algorithms

MDPI

Article

A Survey on Shortest Unique Substring Queries

Paniz Abedin [1,*], M. Oğuzhan Külekci [2] and Shama V. Thankachan [1]

[1] Department of Computer Science, University of Central Florida, Orlando, FL 32816, USA; sharma.thankachan@ucf.edu

[2] Informatics Institute, Istanbul Technical University, Istanbul 34469, Turkey; kulekci@itu.edu.tr

* Correspondence: paniz@cs.ucf.edu

Received: 16 August 2020; Accepted: 4 September 2020; Published: 6 September 2020

Abstract: The shortest unique substring (SUS) problem is an active line of research in the field of string algorithms and has several applications in bioinformatics and information retrieval. The initial version of the problem was proposed by Pei et al. [ICDE'13]. Over the years, many variants and extensions have been pursued, which include positional-SUS, interval-SUS, approximate-SUS, palindromic-SUS, range-SUS, etc. In this article, we highlight some of the key results and summarize the recent developments in this area.

Keywords: string algorithms; shortest unique substring; repeats; compact data structures

1. Introduction

Let $\mathcal{S}$ be a string of length n and $\mathcal{S}[i,j]$ be the substring which starts at position i and ends at position j of $\mathcal{S}$. The substring $\mathcal{S}[i,j]$ is a repeat if it occurs more than once in $\mathcal{S}$; otherwise, it is a unique substring of $\mathcal{S}$. Since finding the shortest unique substrings is a non-trivial problem that has several applications for different purposes, variants of this problem have been studied. Table 1 shows the variants of the Shortest Unique Substring (SUS) problem that we focused on in this survey.

Table 1. This table categorizes the main papers which are reviewed and discussed in this survey.

Variant SUS Queries				
Position-SUS	**Interval-SUS**	**Approximate-SUS**	**Palindromic-SUS**	**Range-SUS**
Pei et al. 2013 [1] Ileri et al. 2014, 2015 [6,7] Tsuruta et al. 2014 [13] Hon et al. 2015, 2017 [3,17] Ganguly et al. 2016 [18] Mieno et al. 2019 [14]	Hu et al. 2014 [2] Mieno et al. 2016 [8] Mieno et al. 2019 [14]	Hon et al. 2017 [3] Allen et al. 2018, 2020 [9,10] Schultz et al. 2018, 2020 [15,16]	Inoue et al. 2018 [4] Wantabe et al. 2019, 2020 [11,12]	Abedin et al. 2019 [5]

In 2005, Haubold et al. [19] explained how the shortest unique substring is a useful construct for alignment-free genome comparison. Unique substrings can help to determine the distinctness and difference between a group of closely related organisms [1,19]. In addition, an algorithm for finding a unique substring can be helpful to build a unique genetic fingerprint from a DNA sample or can help designing polymerase chain reaction (PCR) primer technique in molecule biology [1,20]. In 2015, Adas et al. [21] investigated the usage of shortest unique substrings for alignment and compression of DNA sequences. In addition to the applications in bioinformatics, the shortest unique substrings can be used in information retrieval for document search. The *position based shortest unique substring (position-*SUS*) queries* was first proposed by Pei et al. [1]. Given a string $\mathcal{S}$ of length n and a query point p in the string, the problem is to find a shortest unique substring covering p. They presented an algorithm which costs $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space. In 2014, several publications have revisited this problem. İleri et al. [6] and Tsuruta et al. [13] proposed an optimal $\mathcal{O}(n)$ time and space algorithm for solving this problem. In addition, Hon et al. [3] and Ganguly et al. [18] presented compact data

structures that can answer a position-SUS query using less than $2n$ words of space, where a word size is $\Omega(\log n)$. Several variations of the original problem have been proposed in the following years.

In 2014, Hu et al. [2] generalized the position-SUS problem by considering the positions in an interval rather than a single position in the string. For the *interval*-SUS problem, they preprocess the input string in $\mathcal{O}(n)$ time and space to answer any interval SUS query in constant time and return all SUSs in $\mathcal{O}(occ)$ time, where occ is the number of outputs. In 2019, Mieno et al. [14] improved the space complexity of this problem by providing a compact data structure.

In 2016, Mieno et al. [8] considered SUS queries on *run-length encoded(RLE)* strings. Their motivation was to reduce space and time complexity of processing the input string. They showed how to construct a data structure of size $\mathcal{O}(m + \pi_s(N, m))$ in $\mathcal{O}(m \log m + \pi_c(N, m))$ time that can answer interval-SUS queries in $\mathcal{O}(\pi_q(N, m) + occ)$ time, where occ is the output size, and $\pi_s(N, m)$, $\pi_c(N, m)$ and $\pi_q(N, m)$ are the size, construction time and the query time for a predecessor/successor query on m elements from a universe of $[1, N]$.

In 2017, Hon et al. [3] proposed the *approximate* version of SUS queries, where mismatches are allowed. This version can be applied in computational biology, where factors such as genetic mutation and experimental error make approximate string matching necessary [9]. They presented an in place algorithm for both exact and approximate versions of the problem. Afterwards, different trade-offs have been presented for *k-mismatch* SUS problem [9,10,15,16].

In 2018, Inoue et al. [4] proposed a *palindromic* variant of interval SUS problem. A palindromic string or substring is an important structure in DNA, RNA or protein sequence analysis [22]. In biology data, palindromic structures show the ability of molecules to fold and form double-stranded stems [23]. Given a string $\mathcal{S}$, a *shortest unique palindromic substring* (SUPS) for an interval $[s, t]$ of $\mathcal{S}$ is the shortest substring that is palindromic and unique in $\mathcal{S}$ which contains $[s, t]$. For solving the interval SUPS problem, they preprocess $\mathcal{S}$ in $\mathcal{O}(n)$ time and space to output SUPSs in $\mathcal{O}(occ + 1)$ time. There are other works on SUPS queries by Watanabe et al. [11,12] based on the *RLE* which are space economical solutions.

In 2019, Abedin et al. [5] focused on the *range* version of SUS queries *(Range*-SUS*)* and generalized the problem. Range queries are a classic data structure topic, which has a great motivation in string processing problems [5,24–26]. Given a range $[\alpha, \beta]$, the problem is to return a shortest substring with exactly one occurrence in $[\alpha, \beta]$. They presented an $\mathcal{O}(n \log n)$-word data structure which answers rSUS queries in $\mathcal{O}(\log_w n)$ time per query in the word RAM model, where $w = \Omega(\log n)$ is the word size [5].

In this survey, we are going to discuss all approaches mentioned above on SUS queries. The focus of this work is on techniques applicable to SUS queries and is to compare all the main results in terms of complexities, restrictions, problem definitions, motivations, and applications. The main papers that we are going to focus on have been categorized in Table 1. In the last section, we discuss the related open questions on the variant topics related to SUS queries.

2. Preliminaries

2.1. Definitions

Let $\mathcal{S}[1, n]$ be a string of length n (i.e., $|\mathcal{S}| = n$) over an alphabet set Σ and $\mathcal{S}[i]$ is the ith character of $\mathcal{S}$. The substring of $\mathcal{S}$ which starts at position i and ends at position j of $\mathcal{S}$ is denoted by $\mathcal{S}[i, j]$. We have $1 \leq i \leq j \leq n$. If $j < i$, then $\mathcal{S}[i, j]$ is an empty string. A prefix of $\mathcal{S}$ is a substring $\mathcal{S}[1, i]$ of $\mathcal{S}$ for some $1 \leq i \leq n$. $\mathcal{S}[1, i]$ is a proper prefix if $i \neq n$. A suffix of $\mathcal{S}$ is a substring $[j, n]$ for some $1 \leq j \leq n$. It is a proper suffix if $j \neq 1$. We say $\mathcal{S}[i, j]$ covers position p, if $i \leq p \leq j$. $\mathcal{S}[i', j']$ is a proper substring of $\mathcal{S}[i, j]$ if $i \leq i' \leq j' \leq j$ and $|\mathcal{S}[i, j]| > |\mathcal{S}[i', j']|$. A substring $\mathcal{S}[i, j]$ is a unique substring if there is no other substring $\mathcal{S}[i', j']$ such that $\mathcal{S}[i, j] = \mathcal{S}[i', j']$, where $=$ indicates the identicality of two strings. The shortest unique substring (SUS) covering position p is a unique substring of $\mathcal{S}$ that contains $\mathcal{S}[p]$, and there is no other unique substring with shorter length containing $\mathcal{S}[p]$. Note that there might exist more than one SUS covering position p. $\mathcal{S}[i, j]$ is a *minimal unique substring* (MUS) if

it is a unique substring and there is no proper substring of $\mathcal{S}[i,j]$ that is also unique. $\mathcal{S}[p,j]$ is called the *left-bounded* SUS for position p, $\mathsf{LSUS}(p)$, if $\mathcal{S}[p,j]$ is unique and there is no other substring $\mathcal{S}[p,j']$ which is also unique for $p \leq j' < j$. Symmetrically, $\mathcal{S}[j,p]$ is *right-bounded* SUS for position p, $\mathsf{RSUS}(p)$, if it is a unique substring and there is no other unique substring $\mathcal{S}[j',p]$, where $j < j' \leq p$.

