Sub-linear Algorithms for Shortest Unique Substring and Maximal Unique Matches

by

Sanjeewa Bandara Senanayaka

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science

(Computer Science)

At

The University of Wisconsin–Whitewater

December, 2019

Graduate Studies

The members of the Committee approve the thesis of
Sanjeewa Bandara Senanayaka
presented on (December 6^{th}, 2019 of Oral Exam)

Dr. Arnab Ganguly, Chair

Dr. Athula Gunawardena

Dr. Hien Nguyen

Abstract

Living in an era where we produce quintillions of data on a daily basis means that there are many intriguing (possibly unrevealed) facts behind them. Some of the data that exists in large quantities is encountered in Computational Biology; our DNA strands are extremely long ($\approx$ 3.2 billion characters). Genomic problems such as comparing similarities between two strands of DNA or finding the uniqueness between two similar strands of DNA can hugely benefit in the understanding of evolution and possibly create medical advances. A couple of avenues of addressing a problem of similar flavor (known as gene alignment) are using *Shortest Unique Substrings* and *Maximal Unique Matches*. Existing techniques for these problems either necessitate the use of high performance computers, or using techniques that have large memory footprints, or use of (theoretical) techniques that are complicated to construct.

Prior to our work, Ganguly et al. [TCS' 17] showed that these problems can be solved in $O(n\tau^2 \log \frac{n}{\tau})$ time and $O(n/\tau)$ space for any parameter $\tau = \omega(1)$ and $\tau = o(n)$. However, these techniques require the use of complex data structures and techniques, which makes them practically intractable. Presented here are sub-linear algorithms for the *Shortest Unique Substring problem* and the *Maximal Unique Matches problem*. Besides using minimal amounts of work space and being (reasonably) fast, our algorithms are a combination of two simple techniques – Rabin-Karp fingerprint and Bloom Filters; this makes our work easily amenable to implementations. However, our algorithms suffer from a bounded error probability, which can be reduced using (slightly) more space.

ACKNOWLEDGMENTS

First I would like to thank my thesis advisor Dr. Arnab Ganguly of the Computer Science department at University of Wisconsin-Whitewater. Dr. Ganguly has always been available to assist me and pull me out of tough spots whenever I had any questions or trouble during my research. I appreciate his patience and thoroughness throughout my research. He was able to provide me with guidance that allowed me to excel.

I would also like to thank the experts, Dr. Athula Gunawardena and Dr. Hien Nguyen who were a part of my thesis defense committee. Their passionate participation and their willingness to help allowed me to successfully finalize my research.

I would also like to acknowledge Dr. Jiazhen Zhou, Dr. Lopamudra Mukherjee, Dr. Sobitha Samaranayake, Dr. Zachary Oster and the entire department of Computer Science at University of Wisconsin-Whitewater for being great teachers and for providing all the knowledge and support required for a student to grow.

Finally, I must express my very profound gratitude to my parents and family for providing me with unconditional support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them. Thank you.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

DISCARD THIS PAGE

LIST OF TABLES

Table Page

Appendix
Table Page

DISCARD THIS PAGE

LIST OF FIGURES

Chapter 1

Introduction

Bernard Marr, in the article [BM], "How much data do we create every day? The mind-blowing stats everyone should know", states that around 2.5 quintillion bytes of data are created daily by internet users. Many researchers from a variety of fields (such as Computer Science, Computational Biology/Physics, and Financial Market analysts) have taken up the arduous task of mining this enormous pool of data. A large share of this data can be visualized as a string, i.e., as a sequence of characters, which is not really surprising as in some sense, any data that is used for computation is serialized and stored in the form of a string. Consequently, string data forms the basic backbone of a plethora of engineering tools. This wide-spread use and applicability of string data naturally brings in the demand of efficient ways to store and analyze the data. For example, consider the Illumina next-generation sequencing (NGS) technology, which generates numerous short reads of an organism; the task is to align these short reads to create the entire genome of the organism. Or, consider the 1000 Genome project which is a comprehensive collection of human genome information. At the core of this storage and retrieval lies the classical problem of pattern matching: *find the occurrences of a given string (called pattern) in another (possibly much longer) string (called text)*. This problem forms the backbone of most string related analysis tools.

Early Developments. The text-book algorithms for the pattern matching problem are the Knuth-Morris-Pratt algorithm [KJP77], Boyer-Moore algorithm [BM77], and the Rabin-Karp

algorithm [KR87]. Although these are (near) optimal when the text and the pattern are both provided at query time, these are too slow in the indexing setting: *the text is provided beforehand and is fixed, and the pattern is provided later as a query.* The primary reasons is that these algorithms scan through the entire text for finding occurrences of the pattern, which is rather wasteful, especially when we take into account that texts are much longer than patterns. Moroever, often answers to more involved queries are needed, such as report occurrences with a bounded number of mismatches allowed, report only the occurrences which are non-overlapping, estimate the average distance between consecutive occurrences, report the shortest unique substring. What we need are indexes, i.e., data structures built by pre-processing the text that can be used to answer pattern matching queries, without having us to read through the text completely.

Indexing. Peter Weiner in 1973 invented the *Suffix Tree* data structure [Wei73], which is essentially a compacted trie built out of all the suffixes of the text T (c.f. for more formal definitions). Using the suffix tree, one can report all the occurrences of the pattern in time proportional to the length of the pattern and the number of occurrences. In fact, the suffix tree is much more versatile – it forms the basic building block of many other indexes for answering a host of other interesting queries (such as approximate pattern matching [BGVV14, CEPR10, CGL04, TWLY09], and document retrieval [HTSV13, Mut02, Nav13, NN12]) that are required in many practical settings. Later, a space efficient alternative (constant factor improvements) known as the suffix array (along with Longest-Common-Prefix array or LCP array) was developed [MM90], which still answers pattern matching queries almost as fast.

Succinct Text Indexes. Although suffix trees/arrays are fast and relatively easy to implement, they occupy too much space compared to the string itself. This effect becomes even more pronounced when it comes to the case of DNA (or scenarios where the text length is much larger compared to the alphabet size from which the characters in the text are drawn from). Take for example the human genome which contains approximately 3.2 billion characters from

the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, where the alphabet size $\sigma = 4$. The human genome can be stored in about 1GB of space (or even lower using compression techniques); a suffix tree of the same, on the other hand, even with a space-efficient implementation such as in [Kur99] occupies almost 40GB. Even with the big boom in processing power over the past years, it is still not to a point where an average computer could afford this much space in memory. This clearly suggests that there is a need for further study on alternative methods than using Suffix Trees, not only because of the theoretical intrigue, but also due to its potential applications in the analysis of synthetic chromosomal regions, understanding of evolution, and gene replication [DKF$^+$99].

To formalize the notion, notice that each character in a text T of length n over an alphabet Σ of size σ can be encoded in $\lceil \log \sigma \rceil$ bits; therefore, the total number of bits needed to store T is $n\lceil \log \sigma \rceil$ bits. The suffix tree (or even the suffix array), on the other hand, requires $\Theta(n \log n)$ bits of space; this space blowup factor of $O(\log_\sigma n)$ becomes prohibitively large when $n \gg \sigma$, which is precisely the case in context of genomic data.

To alleviate the above scenario, one work around is to use (rather expensive) computers with large memory handling capacity. However, most commodity hardware is simply incapable of handling storage of this volumnious capacity. The other approach is to look for alternate avenues, chiefly among which are using small spaced data structures and algorithms with low memory footprint, possibly at the cost of computation time. So what this demands is *indexes that provide space-efficient storage of data, yet at the same time supports fast application-specific queries.*

Unfortunately, space-efficient storage and time-efficient analysis of data are often conflicting goals; evidence can be found in the fact that it took us almost three decades after the invention of suffix trees (and about a decade after the invention of suffix arrays) to come up with a space-efficient alternative. More specifically, Ferragina and Manzini [FM00, FM05] and Grossi and Vitter [GV00, GV05] introduced their respective succinct text indexes, known respectively as the *FM Index* and *Compressed Suffix Array (CSA)*. These indexes occupy space $n \log \sigma$ bits plus lower order terms and can report all the occ occurrences of a pattern P of length p in time $O((p + occ) \log^c n)$, where $c > 1$ is an arbitrary small constant.

CSA exploits a crucial structural component (known as suffix links) of the suffix tree/array to achieve compression; FM Index, on the other hand, relies on a reversible transformation of the text, known as the *Burrows-Wheeler Transformation (BWT)* [BW94]. Subsequent developments [BOSS12, FLMM09, MBN$^+$17, NM07, OV11, Sad02, Sad07, SOG12] led to compressed indexes that capture the suffix tree in near-optimal number of bits (w.r.t information theoretic lower bound). Apart from the provably good guarantees, FM Index is also highly practical; for the human genome, it occupies space less than 1.5GB, and query answering is also reasonably fast. Hence, it is now the backbone of numerous Bioinformatics tools such as the Burrows-Wheeler Aligner (BWA) [LD10], Bowtie [LTPS09], and others [APPA15, APPA16, KLL$^+$12, LSM12, LYL$^+$09, SD10].

The Caveats of (Succinct) Indexing. Our problems are in the algorithmic setting, where the string is provided online as a query and the goal is to report some information about the string. Therefore, what we are interested is the working space (i.e., the peak memory usage when the algorithm runs). We could follow the suffix tree/suffix array route, whereby we build the suffix tree/array of the string and use it to report the desired information; however, as explained earlier, this encompasses a large memory footprint (that of the working space needed to build the respective structures). Using succinct indexing (in form of FM Index or CSA) is another option; however, it presents us with the following problems:

- Either, we have to implement algorithms [HSS03, Bel14, MNN17] for constructing the FM Index/CSA in compact space, i.e., $O(n \log \sigma)$ bits of space. These algorithms are (arguably) difficult to implement and (possibly) involve high constant factors.
- Or, resort to slower (small space) FM Index/CSA construction algorithms, which although useful in the indexing setting, not particularly helpful in the online setting.
- Or, use suffix array to construct the FM Index/CSA, which kind of defeats the purpose (as we consider the online setting, where the working space is the deciding factor).

In a nutshell, we need fast algorithms that require a small working memory.

Sub-linear Algorithms. In the sub-linear setting, the string of length n is read-only (we can access it but cannot change it); however, the residual allowed space is sub-linear in the number of bits needed to represent the string (i.e., $n\lceil\log\sigma\rceil$ bits). Although the Knuth-Morris-Pratt (KMP) [KJP77] algorithm works in this setting in some-sense (as it only needs access to the text and maintains a table of size equaling the length of the pattern), it cannot be extended (at least obviously) to our problems. For us, the more useful technique is Rabin-Karp (RK) fingerprint [KR87]. We draw inspiration from the work of Porat and Porat [PP09] who discovered a pattern matching algorithm by using a sketch of just size $O(\log n)$ words (i.e., the memory footprint is $O(\log n)$ words), by using RK fingerprint in a remarkably elegant way. We employ similar strategies, whereby we use RK fingerprint coupled with Bloom Filter [Blo70], which is another sketching technique for answering membership queries (i.e., is an element present in a set or not).

We are now equipped to present the problems and existing results.

1.1 The Problems

Analysis of our genetic code plays an important role in understanding our evolutionary backgrounds, structure, function of certain human characteristics, etc. This area of study had given raise to the *Gene Alignment* problem, which involves aligning DNA, RNA, or protein to identify the similarity between organisms, as a step to understand different aspects of evolution. Among the numerous techniques (see for e.g. [LD10, LTPS09, APPA15, APPA16, KLL$^+$12, LSM12,LYL$^+$09,SD10,LD09] and references therein) for solving the gene alignment problem, two popular techniques are using *Shortest Unique Substrings* and *Maximal Unique Matches*.

Definition 1. *A **Shortest Unique Substring** (SUS) of a string X is a sub-string $X[i,j]$ that satisfies the following properties:*

- *$X[i,j]$ appears exactly once in X*
- *There exists no other substring $X[i',j']$ such that it is unique and $j'-i' < j-i$*

For example, consider the following string: $X = babaccc$. An SUS is **ab** since this is a shortest substring of X that appears only once.

