A Practical and Efficient Algorithm for the k-mismatch Shortest Unique Substring Finding Problem*

Daniel R. Allen
Department of Computer Science
Eastern Washington University
Cheney, WA 99004, USA
dra4@eagles.ewu.edu

Sharma V. Thankachan†
Department of Computer Science
University of Central Florida
Orlando, FL 32816, USA
sharma.thankachan@ucf.edu

Bojian Xu‡
Department of Computer Science
Eastern Washington University
Cheney, WA 99004, USA
bojianxu@ewu.edu

ABSTRACT

This paper revisits the k-mismatch shortest unique substring finding problem and demonstrates that a technique recently presented in the context of solving the k-mismatch average common substring problem can be adapted and combined with parts of the existing solution, resulting in a new algorithm which has expected time complexity of $O(n \log^k n)$, while maintaining a practical space complexity at $O(kn)$, where n is the string length. When $k > 0$, which is the hard case, our new proposal significantly improves the any-case $O(n^2)$ time complexity of the prior best method for k-mismatch shortest unique substring finding. Experimental study shows that our new algorithm is practical to implement and demonstrates significant improvements in processing time compared to the prior best solution's implementation when k is small relative to n. For example, our method processes a 200KB sample DNA sequence with $k = 1$ in just 0.18 seconds compared to 174.37 seconds with the prior best solution. Further, it is observed that significant portions of the adapted technique can be executed in parallel, using two different simple concurrency models, resulting in further significant practical performance improvement. As an example, when using 8 cores, the parallel implementations both achieved processing times that are less than 1/4 that of the serial implementation, when processing a 10MB sample DNA sequence with $k = 2$. In an age where instances with thousands of gigabytes of RAM are readily available for use through Cloud infrastructure providers, it is likely that the trade-off of additional memory usage for significantly improved processing times will be desirable and needed by many users. For example, the best prior solution may spend years to finish a DNA sample of 200MB for any $k > 0$, while this new proposal, using 24 cores, can finish processing a sample of this size with $k = 1$ in 206.376 seconds with a peak memory usage of 46GB, which is both easily available and affordable on Cloud for many users. It is expected that this new efficient and practical algorithm for k-mismatch shortest unique substring finding will prove useful to those using the measure on long sequences in fields such as computational biology.

CCS CONCEPTS

• **Theory of computation** → **Pattern matching**; **Sorting and searching**; *Parallel algorithms*; • **Applied computing** → **Bioinformatics**; • **Information systems** → *Information retrieval*;

KEYWORDS

String, Shortest Unique Substring, Mismatch, Hamming Distance

ACM Reference Format:
Daniel R. Allen, Sharma V. Thankachan, and Bojian Xu. 2018. A Practical and Efficient Algorithm for the k-mismatch Shortest Unique Substring Finding Problem. In *ACM-BCB'18: 9th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics, August 29-September 1, 2018, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3233547.3233564

1 INTRODUCTION

The computer science subfield known as *string processing* focuses on the design and analysis of algorithms which process sequences of characters, commonly referred to as strings. The algorithms from this subfield find applications in many problem spaces. Use cases range from powering fast searches for a word or phrase in electronic documents on personal computing devices or the Web, to efficiently processing a body of text in a text editor or word processor application in order to provide spell-checking and syntax highlighting functionality, to finding faint patterns in DNA and protein sequences [5]. String processing has been said to form the heart of the field of computational molecular biology, where biological constructs such as DNA and proteins are abstracted to sequences of characters which can be studied independently from their complex biological environments [5].

In 2005, Haubold et al. demonstrated that the shortest unique substring (*SUS*) is a useful construct for alignment free genome comparison [6]. A *SUS* as presented by the authors is described as a substring which only occurs once in a sequence such that any reduction of its length would result in the loss of its uniqueness property. These authors presented a string processing algorithm which relies upon generalized suffix trees to detect shortest unique substrings across a set of sequences, but did not analyze the performance of the presented algorithm rigorously.

Nearly a decade later in 2013, the *SUS* finding problem was revisited by Pei et al. where the authors noted additional applications

*Authors are in alphabetical order.
†Supported in part by the U.S. National Science Foundation grant CCF-1703489.
‡Corresponding author.

ACM-BCB'18, August 29-September 1, 2018, Washington, DC, USA

ACM ISBN 978-1-4503-5794-4/18/08…$15.00
https://doi.org/10.1145/3233547.3233564

for the construct including intelligent snippet selection in document search, polymerase chain reaction primer design in molecular biology, identification of unique DNA signatures of closely related organisms, and context extraction in event analysis [11]. Here the authors present algorithms which process an input string of length n and can answer *SUS* queries, that is, they return a single *SUS* which spans over a given index of the input string. One algorithm presented uses a suffix tree and can answer a query in $O(n)$ time. Another algorithm is presented which can find a *SUS* for every index in the string in $O(n^2)$ time and subsequently can answer each query with a precomputed *SUS* value in $O(1)$ time. Both strategies require $O(n)$ space.

The following year, Tsuruta et al. presented an algorithm which calculated a *SUS* for every index of an input string in $O(n)$ time and space using suffix arrays [14]. The same year, another independent $O(n)$ time $O(n)$ space *SUS* finding algorithm was presented by İleri et al. in [9] which was demonstrated through empirical data to be significantly more space efficient in practice than the solution by Tsuruta et al. while the processing times of the two algorithms were nearly the same. Another notable work in 2014 by Hu et al. proposed use of an $O(n)$ space indexing structure which can be constructed in $O(n)$ time and can subsequently be used to answer queries for a *SUS* which contains a given substring of the input in $O(1)$ time [8].

In 2017, Hon, Thankachan, and Xu (HTX) presented a time and space optimal *SUS* finding solution in [7]. The solution has $O(n)$ time complexity for finding a *SUS* for each index in an input string, and works in the space of the two length n output arrays which in the end hold the beginning and end indices of the *SUS* found for each corresponding index in the input string. Presented experimental data indicates that the solution has significantly better time and space performance in practice than comparable existing *SUS* finding solutions.

An additional contribution of [7] was the proposal of an *approximate* version of the *SUS* finding problem where the uniqueness constraint is more strict than in the exact version of the problem. The proposed approximate version requires that the substrings be unique even allowing for up to k mismatches, which is expected to be useful for applications in subfields such as computational biology where factors like genetic mutation and experimental error make approximate string matching necessary. This concept of approximate matching has proven useful with other constructs, for example in [13] experimental results showed that increasing a similar k-mismatch parameter applied to average common substring finding lead to better results when estimating the evolutionary distance between pairs of primate genomes.