$H(\mathcal{S}_1, \mathcal{S}_2)$ denotes the *Hamming distance* between two strings $\mathcal{S}_1$ and $\mathcal{S}_2$ of equal length, defined as the number of string positions where characters differ. $\mathcal{S}[i,j]$ is a *k-mismatch unique substring* if there is no substring $\mathcal{S}[i',j']$ such that $i' \neq i, j-i = j'-i'$ and $H(\mathcal{S}[i,j], \mathcal{S}[i',j']) \leq k$. The k-mismatch left-bounded shortest unique substring LSUS starting at position p, denoted as LSUS_p , is a k-mismatch unique substring $\mathcal{S}[p,j]$, such that either $p = j$ or any proper prefix of $\mathcal{S}[p,j]$ is not k-mismatch unique. k-mismatch RSUS and k-mismatch MUS can be defined similarly.

The longest common prefix of two suffixes $\mathcal{S}[p,n]$ and $\mathcal{S}[q,n]$ denoted by $\mathsf{LCP}(\mathcal{S}[p,n], \mathcal{S}[q,n])$, is the longest common prefix between $\mathcal{S}[p,n]$ and $\mathcal{S}[q,n]$. The k-mismatch longest common prefix of $\mathcal{S}[p,n]$ and $\mathcal{S}[q,n]$ denoted as $\mathsf{LCP}^k(\mathcal{S}[p,n], \mathcal{S}[q,n])$ is the longest prefix which has Hamming distance $\leq$k between two suffixes.

2.2. Data Structures

The *suffix tree* data structure of string $\mathcal{S}[1,n]$ is a compact trie of the n suffixes of $\mathcal{S}$ appended with a letter $\$ \notin \Sigma$ [27]. This suffix tree consists of n leaves (one for each suffix of $\mathcal{S}$) and at most $n-1$ internal nodes. The edges are labeled with substrings of $\mathcal{S}$. We denote the suffix tree of string $\mathcal{S}$ with $\mathsf{ST}(s)$. The *Suffix Array* of string $\mathcal{S}$ of length n is denoted by SA, which is a permutation of $\{1, \cdots, n\}$, such that $\mathsf{SA}[i] = j$ if $\mathcal{S}[j,n]$ is the ith lexicographically smallest suffix of $\mathcal{S}$. The *Inverse Suffix Array* of string $\mathcal{S}$ of length n, is a permutation of $\{1, \cdots, n\}$, such that $\mathsf{SA}^{-1}[\mathsf{SA}[i]] = i$. SA of $\mathcal{S}$ can be constructed in linear time and space [28,29]. The *longest common prefix (*lcp*) array* of a string $\mathcal{S}$ of length n is an integer array of length n such that $\mathsf{lcp}[1] = 0$ and $\mathsf{lcp}[i]$ stores the length of the longest common prefix between $\mathcal{S}[\mathsf{SA}[i-1], n]$ and $\mathcal{S}[\mathsf{SA}[i], n]$. Given the suffix array of $\mathcal{S}$, lcp array can be constructed in $\mathcal{O}(n)$ time. Let $A[1,n]$ be an array of length n. A range minimum query (RMQ) with an input range $[i,j]$ asks to report $RMQ(i,j) = \arg\min_k \{A[k] \mid k \in [i,j]\}$. By maintaining a data structure of size $2n + o(n)$ bits, any RMQ on A can be answered in $O(1)$ time [30] (even without accessing A).

Consider s as a subset of $\{1, 2, \ldots, n\}$. Then, s can be preprocessed into an $O(|s|)$ space data structure, such that for any query p, we can return $\mathsf{pred}(p,s)$ and $\mathsf{succ}(p,s)$ in $O(\log\log n)$ time [31], where

$$\mathsf{pred}(p,s) = \max\{i \mid i \leq p \text{ and } i \in s \cup \{-\infty\}\}$$

$$\mathsf{succ}(p,s) = \min\{i \mid i \geq p \text{ and } i \in s \cup \{\infty\}\}$$

The eertree of a string $\mathcal{S}$, is a pair of two rooted trees T_{odd} and T_{even} that represent all distinct palindromic substrings of $\mathcal{S}$ [32]. T_{odd} and T_{even} store the palindromic substrings of odd and even length, respectively. There is a directed edge (r,a,v) from root r of T_{odd} if v represents a single character $a \in \Sigma$. For any non-root node u in either T_{odd} or T_{even}, there is a labeled directed edge (u,a,v) from u to v with character label a if $aua = v$. There are no two out-going edges from a node with the same label.

3. Position-SUS Queries

3.1. Motivation

Consider the procedure a search engine performs. Once a search query is given into a search engine by users, all the related pages should be identified and ranked properly. An indexing process is needed to organize information before each search query. There are algorithms such as inverted indexing to keep track of the documents with the pointers to text elements. The modern search engines may use a *snippet*, which is a short summarized content of a whole website and is shown in the search results. Finding a proper length for a snippet is critical. Either too short length and too long length would be problematic for making the text elements distinguishable and not overwhelming

for users [1]. If the snippet for each result of a search is the shortest possible text including the query term and different from all other snippets, the search would be optimized. Thus, providing a fast algorithm for finding a shortest snippet is crucial. In addition to information retrieval purposes, there are some motivations in bioinformatics. In 2005, Haubold et al. [19] explained how the shortest unique substring is a useful construct for alignment-free genome comparison. Unique substrings can help to determine the distinctness and difference between a group of closely related organisms [1,19]. Another application is in event analysis when one wants to understand how an event differs from other events of the same type in a long sequence of historical events while extracting the context of the event. The shortest unique substring of the selected event may be helpful to proceed with the event analysis [1].

Regarding these motivations, Pei et al. [1] introduced the following problem.

Problem 1. ***Position*-SUS *Queries***
Input: String $\mathcal{S}[1,n]$ *and a position* $p \in \{1 \cdots n\}$.
Output: SUS *covering p: Substring* $\mathcal{S}[i,j]$ *containing position p, i.e.,* $i \leq p \leq j$ *, such that* $\mathcal{S}[i,j]$ *is unique and as short as possible*

Example 1. *Given* $\mathcal{S} =$ $\overset{1\,2\,3\,4\,5\,6\,7\,8\,9\,10\,11\,12\,13\,14\,15\,16\,17\,18\,19\,20\,21}{\mathbf{caabaaddaacaddaaaabac}}$ *and a query position* $p = 5$*, we need to find a shortest substring* $\mathcal{S}[i,j]$*, with exactly one occurrence in* $\mathcal{S}$ *such that* $i \leq 5 \leq j$*. One possible output is* $\mathcal{S}[4,6] = baa$*. Note that we may have multiple answers for Problem 1.* $\mathcal{S}[5,7] = aad$ *is another output.*

3.2. Suffix Trees Based Approach

Pei et al. [1] stated Problem 1 and presented an algorithm to answer this problem in $\mathcal{O}(n)$ time and space. They construct $\mathsf{ST}(\mathcal{S})$ in $\mathcal{O}(n)$ space and time [33]. Then, they use $\mathsf{ST}(\mathcal{S})$ to get $\mathsf{LSUS}(p)$ in constant time by the following steps:

- Find the leaf node corresponding to the suffix $\mathcal{S}[p,n]$
- If the label of the leaf edge is \$, it means that $\mathsf{LSUS}(p)$ does not exist and we return null; Otherwise, we continue.
- Let l be the length of the label of the leaf edge (excluding \$).
- $\mathcal{S}[p, n-l+1]$ is $\mathsf{LSUS}(p)$.

All of the above steps can be completed in $\mathcal{O}(1)$ time using the suffix tree properties. They make clever use of Lemma 1 and, consequently, they can find a SUS covering p by the following Lemma.

Lemma 1 ([1,6])**.** *Every* SUS *is either an* LSUS *or an extension of an* LSUS*.*

They start with assuming that $\mathsf{LSUS}(p) = \mathcal{S}[i,j]$ is a candidate answer, then they look for a $\mathsf{LSUS}(k)$ where $k < p$ and $k \geq j - i$ with the shortest length. Fore each $\mathsf{LSUS}(k) = \mathcal{S}[k,e]$, if $e < p$, instead of $\mathcal{S}[k,e]$, $\mathcal{S}[k,p]$ should be considered as a candidate answer since it should cover p. This is called an *extension* of an LSUS. Thus, they always make sure that a new candidate covers p. At the end, since there may exist more than one answer, they output the leftmost SUS containing p.

In addition, they show how to preprocess $\mathcal{S}$ in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space to compute the SUS corresponding to every position in the string. By doing so, SUS queries for any position can be answered in constant time. Their technique is based on the fact that each SUS should fall into one of MUS, LSUS, or RSUS. By this observation, SUS corresponding to each position can be precomputed using their propagation procedure [1].

From the space complexity point of view, corresponding to each position p, their algorithm keeps track of a currently shortest MUS that covers p. The total space needed to store this information for all positions is $\mathcal{O}(n)$. At the end, they apply their algorithm on real data sets to show the effectiveness of their algorithm. Theorem 1 summarizes their result.

Theorem 1. *The position-* SUS *problem can be answered in* $\mathcal{O}(n^2)$ *time and* $\mathcal{O}(n)$ *space for every location of a string of length* n.

3.3. Linear Time Approaches

Ileri et al. [6,7] and Tsuruta et al. [13] independently improved the time complexity of the Theorem 1 from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ time. They showed that, by preprocessing the string $\mathcal{S}$ of length n in $\mathcal{O}(n)$ time, Problem 1 can be answered in constant time, which concludes a linear total time complexity for all positions in the string [6,7,13]. First, we discuss Ileri et al.'s approach.