Problem 1 (Shortest Unique Substring Problem [PWY13])**.** *Given a string T, find a shortest unique substring of T.*

Definition 2. *A **Maximal Unique Match** (MUM) of two strings X and Y is a sub-string $X[i, j]$ of X that satisfies the following properties:*

- *$X[i, j]$ appears exactly once in X,*
- *$X[i, j]$ appears exactly once in Y, and*
- *$X[i, j + 1]$ or $X[i - 1, j]$ does not appear in Y*

For example, consider the two strings: $X = eabcdcebea$ and $Y = cabccebdac$.

Substring	MUM?	Reason
abc	Yes	Unique for X and Y. Left extension eabc or right extension abcd does not appear in Y.
ea	No	Appears multiple times in X.
ebe	No	Unique in X, but not present in Y.
ab	No	Unique for X and Y. However, the right extension abc exists in Y.

Problem 2 (Maximal Unique Matches Problem [DKF$^+$99])**.** *Given two strings T_1 and T_2, find a shortest maximal unique match of T_1 and T_2.*

1.2 Applications

Introduced by Pei et al. [PWY13], some of the common applications of *Shortest Unique Substrings* include finding the uniqueness between two closely related organisms [HPMW05], polymerase chain reaction (PCR) primer design through the analysis of DNA sequences [PWY13], genome mapability [DEMS$^+$12], and next-generation short reads sequencing [ABF$^+$15]. They also listed other potential applications, such as document searching on the internet.

For genome alignment, one popular technique [DKF+99, DPCS02] uses the concept of *Maximal Unique Matches* as an intermediary step. More formally, in this intermediary step, one has to compute the maximal unique matches of the genes that are to be aligned. In fact, this principle has been used to develop one of the most popular softwares used for genome alignment, known as MUMmer [DKF+99].

1.3 Previous Work and Motivation

Shortest Unique Substring. Pei et al. [PWY13] introduced a similar problem to that of Problem 1, where we are required to find a SUS covering a given position k in the input string. Using the suffix tree of the input string, they presented an $O(n)$-time and $\Theta(n)$-word solution. Ileri et al. [IKX14] (and Tsuruta et al. [TIBT14] independently) discovered algorithms with competitive practical performance. More importantly, in comparison to the algorithm of Pei et al. [PWY13], they showed that their algorithm not only saves space by a factor of 20, but also attains a speedup by a factor of 4. This is achieved by replacing the suffix tree with a combination of the *Suffix Array*, its inverse, and the *LCP array*. Hon et al. [HTX15] achieved further space improvements by introducing an in-place framework. Specifically, their algorithm needs space $2n$ words in addition to the input string. We refer to [BC14,HPT14,MIBT16] for closely-related works.

Maximal Unique Matches. For Problem 2, Delcher et al. [DKF+99] used the Generalized Suffix Tree of X and Y to devise an $O(n)$ time and $\Theta(n)$ space algorithm for the MUM problem; here and henceforth, $n = |X| + |Y|$ is the total length of the two strings.[1] The software MUMmer 1.0 presented an implementation, which was subsequently improved (a factor of three in terms of space-efficiency) to MUMmer 2.0 in [DPCS02]. Later, Kurtz et al. [KPD+04] developed a more space-efficient version in MUMmer 3.0. In MUMmer 1.0,

[1]The generalized suffix tree (GST) [Gus97] of a collection of strings of total length n is a compact trie that stores all the suffixes of every string in the collection. The GST occupies $O(n)$ words of space.

aligning the 4.7 Mb genome of *E.coli* requires 293 megabytes (MB) of memory. MUMmer 2.0 improved to 102 MB of memory, and then MUMmer 3.0 reduced the space to 77 MB.

The Case for Sub-linear Algorithms. Unfortunately, the underlying use of suffix trees (or equivalent data structures) leads to high memory consumption, making it rather impractical for huge volumes of data. The space occupancy issue is more profound in the typical applications where suffix trees find use, that of DNA, in which the alphabet has size four, but the lengths of the strings (such as in Human Genome) are typically in the billions.[2] Even with a space-efficient construction/implementation, such as in [McC76, Ukk95, Kur99], a suffix tree occupies 40 Gigabytes, whereas the input Human Genome can be stored under one Gigabyte (possibly using compression techniques). Since primary applications of both the problems discussed here involves DNA, the use of suffix trees presents a serious bottleneck; evidence can be found in the experimental results of Ileri et al. [IKX14], who were not able to run Pei et al.'s algorithm [PWY13] on large data. Likewise in the case of the MUM problem, despite the improvements in the MUMmer softwares, the peak memory usage is still roughly 15 times the size of the input sequence. The primary reason for this blowup in space is again due to the use of (generalized) suffix trees.

The reason for this blowup space can be understood if we look at the difference in the number of bits required to store the input string(s) and the space in bits needed by the (generalized) suffix tree. If the input string(s) is of (total) length n, then the space needed to store them is $\lceil n \log \sigma \rceil$ bits, where σ is the size of the alphabet from which characters are drawn. The suffix tree, on the other hand, requires $O(n)$ words, or equivalently $O(n \log n)$ bits in the Word-RAM model.[3] This $O(\log_\sigma n)$ blowup in space becomes prohibitive when one considers the typical setting: DNA or RNA has $\sigma = 4$ and n in billions. One can make slight

[2]For example, in its effort to compare "E.coli K12 vs E.coli O157:H7", the peak memory usage of the latest version [KPD$^+$04] is 77MB of MUMmer (version 3.0), whereas according to [BO98]: "chromosome lengths among natural strains of E. coli can differ by as much as 1Mb, ranging from 4.5 to 5.5 Mb in length". As seen from this, the data has blown up by over 10 times!

[3]In the RAM model, the word size is $\Theta(\log n)$ bits.

improvements (constant factor) by replacing the suffix tree with a combination of suffix array and longest-common-prefix (LCP) array, but still that is too large for these instances.

For the MUM problem, Hon and Sadakane [HS02] showed that the problem can be solved in $O(n \log n)$ time using $O(n \log \sigma)$ bits of working space. The main ingredient is the use of FM-Index (in place of the suffix tree). One can slightly modify the techniques of Belazzougui and Cunial [BC14] to obtain an algorithm that needs $O(n \log \sigma)$ bits of working space for the SUS problem. Although this improves the suffix tree/suffix array based solutions by an $O(\log_\sigma n)$ factor, one still needs constant factor (multiplicative) overhead on the actual number of bits occupied by the input strings.

The first sub-linear solutions for these problems were obtained by Ganguly et al. [GHST17]; they showed that the problem can be solved in $O(n\tau^2 \log \frac{n}{\tau})$ time and $O(n/\tau)$ space for any parameter $\tau = \omega(1)$ and $\tau = o(n)$. Although, this can obtain arbitrary space improvements (in the sub-linear model), it suffers from a drawback, that of limited applicability due to the use of sophisticated data structures and algorithms, which are typically tricky to implement (and may not be practically feasible). We aim to bridge this gap by supplanting theoretical results with their practical counterparts.

Chapter 2

Preliminaries

Throughout this paper, we use the Word-RAM model of computation. In this model, each word consists of $\Theta(\log n)$ bits and all operations on a single word take constant time. For any string S, the i^{th} character of the string is given by $S[i]$, where $1 \leq i \leq |S|$. The concatenation of two strings S and S' is denoted by $S \circ S'$.

2.1 Trie

A trie is basically a tree that stores a collection of strings $\{S_1, S_2, S_3, \ldots, \}$, where s_i is the length of S_i and $n = \sum_{i=1}^{n} s_i$. Initially the trie consists of a root node. During the insertion of the first string $S_1 = S_1[1] \circ S_1[2] \circ \cdots \circ S_1[s_1]$, we form a sequence of nodes as follows: $\langle root = v_0, v_1, \ldots, v_k \rangle$, where v_i is the parent of v_{i+1} and the edge from v_i to v_{i+1} is labeled by $S_1[i+1]$, where $0 \leq i < s_1$.

Once S_i is inserted, the insertion of S_{i+1} is done by traversing down the tree using each character in S_{i+1} until a mismatch occurs, say at a node v after matching x characters. Now, we insert characters $S_{i+1}[x+1], S_{i+1}[x+2], \ldots, S_{i+1}[s_{i+1}]$ by treating v as the root.

A suffix trie is a trie of all the suffixes in a text $T[1, n]$. Before creating the suffix trie a \$ is attached to the end of T so that this will be prefix-free.[1] This ensures that each leaf in the trie corresponds to a suffix. The leaves are arranged from left-to-right in the lexicographic order of the suffix they represent. Additionally, note that each leaf corresponds to a unique suffix of T;

[1] A set of strings is prefix-free if no string appears as the prefix of another string in the collection.

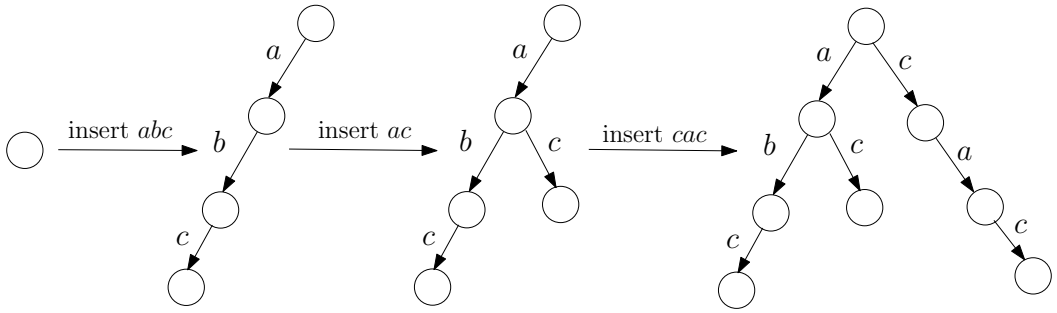


Figure 2.1: Trie of the strings $\{abc, ac, cac\}$

we augment each leaf with the starting position of the corresponding suffix.

To match a pattern P, simply traverse the suffix trie for every character of P. There are two possible outcomes.

- We have completely matched P and the last traversed edge is from u to v. We call v as $\mathsf{locus}(P)$. Then the suffixes that begin with P lie in the subtree of v.
- We have failed to match P. This means that the pattern P does not exist in T.

All the occurrences of the *P* in *T* can be found in $O(p + occ)$, where p is the length of pattern *P* and *occ* is the number of occurrences of *P* in *T*.

In the worst case, where every character in *T* is unique, a node will be created for each character of each suffix in the text. Thus, there will be a total of $O(n^2)$ nodes, thereby making the size of the trie quadratic in the length of the text.

2.2 Suffix Trees and Suffix Array

A suffix tree is a compact version of the suffix trie, which can be obtained as follows. Let v_1 and v_2 be two consecutive nodes on a root to leaf path in the suffix trie. Let v_0 be the parent of v_1. If v_2 is the only child of v_1, then remove v_1 and create a single edge from v_0 to v_2, where the label of this new edge is the concatenation of the labels of the edges from v_0 to v_1 and v_1 to v_2. This ensures that the suffix tree of $T[1, n]$ has at most $2n - 1$ nodes.

A suffix array **SA** is an array of length n that maintains a lexicographic arrangement of all the suffixes of T. More formally, $\mathsf{SA}[i] = j$ and $\mathsf{ISA}[j] = i$ iff the i^{th} lexicographically smallest suffix starts at position j .

Every node in the suffix tree is equipped with perfect hashing [FKS84] so that one can find the edge (or detect there is none) from that node beginning with a particular character in $O(1)$ time. Each edge stores two integers a and b, such that the edge is labeled by $T[a] \circ T[a+1] \circ \cdots \circ T[b]$. At every node u, we store two integers L and R, where $(L-1)$ and $(R-1)$ are respectively the number of leaves to the left of the leftmost and the rightmost leaf under u. Note that L and R are the leftmost and rightmost indexes such that for every $i \in [L, R]$, $\mathsf{SA}[i]$ is the starting position of all those suffixes that are prefixed by $\mathsf{str}(u)$, the concatenation of edge labels from root to u.

Note that the suffix array is just a permutation of $\{1, 2, \ldots, n\}$. Hence, the suffix tree and the suffix array can be stored in $O(n)$ words (or equivalently, in $O(n \log n)$ bits).

To find the occurrences of pattern *P*, find the locus node in the same process as the suffix trie; locus can be found in $O(|P|)$ time. Upon finding the locus node, the occurrences of P can be found by reporting $\mathsf{SA}[i]$ for every $i \in [L, R]$. The time for reporting all the occurrences of $P[1, p]$ is $O(p + occ)$, where occ is the number of occurrences of P in T.

