

M.Sc. ENGG. THESIS

SIMPLE AND LIGHT-WEIGHT FILTER
BASED ALGORITHMS FOR CIRCULAR
STRINGS AND SEQUENCES

by

Md. Aashikur Rahman Azim

Submitted to

Department of Computer Science and Engineering

in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science and Engineering



Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology (BUET)

Dhaka 1000

December 2016

In the name of ALLAH

AUTHOR'S CONTACT

Md. Aashikur Rahman Azim


Lecturer


Department of Computer Science & Engineering
Bangladesh University of Engineering & Technology (BUET).

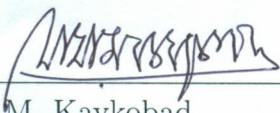
Email: aashik_azim@cse.buet.ac.bd

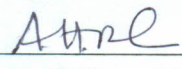
The thesis titled "SIMPLE AND LIGHT-WEIGHT FILTER BASED ALGORITHMS FOR CIRCULAR STRINGS AND SEQUENCES", submitted by Md. Aashikur Rahman Azim, Roll No. 1014052001 P, Session October 2014, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on December 12, 2016.

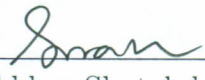
Board of Examiners

1. 

- Dr. M. Sohel Rahman
Professor
Department of Computer Science and Engineering, BUET, Dhaka.
Chairman
(Supervisor)
2. 

- Dr. M. Sohel Rahman
Professor and Head
Department of Computer Science and Engineering, BUET, Dhaka.
Member
(Ex-Officio)
3. 

- Dr. M. Kaykobad
Professor
Department of Computer Science and Engineering, BUET, Dhaka.
Member
4. 

- Dr. Atif Hasan Rahman
Assistant Professor
Department of Computer Science and Engineering, BUET, Dhaka.
Member
5. 

- Dr. Swakkhar Shatabda
Assistant Professor
Department of Computer Science and Engineering,
United International University, Dhaka.
Member
(External)

Candidate's Declaration

This is hereby declared that the work titled "SIMPLE AND LIGHT-WEIGHT FILTER BASED ALGORITHMS FOR CIRCULAR STRINGS AND SEQUENCES" is the outcome of research carried out by me under the supervision of Dr. M. Sohel Rahman, in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka. It is also declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.



Md. Aashikur Rahman Azim

Candidate

Acknowledgment

First of all I would like to thank my supervisor, Dr. M. Sohel Rahman, for assisting me throughout the thesis. Without his continuous supervision, guidance and advice it would not have been possible to complete this thesis. I am especially grateful to him for giving me his time whenever I needed, and always providing continuous support and motivation in my effort.

I also want to thank the other members of my thesis committee: Dr. M. Kaykobad, Dr. Atif Hasan Rahman and specially the external member Dr. Swakkhar Shatabda for their valuable suggestions.

Part of this research has been supported by an INSPIRE Strategic Partnership Award, administered by the British Council, Bangladesh for the project titled “Advances in Algorithms for Next Generation Biological Sequences”. I am really grateful to British Council, Bangladesh for their kind support.

Last but not the least, I am grateful to my guardians, family and friends for their patience, cooperation and inspiration during this period.

Abstract

String matching and sequence alignment problems are classical problems in Computer Science with extensive applications in different branches of science and engineering. These problems are interesting as fundamental computer science problems and are considered as basic requirements in many practical applications. They in fact appear in almost every textbook of algorithms and data structures. Circular strings or sequences appear in a number of biological contexts, in all domains of life: bacteria, archaea, and eukaryotes; and in viruses.

This thesis deals with three pattern matching problems, namely, Classical Pattern Matching problem, Circular Pattern Matching (CPM) problem and Circular Sequence Comparison (CSC). Here, we present some filtering techniques to solve these problems. At first, we propose a concept of a filtering pattern signature. Using this concept we develop an algorithm for search space reduction of the text string. Then we develop algorithms for circular strings and sequences. Our filters are simple and light-weight. Here light-weight means that executing the filters will be computationally easy and memory efficient. These filters will ensure that there are no false negatives; however, there could be false positives which will be handled in a subsequent processing step. In the sequel, only correct solutions will be determined by the combined algorithm. Our algorithms have been implemented and rigorously tested using real genome datasets. We compare our algorithms with the state of the art algorithms and the results are found to be excellent.

Contents

<i>Board of Examiners</i>	ii
<i>Candidate's Declaration</i>	iii
<i>Acknowledgment</i>	iv
<i>Abstract</i>	v
1 Introduction	1
1.1 Introduction	1
1.2 Applications and Motivations	4
1.3 Literature Review	8
1.4 Limitation of Recent Works	10
1.5 Objectives with Specific Aims and Possible Outcome	11
1.6 Thesis Organization	11
2 Preliminaries	13
2.1 Basic definitions in Stringology	13
2.2 Classical Pattern Matching Problem	15
2.3 Circular String	18
2.4 Exact Circular Pattern Matching	19
2.5 Approximate Circular Pattern Matching	20
2.5.1 Algorithm ACSMF-Simple of [1]	21

2.6	Circular Sequence Comparison	22
2.6.1	Algorithm <i>saCSC</i> of [2]	25
2.7	Numerical Representations	28
2.7.1	False Positives and Negatives	29
2.7.2	Category of Filters and Relevant Observations	29
2.7.3	Algorithmic Framework	30
2.8	Filters of Classical Pattern Matching	30
2.8.1	Filter 1	31
2.8.2	Filters 2 and 3	31
2.8.3	Filter 4	33
2.8.4	Filter 5	34
2.8.5	Filter 6	35
2.9	Filters of Circular Pattern Matching and Circular Sequence Comparison	35
2.9.1	Filter 1	36
2.9.2	Filters 2 and 3	36
2.9.3	Filter 4	38
2.9.4	Filter 5	39
2.9.5	Filter 6	40
2.10	Filters of ACPM	40
2.10.1	Filter 1	41
2.10.2	Filters 2 and 3	42
2.10.3	Filters 4	44
2.10.4	Filters 5	45
2.10.5	Filters 6	46
2.11	Summary	48
3	Filter based Algorithmic Framework	49
3.1	Pattern Signature using the Filters	49
3.2	Reduction of Search Space	51

3.2.1	An Illustrative Example for the ECPM Problem	53
3.3	The Combined Algorithm for Classical Pattern Matching Problem	55
3.4	The Combined Algorithm for ECPM and ACPM	57
3.5	The Algorithm for CSC Problem	59
3.6	Summary	60
4	Experimental Studies	62
4.1	Dataset	62
4.2	Environment & Experimental Settings	64
4.3	Effectiveness of Filters	65
4.4	Search Space Reduction	66
4.5	Experimental Results for Classical Pattern Matching	74
4.6	Comparison with Algorithms for ECPM	75
4.7	Experimental Results for ACPM	81
4.8	Experimental Results for CSC	88
4.9	Summary	98
5	Conclusion	99
5.1	Future Works	101

List of Figures

2.1	Graphical representation of Circular String $\mathcal{P} = AGCGGACTCT$ for first three rotations.	19
4.1	Search Space Reduction of text string for $n = 299MB$ and $5 \leq m \leq 35$	70
4.2	Search Space Reduction of text string for $n = 299MB$ and $40 \leq m \leq 450$	72
4.3	A graph representing elapsed-time (in seconds) and speed-up comparisons among FredNava [3], ACSMF-SimpleZero <i>k</i> [1] and Filtered-ECPM on a text of size 299MB	78
4.4	A graph representing elapsed-time (in seconds) and speed-up comparison among FredNava [3], ACSMF-Simple [1] and Filtered-ACPM considering all the six filters in a single pass for a text of size 1GB.	89
4.5	A graph representing elapsed-time (in seconds) and speed-up comparison between saCSC[2] and Filtered-CSC considering all the six filters for a text of size 700MB. Here, $\beta = \sqrt{m}$	92

List of Tables

2.1	Illustrates the arrays SA , iSA , and LCP for $w = \text{abbababba}$	14
2.2	q -gram profiles of strings \mathcal{P} and \mathcal{T} and q -gram distance $D_q(\mathcal{P}, \mathcal{T}) = 8$ between them.	23
2.3	q -gram profiles of strings $\mathcal{P}_1, \mathcal{P}_2, \mathcal{T}_1$ and \mathcal{T}_2 ; q -gram distance between \mathcal{P}_1 and \mathcal{T}_1 ; and q -gram distance between \mathcal{P}_2 and \mathcal{T}_2 , giving $D_{\beta,q}(\mathcal{P}, \mathcal{T}) = 8$	24
2.4	A brief summary of the 6 filters employed in the our algorithms.	32
3.1	An example simulation of Filtered-ECPM	54
4.1	A brief overview of genome data $GRCh37$	63
4.2	An overview of genome datasets: $299MB$, $700MB$ and $1GB$	64
4.3	Search Space Reduction of text string for $n = 299MB$ and $5 \leq m \leq 20$	67
4.4	Search Space Reduction of text string for $n = 299MB$ and $21 \leq m \leq 35$	67
4.5	Search Space Reduction of text string for $n = 299MB$ and $40 \leq m \leq 200$	68
4.6	Search Space Reduction of text string for $n = 299MB$ and $220 \leq m \leq 450$	69
4.7	Elapsed-time (in seconds) and comparisons among KMP, BM, Filtered-KMP, Filtered-BM, Only Filters, KMP in reduced text and BM in reduced text for a text of size 299MB.	73
4.8	Elapsed-time (in seconds) and speed-up comparisons among FredNava [3], ACSMF-SimpleZero <i>k</i> [1] and Filtered-ECPM on a text of size 299MB	76
4.9	Elapsed-time (in seconds) and speed-up comparisons among ACSMF-SimpleZero <i>k</i> and three variants of Filtered-ECPM (considering different combination of the filters) for a text of size 299MB.	80

4.10	Elapsed-time (in seconds) and speed-up comparison among FredNava [3], ACSMF-Simple [1] and Filtered-ACPM considering all the six filters in a single pass for a text of size 1GB	81
4.11	Elapsed-time (in seconds) and speed-up comparisons among ACSMF-Simple and Filtered-ACPM-[1..3] (considering first three combination of the filters) for a text of size 1GB	83
4.12	Elapsed-time (in seconds) and speed-up comparisons among ACSMF-Simple and Filtered-ACPM-[1..4] (considering first four combination of the filters) for a text of size 1GB	85
4.13	Elapsed-time (in seconds) and speed-up comparisons among ACSMF-Simple and Filtered-ACPM-[1..5] (considering first five combination of the filters) for a text of size 1GB	86
4.14	Elapsed-time (in seconds) and speed-up comparison between saCSC[2] and Filtered-CSC considering all the six filters for a text of size 700MB. Here, $\beta = \sqrt{m}$	90
4.15	Elapsed-time (in seconds) and speed-up comparisons between saCSC and Filtered-CSC-[1..3] (considering first three combination of the filters) for a text of size 700MB. Here, $\beta = \sqrt{m}$	93
4.16	Elapsed-time (in seconds) and speed-up comparisons between saCSC and Filtered-CSC-[1..4] (considering first four combination of the filters) for a text of size 700MB. Here, $\beta = \sqrt{m}$	95
4.17	Elapsed-time (in seconds) and speed-up comparisons between saCSC and Filtered-CSC-[1..5] (considering first five combination of the filters) for a text of size 700MB. Here, $\beta = \sqrt{m}$	96

List of Algorithms

2.1	Procedure <i>KMP – MATCHER</i> (\mathcal{T}, \mathcal{P}): KMP Algorithm	16
2.2	Procedure <i>COMPUTE – PREFIX – FUNCTION</i> (\mathcal{P}): KMP Prefix-Function	16
2.3	Procedure <i>BOYER – MOORE – MATCHER</i> (\mathcal{T}, \mathcal{P}): BM Algorithm	18
3.1	Procedure <i>PSF_FT</i> (<i>pattern_type</i>): Pattern Signature Algorithmic Framework using Filters 1 : 6 in a single pass	50
3.2	Procedure <i>RSS_FT</i> : Reduction of Search Space in a Text String using procedure <i>PSF_FT</i> (<i>pattern_type</i>)	52
3.3	Algorithm Filtered-BM/Filtered-KMP using Procedure <i>PSF_FT</i> (<i>pattern_type</i>) Algorithm 3.1	56
3.4	Algorithm Filtered-ECPM/Filtered-ACPM using Procedure <i>PSF_FT</i> (<i>pattern_type</i>) Algorithm 3.1	58
3.5	Algorithm Filtered-CSC ($\mathcal{T}[1 : n], \mathcal{P}[1 : m], \beta, q$): using procedure <i>PSF_FT</i> (<i>pattern_type</i>) Algorithm 3.1	61

Chapter 1

Introduction

The string-matching/pattern-matching and sequence alignment problems are the most studied problems in stringology, and there are many algorithms for solving these problems efficiently. In this chapter, we discuss the term stringology, text string, pattern string, etc. Then, we discuss some interesting problems in stringology, applications and/or motivations behind of these problems, and the literature review of these problems. Consequently, we discuss about the limitations of the present state of the art problem in the context of bioinformatics. Finally, we discuss our thesis objectives and outcomes.

1.1 Introduction

String Algorithms is a subset of stringology. Usually text and string have the same meaning and they are the basic types to carry information. Subfields of stringology include string matching, pattern matching, periodicities, data structures, text compression, sequence alignment, etc. Thus the improvement of string algorithms will benefit many other fields of stringology.

In computer science, pattern or string matching is the act of checking a perceived sequence of tokens for the presence of the constituents of some pattern. In contrast to pattern recognition, the match usually has to be exact. The patterns generally have the form of either sequences or tree structures. Usage of pattern matching include outputting the locations (if any) of a pattern within a token sequence, to output some component of the matched pattern, and to

substitute the matching pattern with some other token sequence (i.e., search and replace). In the context of the scope of this research work, the pattern can be in two different formation: linear and circular.

The classical pattern matching problem is to find all the occurrences of a given pattern \mathcal{P} (linear) of length m in a text \mathcal{T} of length n , both being sequences of characters drawn from a finite character set Σ . This problem is a basic requirement of many practical applications. However in most practical applications it is some sort of approximate version of the classic pattern matching problem that is of more interest.

Approximate pattern matching consists of finding all approximate occurrences of pattern \mathcal{P} in text \mathcal{T} . Approximate occurrences of \mathcal{P} are segments of \mathcal{T} that are close to \mathcal{P} according to a specific distance: their distance to \mathcal{P} must be not greater than a given integer k . Two common distances are the *Hamming distance* and the *Levenshtein distance*. With the *Hamming distance* related to the number of mismatches between the pattern and its approximate occurrences, the problem is also called approximate string matching with k mismatches. With the *Levenshtein distance* (or edit distance) the problem is known as the approximate string matching with k differences. The circular pattern, denoted $\mathcal{C}(\mathcal{P})$, corresponding to a given pattern $\mathcal{P} = \mathcal{P}_1 \dots \mathcal{P}_m$, is formed by connecting \mathcal{P}_1 with \mathcal{P}_m and forming a sort of a cycle; this gives us the notion where the same circular pattern can be seen as m different linear patterns, which would all be considered equivalent. In the Circular Pattern Matching (CPM) problem, we are interested in pattern matching between the text \mathcal{T} and the circular pattern $\mathcal{C}(\mathcal{P})$ of a given pattern \mathcal{P} . We can view $\mathcal{C}(\mathcal{P})$ as a set of m patterns starting at positions $j \in [1 : m]$ and wrapping around the end. In other words, in CPM, we search for all ‘conjugates’¹ of a given pattern in a given text.

In practical pattern-matching applications, the exact matching is not always pertinent. It is often more important to find string that matches a given pattern in a reasonably approximate way. This approximation is measured mainly by the so-called edit distance: the minimal number of local edit operations needed to transform one object into another. This transformation is called *sequence alignment problem*.

¹Two words x, y are conjugate if there exist words u, v such that $x = uv$ and $y = vu$.

Sequence alignment is a subfield of stringology. Sequence alignment is a standard technique in bioinformatics for identifying the relationships between residues in a collection of evolutionarily or structurally related elements/entities. A sequence alignment is a way of arranging the sequences of DNA (deoxyribonucleic acid)², RNA (ribonucleic acid)³ or protein⁴ to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that identical or similar characters are aligned in successive columns. In this thesis, one of the problems we consider is the pairwise circular sequence comparison problem. Under the edit distance model, it consists in finding an optimal linear alignment of two circular strings. This problem for two strings \mathcal{P} and \mathcal{T} of length m and $n \geq m$, respectively, can be solved under the edit distance model. The objective of this research is to develop new filter-based algorithms for different problems in strings and sequences. In this thesis, we are going to consider the following interesting problems:

- Classical Pattern Matching Problem
- Exact Circular Pattern Matching (ECPM)
- Approximate Circular Pattern Matching (ACPM)
- Circular Sequence Alignment/Comparison (CSC)

We plan to develop algorithms that are based on some filtering techniques. We will invent simple filters that can be specially used in the context of circular strings and sequences and that are simple and light-weight. Here light-weight means that executing the filters will be computationally easy and consume less memory. These filters will ensure that there are no false negatives; however, there could be false positives which will be handled in a subsequent

²DNA in turn can be thought of as a string of four types of nucleotides adenine (A), cytosine (C), guanine (G) and thymine (T).

³RNA is an important molecule with long chains of nucleotides. A nucleotide contains a nitrogenous base, a ribose sugar, and a phosphate. Just like DNA, RNA is vital for living beings.

⁴**Protein:** any of a class of nitrogenous organic compounds which have large molecules composed of one or more long chains of amino acids and are an essential part of all living organisms, especially as structural components of body tissues such as muscle, hair, etc., and as enzymes and antibodies.

processing step. In the sequel, only correct solutions will be determined by the combined algorithm.

Filters have been being used in the context of string matching algorithms since long. However, these filters do not work in the context of circular strings. Therefore we will have to consider new filters that are useful in the context of circular strings and sequences. We will employ the filters in a way to preprocess the given pattern and the text. After this preprocessing, we will get a text of reduced length on which we can apply any existing state-of-art algorithms to get the occurrences of the circular pattern. So in some sense our proposed algorithm will be “pluggable”, i.e., any state-of-art algorithm can be plugged into it. Hence our approach sort of promises that if a new algorithm can give good results, if you plug it into ours, the results will be even better.

We will conduct extensive experiments to compare our algorithms with the state of the art algorithms. We will use the datasets that have been used by the works [1, 2] to ensure a level-playing ground for performance comparison. We will consider each filters individually and in combinations. We will further consider different orders of the filters to apply in our algorithm. In the sequel, we will analyze the results and suggest the best possible ordering of the filters for our algorithms.

1.2 Applications and Motivations

The pattern matching problem is one of the most investigated problems in text algorithms. Implementations of algorithms for this problem are used daily for accessing information: to use Google search engine, to make a database query, to use the “search” or “replace” commands in text editors, etc. Here is a simple example of a pattern matching problem: find occurrences of the word “advanced” in the text “Special course in Computer Science: advanced text algorithms”. Pattern matching has to adapt itself to increasingly broader definitions of matching. In computational biology, one may be interested in finding a close mutation⁵, in

⁵The changing of the structure of a gene, resulting in a variant form which may be transmitted to subsequent generations, caused by the alteration of single base units in DNA, or the deletion, insertion, or rearrangement of larger sections of genes or chromosomes.

communications one may want to adjust for transmission noise, in texts it may be desirable to allow common typing errors. In multimedia one may want to adjust for loss compressions, occlusions, scaling, affine transformations or dimension loss.

The notion of a longest common subsequence (LCS) of two strings is widely used to compare files. The *diff*⁶ command of UNIX implements an algorithm based on this notion where lines of the files are considered as symbols. Informally, the result of a comparison gives the minimum number of operations (insert a symbol or delete a symbol) to transform one string into the other. The comparison of molecular sequences is basically done with a related concept, alignment of strings, which consists of aligning their symbols on vertical lines.

Apart from being interesting from pure combinatorial point view, CPM has applications in areas like, geometry, astronomy, computational biology etc. This type of circular patterns occur in the DNA of viruses [4, 5], bacteria [6], eukaryotic cells [7], and archaea [8]. As a result, as has been noted in [9], algorithms on circular strings seem to be important in the analysis of organisms with such structures.

Double-stranded, circular chromosomes and plasmids are found in most bacteria and archaea. Whole-genome comparison is a very useful tool in classifying bacterial strains, as well as inferring phylogenetic associations between them. This is due to the dense structure of bacterial chromosomes, caused by the absence of introns⁷, and the organisation of genes into operons⁸. The extended benefit of aligning plasmids is the ability to identify important genes, such as antibiotic resistance genes, thereby enabling their study and exploitation by genetic engineering techniques [10].

The related studies of sequence alignment include spelling correction, bitext word alignment, file comparison (difference), and amino acid sequences comparison. The early studies of sequence alignment originated from designing spelling correction. In 1957, researchers started to study engineering problems including spelling checkers for bitmap images of cursive writing and finding records in databases in spite of incorrect entries. These problems are basically

⁶The UNIX *diff* command is used to compare (find the differences) between two files.

⁷A segment of a DNA or RNA molecule which does not code for proteins and interrupts the sequence of genes.

⁸A unit made up of linked genes which is thought to regulate other genes responsible for protein synthesis.

consisting of two parts:

- Scanning strings and extracting words, and
- Comparing the extracted words against a known list of correctly spelled words (i.e., the dictionary).

The first part is an essential part of compiler developments, and has been being studied for years at that time, since the first compiler was written by Grace Hopper [11], in 1952, for the A-0 System language. The second part can be modelled as string alignment or string matching with respect to different constraints.

The first spell checker was developed by Les Earnest in 1961 that accessed a list of 10,000 acceptable words. In February 1971, Ralph Gorin created the first true spelling checker program for general English text: Spell for the DEC PDP-10 at Stanford University's Artificial Intelligence Laboratory. The first spelling correction was made by Gorin that searches the word list for plausible correct spellings heuristically. His idea is to search the words that differ by a single letter or adjacent letter transpositions and presenting them to the user.

The problem to search for similarities in the amino acid sequence of two proteins was first considered by Needleman et al. in 1970 [12]. They formulated the problem mathematically and named their model as global alignment. They proposed an algorithm, a.k.a. Needleman-Wunsch algorithm, of global alignment by using dynamic programming. The time complexity of dynamic programming based approach is $\mathcal{O}(n^2)$. Later the string comparison problem was considered in more general way by Wagner et al. [13]. They unified all related problems by the model of string-to-string correction. The possible applications of the string-to-string correction problem are the problems of automatic spelling correction and determining the longest common subsequence of two strings. Circular strings have also been studied in the context of sequence alignment. In [14], basic algorithms for pair wise and multiple circular sequence alignment have been presented. These results have later been improved in [15], where an additional preprocessing stage is added to speed up the execution time of the algorithm. Lee et al. [16] have considered Hamming distance and have presented efficient algorithms for finding the optimal alignment and consensus sequence of circular sequences on this distance metric.

The most familiar examples of such structures in eukaryotes⁹ are mitochondrial (mtDNA) and plastid DNA. MtDNA is, in most cases, inherited solely from the mother, and so is generally conserved. Human mtDNA is double-stranded, with a length of 16,569 base pairs (bp), consisting of just 37 genes encoding 13 proteins and 24 RNA molecules [17]. The absence of recombination in these sequences allows them to be used as simple indicators of phylogenetic evolution, and their high mutation rate is a powerful discriminative feature [18, 19]. There also exist smaller structures, called extrachromosomal circular DNA, which are similar to plasmids in bacterial cells. They are described as one of the characteristics of genomic plasticity in eukaryotes [20] and may be derived from mtDNA [21]. It is common knowledge that many viral genomes are circular. Viral genomes vary greatly in size and structure. They can be made up of either RNA or DNA, and can be single- or double-stranded. Multiple sequence alignment of viral genomes can be useful in the elucidation of novel sites of interest [22], as well as the inference of evolutionary relationships [23]. This is particularly important in studying their pathogenicity¹⁰, due to the rapid rate of mutation of viruses. Viroids are plant pathogens that comprise very small, single-stranded, circular RNA. Their multiple sequence alignment could prove useful in the analysis of their secondary structures and, therefore, the mechanisms by which they infect host plant cells [24].

In genome science, the BLAST [25], which was developed in 1990, is the most well known sequence alignment tool; it is elected as one of the milestones of DNA technology by editors of the *Nature Genetics*. However, to catch up the throughput of sequencing technologies, many alignment tools have been developed in the last few years. Each new sequencing technology has its own features and advantages. The developers of new sequence alignment tools must take them into consideration (e.g., the short sequence length of Illumina, SOLiD and Helicos reads, the di-base encoding of SOLiD reads).