After proposing the k-mismatch *SUS* finding problem, the authors of [7] proceed to present an algorithm which solves the problem when $k > 0$, which is the hard case, for an input string of length n in $O(n^2)$ time and $O(n)$ space by performing a series of calculations and transformations in-place on two length n arrays. Notably, only one step in the series requires greater than $O(n)$ time.

Our contribution.

- This paper's primary contribution is to demonstrate how strategies presented by Thankachan et al. in [13] in the context of solving the k-mismatch average common substring problem can be adapted and applied to solve the aforementioned time-expensive step from the HTX k-mismatch *SUS* finding algorithm. The adaptation leads to a new algorithm with overall expected time complexity of $O(n \log^k n)$ and $O(nk)$ space complexity[1], a significant improvement on the performance of the best prior work for approximate SUS finding.
- An additional contribution of this work comes in the area of practical performance improvement, where it is shown that the most time-expensive step in the new algorithm can be effectively parallelized to take advantage of modern multicore CPUs. Further, it is observed that the concurrency models applied to the new algorithm are also applicable to the k-mismatch average common substring finding algorithm presented in [13].
- The newly proposed algorithm for k-mismatch *SUS* finding has been fully implemented and is ready for use. The implementation is demonstrated to have achieved significantly improved processing times for approximate SUS finding, compared to the implementation of the HTX solution, when k is small relative to n, which is typically true in genomic sequence research due to the fact that the error rate of DNA sequencing instruments keeps coming down. For example, the serial implementation of the new algorithm processes a 200KB sample DNA sequence with $k = 1$ in just 0.18 seconds, compared to 174.37 seconds required by the HTX implementation. As an example, when using 8 cores, the parallel implementation gets a further speedup by a factor of over 4, when processing a 10MB sample DNA sequence with $k = 2$.
- While the new proposal has a higher space complexity than the HTX solution, and does indeed use considerably more memory in practice, this is likely to be an acceptable and needed trade-off for the improved processing times in many cases at the age of affordable Cloud infrastructure. For example, projecting out based on observed run times of the HTX solution, it can be expected that the solution may take more than 7 years to process a 200MB sample DNA input (for any $k > 0$), which is too long for a user to wait. In contrast, the new proposal, using 24 cores, finished processing a sample of this size with $k = 1$ in 206.376 seconds with a peak memory usage of 46GB which is both easily available and affordable from Cloud for many users. It is expected that this new tool for k-mismatch shortest unique substring finding will prove useful to those using the measure on long sequences in fields such as computational biology.

[1]Note that the algorithm presented in [12], which has no implementation yet by the authors of [12], is similarly adaptable, and solves the k-mismatch SUS finding problem in $O(n \log^k n)$ time and $O(n)$ space, in theory. However, this paper focuses on adapting the algorithm from [13] for its practicality of implementation and competitive expected time complexity.

2 PROBLEM FORMULATION AND PREPARATION

Consider a string S of n characters each drawn from an alphabet. $S[1]$ references the first character in S, $S[n]$ references the last character, and $S[i]$ references the i^{th} character in the string. A substring of S spanning from $S[i]$ to $S[j]$ (inclusive, $i \leq j$) is represented as $S[i..j]$. An index m of S is covered by a substring $S[i..j]$ *iff* $i \leq m \leq j$. The length of a substring $S[i..j]$ is denoted $|S[i..j]|$. The suffix of S which begins at index i is represented by S_i.

The Hamming distance between two equal length strings is defined as the number of indices at which characters differ between the two strings. A substring $S[i..j]$ is said to be k-mismatch unique if there exists no other substring of equal length $S[i'..j']$, $i' \neq i$, such that the Hamming distance between the two substrings is $\leq k$. A substring that is not k-mismatch unique is a k-mismatch repeat.

Definition 2.1. Of a given string S, a k-mismatch shortest unique substring covering index m, denoted as SUS^k_m, is a k-mismatch unique substring covering index m, such that no other k-mismatch unique substring covering m with a shorter length exists.

We say that a k-mismatch SUS is an *exact SUS* when $k = 0$, and an*approximate SUS* when $k > 0$.

Problem (k-mismatch *SUS* finding). For a string S of length n and a value k, $1 \leq k \leq n-1$, output two length n arrays A and B such that, for every index i in S, $S[A[i]..B[i]]$ is the rightmost SUS^k_i, using expected $O(n \log^k n)$ time and $O(nk)$ space.

In this work, we focus on the hard case where $1 \leq k \leq n-1$, because: (1) an optimal and practical solution with $O(n)$ time and space complexities already exists for the exact SUS case ($k = 0$) [7]. (2) The solution for the case where $k \geq n$ is trivial, as $SUS^n_m \equiv S$ for any index m.

Definition 2.2. The *k-mismatch longest common prefix* of two suffixes S_p and S_q, denoted as $LCP^k(S_p, S_q)$, represents the k-mismatch longest common prefix to suffixes S_p and S_q, that is, the longest prefix which has Hamming distance $\leq k$ between the two suffixes.

The notation of $LCP^0(S_p, S_q)$ is often simplified as $LCP(S_p, S_q)$ when it is clear from the context.

Definition 2.3. The *k-mismatch left-bounded longest repeat* starting at index i, denoted as LLR^k_i, is a k-mismatch repeat $S[i..j]$ such that $j = n$ or $S[i..j+1]$ is k-mismatch unique.

Clearly, $|LLR^k_i| = \max\{|LCP^k(S_i, S_j)|, j \neq i\}$, for every i.

Idea of the solution. Given an array of length n which at every index i holds the value $|LLR^k_i|$, algorithms presented in [7] can be directly applied to calculate SUS^k_i for every index i in S in $O(n)$ time and $O(n)$ space. Calculating all LLR^k_i values for the string S is the one algorithm presented in [7] that has $O(n^2)$ time complexity when $k > 0$. The dynamic programming-based strategy used in their work involves comparing every pair of distinct suffixes of S which clearly takes $O(n^2)$ time. In [13], an algorithm for finding the k-mismatch average common substring of two input strings X and Y is presented. A step of the algorithm involves calculating, for every index i in X, $\max_j\{|LCP^k(X_i, Y_j)|\}$ in expected $O(m \log^k m)$ time, where m is the combined length of X and Y. This is clearly similar to the calculation of $|LLR^k_i|$ values for each index in S. In the next section, it will be demonstrated that, with modifications, the same strategy from [13] can indeed be applied to calculate all $|LLR^k_i|$ values in expected $O(n \log^k n)$ time.