Since suffix trees consume too much space, one of the challenges in the initial stages was to support pattern matching using an index of smaller size. Indeed, pattern matching can be done using only the suffix array and the original text. As suffix arrays group suffixes together in lexicographic order, this can be used to find all the positions where the pattern occurs. We provide a brief sketch below (for more details, see [Gus97]):

- Use binary search to find $[L, R]$ such that L is the smallest index and R is the largest index in the suffix array, where for any $i \in [L, R]$, the suffix at $\mathsf{SA}[i]$ starts with the character $P[1]$.

- Starting from $[L, R]$ recursively use the same idea to find a maximal range $[L', R']$ such that for any $i \in [L', R']$, the suffix at $\mathsf{SA}[i]$ starts with the string $P[1] \circ P[2] \circ \ldots$

At any point in the process above if $L > R$, then the pattern does not exist in the text; otherwise, at the end we will end with the range $[L, R]$ in the suffix array such that for any $i \in [L, R]$, the suffix at $\mathsf{SA}[i]$ starts with P. Report these occurrences using the suffix array. Since each binary search operation takes $O(\log n)$ time, all the occurrences of the pattern are retrieved in time $O(|P| \log n + occ)$.

2.3 Longest Common Prefix

Define $\mathsf{lcp}(X, Y)$ to be the length of the longest common prefix between two strings X and Y. For example, consider the strings *X* = *cbacbabcb*, *Y* = *cbaccbacb* and *Z* = *acbabcabc*. Then,

v, w	$\mathsf{lcp}(v, w)$	Reason
v = *X*, *w* = *Y*	4	The longest common prefix between X and Y is **cbacb**
v = *X*, *w* = *Z*	0	There is no common prefix

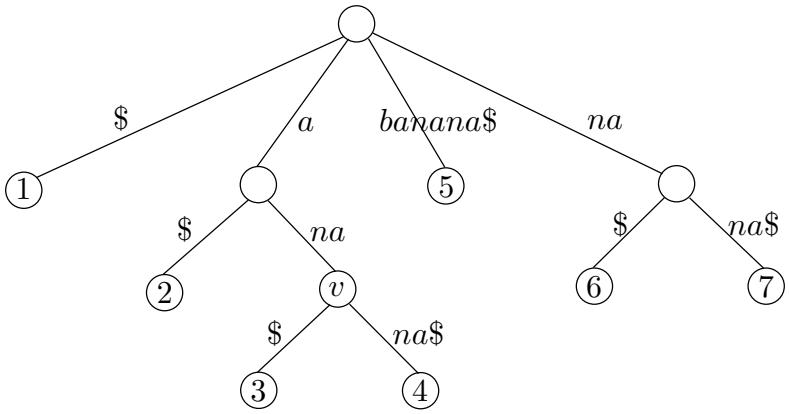
The longest common prefix array (LCP array) is an array $\mathsf{LCP}[1, n]$ that maintains the length of the longest common prefixes of every consecutive suffix in lexicographic order. More specifically,

$$\mathsf{LCP}[n] = \perp$$
$$\mathsf{LCP}[i] = \mathsf{lcp}\Big(T\big[\mathsf{SA}[i], n\big], T\big[\mathsf{SA}[i+1], n\big]\Big), \quad \text{when } i < n$$

Using the Manber-Myers algorithm [MM90], we can compute both the suffix array and the LCP array in $O(n \log n)$ time.

2.4 Bloom Filters

A bloom filter is a space-efficient probabilistic data structure based on hashing that can be used to answer membership queries: *is a number from the universe* $\{1, 2, \ldots, U\}$ *in the data structure or not.* More specifically, a bloom filter is a bit-array BF of size m, which is



i	1	2	3	4	5	6	7
SA$[i]$	7	6	4	2	1	5	3
ISA$[i]$	5	4	7	3	6	2	1
LCP$[i]$	0	1	3	0	0	2	$\perp$

Figure 2.2: Suffix Tree, Suffix Array, and LCP Array for the string $banana\$$

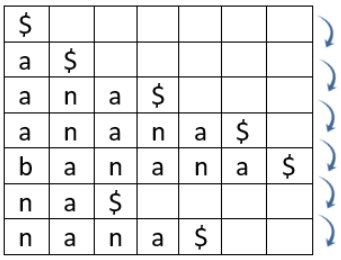


Figure 2.3: Illustration of how to compute LCP array for the string $banana\$$

augmented with β independent hash functions $h_1, h_2, \ldots, h_\beta$, each of which maps a key from $[1, U]$ to $[1, m]$. Initially all entries of the bloom filter are initialized to zero.

- To insert a key k into the Bloom Filter, simply set $BF[h_i(k)] = 1$ for $i = 1, 2, \ldots, \beta$.
- To answer a search query for a key k return $true$ if $BF[h_i(k)] = 1$ for $i = 1, 2, \ldots, \beta$; otherwise, return $false$.

Clearly, the space required is $\Theta(m)$ bits. Also, any operation requires $\Theta(\beta)$ time, assuming that hash values are computed in constant time.

Suppose, a search query one a key k returns $true$. This may be a false positive (i.e., k is actually not in the Bloom filter) because we may have one or more keys that have set the exact same bits to 1 as those set by the hash functions for the key k. However, if we have a reasonable large number of hash functions and a large enough hash table, then the error probability is well-bounded.

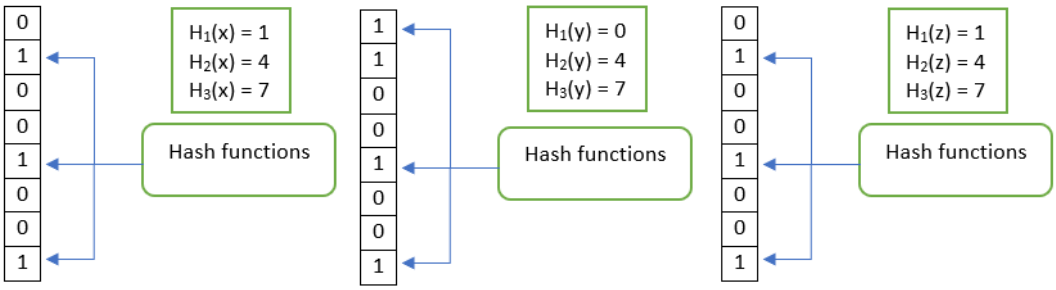


Figure 2.4: Insertion on elements x, y, and z with collision on z

2.5 Rabin-Karp Fingerprint

The Rabin-Karp (RK) Fingerprint algorithm is a string searching algorithm that uses a *rolling hash function*, known as the RK fingerprint, to find positions in the text where a pattern occurs. We sketch here some of the key properties of Rabin-Karp fingerprint.

Let S be a string over an alphabet of size σ, and $p > |S|$ be a prime number. The fingerprint of S is

$$\Phi(S) = \sum_{k=1}^{|S|} S[k] \cdot \sigma^{k-1} \mod p$$

Rolling-Hash Property. For a string $S = xS'$ and two characters x and y, given $\Phi(S)$ and $\sigma^{|S|-1} \mod p$, we can compute $\Phi(S'y)$ in constant time as follows:

$$\begin{aligned}
\Phi(S'y) &= \left(\sum_{k=1}^{|S'|} S'[k] \cdot \sigma^k + y \right) \mod p \\
&= \left(x \cdot \sigma^{|S|} + \sum_{k=1}^{|S'|} S'[k] \cdot \sigma^k - x \cdot \sigma^{|S|} + y \right) \mod p \\
&= \left(\sigma \cdot \left(x \cdot \sigma^{|S|-1} + \sum_{k=1}^{|S'|} S'[k] \cdot \sigma^{k-1} \right) - x \cdot \sigma^{|S|} + y \right) \mod p \\
&= \left(\sigma \cdot \left(\Phi(S) - x \cdot \sigma^{|S|-1} \right) + y \right) \mod p \\
&= \left(\sigma \cdot \left(\Phi(S) - x \cdot (\sigma^{|S|-1} \mod p) \right) + y \right) \mod p
\end{aligned}$$

Chapter 3

Computing Shortest Unique Substring

Throughout this section, we consider the scenario where we are trying to find the shortest unique substring SUS of a given string X of length n.

3.1 Using Suffix Array and LCP array

We first obtain the Suffix Array $\mathsf{SA}[1, n]$ and the LCP array $\mathsf{LCP}[1, n]$ for the input string X of length n. Both these arrays are computed in $\Theta(n \log n)$ time by using the Manber-Myers construction algorithm [MM90]. To compute the SUS, the main idea is to compute the longest prefix of each suffix that occurs more than once; the SUS is clearly one more than the shortest of such prefixes. Following is the outline of the algorithm:

- Initalize $sus \leftarrow n$ and $susIndex \leftarrow 1$
- For $(i = 2, 3, \ldots, n)$, do the following:
  - $longestRepeatPrefix \leftarrow max\{\mathsf{LCP}[i-1], \mathsf{LCP}[i]\}$
  - if ($longestRepeatPrefix+1 < sus$) and ($\mathsf{SA}[i]+longestRepeatPrefix \leq n$)
    - $sus \leftarrow longestRepeatPrefix + 1$
    - $susIndex \leftarrow \mathsf{SA}[i]$

Clearly, the algorithm above finds SUS in $O(n \log n)$ time and uses $O(n \log n)$ bits of working space.

3.2 Our Approach

We use the rolling fingerprint technique from Rabin-Karp fingerprint and two bloom filters BF_1 and BF_2. To get decent probability bounds, we choose an appropriate number, say some constant $c > 1$, of prime numbers, where each prime number is in $\Theta(n)$. For each prime number, we will have a RK hash-function $h_1, h_2, \ldots, h_c$.

The main intuition is as follows: *the length of the SUS is the smallest δ for which there exists a unique substring of length δ.* Based on this, following is the outline of how the algorithm works.

Compute SUS

Return the smallest $\delta \in \{1, 2, \ldots, n\}$ for which there is a unique substring of length δ

Is there a unique substring of length δ?

- Run the bloom filter setting phase
- If the bloom filter reading phase returns 'YES', the return 'YES, else return 'NO'

Setting Phase

- For a window of size δ, compute a set of Rabin-Karp (RK) hash values $h_1, h_2, \ldots, h_c$. Initially, the first δ characters of the string form the first window.
- For each of the computed hash values $\{h_1, h_2, \ldots, h_c\}$, check the corresponding bloom filter BF_1 index. If all the indexes are already set to 1, then for these indexes set BF_2 to 1, else set the corresponding index in BF_1 to 1.
- Slide the window δ by one character, i.e., remove the first character in the window and include the next character of the string within the window. If there is no next character, then end the process.

Reading Phase

- For a window of size δ, compute a set of Rabin-Karp (RK) hash values $h_1, h_2, \ldots, h_c$. Initially, the first δ characters of the string form the first window.
- For each of the computed hash values $\{h_1, h_2, \ldots, h_c\}$, check the corresponding bloom filter BF_2 index. If the index is not set, then the corresponding δ-length string has appeared only once in the string. Return 'YES'.
- Slide the window δ by one character, i.e., remove the first character in the window and include the next character of the string within the window. If there is no next character, then return 'NO'.

Refer to Algorithm 1 for a detailed pseudo-code.

3.3 Illustration

The procedure for finding a SUS for the string $X = babaccc$ can be seen in Table 3.1. This technique however can result in false negatives (i.e., it may skip over a unique string without recognizing it as one). Refer to Table 3.2 to see a scenario where SUS is missed for the string $X = abcabdcacb$.