3.3.1. Ileri et al.'s Framework

Their framework includes two cases; SUS finding for one position, and SUS finding for every position. In the first case, they present an algorithm for finding all the SUSs covering a specific location p in $\mathcal{O}(n)$ time. Similar to the approach explained in Section 3.2, they make this observation that each SUS is an extension of an LSUS. Instead of using suffix tree structure to find LSUSs, they use inverse suffix array and lcp array to compute $\mathsf{LSUS}(i)$ for $i = 1, 2, \cdots n$ as follows:

$$\mathsf{LSUS}(i) = \begin{cases} \mathcal{S}[i, i + l_i] & if\ i + l_i \leq n \\ null & otherwise \end{cases},$$

where $l_i = \max\{\mathsf{lcp}[\mathsf{SA}^{-1}[i]], \mathsf{lcp}[\mathsf{SA}^{-1}[i] + 1]\}$ and *null* means LSUS does not exist. Using the above equation, for each string position p, they simply compute $\mathsf{LSUS}(1) \cdots, \mathsf{LSUS}(p)$ in $\mathcal{O}(p)$ time and maintain the shortest one for position p. Note that, if for some $k < p$, $\mathsf{LSUS}(k)$ does not cover p, we can extend it up to position p. In case of multiple shortest answers, they keep the leftmost candidate. Then, by Lemma 1, they prove their first theorem as follows:

Theorem 2 ([6])**.** *For any location* p *in the string* $\mathcal{S}$*, they can find* SUS *covering* p *using* $\mathcal{O}(n)$ *time and space. If multiple answers exist, the leftmost one is returned.*

In the second case, they extend their algorithm to find all the SUSs for every location $p \in \{1, 2 \cdots n\}$. Instead of iteratively running the algorithm for finding a SUS of a specific position n times, they use the following lemma to reduce the time complexity to the amortized cost for finding each SUS in $\mathcal{O}(1)$ time.

Lemma 2 ([6])**.** *For any* $k \in \{2, 3, \cdots, n\}$*, if* SUS *for position* k *is an extension of an* LSUS*, then (1)* SUS *for position* $k - 1$ *must be a substring whose right boundary is the character* $\mathcal{S}[k - 1]$*, and (2)* SUS *for position* k *is the substring* SUS *for position* $k - 1$ *appended by the character* $\mathcal{S}[k]$.

For finding SUS of every position, they begin with SUS of the first position which is LSUS of that position; then, by Lemma 2, they compute SUS of position k using the already calculated SUS of position $k - 1$. Their algorithm costs $\mathcal{O}(occ)$ for reporting the SUSs covering a particular location. By providing an efficient constant time algorithm for computing the shortest LSUS covering each string position, they prove the following theorem.

Theorem 3. *All* SUS*s corresponding to all the positions of string* $\mathcal{S}[1, n]$ *can be computed in* $\mathcal{O}(n)$ *time and space.*

For the implementation, they use *libdvsufsort* library for implementing the suffix array and lcp array. They compare their results with Tsuruta et al.'s work and shows that, in terms of time complexity, both algorithms have almost the same processing time; however, their space usage is at least four times less for finding a single SUS and two times less for finding all SUSs.

3.3.2. Tsuruta et al.'s Framework

Tsuruta et al. [13] improved Pei et al.'s (ICDE 2013) upper bounds for the shortest unique substring problem as well. They consider two types of SUS queries. The first one computes a SUS for any position in the string $\mathcal{S}$ in constant time after preprocessing $\mathcal{S}$ in $\mathcal{O}(n)$ time. Thus, reporting a SUS for all positions in $\mathcal{S}$ takes $\mathcal{O}(n)$ time in total. Their next algorithm outputs all the SUSs covering a query position. Note that Pei et al.'s algorithm reports only one SUS for a query position. By preprocessing $\mathcal{S}$ in $\mathcal{O}(n)$ time, Tsuruta et al. provided an algorithm which reports all SUSs containing a query position in $\mathcal{O}(occ)$ time, where occ is the number of output.

Although their algorithm follows a similar technique to Ileri et al. [6], their work was independent. Their main idea is finding SUSs from MUSs using the following lemma.

Lemma 3 ([13])**.** *Every position-*SUS *contains exactly one* MUS*.*

They use lcp array data structure to compute all MUSs of $\mathcal{S}$ in linear time. We summarize their results in the following lemma.

Lemma 4. *All the* LSUS*,* RSUS*, and* MUS *can be computed from $\mathcal{S}$ in $\mathcal{O}(n)$ time.*

In order to compute SUSs from MUSs, they define the concept of *meaningful* and *meaningless* MUSs, where meaningful MUSs drive SUSs corresponding to some positions. Then, they present an algorithm to collect all the SUSs from the meaningful MUSs in linear time. Theorem 4 summarizes their results.

Theorem 4. *A string S of length n can be preprocessed in $\mathcal{O}(n)$ time and space so that the shortest unique substring queries can be answered in $\mathcal{O}(occ)$ time, where occ is the number of shortest substrings returned. Notably, outputting a single* SUS *can be done in $\mathcal{O}(1)$ time.*

3.4. In Place and Compact Data Structures' Approaches

The suffix tree of a string $\mathcal{S}$ of length n occupies space almost 20 times larger than the space needed to store $\mathcal{S}$, which is $n \log |\Sigma|$ bits. All aforementioned data structures in Section 3.3 require $\Theta(n)$ words, where n is the length of the string $\mathcal{S}$. When n is large, the memory usage would be problematic. In order to avoid this issue, Hon et al. [3] proposed a data structure including the input string and two integer arrays for storing the starting positions and ending positions of SUSs. The preprocessing time for their algorithm is linear and takes $\mathcal{O}(\log n)$ bits of additional working space. If we want to consider the space needed to store all SUSs for all positions, their algorithm takes the least amount of space to do that, which is $2n$ words to maintain the starting and ending positions of SUSs and another n bytes to store the input string. Previous works need $\mathcal{O}(n)$ space while the hidden constant is large; however, that overhead does not exist in Hon et al.'s work. They use suffix arrays instead of suffix trees in their construction. Moreover, they handle all computations in the place of two integer arrays. As a result, their algorithm can find the position-SUS for every string position in $\mathcal{O}(n)$ time.

There was still a question of whether we can solve SUS problem in sub-linear space. To answer this question, Ganguly et al. [18] presented the first time-space trade off algorithm which uses $\mathcal{O}(n/\tau)$ words of additional space. Given a position $p \in \{1 \cdots n\}$, their algorithm answers Problem 1 in $\mathcal{O}(n\tau^2 \log \frac{n}{\tau})$ time, where parameter $\tau \geq 1$. Another query is reporting SUSs for all positions of $\mathcal{S}$. For this type of query, they present an $\mathcal{O}(n\tau^2 \log n)$ algorithm using $\mathcal{O}(n/\tau)$ words and $4n + o(n)$ bits of additional working space. In addition to these deterministic algorithms, they present a randomized algorithm in which the time complexity is $\mathcal{O}(n\tau \log^{c+1} n)$ by using additional $n/\log^c n$ words, where $c \geq 0$ is an arbitrary constant. There is a chance of at most $n^{-\mathcal{O}(1)}$ that the reported substring is unique and covers the query position but not the shortest one.

In their deterministic algorithms, their intuition is that each SUS for a position k is definitely the shortest unique prefix of $\mathcal{S}[i,n]$ for some $i \leq k$, or the smallest right extension till position k of such prefix. For each suffix $\mathcal{S}[i,n]$, they define LS_i as the shortest unique prefix of $\mathcal{S}[i,n]$. In order to reduce the space complexity, instead of storing LS_is for all $\mathcal{S}[i,n]$, $i \in \{1 \cdots n\}$, they choose a set of $\mathcal{O}(n/\tau)$ suffixes and compute the corresponding LSs. After computing all the LSs of the suffixes in the chosen set, using a brute force approach, for any suffix $\mathcal{S}[j,n]$, they can compute LS_j. Their results are summarized in Theorem 5.

Theorem 5 ([18]). *For any given string position $p \in \{1 \cdots n\}$, an* SUS *covering p can be computed in $\mathcal{O}(n\tau^2 \log \frac{n}{\tau})$ time and additional $\mathcal{O}(n/\tau)$ working space. In addition, computing any* SUS *for every position in the string can be reported in $\mathcal{O}(n\tau^2 \log n)$ time and additional $\mathcal{O}(n/\tau)$ words and $4n + o(n)$ bits of additional working space.*

Ganguly et al.'s algorithm reports only one SUS for each given query. In 2019, Mieno et al. [14] presented a data structure of size $\lceil (\log_2 3 + 1)n) \rceil + o(n)$ bits, which can answer Problem 1 in $\mathcal{O}(occ)$ time, where *occ* is the number of SUSs for the given query point. The main intuition of their algorithm is based on Lemma 3. Their data structure includes two bit arrays $\mathsf{MB}_\mathcal{S}$ and $\mathsf{ME}_\mathcal{S}$, each of length n, to keep track of the starting and ending positions of the MUSs . As a result, they prove that there exists a data structure of size $\lceil (\log_2 3 + 1)n) \rceil + o(n)$ bits that can answer SUSs in $\mathcal{O}(occ)$ time. Given the bit arrays $\mathsf{MB}_\mathcal{S}$ and $\mathsf{ME}_\mathcal{S}$, the data structure can be constructed in $\mathcal{O}(n)$ time using $3n + \lceil n \log_2 3 \rceil + o(n)$ bits of total working space.