⁹An organism consisting of a cell or cells in which the genetic material is DNA in the form of chromosomes contained within a distinct nucleus. Eukaryotes include all living organisms other than the eubacteria and archaea.

¹⁰Pathogenicity refers to the ability of an organism to cause disease (i.e., harm the host). This ability represents a genetic component of the pathogen and the overt damage done to the host is a property of the host-pathogen interactions.

1.3 Literature Review

The first linear-time string-matching algorithm was discovered by Morris and Pratt in 1970. It has been improved by Knuth in 1976 [26]. The search behaves like a recognition process by automation, and a character of the text is compared to a character of the pattern no more than $\log_\chi(m+1)$ (χ is the golden ratio $(1+\sqrt{5})/2$), where m is the length of pattern. Again, Boyer and Moore's algorithm [27] is considered the most efficient string matching algorithm in usual applications. A simplified version of it (or the entire algorithm) is often implemented in text editors for the “search” and “substitute” commands. We will further discuss these two algorithms in the next chapter (Chapter 2 in Section 2.2). Again, approximate or fuzzy string searching is a lively domain of research. It includes, for instance, the notion of *regular expressions* (a sequence of symbols and characters expressing a string or pattern to be searched for within a longer piece of text) to represent sets of strings. The *Shift-Or* algorithm was invented by Baeza-Yates and Gonnet in 1992 [28] to solve the fuzzy string matching. Perhaps, it is the first algorithm to solve the approximate pattern searching problem. Based on this work, later Wu and Manber in 1992 [29] invented *bitap algorithm* (also known as the shift-or, shift-and algorithm) to solve the same problem. This *bitap algorithm* is very fast in practice and very easy to implement.

Perhaps the first attempt to solve the problem of circular pattern matching has been documented in [30], where an $\mathcal{O}(n)$ -time algorithm is presented. A naive solution with quadratic complexity would be to apply a classical algorithm for searching a finite set of strings on the *trie* of rotations of \mathcal{P} after constructing it. The approach presented in [30] preprocesses \mathcal{P} by constructing a *suffix automaton* (A suffix automaton (also known as directed acyclic word graph (DAWG)) is a finite automaton that recognizes the set of suffixes¹¹ of a given string. For example, a suffix automaton for the string “suffix” can be queried for other strings; it will report “true” for any of the strings “suffix”, “uffix”, “ffix”, “fix”, “ix” and “x”, and “false” for any other string.) of the string \mathcal{PP} , because, every rotation of \mathcal{P} is a factor of \mathcal{PP} . Then, by feeding \mathcal{T} into the automaton, the lengths of the longest factors of \mathcal{PP} occurring in \mathcal{T} can be

¹¹A suffix of a string \mathcal{S} is a substring that occurs at the end of \mathcal{S}

found by the links followed in the automaton in time $\mathcal{O}(n)$. In [3], an optimal average-case algorithm for CPM has been presented. In particular, here the authors have shown that the average-case lower bound for the (linear) pattern matching of $\mathcal{O}(n \log_{\sigma} m/m)$ also holds for CPM, where $\sigma = |\Sigma|$. Recently, Chen et al. [31] have exploited word-level parallelism to present two fast average-case algorithms. The approximate version of the problem has also received attention in the literature very recently [1]. In [1], Barton et al. have first presented an efficient algorithm for CPM that runs in $\mathcal{O}(n)$ time on average. Based on the above, they have also devised fast average-case algorithms (ACSMF-Simple) for approximate circular string matching with k -mismatches. They have built a library for ACSMF-Simple algorithm. The library is freely available [32]. Notably, indexing circular patterns [33] has also been considered in the literature. In traditional pattern matching problem, indexing has always received particular attention. This is because in many practical problems, we need to handle batch of queries and, hence, it is computationally advantageous to preprocess the text in such a way that allows efficient query processing afterwards. The indexing version of Problem CPM is formally defined below:

Indexing for Circular Pattern Matching (ICPM): Given a text \mathcal{T} of length n , preprocess \mathcal{T} to answer the following form of queries:

Query: Given a pattern \mathcal{P} of length m , find the indices $i \in [1..n - m + 1]$ at which $\mathcal{C}(\mathcal{P})$ matches \mathcal{T} .

Again, an other variations of approximate circular pattern matching under the edit distance model [34] have also been considered in the literature. They have invented an algorithm to finds all k -approximate occurrences of pattern \mathcal{P} in text \mathcal{T} . They then extend each algorithm to solve the all-against-all variant of the CPM problem for both exact and k -approximate matches. Although the CPM problem has been studied since the 1980s, this was the first attempt on the all-against-all variant, without using a trivial application of standard CPM algorithms. Their algorithms solve the all-against-all ACPM problem in $\mathcal{O}(kmn)$ time on average, where k is number of mismatches, m is the length of pattern \mathcal{P} and n is the length of text \mathcal{T} .

On the other hand, the sequence alignment problem for two strings \mathcal{P} and \mathcal{T} of length m and $n \geq m$, respectively, can be solved under the edit distance model in time $\mathcal{O}(nm \log m)$

[35]. Several other super-quadratic [36] and approximate quadratic-time [37] algorithms exist. Trivially, for molecular biology applications, the same problem can be solved in time $\mathcal{O}(nm^2)$, if extending the problem with scoring matrices and affine gap penalty scores. A direct application of pairwise circular sequence comparison is progressive multiple circular sequence alignment [24, 38, 39]: A multiple sequence alignment (MSA) is a sequence alignment of three or more biological circular sequences, generally protein, DNA, or RNA.¹² Multiple circular sequence alignment has also been considered in [40] under the Hamming distance model.

In [2], the authors introduced a fast exact algorithm for circular sequence comparison under some realistic model. They introduced the β -blockwise q -gram distance between two strings \mathcal{P} and \mathcal{T} , that is, a more powerful generalization of the q -gram distance introduced as a string distance measure in [41]. Intuitively, and similarly to [42, 43], this generalization comprises partitioning \mathcal{P} and \mathcal{T} into β blocks each, as evenly as possible, computing the q -gram distance between the corresponding block pairs, and then summing up the distances computed block-wise. They presented an algorithm based on the suffix array [44] that finds the rotation of \mathcal{P} such that the β -blockwise q -gram distance between the rotated \mathcal{P} and \mathcal{T} is minimal, in time and space $\mathcal{O}(\beta m + n)$, where $m = |\mathcal{P}|$ and $n = |\mathcal{T}|$, thereby solving exactly the circular sequence comparison problem under the β -blockwise q -gram distance measure. They also presented a simple heuristic algorithm to solve an approximate version of the problem.

1.4 Limitation of Recent Works

The main context where these problems are useful, in is bioinformatics. Sadly, asymptotically fast or faster algorithms are sometimes not enough in the context of bioinformatics. It still requires huge time while running the state of art algorithms over the text of DNA (2-3 GB of size). This happens due to the searching of the actual pattern all over the text string of DNA. An alternate technique may be useful where we can think of reducing the search space of the text string: we then can have a reduced text string in which the state of art algorithms could

¹²The most widely used approach to multiple sequence alignments uses a heuristic search known as progressive technique (also known as the hierarchical or tree method).

be applied. Our main goal is to minimize the search space for these state of the art algorithms.

1.5 Objectives with Specific Aims and Possible Outcome

The main objectives of our thesis are as follows:

1. To study the state of the art of the classical string matching problem.
2. To study the state of the art of the circular string matching problem.
3. To study the state of the art of the circular sequence alignment problem.
4. To propose and study new biologically interesting and useful variants of the classical version of these problems.
5. To develop new filter-based algorithms for the above-mentioned problems that are simple and light-weight.
6. To conduct a detailed experimental study for performance comparison among the state of the art solutions and the proposed algorithms.

The outcomes of our study are as follows:

1. Some simple and new filters that could be useful in reducing the problem search space.
2. A repertoire of simple and light-weight algorithms for solving the problems with performance comparison results.
3. A software tool with all developed algorithms that can be used by bioinformaticians.

1.6 Thesis Organization

The remainder of this thesis is organized as follows.

In Chapter 2, we briefly discuss the concepts that are necessary to understand the idea of this study. We discuss a preliminary description of some terminologies and concepts related to stringology that will be used throughout this paper. We discuss the problems statement formally which will be used in this thesis. We also describe the main working principals of filtering techniques.

Chapter 3 presents the major contribution of this study. We formulate a framework based on filtering techniques described in Chapter 2. We present the different algorithms to solve the problems described in Chapter 2 using this framework.

In Chapter 4, the experimental analysis regarding the performance of the proposed methodology is presented. We first present a description of the various datasets that are used for our experiments. We then discuss the performance comparison between our proposed methodologies and the state of arts.

Finally, in Chapter 5 we briefly conclude this study. We try to provide some directions mention for future research.

Chapter 2

Preliminaries

This chapter presents the ideas necessary to comprehend the topics covered in this thesis. Related algorithms and methods employed in this work are also described in this chapter. We also formally present the problems that we handle in this thesis including some classical pattern matching problem, circular pattern matching problem and circular sequence comparison problem. Previous works of different topics related to this thesis are also presented in this chapter.

2.1 Basic definitions in Stringology

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ϵ is a string of length 0, that is, $|\epsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\epsilon\}$. For a string $w = xyz$, x , y and z are called a *prefix*, *factor* (or equivalently, *substring*), and *suffix* of w , respectively. The i -th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the factor of a string w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. For convenience, we assume $w[i : j] = \epsilon$ if $j < i$. A k -factor is a factor of length k .

Again, let Σ^q denote the set of all strings of length q over Σ for $q = 1, 2, \dots, \infty$. A q -gram is any string $v = a_1a_2 \dots a_q$ in Σ^q .

The *Parikh* vector associated with a string $w \in \Sigma^*$ is denoted by $P(w)$ and represents

a vector of size $|\Sigma|$, where each component denotes the number of occurrences in w of the corresponding letter from Σ . We denote by SA the *suffix array* of w of length n , that is, an integer array of size n storing the starting positions of all lexicographically sorted suffixes of w , i.e. for all $1 \leq r < n$, we have $w[SA[r-1]..n-1] < w[SA[r]..n-1]$. Let $lcp(r, s)$ denote the length of the longest common prefix between $w[SA[r]..n-1]$ and $w[SA[s]..n-1]$, for all positions r, s on w , and 0 if they do not have a common prefix. We denote by LCP the longest common prefix array of w defined by $LCP[r] = lcp(r-1, r)$, for all $1 \leq r < n$, and $LCP[0] = 0$. The inverse iSA of the array SA is defined by $iSA[SA[r]] = r$, for all $0 \leq r < n$. SA , iSA , and LCP of w can be computed in $\mathcal{O}(n)$ time and space [45]. Let the string $w = abbababba$. The following table (Table 2.1) illustrates the arrays SA , iSA , and LCP for w .

Table 2.1: Illustrates the arrays SA , iSA , and LCP for $w = abbababba$.

i	0	1	2	3	4	5	6	7	8
$w[i]$	a	b	b	a	b	a	b	b	a
$SA[i]$	8	3	5	0	7	2	4	6	1
$iSA[i]$	3	8	5	1	6	2	7	4	0
$LCP[i]$	0	1	2	4	0	2	3	1	3

Longest Common Extension (LCE): The *longest common extension* (LCE) problem takes as input a string s and many pairs (i, j) and computes, for each pair (i, j) , the longest substring of s that occurs both starting at position i and at j in s . To compute LCE, we perform the following linear-time and linear-space preprocessing:

- Compute arrays SA and iSA of \mathcal{P} [46].
- Compute array LCP of \mathcal{P} [45].
- Preprocess array LCP for *range minimum queries*¹, we denote this by RMQ_{LCP} [47].

With the preprocessing complete, the LCE of two suffixes of \mathcal{P} starting at positions p and q

¹In computer science, a range minimum query (RMQ) solves the problem of finding the minimal value in a sub-array of an array of comparable objects. Range minimum queries have several use cases in computer science such as the lowest common ancestor problem or the longest common prefix problem (LCP).

can be computed in constant time in the following way [48]:

$$LCE(\mathcal{P}, p, q) = LCP[RMQ_{LCP}(iSA[p] + 1, iSA[q])]. \quad (2.1)$$

We have $LCE(\mathcal{P}, 1, 2) = LCP[RMQ_{LCP}(iSA[2] + 1, iSA[1])] = LCP[RMQ_{LCP}(6, 8)] = 1$, implying that the LCE of *bbababba* and *bababba* is 1 using Table 2.1.

2.2 Classical Pattern Matching Problem

Informally, given a text and a pattern, the pattern matching problem compute the occurrences of the pattern within the text. The formal definition of classical pattern matching problem is given below.

Problem 1. (*Classical Pattern Matching Problem*). Given a pattern \mathcal{P} of length m and a text \mathcal{T} of length $n \geq m$, find all factors \mathcal{F} of \mathcal{T} such that $\mathcal{F} = \mathcal{P}$. Report all the starting positions i of all such factors \mathcal{F} in \mathcal{T} where $0 \leq i \leq n - m$.

Example 1. Suppose we have a pattern $\mathcal{P} = \text{atcgatg}$ and a text $\mathcal{T} = \text{aaagatcgatggg}$. It can be easily verified that the pattern \mathcal{P} matches \mathcal{T} at position 4, i.e., $\mathcal{T}[4 \dots 10] = \mathcal{P}$.

We will discuss here two fundamental pattern matching algorithms briefly. These are Knuth-Morris-Pratt [26] (KMP in short) algorithm and Boyer-Moore [27] (BM in short) algorithm. The main properties of KMP [26] algorithm is given below:

- The Knuth-Morris-Pratt algorithm uses the information gained by previous symbol comparisons.
- It never compares again a text symbol that has already matched a pattern symbol.
- As a result, the complexity of the searching phase of the Knuth-Morris-Pratt algorithm is in $\mathcal{O}(n)$, where n is the length of text string.
- However, a pre-processing of the pattern is necessary in order to analyze its structure.

- The pre-processing phase has a complexity of $\mathcal{O}(m)$, where m is the length of the pattern string.
- Since $m \leq n$, the overall complexity of the Knuth-Morris-Pratt algorithm is in $\mathcal{O}(n)$.

The KMP matching algorithm is given in pseudocode below (Algorithm 2.1) as the procedure *KMP-MATCHER*. This procedure calls the auxiliary procedure *COMPUTE-PREFIX-FUNCTION* (Algorithm 2.2).

Algorithm 2.1 Procedure *KMP-MATCHER*(\mathcal{T}, \mathcal{P}): KMP Algorithm

```

 $n \leftarrow |\mathcal{T}|$ 
 $m \leftarrow |\mathcal{P}|$ 
 $\pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(\mathcal{P})$ 
 $q \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$  do
  do while  $q > 0$  and  $\mathcal{P}[q + 1] \neq \mathcal{T}[i]$ 
  do  $q \leftarrow \pi[q]$ 
  if  $\mathcal{P}[q + 1] = \mathcal{T}[i]$  then
     $q \leftarrow q + 1$ 
  end if
  if  $q = m$  then
    print "Pattern occurs with shift"  $i - m$ 
     $q \leftarrow \pi[q]$ 
  end if
end for

```

Algorithm 2.2 Procedure *COMPUTE-PREFIX-FUNCTION*(\mathcal{P}): KMP Prefix-Function

```

 $m \leftarrow |\mathcal{P}|$ 
 $\pi[1] \leftarrow 0$ 
 $k \leftarrow 0$ 
for  $q \leftarrow 2$  to  $m$  do
  do while  $k > 0$  and  $\mathcal{P}[k + 1] \neq \mathcal{P}[q]$ 
  do  $k \leftarrow \pi[k]$ 
  if  $\mathcal{P}[k + 1] = \mathcal{P}[q]$  then
     $k \leftarrow k + 1$ 
  end if
   $\pi[q] \leftarrow k$ 
end for
return  $\pi$ 

```

The main properties of BM [27] algorithm is given below:

- This algorithm tends to have the best performance in practice, as it often runs in sub linear time.
- The worst case running time is as bad as that of the naive algorithm.
- At any moment, imagine that the pattern is *aligned* with a portion of the text of the same length, though only a part of the aligned text may have been matched with the pattern.
- Henceforth, alignment refers to the substring of text \mathcal{T} that is aligned with pattern \mathcal{P} and l is the index of the left end of the alignment; i.e., $\mathcal{P}[0]$ is aligned with $\mathcal{T}[l]$ and, in general, $\mathcal{P}[i]$, $0 \leq i < m$, with $\mathcal{T}[l + i]$, where m is length of \mathcal{P} and n is the length of \mathcal{T} .
- Whenever there is a mismatch, the pattern is shifted to the right, i.e., l is increased, as explained in [27].
- The overall complexity of the BM algorithm is in $\mathcal{O}(mn)$.

We define a function $last(c)$ that takes a character c from the alphabet and specifies how far we may shift the pattern \mathcal{P} if a character equal to c is found in the text \mathcal{T} that does not match the pattern.

$$last(c) = \begin{cases} \text{index of the last occurrence of } c \text{ in pattern } \mathcal{P}, & \text{if } c \text{ is in } \mathcal{P} \\ -1, & \text{otherwise} \end{cases}$$

The MB matching algorithm is given in pseudocode below (Algorithm 2.3) as the procedure *BOYER – MOORE – MATCHER*. This procedure uses the auxiliary function $last$ as described before.

Again, approximate string matching is also another main problem in classical string algorithms, with applications to text searching, computational biology, pattern recognition, etc. Given a text \mathcal{T} of length n , a pattern \mathcal{P} of length m , and a maximal number of errors allowed,

Algorithm 2.3 Procedure *BOYER – MOORE – MATCHER*(\mathcal{T}, \mathcal{P}): BM Algorithm

```

Compute function last
 $n \leftarrow |\mathcal{T}|$ 
 $m \leftarrow |\mathcal{P}|$ 
 $i \leftarrow m - 1$ 
 $j \leftarrow m - 1$ 
repeat
  if  $\mathcal{P}[j] = \mathcal{T}[i]$  then
    if  $j = 0$  then
      return  $i$  /*We have a match*/
    else
       $i \leftarrow i - 1$ 
       $j \leftarrow j - 1$ 
    end if
  else
     $i \leftarrow i + m - \text{Min}(j, 1 + \text{last}[\mathcal{T}[i]])$ 
     $j \leftarrow m - 1$ 
  end if
until  $i > n - 1$ 

```

k , we want to compute all text positions where the pattern \mathcal{P} matches the text \mathcal{T} up to k errors. Errors can be substituting, deleting or inserting a character. The formal definition of approximate classical pattern matching problem is given below.

Approximate Classical Pattern Matching Problem with k -Errors: *Given a pattern \mathcal{P} of length m and a text \mathcal{T} of length $n \geq m$, and an integer threshold $k < m$, find all factors \mathcal{F} of \mathcal{T} such that $\mathcal{F} \equiv_k \mathcal{P}$. Report all the starting positions i of all such factors \mathcal{F} in \mathcal{T} where $0 \leq i \leq n - m$.*

Example 2. *Suppose we have a pattern $\mathcal{P} = \text{atc gatg}$, a text $\mathcal{T} = \text{aatcgatttggg}$ and $k = 1$. It can be easily verified that the pattern \mathcal{P} matches \mathcal{T} at position 1 with $k = 1$, i.e., $\mathcal{T}[1 \dots 7] \equiv_1 \mathcal{P}$.*

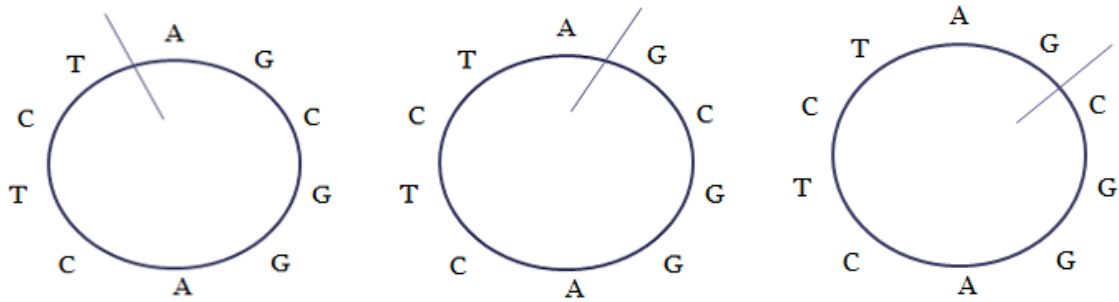
2.3 Circular String

A circular string of length m can be viewed as a traditional linear string which has the left-most and right-most symbols wrapped around and stuck together in some way. Under this notion, the same circular string can be seen as m different linear strings, which would all be considered

equivalent. Given a string \mathcal{P} of length m , we denote by $\mathcal{P}^i = \mathcal{P}[i : m]\mathcal{P}[1 : i - 1]$, $0 < i < m$, the i -th rotation of \mathcal{P} and $\mathcal{P}^0 = \mathcal{P}$. An example 3 is shown illustrating all rotations of circular string $\mathcal{P} = atcgatg$.

Example 3. Suppose we have a pattern $\mathcal{P} = atcgatg$. The pattern \mathcal{P} has the following rotations (i.e., conjugates): $\mathcal{P}^1 = tcgatga$, $\mathcal{P}^2 = cgatgat$, $\mathcal{P}^3 = gatgatc$, $\mathcal{P}^4 = atgatcg$, $\mathcal{P}^5 = tgatcga$, $\mathcal{P}^6 = gatcgat$.

Consider another circular string $\mathcal{P} = AGCGGACTCT$. The graphical representation of first three rotations are shown in Figure 2.1.



(a) $\mathcal{P}^0 = AGCGGACTCT$. (b) $\mathcal{P}^1 = GCGGACTCTA$. (c) $\mathcal{P}^2 = CGGACTCTAG$.

Figure 2.1: Graphical representation of Circular String $\mathcal{P} = AGCGGACTCT$ for first three rotations.

2.4 Exact Circular Pattern Matching

Here we consider the problem of finding occurrences of a pattern string \mathcal{P} of length m with circular structure in a text string \mathcal{T} of length n with linear structure. For instance, the DNA sequence of many viruses has a circular structure. So if a biologist wishes to find occurrences of a particular virus in a carrier's DNA sequence, which may not be circular, (s)he must locate all positions in \mathcal{T} where at least one rotation of \mathcal{P} occurs. This is the problem of *circular pattern matching* (CPM). The ECPM problem we handle in this thesis can be formally defined as follows.

Problem 2. (*Exact Circular Pattern Matching (ECPM)*). Given a pattern \mathcal{P} of length m and a text \mathcal{T} of length $n \geq m$, find all factors \mathcal{F} of \mathcal{T} such that $\mathcal{F} = \mathcal{P}^i$, for some $0 \leq i < m$. And if we have $\mathcal{F} = \mathcal{P}^i$ for some $0 \leq i < m$, then we say that the circular pattern $\mathcal{C}(\mathcal{P})$ matches \mathcal{T} at position i .

Example 4. Suppose we have a pattern $\mathcal{P} = atcgatg$ and a text $\mathcal{T} = aaggc gatg$. Based on Example 3 (represents all rotations of pattern \mathcal{P}) it can be easily verified that the pattern \mathcal{P}^2 matches \mathcal{T} at position 4.

2.5 Approximate Circular Pattern Matching

The *Hamming distance* between strings \mathcal{P} and \mathcal{T} , both of length n , is the number of positions i , $1 \leq i \leq n$, such that $\mathcal{P}[i] \neq \mathcal{T}[i]$. Given a non-negative integer k , we write $\mathcal{P} \equiv_k \mathcal{T}$ or equivalently say that \mathcal{P} k -matches \mathcal{T} , if the Hamming distance between \mathcal{P} and \mathcal{T} is at most k . In biology, the *Hamming distance* is popularly referred to as the *Mutation distance*. A little mutation could be considered and in fact anticipated while finding the occurrences of a particular (circular) virus in a carrier's DNA sequence. This scenario in fact refers to *approximate circular pattern matching* (ACPM). If, $k = 0$, then we get the exact CPM, i.e., mutations are not considered. Note carefully that in this setting, ACPM also returns all the occurrences returned by CPM; it computes the occurrences allowing **up to** k mismatches/mutations. The ACPM problem we handle in this thesis can be formally defined as follows.

Problem 3. (*Approximate Circular Pattern Matching with k -mismatches under the Hamming distance model (i.e., mutations) (ACPM)*). Given a pattern \mathcal{P} of length m , a text \mathcal{T} of length $n > m$, and an integer threshold $k < m$, find all factors \mathcal{F} of \mathcal{T} such that $\mathcal{F} \equiv_k \mathcal{P}^i$ for some $0 \leq i < m$. And when we have a factor $\mathcal{F} = \mathcal{T}[j : j + |\mathcal{F}| - 1]$ such that $\mathcal{F} \equiv_k \mathcal{P}^i$ we say that the circular pattern $\mathcal{C}(\mathcal{P})$ k -matches \mathcal{T} at position j . We also say that this k -match is due to \mathcal{P}^i , i.e., the i th rotation of \mathcal{P} .