3 THE ALGORITHM

This section presents an adaptation and modification of the algorithm and associated analysis from [13], to make it operate on the single input string S and to calculate $|LLR^k_i|$ for every index i in S.

Definition 3.1 (Order-h Partition). An order-h partition C^h, where h is an integer $1 \leq h \leq k$, is a collection $\{P_1, P_2, \ldots\}$ of subsets of the set of all suffixes of S, such that for each $(S_i, S_j), i \neq j$ pair of suffixes of S, there exists a subset P in C^h where

$$|LCP^{h-1}(S_i, S_j)| = \min\left\{|LCP^{h-1}(s, s')| \mid s, s' \in P\right\}$$

The weight of C^h, $W(C^h)$, is the sum of sizes of all $P \in C^h$. Let $\Psi^{h-l}(P) = \min\{|LCP^{h-1}(s, s')| \mid s, s' \in P\}$.

The following subsections will demonstrate how an order-k partition with expected weight $O(n \log^k n)$ can be constructed, and that an order-k partition can be used to populate an array holding every $|LLR^k_i|$ value in linear time with respect to the partition's weight.

3.1 Constructing an order-k partition

The approach presented here to construct an order-k partition is iterative. First, an order-1 partition is constructed using the suffix tree of S, then an order-2 partition is constructed using the order-1 partition, and so on until finally an order-k partition is constructed.

For the purposes of this algorithm, two properties of compact tries over sets of suffixes (for which no suffix is a prefix of any other suffix) are important:

(1) Each non-leaf node is the lowest common ancestor of at least 2 suffixes since each non-leaf node has at least 2 non-empty sub-trees descending from it.
(2) Every pair of suffixes contained in such a trie will have 1 lowest common ancestor non-leaf node.

In order to ensure that no suffix is a k-mismatch prefix of another, each suffix of S has a sequence $\$_1\$_2 \ldots \$_{k+1}$ of $k+1$ special characters which do not appear in S appended to its end. Now, as an initial step, a suffix tree (a compact trie over all suffixes) of S is constructed which will be maintained throughout the LLR^k finding algorithm. The suffix tree requires $O(n)$ space and construction takes $O(n)$ time [15].

To generate C^1, iterate over each non-leaf node u of the suffix tree of S, and at each such node, collect a subset $P \in C^1$ which consists of all of the suffixes corresponding to leaves which are descendants of u. For correctness, observe that each pair $(S_i, S_j), i \neq j$ of suffixes will be included in the subset P, collected at the non-leaf node that is their lowest common ancestor in the tree, and that both $|LCP^0(S_i, S_j)|$ and $\Psi^0(P)$ are equal to the string-depth of this node. Additionally, since each suffix of S belongs to at most 1 non-leaf node at each level of the suffix tree, it can immediately be seen that $W(C^1) \leq nH$, where H is the height of the suffix tree. Another

way to think about each subset P collected is that, each contains at least 2 suffixes that have different characters at index $\Psi^{h-1}(P)+1$, while all of the included suffixes have length $\Psi^{h-1}(P)$ prefixes that are within Hamming distance $h-1$ of each other; this is clearly the case in the outlined $h=1$ case, and will be maintained as an invariant across each iteration to generate subsequent higher order partitions.

Now it will be demonstrated generally how a partition C^h can be generated from a partition C^{h-1}. For each P in C^{h-1}, create a new set P' which consists of the suffixes from P with each having had its first $\Psi^{h-2}(P)+1$ characters deleted, and create a compact trie Δ over the suffixes in P'. Then, iterate over each non-leaf node w in Δ, and at each such node collect a subset $P'' \in C^h$ which has one entry for each suffix corresponding to a leaf node in the trie which is a descendant of w. Rather than adding the suffix for each descendant leaf node directly to P'', instead the original suffix which had a prefix deleted to create the corresponding entry in P' is used. This can be equivalently expressed as, for each P in C^{h-1}:

$$P' = \{S_{i+\Psi^{h-2}(P)+1} \mid S_i \in P\}$$

and, where Z is the set of suffixes corresponding to the descendant leaves of w:

$$P'' = \{S_i \mid S_{i+\Psi^{h-2}(P)+1} \in Z\}$$

Conceptually, the $\Psi^{h-2}(P)+1$ length prefix deletion when generating each P' can be thought of as accepting and moving past the mismatch occurring at index $\Psi^{h-2}(P)+1$ in at least 2 of the suffixes in P. The subsequent processing of P' follows the same logic used when processing the set of all suffixes of S in the $h=1$ case, once again a compact trie structure is used to identify indices where next mismatches occur between suffixes with length $\Psi^{h-1}(P'')$ prefixes that are within Hamming distance $h-1$ of each other. Note that the height of Δ is $\leq H$, this is clear because the compact trie is created over a subset of the suffixes over which the suffix tree of S was created. It follows that $W(C^h) \leq H \cdot W(C^{h-1})$ since $W(C^{h-1})$ is the total number of suffixes across all P', and each suffix in a particular P' corresponds to a leaf node which is the descendant of just 1 non-leaf node per level in the corresponding Δ. Combining this observation with the known bound on $W(C^1)$, it is seen that $W(C^k) = O(nH^k)$.

3.1.1 Correctness. Under the assumption that C^{h-1} is an order-$(h-1)$ partition, it will now be formally proven that the collection C^h, generated as specified previously, is an order-h partition. By our assumption, it is the case that for any $(S_i, S_j), i \neq j$ pair there exists a $P \in C^{h-1}$ such that $|LCP^{h-2}(S_i, S_j)| = \Psi^{h-2}(P)$. Consider Δ to be the trie constructed while processing P. Based on the definition of P' over which Δ was created, and previously noted trie properties, it is known that a node w exists in Δ which is the lowest common ancestor of the leaves corresponding to suffixes $S_{i+\Psi^{h-2}(P)+1}$ and $S_{j+\Psi^{h-2}(P)+1}$ and the string-depth of w in Δ is equal to $|LCP(S_{i+\Psi^{h-2}(P)+1}, S_{j+\Psi^{h-2}(P)+1})|$. It follows then, based on its definition, that the new set $P'' \in C^h$ constructed at w contains both S_i and S_j. Further, it is clear that $\Psi^{h-1}(P'') = |LCP^{h-1}(S_i, S_j)|$ since exactly one additional mismatch between S_i and S_j was bypassed when processing P. This completes the proof.