Algorithm 1 Shortest Unique Substring (SUS)

function INITIALIZE($X[1,n]$, σ, $primes[1,c]$)
  for ($j \leftarrow 1$ to c) **do**
    $h[j] \leftarrow \sigma^{\delta-1} \mod primes[j]$
    $\Phi[j] \leftarrow X[1] \cdot \sigma^{\delta-1} + X[2] \cdot \sigma^{\delta-2} + \cdots + X[\delta]$
  return $\langle h, \Phi \rangle$

function SLIDE($X[1,n]$, σ, $primes[1,c]$, i, Φ, h, δ)
  for ($j \leftarrow 1$ to c) **do**
    $\Phi[j] \leftarrow \big(\sigma \cdot (\Phi[j] - X[i] \cdot h[j]) + X[i+\delta]\big) \mod primes[j]$

function SUS($X[1,n]$, σ, $primes[1,c]$)
  $\delta \leftarrow 1$
  $t =$ maximum value in $primes$
  Initialize two bloom filters BF_1 and BF_2, each of size t
  while ($\delta \leq n$) **do**
    $\langle h, \Phi \rangle =$INITIALIZE($X, \sigma, primes$) ▷ Begin Setting Phase
    for ($i \leftarrow 1$ to $n - \delta + 1$) **do**
      $allSet \leftarrow 1$
      for $j \leftarrow 1$ to c **do**
        if ($BF_1[\Phi[j]] = 0$) **then**
          $allSet \leftarrow 0$
          $BF_1[\Phi[j]] \leftarrow 1$
      if ($allSet = 1$) **then**
        for $j \leftarrow 1$ to c **do** $BF_2[\Phi[j]] \leftarrow 1$
      if ($i \leq n - \delta$) **then**
        SLIDE($X, \sigma, primes, i, \Phi, h, \delta$)
    $\langle h, \Phi \rangle \leftarrow$INITIALIZE($X, \sigma, primes$) ▷ Begin Reading Phase
    for ($i \leftarrow 1$ to $n - \delta + 1$) **do**
      for $j \leftarrow 1$ to c **do**
        if ($BF_2[\Phi[j]] = 0$) **then**
          return $\langle i, \delta \rangle$
      if ($i \leq n - \delta$) **then**
        SLIDE($X, \sigma, primes, i, \Phi, h, \delta$)
    Reset all bits of BF_1 and BF_2
    $\delta \leftarrow \delta + 1$
  return n

Window Size	Hash Values	Analysis
1	• $h_1(a) = 1$ • $h_2(a) = 6$ • $h_3(a) = 7$ • $h_1(b) = 2$ • $h_2(b) = 5$ • $h_3(b) = 10$ • $h_1(c) = 9$ • $h_2(c) = 8$ • $h_3(c) = 3$	• The character **b** at position 1 generates the same hash values as **b** at position 3. Therefore, the second bloom filter is populated. • The character **a** at positions 2 and 4 generate the same hash values. Therefore, second bloom filter is populated. • The character **c** at positions $5, 6$, and 7 all generate the same hash value. Therefore, second bloom filter is populated. • In the final process when checking with the second bloom filter, all of them are hashed meaning that there is NO unique substring of length 1.
2	• $h_1(ba) = 1$ • $h_2(ba) = 7$ • $h_3(ba) = 2$ • $h_1(ab) = 8$ • $h_2(ab) = 3$ • $h_3(ab) = 9$ • $h_1(ac) = 2$ • $h_2(ac) = 10$ • $h_3(ac) = 5$ • $h_1(cc) = 2$ • $h_2(cc) = 6$ • $h_3(cc) = 12$	• **ba** at position 1 generates the same hash values as **ba** at position 3. Therefore, the second bloom filter is populated. • **ab** at position 2 has no hash value that collide; hence, it is not hashed into the second bloom filter. • **ac** has its first hash value collide with that of $h_3(ba)$; however, the rest of the hash values do not collide. Hence, nothing is hashed to the second bloom filter. • **cc** at positions 5 and 6 generate the same hash values. Therefore, the second bloom filter is populated. • In the final process when checking with the second bloom filter, **ab** does not have any of its hash values hashed, whereas **ac** has its first hash value hashed. Since in either case the entire hash family is not hashed in the second bloom filter, this means that both **ab** and **ac** are reported as unique.

Table 3.1: Explanation of the process for finding the SUS in the string $babaccc$

Window Size	Hash Values	Analysis
1	• $h_1(a) = 1$ • $h_2(a) = 6$ • $h_3(a) = 7$ • $h_1(b) = 2$ • $h_2(b) = 5$ • $h_3(b) = 10$ • $h_1(c) = 9$ • $h_2(c) = 8$ • $h_3(c) = 3$ • $h_1(d) = 6$ • $h_2(d) = 2$ • $h_3(d) = 3$	• The **a** at position 0, 3, and 7 generates the same hash values. Therefore collision, second bloom filter is populated • The **a** at position 1, 4, and 9 generate the same hash values. Therefore, second bloom filter is populated • The **c** at positions 2,6,8 all generate the same hash value. Therefore, second bloom filter is populated. • **d** that occurs only once at position 5 has a collision with each of the other characters. Therefore it gets hashed to the second bloom filter. • In the final process when checking with the second bloom filter, for all the characters, their hash family values are hashed. This will mean that there exists no SUS with size 1 and it will have to move to the next window size. Even though d was an SUS, it was **falsely** marked as not a SUS.

Table 3.2: Explanation of a false negative in the string $abcabdcacb$

Chapter 4

Computing Maximal Unique Matches

Throughout this section, we consider the scenario where we are trying to find the shortest maximal unique match of two strings X_1 and X_2 of respective lengths n and m. Without loss of generality, we assume $n \leq m$.

4.1 Using Generalized Suffix Trees

To find all the maximal unique matches between two strings X_1 and X_2, we use the generalized suffix trees (GST) of X_1 and X_2. (Recall the GST is a compacted trie of all the suffixes of X_1 and X_2.) We assume that X_1 and X_2 end in two unique characters \$ and # that do not appear anywhere else in the string. Also, the **suffix link** of a node u points to a node v iff $\mathsf{str}(v)$ is the same as $\mathsf{str}(u)$ with the first character removed. Here, is a sketch of the algorithm [GHST17]:

- Given two texts X_1 and X_2, construct their generalized suffix tree.
- Pick a node u. Then, $\mathsf{str}(u)$ is an MUM of X_1 and X_2 if and only if the following holds:
  - **Leaves of the node**: The node contains exactly two leaves in its subtree, one from X_1 and one from X_2
  - **Suffix links**: There is no suffix link coming into the node from another node, which has exactly two leaves in its subtree, one from X_1 and one from X_2.

For example, consider the two strings $X_1 : abcaxy\$$ and $X_2 : abcayx\#$. Its MUMs are x, y, and $abca$. Correspondingly, we see that in Figure 4.1, each of the three nodes u, v, and w

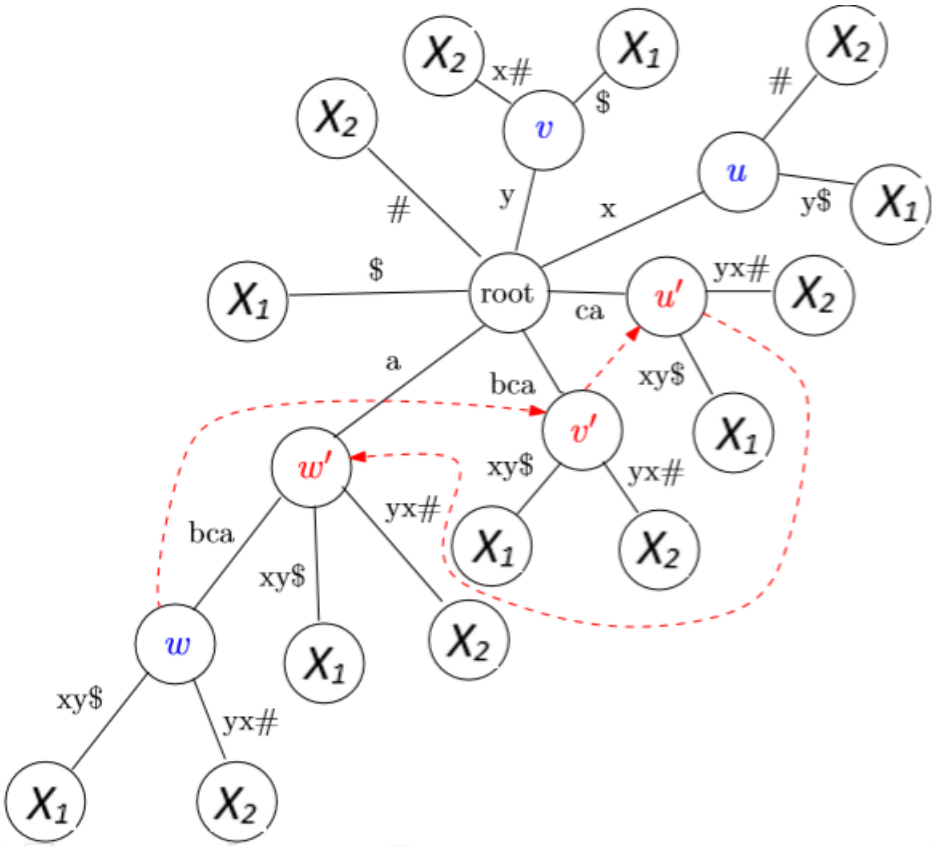


Figure 4.1: Generalized Suffix Tree of the strings X_1 : $abcaxy\$$ and X_2 : $abcayx\#$. Red dashed arrows are the suffix links, whereas blue nodes correspond to MUMs.

satisfy the criteria above. However, neither a, nor bca, nor ca is an MUM, because w', v', u' violate at least one of the three crietria.

For a given set of strings $\{X_1, X_2, X_3, \ldots, X_k\}$, $n = \sum_{i=1}^{k} X_i$, where X_i is the length of X_i, the time required to construct the generalized suffix tree and finding all the maximal unique matches is $O(n)$. The space needed is $\Theta(n)$ words, or equivalently $\Theta(n \log n)$ bits.

4.2 Our Approach

The shortest unique substring problem can be extended and modified to solve the maximal unique matches problem. The procedure is as follows:

Compute MUM

Return the smallest $\delta \in \{1, 2, \ldots, n\}$ for which there is an MUM of length δ

Is there a maximal unique match of length δ?

- Run the bloom filter setting phase
- If the bloom filter reading phase returns 'YES', the return 'YES, else return 'NO'

Setting Phase

- For a window of size δ, compute a set of Rabin-Karp (RK) hash values $h_1, h_2, \ldots, h_c$ for the string X_1. Initially, the first δ characters of the string form the first window.
- For each of the computed hash values $\{h_1, h_2, \ldots, h_c\}$, check the corresponding bloom filter BF_1 index. If all the indexes are already set to 1, then for these indexes set BF_2 to 1, else set the corresponding index in BF_1 to 1.
- Consider string X_2. Do the same as above but for another pair of bloom filters BF_3 and BF_4.
- Slide the window δ by one character, i.e., remove the first character in the window and include the next character of the string within the window. If there is no next character, then end the process.

Reading Phase

- For a window of size δ, compute a set of Rabin-Karp (RK) hash values $h_1, h_2, \ldots, h_c$ for the string X_1. Initially, the first δ characters of the string form the first window.
- For each of the computed hash values $\{h_1, h_2, \ldots, h_c\}$, check the corresponding index in bloom filter BF_2. We have the following scenarios.
  - If BF_2 is set for all the hash values, then the string has appeared more than once in X_1 and is not an MUM.
  - If at least one index in BF_2 is not set, then the corresponding δ-length string has appeared only once in X_1. Check the index in BF_3 and BF_4.
  - If any one index in BF_3 is not set, then the string has not appeared in X_2 and is not an MUM.
  - If every BF_3 index is set and every BF_4 index is also set, then the string has appeared more than once in X_2 and is not an MUM.
  - Otherwise, then the string has appeared once in X_2 and is a candidate for MUM. To verify whether it is an MUM or not, obtain the set of hash values for the left and right extensions of the string. Use a similar process to check if the left or right extension appears in X_2. If X_2 does not contain the left or right extension, then it is an MUM and return 'YES', else not.
- Slide the window δ by one character, i.e., remove the first character in the window and include the next character of the string within the window. If there is no next character, then return 'NO'.

Refer to Algorithm 2 and 3 for a detailed pseudo-code.