Example 5. Suppose we have a pattern $\mathcal{P} = atc gatg$, a text $\mathcal{T} = cgatgaaaatt$ and $k = 1$. Based on Example 3 (represents all rotations of pattern \mathcal{P}) it can be easily verified that the

pattern \mathcal{P}^2 matches \mathcal{T} at position 0 with $k = 1$ means $\mathcal{T}[0 \dots 6] \equiv_1 \mathcal{P}^2$.

2.5.1 Algorithm ACSMF-Simple of [1]

In [1], the authors showed an algorithm to solve the problem ACPM. Here, we describe the steps of the algorithm as follows:

1. Construct the string $\mathcal{P}' = \mathcal{P}[0..m-1]\mathcal{P}[0..m-2]$ of length $2m-1$. According to [1], any rotation of \mathcal{P} is a factor of \mathcal{P}' .
2. The pattern \mathcal{P}' is partitioned in $2k+4$ fragments of length $\lfloor \frac{(2m-1)}{(2k+4)} \rfloor$ and $\lceil \frac{(2m-1)}{(2k+4)} \rceil$. According to [1], at least $k+1$ of the $2k+4$ fragments are factors of any rotation of \mathcal{P} .
3. Match the $2k+4$ fragments against the text \mathcal{T} using an *Aho Corasick automaton* [49]. Let \mathcal{L} be a list of size Occ of tuples, where $\langle p_{\mathcal{P}'}, l, p_{\mathcal{T}} \in \mathcal{L} \rangle$ is a 3-tuple such that $0 \leq p_{\mathcal{P}'} < 2m-1$ is the position where the fragment occurs in \mathcal{P}' , l is the length of the corresponding fragment, and $0 \leq p_{\mathcal{T}} < n$ is the position where the fragment occurs in \mathcal{T} .
4. For each tuple $\langle p_{\mathcal{P}'}, l, p_{\mathcal{T}} \in \mathcal{L} \rangle$, they try to extend $k+1$ times to the right via computing

$$\mathcal{E}_r^0 \leftarrow LCE(B, p_{\mathcal{P}'} + l, 2m - 1 + p_{\mathcal{T}} + l) + 1 \quad (2.2)$$

$$\mathcal{E}_r^1 \leftarrow LCE(B, p_{\mathcal{P}'} + l + \mathcal{E}_r^0, 2m - 1 + p_{\mathcal{T}} + l + \mathcal{E}_r^0) + 1 \quad (2.3)$$

...

$$\mathcal{E}_r^{k-1} \leftarrow LCE(B, p_{\mathcal{P}'} + l + \mathcal{E}_r^{k-2}, 2m - 1 + p_{\mathcal{T}} + l + \mathcal{E}_r^{k-2}) + 1 \quad (2.4)$$

$$\mathcal{E}_r^k \leftarrow LCE(B, p_{\mathcal{P}'} + l + \mathcal{E}_r^{k-1}, 2m - 1 + p_{\mathcal{T}} + l + \mathcal{E}_r^{k-1}) + 1; \quad (2.5)$$

in other words, they compute the length \mathcal{E}_r^k of the longest common prefix of $\mathcal{P}'[p_{\mathcal{P}'} + l..2m-1]$ and $\mathcal{T}[p_{\mathcal{T}} + l..n-1]$, both being suffixes of B , with k mismatches. Similarly,

they try to extend to the left $k + 1$ times via computing \mathcal{E}_l^k using LCE queries on the suffixes of B_r .

5. For each tuple $\langle p_{\mathcal{P}'}, l, p_{\mathcal{T}} \in \mathcal{L} \rangle$, they try to extend, they also maintain an array M of size $m - 1$ initialised with zeros, where they mark the position of the i -th left and right mismatch, $1 \leq i \leq k$, by setting:

$$M[p_{\mathcal{P}'} - \mathcal{E}_l^{i-1} - 1] \leftarrow 1 \text{ and } M[p_{\mathcal{P}'} + \mathcal{E}_r^{i-1} + l] \leftarrow 1. \quad (2.6)$$

6. For each $\mathcal{E}_l^k, \mathcal{E}_r^k, M$ computed for tuple $\langle p_{\mathcal{P}'}, l, p_{\mathcal{T}} \in \mathcal{L} \rangle$, they report all the valid starting positions in \mathcal{T} by first checking if the total length $\mathcal{E}_l^k + l + \mathcal{E}_r^k \geq m$; that is the length of the full extension of the fragment is greater than or equal to m . If that is the case, then they count the total number of mismatches of the occurrences at starting positions $\max\{p_{\mathcal{T}} - \mathcal{E}_l^k, p_{\mathcal{T}} + l - m\}, \dots, \min\{p_{\mathcal{T}} + l - m + \mathcal{E}_r^k, p_{\mathcal{T}}\}$, by first summing up the mismatches for the leftmost starting position $\mu_j \leftarrow M[p_{\mathcal{P}'} - \mathcal{E}_l^k] + \dots + M[p_{\mathcal{P}'} - \mathcal{E}_l^k + m - 1]$, where $j = \max\{p_{\mathcal{T}} - \mathcal{E}_l^k, p_{\mathcal{T}} + l - m\}$.

For each subsequent position $j + 1$, they subtract the value of the leftmost element of M computed for μ_j and add the value of the next element to compute μ_{j+1} . In case $\mu_j \leq k$, they report position j .

2.6 Circular Sequence Comparison

Circular Sequence Comparison (CSC) was introduced in [41, 2]. Based on the some terminology introduced in [41, 2], we describe some definitions here. The q -gram profile of a string \mathcal{P} is the vector $G_q(\mathcal{P})$, where $q > 0$ and $G_q(\mathcal{P})[v]$ denotes the total number of occurrences of q -gram $v \in \Sigma^q$ in \mathcal{P} . The q -gram distance between two strings \mathcal{P} and \mathcal{T} is defined as follows:

$$D_q(\mathcal{P}, \mathcal{T}) = \sum_{v \in \Sigma^q} |G_q(\mathcal{P})[v] - G_q(\mathcal{T})[v]| \quad (2.7)$$

Note that D_q is a pseudo-metric as $D_q(\mathcal{P}, \mathcal{T})$ can be 0 even if $\mathcal{P} \neq \mathcal{T}$. And the β -blockwise

q -gram distance two strings \mathcal{P} and \mathcal{T} of length m and n , respectively, is defined as follows:

$$D_{\beta,q}(\mathcal{P}, \mathcal{T}) = \sum_{j=0}^{\beta-1} D_q(\mathcal{P}[\frac{jm}{\beta}..\frac{(j+1)m}{\beta} - 1], \mathcal{T}[\frac{jn}{\beta}..\frac{(j+1)n}{\beta} - 1]) \quad (2.8)$$

Example 6. Let $\mathcal{P} = GGAGTCTA$, $\mathcal{T} = TTCTAGCG$, and $q = 3$. Table 2.2 shows the q -gram profiles of strings \mathcal{P} and \mathcal{T} and the q -gram distance between them.

Table 2.2: q -gram profiles of strings \mathcal{P} and \mathcal{T} and q -gram distance $D_q(\mathcal{P}, \mathcal{T}) = 8$ between them.

	$G_q(\mathcal{P})$	$G_q(\mathcal{T})$	$D_q(\mathcal{P}, \mathcal{T})$
AAA	0	0	0
AGC	0	1	1
AGT	1	0	1
CCC	0	0	0
CTA	1	1	0
GAG	1	0	1
GCG	0	1	1
GGA	1	0	1
GGG	0	0	0
GTC	1	0	1
TAG	0	1	1
TCT	1	1	0
TTC	0	1	1
TTT	0	0	0

Example 7. Following Example 6, let $\mathcal{P} = GGAGTCTA$ and $\mathcal{T} = TTCTAGCG$, $q = 3$, and $\beta = 2$. Further let $\mathcal{P}_1 = GGAG$, $\mathcal{P}_2 = TCTA$ and $\mathcal{T}_1 = TTCT$, $\mathcal{T}_2 = AGCG$ be the two blocks of \mathcal{P} and \mathcal{T} , respectively. Table 2.3 shows the q -gram profiles of strings \mathcal{P}_1 , \mathcal{P}_2 , \mathcal{T}_1 , and \mathcal{T}_2 ; and the q -gram distance between \mathcal{P}_1 and \mathcal{T}_1 and the q -gram distance between \mathcal{P}_2 and \mathcal{T}_2 .

In this thesis, we consider the following CSC problem, where we search for the i -th rotation of \mathcal{P} that minimizes its blockwise distance from \mathcal{T} as defined in ([50]). Ties are broken

Table 2.3: q -gram profiles of strings \mathcal{P}_1 , \mathcal{P}_2 , \mathcal{T}_1 and \mathcal{T}_2 ; q -gram distance between \mathcal{P}_1 and \mathcal{T}_1 ; and q -gram distance between \mathcal{P}_2 and \mathcal{T}_2 , giving $D_{\beta,q}(\mathcal{P}, \mathcal{T}) = 8$.

	$G_q(\mathcal{P}_1)$	$G_q(\mathcal{T}_1)$	$D_q(\mathcal{P}_1, \mathcal{T}_1)$	$G_q(\mathcal{P}_2)$	$G_q(\mathcal{T}_2)$	$D_q(\mathcal{P}_2, \mathcal{T}_2)$
AAA	0	0	0	0	0	0
AGC	0	0	0	0	1	1
AGT	0	0	0	0	0	0
CCC	0	0	0	0	0	0
CTA	0	0	0	1	0	1
GAG	1	0	1	0	0	0
GCG	0	0	0	0	1	1
GGA	1	0	1	0	0	0
GGG	0	0	0	0	0	0
GTC	0	0	0	0	0	0
TAG	0	0	0	0	0	0
TCT	0	1	1	1	0	1
TTC	0	1	1	0	0	0
TTT	0	0	0	0	0	0

arbitrarily.

Problem 4. (*Circular Sequence Comparison (CSC)*). Given a pattern \mathcal{P} of length m , a text \mathcal{T} of length $n \geq m$, and integers $\beta \geq 1$ and $q < m$, find i such that $D_{\beta,q}(\mathcal{P}^i, \mathcal{T})$ is minimal.

2.6.1 Algorithm *saCSC* of [2]

In [2], the authors showed an algorithm to solve the problem CSC in exact fashion. This algorithm is based on *suffix array*. For this reason, they named this algorithm as **saCSC**. They partially followed the idea from [51]. The work of [51] investigates the string matching problem in the setting of k -abelian equivalences: two strings are considered k -abelian equivalent for some positive integer k , if they have the same length and share the same factors of length at most k , including multiplicities. Note that if k is greater than or equal to the string's length, then the strings must be equal. A version of this result, called extended k -abelian equivalence, focuses only on the factors of length k . By setting $k = q$, it is quite straightforward to notice the equivalence with q -grams. Therefore, in order to avoid confusion we refer to the former notion from now on as q -abelian equivalence.

In [51], the authors propose a linear-time algorithm to solve the string matching problem when looking at q -abelian equivalent strings: given a string \mathcal{P} of length m , a string \mathcal{T} of length $n \geq m$, and a positive integer $q < m$, all factors of \mathcal{T} that are q -abelian equivalent to \mathcal{P} can be found in time and space $\mathcal{O}(m + n)$. The idea of the algorithm in [51] consists of constructing the suffix array of the string $\mathcal{P}\mathcal{T}$ and ranking sets of identical q -length prefixes in the suffix array in the order of their appearance. Then it constructs new strings based on this ranking, and solves the problem as in the *jumbled matching* case [52], i.e, identifying within \mathcal{T} all factors that have the same Parikh vector (see Section 2.1) as \mathcal{P} .

They construct the suffix array of the string $\mathcal{P}\mathcal{P}\mathcal{T}$ and assign a rank to the prefix with length q of each suffix with length at least q , based on its order in the suffix array. That is, the first i_0 suffixes in the suffix array, all sharing the same prefix of length at least q , will get rank 0; the next i_1 suffixes sharing the same prefix of length at least q , different from the previous one, will get rank 1, and so on. Next, based on this ranking, they construct two new strings \mathcal{P}'

of length $2mq + 1$ and \mathcal{T}' of length $nq + 1$, such that $\mathcal{P}'[i] = j$, if j is the rank of the q -length prefix of the $(i + 1)$ th suffix of $\mathcal{P}\mathcal{P}$ in the suffix array of $\mathcal{P}\mathcal{P}\mathcal{T}$ (the same goes for \mathcal{T}). It is not difficult to see that the ranks go up at most to value $m + nq + 1$. However, they reduce this value to $m + 2$ by introducing two new ranks $a_{\mathcal{P}}$ and $a_{\mathcal{T}}$: they can conceptually replace $a_{\mathcal{P}}$ by every letter of \mathcal{P}' that does not occur in \mathcal{T}' , and by $a_{\mathcal{T}}$ every letter of \mathcal{T}' that does not occur in \mathcal{P}' . Hence they consider that the new strings \mathcal{P}' and \mathcal{T}' are defined over an integer alphabet of size at most $\min(nq + 1, m) + 2 \leq m + 2$. Based on these preliminaries, the authors showed the following algorithmic steps to solve the problem CSC for $\beta = 1$.

Step1: Construct the SA, iSA, and LCP of $\mathcal{P}\mathcal{P}\mathcal{T}$. Rank the q -length prefixes of suffixes using LCP-array queries. Construct \mathcal{P}' and \mathcal{T}' , as well as $P(\mathcal{T}')$, the Parikh vector recording for each letter of \mathcal{T}' the number of its occurrences in \mathcal{T}' with the proper use of $a_{\mathcal{P}}$ and $a_{\mathcal{T}}$, the ranks that do not occur in either \mathcal{T}' or \mathcal{P}' , respectively. Moreover, create $diff = P(\mathcal{T}')$ and $\delta_0 = \sum_{i=0}^{|\mathcal{P}(\mathcal{T}')|} P(\mathcal{T}'[i])$.

Step2: Read the first $mq + 1$ letters of \mathcal{P}' , which constitute our sliding window of length m on the string $\mathcal{P}\mathcal{P}$. When reading letter $\mathcal{P}'[i]$, update $diff$ by decreasing by 1 the value of the newly read element, and update δ_0 , by either increasing the current value of the difference when there were read too many of the current letters, or decreasing it, when more of these letters still occur in \mathcal{T}' .

$$diff[\mathcal{P}'[i]] = diff[\mathcal{P}'[i]] - 1 \quad (2.9)$$

and

$$\delta_0 = \begin{cases} \delta_0 - 1, & \text{if } diff[\mathcal{P}'[i]] \geq 0 \\ \delta_0 + 1, & \text{if } diff[\mathcal{P}'[i]] < 0 \end{cases} \quad (2.10)$$

Step3: Let i be the current position in \mathcal{P}' and repeat this step, one position at a time. Shift the window to the right, update the information for $diff$ as follows:

$$diff[\mathcal{P}'[i]] = diff[\mathcal{P}'[i]] + 1 \quad (2.11)$$

and

$$diff[\mathcal{P}'[i+m]] = diff[\mathcal{P}'[i+m]] - 1 \quad (2.12)$$

and calculate δ_{i+1} , based on this information, sequentially applying the two following rules.

$$\delta_{i+1} = \begin{cases} \delta_i - 1, & \text{if } diff[\mathcal{P}'[i]] \leq 0 \\ \delta_i + 1, & \text{if } diff[\mathcal{P}'[i]] > 0 \end{cases} \quad (2.13)$$

and

$$\delta_{i+1} = \begin{cases} \delta_{i+1} - 1, & \text{if } diff[\mathcal{P}'[i+m]] \leq 0 \\ \delta_{i+1} + 1, & \text{if } diff[\mathcal{P}'[i+m]] > 0 \end{cases} \quad (2.14)$$

General Algorithm for $\beta \geq 1$: They generalized this algorithm to solve the CSC problem for any $\beta \geq 1$, which gives algorithm saCSC. they maintained a Parikh vector for each block, and applied the above basic algorithm for each pair of blocks, computing their q -gram distance. If they denote by $P_j(\mathcal{T}')$ and $diff_j$, for all $0 \leq j < \beta$, the β Parikh vectors of \mathcal{T}' and of the q -gram distances, respectively, as well as by $\delta_{i,j}$ the q -gram distance between the j th block of \mathcal{T} and \mathcal{P}^i , then the updates will be given by the formulae below. Hence, at each position $i < m$, they update all of the β Parikh vectors corresponding to the blocks, as previously described, in time $\mathcal{O}(\beta)$. As an example, see here the modification of the previous **Step 3**, with the other two steps being easily adapted in a similar fashion.

Step 3': When shifting the window one position to the right from position i , update the information for every $diff_j$, where $0 \leq j < \beta$, as follows:

$$diff_j[\mathcal{P}'[i + \frac{jm}{\beta}]] = diff_j[\mathcal{P}'[i \frac{jm}{\beta}]] + 1 \quad (2.15)$$

and

$$diff_j[\mathcal{P}'[i + \frac{(j+1)m}{\beta}]] = diff_j[\mathcal{P}'[i \frac{(j+1)m}{\beta}]] - 1, \quad (2.16)$$

and calculate $\delta_{i+1,j}$ based on this, sequentially applying the two following rules.

$$\delta_{i+1,j} = \begin{cases} \delta_{i,j} - 1, & \text{if } \text{diff}_j[\mathcal{P}'[i + \frac{jm}{\beta}]] \leq 0 \\ \delta_{i,j} + 1, & \text{if } \text{diff}_j[\mathcal{P}'[i + \frac{jm}{\beta}]] > 0 \end{cases} \quad (2.17)$$

and

$$\delta_{i+1,j} = \begin{cases} \delta_{i+1,j} - 1, & \text{if } \text{diff}_j[\mathcal{P}'[i + \frac{(j+1)m}{\beta}]] \geq 0 \\ \delta_{i+1,j} + 1, & \text{if } \text{diff}_j[\mathcal{P}'[i + \frac{(j+1)m}{\beta}]] < 0 \end{cases} \quad (2.18)$$

Algorithm saCSC solves the CSC problem in $\mathcal{O}(\beta m + n)$ time and space [2].

2.7 Numerical Representations

We consider the DNA alphabet, i.e., $\Sigma = \{a, c, g, t\}$. In our approach, each character of the alphabet is associated to a numeric value as follows. Each character is assigned a unique numbers from the range $[1 \dots |\Sigma|]$. Although this is not essential, we conveniently assign the numbers from the range $[1 \dots |\Sigma|]$ to the characters of Σ following their inherent lexicographical order. We use $\text{num}(x), x \in \Sigma$ to denote the numeric value of the character x . So, we have $\text{num}(a) = 1, \text{num}(c) = 2, \text{num}(g) = 3$ and $\text{num}(t) = 4$. For a string S , we use the notation S_N to denote the numeric representation of the string S ; and $S_N[i]$ denotes the numeric value of the character $S[i]$. So, if $S[i] = g$ then $S_N[i] = \text{num}(g) = 3$. The concept of circular string and their rotations also apply naturally on their numeric representations as is illustrated in Example 3 below.

Example 8. Suppose we have a pattern $\mathcal{P} = \text{atcgatg}$. The numeric representation of \mathcal{P} is $\mathcal{P}_N = 1423143$. And this numeric representation has the following rotations: $\mathcal{P}_N^1 = 4231431$, $\mathcal{P}_N^2 = 2314314$, $\mathcal{P}_N^3 = 3143142$, $\mathcal{P}_N^4 = 1431423$, $\mathcal{P}_N^5 = 4314231$, $\mathcal{P}_N^6 = 3142314$.

2.7.1 False Positives and Negatives

In the context of our filter based algorithm the concept of false positives and negatives is important. So, we briefly discuss this concept here. Suppose we have an algorithm \mathcal{A} to solve a problem \mathcal{B} . Now suppose that \mathcal{S}_{true} represents the set of true solutions for the problem \mathcal{B} . Further suppose that \mathcal{A} computes the set $\mathcal{S}_{\mathcal{A}}$ as the set of solutions for \mathcal{B} . Now assume that $\mathcal{S}_{true} \neq \mathcal{S}_{\mathcal{A}}$. Then, the set of false positives can be computed as follows: $\mathcal{S}_{\mathcal{A}} \setminus \mathcal{S}_{true}$. In other words, the set computed by \mathcal{A} contains some solutions that are not true solutions for problem \mathcal{B} . And these are the false positives, because, $\mathcal{S}_{\mathcal{A}}$ falsely marked these as solutions (i.e., positive). On the other hand, the set of false negatives can be computed as follows: $\mathcal{S}_{true} \setminus \mathcal{S}_{\mathcal{A}}$. In other words, false negatives are those members in \mathcal{S}_{true} that are absent in $\mathcal{S}_{\mathcal{A}}$. These are false negatives because $\mathcal{S}_{\mathcal{A}}$ falsely marked these as non-solutions (i.e., negative).

2.7.2 Category of Filters and Relevant Observations

As has been mentioned before, our algorithm is based on some filtering techniques. In this subsection we discuss the filters and relevant observations which will be used throughout this thesis. Our filters although essentially are based on the same concept, are used in slightly different way for the different problems we handle.

- Observations for classical pattern (linear string) matching
- Observations for Exact Circular Pattern Matching (ECPM)/Circular Sequence Comparison (CSC)
- Observations for Approximate Circular Pattern Matching (ACPM)

Now, suppose we are given a pattern \mathcal{P} and a text \mathcal{T} . We will frequently and conveniently use the expression “ \mathcal{P} matches \mathcal{T} at position i ” to indicate \mathcal{P} linearly matches \mathcal{T} at position i in the context of the classical pattern matching problem. And we will frequently and conveniently use the expression “ $\mathcal{C}(\mathcal{P})$ matches \mathcal{T} at position i ” (or equivalently, “ \mathcal{P} circularly matches \mathcal{T} at position i ”) to indicate that one of the conjugates of \mathcal{P} matches \mathcal{T} at position i for CPM problem. Again, we will also frequently and conveniently use the expression “ $\mathcal{C}(\mathcal{P})$ k -matches

\mathcal{T} at position i ” (or equivalently, “ \mathcal{P} circularly k -matches \mathcal{T} at position i ”) to indicate that one of the conjugates of \mathcal{P} k -matches \mathcal{T} at position i (or equivalently, $\mathcal{C}(\mathcal{P}) \equiv_k \mathcal{T}$) for ACPM problem. We start with an brief overview of our approach in the subsections below.

2.7.3 Algorithmic Framework

In this framework, we first employ a number of filters to compute a set \mathcal{N} of indexes of \mathcal{T} such that $\mathcal{C}(\mathcal{P})$ matches \mathcal{T} at position $i \in \mathcal{N}$. As will be clear shortly, our filters are unable to compute the true set of indexes and hence \mathcal{N} may have false positives. However, our filters are designed in such a way that there are no false negatives. Hence, for all $j \notin \mathcal{N}$, we can be sure that there is **no** match. On the other hand, for all $i \in \mathcal{N}$, we **may or may not** have a match, i.e., we may have false positives. So, after we have computed \mathcal{N} , we compute \mathcal{T}' , a reduced version of \mathcal{T} concatenating all the factors $\mathcal{F}[i..i+m-1], i \in \mathcal{N}$ putting a special character $\$ \notin \Sigma$ in between the factors. One essential detail is as follows. There can be $i, j \in \mathcal{N}$ such that $1 < j - i < p$. In other words, there can exist overlapping factors matching with $\mathcal{C}(\mathcal{P})$. However, this can be handled easily through simple book-keeping as will be evident from our algorithm in later sections. Clearly, once we have computed the reduced text \mathcal{T}' we can employ any state of the art algorithm to solve problem (Problem 1 to Problem 4) on \mathcal{T}' to get the actual occurrences. So the most essential and useful feature of this framework is the application of filters to get a reduced text on which any existing algorithm can be applied to solve these problems (Problem 1 to Problem 4).

2.8 Filters of Classical Pattern Matching

In our algorithms, we employ a total of 6 filters (Table 2.4). In this section we describe these filters. We also discuss the related notions and notations needed to describe these filters. In what follows we describe our filters in the context of two strings of equal length n , namely, \mathcal{P} and \mathcal{T} , where the both are linear strings. We will devise and apply different functions on these strings and present observations related to these functions which in the sequel will lead us to our desired filter. The key to our observations and the resulting filters is the fact that each

function we devise results in a unique output when applied to the linear string.