4. Interval-SUS Queries

In this section, we discuss a generalization of position-SUS problem which is defined as follows:

Problem 2. ***Interval*** SUS ***Queries***
Input: String $\mathcal{S}[1,n]$ and a query interval $[s,t] \in [1,n]$.
Output: All SUS*s of $\mathcal{S}$ containing $[s,t]$*

Example 2. *Given* $\mathcal{S} =$ $\overset{1\,2\,3\,4\,5\,6\,7\,8\,9\,10\,11\,12\,13\,14\,15\,16\,17\,18\,19\,20\,21}{\mathbf{caabaaddaacaddaaaabac}}$ *and an interval query position $[14,16]$, we need to find a shortest substring $\mathcal{S}[i,j]$, with exactly one occurrence in $\mathcal{S}$ such that $i \leq 14 < 16 \leq j$. The output here is $\mathcal{S}[14,17] = daaa$ which includes $\mathcal{S}[14,16] = daa$ and it does not have any other occurrence in $\mathcal{S}$.*

4.1. Linear Time Approaches

Hu et al. [2] generalized Problem 1 to Problem 2 and presented a linear time and space data structure which can solve Problem 2 in $\mathcal{O}(occ)$ time, where *occ* is the number of all SUSs containing the query interval. The interval SUS problem is more difficult to be solved in $\mathcal{O}(occ)$ time, since there exist $\Theta(n^2)$ intervals and we cannot store the candidate answer for every possible interval. In order to deal with this issue, Hu et al. use the concept of LSUS, RSUS, MUS, and Corollary 1 to prove Lemma 5.

Corollary 1 ([34]). *There exists a data structure of $\mathcal{O}(n)$ size which can be constructed in $\mathcal{O}(n)$ time that can check whether a given substring of $\mathcal{S}$ is unique in $\mathcal{O}(1)$ time.*

Lemma 5 ([34]). *The answer of the interval-*SUS *problem with the query interval $[x,y]$ must be the shortest of the following candidates:*

1. *$\mathcal{S}[x,y]$ if it is unique. This can be checked in constant time and linear space.*
2. *$\mathsf{LSUS}(x)$: This can be computed in constant time using linear space by Lemma 4.*
3. *$\mathsf{RSUS}(y)$: This can be computed in constant time using linear space by Lemma 4.*
4. *The shortest* MUS *containing $[x,y]$: It remains to show their structure of computing this candidate.*

Let M be the set of all MUSs in $\mathcal{S}$. For presenting their data structure, they define a set of intervals denoted by I, such that $[i,j] \in I$ iff $\mathcal{S}[i,j]$ is in M. Then, they reduce the interval-SUS problem to *Containment Min*. Given an interval $[x,y]$, a containment min query returns the shortest interval in I which contains $[x,y]$. Hu et al. [2] proposed a data structure of size $\mathcal{O}(n)$ in which RMQ is performed to answer a containment min query in $\mathcal{O}(1)$ time.

In addition, they show how to report all SUSs in the case that interval SUS has more than one answer. For this purpose, they use an auxiliary problem which is defined below.

Position Constraint Query. Given a substring $\mathcal{S}[x,y]$ and two ranges $range_{start} = [s_1,s_2]$ and $range_{end} = [e_1,e_2]$ both in $[1,n]$, the problem is to return a unique substring $\mathcal{S}[i,j]$ with the minimum length that contains $\mathcal{S}[x,y]$ such that $i \in [s_1,s_2]$ and $j \in [e_1,e_2]$.

This problem is a constraint version of the interval SUS problem where the starting and ending positions of the answer substring should be contained in the two intervals specified in the problem. They start with the normal interval-SUS query. Given the input interval $[x,y]$, let $\mathcal{S}[i,j]$ be the answer of the interval-SUS. They consider two intervals $[1,i-1]$ and $[i+1,x]$. Now, using two position constraint queries, the next answers can be found. First, they run a position constrained query with $\mathcal{S}[x,y]$, $range_{start} = [1,i-1]$ and $range_{end} = [y,n]$ to return the other possible answers. The other query corresponding to $[i+1,x]$ is symmetrical. In total, their algorithm takes $\mathcal{O}(occ)$ time to report all the SUSs.

The mentioned algorithms are in the RAM model. They also consider solving Problem 2 in the standard external memory model [35]. They follow a similar technique. Assume that $\mathcal{O}(SORT(n))$ is the number of I/Os needed to sort n elements and B is the number of words in a block of a disk. They pre-compute an index structure from $\mathcal{S}$ in $\mathcal{O}(SORT(n))$ I/Os in external memory that occupies $\mathcal{O}(n/B)$ blocks and can answer any shortest unique substring query in $\mathcal{O}(1)$ I/Os [2].

4.2. RLE-Based Approaches

The *Run Length Encoding (RLE)* of a string is a compressed representation in which each maximal run of a character c of length ℓ is encoded as c^{ℓ}. For instance, the *RLE* of string *aaaaaaabbbbaaac* is $a^7b^4a^3c^1$. Mieno et al. [8] considered solving Problem 2 in the case where the input string is given in *RLE* representation. Their motivation was to reduce space and time complexity of processing the input string. They presented a data structure of size $\mathcal{O}(m + \pi_s(N,m))$ that can be constructed in $\mathcal{O}(m \log m + \pi_c(N,m))$ time to answer interval-SUS queries in $\mathcal{O}(\pi_q(N,m) + occ)$ time, where occ is the output size, and $\pi_s(N,m)$, $\pi_c(N,m)$ and $\pi_q(N,m)$ are the size, construction time and the query time for a predecessor/successor query on m elements from a universe of $[1,N]$. In their approach, they use combinatorial properties on MUSs and *RLE* strings. Let m be the length of the string $\mathcal{S}$ in *RLE* representation. We denote *RLE* representation of $\mathcal{S}$ by $RLE(\mathcal{S})$. They show that the number of MUSs in $RLE(\mathcal{S})$ is $2m-1$. Thus, instead of dealing with $\Theta(n)$ MUSs like the previous related results, a considerable amount of space and time would be reduced if there exist runs in the string. By doing so, they were able to build an $\mathcal{O}(m)$ size data structure for the *RLE* version of SUS problem, which is formally defined as follows:

Problem 3. ***RLE*-SUS *Queries***
Preprocess: $RLE(\mathcal{S}) = c_1^{\ell_1}c_2^{\ell_2}\cdots c_m^{\ell_m}$
Query: An interval $[s,t] \in [1,n]$
Return: All SUS *of* $\mathcal{S}$ *containing the query interval* $[s,t]$

For solving Problem 3, they show how to precompute all the MUSs using a specific type of suffix arrays for *RLE* strings [36]. Before discussing their techniques, we bring some definitions. Let $bpos(i)$, $epos(i)$ and $exp(i)$ be the beginning position, ending position, and exponent of the ith run of $RLE(\mathcal{S})$. Let P be be any subset of positions of $\mathcal{S}$. The sparse suffix array of $\mathcal{S}$ w.r.t P denoted by SSA_P is an

array of size $|P|$ such that $\mathsf{SSA}_P[i] \in P$ for all $1 \leq i \leq |P|$ and also like a normal suffix array $\mathsf{SSA}_P[i]$ is lexicographically smaller than $\mathsf{SSA}_P[i+1]$. They use *trancated RLE suffix array* for $RLE(\mathcal{S})$ denoted by $tRLESA$, in which P is the set of $epos(i)$ for $1 \leq i \leq m$. In addition, let EXP be an array of length m such that $EXP[i] = exp(k)$, where $tRLESA[i] = epos(k)$. Given $RLE(s)$, all the defined arrays can be computed in $\mathcal{O}(m \log m)$ time with $\mathcal{O}(m)$ working space [14].

In order to compute MUSs from $RLE(\mathcal{S})$, they consider three disjoint partitions of MUS, M_1, M_2, M_3. The first partition consists of MUSs which are contained in runs. The second one is MUSs that start at the last characters of runs, and all the other MUSs considered to be in the third partitions. By the fact that each MUS cannot be a proper substring of another MUS and discussing the size of each partition separately, they prove that $|M| \leq 2m - 1$. Then, they show how to find MUS in $\mathcal{O}(m \log m)$ time and $\mathcal{O}(m)$ space, by computing the partitions.

In order to compute M_1, for each character used in $\mathcal{S}$, they check whether there exists a run of that character with a unique maximum exponent, since the number of distinct character used in $\mathcal{S}$ is $\leq m$, this procedure can be done in $\mathcal{O}(m \log m)$ time and $\mathcal{O}(m)$ space using a sorting algorithm. For computing M_2 and M_3, they use $tRLESA$, $tRLESA^{-1}$ (inverse of $tRLESA$), and EXP arrays. Similar to M_1, M_2 and M_3 can be computed in $\mathcal{O}(m \log m)$ time and $\mathcal{O}(m)$ space. After the preprocessing step, they could answer Problem 3 using RMQ, predecessor and successor queries on the array MUS_{len} that stores the lengths of the MUSs. Their results were summarized in the following theorem.