Algorithm 2 Smallest Maximal Unique Match (MUM) Helper Functions

function INITIALIZE($X_1[1,n]$, $X_2[1,m]$, σ, $primes[1,c]$)
  for ($j \leftarrow 1$ to c) **do**
    $h[j] \leftarrow \sigma^{\delta-1} \mod primes[j]$
    $\Phi_1[j] \leftarrow X_1[1] \cdot \sigma^{\delta-1} + X_1[2] \cdot \sigma^{\delta-2} + \cdots + X_1[\delta]$
    $\Phi_2[j] \leftarrow X_2[1] \cdot \sigma^{\delta-1} + X_2[2] \cdot \sigma^{\delta-2} + \cdots + X_2[\delta]$
  return $\langle h, \Phi_1, \Phi_2 \rangle$

function SLIDE($X[1,n]$, σ, $primes[1,c]$, i, Φ, h, δ)
  for ($j \leftarrow 1$ to c) **do**
    $\Phi[j] \leftarrow \big(\sigma \cdot (\Phi[j] - X[i] \cdot h[j]) + X[i+\delta]\big) \mod primes[j]$

function LEFTEXT($X[1,n]$, σ, $primes[1,c]$, i, Φ, h, δ)
  for ($j \leftarrow 1$ to c) **do**
    $\Phi_\ell[j] \leftarrow \big(\Phi[j] + \sigma \cdot X[i-1] \cdot h[j]\big) \mod primes[j]$
  return Φ_ℓ

function RIGHTEXT($X[1,n]$, σ, $primes[1,c]$, i, Φ, h, δ)
  for ($j \leftarrow 1$ to c) **do**
    $\Phi_r[j] \leftarrow \big(\Phi[j] \cdot \sigma + X[i+\delta]\big) \mod primes[j]$
  return Φ_r

function SET($X[1,x]$, δ σ, $primes[1,c]$, BF, BF_{aux}, Φ, h)
  for ($i \leftarrow 1$ to $n - \delta + 1$) **do**
    $allSet \leftarrow 1$
    for $j = 1$ to c **do**
      if ($BF[\Phi[j]] = 0$) **then**
        $allSet \leftarrow 0$
        $BF[\Phi[j]] \leftarrow 1$
    if ($allSet = 1$) **then**
      for $j \leftarrow 1$ to c **do** $BF_{aux}[\Phi[j]] \leftarrow 1$
    if ($i \leq n - \delta$) **then**
      SLIDE($X, \sigma, primes, i, \Phi, h, \delta$)

Algorithm 3 Smallest Maximal Unique Match (MUM)

function MUM($X_1[1,n]$, $X_2[1,m]$ σ, $primes[1,c]$)
  $\delta \leftarrow 1$
  $t \leftarrow$ maximum value in $primes$
  Initialize four bloom filters BF_1 through BF_4, each of size t
  while ($\delta \leq n$) **do**
    $\langle h, \Phi_1, \Phi_2 \rangle \leftarrow$INITIALIZE($X, \sigma, primes$)
    SET($X_1, \delta, \sigma, primes, BF_1, BF_2, \Phi_1, h$) ▷ Begin Setting Phase
    SET($X_2, \delta, \sigma, primes, BF_3, BF_4, \Phi_2, h$)
    $\langle h, \Phi_1, \Phi_2 \rangle =$INITIALIZE($X, \sigma, primes$) ▷ Begin Reading Phase
    for ($i \leftarrow 1$ to $n - \delta + 1$) **do**
      $flagBF_2 \leftarrow 1, flagBF_3 \leftarrow 1, flagBF_4 \leftarrow 1$
      for ($j = 1$ to c) **do**
        if ($BF_2[\Phi_1[j]] = 0$) **then**
          $flagBF_2 \leftarrow 0$
        if ($BF_3[\Phi_1[j]] = 0$) **then**
          $flagBF_3 \leftarrow 0$
        if ($BF_4[\Phi_1[j]] = 0$) **then**
          $flagBF_4 \leftarrow 0$
      if ($flagBF_2 = 0$ and $flagBF_3 = 1$ and $flagBF_4 = 0$) **then**
        if ($i > 1$) **then**
          $\Phi_\ell \leftarrow$ LEFTEXT($X_1, \sigma, primes, i, \Phi_1, h, \delta$)
        if ($i < n$) **then**
          $\Phi_r \leftarrow$ RIGHTEXT($X_1, \sigma, primes, i, \Phi_1, h, \delta$)
        for ($j = 1$ to c) **do**
          if (($i = 1$ or $BF_3[\Phi_\ell[j]] = 0$) and ($i = n$ or $BF_3[\Phi_r[j]] = 0$) **then**
            return $\langle i, \delta \rangle$
      if ($i \leq n - \delta$) **then**
        SLIDE($X_1, \sigma, primes, i, \Phi_1, h, \delta$)
    Reset all bits of BF_1, BF_2, BF_3, and BF_4
    $\delta \leftarrow \delta + 1$
  return n

4.3 Illustration

The procedure for finding an MUM are illustrated in Tables 4.1, 4.5, and 4.6. This technique however can result in false negatives (i.e., it may skip over a unique string without recognizing it as one); this case is the same as SUS and we skip it for the sake of simplicity.

Window Size	Hash Values	Analysis
1	• For X_1 : – $h_1(a) = 6$ – $h_2(a) = 2$ – $h_1(b) = 1$ – $h_2(b) = 4$ – $h_1(c) = 5$ – $h_2(c) = 7$ – $h_1(d) = 3$ – $h_2(d) = 9$ – $h_1(e) = 8$ – $h_2(e) = 10$ • For X_2 : – $h_1(a) = 6$ – $h_2(a) = 2$ – $h_1(b) = 1$ – $h_2(b) = 4$ – $h_1(c) = 5$ – $h_2(c) = 7$ – $h_1(d) = 3$ – $h_2(d) = 9$ – $h_1(e) = 8$ – $h_2(e) = 10$	• The character **a** in string X_1 at positions $2, 6$, and 10 and in string X_2 at position 2 and 9 both hash to the same place. However, before proceeding to string X_2, it can be seen that it **a** is not unique in string X_1. • The character **b** in string X_1 at position 3 and 8 and in string X_2 at position 3 and 7 both hash to the same place. However, before proceeding to string X_2, it can be seen that it **b** is not unique in string X_1. • The character **c** in string X_1 at positions 4 and 7 and in string X_2 at positions 1, 4, 5, and 10 hash to the same place. However, before proceeding to string X_2, it can be seen that it **c** is not unique in string X_1. • The character **d** in string X_1 at position 5 and in string X_2 at position 8 both hash to the same place in their respective bloom filters. When checking second bloom filter, it will show that in string X_1, **d** is unique and when checking the fourth bloom filter it will show that **d** is unique in X_2. Now that a unique substring common to both strings are found, the right and left extensions need to be checked. When the left extension is checked for X_1 (**cd**) with that of X_2 (**bd**) it revealed that the left extension property passes. However when checking the right extension of X_1 (**da**) and right extension of X_2 (**da**) it shows that they are the same. Therefore, right extension property fails making **d** not a MUM. • The character **e** in string X_1 at positions 1 and 9 and in string X_2 at position 6 hash to the same place. Even though **e** is unique in string X_2 and both left and right extension properties hold, **e** is not an MUM since it is not unique in string X_1. • Since there were no MUMs with length 1, the process will repeat with window size 2.

Table 4.1: Explanation of MUM showing the cases where MUM is found and MUM fails. MUM calculation shown for $X_1 = eabcdacbea$ and $X_2 = cabccebdac$ with window size 1.

Window Size	Hash Values	Analysis
2	• For X_1 : – $h_1(ab) = 6$ – $h_2(ab) = 2$ – $h_1(bc) = 1$ – $h_2(bc) = 4$ – $h_1(be) = 11$ – $h_2(be) = 13$ – $h_1(cb) = 5$ – $h_2(cb) = 7$ – $h_1(cd) = 14$ – $h_2(cd) = 12$ – $h_1(da) = 3$ – $h_2(da) = 9$ – $h_1(ea) = 8$ – $h_2(ea) = 10$	• The substring **ab** in string X_1 at position 2 and in string X_2 at position 2 hash to the same place. Checking the second bloom filter will reveal that **ab** is unique in string X_1 and checking the fourth bloom filter will reveal it is unique for string X_2. Since this is passed as unique, the left and right extensions will be checked. Checking the left extension in X_1 (**eab**) and that of string X_2 (**cab**) show that it passes the left extension property. However, checking the right extension of string X_1 (**abc**) with that of string X_2 (**abc**) shows the property fails. Therefore **ab** is not a MUM. The scenario is the same for **da**. • The substring **bc** in string X_1 at position 3 and in string X_2 at position 3 both hash to the same place. Checking the second bloom filter will reveal that **bc** is unique in string X_1 and checking the fourth bloom filter will reveal it is unique for string X_2. Since this is passed as unique the left and right extensions will be checked. Checking the right extension in X_1 (**bcd**) and that of string X_2 (**bcc**) show that it passes the right extension property. However, checking the left extension of string X_1 (**abc**) with that of string X_2 (**abc**) shows the property fails. Therefore **bc** is not a MUM.

Table 4.3: Explanation of MUM showing the cases where MUM is found and MUM fails. MUM calculation shown for $X_1 = eabcdacbea$ and $X_2 = cabccebdac$ with window size 2.

Window Size	Hash Values	Analysis
2	• For X_2 : – $h_1(ab) = 6$ – $h_2(ab) = 2$ – $h_1(ac) = 6$ – $h_2(ac) = 2$ – $h_1(bc) = 1$ – $h_2(bc) = 4$ – $h_1(bd) = 11$ – $h_2(bd) = 13$ – $h_1(ca) = 5$ – $h_2(ca) = 7$ – $h_1(cc) = 14$ – $h_2(cc) = 12$ – $h_1(da) = 3$ – $h_2(da) = 9$ – $h_1(eb) = 8$ – $h_2(eb) = 10$	• The substring **be** in string X_1 at position 8 is found to be unique when checking the second bloom filter. However, when checking string X_2 it shows that **be** is not present. This means that **be** is not an MUM. The scenario is the same for **cb** and **cd**. • The substring **ea** which appears at position 1 and 9 both hash into the same place. Therefore, when checking the second bloom filter it will reveal that it is not unique making it not a MUM.

Table 4.5: Continued explanation of MUM showing the cases where MUM is found and MUM fails. MUM calculation shown for $X_1 = eabcdacbea$ and $X_2 = cabccebdac$ with window size 2.

Window Size	Hash Values	Analysis
1	• $h_1(a) = 1$ • $h_2(a) = 6$	• Since the entire window contains only **a**, all the positions will hash to the same position. • In the final process when checking with the second bloom filter, for all the characters, their hash family values are hashed. This will mean that there exists no unique substring with size 1 and it will have to move to the next window size.
2	• $h_1(aa) = 4$ • $h_2(aa) = 7$	• Since the entire window contains the pair **aa**, all the positions will hash to the same position. • In the final process when checking with the second bloom filter, for all the characters, their hash family values are hashed. This will mean that there exists no unique substring with size 2, and it will have to move to the next window size. • This pattern will continue throughout the string. Therefore there is nothing unique in the first string, it would mean there is no MUM.

Table 4.6: Explanation of MUM failure when the string contains nothing unique, as shown for string $aaaa$

Chapter 5

Experiments for Shortest Unique Substring

The Genome data required for testing contained real data gathered from the `National Center for Biotechnology Information`[1] along with synthetic test data constructed through randomization. The following experiments were conducted on a daily use laptop with Intel(R) Core(TM) i7-8550 CPU @ 1.80GHz processor and 16GB RAM. The code was written in Java.

5.1 Setup

For each of the experiments, we chose 4 random prime numbers in the range $[n/8, n/4]$, where n is the length of the corresponding string. The prime numbers are chosen by the Sieve of Eratosthenes method in $O(n \log n)$ deterministic time and using n bits of space. On average the length of an SUS is expected to be $O(\log_\sigma n)$ when the text is assumed to be independently and identically distributed over the input alphabet [DSR92]; this corroborated by the experimental results. Hence, our proposed algorithm runs in $O(n \log n)$ expected time. Moreover, the use of Bloom Filters and the choice of primes ensure that we do not use more than n bits of space in addition to the input string.

5.2 Synthetic Genome Data

Our first set of experiment are on synthetic Genome data of varying lengths. Comparing the results from Table 5.1 and Figure 5.1, it showed that the time required for the computation of SUS via Bloom Filters and RK Fingerprint was around 2.5 times faster than the Manber

[1] https://www.ncbi.nlm.nih.gov/genome

Myers based algorithms. Even though our algorithm performed faster it showed that it had a percentage error of 14.2%.