Note that, the idea of these filters and the associated algorithmic framework originated at different research sessions arranged through the joint ongoing research collaborations² between King's College, London & BUET.

2.8.1 Filter 1

We define the function sum on a string \mathcal{P} of length m as follows: $sum(\mathcal{P}) = \sum_{i=1}^m P_N[i]$. Our first filter, Filter 1, is based on this sum function. We have the following observation.

Observation 1. *Consider a linear pattern \mathcal{P} and a linear string \mathcal{T} both having length n . If \mathcal{P} matches \mathcal{T} , then we must have $sum(\mathcal{P}) = sum(\mathcal{T})$.*

Example 9. *Consider $\mathcal{P} = atcgatg$ and $\mathcal{T} = atcgatg$. As can be easily verified, here \mathcal{P} linearly matches \mathcal{T} . Now we have $\mathcal{P}_N = 1423143 = \mathcal{T}_N$ and $sum(\mathcal{P}) = 18 = sum(\mathcal{T})$.*

Now consider another string $\mathcal{T}' = atagctg$, which is slightly different from \mathcal{T} . It can be easily verified that \mathcal{P} does not match \mathcal{T}' . Now, $\mathcal{T}'_N = 1413243$ and hence here as well we have $sum(\mathcal{T}') = 18 = sum(\mathcal{P})$. This is an example of a false positive with respect to Filter 1 in 2.8.1.

Again, consider another string $\mathcal{T}'' = atcgatt$, which is slightly different from \mathcal{T} . It can be easily verified that \mathcal{P} also does not match \mathcal{T}'' . Now, $\mathcal{T}''_N = 1423244$ and hence here we have $sum(\mathcal{T}'') = 20$ and $sum(\mathcal{P}) = 18$. This is an example of a false negative with respect to Filter 1 in 2.8.1.

2.8.2 Filters 2 and 3

Our second and third filters, i.e., Filters 2 and 3, depend on a notion of distance between consecutive characters of a string. The *distance* between two consecutive characters of a string \mathcal{P} of length m is defined by $distance(\mathcal{P}[i], \mathcal{P}[i+1]) = \mathcal{P}_N[i] - \mathcal{P}_N[i+1]$, where $1 \leq i \leq m-1$. We

²Part of this research has been supported by an INSPIRE Strategic Partnership Award, administered by the British Council, Bangladesh for the project titled "Advances in Algorithms for Next Generation Biological Sequences."

Table 2.4: A brief summary of the 6 filters employed in the our algorithms.

Filters	Descriptions
Filter 1	It is based on the sum function on the string.
Filter 2 & 3	These two filters are based on the distance between consecutive characters of the string. Filter 2 calculates only absolute distance whereas Filter 3 calculates actual distance.
Filter 4	It is based on the sum function like Filter 1 but in a slightly different way: here the sum function is applied for each individual character.
Filter 5	It is based on the modulo function on the string.
Filter 6	It is based on the bitwise exclusive-OR (XOR) function on the string.

define $total_distance(P) = \sum_{i=1}^{m-1} distance(\mathcal{P}[i], \mathcal{P}[i+1])$. We also define an absolute version of it: $abs_total_distance(P) = \sum_{i=1}^{m-1} abs(distance(\mathcal{P}[i], \mathcal{P}[i+1]))$, where $abs(x)$ returns the magnitude of x ignoring the sign.

Observation 2. Consider a linear pattern \mathcal{P} and a linear string \mathcal{T} both having length n . If \mathcal{P} matches \mathcal{T} , then, we must have $abs_total_distance(\mathcal{P}) = abs_total_distance(\mathcal{T})$.

Example 10. Consider the same two strings of Example 9, i.e., $\mathcal{P} = atc gatg$ $\mathcal{T} = atc gatg$. Here \mathcal{P} linear matches \mathcal{T} . We have $\mathcal{P}_N = 1423143 = \mathcal{T}_N$. Hence, $abs_total_distance(\mathcal{P}) = 12 = abs_total_distance(\mathcal{T})$.

Now consider another string $\mathcal{T}' = atagctg$ of the same length, which is slightly different from \mathcal{T} . It can easily be checked that \mathcal{P} doesn't match \mathcal{T}' . However, we find that $abs_total_distance(\mathcal{T}')$ is still 12. So, This is an example of a false positive with respect to Filter 2 in 2.8.2.

Again, consider another string $\mathcal{T}'' = atcgatt$, which is slightly different from \mathcal{T} . It can be easily verified that \mathcal{P} also does not match \mathcal{T}'' . Now, $\mathcal{T}''_N = 1423244$ and hence here we have $abs_total_distance(\mathcal{T}'') = 11$ and $abs_total_distance(\mathcal{P}) = 12$. This is an example of a false negative with respect to Filter 2 in 2.8.2.

Now we present the following related observation which is the basis of our Filter 3. Note that Observation 2 differs with Observation 3 only through using the absolute version of the

function used in the Section 2.8.2.

Observation 3. *Consider a linear pattern \mathcal{P} and a linear string \mathcal{T} both having length n . If \mathcal{P} linearly matches \mathcal{T} , then, we must have $\text{total_distance}(\mathcal{P}) = \text{total_distance}(\mathcal{T})$.*

Example 11. *Consider the same two strings of previous examples, Example 9, i.e., $\mathcal{P} = \text{atc gatg}$ $\mathcal{T} = \text{atc gatg}$. Here \mathcal{P} linearly matches \mathcal{T} . We have $\mathcal{P}_N = 1423143 = \mathcal{T}_N$. Hence, $\text{total_distance}(\mathcal{P}) = 1 = \text{total_distance}(\mathcal{T})$.*

Now consider another string $\mathcal{T}' = \text{ctc gatg}$ of the same length, which is slightly different from \mathcal{T} . It can easily be checked that \mathcal{P} doesn't match \mathcal{T}' . However, we find that $\text{total_distance}(\mathcal{T}')$ is still 1. So, This is an example of a false positive with respect to Filter 3 in 2.8.2.

Again, consider another string $\mathcal{T}'' = \text{atc gat}$, which is slightly different from \mathcal{T} . It can be easily verified that \mathcal{P} also does not match \mathcal{T}'' . Now, $\mathcal{T}''_N = 1423244$ and hence here we have $\text{total_distance}(\mathcal{T}'') = -3$ and $\text{total_distance}(\mathcal{P}) = 1$. This is an example of a false negative with respect to Filter 3 in 2.8.2.

2.8.3 Filter 4

Filter 4 uses the $\text{sum}()$ function used by Filter 1, albeit, in a slightly different way. In particular, it applies the $\text{sum}()$ function on individual characters. So, for $x \in \Sigma$ we define $\text{sum}_x(\mathcal{P}) = \sum_{1 \leq i \leq |\mathcal{P}|, \mathcal{P}[i]=x} P_N[i]$. Now we have the following observation.

Observation 4. *Consider a linear pattern \mathcal{P} and a linear string \mathcal{T} both having length n . If \mathcal{P} linearly matches \mathcal{T} , then, we must have $\text{sum}_x(\mathcal{P}) = \text{sum}_x(\mathcal{T})$ for all $x \in \Sigma$.*

Example 12. *Consider the same two strings of previous examples, Example 9, i.e., $\mathcal{P} = \text{atc gatg}$ $\mathcal{T} = \text{atc gatg}$. Recall that \mathcal{P} linearly matches \mathcal{T} . It is easy to calculate that $\text{sum}_a(\mathcal{T}) = 2$, $\text{sum}_c(\mathcal{T}) = 2$, $\text{sum}_g(\mathcal{T}) = 6$ and $\text{sum}_t(\mathcal{T}) = 8$. Hence according to Observation 4, the individual sum values for the \mathcal{P} must also match this. It can be easily verified that this is indeed the case.*

Now consider the other string $\mathcal{T}' = \text{atagctg}$ of the same length, which is slightly different from \mathcal{T} . It can easily be checked that \mathcal{P} doesn't match \mathcal{T}' . However, as we can see, still we

have $sum_a(\mathcal{T}') = 2$, $sum_c(\mathcal{T}') = 2$, $sum_g(\mathcal{T}') = 6$ and $sum_t(\mathcal{T}') = 8$. This is an example of a false positive with respect to Filter 4 in 2.8.3.

Again, consider another string $\mathcal{T}'' = atcgatt$, which is slightly different from \mathcal{T} . It can easily be checked that \mathcal{P} also doesn't match \mathcal{T}'' . However, as we can see that $sum_a(\mathcal{T}'') = 2$, $sum_c(\mathcal{T}'') = 2$, $sum_g(\mathcal{T}'') = 3$ and $sum_t(\mathcal{T}'') = 12$. This is an example of a false negative with respect to Filter 4 in 2.8.3.

2.8.4 Filter 5

Filter 5 depends on modulo operation between two consecutive characters. A modulo operation between two consecutive characters of a string \mathcal{P} of length m is defined as follows: $modulo(\mathcal{P}[i], \mathcal{P}[i+1]) = \mathcal{P}_N[i] \% \mathcal{P}_N[i+1]$, where $1 \leq i \leq m-1$. We define $sum_modulo(\mathcal{P})$ to be the summation of the results of the modulo operations on the consecutive characters of \mathcal{P} . More formally, $sum_modulo(\mathcal{P}) = \sum_{i=1}^{m-1} modulo(\mathcal{P}[i], \mathcal{P}[i+1])$. Now we present the following observation which is the basis of Filter 5.

Observation 5. Consider a linear pattern \mathcal{P} and a linear string \mathcal{T} both having length n . If \mathcal{P} linearly matches \mathcal{T} , then, we must have $sum_modulo(\mathcal{P}) = sum_modulo(\mathcal{T})$.

Example 13. Consider the same two strings of previous examples, Example 9, i.e., $\mathcal{P} = atcgatg$ $\mathcal{T} = atcgatg$. Recall that \mathcal{P} linearly matches \mathcal{T} . We have $\mathcal{P}_N = 1423143 = \mathcal{T}_N$. Hence, $sum_modulo(\mathcal{P}) = 5 = sum_modulo(\mathcal{T})$.

Now consider another string $\mathcal{T}' = ctgatg$ of the same length, which is different from \mathcal{T} . It can easily be checked that \mathcal{P} doesn't match \mathcal{T}' . However, we find that $sum_modulo(\mathcal{T}')$ is still 5. So, This is an example of a false positive with respect to Filter 5 in 2.8.4.

Again, consider another string $\mathcal{T}'' = tgatgc$, which is slightly different from \mathcal{T} . It can be easily verified that \mathcal{P} also does not match \mathcal{T}'' . Now, $\mathcal{T}''_N = 4313142$ and hence here we have $sum_modulo(\mathcal{T}'') = 3$ and $sum_modulo(\mathcal{P}) = 5$. This is an example of a false negative with respect to Filter 5 in 2.8.4.

2.8.5 Filter 6

In Filter 6 we employ the $xor()$ operation. A bitwise exclusive-OR ($xor()$) operation between two consecutive characters of a string \mathcal{P} of length m is defined as follows: $xor(\mathcal{P}[i], \mathcal{P}[i+1]) = \mathcal{P}_N[i] \wedge \mathcal{P}_N[i+1]$, where $1 \leq i \leq m-1$. We define $sum_xor(\mathcal{P})$ to be the summation of the results of the xor operations on the consecutive characters of \mathcal{P} . More formally, $sum_xor(\mathcal{P}) = \sum_{i=1}^{m-1} xor(\mathcal{P}[i], \mathcal{P}[i+1])$. Now we present the following observation which is the basis of Filter 6.

Observation 6. *Consider a linear pattern \mathcal{P} and a linear string \mathcal{T} both having length n . If \mathcal{P} linearly matches \mathcal{T} , then, we must have $sum_xor(\mathcal{P}) = sum_xor(\mathcal{T})$.*

Example 14. *Consider two linear strings of same length, i.e., $\mathcal{P} = atc gatg$ $\mathcal{T} = atc gatg$. As can be easily verified, here \mathcal{P} linearly matches \mathcal{T} . We have $\mathcal{P}_N = 1423143 = \mathcal{T}_N$. Hence, $sum_xor(\mathcal{P}) = 26 = sum_xor(\mathcal{T})$.*

Now consider another string $\mathcal{T}' = gtagata$ of the same length, which is different from \mathcal{T} . It can easily be checked that \mathcal{P} doesn't match \mathcal{T}' . However, we find that $sum_xor(\mathcal{T}')$ is still 26. So, This is an example of a false positive with respect to Filter 6 in 2.8.5.

Again, consider another string $\mathcal{T}'' = tgagatc$, which is slightly different from \mathcal{T} . It can be easily verified that \mathcal{P} also does not match \mathcal{T}'' . Now, $\mathcal{T}''_N = 4313142$ and hence here we have $sum_xor(\mathcal{T}'') = 24$ and $sum_xor(\mathcal{P}) = 26$. This is an example of a false negative with respect to Filter 6 in 2.8.5.

2.9 Filters of Circular Pattern Matching and Circular Sequence Comparison

In this section we adopt the 6 filters described in Section 2.8 for circular string to solve the problems ECPM and CSC problems. In particular we will be extending the filters used in Section 2.8 to make it useful and effective in the context of Circular Pattern Matching Problem and Circular Sequence Comparison. In what follows, we follow the notations described in Section 2.8. Recall to Section 2.8, the key to our observations and the resulting filters is the

fact that each function we devise results in a unique output when applied to the rotations of a circular string. For example, consider a hypothetical function \mathcal{X} . We will always have the relation that $\mathcal{X}(\mathcal{P}) = \mathcal{X}(\mathcal{P}^i)$ for all $1 \leq i < n$. Recall that, \mathcal{P}^0 actually denotes \mathcal{P} . For the sake of conciseness, for such functions, we will abuse the notation a bit and use $\mathcal{X}(\mathcal{C}(\mathcal{P}))$ to represent $\mathcal{X}(\mathcal{P}^i)$ for all $0 \leq i < |\mathcal{P}|$.

2.9.1 Filter 1

Following the notation described in Section 2.8.1, we have the following observations for circular version of Filter 1.

Observation 7. *Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n . If $\mathcal{C}(\mathcal{P})$ matches \mathcal{T} , then we must have $\text{sum}(\mathcal{C}(\mathcal{P})) = \text{sum}(\mathcal{T})$.*

Example 15. *Consider $\mathcal{P} = \text{atcgatg}$ $\mathcal{T} = \text{tgatcga}$. As can be easily verified, here \mathcal{P} circularly matches \mathcal{T} . In fact the match is due to the conjugate \mathcal{P}^5 . Now we have $\mathcal{T}_N = 4314231$ and $\text{sum}(\mathcal{T}) = 18$. Then, according to Observation 7, we must have $\text{sum}(\mathcal{C}(\mathcal{P})) = 18$. This can indeed be verified easily.*

Now consider another string $\mathcal{T}' = \text{atagctg}$, which is slightly different from \mathcal{T} . It can be easily verified that $\mathcal{C}(\mathcal{P})$ does not match \mathcal{T}' . Now, $\mathcal{T}'_N = 1413243$ and hence here as well we have $\text{sum}(\mathcal{T}') = 18 = \text{sum}(\mathcal{C}(\mathcal{P}))$. This is an example of a false positive with respect to Filter 1 in 2.9.1.

Again, consider another string $\mathcal{T}'' = \text{atcgatt}$, which is slightly different from \mathcal{T} . It can be easily verified that \mathcal{P} also does not match \mathcal{T}'' . Now, $\mathcal{T}''_N = 1423244$ and hence here we have $\text{sum}(\mathcal{T}'') = 20$ and $\text{sum}(\mathcal{P}) = 18$. This is an example of a false negative with respect to Filter 1 in 2.9.1.

2.9.2 Filters 2 and 3

Following the notation described in Section 2.8.2, we introduce observations for circular version of Filter 2 and Filter 3. Before we apply these two functions on our strings to get our filters, we need to do a simple pre-processing on the respective string, i.e., \mathcal{P} in this case as follows.

We extend the string \mathcal{P} by concatenating the first character of \mathcal{P} at its end. We use $ext(\mathcal{P})$ to denote the resultant string. So, we have $ext(\mathcal{P}) = \mathcal{P}\mathcal{P}[1]$. Since, $ext(\mathcal{P})$ can simply be treated as another string, we can easily extend the notation and concept of $\mathcal{C}(\mathcal{P})$ over $ext(\mathcal{P})$.

Now we have the following observation which is the basis of our Filter 2.

Observation 8. *Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. If $\mathcal{C}(\mathcal{P})$ matches \mathcal{T} , then, we must have $abs_total_distance(\mathcal{C}(\mathcal{A})) = abs_total_distance(\mathcal{B})$. Note carefully that the function $abs_total_distance()$ has been applied on the extended strings.*

Example 16. *Consider the same two strings of Example 15, i.e., $\mathcal{P} = atcgatg$ $\mathcal{T} = tgatcga$. Here \mathcal{P} circularly matches \mathcal{T} (due to the conjugate \mathcal{P}^5). Now consider the extended strings and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. We have $\mathcal{T}_N = 4314231$. Hence $\mathcal{B}_N = 43142314$. Hence, $abs_total_distance(\mathcal{B}) = 14$. It can be easily verified that $abs_total_distance(\mathcal{C}(\mathcal{A}))$ is also 14.*

Now consider another string $\mathcal{T}' = atagctg$ of the same length, which is slightly different from \mathcal{T} . It can easily be checked that $\mathcal{C}(\mathcal{P})$ doesn't match \mathcal{T}' . However, assuming that $\mathcal{B}' = ext(\mathcal{T}')$ we find that $abs_total_distance(\mathcal{B}')$ is still 14. So, This is an example of a false positive with respect to Filter 2 in 2.9.2.

Again, consider another string $\mathcal{T}'' = atggatg$, which is slightly different from \mathcal{T} . It can be easily checked that $\mathcal{C}(\mathcal{P})$ also does not match \mathcal{T}'' . However, assuming that $\mathcal{B}'' = ext(\mathcal{T}'')$ we find that $abs_total_distance(\mathcal{B}'')$ = 12. This is an example of a false negative with respect to Filter 2 in 2.9.2.

Now we present the following related observation which is the basis of our Filter 3. Note that Observations 8 differs with Observation 9 only through using the absolute version of the function used in the latter.

Observation 9. *Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. If $\mathcal{C}(\mathcal{P})$ matches \mathcal{T} , then, we must have $total_distance(\mathcal{C}(\mathcal{A})) = total_distance(\mathcal{B})$. Note carefully that the function $total_distance()$ has been applied on the extended strings.*

Example 17. Consider the same two strings of previous examples, Example 15, i.e., $\mathcal{P} = \text{atcgatg}$ $\mathcal{T} = \text{tgatcga}$. Here \mathcal{P} circularly matches \mathcal{T} (due to the conjugate \mathcal{P}^5). Now consider the extended strings and assume that $\mathcal{A} = \text{ext}(\mathcal{P})$ and $\mathcal{B} = \text{ext}(\mathcal{T})$. We have $\mathcal{T}_N = 4314231$. Hence $\mathcal{B}_N = 43142314$. Hence, $\text{total_distance}(\mathcal{B}) = 0$. It can be easily verified that $\text{total_distance}(\mathcal{C}(\mathcal{A}))$ is also 0.

Now consider another string $\mathcal{T}' = \text{atagctg}$ of the same length, which is slightly different from \mathcal{T} . It can easily be checked that $\mathcal{C}(\mathcal{P})$ doesn't match \mathcal{T}' . However, assuming that $\mathcal{B}' = \text{ext}(\mathcal{T}')$ we find that $\text{total_distance}(\mathcal{B}')$ is still 0. So, This is an example of a false positive with respect to Filter 3 in 2.9.2.

Again, consider another string $\mathcal{T}'' = \text{ttatcga}$, which is slightly different from \mathcal{T} . It can be easily checked that $\mathcal{C}(\mathcal{P})$ also does not match \mathcal{T}'' . However, assuming that $\mathcal{B}'' = \text{ext}(\mathcal{T}'')$ we find that $\text{total_distance}(\mathcal{B}'') = -4$. This is an example of a false negative with respect to Filter 3 in 2.9.2.

2.9.3 Filter 4

Following the notation described in Section 2.8.3, we have the following observations for circular version of Filter 4.

Observation 10. Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n . If $\mathcal{C}(\mathcal{P})$ matches \mathcal{T} , then, we must have $\text{sum}_x(\mathcal{C}(\mathcal{P})) = \text{sum}_x(\mathcal{T})$ for all $x \in \Sigma$.

Example 18. Consider the same two strings of previous examples, Example 15, i.e., $\mathcal{P} = \text{atcgatg}$ $\mathcal{T} = \text{tgatcga}$. Recall that \mathcal{P} circularly matches \mathcal{T} (due to the conjugate \mathcal{P}^5). It is easy to calculate that $\text{sum}_a(\mathcal{T}) = 2$, $\text{sum}_c(\mathcal{T}) = 2$, $\text{sum}_g(\mathcal{T}) = 6$ and $\text{sum}_t(\mathcal{T}) = 8$. Hence according to Observation 10, the individual sum values for all the conjugates of \mathcal{P} must also match this. It can be easily verified that this is indeed the case.

Now consider the other string $\mathcal{T}' = \text{atagctg}$ of the same length, which is slightly different from \mathcal{T} . It can easily be checked that $\mathcal{C}(\mathcal{P})$ doesn't match \mathcal{T}' . However, as we can see, still we have $\text{sum}_a(\mathcal{T}') = 2$, $\text{sum}_c(\mathcal{T}') = 2$, $\text{sum}_g(\mathcal{T}') = 6$ and $\text{sum}_t(\mathcal{T}') = 8$. This is an example of a false positive with respect to Filter 4 in 2.9.3.

Again, consider another string $\mathcal{T}'' = atggatg$, which is slightly different from \mathcal{T} . It can be easily checked that $\mathcal{C}(\mathcal{P})$ also does not match \mathcal{T}'' . Hence, as we can see, $sum_a(\mathcal{T}'') = 1$, $sum_c(\mathcal{T}'') = 0$, $sum_g(\mathcal{T}'') = 9$ and $sum_t(\mathcal{T}'') = 8$. This is an example of a false negative with respect to Filter 4 in 2.9.3.

2.9.4 Filter 5

Following the notation described in Section 2.8.4, we have the following observations for linear version of Filter 5. Note that this observation is applied on the extended versions of the respective strings.

Observation 11. Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. If $\mathcal{C}(\mathcal{P})$ matches \mathcal{T} , then, we must have $sum_modulo(\mathcal{C}(\mathcal{A})) = sum_modulo(\mathcal{B})$. Note carefully that the function $sum_modulo()$ has been applied on the extended strings.

Example 19. Consider the same two strings of previous examples, Example 15, i.e., $\mathcal{P} = atcgatg$ $\mathcal{T} = tgatega$. Recall that \mathcal{P} circularly matches \mathcal{T} (due to the conjugate \mathcal{P}^5). Now consider the extended strings and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. We have $\mathcal{T}_N = 4314231$. Hence $\mathcal{B}_N = 43142314$. Hence, $sum_modulo(\mathcal{B}) = 5$. Now according to Observation 11, we must also have $sum_modulo(\mathcal{C}(\mathcal{A})) = 5$. This is indeed true.

Now consider another string $\mathcal{T}' = tgatega$ of the same length, which is different from \mathcal{T} . It can easily be checked that $\mathcal{C}(\mathcal{P})$ doesn't match \mathcal{T}' . However, assuming that $\mathcal{B}' = ext(\mathcal{T}')$ we find that $sum_modulo(\mathcal{B}')$ is still 5. So, This is an example of a false positive with respect to Filter 5 in 2.9.4.

Again, consider another string $\mathcal{T}'' = ttatcga$, which is slightly different from \mathcal{T} . It can be easily checked that $\mathcal{C}(\mathcal{P})$ also does not match \mathcal{T}'' . However, assuming that $\mathcal{B}'' = ext(\mathcal{T}'')$ we find that $sum_modulo(\mathcal{B}'') = 4$. This is an example of a false negative with respect to Filter 5 in 2.9.4.

2.9.5 Filter 6

Following the notation described in Section 2.8.5, we have the following observations for linear version of Filter 6. Note that this observation is applied on the extended versions of the respective strings.

