \(B[i] = \text{NIL}\) can be replaced by \(B[i] = 1\), with an extra boolean bit to indicate if \(B[1] = \text{NIL}\) or not.
- Stage 2: (1) In the discussion after Lemma 5.2 and its proof, \(t_1^{-1} = \text{NIL}\) can be replaced by \(t_1^{-1} = n\), with an extra boolean bit to indicate if \(t_1^{-1} = \text{NIL}\) or not. (2) \(\text{pred}[i] = \text{NIL}\) can be replaced by \(\text{pred}[i] = n\). (3) \(A[i] = \text{NIL}\) can be replaced by \(A[i] = n\), with an extra boolean bit to indicate if \(A[n] = \text{NIL}\) or not.
- Stage 3: \(B[i] = \text{NIL}\) can be replaced by \(B[i] = 1\). For every \(i > 1\), \(B[i] = 1\) indicates \(B[i] = \text{NIL}\). \(B[1] = 1\) indicates \(B[1] = \text{NIL}\) if and only if \(A[1] = \text{NIL}\), which was already handled at Stage 2.

Also, throughout the presentation of this paper, we use 1-based indexing arrays. However, it is trivial to change them to be 0-based indexing, so that the \(\lceil \log_2 n \rceil\)-bit word is large enough to encode the integer numbers \(\{0, 1, \ldots, n – 1\}\).

9. Conclusion

In this paper, we revisited the exact SUS finding problem, and proposed its approximate version where mismatches are allowed, and thus extended the usage of SUS finding in subfields such as computational biology. We designed a generic in-place algorithmic framework that uses the minimum \(2n\) words, each of \(\lceil \log_2 n \rceil\) bits, plus \(n\) bytes space and can fit to find both exact and approximate \(k\)-mismatch SUS, with \(O(n)\) and \(O(n^2)\) time complexities, respectively, regardless of the value of any \(k \geq 1\). Experimental study showed our proposal is both more space saving and faster than the currently best proposal that has implementation for exact SUS finding.

A future work will be researching for a faster (and still practical) in-place algorithm for finding approximate LSUS to replace the current algorithm discussed in Section 4.2. Such new algorithm will lead to an overall faster in-place solution for approximate SUS finding. Another future work can be looking for an I/O-efficient version of our solution, which seems possible with the existing work on the I/O-efficient construction of the suffix array [13–15] and LCP array [15–17]. However, our current Algorithms 1 and 3 have random access to the relevant arrays and it is not immediately clear how to make them I/O-efficient.

References