3.1.2 Time and space complexity. When processing each $P \in C^{h-1}$, the set P' can be collected in $O(|P|)$ time. Construction of the corresponding compact trie Δ can be completed in overall $O(|P'|\log|P'|)$ time by lexicographically sorting the suffixes in P', computing the longest common prefix lengths between all pairs of suffixes which are consecutive in the sorted order in $O(|P'|)$ time, and then using a standard linear time suffix tree construction technique [2, 13]. Combining for all $P \in C^{h-1}$ the total time spent constructing the compact tries while generating C^h from C^{h-1} is $O(W(C^{h-1})\log n)$. Producing the P'' sets from the generated tries takes, in total, time proportional to the sum of sizes across all of the sets that are generated, which is known to be $O(W(C^h)) = O(W(C^{h-1})H)$. Adding the time for trie creation with the time spend generating P'' sets results in the total time spent generating C^h from C^{h-1}: $O(W(C^{h-1})(\log n + H))$. The total time for creating C^k is then $(\log n + H)\sum_{h=1}^{k-1} W(C^h) = O(nH^{k-1}(H+\log n))$.

On the topic of space complexity, observe that when creating a $P'' \in C^h$ only a single $P \in C^{h-1}$ is needed. Based on this observation, it is clearly possible to generate the members of C^k in a depth-first manner in which there is only ever one member in existence at a time for each C^h for $1 \leq h < k$. Using this strategy, $O(nk)$ space complexity can be achieved, because the space usage of one member from C^h, $1 \leq h < k$, is $O(n)$. Also note that we do not store all the members of C^k in memory. Rather, once a member of C^k is generated in the depth-first manner, it will be processed and then discarded (see details in Section 3.2).

Lemma 3.2. *Members of an order-k partition C^k of total weight $O(nH^k)$ can be generated in sequence using $O(nk)$ working space in $O(nH^{k-1}(H+\log n))$ time.*

3.2 Processing members of an order-k partition

An array B of length n is initialized such that all elements are 0. As each member $P \in C^k$ is generated (Section 3.1), it is processed right away on the fly, possibly resulting in updates to elements in B, and then it is discarded. When processing of all members $P \in C^k$ is complete, B will hold at each index i the value $|LLR_i^k|$. Processing of each member P consists of the following steps:

(1) For each suffix $s \in P$, find the lexicographic rank in S of the suffix s' which is obtained by deleting the length $(\Psi^{k-1}(P)+1)$ prefix from s, and place this rank in a pair with s'. Conceptually, the s' suffixes are the remainder of the suffixes in P after deleting prefixes up to and including the character at the index of the first k^{th} mismatch occurrence across all of the suffixes in P. Note then, that the first mismatch occurring between any two s' suffixes will be no greater than the $(k+1)^{th}$ mismatch between the corresponding two members of P. The lexicographic rank of a given s' can be computed in $O(1)$ time using the suffix tree of S [13].
(2) Sort all pairs from the previous step in an array V by their s' rank. Note that this sorting step moves pairs which have the longest common prefixes between their s' suffixes closer together.
(3) Let $\delta = (\Psi^{k-1}(P)+1)$ and $lcaStringDepth(S_x, S_y)$ be a function that returns the string-depth of the lowest common

ancestor node of the two leaf nodes in the suffix tree of S which correspond to the distinct suffix arguments S_x and S_y. Iterate over the indices into the array V of sorted pairs from index $p = 1$ to $p = |V|$. At each index, let i be the index in S at which the suffix s starts, where s is the suffix in P from which $V[p].s'$ was created, and calculate two candidate values based on adjacent pairs:

$$a = \begin{cases} \delta + lcaStringDepth(V[p].s', V[p-1].s'), & \text{if } p > 1 \\ 0, & \text{otherwise} \end{cases}$$

and:

$$b = \begin{cases} \delta + lcaStringDepth(V[p].s', V[p+1].s'), & \text{if } p < |V| \\ 0, & \text{otherwise} \end{cases}$$

then update:

$$B[i] \leftarrow \max\{B[i], a, b\}$$

Note that $lcaStringDepth(S_x, S_y)$ can be computed in $O(1)$ time using the suffix tree of S [3].

3.2.1 Correctness. Observe that the candidate values used to update an element at index i in B are always either less than or equal to $|LCP^k(S_i, S_o)|$ where S_o is the other suffix s corresponding to the s' from the relevant adjacent pair in V. This is clear because it is known that all members of P had at most k mismatches up to and including index $(\Psi^{k-1}(P) + 1)$, and by adding the string-depth of the lowest common ancestor of the two s' suffixes to this index, the index just prior to the next mismatch between S_i and S_o was calculated. From this observation, and the fact that no suffix S_i appears multiple times in the same $P \in C^k$, it follows that the final value at index i in B after processing all members of C^k is no greater than $\max_{j \neq i}|LCP^k(S_i, S_j)|$. Let $j = m$ be the index where $|LCP^k(S_i, S_j)|$ is maximized for any given i. By definition, C^k must include a member P such that $S_i, S_m \in P$ and $\Psi^{k-1}(P) = |LCP^{k-1}(S_i, S_m)|$. During processing of this P, the sorting in step 2 will arrange the pairs corresponding to S_i and S_m to be adjacent and $B[i]$ will be updated to the value $|LCP^k(S_i, S_m)|$. This concludes the proof that after processing all members of C^h, the array B will have been correctly updated to hold at each index i the value $|LLR_i^k|$.

3.2.2 Time complexity. The processing of C^k consists of sorting and iterating over sets which altogether have a total size of $O(nH^k)$, so a time complexity bound of $O(nH^k(\log n))$ is obvious. However, as described in section 2.2 of [13] the $\log n$ factor can be eliminated by observing that all of the sorting required is over integers in the range from 1 to n and thus can be accomplished using linear-time sorting algorithms like count sort. This optimization leaves a time complexity of $O(nH^k)$.

Lemma 3.3. *An array B of length n containing at each index i the value $|LLR_i^k|$ can be computed by processing C^k in $O(nH^k)$ time.*

Combining lemma 3.2 and lemma 3.3 yields the following theorem.