Observation 12. *Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n and assume that $\mathcal{A} = \text{ext}(\mathcal{P})$ and $\mathcal{B} = \text{ext}(\mathcal{T})$. If $\mathcal{C}(\mathcal{P})$ matches \mathcal{T} , then, we must have $\text{sum_xor}(\mathcal{C}(\mathcal{A})) = \text{sum_xor}(\mathcal{B})$. Note carefully that the function $\text{sum_xor}()$ has been applied on the extended strings.*

Example 20. *Consider the same two strings of previous examples, i.e., $\mathcal{P} = \text{atcgatg}$ $\mathcal{T} = \text{tgatcga}$. Recall that \mathcal{P} circularly matches \mathcal{T} (due to the conjugate \mathcal{P}^5). Now consider the extended strings and assume that $\mathcal{A} = \text{ext}(\mathcal{P})$ and $\mathcal{B} = \text{ext}(\mathcal{T})$. We have $\mathcal{T}_N = 4314231$. Hence $\mathcal{B}_N = 43142314$. Hence, $\text{sum_xor}(\mathcal{B}) = 28$. Now according to Observation 12, we must also have $\text{sum_xor}(\mathcal{C}(\mathcal{A})) = 28$. As can be verified easily, this is indeed the case.*

Now consider another string $\mathcal{T}' = \text{gtagatc}$ of the same length, which is different from \mathcal{T} . It can easily be checked that $\mathcal{C}(\mathcal{P})$ doesn't match \mathcal{T}' . However, assuming that $\mathcal{B}' = \text{ext}(\mathcal{T}')$ we find that $\text{sum_xor}(\mathcal{B}')$ is still 28. So, This is an example of a false positive with respect to Filter 6 in 2.9.5.

Again, consider another string $\mathcal{T}'' = \text{ttatcga}$, which is slightly different from \mathcal{T} . It can be easily checked that $\mathcal{C}(\mathcal{P})$ also does not match \mathcal{T}'' . However, assuming that $\mathcal{B}'' = \text{ext}(\mathcal{T}'')$ we find that $\text{sum_xor}(\mathcal{B}'') = 24$. This is an example of a false negative with respect to Filter 6 in 2.9.5.

2.10 Filters of ACPM

In this section we adopt the 6 filters described in Section 2.8 for approximate version of circular pattern matching. In particular we will be extending the filters used in Section 2.8 to make it useful and effective in the context of approximate circular pattern matching. In what follows, we follow the notations described in Section 2.8.

2.10.1 Filter 1

Following the notation described in Section 2.8.1, we have the following observations for approximate version of Filter 1.

Observation 13. *Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n . If $\mathcal{C}(\mathcal{P}) \equiv_k \mathcal{T}$, where $0 \leq k < n$, then we must have*

$$\text{sum}(\mathcal{T}) - k \times 4 + k \times 1 \leq \text{sum}(\mathcal{C}(\mathcal{P})) \leq \text{sum}(\mathcal{T}) + k \times 4 - k \times 1.$$

Reason behind the fixed values (1 and 4) used in the above filter (Observation 13): It has been seen that, for k -mismatches (i.e mutations), the k number of ‘A’ character can be replaced by k number of ‘T’ character from the circular string, at most. In this way, we can get the upper bound. Again, if the k number of ‘T’ character can be replaced by k number of ‘A’ character from the circular string, at least, then we get the lower bound. This is due to the minimum numeric number, $\text{num}(a) = 1$ corresponds to character ‘A’ and the maximum numeric number, $\text{num}(t) = 4$ corresponds to character ‘T’.

Example 21. *Consider $\mathcal{P} = \text{atcgatg}$. We can easily calculate that $\text{sum}(\mathcal{C}(\mathcal{P})) = 18$. Now, consider $\mathcal{T}1 = \text{aacgatg}$, slightly different from \mathcal{P} , i.e, $\mathcal{P}[2] = t \neq \mathcal{T}1[2] = a$. As can be easily verified, here $\mathcal{P} \equiv_1 \mathcal{T}1$. According to Observation 13, in this case the lower (upper) bound is 15 (18). Indeed, we have $\mathcal{T}1_N = 1123143$ and $\text{sum}(\mathcal{T}1) = 15$, which is within the bounds. Now consider $\mathcal{T}2 = \text{ttcgatg}$, slightly different from \mathcal{P} , i.e, $\mathcal{P}[1] = a \neq \mathcal{T}2[1] = t$. As can be easily verified, here $\mathcal{P} \equiv_1 \mathcal{T}2$. Therefore, in this case as well, the lower and upper bound mentioned above hold. And indeed we have $\mathcal{T}2_N = 4423143$ and $\text{sum}(\mathcal{T}2) = 21$, which is within the bounds. Finally, consider another string $\mathcal{T}' = \text{atagctg}$. It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}'$. Again, the previous bounds hold in this case and we find that $\mathcal{T}'_N = 1413243$ and $\text{sum}(\mathcal{T}') = 18$. Clearly this is within the bounds of Observation 13 and in fact it is exactly equal to $\text{sum}(\mathcal{C}(\mathcal{P}))$. This is an example of a false positive with respect to Filter 1 in 2.10.1.*

Again, consider another string $\mathcal{T}'' = \text{atcgatt}$, which is slightly different from \mathcal{T} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}''$. Now, $\mathcal{T}''_N = 1423244$ and hence here we have $\text{sum}(\mathcal{T}'') = 20$.

Clearly this is not within the bounds of Observation 13. This is an example of a false negative with respect to Filter 1 in 2.10.1.

2.10.2 Filters 2 and 3

Following the notation described in Section 2.8.2, we have the following observations for approximate version of Filter 2 and Filter 3. Note that these two observations are applied on the extended versions of the respective strings.

Observation 14. Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n and assume that $\mathcal{A} = \text{ext}(\mathcal{P})$ and $\mathcal{B} = \text{ext}(\mathcal{T})$. If $\mathcal{C}(\mathcal{P}) \equiv_k \mathcal{T}$, where $0 \leq k < n$, then we must have

$$\text{abs_total_distance}(\mathcal{B}) - k \times 4 + k \times 1 \leq$$

$$\text{abs_total_distance}(\mathcal{C}(\mathcal{A})) \leq$$

$$\text{abs_total_distance}(\mathcal{B}) + k \times 4 - k \times 1.$$

Reason behind the fixed values (1 and 4) used in the above filter (Observation 14): Same reason described for Filter 13.

Example 22. Consider the same strings of Example 21, i.e., $\mathcal{P} = \text{atcgatg}$, $\mathcal{T}1 = \text{aacgatg}$ and $\mathcal{T}2 = \text{ttcgatg}$. As can be easily verified, here $\mathcal{P} \equiv_1 \mathcal{T}1$ and $\mathcal{P} \equiv_1 \mathcal{T}2$. Now consider the extended strings and assume that $\mathcal{A} = \text{ext}(\mathcal{P})$, $\mathcal{B}1 = \text{ext}(\mathcal{T}1)$ and $\mathcal{B}2 = \text{ext}(\mathcal{T}2)$. It can be easily verified that $\text{abs_total_distance}(\mathcal{C}(\mathcal{A}))$ is 14. Recall that $\mathcal{T}1$ is slightly different from \mathcal{P} , i.e., $\mathcal{P}[2] = t \neq \mathcal{T}1[2] = a$. Now we have $\mathcal{T}1_N = 1123143$. Hence $\mathcal{B}1_N = 11231431$. Hence, $\text{abs_total_distance}(\mathcal{B}1) = 10$ which is indeed within the bounds of Observation 14. Now consider $\mathcal{T}2$, which is slightly different from \mathcal{P} , i.e., $\mathcal{P}[1] = a \neq \mathcal{T}2[1] = t$. Now we have $\mathcal{T}2_N = 4423143$. Hence $\mathcal{B}2_N = 44231434$. Hence, $\text{abs_total_distance}(\mathcal{B}2) = 10$, which is also within the bounds. Again, consider $\mathcal{T}' = \text{atagctg}$, which is again slightly different from \mathcal{P} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}'$. However, assuming that $\mathcal{B}' = \text{ext}(\mathcal{T}')$ we find that $\text{abs_total_distance}(\mathcal{B}')$ is still 14, which is in the range of Observation 14. This is an example of a false positive with respect to Filter 2 in 2.10.2.

Finally, consider another string $\mathcal{T}'' = aaaaaaa$, which is totally different from \mathcal{T} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}''$. Now, $\mathcal{T}''_N = 1111111$ and hence assuming that $\mathcal{B}'' = \text{ext}(\mathcal{T}'')$ we find that $\text{abs_total_distance}(\mathcal{B}'') = 0$. Clearly this is not within the bounds of Observation 14. This is an example of a false negative with respect to Filter 2 in 2.10.2.

Now we present the following related observation which is the basis of our Filter 3. Note that Observation 14 differs with Observation 15 only through using the absolute version of the function used in the Section 2.8.2. Note that this observation is applied on the extended versions of the respective strings.

Observation 15. Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n and assume that $\mathcal{A} = \text{ext}(\mathcal{P})$ and $\mathcal{B} = \text{ext}(\mathcal{T})$. If $\mathcal{C}(\mathcal{P}) \equiv_k \mathcal{T}$, where $0 \leq k < n$, then we must have

$$\text{total_distance}(\mathcal{B}) - k \times 4 + k \times 1 \leq \text{total_distance}(\mathcal{C}(\mathcal{A}))$$

$$\leq \text{total_distance}(\mathcal{B}) + k \times 4 - k \times 1.$$

Reason behind the fixed values (1 and 4) used in the above filter (Observation 15): Same reason described for Filter 13.

Example 23. Consider the same strings of Example 21, i.e., $\mathcal{P} = \text{atcgatg}$, $\mathcal{T}1 = \text{aacgatg}$ and $\mathcal{T}2 = \text{ttcgatg}$. As can be easily verified, here $\mathcal{P} \equiv_1 \mathcal{T}1$ and $\mathcal{P} \equiv_1 \mathcal{T}2$. Now consider the extended strings and assume that $\mathcal{A} = \text{ext}(\mathcal{P})$, $\mathcal{B}1 = \text{ext}(\mathcal{T}1)$ and $\mathcal{B}2 = \text{ext}(\mathcal{T}2)$. It can be easily verified that $\text{abs_total_distance}(\mathcal{C}(\mathcal{A}))$ is 14. Recall that $\mathcal{T}1$ is slightly different from \mathcal{P} , i.e., $\mathcal{P}[2] = t \neq \mathcal{T}1[2] = a$. Now we have $\mathcal{T}1_N = 1123143$. Hence $\mathcal{B}1_N = 11231431$. Hence, $\text{total_distance}(\mathcal{B}1) = 0$ which is indeed within the bounds of Observation 14. Now consider $\mathcal{T}2$, which is slightly different from \mathcal{P} , i.e., $\mathcal{P}[1] = a \neq \mathcal{T}2[1] = t$. Now we have $\mathcal{T}2_N = 4423143$. Hence $\mathcal{B}2_N = 44231434$. Hence, $\text{total_distance}(\mathcal{B}2) = 10$, which is also within the bounds. Again, consider $\mathcal{T}' = \text{atagctg}$, which is again slightly different from \mathcal{P} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}'$. However, assuming that $\mathcal{B}' = \text{ext}(\mathcal{T}')$ we find that $\text{total_distance}(\mathcal{B}')$ is still 0, which is in the range of Observation 15. This is an example of a false positive with respect to Filter 3 in 2.10.2.

Finally, consider another string $\mathcal{T}'' = aaaaaaa$, which is totally different from \mathcal{T} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}''$. Now, $\mathcal{T}''_N = 1111111$ and hence assuming that $\mathcal{B}'' = \text{ext}(\mathcal{T}'')$ we find that $\text{abs_total_distance}(\mathcal{B}'') = 0$. Clearly this is not within the bounds of Observation 15. This is an example of a false negative with respect to Filter 3 in 2.10.2.

2.10.3 Filters 4

Following the notation described in Section 2.8.3, we have the following observations for approximate version of Filter 4.

Observation 16. Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n . If $\mathcal{C}(\mathcal{P}) \equiv_k \mathcal{T}$, where $0 \leq k < n$, then we must have

$$\begin{aligned} \text{sum}_x(\mathcal{T}) - k \times \text{num}(x) &\leq \text{sum}_x(\mathcal{C}(\mathcal{P})) \\ &\leq \text{sum}_x(\mathcal{T}) + k \times \text{num}(x) \end{aligned}$$

for all $x \in \Sigma$.

Reason behind the fixed values (1 and 4) used in the above filter (Observation 16): Same reason described for Filter 13.

Example 24. Consider the same strings of Example 22, i.e., $\mathcal{P} = \text{atcgatg}$, $\mathcal{T}1 = \text{aacgatg}$ and $\mathcal{T}2 = \text{ttcgatg}$. Recall that here $\mathcal{P} \equiv_1 \mathcal{T}1$ and $\mathcal{P} \equiv_1 \mathcal{T}2$. Now, as has been described in Example 22 that $\mathcal{T}1$ is slightly different from \mathcal{P} , i.e., $\mathcal{P}[2] = t \neq \mathcal{T}1[2] = a$. Now we have $\mathcal{T}1_N = 1123143$. Hence, $\text{sum}_a(\mathcal{T}1) = 3$, $\text{sum}_c(\mathcal{T}1) = 2$, $\text{sum}_g(\mathcal{T}1) = 6$ and $\text{sum}_t(\mathcal{T}1) = 4$. According to Observation 16, in this case the lower (upper) bound for character 'A' is 2 (3). And the lower (upper) bound for character 'T' is 4 (8), which is in the bounds. Now consider that $\mathcal{T}2$ which is slightly from \mathcal{P} , i.e. $\mathcal{P}[1] = a \neq \mathcal{T}2[1] = t$. Now we have $\mathcal{T}2_N = 4423143$. Hence, $\text{sum}_a(\mathcal{T}2) = 1$, $\text{sum}_c(\mathcal{T}2) = 2$, $\text{sum}_g(\mathcal{T}2) = 6$ and $\text{sum}_t(\mathcal{T}2) = 12$. Here, in this case the lower (upper) bound for character 'A' is 1 (2). And the lower (upper) bound for character 'T' is 8 (12), which is also in the bounds. Therefore, we can summarize that we have got

a lower bound and an upper bound according to Observation 16 for these two characters ‘A’ and ‘T’, others are unchanged. For this example the Observation 16 shows the overall lower (upper) bound for character ‘A’ is 1 (3) and the lower (upper) bound for character ‘T’ is 4 (12). Again, consider $\mathcal{T}' = \text{atagctg}$, which is again slightly different from \mathcal{P} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}'$. Now, $\mathcal{T}'_N = 1413243$. However, as we can still find that, $\text{sum}_a(\mathcal{T}') = 2$, $\text{sum}_c(\mathcal{T}') = 2$, $\text{sum}_g(\mathcal{T}') = 6$ and $\text{sum}_t(\mathcal{T}') = 8$, which is in the bounds of Observation 16. This is an example of a false positive with respect to Filter 4 in 2.10.3. Finally, consider another string $\mathcal{T}'' = \text{aaaaaaa}$, which is totally different from \mathcal{T} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}''$. Hence, as we can see, $\text{sum}_a(\mathcal{T}'') = 7$, $\text{sum}_c(\mathcal{T}'') = 0$, $\text{sum}_g(\mathcal{T}'') = 0$ and $\text{sum}_t(\mathcal{T}'') = 0$. Clearly this is not within the bounds of Observation 16. This is an example of a false negative with respect to Filter 4 in 2.9.3.

2.10.4 Filters 5

Following the notation described in Section 2.8.4, we have the following observations for approximate version of Filter 5. Note that this observation is applied on the extended versions of the respective strings.

Observation 17. Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n and assume that $\mathcal{A} = \text{ext}(\mathcal{P})$ and $\mathcal{B} = \text{ext}(\mathcal{T})$. If $\mathcal{C}(\mathcal{P}) \equiv_k \mathcal{T}$, where $0 \leq k < n$, then, we must have

$$\begin{aligned} \text{sum_modulo}(\mathcal{B}) - k \times 4 + k \times 0 &\leq \text{sum_modulo}(\mathcal{C}(\mathcal{A})) \\ &\leq \text{sum_modulo}(\mathcal{B}) + k \times 4 - k \times 0. \end{aligned}$$

Note carefully that the function $\text{sum_modulo}()$ has been applied on the extended strings.

Reason behind the fixed values (0 and 4) used in the above filter (Observation 17): As we are doing modulo operation for the Observation 5, the replacement of a single character (any of Σ) with another character (also taken from Σ) may change the sum of modulo operation in a way that we can have a upper bound. This upper bound for a single character is 4. In this way, we can get the upper bound for k mismatches. Again, for the lower

bound, it has been seen that, for a single character change the minimum change in the sum of the modulo operation could be 0. By this way, we can get the lower bound for k mismatches.

Example 25. Consider the same four strings of Example 22, i.e., $\mathcal{P} = atc gatg$, $\mathcal{T}1 = aac gatg$ and $\mathcal{T}2 = ttc gatg$. As can be easily verified, here $\mathcal{P} \equiv_1 \mathcal{T}1$ and $\mathcal{P} \equiv_1 \mathcal{T}2$. Now consider the extended strings and assume that $\mathcal{A} = ext(\mathcal{P})$, $\mathcal{B}1 = ext(\mathcal{T}1)$ and $\mathcal{B}2 = ext(\mathcal{T}2)$. It can be easily verified that $sum_modulo(\mathcal{C}(\mathcal{A}))$ is 5. Recall that $\mathcal{T}1$ is slightly different from \mathcal{P} , i.e., $\mathcal{P}[2] = t \neq \mathcal{T}1[2] = a$. Now we have $\mathcal{T}1_N = 1123143$. Hence $\mathcal{B}1_N = 11231431$. Hence, $sum_modulo(\mathcal{B}1) = 5$ which is indeed within the bounds of Observation 17. Now consider $\mathcal{T}2$, which is slightly different from \mathcal{P} , i.e., $\mathcal{P}[1] = a \neq \mathcal{T}2[1] = t$. Now we have $\mathcal{T}2_N = 4423143$. Hence $\mathcal{B}2_N = 44231434$. Hence, $sum_modulo(\mathcal{B}2) = 7$, which is also within the bounds. Again, consider $\mathcal{T}' = atagctg$, which is again slightly different from \mathcal{P} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}'$. However, assuming that $\mathcal{B}' = ext(\mathcal{T}')$ we find that $sum_modulo(\mathcal{B}')$ is still 5, which is in the range of Observation 17. This is an example of a false positive with respect to Filter 5 in 2.10.4.

Finally, consider another string $\mathcal{T}'' = aaaaaaa$, which is totally different from \mathcal{T} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}''$. Now, $\mathcal{T}''_N = 1111111$ and hence assuming that $\mathcal{B}'' = ext(\mathcal{T}'')$ we find that $sum_modulo(\mathcal{B}'') = 0$. Clearly this is not within the bounds of Observation 17. This is an example of a false negative with respect to Filter 5 in 2.10.4.

2.10.5 Filters 6

Following the notation described in Section 2.8.5, we have the following observations for approximate version of Filter 6. Note that this observation is applied on the extended versions of the respective strings.

Observation 18. Consider a circular string \mathcal{P} and a linear string \mathcal{T} both having length n and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. If $\mathcal{C}(\mathcal{P}) \equiv_k \mathcal{T}$, then, we must have

$$sum_xor(\mathcal{B}) - k \times 14 + k \times 0 \leq sum_xor(\mathcal{C}(\mathcal{A}))$$

$$\leq \text{sum_xor}(\mathcal{B}) + k \times 14 + k \times 0.$$

Note carefully that the function $\text{sum_xor}()$ has been applied on the extended strings.

Reason behind the fixed values (0 and 14) used in the above filter (Observation 18): As we are doing xor operation for the Observation 6, the replacement of a single character (any of Σ) with another character (also taken from Σ) may change the sum of xor operation in a way that we can have an upper bound. This upper bound for a single character is 14. In this way, we can get the upper bound for k mismatches. Again, for the lower bound, it has been seen that, for a single character change the minimum change in the sum of the xor operation could be 0. By this way, we can get the lower bound for k mismatches.

Example 26. Consider the same four strings of Example 22, i.e., $\mathcal{P} = \text{atcgatg}$, $\mathcal{T}1 = \text{aacgatg}$ and $\mathcal{T}2 = \text{ttcgatg}$. As can be easily verified, here $\mathcal{P} \equiv_1 \mathcal{T}1$ and $\mathcal{P} \equiv_1 \mathcal{T}2$. Now consider the extended strings and assume that $\mathcal{A} = \text{ext}(\mathcal{P})$, $\mathcal{B}1 = \text{ext}(\mathcal{T}1)$ and $\mathcal{B}2 = \text{ext}(\mathcal{T}2)$. It can be easily verified that $\text{sum_xor}(\mathcal{C}(\mathcal{A}))$ is 28. Recall that $\mathcal{T}1$ is slightly different from \mathcal{P} , i.e., $\mathcal{P}[2] = t \neq \mathcal{T}1[2] = a$. Now we have $\mathcal{T}1_N = 1123143$. Hence $\mathcal{B}1_N = 11231431$. Hence, $\text{sum_xor}(\mathcal{B}1) = 20$ which is indeed within the bounds of Observation 18. Now consider $\mathcal{T}2$, which is slightly different from \mathcal{P} , i.e., $\mathcal{P}[1] = a \neq \mathcal{T}2[1] = t$. Now we have $\mathcal{T}2_N = 4423143$. Hence $\mathcal{B}2_N = 44231434$. Hence, $\text{sum_xor}(\mathcal{B}2) = 28$, which is also within the bounds. Again, consider $\mathcal{T}' = \text{atagctg}$, which is again slightly different from \mathcal{P} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}'$. However, assuming that $\mathcal{B}' = \text{ext}(\mathcal{T}')$ we find that $\text{sum_xor}(\mathcal{B}')$ is still 28, which is in the range of Observation 18. This is an example of a false positive with respect to Filter 6 in 2.10.5. Finally, consider another string $\mathcal{T}'' = \text{aaaaaaa}$, which is totally different from \mathcal{T} . It can be easily verified that $\mathcal{C}(\mathcal{P}) \not\equiv_1 \mathcal{T}''$. Now, $\mathcal{T}''_N = 1111111$ and hence assuming that $\mathcal{B}'' = \text{ext}(\mathcal{T}'')$ we find that $\text{sum_xor}(\mathcal{B}'') = 0$. Clearly this is not within the bounds of Observation 18. This is an example of a false negative with respect to Filter 6 in 2.10.5.

2.11 Summary

In this chapter, we presented a brief overview of notations related to stringology. We described our problems (Problem 1 to Problem 4) based on their present state of the condition. Then we discussed our filtering based observations. Our filter-based observations have been shown in 3 different ways. At first we described filtering observations for exact circular string. Secondly, we described filtering observations for approximate circular string. Finally, in the end we showed filtering observations for classical linear pattern. Furthermore, we showed examples related to each observations. In the next chapter, we will represent our filter-based pattern signature and a framework for search space reduction. Then, using this framework, we will show our algorithms to solve all the problems (Problem 1 to Problem 4).

Chapter 3

Filter based Algorithmic Framework

As discussed in Chapters 1 and 2, our algorithms are based on some filters. In this chapter, we formulate a framework to solve the problems, Problems 1 to 4 as described in Chapter 2. At first, we introduce the concept of a pattern signature which will be useful for all the problems. Subsequently, we propose a framework which will be used to minimize the search space of the algorithms. Finally, we show different algorithms for different problems, Problem 1 to 4 as described in Chapter 2.

3.1 Pattern Signature using the Filters

In this section, we discuss the concept of a pattern signature based on the filters that will be used in our algorithms. This concept will be applicable to the circular pattern $\mathcal{C}(\mathcal{P})$ as well as to the (linear) pattern \mathcal{P} itself. This signature is used at a later stage to filter the text. Here, we need five variables to save the output of the functions used for Filters 1, 2, 3, 5 and 6 for each problem (based on *Observations* 1 to 18). And we need a list of size 4 to save the values of the function used in Filter 4 (*Observations* 4, 10 and 16). We start with the extended string $ext(\mathcal{P}) = \mathcal{P}[1 : m]\mathcal{P}[1]$ for circular string or \mathcal{P} for linear string and compute the values according to *Observations* 1 to 18. The algorithm will iterate $m + 1$ (for circular string) or m (for linear string) times and hence the overall runtime of the algorithm is $\mathcal{O}(m)$. The algorithm is presented in Procedure $PSF_FT(pattern_type)$ (Algorithm 3.1).

Here, *pattern_type* is a parameter which is being passed to the procedure. In case of, classical pattern matching, we pass “linear” as pattern type and on the other side, we pass “circular” as pattern type for circular pattern matching as well as circular sequence comparison.