5.3 Real Genome Data

When this test was performed on real Genome data it showed that our algorithm outperformed the Manber-Myers based algorithm. Additionally, no errors were found. Details of the experiments are in Tables 5.2 and Figure 5.2. For `C.elegans` and `T.nigroviridis`, we were unable to run the Manber-Myers based method due to Java heap size limitations.

5.4 Synthetic Data of Varying Alphabet Size

The next few tests were done to see the performance of the algorithms depending on the size of the alphabet. As seen in Table 5.3 and Figure 5.3, when the size of the alphabet was 2, the performance between the two algorithms differed by an estimated factor of 1.5 with 0% error. However, as we move from alphabet size 4 to 1024 (i.e., as the size of the alphabet grows), the difference in performance grows with minor percentages of error. The percentage error for these experiments were calculated by comparing the difference in length of the SUS found by both techniques.

5.5 Peak Memory Usage

Comparing the peak memory usage for all the mentioned experiments, all of them showed the same exact pattern. The Manber Myers method suffered from a huge blow up in the peak memory usage compared to the input data (and our algorithm). On the other hand, comparing our algorithm with the actual space required, it showed little to no difference making our algorithm much more space-efficient.

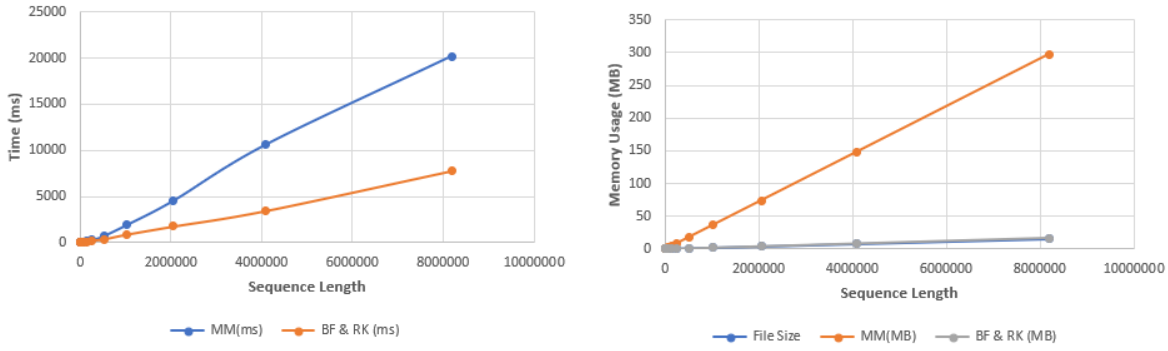


Figure 5.1: Figures associated with Table 5.1

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	4	4	5	5	0.03626	0.00216
2000	0.00381	4	5	4	3	0.07250	0.00423
4000	0.00763	5	5	8	6	0.14498	0.00836
8000	0.01526	5	6	10	16	0.28994	0.01662
16000	0.03052	6	6	18	17	0.57986	0.03295
32000	0.06104	6	6	36	28	1.15969	0.06590
64000	0.12207	7	7	55	44	2.31936	0.13193
128000	0.24414	7	7	162	94	4.63869	0.26131
256000	0.48828	8	8	270	235	9.27737	0.52112
512000	0.97656	8	8	735	368	18.55471	1.04835
1024000	1.95313	9	9	1942	881	37.10940	2.09732
2048000	3.90625	9	9	4549	1750	74.21877	4.17577
4096000	7.81250	9	9	10623	3441	148.43752	8.35973
8192000	15.62500	10	10	20147	7758	296.87502	16.72838

Table 5.1: Table displaying the Length of SUS found, the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly and contains 4 characters. Percentage of error **= 14.2%**

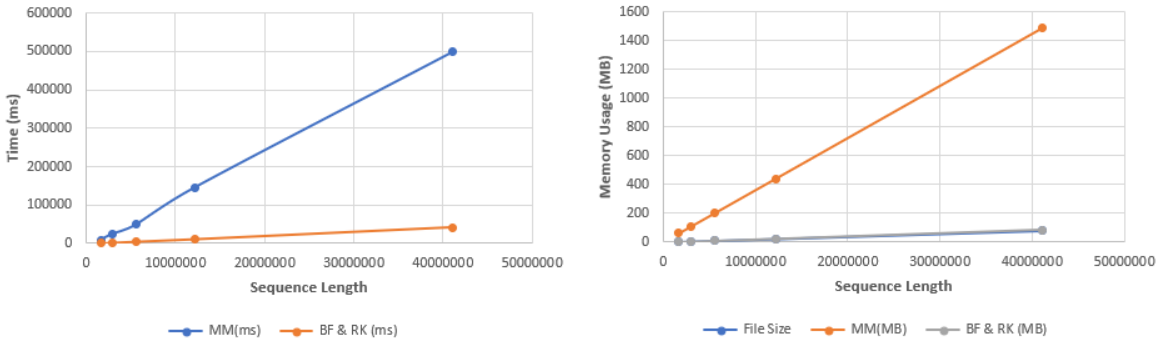


Figure 5.2: Figures associated with Table 5.2

Genome	Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
			MM	BF & RK	MM	BF & RK	MM	BF & RK
H.pylori	1667825	3.18112	6	6	9408	945	60.44137	3.40915
L.monocytogenes	2944528	5.61624	8	8	24503	2294	106.70861	5.98914
E.Coli	5594605	10.67086	8	8	48721	5135	202.74640	11.34465
S.cerevisiae	12157105	23.18784	9	9	145289	11397	440.56894	24.75500
N.crassa	41061603	78.31879	10	10	498736	41396	1488.05708	83.73215
C.elegans	100286401	191.28113	-	10	-	98277	-	202.07431
T.nigroviridis	312394419	595.84507	-	11	-	352467	-	634.12550

Table 5.2: Table displaying the Length of SUS found the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The Genomes are real. Percentage of error **= 0%**

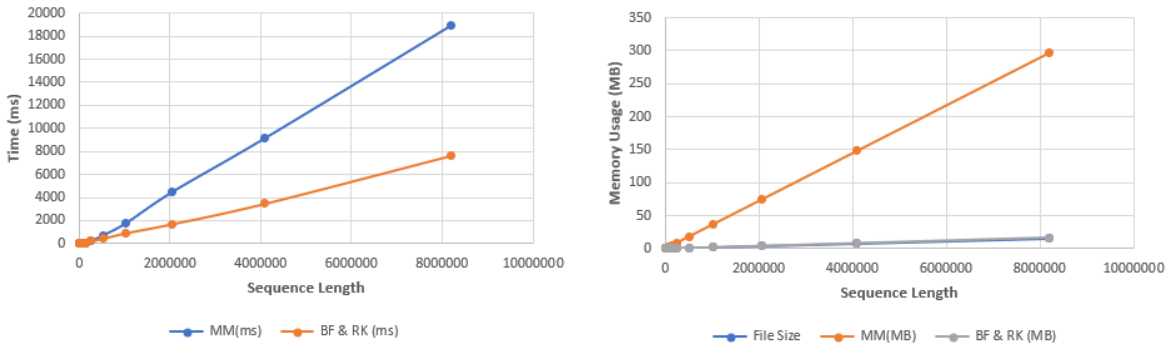


Figure 5.3: Figures associated with Table 5.3

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	7	7	6	2	0.03625	0.00214
2000	0.00381	8	8	5	3	0.07249	0.00424
4000	0.00763	9	9	7	6	0.14497	0.00833
8000	0.01526	10	10	11	17	0.28993	0.01662
16000	0.03052	11	11	16	20	0.57985	0.03283
32000	0.06104	12	12	718	38	1.15968	0.06599
64000	0.12207	13	13	47	83	2.31935	0.13148
128000	0.24414	14	14	109	199	4.63868	0.26280
256000	0.48828	15	15	365	426	9.27736	0.52396
512000	0.97656	16	16	1111	934	18.55470	1.04140
1024000	1.95313	17	17	2208	1783	37.10939	2.09617
2048000	3.90625	18	18	5344	3763	74.21876	4.18944
4096000	7.81250	19	19	8977	7816	148.43751	8.34267
8192000	15.62500	20	20	21906	16440	296.87501	16.70374

Table 5.3: Table displaying the Length of SUS found, the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly for an alphabet of size 2. Percentage of error **= 0%**

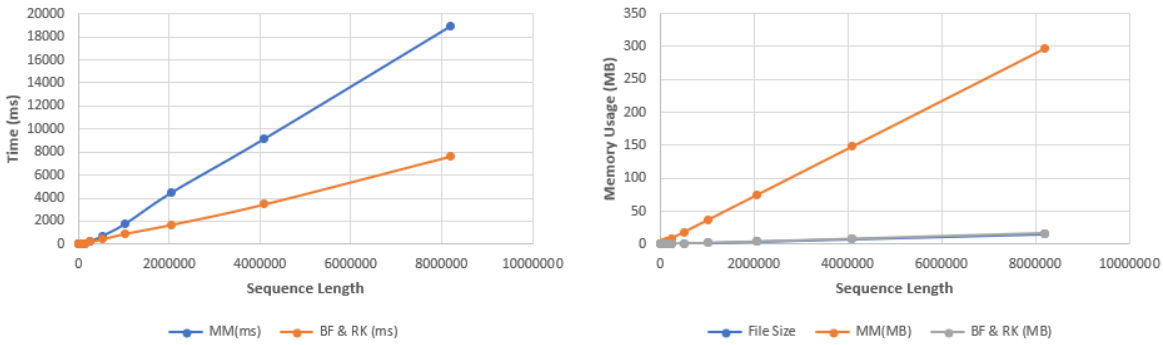


Figure 5.4: Figures associated with Table 5.4

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	4	4	0	0	0.03626	0.00216
2000	0.00381	4	4	1	1	0.07250	0.00424
4000	0.00763	5	5	1	1	0.14498	0.00831
8000	0.01526	5	5	3	3	0.28994	0.01656
16000	0.03052	6	6	7	9	0.57986	0.03282
32000	0.06104	6	6	14	21	1.15969	0.06609
64000	0.12207	7	7	33	50	2.31936	0.13168
128000	0.24414	7	7	97	92	4.63869	0.26318
256000	0.48828	8	8	278	217	9.27737	0.52486
512000	0.97656	8	8	734	432	18.55471	1.04748
1024000	1.95313	9	9	1802	913	37.10940	2.09533
2048000	3.90625	9	9	4509	1695	74.21877	4.18998
4096000	7.81250	9	9	9168	3466	148.43752	8.34682
8192000	15.62500	10	10	18880	7604	296.87502	16.69284

Table 5.4: Table displaying the Length of SUS found, the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly for an alphabet of size 4. Percentage of error **= 0%**

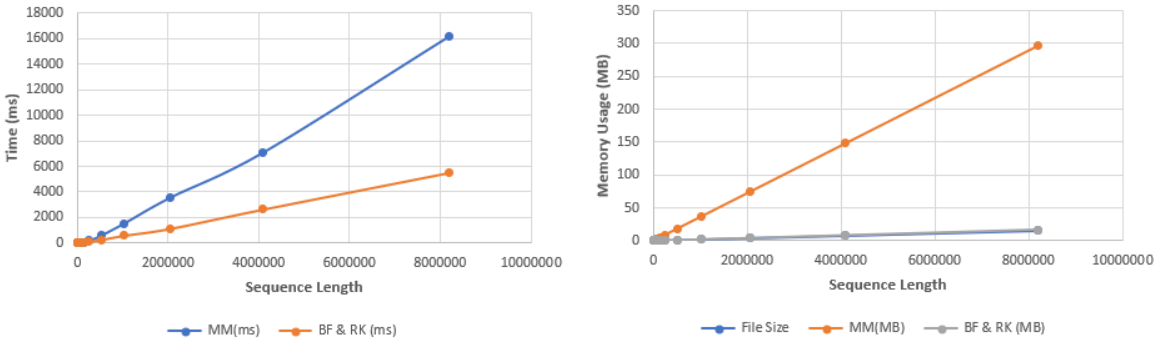


Figure 5.5: Figures associated with Table 5.5

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	3	3	0	0	0.03629	0.00215
2000	0.00381	3	3	0	1	0.07253	0.00424
4000	0.00763	4	4	1	2	0.14500	0.00837
8000	0.01526	4	4	2	3	0.28996	0.01665
16000	0.03052	4	4	6	6	0.57988	0.03297
32000	0.06104	4	4	12	10	1.15971	0.06606
64000	0.12207	5	5	30	33	2.31938	0.13119
128000	0.24414	5	5	69	56	4.63872	0.26179
256000	0.48828	5	5	196	118	9.27739	0.52265
512000	0.97656	5	5	595	246	18.55473	1.04391
1024000	1.95313	6	6	1551	578	37.10942	2.08998
2048000	3.90625	6	6	3583	1122	74.21880	4.18721
4096000	7.81250	7	7	7102	2630	148.43755	8.36772
8192000	15.62500	7	7	16153	5493	296.87505	16.67570