Theorem 6 ([8]). *Given $RLE(\mathcal{S})$ of size m, there is a data structure of size $\mathcal{O}(m + \pi_s(N, m))$ that can be computed in $\mathcal{O}(m \log m + \pi_c(N, m))$ time which can answer interval* SUS *queries in $\mathcal{O}(\pi_q(N, m) + occ)$ time, where occ is the output size, and $\pi_s(N, m)$, $\pi_c(N, m)$, and $\pi_q(N, m)$ are the size, construction time and the query time for a predecessor/successor query on m elements from a universe of $[1, N]$.*

4.3. Compact Data Structures' Approaches

In Section 3.4, we have discussed Mieno et al's work [8] for solving Problem 1. They also presented a compact data structure for Problem 2 [14]. Their data structure has the size of $2n + 2m + o(n)$ bits and outputs an interval SUS query in $\mathcal{O}(occ)$ time, where m is the number of minimal unique substrings (MUSs) of the input string which is at most n. Their technique is based on Mieno et al.'s work [8], which is discussed in Section 4.2. Similar to the technique for solving position-RSUS, their structure is based on the two bit arrays MB and ME. In addition, they use array MUS_{len} which has been used in Mieno et al.'s work for storing the length of the MUSs [8]. On the top of MB and ME, a successor and a predecessor data structures are maintained. MUS_{len} is also endowed with RMQ data structure. Now, once an interval query $[s, t]$ comes, first we can find $g = \mathsf{pred}_{\mathsf{ME}}(y)$ and $r = \mathsf{succ}_{\mathsf{MB}}(s)$, then we can find the range of SUS covering $[s, t]$. Consequently, it is sufficient to answer RMQ queries on the corresponding range on MUS_{len}. The space needed for MB and ME and the predecessor and successor data structure is $2n + o(n)$ bits. The RMQ data structure on MUS_{len} takes $2m + o(m)$ bits of space. Thus, using the constant query time of the RMQ data structure (see Section 2.2), their results can be summarized as follows:

Theorem 7. *There exists a data structure of size $2n + 2m + o(n)$ bits that can answer interval* SUS *problem in $\mathcal{O}(occ)$ time, where occ is the number of the answers corresponding to the given interval.*

5. Approximate-SUS Queries

5.1. Motivation

In molecular biology, shortest unique substrings found in DNA sequences can be used to compare similar organisms and determine unique patterns. It also helps to design polymer chain reaction (PCR) [1,15]. If we just consider exact shortest unique substrings while comparing distinct organisms, possible patterns might be disregarded due to errors or mutations. In this section, we discuss the

approximate version of SUS problem which is proposed by Hon et al. [3], in which the uniqueness constraint is more strict. In this variant of SUS problem, the unique substrings are allowed for up to k mismatches. This version can be applied in computational biology, where factors such as genetic mutation and experimental error make approximate string matching necessary [9]. Another useful application of this approximate version is in computing average common substring, which has been considered as an approach to phylogenomic reconstruction [37,38]. In order to estimate the evolutionary distance between pairs of primate genomes, Thankachan et al. [39] showed that adding a similar k-mismatch parameter to average common substring finding equation leads to better results [9]. A k-mismatch shortest unique substring covering a position p, denoted by SUS_p^k, is a k-mismatch unique substring $\mathcal{S}[i,j]$ such that satisfies the condition $i \leq p \leq j$ and there is no other k-mismatch unique substring with shorter length which satisfies the condition. Note that, similar to the definition of k-mismatch SUS in Section 2.1, we also consider the Hamming distance for the k-mismatch SUS problem. The problem is formally defined as follows:

Problem 4. ***k-mismatch*** SUS ***Queries***
Input: String $\mathcal{S}[1,n]$ and integer k
Output: Two integer arrays A and B s.t $\mathcal{S}[A[i]\cdots B[i]]$ is the rightmost SUS_i^k for every index i

Example 3. *If* $\mathcal{S} =$ $\overset{1\,2\,3\,4\,5\,6\,7\,8\,9\,10\,11\,12\,13\,14\,15\,16\,17\,18\,19\,20\,21}{\mathbf{caabaaddaacaddaaaabac}}$, $\mathsf{SUS}_5^1 = \mathcal{S}[4,7] = baad$. *Note that* $\mathsf{SUS}_5^0 = \mathcal{S}[4,6] = baa$ *as we saw in Example 1. Since there exists a substring caa such that* $H(baa, caa) = 1$, $\mathcal{S}[4,6] = baa$ *cannot be a* SUS_5^1.

5.2. Hon et al.'s Framework

Hon, Thankachan, and Xu considered solving Problem 4 in the place of S, A, and B, where S is for storing string $\mathcal{S}$, A and B are two empty arrays for storing the starting and ending positions of the rightmost k-mismatch SUS of each position. They prove the k-mismatch version of Lemma 2. Here, we explain the high level of their in-place framework for solving both Problems 1 and 4 which has three stages.

Let LSUS_p^k be the k-mismatch left bounded shortest unique substring starting at position p. In the first stage, they compute LSUS_i^k for all is, in place of A and B. At the end of this step, each $B[i]$ stores the ending position of LSUS_i^k. This procedure takes linear time when $k = 0$ and $\mathcal{O}(n^2)$ time for $k \geq 1$ because of their dynamic programming approach. Let SLS_p^k be the shortest k-mismatch LSUS covering position p. In the second stage, they use array B, computed from the previous stage, to find the rightmost SLS_i^k for all i in place of A and B. Each $A[i]$ stores the largest j such that LSUS_j^k is an SLS_j^k. Thus, if SLS_i^k exists, it is equal to $\mathcal{S}[A[i]\cdots B[i]]$. This takes $\mathcal{O}(n)$ time for all $k \geq 0$. At the last stage, they use both A and B to compute SUS_i^k for all i, in the place of A and B. At the end of the stage, $A[i]$ and $B[i]$ store the starting and ending positions of the rightmost SUS_i^k. This step also takes linear time for any $k \geq 0$. Note that only one stage requires the quadratic time. They implemented their algorithm in C using *libdivsufsort* library for the suffix array construction. Their contribution can be summarized as follows:

Theorem 8. *Using an additional $2n$ words each of size $\lceil \log_2(n) \rceil$ bits, and n bytes of space for storing the input string, Hon et al. [3] provided a theoretically in-place framework to solve both exact and the approximate position* SUS *using a total of $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ time, respectively.*

5.3. Allen et al.'s Framework

In order to reduce the quadratic time complexity for solving k-mismatch SUS queries, Allen et al. [9] presented an average time complexity of $\mathcal{O}(n \log^k n)$ using $\mathcal{O}(kn)$ space. They follow the technique provided by Thankachan et al. [39] for solving k-mismatch average common substring.

Let the k-mismatch left-bounded longest repeat starting at position i denoted by LLR_i^k be a k-mismatch repeat substring $\mathcal{S}[i,j]$ such that either $j = n$ or $\mathcal{S}[i,j+1]$ is a k-mismatch unique. Consider L as an array of length n where $L[i] = |\mathsf{LLR}_i^k|$. Now, the algorithm discussed in Section 5.2 can be applied to compute SUS_i^k for each position i in $\mathcal{S}$ in linear time and space. Computing array L can be a substitution for stage one of the Hon et al.'s work [3]. To avoid $\mathcal{O}(n^2)$ time, instead of comparing every pair of suffixes of $\mathcal{S}$, they presented an expected $\mathcal{O}(m \log^k m)$ time for calculating each entry of L using the technique in [39], where m is the total length of two suffixes used in computing $L[i]$ as follows:

$$L[i] = \max\{|\mathsf{LCP}^k(\mathcal{S}_i, \mathcal{S}_j)| j \neq i\}$$

The total time complexity to compute every entry of L would be expected $\mathcal{O}(n \log^k n)$.

In 2020, Allen et al. [10] published a new version of their work which provides algorithmic bounds for k-mismatch SUS problem. They presented an algorithm which can solve Problem 4 using $\mathcal{O}(n \log^k n)$ time and $\mathcal{O}(n)$ space. This worst-case bound is asymptotically much better than the practical algorithm. In comparison to Hon et al's experimental results [3], their new implementation shows that the practical algorithm is easy to implement and it takes less time when k is small relative to n.

5.4. GPU Based Approach

All the works that have been discussed on k-mismatch SUS query were focusing on improving the time and space complexity for SUS computation in the sequential CPU model. In 2018, Schultz and Xu [15] presented the first parallel approach for k-mismatch SUS problem in the shared-memory model, particularly leveraging on the massive multi-threading GPU technology. Obviously, this approach is experimentally faster than the CPU solutions. Their experimental results on a mid-end GPU show that the GPU approach is at least six times faster than the CPU approach in the case of $k = 0$ (exact match), and at least 23 times faster in the general case ($k > 0$). This is essential when the input string is massively long as genomic sequences are studied. In terms of memory usage, this approach is almost the same as the sequential CPU approaches. Their algorithms totally happen on GPU except a transferring input string to GPU, which is done by the CPU host. When there are no mismatches ($k = 0$), they implicitly use the efficient data structures like suffix array and lcp array. However, when $k > 0$, using these data structures would be problematic since they do not have enough information such as k-mismatch LCPs. To solve this issue, Schultz and Xu [15] designed a method that can be parallelized in the GPU architecture.

In 2019, they published an extension of their paper [16], providing more experimental results on exact SUS and approximate SUS computation. They show the speedup gained by the GPU-based approach against the sequential solution, including or excluding suffix array construction. It is important to observe the difference of speedup in case of whether the suffix array is given or not. Corresponding to their results, almost 50% of the time of the sequential approach is to spend on suffix array construction. Thus, if the suffix array is given, it can be copied to the GPU memory directly to achieve a better speedup.