Algorithm 3.1 Procedure *PSF_FT(pattern_type)*: Pattern Signature Algorithmic Framework using Filters 1 : 6 in a single pass

```

1: define five variables for filters 1, 2, 3, 5, 6
2: define an array of size 4 for filters 4
3: define an array of size 4 to keep fixed value of A, C, G, T
4: if (pattern_type = circular) then
5:    $s \leftarrow \mathcal{P}[1 : m]\mathcal{P}[1]$  /*For circular pattern matching. Here,  $s = ext(\mathcal{P})$ */
6: else
7:   /*For classical pattern matching. (pattern_type = linear)*/
8:    $s \leftarrow \mathcal{P}[1 : m]$  /*Here,  $s = \mathcal{P}$ */
9: end if
10: initialize all defined variables to zero
11: initialize fixed array to {1, 2, 3, 4}
12: for  $i \leftarrow 1$  to  $|s|$  do
13:   if  $i \neq |s|$  then
14:     calculate different filtering values via observations 1 & 4 and make a running sum
15:   end if
16:   calculate different filtering values via observations 2, 3, 5 & 6 and make a running sum
17: end for
18:
19: return all observations values

```

So, at this point, we can summarize, how the Procedure *PSF_FT(pattern_type)* works for both circular and linear pattern as follows:

- **Linear Pattern:** In classical pattern matching, we need to consider only the linear version of pattern string itself. That’s why we only copy the pattern string \mathcal{P} in s in the Procedure *PSF_FT(pattern_type)* in line 8. Except this, all are same for both linear and circular string. This is being done when the passed parameter to the procedure is *linear*, i.e., *pattern_type* = *linear*.
- **Circular Pattern:** In circular pattern matching as well as circular sequence comparison, we need to consider the circular version of pattern string. That’s why we need to consider the extended pattern string that means $s = ext(\mathcal{P})$. In the Procedure

$PSF_FT(pattern_type)$ (in line 5), we append the first character at the last position to make the linear string circular means we make extended pattern string (we have discussed this before in Chapter 2). This is being done when the passed parameter to the procedure is *circular*, i.e., $pattern_type = circular$.

Hence, the running time of Procedure $PSF_FT(pattern_type)$ is $\mathcal{O}(m)$ for both the circular and linear pattern string, where m is the length of the pattern string.

3.2 Reduction of Search Space

In this section, we present our main idea to reduce the search space of the text applying the six filters presented in Chapter 2 for each problems. We develop a *Procedure* named RSS_FT (Algorithm 3.2) which will be used to reduce the search space for both the linear and circular pattern strings. This procedure takes as input the pattern $\mathcal{P}[1 : m]$ of length m and the text $\mathcal{T}[1 : n]$ of length n . Procedure RSS_FT calls Procedure $PSF_FT(pattern_type)$ with $\mathcal{P}[1 : m]$ and pattern type (linear or circular) as parameter and uses the output that returned by the Procedure $PSF_FT(pattern_type)$. This Procedure (RSS_FT) then applies the same technique that is applied in Procedure $PSF_FT(pattern_type)$ (Algorithm 3.1) to calculate the values of relevant *Observations* to reduce the search space. This means, we apply a sliding window approach with window length of m and calculate the values applying the functions according to Filters 1 : 6 for each problem on the factor of \mathcal{T} captured by the window. Note that for Filters 2, 3, 5 and 6, we need to consider the extended string (in case of circular) and hence the factor of \mathcal{T} within the window need be extended accordingly for calculating the values. After we calculate the values for a factor of \mathcal{T} , we check it against the returned values of Procedure PSF_FT . If it matches, then we output the factor to a file. Note that in case of overlapping factors (e.g., when the consecutive windows need to output the factors to a file), Procedure RSS_FT outputs only the non-overlapped characters. And Procedure RSS_FT uses a \$ marker to mark the boundaries of non-consecutive factors, where $\$ \notin \Sigma$.

Now note that we can compute the values of consecutive factors of \mathcal{T} using the sliding window approach quite efficiently as follows. For the first factor, i.e., $\mathcal{T}[1..m]$ we exactly follow

Algorithm 3.2 Procedure *RSS_FT*: Reduction of Search Space in a Text String using procedure *PSF_FT*(*pattern_type*)

```

1: call PSF_FT ( $\mathcal{P}[1 : m]$  and pattern_type) /*Pass ‘circular’ incase of circular pattern
   matching or ‘linear’ for classical pattern matching*/
2: save the return value of filters 1 : 6 for further use here
3: define an array of size 4 to keep fixed value of A, C, G, T
4: initialize fixed array to {1, 2, 3, 4}
5: for  $i \leftarrow 1$  to  $m$  do
6:   calculate different filtering values in  $\mathcal{T}[1 : m]$  via filters 1 : 6 and make a running sum
7: end for
8: if 1 : 6 filters values of  $\mathcal{P}[1 : m]$  vs 1 : 6 filters values of  $\mathcal{T}[1 : m]$  have a match then
9:   /*Found a filtered match*/
10:  Output to file  $\mathcal{T}[1 : m]$ 
11: end if
12: for  $i \leftarrow 1$  to  $n - m$  do
13:  calculate different filtering values in  $\mathcal{T}[1 : m]$  via filters 1 : 6 by subtracting  $i$ -th
   value along with wrapped value (in case of circular) and adding  $i + m$ -th value
   and new wrapped value to the running sum /*wrap the text string in case of
   ‘circular pattern matching’ and ‘circular sequence comparison’, otherwise go
   straight without wrapping for ‘classical pattern matching’*/
14:  if 1 : 6 filtering values of  $\mathcal{P}[1 : m]$  vs 1 : 6 filtering values of  $\mathcal{T}[i + 1 : i + m]$  have a match
   then
15:    /*Found a filtered match*/
16:    check whether non-consecutive sequence or not. if non-consecutive put an end marker
   $ to file
17:    Output to file  $\mathcal{T}[i + 1 : i + m]$ 
18:  end if
19: end for

```

the strategy of Procedure $PSF_FT(pattern_type)$. When it is done, we slide the window by one character and we only need to remove the contribution of the left most character of the previous window and add the contribution of the rightmost character of the new window. The functions are such that this can be done very easily using simple constant time operations. The only other issue that needs be taken care of is due to the use of the extended string in four of the filters in case of circular string. On the other hand, we don't need to consider extended string for linear string, i.e., for classical pattern matching. But this too do not need more than simple constant time operations. Therefore, overall runtime of the algorithm is $\mathcal{O}(m) + \mathcal{O}(n - m) = \mathcal{O}(n)$. The algorithm is presented in the form of Procedure RSS_FT (Algorithm 3.2). Finally, we can summarize, how the Procedure RSS_FT works for both circular and linear pattern as follows:

- **Linear Pattern:** In classical pattern matching, we need to consider only the linear version of pattern string itself. When the Procedure RSS_FT class Procedure $PSF_FT(pattern_type)$, it passes pattern type “linear” in line 1. Again, when this procedure calculate sliding window for the *Observations*, it doesn't consider extended string in line 13. Except these, all are same for both liner and circular string. This is being done for classical pattern matching.
- **Circular Pattern:** Again, in circular pattern matching as well as circular sequence comparison, we need to consider the circular version of pattern string. That's why When the Procedure RSS_FT class Procedure $PSF_FT(pattern_type)$, it passes pattern type “circular” in line 1. Further, when this procedure calculate sliding window for the *Observations*, it does consider extended string in line 13. Except these, all are same for both liner and circular string.

3.2.1 An Illustrative Example for the ECPM Problem

Now that we have fully described the search space reduction algorithm (Algorithm 3.2), in this section, we present the simulation of the algorithm RSS_FT (actually we consider exact

Table 3.1: An example simulation of Filtered-ECPM

iteration	local total sum	abs sum	actual sum	local individual sum[0:4]	modulas sum	xor sum	match with pattern?	Output File
1	18	14	0	{2, 2, 6, 8}	5	28	YES	tgatcga
2	15	12	0	{3, 2, 6, 4}	4	18	NO	\$
3	13	8	0	{4, 2, 3, 4}	3	14	NO	
4	15	8	0	{3, 2, 6, 4}	6	18	NO	
5	15	8	0	{3, 2, 6, 4}	6	18	NO	
6	14	10	0	{4, 0, 6, 4}	5	18	NO	
7	12	6	0	{5, 0, 3, 4}	4	14	NO	
8	15	12	0	{4, 0, 3, 8}	5	24	NO	
9	16	12	0	{3, 2, 3, 8}	5	28	NO	
10	18	10	0	{2, 2, 6, 8}	6	24	NO	
11	16	14	0	{3, 2, 3, 8}	4	24	NO	
12	16	14	0	{3, 2, 3, 8}	4	24	NO	
13	18	14	0	{2, 2, 6, 8}	5	28	YES	atcgatg

circular pattern here) on a particular example. We only show the simulation upto the output of Procedure *RSS_FT*, i.e., the output of the reduced text, because afterwards we can employ any state of the art algorithm that can solve ECPM. Consider a pattern $\mathcal{P} = atcgatg$. The values computed by Procedure *PSF_FT* according to the Observations 7 through 12 are as follows, respectively (see examples 15 to 20): $local_total_sum = 18$, $abs_sum = 14$, $actual_sum = 0$, $local_individual_sum[0 : 4] = \{2, 2, 6, 8\}$, $modulas_sum = 5$ and $xor_sum = 28$.

Again consider a text string $\mathcal{T} = tgatcgaagtaatcgatg\$$. For the first sliding window we need to calculate the observation values from $\mathcal{T}[1 : 7]$. The observation values according to Procedure *RSS_FT* are as follows for $\mathcal{T}[1 : 7]$: $local_total_sum = 18$, $abs_sum = 14$, $actual_sum = 0$, $local_individual_sum[0 : 4] = \{2, 2, 6, 8\}$, $modulas_sum = 5$ and $xor_sum = 28$.

The length of \mathcal{T} is 19. And the length of \mathcal{P} is 7. So, the algorithm iterates exactly $19 - 7 + 1 = 13$ times. Each iteration is illustrated in *Table 3.1*.

Finally, we apply state of the art algorithm into the filtered text to solve the ECPM. Here, applying any state of art algorithm we find that the patten \mathcal{P} matches text \mathcal{T} at positions 1 and 13. This happens due \mathcal{P}^2 matches \mathcal{T} at position 1 and \mathcal{P}^0 matches \mathcal{T} at position 13. So, this algorithm reports output: 1 and 13.

3.3 The Combined Algorithm for Classical Pattern Matching Problem

In this section, we present an algorithm to solve the classical pattern matching problem applying the six filters presented in Section 2.8, Chapter 2. In our algorithm we use two different state of the art algorithms to solve the classic pattern matching problem in the reduced text. These two state of the art algorithms are: KMP algorithm [26] and BM algorithm [27]. When we use KMP algorithm to solve the classical pattern matching problem in our algorithm, then we call our classical pattern matching algorithm as *Filtered-KMP*. Again, on the other hand, when we use BM algorithm to solve the same problem, then we call our algorithm as *Filtered-BM* algorithm. Hence, from this point of view, we will use these two names (Filtered-KMP and Filtered-BM) to refer to our classical pattern matching algorithm. Both the Filtered-KMP and Filtered-BM take as input the pattern $\mathcal{P}[1 : m]$ of length m , the text $\mathcal{T}[1 : n]$ of length n . Both the algorithms call Procedure $PSF_FT(pattern_type)$ 3.1 with $\mathcal{P}[1 : m]$ and “linear” as parameters and uses the output of it. We apply a sliding window approach with window length of m and calculate the values applying the functions according to Observations 1 : 6 on the factor of \mathcal{T} captured by the window. After we calculate the values for a factor of \mathcal{T} , we check it against the returned values of Procedure $PSF_FT(pattern_type)$. If it matches, then we save the matched index and length as bookkeeping. Note that in case of overlapping factors (e.g., when the consecutive windows need to save), Procedure $PSF_FT(pattern_type)$ saves only the non-overlapped indexes and lengths.

Now note that we can compute the values of consecutive factors of \mathcal{T} using the sliding window approach quite efficiently as follows. For the first factor, i.e., $\mathcal{T}[1..m]$ we exactly follow the strategy of Procedure $PSF_FT(pattern_type)$. When it is done, we slide the window by one character and we only need to remove the contribution of the left most character of the previous window and add the contribution of the rightmost character of the new window. The functions are such that this can be done very easily using simple constant time operations. Therefore, overall runtime of the algorithm is $\mathcal{O}(m) + \mathcal{O}(n - m) = \mathcal{O}(n)$ like as the previously described algorithms for search space reduction. In our implementation we have used two

Algorithm 3.3 Algorithm Filtered-BM/Filtered-KMP using Procedure PSF_FT(pattern.type) Algorithm 3.1

```

1: call PSF_FT(linear) Algorithm 3.1 /*make pattern signature for classical pattern
   matching using linear pattern itself.*/
2: save the return value of filters 1 : 6 for further use here
3: define an array of size 4 to keep fixed value of A, C, G, T
4: initialize fixed array to {1, 2, 3, 4}
5: lastIndex  $\leftarrow$  1
6: startIndex  $\leftarrow$  -1
7: length  $\leftarrow$  -1
8: define vector pair bookShelf to save startIndex and length
9: for  $i \leftarrow 1$  to  $m$  do
10: calculate different filtering values in  $\mathcal{T}[1 : m]$  via observations 1 : 6 and make a running
    sum
11: end for
12: if 1 : 6 observations values of  $\mathcal{P}[1 : m]$  vs 1 : 6 observations values of  $\mathcal{T}[1 : m]$  have a match
    then
13: /*Found a filtered match*/
14: lastIndex  $\leftarrow$   $m$ 
15: startIndex  $\leftarrow$  1
16: length  $\leftarrow$   $m$ 
17: end if
18: for  $i \leftarrow 1$  to  $n - m$  do
19: calculate different filtering values in  $\mathcal{T}[1 : m]$  via observations 1 : 6 by subtracting  $i$ -th
    value and adding  $i + m$ -th value to the running sum
20: if 1 : 6 filtering values of  $\mathcal{P}[1 : m]$  vs 1 : 6 filtering values of  $\mathcal{T}[i + 1 : i + m]$  have a match
    then
21: /*Found a filtered match*/
22: if  $i > \textit{lastIndex}$  then
23: if  $\textit{startIndex} \neq -1$  then
24: bookShelf.add(make_pair(startIndex, length))
25: end if
26: startIndex  $\leftarrow$   $i + 1$ 
27: length  $\leftarrow$  0
28: end if
29: if  $i + m > \textit{lastIndex}$  then
30: if  $i < \textit{lastIndex}$  then
31:  $j \leftarrow \textit{lastIndex} + 1$ 
32: else
33:  $j \leftarrow i + 1$ 
34: end if
35: length  $\leftarrow$  length +  $(i + 1 + m) - j$ 
36: lastIndex  $\leftarrow$   $i + m$ 
37: end if
38: end if
39: end for
40: bookShelf.add(make_pair(startIndex, length))
41: for  $k \leftarrow 1$  to bookShelf.size() do
42: call BM Algorithm [27] (for Filtered-BM) or KMP Algorithm [26] (for Filtered-KMP)
    for each matched pair and report the occurrences.
43: end for

```

different efficient string matching algorithms. First one is *KMP* algorithm [26] and second one is *BM* algorithm [27]. In particular, in [26, 27], the authors have presented two different efficient algorithms for pattern matching. We only apply *KMP* or *BM* algorithm on the reduced string.

Finally, we can summarize our algorithms for classical pattern matching problem as follows:

- **Filterd-BM Algorithm:** This algorithm uses *BM Algorithm* [27] to solve the classical pattern matching problem. In our algorithm (Algorithm 3.3), we call *BM Algorithm* [27] in line 42 in case of Filtered-BM algorithm. Except this all others are same for both Filterd-KMP and Filterd-BM algorithms.
- **Filterd-KMP Algorithm:** Again, this algorithm uses *KMP Algorithm* [26] to solve the classical pattern matching problem. In our algorithm (Algorithm 3.3), we call *KMP Algorithm* [26] in line 42 in case of Filtered-KMP algorithm.

3.4 The Combined Algorithm for ECPM and ACPM

In this section we combine the Algorithms 3.1 and 3.2 presented so far and present the complete view of ECPM and ACPM. We have already described our algorithm to solve ECPM and ACPM problems, namely, Procedure *PSF_FT(pattern_type)* and Procedure *RSS_FT* that in fact calls the former. We develop two different algorithms using filters to solve these problems. We name our algorithm *Filtered-ECPM* to solve the problem ECPM. Again, on the other side, we name our algorithm *Filtered-ACPM* to solve the problem ACPM. Now Procedure *RSS_FT* provides a reduced text \mathcal{T}' (say) after filtering. At this point Algorithms Filtered-ECPM/Filtered-ACPM can use any algorithm that can solve CPM and apply it over \mathcal{T}' and output the occurrences. Now, suppose Filtered-ECPM/Filtered-ACPM uses Algorithm \mathcal{A} at this stage which runs in $\mathcal{O}(f(|\mathcal{T}'|))$ time. Then, clearly, the overall running time of Filtered-ECPM/Filtered-ACPM is $\mathcal{O}(n) + \mathcal{O}(f(|\mathcal{T}'|))$. For example, if Filtered-ECPM/Filtered-ACPM uses the linear time algorithm of [30], then clearly the overall theoretical running time of Filtered-ECPM/Filtered-ACPM will be $\mathcal{O}(n)$.

Algorithm 3.4 Algorithm Filtered-ECPM/Filtered-ACPM using Procedure PSF_FT(pattern_type) Algorithm 3.1

```

1: call PSF_FT 3.1 ( $\mathcal{P}[1 : m]$ , circular, exact/approximate)
2: save the return value of filters 1 : 6 for further use here
3: define an array of size 4 to keep fixed value of A, C, G, T
4: initialize fixed array to {1, 2, 3, 4}
5:  $lastIndex \leftarrow 1$ 
6:  $startIndex \leftarrow -1$ 
7:  $length \leftarrow -1$ 
8: define vector pair bookShelf to save startIndex and length
9: for  $i \leftarrow 1$  to  $m$  do
10:   calculate different filtering values in  $\mathcal{T}[1 : m]$  via observations 7 : 12 (exact) or 13 : 18 (approximate) and make a running sum
11: end for
12: if 7 : 12 (exact) or 13 : 18 (approximate) observations values of  $\mathcal{P}[1 : m]$  vs 7 : 12 (exact) or 13 : 18 (approximate) observations values of  $\mathcal{T}[1 : m]$  have a match then
13:   /*Found a filtered match*/
14:    $lastIndex \leftarrow m$ 
15:    $startIndex \leftarrow 1$ 
16:    $length \leftarrow m$ 
17: end if
18: for  $i \leftarrow 1$  to  $n - m$  do
19:   calculate different filtering values in  $\mathcal{T}[1 : m]$  via observations 7 : 12 (exact) or 13 : 18 (approximate) by subtracting  $i$ -th value along with wrapped value and adding  $i + m$ -th value and new wrapped value to the running sum
20:   if 7 : 12 (exact) or 13 : 18 (approximate) observations values of  $\mathcal{P}[1 : m]$  vs 7 : 12 (exact) or 13 : 18 (approximate) observations values of  $\mathcal{T}[1 : m]$  have a match then
21:     /*Found a filtered match*/
22:     if  $i > lastIndex$  then
23:       if  $startIndex \neq -1$  then
24:         bookShelf.add(make_pair(startIndex, length))
25:       end if
26:        $startIndex \leftarrow i + 1$ 
27:        $length \leftarrow 0$ 
28:     end if
29:     if  $i + m > lastIndex$  then
30:       if  $i < lastIndex$  then
31:          $j \leftarrow lastIndex + 1$ 
32:       else
33:          $j \leftarrow i + 1$ 
34:       end if
35:        $length \leftarrow length + (i + 1 + m) - j$ 
36:        $lastIndex \leftarrow i + m$ 
37:     end if
38:   end if
39: end for
40: bookShelf.add(make_pair(startIndex, length))
41: for  $k \leftarrow 1$  to bookShelf.size() do
42:   call ACSMF-SimpleZero $k$  (exact) or ACSMF-Simple (approximate) [1] for each matched pair and report the occurrences.
43: end for

```

In our implementation however, we have used the recent algorithm of [1], which is a linear time algorithm on average and the fastest algorithm in practice to the best of our knowledge. In particular, in [1], the authors have presented an approximate circular string matching algorithm with k -mismatches (ACSMF-Simple) via filtering. They have built a library for ACSMF-Simple algorithm. The library is freely available and can be found at: [32]. In this algorithm, if we set $k = 0$, then ACSMF-Simple solves problem the exact matching case; otherwise it solves the approximate circular pattern matching problem. In what follows, we will refer to this algorithm with $k = 0$ as ACSMF-SimpleZero k . We have implemented Filtered-ECPM algorithm using ACSMF-SimpleZero k , i.e., we have used ACSMF-Simple algorithm simply by putting $k = 0$. Again, we have implemented Filtered-ACPM algorithm using ACSMF-Simple [1]. At this point, finally, we can summarize our algorithms for CPM problems (both exact and approximate) as follows:

- **Filtered-ECPM Algorithm:** This algorithm uses *Observations 7* to *Observations 12* to calculate pattern signature by calling $PSF_FT(pattern_type)$ in line 1. Again, it uses same *Observations* in lines 10, 12, 19 and 20. This algorithm also calls ACSMF-Simple with $k = 0$ in line 42. Except these, all others are same for both Filtered-ECPM and Filtered-ACPM.
- **Filtered-ACPM Algorithm:** This algorithm uses *Observations 13* to *Observations 18* to calculate pattern signature by calling $PSF_FT(pattern_type)$ in line 1. Note that, these all *Observations* are according to approximate version of circular string. Again, it uses same *Observations* in lines 10, 12, 19 and 20. This algorithm also calls ACSMF-Simple [1] for all values of k , i.e., $0 < k < m$, where m is the length of circular pattern.

3.5 The Algorithm for CSC Problem

In this section, we present an algorithm to solve the CSC problem applying the six filters presented in Section 2.9, Chapter 2. Here, we name our algorithm as *Filtered-CSC* to solve the circular sequence comparison problem using filters. This algorithm takes as input the

pattern $\mathcal{P}[1 : m]$ of length m , the text $\mathcal{T}[1 : n]$ of length n , block-size β and q -gram size q . Like as Algorithm *Filtered-ECPM*, this algorithm calls Procedure *PSF_FT(pattern_type)* 3.1 with $\mathcal{P}[1 : m]$ and “circular” type as parameters and uses the output of it. In our algorithm (Algorithm *Filtered-CSC* 3.5), upto line 40 everything is same as before (like Algorithm *Filtered-ECPM*); subsequently we use a state of the art algorithm to solve the CSC problem in the reduced text \mathcal{T}' (say). For rest of the Algorithm from line 43, using this reduced text \mathcal{T}' after filtering where the start index and length of each reduced text are saved as bookkeeping in the Algorithm 3.5. At this point we can use any algorithm that can solve the CSC problem and apply it over \mathcal{T}' and output the best comparison which minimizes the q -gram distance based on the distance matrix described for Problem 4. Now, suppose we use Algorithm \mathcal{A} at this stage which runs in $\mathcal{O}(f(|\mathcal{T}'|))$ time. Then, clearly, the overall running time of our approach is $\mathcal{O}(n) + \mathcal{O}(f(|\mathcal{T}'|))$.

In our implementation of *Filtered-CSC* algorithm, we have used the Algorithm *saCSC*, the recent algorithm of [2]. In particular, in [2], the authors have presented a circular sequence comparison algorithm using suffix-array construction. They have built a library to solve CSC problem. The library is freely available and can be found at: [53]. We only apply Algorithm *saCSC* on the reduced string to solve the CSC problem.

3.6 Summary

In this chapter, we have described our proposed algorithmic framework based on filters. We have presented our filter-based algorithms for circular strings and sequences based on this proposed framework. We have also presented a filter-based classical pattern matching algorithm. In the next chapter we present our experimental study based on these algorithms. We have also shown comparisons among our proposed algorithms and the state of art algorithms.