Theorem 3.4. *Given a string S of length n, and an integer $k \geq 1$, an array B of length n can be computed such that for every index $1 \leq i \leq n$ the value at $B[i]$ is equal to $|LLR_i^k|$ in $O(nH^{k-1}(H + \log n))$ time using $O(nk)$ space.*

Since the expected height of a suffix tree for a string of length n is $O(\log n)$ [1, 13], we can have:

Corollary 3.5. *Given a string S of length n, and an integer $k \geq 1$, an array B of length n can be computed such that for every index $1 \leq i \leq n$ the value at $B[i]$ is equal to $|LLR_i^k|$ in $O(n \log^k n)$ time using $O(nk)$ space.*

3.3 Parallel order-k partition construction and processing

It has been demonstrated in the prior subsections that each member of an order-k partition can be constructed through independent processing of each non-leaf node of the suffix tree of S. Further, it has been shown that each member of an order-k partition can be processed independently to generate candidate values for each index i of the $|LLR^k|$ array, and that the maximal candidate value generated in this way for any index i will be equal to $|LLR_i^k|$. A contribution of this paper is the observation that this independence means that multiple members of C^k can be computed and processed concurrently, each independently on separate computing threads with some form of synchronization only required when comparing candidate values, for the same index of the $|LLR^k|$ array, which were generated by different threads. While this parallelism can provide significant practical improvement to processing times on modern multi-core machines, these gains clearly come at the cost of an additional factor t, the number of concurrent threads, on the space complexity of the solution, because there are t independent instances of the depth-first search like procedure for the construction and processing of the members of C^k. However, it is shown in section 4 that with a good choice of concurrency model, the additional space usage observed in practice is often fairly minimal and that significant processing time improvements can be achieved even with a relatively low t value. It is worth noting that this strategy for parallelism can be similarly applied to the k-mismatch average common substring finding proposal from [13].

3.4 Computing SUS values

Definition 3.6 (k-mismatch LSUS). The k-mismatch left-bounded shortest unique substring that starts at index i, denoted as $LSUS_i^k$, is a k-mismatch unique substring $S[i..j]$, such that $i = j$ or otherwise every proper prefix of $S[i..j]$ is a k-mismatch repeat.

Prior to passing the array B as input into the standalone algorithms presented in [7], it is necessary to make a final transformation such that the array holds, at every index i, the ending index of $LSUS_i^k$, or NIL if no such $LSUS_i^k$ exists. Fact 4.2 from [7] can be used to update B, holding all $|LLR_i^k|$ values, such that at each index i it instead holds the ending index of $LSUS_i^k$, if it exists, and NIL otherwise, in one $O(n)$-time iteration as follows.

$$B[i] = \begin{cases} NIL, & \text{if } B[i] = n - i + 1 \\ i + B[i], & \text{otherwise} \end{cases}$$

Finally, a new array A of length n can be passed along with B into Algorithms 3 and then 4 from [7] in succession to update the two arrays in place such that, for every index i in S, $S[A[i]..B[i]]$ is the rightmost SUS_i^k. These algorithms each require $O(n)$ time and $O(1)$

additional working space. Clearly, the time and space spent creating and processing the order-k partition C^k dominates, and thus the overall expected time complexity of this k-mismatch *SUS* finding algorithm is $O(n \log^k n)$ while the space complexity is $O(kn)$.

Theorem 3.7. *Given a string S of size n and an integer k, one can find SUS_i^k of S for every index i using $O(n \log^k n)$ expected time and $O(kn)$ space.*

4 EXPERIMENTAL STUDY

Note that our proposal and implementation can also be applied to the exact *SUS* finding problem ($k = 0$). However, the experimental results are uninteresting and thus have been omitted, since the optimal $O(n)$ time and space in-place solution for exact *SUS* finding presented in [7] is clearly superior. This is consistent with what we claimed earlier in the paper that the main contribution of this work lies in the approximate SUS finding ($k > 0$), which is the harder case, and for which the best prior work has an any-case $O(n^2)$ time complexity and thus does not scale well to long strings.

Setup. Experiments were run on a dedicated c5.9xlarge EC2 instance hosted by Amazon Web Services,[2] featuring 3.0 GHz Intel Xeon Platinum processors with 36 cores, and 72GB RAM, running the Amazon Linux 2 operating system. In each experiment, the input string S of length n was drawn from the first n characters of the largest `DNA` file available from the Pizza&Chili corpus[3]. The peak memory usage data presented in this section was collected using the GNU `time` executable. The presented timing data was collected by adding code to the implementations which records the start and end time of processing. This internal timing strategy was used in order to focus on processing times of the implementations without including time spent on disk I/O operations required to read input and write output.

Implementation. In order to explore the practical performance of the algorithm presented in this paper, the `C++` implementation from [13] was modified to use the presented algorithm to calculate SUS_i^k values for every index i of an input string[4]. The adapted implementation maintains the same strategy for simulating operations on the suffix tree, using a suffix array (SA), inverse suffix array (ISA), LCP array, and range minimum query (RMQ) tables. SA construction makes use of the `libdivsufsort` library [10], while the ISA, LCP array, and RMQ tables are built using the SDSL library [4]. As [13], our implementation did not use supported compression techniques on the structures produced by the SDSL library in order to optimize for time performance. The executable used for collecting experimental results was compiled using version 7.2.1 of the `GCC C++` compiler with the `-O3` optimization option applied.

4.1 Two parallel strategies

It was noted in section 3.3 that construction and processing of relevant order-k partitions can be completed in parallel across t threads. In order to demonstrate the practicality and effectiveness of this parallelization, two parallel strategies, each using a different concurrency model, were implemented and evaluated in addition to the serial algorithm.

- The first strategy uses a simple non-shared approach wherein each thread has its own independent length n array in which to store candidate values for the final B array holding $|LLR_i^k|$ values. Then, after all members have been constructed and processed, passes are made in serial over each of the t arrays to populate the final B array with the overall maximum value occurring at each index.
- The second strategy uses a shared approach where a single length n array B is shared across all t threads. This implementation uses lock-free atomic operations when accessing or updating a value stored at a particular index in the shared B array to control data races and ensure correctness.

In both of the parallel strategies, non-leaf nodes of the suffix tree of S (from which members of the order-k partition are generated) are initially divided evenly among the t threads. The non-shared implementation distributes the nodes such that nodes with a lower string-depth in the suffix tree will be processed first, in an effort to ensure that the most expensive, with regards to the amount of work necessary to construct them, members of the order-k partition are constructed early, and in an attempt to roughly balance the number of expensive members initially assigned to each thread. The shared implementation shuffles the non-leaf nodes and distributes them randomly to the threads, in an effort to avoid collisions between updates to the value at the same index of the shared B array, while maintaining an expected rough balance of expensive members across threads. Each thread of both parallel implementations uses a simple work-stealing strategy to dynamically rebalance remaining work any time an individual thread finishes its assigned work, until no work remains across all threads. The non-shared approach has the advantage of being quite simple and not needing to worry about possible performance degradation due to update collisions, but this clearly comes at the cost of additional memory use.