6. SUPS Queries

6.1. Motivation

We call a substring $\mathcal{S}[i,j]$ is a palindrome if it is identical with its reverse. There have been many studies on palindromic strings and their combinatorial properties [32,40,41]. A palindromic string or substring is an important structure in DNA, RNA, or protein sequence analysis [22]. In biology data, palindromic structures show the ability of molecules to fold and form double-stranded stems [23]. In addition, by the similar palindromic structures of the protein strands, we can guess their similar secondary structures. Another application of palindromic substructures is in gene editing and gene regulation in species [22,42]. In this section, we are going to discuss a new version of SUS queries

named shortest unique palindromic substring(SUPS) problem. A *shortest unique palindromic substring* SUPS for an interval $[s,t]$ is the shortest substring $\mathcal{S}[i,j]$ such that it is unique in $\mathcal{S}$ and contains $[s,t]$, and any other palindromic substring of $\mathcal{S}$, which contains $[s,t]$ and is shorter than $\mathcal{S}[i,j]$, which occurs more than once in $\mathcal{S}$. The SUPS problem is formally defined as follows:

Problem 5. SUPS ***Queries***
Preprocess: $\mathcal{S}[1,n]$
Query: An interval $[s,t] \in [1,n]$
Return: All SUPS *of* $\mathcal{S}$ *containing the query interval* $[s,t]$

Example 4. *If* $\mathcal{S} =$ $\overset{1\,2\,3\,4\,5\,6\,7\,8\,9\,10\,11\,12\,13\,14\,15\,16\,17\,18\,19\,20\,21}{\mathbf{caabaaddaacaddaaaabac}}$, *given the query interval* $[8,9]$, SUPS *problem outputs* $\mathcal{S}[6,9] = adda$, *which is the shortest palindrome containing* $\mathcal{S}[8,9] = da$.

6.2. Optimal Approaches

In 2018, Inoue et al. [4] introduced Problem 5, which is a version of SUS queries focusing on palindromes. They propose an optimal algorithm which takes $\mathcal{O}(n)$ time to preprocess the string $\mathcal{S}$ and can return all SUPS for any interval query in $\mathcal{O}(occ+1)$ time, where occ is the number of outputs. For any interval $[i,j]$, checking whether $\mathcal{S}[i,j]$ is a palindromic substring or not takes $\mathcal{O}(n)$ preprocessing time and constant query time using $\mathcal{O}(n)$ space. Similar to the approach for solving position/interval SUS, they define *Minimal Unique Palindromic substring(MUPS)*. *MUPS* $\mathcal{S}[i,j]$ is a unique palindrome substring in $\mathcal{S}$ such that $\mathcal{S}[i+1,j-1]$ is a repeat in $\mathcal{S}$ or $1 \leq |\mathcal{S}[i,j]| \leq 2$. Similar to MUSs, *MUSPs* cannot contain each others. In addition, for each SUPS $\mathcal{S}[i,j]$ for some interval, there exists exactly one *MUPS* that is contained in $[i,j]$ with the same center as $\mathcal{S}[i,j]$ $(\frac{i+j}{2})$. Using this fact, they could preprocess $\mathcal{S}$ in linear time to report all SUPSs.

In addition, they provided some bounds on the number of SUPSs. They showed the maximum number of point and interval SUPS in $\mathcal{S}$, which is a string from an arbitrary size alphabet, is at most n. For a binary alphabet, they gave a lower bound for the number of point SUPS. Let $\mathcal{PS}_\mathcal{S}$ and $\mathcal{IS}_\mathcal{S}$ be the set of substrings of $\mathcal{S}$ where each of the substrings in $\mathcal{PS}_\mathcal{S}$ is an SUPS for some point and each of the substrings in $\mathcal{IS}_\mathcal{S}$ is an SUPS for some interval.

Theorem 9 ([4]). *There exists a binary string* $\mathcal{S}_k$ *of length* $6k+8$ *such that* $\mathcal{PS}_{\mathcal{S}_k} = \mathcal{IS}_{\mathcal{S}_k} = 4k+6$ *for any* $k \geq 0$. *Thus,* $\mathcal{S}_k$ *contains at least* $\frac{2}{3}n$ *point and interval* SUPSs *where* $n = |\mathcal{S}_k|$

In the end, they discuss the number of point and interval SUPSs on palindromic rich strings. A string is a palindromic rich if it has $n+1$ distinct palindromic substring including an empty string. They consider the palindromic rich string $R_k = a^1b^1 \cdots a^kb^k$ and prove two following theorems.

Theorem 10 ([4]). *There exists a binary string of length* n *such that the number of interval* SUPSs *is* $n - \sqrt{1+4n} + 3$.

Theorem 11 ([4]). *There exists a binary string* R_k *of size* $k(k+1)$ *s.t* $\mathcal{IS}_{R_k} = k(k-1)+2$ *for any* $k \geq 1$. *Thus,* R_k *contains* $n - \sqrt{1+4|R_k|} + 3$ *interval* SUPS.

6.3. RLE-Based Approaches

Watanabe et al. [11] considered the shortest unique palindromic substring when the input string is given in its *RLE* representation. They showed how to preprocess a given $RLE(S)$ of length m in $\mathcal{O}(m)$ space and $\mathcal{O}(m \log \sigma_{RLE_\mathcal{S}} + m\sqrt{\log m / \log\log m})$ time, where σ_{RLE_s} is the umber of distinct runs of RLE_s. Their work is the first space-economial for SUPS problem and can answer queries in $\mathcal{O}(\sqrt{\log m / \log\log m} + occ)$ time.

Similar to the technique for solving the normal SUPS problem which is discussed in the previous subsection, they compute *MUPS*s of S in the preprocessing step. Their idea is based on the fact that

the number of $MUPS$s of any string $\mathcal{S}$ is at most m [11]. They first compute the palindromic substrings whose center is the same as the center for some run in the string. These palindromic substrings are called *run-centered* palindromes. In order to compute these substrings, they utilize Manacher's algorithm [43]. This algorithm scans the string and constructs an array $MaxPal$ of length $2n+1$ in which the ith entry is the length of the maximal palindrome with center i. Using this technique, they compute all the RLE-maximal palindromes of $\mathcal{S}$ in $\mathcal{O}(m)$ time and space.

In order to improve the space complexity, they build a data structure which is a modified eertree of the input string to deal with the run-length encoded string instead of the original one. They prove that this new data structure, RLE-eertree (e^2rtre^2), has $2m+1$ nodes which is linear to the length of the $RLE(\mathcal{S})$. Thus, it takes less space complexity compared with the original eertree. The construction time for e^2rtre^2 is $\mathcal{O}(m \log \sigma_{RLE})$. They used this tree to compute all the $MUSPS$s of $\mathcal{S}$. Using the list of $MUSPS$s, they present their algorithm for SUPS queries. Their technique is similar to Inoue et al.'s work [4] with the difference that the space complexity should be $\mathcal{O}(m)$.

In 2020, Watanabe et al. [12] published an extension of their work by considering a variant of SUPS problem where a query interval is also given in a run-length encoded representation. Their technique is similar to their previous approach [11]. They used combinatorial properties of maximal palindromes and presented an $\mathcal{O}(m)$ space data structure to answer queries in $\mathcal{O}(\log \log m + occ)$ time [12].

7. Range-SUS

Range queries are a classic data structure topic, which has a great motivation in string processing problems [5,24–26]. In 2019, Abedin et al. [5] studied the local shortest unique substring which starts in a specific region or range of the string. Given a range $[\alpha, \beta]$, the problem is to return a shortest substring $\mathcal{S}[k, k+h-1]$ of $\mathcal{S}$ with exactly one occurrence in $[\alpha, \beta]$; i.e., $k \in [\alpha, \beta]$, there is no $k' \in [\alpha, \beta]$ such that $\mathcal{S}[k, k+h-1] = \mathcal{S}[k', k'+h-1]$, and h is minimal. Abedin et al. presented an $\mathcal{O}(n \log n)$-word data structure which answers rSUS queries in $\mathcal{O}(\log_w n)$ time per query in the word RAM model, where $w = \Omega(\log n)$ is the word size [5]. The *Range Shortest Unique Substring* problem is formally defined as follows:

Problem 6. rSUS ***Queries***
Preprocess: String $\mathcal{S}[1, n]$.
Query: Range $[\alpha, \beta]$, *where* $1 \leq \alpha \leq \beta \leq n$.
Output: (p, ℓ) *such that* $\mathcal{S}[p, p+\ell-1]$ *is a shortest string with exactly one occurrence in* $[\alpha, \beta]$.

If $\alpha = \beta$, the answer $(\alpha, 1)$ is trivial. Thus, in the rest, we assume that $\alpha < \beta$.

Example 5 ([5]). *Given* $\mathcal{S} =$ **caabcaddaacaddaaaabac** *and a query* $[\alpha, \beta] = [5, 16]$, *we need to find a shortest substring of* $\mathcal{S}$ *with exactly one occurrence in* $[5, 16]$. *The output here is* $(p, \ell) = (10, 2)$ *since* $\mathcal{S}[10, 11] =$ **ac** *is the shortest substring of* $\mathcal{S}$ *with exactly one occurrence in* $[5, 16]$.

For each position $k \in [1, n]$, Abedin et al. [5] provide a data structure to keep track of the last and next occurrence of substring $\mathcal{S}[k, k+h-1]$, denoted by $\mathsf{Prev}(k, h)$ and $\mathsf{Next}(k, h)$, respectively. They define $\lambda(a, b, k)$ and C_k as follows:

$$\lambda(a, b, k) = \min\{h \mid \mathsf{Prev}(k, h) < a \ \textbf{and} \ \mathsf{Next}(k, h) > b\}.$$

$$C_k = \{h \mid (\mathsf{Next}(k, h), \mathsf{Prev}(k, h)) \neq (\mathsf{Next}(k, h-1), \mathsf{Prev}(k, h-1))\}.$$

Their main result is providing an upper bound on the size of all C_ks, which is given in the following lemma.