Table 5.5: Table displaying the Length of SUS found, the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly for an alphabet of size 8. Percentage of error **= 0%**

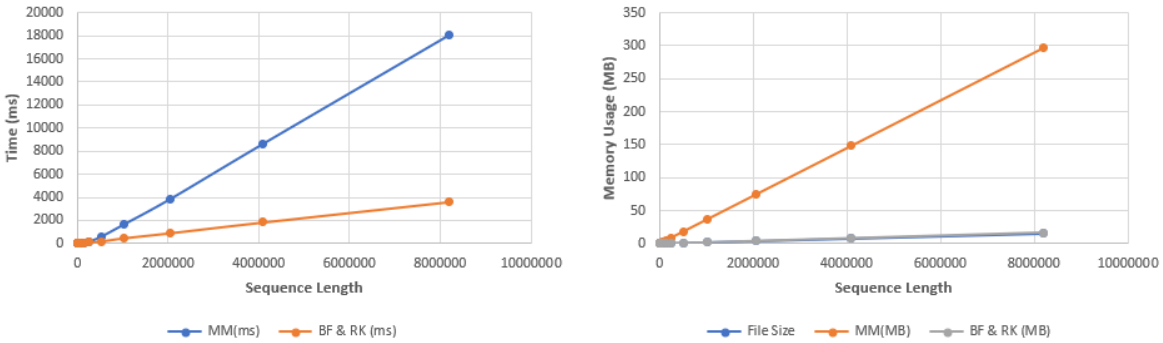


Figure 5.6: Figures associated with Table 5.6

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	2	2	0	0	0.03633	0.00216
2000	0.00381	2	2	1	1	0.07257	0.00421
4000	0.00763	3	3	1	1	0.14505	0.00833
8000	0.01526	3	3	3	2	0.29001	0.01650
16000	0.03052	3	3	4	3	0.57993	0.03310
32000	0.06104	3	3	12	9	1.15976	0.06590
64000	0.12207	3	3	28	19	2.31943	0.13115
128000	0.24414	4	4	68	47	4.63876	0.26056
256000	0.48828	4	4	195	99	9.27744	0.52428
512000	0.97656	4	4	613	190	18.55478	1.04696
1024000	1.95313	5	5	1693	469	37.10947	2.09132
2048000	3.90625	5	5	3894	913	74.21884	4.18527
4096000	7.81250	5	5	8667	1847	148.43759	8.34720
8192000	15.62500	5	5	18047	3620	296.87509	16.67537

Table 5.6: Table displaying the Length of SUS found the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly for an alphabet of size 16. Percentage of error **= 0%**

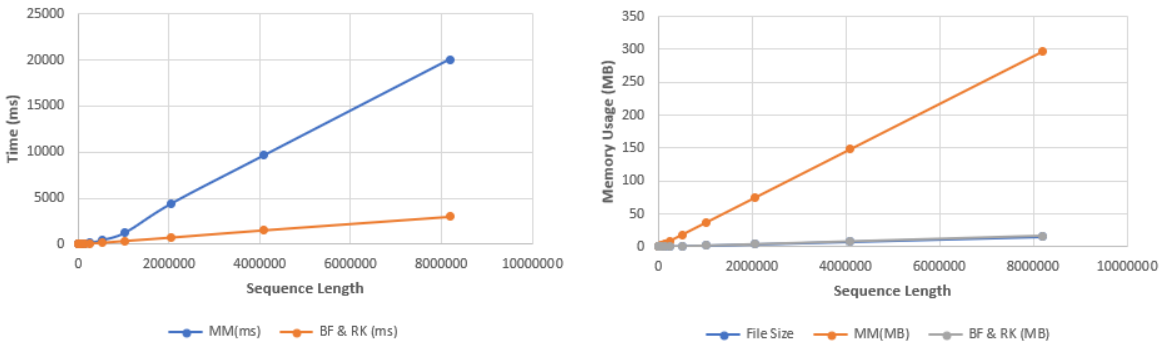


Figure 5.7: Figures associated with Table 5.7

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	2	2	0	0	0.03642	0.00215
2000	0.00381	2	2	1	0	0.07266	0.00424
4000	0.00763	2	2	2	0	0.14514	0.00833
8000	0.01526	2	2	2	1	0.29010	0.01663
16000	0.03052	3	3	5	4	0.58002	0.03298
32000	0.06104	3	3	10	10	1.15985	0.06585
64000	0.12207	3	3	27	17	2.31952	0.13183
128000	0.24414	3	3	47	34	4.63885	0.26315
256000	0.48828	3	3	161	72	9.27753	0.52189
512000	0.97656	4	4	462	174	18.55487	1.04227
1024000	1.95313	4	4	1279	369	37.10956	2.09272
2048000	3.90625	4	4	4443	740	74.21893	4.14534
4096000	7.81250	4	4	9673	1517	148.43768	8.37418
8192000	15.62500	4	4	20053	2973	296.87518	16.70149

Table 5.7: Table displaying the Length of SUS found, the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly for an alphabet of size 32. Percentage of error **= 0%**

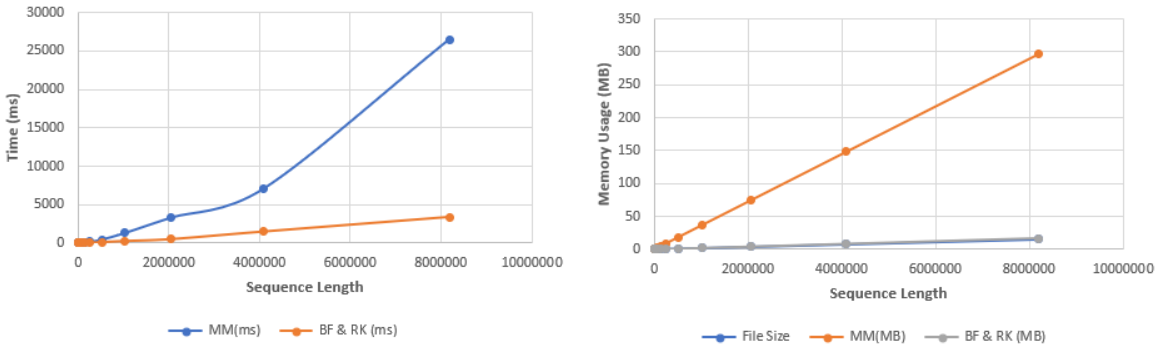


Figure 5.8: Figures associated with Table 5.8

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	2	2	1	0	0.03661	0.00215
2000	0.00381	2	2	1	0	0.07285	0.00424
4000	0.00763	2	2	0	0	0.14532	0.00831
8000	0.01526	2	2	2	1	0.29028	0.01660
16000	0.03052	2	2	4	3	0.58020	0.03278
32000	0.06104	2	2	11	5	1.16003	0.06574
64000	0.12207	3	3	19	17	2.31970	0.13178
128000	0.24414	3	3	56	34	4.63904	0.26269
256000	0.48828	3	3	163	68	9.27771	0.52386
512000	0.97656	3	3	471	141	18.55505	1.04871
1024000	1.95313	3	3	1333	284	37.10974	2.09658
2048000	3.90625	3	3	3334	527	74.21912	4.17215
4096000	7.81250	4	4	7123	1500	148.43787	8.32667
8192000	15.62500	4	4	26498	3365	296.87537	16.70483

Table 5.8: Table displaying the Length of SUS found, the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly for an alphabet of size 64. Percentage of error **= 0%**

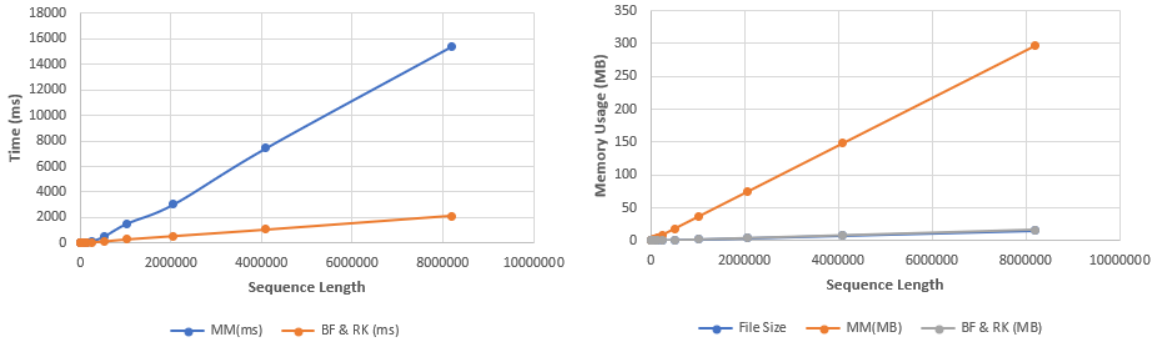


Figure 5.9: Figures associated with Table 5.9

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	2	2	1	0	0.03697	0.00215
2000	0.00381	2	2	0	1	0.07321	0.00422
4000	0.00763	2	2	1	1	0.14569	0.00836
8000	0.01526	2	2	3	1	0.29065	0.01659
16000	0.03052	2	2	5	3	0.58057	0.03305
32000	0.06104	2	2	15	6	1.16040	0.06598
64000	0.12207	2	2	29	11	2.32007	0.13158
128000	0.24414	2	2	50	22	4.63940	0.26172
256000	0.48828	3	3	166	73	9.27808	0.52451
512000	0.97656	3	3	521	143	18.55542	1.04820
1024000	1.95313	3	3	1541	302	37.11011	2.08956
2048000	3.90625	3	3	3026	558	74.21948	4.18748
4096000	7.81250	3	3	7423	1065	148.43823	8.29085
8192000	15.62500	3	3	15340	2139	296.87573	16.61886

Table 5.9: Table displaying the Length of SUS found, the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly for an alphabet of size 128. Percentage of error **= 0%**

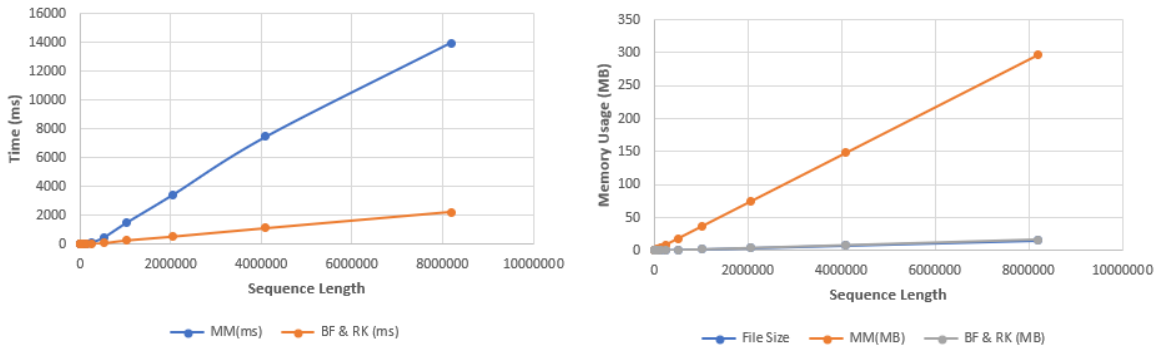


Figure 5.10: Figures associated with Table 5.10

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	1	1	2	0	0.03770	0.00214
2000	0.00381	1	1	2	1	0.07394	0.00422
4000	0.00763	2	2	3	1	0.14642	0.00835
8000	0.01526	2	2	2	1	0.29138	0.01664
16000	0.03052	2	2	3	4	0.58130	0.03285
32000	0.06104	2	2	12	5	1.16113	0.06585
64000	0.12207	2	2	28	10	2.32080	0.13169
128000	0.24414	2	2	51	20	4.64014	0.26166
256000	0.48828	2	2	158	45	9.27881	0.52523
512000	0.97656	2	2	491	85	18.55615	1.04861
1024000	1.95313	3	3	1527	279	37.11084	2.08475
2048000	3.90625	3	3	3459	546	74.22021	4.16340
4096000	7.81250	3	3	7463	1123	148.43896	8.31774
8192000	15.62500	3	3	13967	2243	296.87646	16.67623