4.2 Results

A note regarding the experimental results on peak memory usage presented in this section is that, a brief initial spike in memory usage was generally observed during the RMQ table construction. As a result, expected slopes in peak memory usage plots (as explained later in this section by varying t or k values) do not emerge until these values are sufficiently high, as to cause memory use during partition construction to surpass the initial RMQ construction spike. This factor should be kept in mind when interpreting the peak memory usage graphs presented in the rest of this section.

Performance affected by the values of k.

- Time: (1) In the processing time graph included in Figure 1, it can be seen that all three implementations of this paper's proposed algorithm perform significantly better than the existing HTX solution from [7], when k values are small, which is typically true due to the error rate of DNA sequencing instruments decreasing over time. (2) Also seen is the expected exponential growth in processing time as k increases.

[2] https://aws.amazon.com/
[3] http://pizzachili.dcc.uchile.cl/texts.html
[4] Our `C++` implementation: https://github.com/dra4/k_mismatch_sus_finding

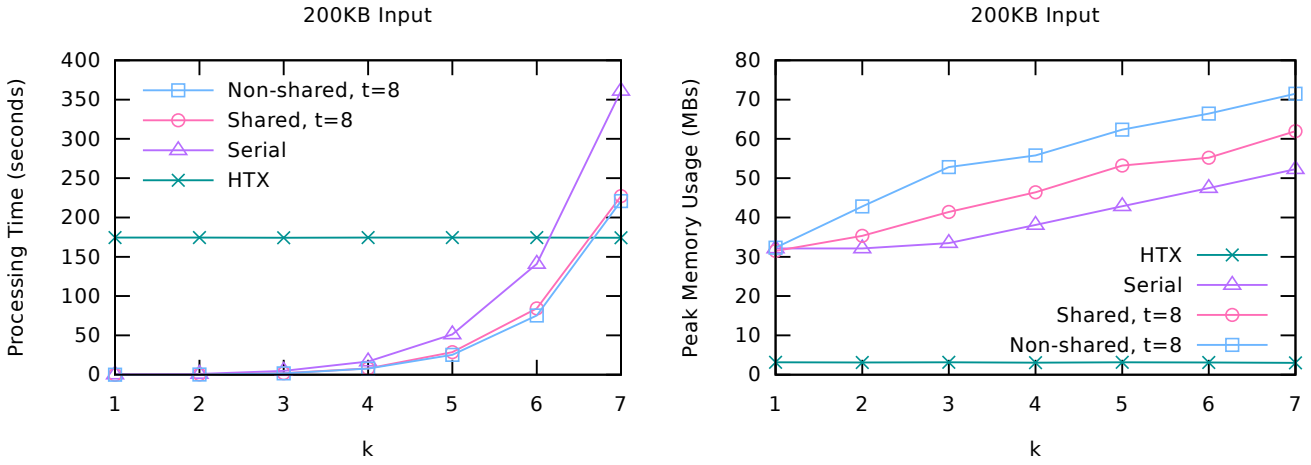

Figure 1: Processing time and peak memory usage measurements across implementations, given a 200KB input string and varying k values. HTX from [7], along with the serial and two parallel implementations of this paper's proposed algorithm.

It is clear that after k grows beyond a certain point, relative to the input string length, the HTX solution (which has a processing time independent of k) offers superior time performance. (3) The non-shared and shared parallel implementations consistently outperform the serial implementation of this paper's proposal. Time performance between the two parallel implementations is quite similar, with the non-shared approach achieving slightly faster times.

- Space: (1) The graph in Figure 1 showing peak memory usage shows that, as expected, all implementations of this paper's proposal use more memory than the in-place HTX algorithm. (2) This graph also illustrates the expected linear relationship between the k value and peak memory usage by this paper's implementations while t and n values are held constant. (3) As anticipated, among this paper's implementations, the serial version of the algorithm uses the smallest amount of memory, while the non-shared parallel strategy uses the most.

Performance improvement via parallelism. Graphs in Figure 2 depict the impact of the thread count, t, on processing time and peak memory usage required by the parallel implementations of our proposal. Note that the advantage of our proposal against the HTX solution has been demonstrated in Figure 1, and thus in this figure we focus on the comparison of the serial and parallel implementations of our proposal.

- Time: (1) The processing time plots show that the first additional threads result in the largest step improvements to processing time with returns diminishing and eventually leveling out and subsequently even starting to degrade. (2) Notably, the level point occurs later for the larger input string. This pattern is fairly intuitive, as there must be enough work available for assignment to each thread to offset the costs associated with allocating that thread and dividing and/or combining work across additional threads. This trend was observed to continue in an additional experiment with the shared parallel implementation which processed a 200MB input string when $k = 2$ in 1,367.22 seconds with t set to 12, and processed the same input in 1,249.94 seconds with t set at 24. (3) The processing time results in these graphs show that with sufficiently high values of t in these scenarios both parallel implementations were able to achieve speeds more than 4 times faster than the serial implementation, with the non-shared implementation again slightly faster than the shared implementation.
- Space: While the peak memory usage of both parallel implementations diverges from the reference point set by the serial implementation as t grows large, as expected, growth is much steeper for the non-shared implementation.

Scalability. The graphs of Figure 3 present the scalability of our proposal when the input string size n gets larger. Again, here we focus on the comparison of the serial and parallel implementations of our proposal, as their advantage against the HTX solution has been well demonstrated by Figure 1.

- Time: (1) When k is relatively small, our proposal scales well when the string size grows, showing its nearly linear time complexity, in its both serial and parallel implementations. (2) Comparing the processing time graphs for $k = 1$ and $k = 2$, it can be observed that the factor, by which parallelism increases processing speed, is consistently larger in the $k = 2$ case, where there is overall a greater amount of work to be done in the partition generation and processing stage. (3) In both cases, the parallel implementations show consistent significant improvements in processing time, when compared to the serial implementation. (4) Once again, processing times differ only slightly between the two parallel implementations, with the non-shared implementation showing a relatively small speed advantage when compared to the shared implementation.