Algorithm 3.5 Algorithm Filtered-CSC ($\mathcal{T}[1 : n]$, $\mathcal{P}[1 : m]$, β , q): using procedure PSF_FT(pattern_type) Algorithm 3.1

```

1: call PSF_FT ( $\mathcal{P}[1 : m]$ , circular, exact)
2: save the return value of filters 1 : 6 for further use here
3: define an array of size 4 to keep fixed value of A, C, G, T
4: initialize fixed array to {1, 2, 3, 4}
5:  $lastIndex \leftarrow 1$ 
6:  $startIndex \leftarrow -1$ 
7:  $length \leftarrow -1$ 
8: define vector pair bookShelf to save startIndex and length
9: for  $i \leftarrow 1$  to  $m$  do
10:   calculate different filtering values in  $\mathcal{T}[1 : m]$  via observations 7 : 12 and make a running sum
11: end for
12: if 7 : 12 observations values of  $\mathcal{P}[1 : m]$  vs 7 : 12 observations values of  $\mathcal{T}[1 : m]$  have a match
then
13:   /*Found a filtered match*/
14:    $lastIndex \leftarrow m$ 
15:    $startIndex \leftarrow 1$ 
16:    $length \leftarrow m$ 
17: end if
18: for  $i \leftarrow 1$  to  $n - m$  do
19:   calculate different filtering values in  $\mathcal{T}[1 : m]$  via observations 7 : 12 by subtracting  $i$ -th value
   along with wrapped value and adding  $i + m$ -th value and new wrapped vale to the running sum
20:   if 7 : 12 filtering values of  $\mathcal{P}[1 : m]$  vs 7 : 12 filtering values of  $\mathcal{T}[i + 1 : i + m]$  have a match
   then
21:     /*Found a filtered match*/
22:     if  $i > lastIndex$  then
23:       if  $startIndex \neq -1$  then
24:          $bookShelf.add(make\_pair(startIndex, length))$ 
25:       end if
26:        $startIndex \leftarrow i + 1$ 
27:        $length \leftarrow 0$ 
28:     end if
29:     if  $i + m > lastIndex$  then
30:       if  $i < lastIndex$  then
31:          $j \leftarrow lastIndex + 1$ 
32:       else
33:          $j \leftarrow i + 1$ 
34:       end if
35:        $length \leftarrow length + (i + 1 + m) - j$ 
36:        $lastIndex \leftarrow i + m$ 
37:     end if
38:   end if
39: end for
40:  $bookShelf.add(make\_pair(startIndex, length))$ 
41:  $index \leftarrow \infty$ 
42:  $saveI \leftarrow -1$ 
43: for  $k \leftarrow 1$  to  $bookShelf.size()$  do
44:   call saCSC [2] ( $bookShelf.get(pair(k))$ ,  $\beta$ ,  $q$ )
45:   save the minimum distance in minDistance and corresponding index  $i$  in saveI
46: end for
47: Output saveI and minDistance

```

Chapter 4

Experimental Studies

This chapter evaluates the performance of our proposed algorithms using real genome data. We first describe the datasets and implementation strategy. Then we show effectiveness of our filters used for the reduction of the search space in the text. We also analyze the effect of our algorithms comparing with the most recent state of art algorithms to solve the problems, Problem 1 to Problem 4.

4.1 Dataset

We have used real genome data in our experiments as the text string, \mathcal{T} . This data has been collected from [54]. This genome dataset contains the Feb. 2009 assembly of the human genome (hg19¹, GRCh37 Genome Reference Consortium Human Reference 37 (GCA_000001405.1)), as well as repeat annotations and GenBank sequences. The Feb. 2009 human reference sequence (GRCh37) was produced by the Genome Reference Consortium: [55]. A brief description of our used genome data (GRCh37) has been shown in Table 4.1.

We have made three different text strings for our experiments based on the text length. These are *299MB*, *700MB* and *1GB* of data. Table 4.2 shows the techniques how we have

¹Since the release of the UCSC hg19 assembly, the Homo sapiens mitochondrion sequence (represented as “chrM” in the Genome Browser) has been replaced in GenBank with the record NC_012920. We have not replaced the original sequence, NC_001807, in the hg19 Genome Browser. We plan to use the Revised Cambridge Reference Sequence (rCRS, <http://mitomap.org/bin/view.pl/MITOMAP/HumanMitoSeq>) in the next human assembly release.

Table 4.1: A brief overview of genome data *GRCh37*

Category	Description
Description	Genome Reference Consortium Human Build 37
Organism name	Homo sapiens
Submitter	Genome Reference Consortium
Year	2009
Synonyms	hg37
Assembly type	haploid-with-alt-loci
Assembly level	Chromosome
Genome representation	full
RefSeq category	representative genome
GenBank assembly accession	GCA_000001405.15 (replaced)
RefSeq assembly accession	GCF_000001405.26 (replaced)
RefSeq assembly and GenBank assembly identical	yes

generated these three different sets of data. We have generated random patterns of different lengths by a *random indexing technique* (see description below) in these *299MB*, *700MB* and *1GB* of text string.

Random Indexing Technique: We have coded a program in *Java* programming language to generate patterns of different sizes from these three (mentioned above) sets of data. We have used *Random*² class in *Java* to generate random indices (integer numbers) between indices 0 and $n - 1$, where n is the length of the data. For our experiments, the values of n are: *299MB*, *700MB* and *1GB*. Suppose, we have determined a random index, i.e., r (say), which is a integer number. Using this index (randomly generated), we have generated patterns of different sizes starting from this index r . Assume that, we want to generate a random pattern of size 1000, then we have to start at position r and end at position $r + 1000 - 1$ to get the random pattern of size 1000. In our experiments, this pattern size is: $5 \leq m \leq 1200000$, where m is the length of pattern. The length of

²public class Random extends Object implements Serializable. An instance of this class is used to generate a stream of pseudorandom numbers.

Table 4.2: An overview of genome datasets: *299MB*, *700MB* and *1GB*.

Dataset	Description
299MB	We have used real genome data (mentioned before) to generate this text of size 299MB. The data, we have collected from [54] was in text form. This text was sequence of ‘A’, ‘C’, ‘G’ & ‘T’ (the DNA nucleotides). This text was in multi-line format. We have coded a program in <i>Java</i> programming language to read this text data and to convert this multi-line text data into a single line text data. After converting this multi-line text into a single line text, we have found that the size of the text data is exactly <i>299MB</i> . Then, we have kept this 299MB of text intact for our experiments as a one set of data.
700MB	To generate this text of size 700MB, we have used the single line text of size 299MB which was generated earlier in our experiments. Again, we have coded a program in same language (mentioned above) to generate single line text of size 700MB. Then we have appended 299MB of text with itself to generate 700MB of text.
1GB	Using the same technique (the technique by which we generated 700MB of text), we have generated 1GB of single line text. This time, we have started from 700MB of initial text (as input).

pattern varies as per experiment basis.

4.2 Environment & Experimental Settings

We have used three different environmental settings. These are as follows.

Setup 1: The experiments to solve classical pattern matching problem using Filtered-KMP and Filtered-BM algorithms were conducted on an i3-core 330 CPU at 3.4 GHz machine having 2GB of RAM running Linux operating system. We have implemented Filtered-KMP and Filtered-BM algorithms in the *JAVA* programming platform and developed under *Java Development Kit 7* (jdk) in *Linux*.

Setup 2: We have conducted our experiments for the algorithms Filtered-ECPM and Filtered-ACPM to solve ECPM and ACPM problems on a PowerEdge R820 rack server PC with 6-core of Intel Xeon processor *E5-4600* product family and 64GB of RAM under

GNU/Linux. We have coded the Filtered-ECPM and Filtered-ACPM algorithms in C++ using a GNU compiler with General Public License (GPL). As has been mentioned already above, our implementation of the Filtered-ECPM and Filtered-ACPM algorithms use the Algorithm ACSMF-Simple [1]. With the help of the library used in [1], we have compared the running time of ACSMF-Simple of [1] with Filtered-ACPM algorithm and the running time of Filterd-ECPM algorithm with ACSMF-SimpleZero*k* of [1].

Setup 3: We have conducted our experiments of Filtered-CSC algorithm to solve CSC problem on a Samsung Laptop with Intel Core(TM) i5-2430M CPU @2.40GHz processor product family and 4GB of RAM under GNU/Linux. We have coded the Filtered-CSC algorithm in C++ using a GNU compiler with General Public License (GPL). As has been mentioned already above, our implementation of the Filtered-CSC algorithm uses the Algorithm saCSC [2]. With the help of the library used in [2], we have compared the running time of saCSC of [2] and the Filtered-CSC algorithm.

4.3 Effectiveness of Filters

Because we have 6 different filters, the order of applying the filters may turn out to be important in the final performance of the algorithm. We have done preliminary experiments to identify the best order to apply the filters. We have run this experiment on three candidate patterns of different length. As there could be 6!, i.e., 720 combinations, we have recorded the running times of our algorithm for all these combinations considering the three candidate patterns. Although it was not possible to identify one ordering that performs best in all instances, after some analysis we were able to identify an ordering that is among the top performers in every instance. The analysis is available online and can be easily accessed from: [56]. This order is shown below:

$$\text{Filter 4} \rightarrow \text{Filter 2} \rightarrow \text{Filter 5} \rightarrow \text{Filter 3} \rightarrow \text{Filter 1} \rightarrow \text{Filter 6} \quad (4.1)$$

Our main experiments have been conducted using this order listed above. Note that, this analysis has been done only on circular pattern string. We have used this order to solve the ECPM, ACPM and CSC problems. We have also used this order to solve the classical pattern matching problem (in Algorithm Filtered-KMP and Algorithm Filtered-BM). Though, we haven't done same analysis for linear string (classical pattern matching problem), we have used the same sequence to solve this problem also.

4.4 Search Space Reduction

In this section, we report the reduction of search space using Algorithm *RSS_FT* 3.2. Actually, we have done this experiments to show the effectiveness of our filters. This means, we will show that using Algorithm *RSS_FT* how much search space in the text is being reduced. By doing this experiments, we are trying to study and analyze the reduction of space against the number of filters applied. To do this study, we have used circular pattern as signature. And these experiments are done using the environment and experimental settings: **Setup 1** (mentioned in 4.2).

To report the effectiveness of filters we have done three different types experiments using Algorithm *RSS_FT*. First one is, we have run Algorithm *RSS_FT* using Filters 1 to 4 in a single pass. Secondly, we have run Algorithm *RSS_FT* using Filters 1 to 5 in a single pass. Finally, we have Algorithm *RSS_FT* using all filters in a single pass. These three experiments have been done using different pattern sizes. So, in total we report the reduction for three separate versions of our algorithm.

The results of the experiments are reported in the Table 4.3 to 4.6. In those tables, we report the reduction of search space in the filtered text string. As has been described before that, we have used real genome data as our experimental text string, T . We have collected that data from [54]. Here, we have taken *299MB* of data for our experiment. We have generated random patterns of different length by a random indexing technique in this *299MB* of text string.

As can be seen from Table 4.3, the *299MB* of text data is reduced to less than *30MB* of text

Table 4.3: Search Space Reduction of text string for $n = 299MB$ and $5 \leq m \leq 20$

m	1:4 Filters(MB)	1:5 Filters(MB)	1:6 Filters(MB)
5	11.6	7.6	7.45
6	27.5	19	8.4
7	9.6	7.2	3.7
8	6.8	3.9	2.55
9	4.3	2.8	1.5
10	4.8	3.35	1.75
11	4.2	2.8	1.7
12	2.1	0.73	0.5
13	3.4	2.1	0.86
14	6	1.2	0.52
15	4	1.8	1.2
16	1.6	1.3	0.61
17	1.7	0.07	0.03
18	1.5	0.87	0.2
19	6.76	2.1	0.233
20	1.13	0.24	0.12

Table 4.4: Search Space Reduction of text string for $n = 299MB$ and $21 \leq m \leq 35$

m	1:4 Filters(MB)	1:5 Filters(KB)	1:6 Filters(KB)
21	2.3	540.23	148.5
22	1.4	382.5	156.7
23	1.1	172.5	13.3
24	4.02	1171.9	450.6
25	2.7	408.55	16.4
26	2.3	1057.6	9.22
27	3.43	1681.7	567.3
28	0.4	187.2	11.2
29	3.5	933.5	267.3
30	2.4	843.7	196.6
31	1.7	309.1	13.3
32	3.1	1246.4	486.4
33	1.9	139.3	20.5
34	2.7	1347.8	52.22
35	2.1	644.9	183.3

data applying first four filters (Filter 1 to 4) in a single pass in Algorithm *RSS_FT*. Again, referring to the same table, it can be seen that applying first five filter (Filter 1 to 5) in a single pass in Algorithm *RSS_FT*, the 299MB of text data is reduced to less than 8MB of text data. Finally, we can see from the same table that, applying all filters (Filter 1 to 6) in a single pass in our Algorithm *RSS_FT*, the 299MB of text data is reduced to less than 7.5MB of text data. For these three types of experiments, we have used different sizes of pattern, i.e., $5 \leq m \leq 20$, where m is the length of pattern. At this point, we can claim that the more filters we apply, the more space reduction is done.

Table 4.5: Search Space Reduction of text string for $n = 299MB$ and $40 \leq m \leq 200$

m	1:4 Filters(KB)	1:5 Filters(KB)	1:6 Filters
40	307	133.1	5.1KB
50	665	218.3	44.03KB
60	215.12	80.73	2.33KB
70	606.8	99.7	20.5KB
80	603.8	38.8	1.9KB
90	246.7	56.9	4.5KB
100	456.8	39.9	4.9KB
110	273.1	20.3	333B
120	292.9	76.1	1.3KB
130	95.6	15.9	393B
140	212.5	27.9	1.1KB
150	162.7	37.6	453B
160	137.8	14.5	161B
170	12.5	3.7	171B
180	110.4	0.2	181B
190	49.1	0.6	191B
200	128.4	10.4	201B

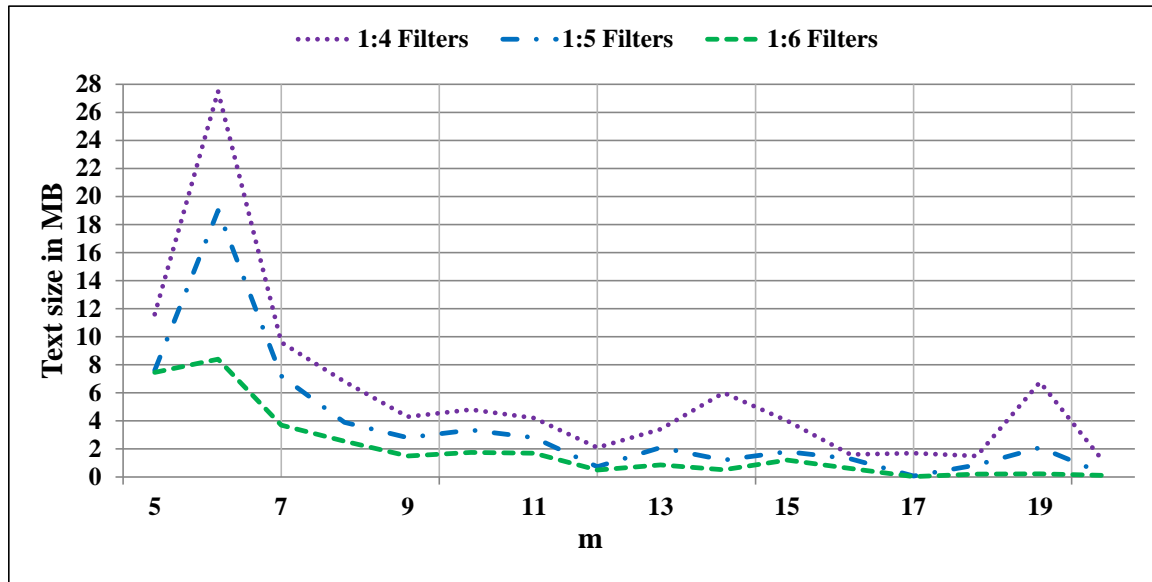
Again, it also can be seen from Table 4.4, the 299MB of text data is reduced to less than 5MB of text data applying first four filters (Filter 1 to 4) in a single pass in Algorithm *RSS_FT*. Again, referring to the same table, it can be seen that applying first five filter (Filter 1 to 5) in a single pass in Algorithm *RSS_FT*, the 299MB of text data is reduced to less than 1.5MB of text data. Finally, we can see from the same table that, applying all filters (Filter 1 to 6) in a single pass in our Algorithm *RSS_FT*, the 299MB of text data is reduced to less than 1MB

Table 4.6: Search Space Reduction of text string for $n = 299MB$ and $220 \leq m \leq 450$

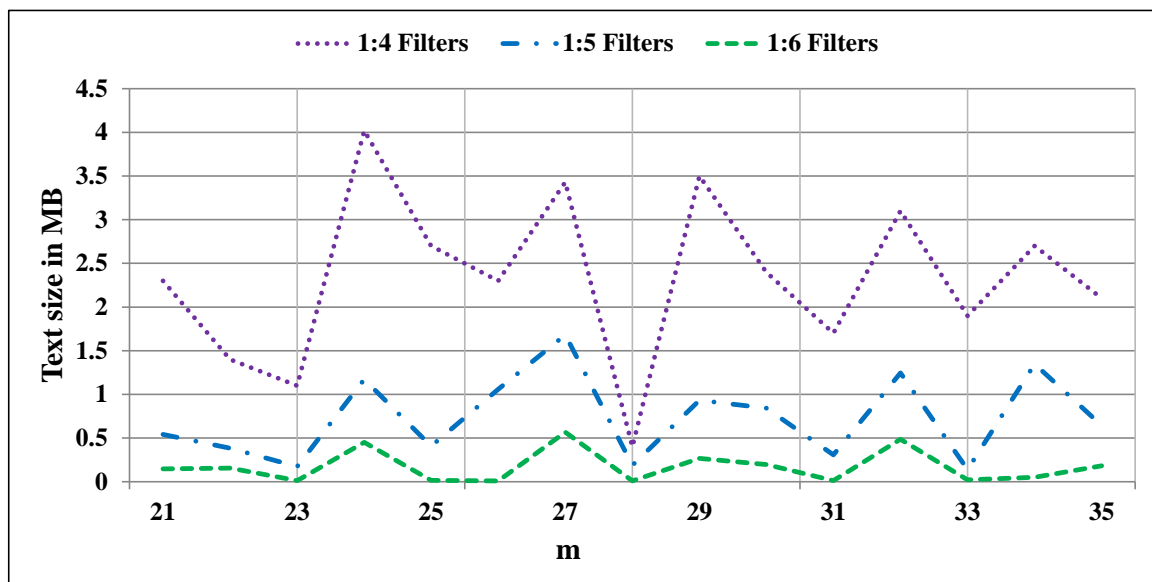
m	1:4 Filters	1:5 Filters	1:6 Filters
220	86.8KB	11.1KB	443B
240	11KB	2.3KB	483B
260	46.3KB	3.5KB	261B
280	56.12KB	3.2KB	281B
300	60.2KB	5.5KB	602B
320	35.2KB	1.4KB	642B
340	27.1KB	2.5KB	682B
360	8671B	1KB	361B
380	1908B	1KB	381B
400	6856B	802B	802B
410	4941B	411B	411B
420	6335B	844B	844B
430	8225B	431B	431B
440	1769B	882B	441B
450	18530B	451B	451B

of text data. For these three types of experiments, we have used different sizes of pattern, i.e., $21 \leq m \leq 35$, where m is the length of pattern. At this point, we can also claim that the more filters we apply, the more space reduction is done.

We have plotted table data of Table 4.3 and Table 4.4 to show the reduction of space in the text in a graphical manner. Figure 4.1 reports the reduction of space in text as well. Here, we have kept values of m in X-axis and values of text size (in Mega-Bytes (MB)) in Y-axis. From Figure 4.1 (a), we can see that, the reduction of space in the text is better when we apply all the 6 filters in a single pass rather than others (4 filters in single pass and 5 filters in a single pass). Again, referring to the same figure we can also see that, when the size of m increases, the reduction of space increases. Figure 4.1 (a) reports the reduction space for $5 \leq m \leq 20$. Subsequently, from Figure 4.1 (b), we can also observe that, the reduction of space is better for applying 6 filters in a single pass as well as for the incrementation of the pattern sizes. Figure 4.1 (b) reports the reduction space for $21 \leq m \leq 35$. From these two sub-figures it has been seen that, huge reduction has been done in the 299MB of text. In Figure 4.1 (a), we see that the reduced text size is less than 1MB whereas in Figure 4.1 (b), we see that the reduced text



(a) A graph representing the Search Space Reduction. Here, $5 \leq m \leq 20$.



(b) A graph representing the Search Space Reduction. Here, $21 \leq m \leq 35$.

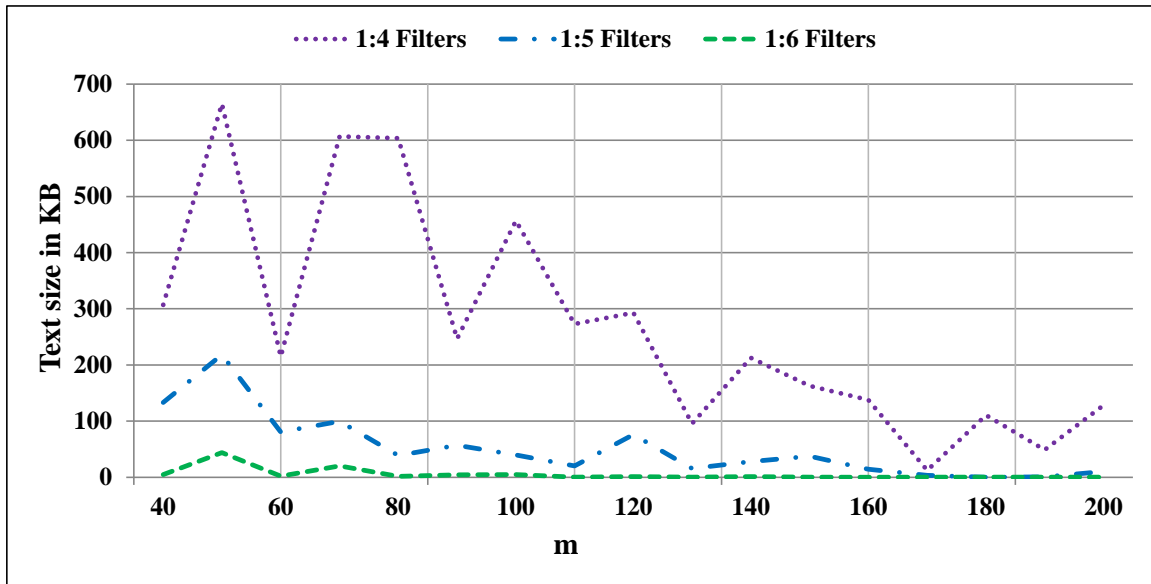
Figure 4.1: Search Space Reduction of text string for $n = 299MB$ and $5 \leq m \leq 35$.

size is less than 0.5MB for applying 6 filters in a single pass for both cases.

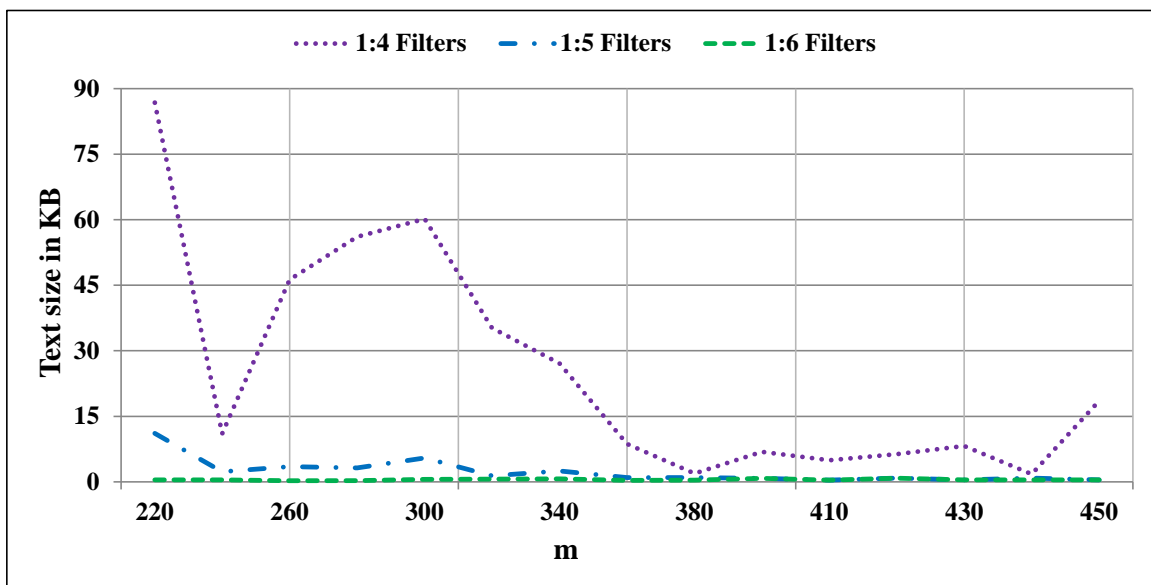
Subsequently, we come to the Table 4.5. Here, the 299MB of text data is reduced to less than 0.5MB of text data applying first four filters (Filter 1 to 4) in a single pass in Algorithm *RSS_FT*. Again, referring to the same table, it can be seen that applying first five filter (Filter 1 to 5) in a single pass in Algorithm *RSS_FT*, the 299MB of text data is reduced to less than 0.3MB of text data. Finally, we can see from the same table that, applying all filters (Filter 1 to 6) in a single pass in our Algorithm *RSS_FT*, the 299MB of text data is reduced to less than 0.1MB of text data. For these three types of experiments, we have used different sizes of pattern, i.e., $40 \leq m \leq 200$, where m is the length of pattern. Again, we can also claim that the more filters we apply, the more space reduction is done.