Lemma 6 ([5]). $\sum_k |C_k| = \mathcal{O}(n \log n)$

By the definition, $\lambda(a, b, k)$ is the length of the shortest substring that starts at position k, and C_k is the set of candidate length for the RSUS answer, which starts at position k.

Abedin et al. solved rSUS queries by reducing the problem to an top-1 rectangle stabbing query on a set of rectangles with input point (α, β). Assume that the answer for Problem 6 is (p, ℓ). Given a query range $[\alpha, \beta]$, the answer (p, ℓ) we are looking for is the pair (k, h) with the minimum h under the following conditions: $k \in [\alpha, \beta]$, $h \in C_k$, $\mathsf{Prev}(k, h) < \alpha$ and $\mathsf{Next}(k, h) > \beta$. Equivalently, (p, ℓ) is the pair (k, h) with the minimum h, such that $h \in C_k$, $\alpha \in (\mathsf{Prev}(k, h), k]$, and $\beta \in [k, \mathsf{Next}(k, h))$.

In the preprocessing step, they map each $h \in C_k$ into a weighted rectangle $R_{k,h}$ with weight h and defined as follows:

$$R_{k,h} = [\mathsf{Prev}(k, h) + 1, k] \times [k, \mathsf{Next}(k, h) - 1].$$

After query $[\alpha, \beta]$ comes, the lowest weighted rectangle which stabbed by the point (α, β) is $R_{p,\ell}$. By combining the optimal data structure for top-1 rectangle stabbing presented by Chan et al. [44] and the bound on the number of candidate lengths in Lemma 6, they prove the following result.

Theorem 12. *There exists an $\mathcal{O}(n \log n)$-word data structure which answers* rSUS *queries in $\mathcal{O}(\log_w n)$ time per query in the word RAM model, where $w = \Omega(\log n)$ is the word size.*

8. Discussion and Future Work

In this paper, we reviewed several types of shortest unique substring queries and their corresponding solutions. All the discussed problems in this manuscript are highly motivated topics in string processing and computational biology research areas. Although we discussed more than 10 approaches for variants of SUS queries, there still exist related topics which have not been studied, or there is no efficient algorithm to solve them. In this section, we discuss some of such topics and open questions for future work:

- We discussed all the solutions to solve approximate SUS queries in Section 5. However, there is no efficient in-place algorithm which can find LSUSs to get SUSs afterward. Another technique that can be applied to solve approximate SUS queries is considering the *RLE* representation of the input string. Section 4.2 shows this technique for solving interval-SUS queries. To our knowledge, an *RLE* based approach for solving approximate SUS queries has not been studied. In addition, there is no work considering the standard external memory model for solving an approximate SUS problem. As the I/O-efficient construction of the suffix array and lcp array exist [45–48], it seems to be possible to change the RAM model algorithm for the construction of these arrays to the external memory model.
- In Section 4.2, the $\pi_q(N, m)$ in the query time of Theorem 6 is $\sqrt{\log m / \log \log m}$, which is actually the time for performing dynamic predecessor/successor queries using $\mathcal{O}(|RLE(\mathcal{S}))$ space [8]. In order to make the query time faster using the same space, the question is if there exists a data structure of size $\mathcal{O}(|RLE(\mathcal{S})|)$ that can efficiently answer Problem 3 without using predecessor/successor.
- As we discussed in Section 6, palindromic substrings have great motivations in computational biology. All the reviewed works are on finding the exact SUPSs. Similar to the approximate SUS problem, approximate SUPS query is also important to be studied for considering errors and mutations. Besides the definition of Problem 5, the following definition has a great motivation in bioinformatics. A nucleotide sequence is considered as a palindrome if the reverse of its complementary strand is equal to the original sequence [49]. The question is if the methods discussed in Section 6 can be applied to efficiently solve this problem.
- The last topic that we discussed was the rSUS problem. According to Theorem 12, rSUS queries can be solved in $\mathcal{O}(\log_\omega n)$ time using a data structure of size $\mathcal{O}(n \log n)$ word. The question

is whether we can design an efficient $\mathcal{O}(n)$-word data structure for the rSUS problem. In addition, the approximate version of rSUS queries has not been studied. It is possible to combine the technique discussed in Section 7 and the framework of Thankachan et al. [50] to provide an efficient algorithm for approximate rSUS problem.

- Besides shortest unique substrings, *Maximal Unique Matches* is an important concept in computational biology for aligning two long genome sequences [51]. Ganguly et al. [18] applied a similar technique discussed in Section 3.4 to find maximal unique matches of two strings. As far as we are aware, the dynamic version (when mismatches are allowed) of this problem has not been studied yet. We believe that, by modifying the techniques on the dynamic longest common substring problem (LCS after k mismatches) [52–54], the approximate Maximal Unique Matches problem can be solved in subquadratic time.

Author Contributions: Methodology, writing—original draft, P.A.; writing—review and editing, M.O.K.; supervision, writing—review and editing, S.V.T. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported in part by the U.S. National Science Foundation (NSF) under CCF-1703489.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Pei, J.; Wu, W.C.H.; Yeh, M.Y. On Shortest Unique Substring Queries. In Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE), Brisbane, Australia, 8–11 April 2013; pp. 937–948.
2. Hu, X.; Pei, J.; Tao, Y. Shortest Unique Queries on Strings. In Proceedings of the String Processing and Information Retrieval-21st International Symposium—SPIRE 2014, Ouro Preto, Brazil, 20–22 October 2014; Lecture Notes in Computer Science; de Moura, E.S., Crochemore, M., Eds.; Springer: Cham, Switzerland, 2014; Volume 8799, pp. 161–172. [CrossRef]
3. Hon, W.; Thankachan, S.V.; Xu, B. In-place algorithms for exact and approximate shortest unique substring problems. *Theor. Comput. Sci.* **2017**, *690*, 12–25. [CrossRef]
4. Inoue, H.; Nakashima, Y.; Mieno, T.; Inenaga, S.; Bannai, H.; Takeda, M. Algorithms and combinatorial properties on shortest unique palindromic substrings. *J. Discrete Algorithms* **2018**, *52*, 122–132. [CrossRef]
5. Abedin, P.; Ganguly, A.; Pissis, S.P.; Thankachan, S.V. Range Shortest Unique Substring Queries. In Proceedings of the International Symposium on String Processing and Information Retrieval, Segovia, Spain, 7–9 October 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 258–266.
6. Ileri, A.M.; Külekci, M.O.; Xu, B. Shortest unique substring query revisited. In *Symposium on Combinatorial Pattern Matching*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 172–181.
7. Ileri, A.M.; Külekci, M.O.; Xu, B. A simple yet time-optimal and linear-space algorithm for shortest unique substring queries. *Theor. Comput. Sci.* **2015**, *562*, 621–633. [CrossRef]
8. Mieno, T.; Inenaga, S.; Bannai, H.; Takeda, M. Shortest Unique Substring Queries on Run-Length Encoded Strings. In Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, Kraków, Poland, 22–26 August 2016; LIPIcs; Faliszewski, P., Muscholl, A., Niedermeier, R., Eds.; Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2016; Volume 58, pp. 69:1–69:11. [CrossRef]
9. Allen, D.R.; Thankachan, S.V.; Xu, B. A Practical and Efficient Algorithm for the k-mismatch Shortest Unique Substring Finding Problem. In Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics—BCB 2018, Washington, DC, USA, 29 August–1 September 2018; Shehu, A., Wu, C.H., Boucher, C., Li, J., Liu, H., Pop, M., Eds.; ACM: New York, NY, USA, 2018; pp. 428–437. [CrossRef]
10. Allen, D.R.; Thankachan, S.V.; Xu, B. An Ultra-Fast and Parallelizable Algorithm for Finding k-Mismatch Shortest Unique Substrings. *IEEE/ACM Trans. Comput. Biol. Bioinform.* **2020**. [CrossRef] [PubMed]
11. Watanabe, K.; Nakashima, Y.; Inenaga, S.; Bannai, H.; Takeda, M. Shortest Unique Palindromic Substring Queries on Run-Length Encoded Strings. In Proceedings of the Combinatorial Algorithms-30th International Workshop, IWOCA 2019, Pisa, Italy, 23–25 July 2019; pp. 430–441. [CrossRef]