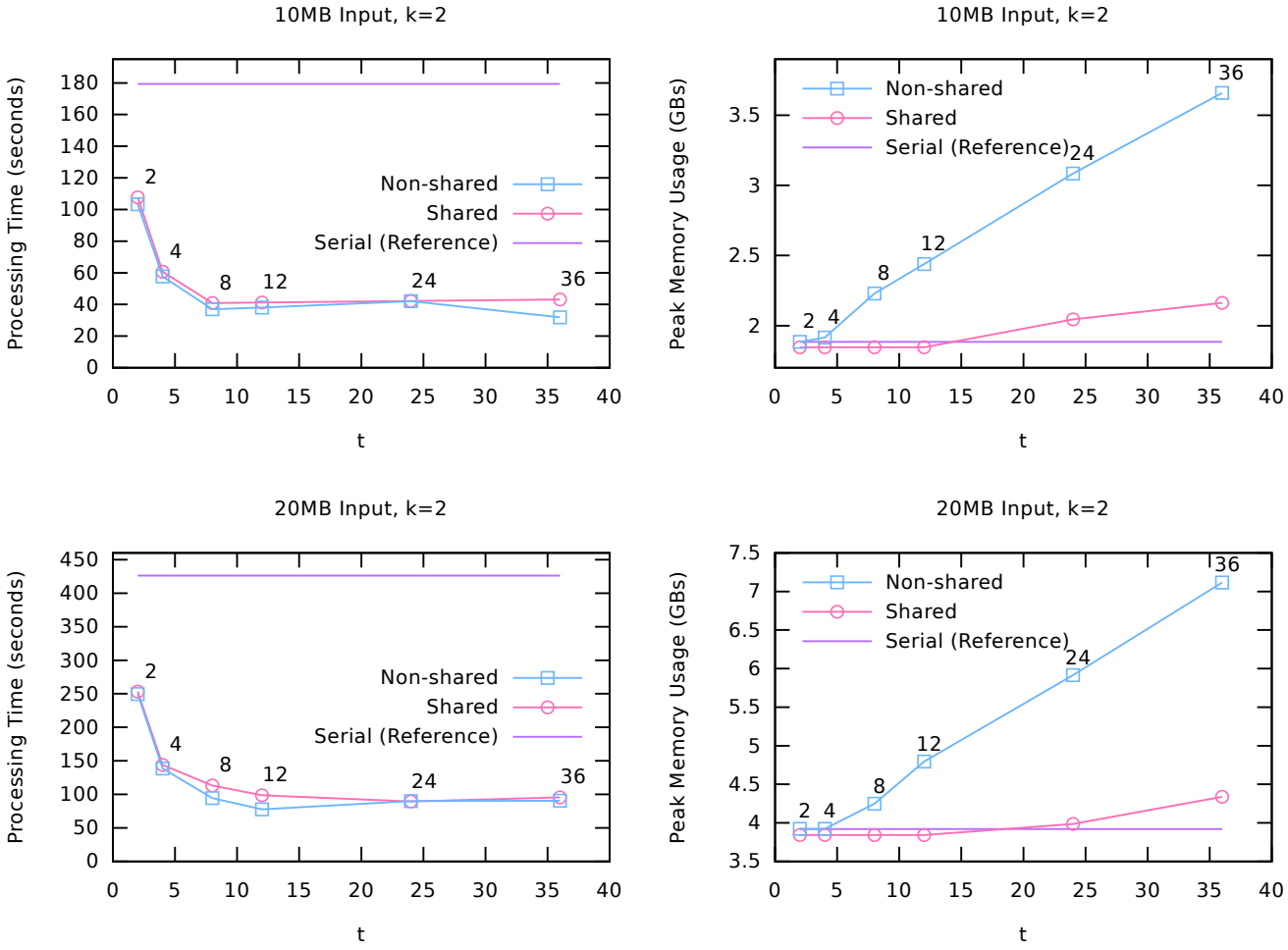

Figure 2: Processing time and peak memory usage measurements across implementations, given 10MB and 20MB input strings and varying t (thread count) values. Measurements from the two parallel implementations of this paper's proposed algorithm are included along with measurements from the serial implementation using 1 thread as a reference point.

- Space: (1) The peak memory usage graphs show that in the $k = 1$ cases, neither parallel implementation needs more space than the serial implementation, because all implementations do not need enough extra memory to overcome the initial memory peak seen during RMQ table construction. (2) However, in the $k = 2$ cases, the non-shared implementation does surpass that point and starts diverging upwards as n increases, as expected.

4.3 Summary

As demonstrated by the experimental results presented in this section, the primary advantage of the newly proposed algorithm over the prior best solution from [7] is significantly lower processing times when k is small relative to n. The improved processing times clearly come at the cost of additional memory usage. In an age where instances with thousands of gigabytes of RAM are readily available for use through Cloud infrastructure providers, this is expected to be an acceptable trade-off in many cases where the improved processing times make processing much longer input strings feasible. The results from the parallel implementations demonstrate that further significant practical improvement to processing times can be achieved through parallelism when multiple CPU cores are available. It is expected that the shared parallel implementation will be preferable as it has been observed to consistently perform nearly as well as the non-shared parallel implementation while using considerably less memory with high n and t values. When multiple CPU cores are available, choosing an initial t value which is equal to the number of available cores may be sensible since little degradation of processing time was observed for having "too high" of a t value. If memory is constrained, choosing a lower t value may be preferable and still provide significant practical performance improvement since the first few additional threads were observed to provide the largest incremental processing time improvements.