Table 5.10: Table displaying the Length of SUS found, the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly for an alphabet of size 256. Percentage of error **= 0%**

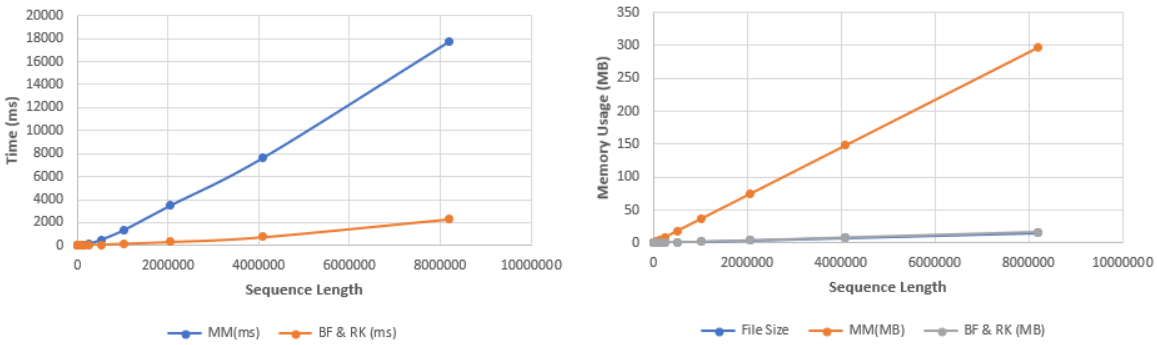


Figure 5.11: Figures associated with Table 5.11

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	1	4	0	0	0.03917	0.00215
2000	0.00381	1	1	0	0	0.07541	0.00420
4000	0.00763	1	1	1	0	0.14789	0.00835
8000	0.01526	2	2	2	1	0.29285	0.01665
16000	0.03052	2	2	4	3	0.58276	0.03291
32000	0.06104	2	2	10	6	1.16260	0.06581
64000	0.12207	2	2	17	12	2.32227	0.13171
128000	0.24414	2	2	34	22	4.64160	0.26276
256000	0.48828	2	2	185	46	9.28027	0.52377
512000	0.97656	2	2	530	90	18.55762	1.04430
1024000	1.95313	2	2	1393	180	37.11230	2.09334
2048000	3.90625	2	2	3513	327	74.22168	4.18058
4096000	7.81250	2	2	7662	753	148.44043	8.34642
8192000	15.62500	3	3	17728	2322	296.87793	16.62958

Table 5.11: Table displaying the Length of SUS found, the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly for an alphabet of size 512. Percentage of error **= 7.1%**

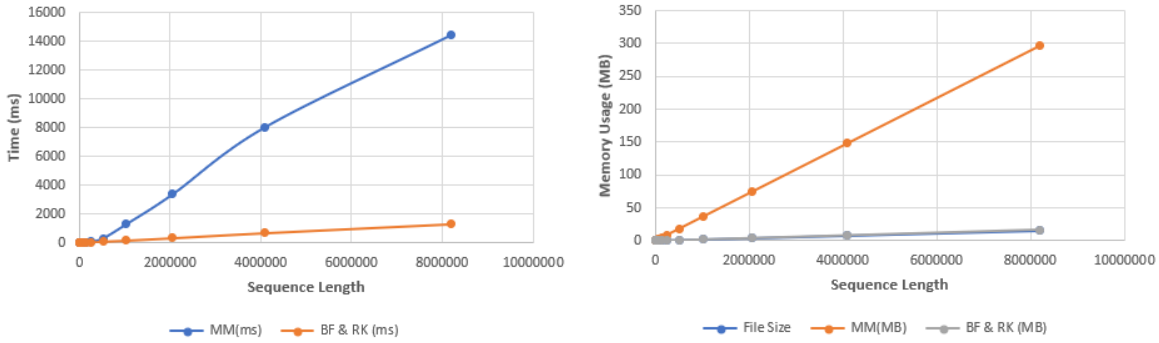


Figure 5.12: Figures associated with Table 5.12

Sequence Length	File Size (in MB)	SUS Length		Time (in millisec)		Peak Memory (in MB)	
		MM	BF & RK	MM	BF & RK	MM	BF & RK
1000	0.00191	1	1	0	0	0.04210	0.00215
2000	0.00381	1	1	1	0	0.07834	0.00420
4000	0.00763	1	1	1	1	0.15082	0.00838
8000	0.01526	1	1	2	0	0.29578	0.01664
16000	0.03052	2	2	4	3	0.58569	0.03311
32000	0.06104	2	2	7	6	1.16553	0.06540
64000	0.12207	2	2	16	10	2.32520	0.13094
128000	0.24414	2	2	39	24	4.64453	0.26215
256000	0.48828	2	2	103	41	9.28320	0.52495
512000	0.97656	2	2	338	98	18.56055	1.04609
1024000	1.95313	2	2	1320	173	37.11523	2.08223
2048000	3.90625	2	2	3395	344	74.22461	4.18626
4096000	7.81250	2	2	8064	700	148.44336	8.27443
8192000	15.62500	2	2	14432	1332	296.88086	16.65580

Table 5.12: Table displaying the Length of SUS found, the time required, and the memory usage to find the SUS using Manber Myers (MM) and Bloom Filter technique (BF). The string is generated randomly for an alphabet of size 1024. Percentage of error **= 0%**

LIST OF REFERENCES

[ABF+15] Boran Adas, Ersin Bayraktar, Simone Faro, Ibraheem Elsayed Moustafa, and M. Oguzhan Külekci. Nucleotide sequence alignment and compression via shortest unique substring. In *Bioinformatics and Biomedical Engineering - Third International Conference, IWBBIO 2015, Granada, Spain, April 15-17, 2015. Proceedings, Part II*, pages 363–374, 2015.

[APPA15] José M Abuín, Juan C Pichel, Tomás F Pena, and Jorge Amigo. Bigbwa: approaching the burrows–wheeler aligner to big data technologies. *Bioinformatics*, 31(24):4003–4005, 2015.

[APPA16] José M Abuín, Juan C Pichel, Tomás F Pena, and Jorge Amigo. Sparkbwa: speeding up the alignment of high-throughput dna sequencing data. *PloS one*, 11(5):e0155461, 2016.

[BC14] Djamal Belazzougui and Fabio Cunial. Indexed matching statistics and shortest unique substrings. In *String Processing and Information Retrieval - 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, pages 179–190, 2014.

[Bel14] Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 148–193, 2014.

[BGVV14] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory of Computing Systems*, 55(1):41–60, 2014.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[BM] https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#76d71d3a60ba.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[BO98] Ulfar Bergthorsson and H Ochman. Distribution of chromosome length variation in natural isolates of escherichia coli. *Molecular biology and evolution*, 15:6–16, 1998.

[BOSS12] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 225–235, 2012.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[CEPR10] Raphaël Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. Pattern matching with don't cares and few errors. *Journal of Computer and System Sciences*, 76(2):115–124, 2010.

[CGL04] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100. ACM, 2004.

[DEMS+12] Thomas Derrien, Jordi EstellÃ, Santiago Marco Sola, David G. Knowles, Emanuele Raineri, Roderic GuigÃ, and Paolo Ribeca. Fast computation and applications of genome mappability. *PLoS ONE*, 7:1–16, 01 2012.

[DKF+99] Arthur L Delcher, Simon Kasif, Robert D Fleischmann, Jeremy Peterson, Owen White, and Steven L Salzberg. Alignment of whole genomes. *Nucleic acids research*, 27(11):2369–2376, 1999.

[DPCS02] Arthur L Delcher, Adam Phillippy, Jane Carlton, and Steven L Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic acids research*, 30(11):2478–2483, 2002.

[DSR92] Luc Devroye, Wojciech Szpankowski, and Bonita Rais. A note on the height of suffix trees. *SIAM J. Comput.*, 21(1):48–53, 1992.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with O(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[FLMM09] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398, 2000.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[GHST17] Arnab Ganguly, Wing-Kai Hon, Rahul Shah, and Sharma V. Thankachan. Space-time trade-offs for finding shortest unique substrings and maximal unique matches. *Theor. Comput. Sci.*, 700:75–88, 2017.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[GV00] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 397–406, 2000.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[HPMW05] Bernhard Haubold, Nora Pierstorff, Friedrich Möller, and Thomas Wiehe. Genome comparison without alignment using shortest unique substrings. *BMC Bioinformatics*, 6(1):1–11, 2005.

[HPT14] Xiaocheng Hu, Jian Pei, and Yufei Tao. Shortest unique queries on strings. In *String Processing and Information Retrieval - 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, pages 161–172, 2014.

[HS02] Wing-Kai Hon and Kunihiko Sadakane. Space-economical algorithms for finding maximal unique matches. In *Combinatorial Pattern Matching, 13th Annual Symposium, CPM 2002, Fukuoka, Japan, July 3-5, 2002, Proceedings*, pages 144–152, 2002.

[HSS03] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 251–260, 2003.

[HTSV13] Wing-Kai Hon, Sharma V Thankachan, Rahul Shah, and Jeffrey Scott Vitter. Faster compressed top-k document retrieval. In *2013 Data Compression Conference*, pages 341–350. IEEE, 2013.

[HTX15] Wing-Kai Hon, Sharma V. Thankachan, and Bojian Xu. An in-place framework for exact and approximate shortest unique substring queries. In *Algorithms and Computation - 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, pages 755–767, 2015.

[IKX14] Atalay Mert Ileri, M. Oguzhan Külekci, and Bojian Xu. Shortest unique substring query revisited. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings*, pages 172–181, 2014.

[KJP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[KLL+12] Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles SH Yeo, and Brian YH Lam. Barracuda-a fast short read sequence aligner using graphics processing units. *BMC research notes*, 5(1):27, 2012.

[KPD+04] Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. Versatile and open software for comparing large genomes. *Genome biology*, 5(2):R12, 2004.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[Kur99] Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw., Pract. Exper.*, 29(13):1149–1171, 1999.

[LD09] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[LD10] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[LSM12] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. Cushaw: a cuda compatible short read aligner to large genomes based on the burrows–wheeler transform. *Bioinformatics*, 28(14):1830–1837, 2012.

[LTPS09] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

[LYL+09] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[MBN+17] Martin D. Muggli, Alexander Bowe, Noelle R. Noyes, Paul S. Morley, Keith E. Belk, Robert Raymond, Travis Gagie, Simon J. Puglisi, and Christina Boucher. Succinct colored de bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[MIBT16] Takuya Mieno, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Shortest unique substring queries on run-length encoded strings. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*, pages 69:1–69:11, 2016.

[MM90] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 319–327, 1990.

[MNN17] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 408–424, 2017.

[Mut02] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 657–666, 2002.

[Nav13] Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):52:1–52:47, 2013.

[NM07] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[NN12] Gonzalo Navarro and Yakov Nekrich. Top-k document retrieval in optimal time and linear space. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1066–1077. SIAM, 2012.

[OV11] Alessio Orlandi and Rossano Venturini. Space-efficient substring occurrence estimation. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 95–106, 2011.

[PP09] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, pages 315–323, 2009.

[PWY13] Jian Pei, Wush Chi-Hsuan Wu, and Mi-Yen Yeh. On shortest unique substring queries. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 937–948, 2013.

[Sad02] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 225–232, 2002.

[Sad07] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

[SD10] Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373, 2010.

[SOG12] Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Inf. Comput.*, 213:13–22, 2012.

[TIBT14] Kazuya Tsuruta, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Shortest unique substrings queries in optimal time. In *SOFSEM 2014: Theory and Practice of Computer Science - 40th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 26-29, 2014, Proceedings*, pages 503–513, 2014.

[TWLY09] Alan Tam, Edward Wu, Tak-Wah Lam, and Siu-Ming Yiu. Succinct text indexing with wildcards. In *International Symposium on String Processing and Information Retrieval*, pages 39–50. Springer, 2009.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973.