12. Watanabe, K.; Nakashima, Y.; Inenaga, S.; Bannai, H.; Takeda, M. Fast Algorithms for the Shortest Unique Palindromic Substring Problem on Run-Length Encoded Strings. *Theory Comput. Syst.* **2020**. [CrossRef]
13. Tsuruta, K.; Inenaga, S.; Bannai, H.; Takeda, M. Shortest Unique Substrings Queries in Optimal Time. In Proceedings of the SOFSEM 2014: Theory and Practice of Computer Science-40th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, 26–29 January 2014; Lecture Notes in Computer Science; Geffert, V., Preneel, B., Rovan, B., Stuller, J., Tjoa, A.M., Eds; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8327, pp. 503–513. [CrossRef]
14. Mieno, T.; Köppl, D.; Nakashima, Y.; Inenaga, S.; Bannai, H.; Takeda, M. Compact Data Structures for Shortest Unique Substring Queries. In Proceedings of the International Symposium on String Processing and Information Retrieval, Segovia, Spain, 7–9 October 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 107–123.
15. Schultz, D.W.; Xu, B. On k-Mismatch Shortest Unique Substring Queries Using GPU. In Proceedings of the Bioinformatics Research and Applications-14th International Symposium—ISBRA 2018, Beijing, China, 8–11 June 2018; pp. 193–204. [CrossRef]
16. Schultz, D.W.; Xu, B. Parallel Methods for Finding k-Mismatch Shortest Unique Substrings Using GPU. *IEEE/ACM Trans. Comput. Biol. Bioinform.* **2019**. [CrossRef]
17. Hon, W.; Thankachan, S.V.; Xu, B. An In-place Framework for Exact and Approximate Shortest Unique Substring Queries. In Proceedings of the Algorithms and Computation-26th International Symposium—ISAAC 2015, Nagoya, Japan, 9–11 December 2015; pp. 755–767. [CrossRef]
18. Ganguly, A.; Hon, W.K.; Shah, R.; Thankachan, S.V. Space-time trade-offs for the shortest unique substring problem. In Proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC 2016), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Sydney, Australia, 12–14 December 2016.
19. Haubold, B.; Pierstorff, N.; Möller, F.; Wiehe, T. Genome comparison without alignment using shortest unique substrings. *Bmc Bioinform.* **2005**, *6*, 123. [CrossRef] [PubMed]
20. Tarhio, J.; Peltola, H. String matching in the DNA alphabet. *Software Pract. Exp.* **1997**, *27*, 851–861. [CrossRef]
21. Adas, B.; Bayraktar, E.; Faro, S.; Moustafa, I.E.; Külekci, M.O. Nucleotide Sequence Alignment and Compression via Shortest Unique Substring. In Proceedings of the Bioinformatics and Biomedical Engineering-Third International Conference—IWBBIO 2015, Granada, Spain, 15–17 April 2015; Lecture Notes in Computer Science; Guzman, F.M.O., Rojas, I., Eds; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9044, Part II, pp. 363–374. [CrossRef]
22. Kim, H.; Han, Y.S. OMPPM: online multiple palindrome pattern matching. *Bioinformatics* **2016**, *32*, 1151–1157. [CrossRef]
23. Kolpakov, R.; Kucherov, G. Searching for gapped palindromes. *Theor. Comput. Sci.* **2009**, *410*, 5365–5373. [CrossRef]
24. Amir, A.; Apostolico, A.; Landau, G.M.; Levy, A.; Lewenstein, M.; Porat, E. Range LCP. *J. Comput. Syst. Sci.* **2014**, *80*, 1245–1253. [CrossRef]
25. Abedin, P.; Ganguly, A.; Hon, W.K.; Matsuda, K.; Nekrich, Y.; Sadakane, K.; Shah, R.; Thankachan, S.V. A linear-space data structure for range-LCP queries in poly-logarithmic time. *Theor. Comput. Sci.* **2020**, *163*, 245–251.
26. Kociumaka, T.; Radoszewski, J.; Rytter, W.; Waleń, T. Internal pattern matching queries in a text and applications. In Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms, Portland, OR, USA, 5–7 January 2014; SIAM: Philadelphia, PA, USA, 2014; pp. 532–551.
27. Weiner, P. Linear Pattern Matching Algorithms. In Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973), Iowa City, IA, USA, 15–17 October 1973; IEEE Computer Society: Washington, DC, USA, 1973; pp. 1–11. [CrossRef]
28. Manber, U.; Myers, G. Suffix arrays: A new method for online string searches. *Siam J. Comput.* **1993**, *22*, 935–948. [CrossRef]
29. Kärkkäinen, J.; Sanders, P. Simple linear work suffix array construction. In Proceedings of the International Colloquium on Automata, Languages, and Programming, Eindhoven, The Netherlands, 30 June–4 July 2003; Springer: Berlin/Heidelberg, Germany, 2003; pp. 943–955.
30. Fischer, J.; Heun, V. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM J. Comput.* **2011**, *40*, 465–492. [CrossRef]

31. Willard, D.E. Log-Logarithmic Worst-Case Range Queries are Possible in Space Theta(N). *Inf. Process. Lett.* **1983**, *17*, 81–84. [CrossRef]
32. Rubinchik, M.; Shur, A.M. EERTREE: An efficient data structure for processing palindromes in strings. In *International Workshop on Combinatorial Algorithms*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 321–333.
33. Ukkonen, E. On-line construction of suffix trees. *Algorithmica* **1995**, *14*, 249–260. [CrossRef]
34. Pei, J.; Wu, W.C.; Yeh, M. On shortest unique substring queries. In Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, 8–12 April 2013; Jensen, C.S., Jermaine, C.M., Zhou, X., Eds.; IEEE Computer Society: Washington, DC, USA, 2013; pp. 937–948. doi:10.1109/ICDE.2013.6544887. [CrossRef]
35. Aggarwal, A.; Vitter, Jeffrey, S. The input/output complexity of sorting and related problems. *Commun. ACM* **1988**, *31*, 1116–1127. [CrossRef]
36. Tamakoshi, Y.; Goto, K.; Inenaga, S.; Bannai, H.; Takeda, M. An opportunistic text indexing structure based on run length encoding. In Proceedings of the International Conference on Algorithms and Complexity, Paris, France, 20–22 May 2015; Springer: Berlin/Heidelberg, Germany, 2015, pp. 390–402.
37. Ulitsky, I.; Burstein, D.; Tuller, T.; Chor, B. The average common substring approach to phylogenomic reconstruction. *J. Comput. Biol.* **2006**, *13*, 336–350. [CrossRef]
38. Hooshmand, S.; Tavakoli, N.; Abedin, P.; Thankachan, S.V. On computing average common substring over run length encoded sequences. *Fundam. Informaticae* **2018**, *163*, 267–273. [CrossRef]
39. Thankachan, S.V.; Chockalingam, S.P.; Liu, Y.; Apostolico, A.; Aluru, S. ALFRED: A practical method for alignment-free distance computation. *J. Comput. Biol.* **2016**, *23*, 452–460. [CrossRef]
40. Bannai, H.; Gagie, T.; Inenaga, S.; Kärkkäinen, J.; Kempa, D.; Piątkowski, M.; Puglisi, S.J.; Sugimoto, S. Diverse palindromic factorization is NP-complete. In Proceedings of the International Conference on Developments in Language Theory, Liverpool, UK, 27–30 July 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 85–96.
41. Borozdin, K.; Kosolobov, D.; Rubinchik, M.; Shur, A.M. Palindromic length in linear time. In Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Warsaw, Poland, 4–6 July 2017.
42. Mali, P.; Esvelt, K.M.; Church, G.M. Cas9 as a versatile tool for engineering biology. *Nat. Methods* **2013**, *10*, 957–963. [CrossRef] [PubMed]
43. Manacher, G. A New Linear-Time"On-Line"Algorithm for Finding the Smallest Initial Palindrome of a String. *J. ACM (JACM)* **1975**, *22*, 346–351. [CrossRef]
44. Chan, T.M.; Larsen, K.G.; Patrascu, M. Orthogonal Range Searching on the RAM, Revisited. In Proceedings of the 27th Annual Symposium on Computational Geometry 2011, Paris, France, 13–15 June 2011; pp. 1–10.
45. Kärkkäinen, J.; Kempa, D.; Puglisi, S.J. Parallel external memory suffix sorting. In *Annual Symposium on Combinatorial Pattern Matching*; Springer: Berlin/Heidelberg, Germany, 2015, pp. 329–342.
46. Kärkkäinen, J.; Kempa, D.; Puglisi, S.J.; Zhukova, B. Engineering external memory induced suffix sorting. In Proceedings of the 2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX), Barcelona, Spain, 17–18 January 2017; pp. 98–108.
47. Kärkkäinen, J.; Kempa, D. Faster external memory LCP array construction. In Proceedings of the 24th Annual European Symposium on Algorithms (ESA 2016), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Aarhus, Denmark, 22–24 August 2016.
48. Kärkkäinen, J.; Kempa, D. LCP array construction using O (sort (n))(or less) I/Os. In Proceedings of the International Symposium on String Processing and Information Retrieval, Beppu, Japan, 18–20 October 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 204–217.
49. Anjana, R.; Shankar, M.; Vaishnavi, M.K.; Sekar, K. A method to find palindromes in nucleic acid sequences. *Bioinformation* **2013**, *9*, 255. [CrossRef] [PubMed]
50. Thankachan, S.V.; Aluru, C.; Chockalingam, S.P.; Aluru, S. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In Proceedings of the International Conference on Research in Computational Molecular Biology, Paris, France, 21–24 April 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 211–224.
51. Delcher, A.L.; Kasif, S.; Fleischmann, R.D.; Peterson, J.; White, O.; Salzberg, S.L. Alignment of whole genomes. *Nucleic Acids Res.* **1999**, *27*, 2369–2376. [CrossRef]

52. Kociumaka, T.; Radoszewski, J.; Starikovskaya, T. Longest common substring with approximately k mismatches. *Algorithmica* **2019**, *81*, 2633–2652. [CrossRef]
53. Abedin, P.; Hooshmand, S.; Ganguly, A.; Thankachan, S.V. The heaviest induced ancestors problem revisited. In Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Qingdao, China, 2–4 July 2018.
54. Flouri, T.; Giaquinta, E.; Kobert, K.; Ukkonen, E. Longest common substrings with k mismatches. *Inf. Process. Lett.* **2015**, *115*, 643–647. [CrossRef]