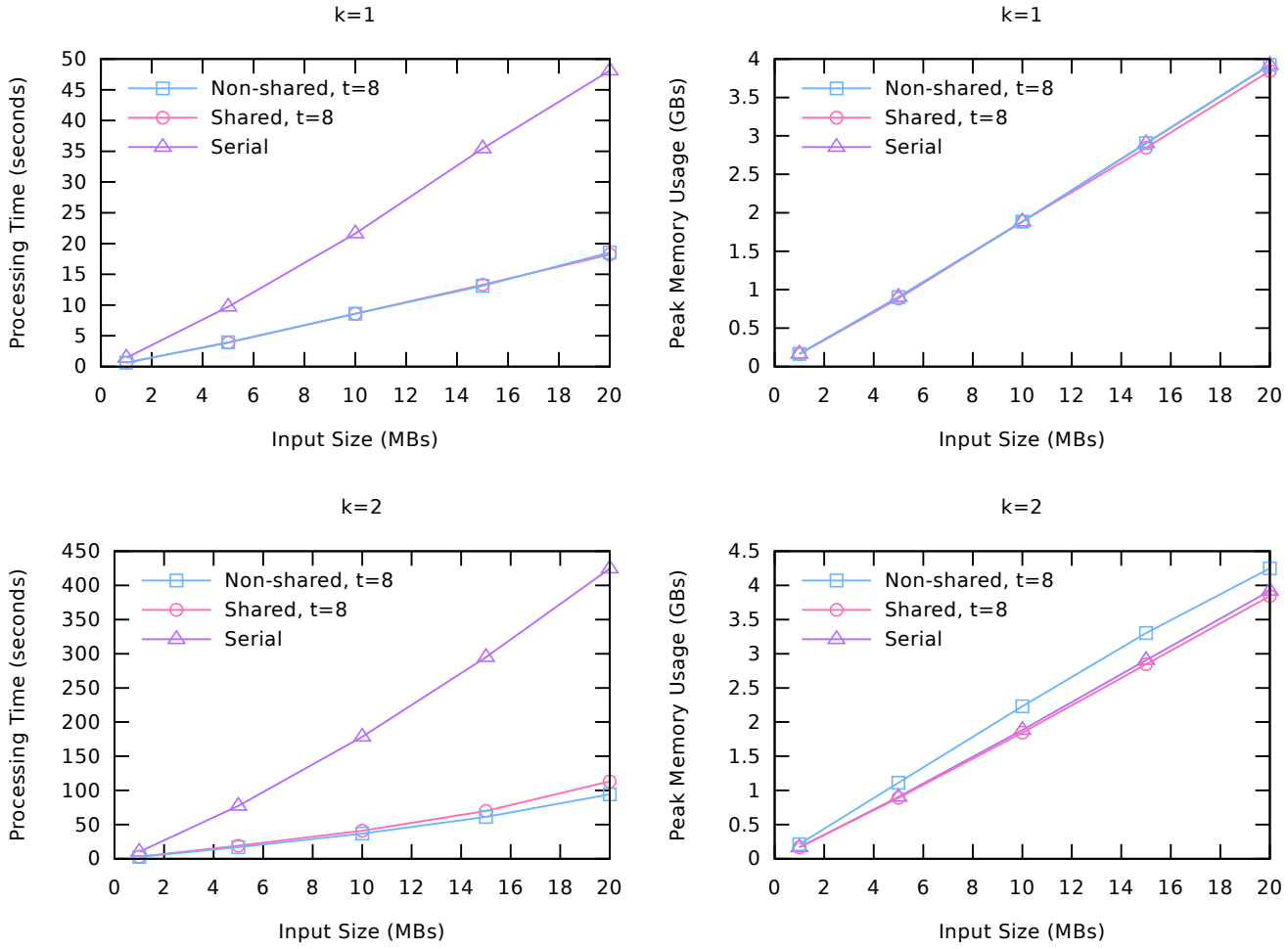

Figure 3: Processing time and peak memory usage measurements across implementations, given input strings of varying sizes and k values of 1 and 2. Measurements from the serial and two parallel implementations of this paper's proposed algorithm are included.

5 CONCLUSION

This paper revisited the k-mismatch shortest unique substring finding problem proposed by [7] and demonstrated that techniques presented in [13] could be adapted to help solve the problem in improved expected time complexity of $O(n \log^k n)$ while maintaining a practical space complexity of $O(kn)$. Further, it was observed that the techniques from [13] could be executed in parallel both in this problem's context as well as in the context of the k-mismatch average common substring problem which was worked on in the referenced paper. Experimental study showed that the new algorithm is practical to implement and demonstrated significantly improved processing times for small k values relative to n when compared to the implementation of the best prior solution from [7]. Experimental results were also presented which showed further practical performance improvement achieved through parallelism using two simple concurrency models. It is expected that this new efficient and practical algorithm for k-mismatch shortest unique substring finding will prove useful to those using the measure on long sequences in fields such as computational biology.

REFERENCES

[1] Luc Devroye, Wojciech Szpankowski, and Bonita Rais. 1992. A Note on the Height of Suffix Trees. *SIAM J. Comput.* 21 (1992), 48–53.
[2] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. 2000. On the Sorting-complexity of Suffix Tree Construction. *J. ACM* 47 (2000), 987–1011.
[3] Johannes Fischer and Volker Heun. 2006. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*. 36–48.
[4] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proceedings of the International Symposium on Experimental Algorithms*. 326–337.
[5] Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press.
[6] Bernhard Haubold, Nora Pierstorff, Friedrich Möller, and Thomas Wiehe. 2005. Genome comparison without alignment using shortest unique substrings. *BMC Bioinformatics* 6 (2005), 123.

[7] Wing-Kai Hon, Sharma V. Thankachan, and Bojian Xu. 2017. In-place algorithms for exact and approximate shortest unique substring problems. *Theoretical Computer Science* 690 (2017), 12 – 25.
[8] Xiaocheng Hu, Jian Pei, and Yufei Tao. 2014. Shortest Unique Queries on Strings. In *Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE)*. 161–172.
[9] Atalay Mert Ileri, M Oğuzhan Külekci, and Bojian Xu. 2015. A simple yet time-optimal and linear-space algorithm for shortest unique substring queries. *Theoretical Computer Science* 562 (2015), 621–633.
[10] Yuta Mori. [n. d.]. libdivsufsort: A lightweight suffix-sorting library. *https://github.com/y-256/libdivsufsort* ([n. d.]).
[11] Jian Pei, Wush Chi-Hsuan Wu, and Mi-Yen Yeh. 2013. On shortest unique substring queries. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. 937–948.
[12] Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. 2016. A Provably Efficient Algorithm for the k-Mismatch Average Common Substring Problem. *Journal of Computational Biology* 23 (2016), 472–482.
[13] Sharma V. Thankachan, Sriram P. Chockalingam, Yongchao Liu, Alberto Apostolico, and Srinivas Aluru. 2016. ALFRED: A Practical Method for Alignment-Free Distance Computation. *Journal of Computational Biology* 23 (2016), 452–460.
[14] Kazuya Tsuruta, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. 2014. Shortest Unique Substrings Queries in Optimal Time. In *Proceedings of the International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*. 503–513.
[15] P. Weiner. 1973. Linear pattern matching algorithms. In *Proceedings of the Annual Symposium on Switching and Automata Theory (SWAT)*. 1–11.