Finally, in Table 4.6, we can see that the *299MB* of text is reduced to less than *0.1MB* of text data applying first four filters (Filter 1 to 4) in a single pass and less than *0.02MB* of text data applying first five filters (Filter 1 to 5) in a single pass in Algorithm *RSS_FT*. It is worth noting that the *299MB* of text data has been reduced to only *bytes* of data applying all the filters in a single pass. Here, we have used different pattern sizes, $200 \leq m \leq 450$.

Again, we have plotted table data of Table 4.5 and Table 4.6 to show the reduction of space in the text in a graphical manner. Figure 4.2 reports the reduction of space in text as well. Here, we have kept values of m in X-axis and values of text size (in Kilo-Bytes (KB)) in Y-axis. From Figure 4.2 (a), we can see that, the reduction of space in the text is better when we apply all the 6 filters in a single pass rather than others (4 filters in single pass and 5 filters in a single pass). Again, referring to the same figure we can also see that, when the size of m increases, the reduction of space increases as well. Figure 4.2 (a) reports the reduction space for $40 \leq m \leq 200$. Subsequently, from Figure 4.2 (b), we can also observe that, the reduction of space is better for applying 6 filters in a single pass as well as for the incrementation of the pattern sizes. Figure 4.2 (b) reports the reduction space for $220 \leq m \leq 450$. From these two sub-figures it has been seen that, huge reduction has been done in the 299MB of text. In Figure 4.2 (a), we see that the reduced text size is less than 500KB whereas in Figure 4.2 (b), we see that the reduced text size is less than 1KB (almost in Bytes) for applying 6 filters in a single pass for both cases.



(a) A graph representing the Search Space Reduction. Here, $40 \leq m \leq 200$.



(b) A graph representing the Search Space Reduction. Here, $220 \leq m \leq 450$.

Figure 4.2: Search Space Reduction of text string for $n = 299MB$ and $40 \leq m \leq 450$.

By analyzing above four tables (Table 4.3 to Table 4.6) and their respective figures (Figure 4.1 to Figure 4.2) , we can summarize that, the more filters we apply, the more space in text is reduced. Again, it is also clear that as the pattern size increases, the reduction in search space in the text increases as well.

Conjecture 1: The more filters we apply, the more space in text is reduced.

Conjecture 2: The more pattern size we increase, the more space in the text is reduced.

Table 4.7: Elapsed-time (in seconds) and comparisons among KMP, BM, Filtered-KMP, Filtered-BM, Only Filters, KMP in reduced text and BM in reduced text for a text of size 299MB.

m	Elapsed Time(s) of KMP [26]	Elapsed Time(s) of BM [27]	Elapsed Time(s) of Filtered-KMP	Elapsed Time(s) of Filtered-BM	Elapsed Time(s) of Only Fil- ters	Elapsed Time(s) of KMP in re- duced Text	Elapsed Time(s) of BM in re- duced Text
500	2.32	6.79	13.42	13.42038746	13.42	1×10^{-9}	1×10^{-8}
550	2.28	6.75	13.21	13.21033443	13.21	1×10^{-9}	1×10^{-8}
600	2.18	6.65	13.205	13.20539846	13.20	1×10^{-9}	1×10^{-8}
650	2.285	6.75	13.19	13.19034698	13.19	1×10^{-9}	1×10^{-8}
700	2.35	6.82	13.19	13.19034587	13.19	1×10^{-9}	1×10^{-8}
750	2.335	6.80	13.195	13.19533487	13.19	1×10^{-9}	1×10^{-8}
800	2.095	6.56	13.195	13.19539845	13.19	1×10^{-9}	1×10^{-8}
850	2.02	6.49	13.2	13.20039284	13.20	1×10^{-9}	1×10^{-8}
900	2.21	6.68	13.195	13.19539847	13.19	1×10^{-9}	1×10^{-8}
950	2.207	6.67	13.2	13.20036592	13.20	1×10^{-9}	1×10^{-8}
1000	2.335	6.80	13.2	13.20039844	13.20	1×10^{-9}	1×10^{-8}
1600	2.285	6.75	13.205	13.20539586	13.20	1×10^{-9}	1×10^{-8}
1650	2.325	6.79	13.235	13.23584645	13.23	1×10^{-9}	1×10^{-8}
1700	2.18	6.65	13.255	13.25530934	13.25	1×10^{-9}	1×10^{-8}
1750	2.305	6.77	13.255	13.25530955	13.25	1×10^{-9}	1×10^{-8}
1800	2.13	6.60	13.26	13.26038746	13.26	1×10^{-9}	1×10^{-8}
1850	2.11	6.58	13.26	13.26038444	13.26	1×10^{-9}	1×10^{-8}

1900	2.315	6.78	13.265	13.26537445	13.26	1×10^{-9}	1×10^{-8}
1950	2.225	6.69	13.25	13.25038746	13.25	1×10^{-9}	1×10^{-8}
2000	2.185	6.65	13.245	13.24538755	13.24	1×10^{-9}	1×10^{-8}
2050	2.28	6.75	13.25	13.25035567	13.25	1×10^{-9}	1×10^{-8}
2100	2.15	6.62	13.25	13.25035656	13.25	1×10^{-9}	1×10^{-8}
2150	2.165	6.63	13.255	13.25535677	13.25	1×10^{-9}	1×10^{-8}
2200	2.095	6.56	13.25	13.25035667	13.25	1×10^{-9}	1×10^{-8}
2250	2.365	6.83	13.255	13.25538746	13.25	1×10^{-9}	1×10^{-8}
2300	2.165	6.63	13.36	13.36039978	13.36	1×10^{-9}	1×10^{-8}
2350	2.3	6.77	13.255	13.25533455	13.25	1×10^{-9}	1×10^{-8}
2400	2.355	6.82	13.25	13.2505677	13.25	1×10^{-9}	1×10^{-8}
2450	2.28	6.75	13.485	13.4853566	13.48	1×10^{-9}	1×10^{-8}
2500	2.31	6.78	13.26	13.26036789	13.26	1×10^{-9}	1×10^{-8}
2550	2.21	6.68	13.26	13.26038234	13.26	1×10^{-9}	1×10^{-8}
2600	2.145	6.61	13.25	13.25038667	13.25	1×10^{-9}	1×10^{-8}
2650	2.14	6.61	13.26	13.26030923	13.26	1×10^{-9}	1×10^{-8}
2700	2.345	6.81	13.257	13.25739823	13.26	1×10^{-9}	1×10^{-8}
2750	2.32	6.79	13.24	13.24031256	13.24	1×10^{-9}	1×10^{-8}
2800	2.15	6.62	13.24	13.24038112	13.24	1×10^{-9}	1×10^{-8}
2850	2.155	6.62	13.245	13.24530923	13.24	1×10^{-9}	1×10^{-8}
2900	2.21	6.68	13.285	13.28538012	13.28	1×10^{-9}	1×10^{-8}
2950	2.205	6.67	13.242	13.24238233	13.24	1×10^{-9}	1×10^{-8}
3000	2.18	6.65	13.245	13.24539922	13.24	1×10^{-9}	1×10^{-8}

4.5 Experimental Results for Classical Pattern Matching

In this section we discuss the experimental result of our Filtered-KMP and Filtered-BM algorithms. We compare the results of our algorithm with actual KMP and BM algorithms of

[26, 27], respectively. For this experiment we have used environment & experimental settings: **Setup 1** (see description in Section 4.2).

We have implemented Filtered-KMP and Filtered-BM algorithms and conducted extensive experiments to analyze their performance. We show the comparison based on the experimental result among actual KMP and actual BM algorithms of [26, 27] and our algorithms, namely Filtered-KMP and Filtered-BM, respectively. All the four algorithms have been implemented in the *JAVA* programming platform and developed under *Java Development Kit 7* (jdk) in *Linux*. All four algorithms mentioned above take as input the pattern \mathcal{P} of length m , the text \mathcal{T} of length n and returns the indexes all occurrences of \mathcal{P} in \mathcal{T} .

Again, as has been mentioned before, we have used real genome data in our classical pattern matching experiments as the text string, \mathcal{T} . Here, we have taken *299MB* of data for our experiments. Again, we have generated random patterns of different length by a random indexing technique in this *299MB* of text string.

Here we represent the experimental results and comparisons among KMP [26], BM [27] and our algorithms Filtered-KMP and Filtered-BM. Table 4.4 reports the elapsed time and speed-up comparisons for various pattern sizes ($500 \leq m \leq 3000$). It has been seen from Table 4.8, original KMP [26], BM [27] runs faster than our algorithms, Filtered-KMP and Filtered-BM in all cases. To get a better analysis, we also report the required time to run only the filters in our algorithms, Filtered-KMP and Filtered-BM respectively. Then, we report the running time of KMP and BM algorithms in only the reduced text after filtered have been done. It can also be seen from Table 4.8, the running time of KMP and BM algorithms in the reduced text are much smaller than that of the running time of KMP and BM algorithms in the actual text. In fact, the running time is almost *ZERO* in each cases. It is actually 1×10^{-9} seconds for KMP algorithm and 1×10^{-8} seconds for BM algorithm.

4.6 Comparison with Algorithms for ECPM

In this section, we discuss the experimental results our algorithm with the state of art algorithms. Here, we compare our algorithm, Filtered-ECPM with the most recent state of art

algorithms of [1, 3]. For this experiment we have used the environmental & experimental settings: **Setup 2** (see description in Section 4.2).

Table 4.8: Elapsed-time (in seconds) and speed-up comparisons among FredNava [3], ACSMF-SimpleZeroK [1] and Filtered-ECPM on a text of size 299MB

m	Elapsed Time(s) of FredNava	Elapsed Time(s) of ACSMF- simpleZeroK	Elapsed Time(s) of Filtered-ECPM	Speed up: FredNava vs. Filtered-ECPM	Speed up: ACSMF- simpleZeroK vs. Filtered-ECPM
500	11.67	5.938	1.167	10	5
550	18.928	7.914	1.456	13	5
600	40.92	7.691	1.364	30	6
650	13.078	7.836	1.006	13	8
700	15.42	7.739	1.028	15	8
750	37.555	7.82	1.073	35	7
800	417.04	7.839	1.04	401	8
850	100.225	8.382	1.055	95	8
900	25.56	7.646	1.278	20	6
950	35.05	7.876	1.402	25	6
1000	54.72	7.731	1.216	45	6
1600	1186.728	7.334	1.182	1004	6
1650	100.776	8.239	0.969	104	9
1700	2365.9	7.572	1.18	2005	6
1750	805.376	5.968	1.144	704	5
1800	819.405	7.551	1.179	695	6
1850	105.342	7.407	1.086	97	7
1900	50.67	7.861	1.126	45	7
1950	97.66	7.339	1.028	95	7
2000	4584.918	7.814	1.118	4101	7
2050	399.588	5.969	1.988	201	3
2100	1071.861	5.173	1.187	903	4
2150	174.629	5.317	1.919	91	3
2200	158.014	6.032	1.927	82	3
2250	104.225	5.009	1.895	55	3

2300	126.697	5.029	1.891	67	3
2350	118.881	5.041	1.887	63	3
2400	87.86	6.036	1.91	46	3
2450	50.922	6.04	1.886	27	3
2500	69.16	7.046	1.976	35	4
2550	1394.874	7.042	1.987	702	4
2600	1741.332	8.043	2.883	604	3
2650	106.708	8.049	2.884	37	3
2700	274.74	8.031	2.892	95	3
2750	95.106	8.039	2.882	33	3
2800	72.15	9.026	2.886	25	3
2850	46.416	9.154	2.901	16	3
2900	241.318	10.049	3.134	77	3
2950	244.188	11.044	3.876	63	3
3000	163.8	12.044	3.9	42	3

As has been mentioned already before, we have implemented Filtered-ECPM and conducted extensive experiments to analyze its performance. We have coded Filtered-ECPM in C++ using a GNU compiler with General Public License (GPL). Our code is available at [57]. Recall that, our implementation of Filtered-ECPM uses the ACSMF-SimpleZero k [1] and FredNava [3]. ACSMF-Simple [1] has been implemented as library functions in the C programming language under GNU/Linux operating system. The library implementation is distributed under the GNU General Public License (GPL). It takes as input the pattern \mathcal{P} of length m , the text \mathcal{T} of length n , and the integer threshold $k < m$ and returns the list of starting positions of the occurrences of the rotations of \mathcal{P} in \mathcal{T} with k -mismatches as output. In our case we use $k = 0$. Again, the Algorithm FredNava of [3] has been implemented in C programming language for our performance measurement.

Table 4.6 reports the elapsed time and speed-up comparisons for various pattern sizes ($500 \leq m \leq 3000$). As can be seen from Table 4.6, Filtered-ECPM runs much faster than *FredNava* in all cases. And in fact Filtered-ECPM achieves a minimum of ten-fold speed-up over FredNava for all the pattern sizes. Again, referring to the same table, Filtered-ECPM runs

even faster than ACSMF-SimpleZero k (most recent efficient algorithm of [1]) in all cases. And in fact Filtered-ECPM achieves a minimum of three-fold speed-up over ACSMF-SimpleZero k for all the pattern sizes.

We have plotted a graph for the values of Table 4.6 to show the comparisons among the algorithms in a graphical manner. Figure 4.3 reports the elapsed-time (in seconds) and speed-up comparisons among the algorithms. Here, we have kept values of m in X-axis and elapsed-time (running time) values of each algorithms in Y-axis. Here, note that, for better & precise representation of graph, we have converted the elapsed-time values into \log (10-based \log) values. From Figure 4.3, we can see that, the elapsed time for ACSMF-SimpleZero k is less than the elapsed time of FredNava for all values of m . Again, referring to the same figure we can also see that, the elapsed time for Filtered-ECPM is less than the elapsed time for ACSMF-SimpleZero k for all pattern sizes. Figure 4.3, reports elapsed time for $40 \leq m \leq 200$.

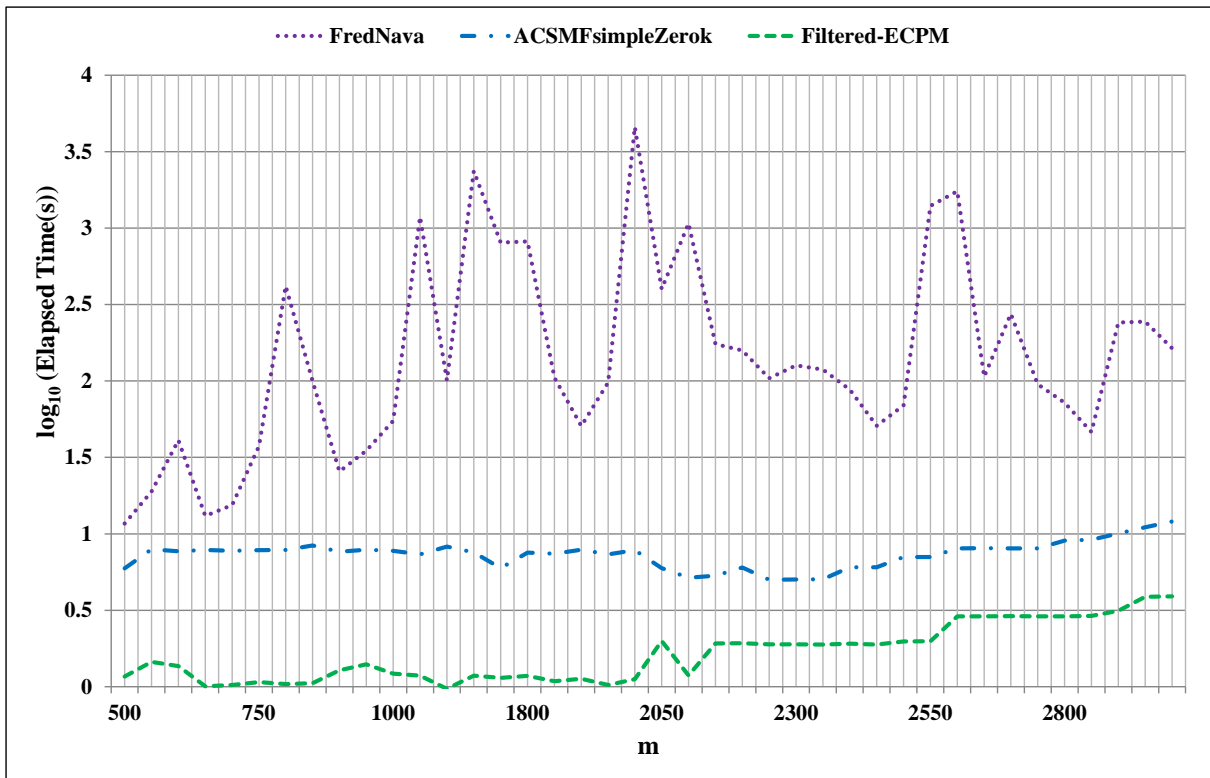


Figure 4.3: A graph representing elapsed-time (in seconds) and speed-up comparisons among FredNava [3], ACSMF-SimpleZero k [1] and Filtered-ECPM on a text of size 299MB

In order to analyze and understand the effect of our filters we have run a second set of

experiments for Filtered-ECPM algorithm as follows. We have run experiments on three variants of Filtered-ECPM where the first variant (Filtered-ECPM-[1..3]) only employs Filters 1 (in Chapter 2, Section 2.9.1) through 3 (in Chapter 2, Section 2.9.2), the second variant (Filtered-ECPM-[1..4]) only employs Filters 1 (in Chapter 2, Section 2.9.1) through 4 (in Chapter 2, Section 2.9.3), and finally the third variant (Filtered-ECPM-[1..5]) employs Filters 1 (in Chapter 2, Section 2.9.1) through 5 (in Chapter 2, Section 2.9.4). Table 4.9 reports the elapsed time and speed-up comparisons considering various pattern sizes ($500 \leq m \leq 2000$) for ACSMF-SimpleZero k and the above-mentioned three variants of Filtered-ECPM. As can be seen from Table 4.9, ACSMF-SimpleZero k is able to beat Filtered-ECPM-[1..3] in a number of cases. However, Filtered-ECPM-[1..4] and Filtered-ECPM-[1..5] run significantly faster than ACSMF-SimpleZero k in all cases. This indicates that as more and more effective filters are imposed, Filtered-ECPM algorithm performs better.

Table 4.9: Elapsed-time (in seconds) and speed-up comparisons among ACSMF-SimpleZero*k* and three variants of Filtered-ECPM (considering different combination of the filters) for a text of size 299MB.

m	Filters 1 to 3			Filters 1 to 4			Filters 1 to 5		
	Elapsed Time(s) of ACSMF-simpleZero <i>k</i>	Elapsed Time(s) of Filtered-ECPM-[1..3]	Speed up: ACSMF-simpleZero <i>k</i> vs. Filtered-ECPM-[1..3]	Elapsed Time(s) of ACSMF-simpleZero <i>k</i>	Elapsed Time(s) of Filtered-ECPM-[1..4]	Speed up: ACSMF-simpleZero <i>k</i> vs. Filtered-ECPM-[1..4]	Elapsed Time(s) of ACSMF-simpleZero <i>k</i>	Elapsed Time(s) of Filtered-ECPM-[1..5]	Speed up: ACSMF-simpleZero <i>k</i> vs. Filtered-ECPM-[1..5]
500	6.355	3.522	2	6.373	4.973	1	6.397	2.523	3
550	8.526	20.43	0	8.564	4.866	2	8.38	2.545	3
600	8.149	43.544	0	8.286	4.902	2	8.359	2.518	3
650	8.315	4.35	2	8.448	4.894	2	8.324	2.47	3
700	9.063	7.596	1	8.71	4.9	2	8.249	2.493	3
750	8.399	6.837	1	8.643	5.101	2	8.326	2.478	3
800	8.357	16.293	1	8.346	4.915	2	8.265	2.48	3
850	8.79	10.651	1	8.309	4.924	2	8.48	2.562	3
900	7.959	23.181	0	8.411	4.916	2	8.223	2.525	3
950	8.652	15.443	1	8.552	4.93	2	8.678	2.519	3
1000	8.285	12.399	1	8.371	4.916	2	8.375	2.616	3
1600	7.846	6.074	1	7.927	4.915	2	7.872	2.529	3
1650	8.918	2.691	3	8.878	4.904	2	8.854	2.523	4
1700	7.839	6.506	1	7.697	4.897	2	7.8	2.522	3
1750	6.252	30.173	0	6.523	5.09	1	6.399	2.526	3
1800	8.643	26.655	0	8.218	4.918	2	8.143	2.487	3
1850	8.072	2.901	3	8.026	4.901	2	8.095	2.532	3
1900	8.442	30.468	0	8.495	4.927	2	8.297	2.516	3
1950	8.123	2.542	3	8.367	4.927	2	7.951	2.495	3
2000	8.366	12.175	1	8.58	5.13	2	8.394	2.533	3

4.7 Experimental Results for ACPM

In this section, we discuss the experimental result of our Filtered-ACPM algorithm. We compare the results of our algorithm with the most recent state of art algorithm of [1, 3]. For this experiment we have also used the environment and experimental settings: **Setup 1** (see description in Section 4.2).

Table 4.10: Elapsed-time (in seconds) and speed-up comparison among FredNava [3], ACSMF-Simple [1] and Filtered-ACPM considering all the six filters in a single pass for a text of size 1GB

m	k	Elapsed Time(s) of FredNava	Elapsed Time(s) of ACSMF-simple	Elapsed Time(s) of Filtered-ACPM	Speed up: FredNava vs Filtered-ACPM	Speed up: ACSMF-Simple vs Filtered-
2000	2	144	33	16	9	2.1
3000	2	210	28	14	15	2
4000	2	75	32	15	5	2.1
5000	5	108	40	18	6	2.2
6000	5	180	36	15	12	2.4
2000	5	187	38	17	11	2.2
3000	5	266	40	19	14	2.1
4000	5	240	28	12	20	2.3
5000	5	500	40	20	25	2
6000	5	675	36	15	45	2.4
5000	7	1100	44	20	55	2.2
6000	7	924	64	28	33	2.3
7000	7	575	59	25	23	2.4
8000	7	768	63	32	24	2
9000	7	297	59	27	11	2.2
10000	10	360	46	20	18	2.3
50000	10	380	47	20	19	2.4
100000	10	598	100	46	13	2.2

150000	10	672	105	56	12	1.9
200000	10	2046	122	62	33	2
250000	12	2835	126	63	45	2
300000	12	1704	138	71	24	1.9
350000	12	3525	145	75	47	1.9
400000	12	4104	155	72	57	2.2
450000	12	3952	170	76	52	2.2
500000	12	5184	200	81	64	2.5
550000	15	2200	251	100	22	2.5
650000	15	1440	270	120	12	2.3
700000	15	3384	292	141	24	2.1
750000	15	7740	330	172	45	1.9
800000	15	2316	401	193	12	2.1
850000	15	5522	498	251	22	2
900000	20	3133	561	241	13	2.3
950000	20	6622	706	301	22	2.3
1000000	20	4823	870	371	13	2.3
1100000	20	3608	901	451	8	2
1150000	20	8496	931	531	16	1.8
1200000	20	9384	955	552	17	1.7

As has been mentioned already before that, we have implemented Filtered-ACPM algorithm and conducted extensive experiments to analyze its performance. We have coded Filtered-ACPM in C++ using a GNU compiler with General Public License (GPL). Our code is available at [57]. Again, our implementation of ACPM uses the Algorithm ACSMF-Simple [1] and Algorithm FredNava [3]. We have already described the implementation procedure of ACSMF-Simple and FredNava in Section 4.6. These algorithm take as input the pattern \mathcal{P} of length m , the text \mathcal{T} of length n , and the integer threshold $k < m$ and returns the list of starting positions of the occurrences of the rotations of \mathcal{P} in \mathcal{T} with k -mismatches as output.

Again, as has been mentioned before, we have used real genome data as the text string, \mathcal{T} in our experiments to conduct algorithm Filtered-ACPM. Here, we have taken *1GB* of data for our experiments. Again, we have generated random patterns of different length by a random indexing technique in this *1GB* of text string.

Here we report the experimental results and comparisons among Filtered-ACPM, ACSMF-Simple of [1] and *FredNava* [3]. Table 4.7 reports the elapsed time and speed-up comparisons for various pattern sizes ($2000 \leq m \leq 1200000$) and for various mismatch sizes ($2 \leq k \leq 20$). As can be seen from Table 4.7, our algorithm runs much faster than *FredNava* in all cases. And in fact Filtered-ACPM achieves a minimum of five-fold speed-up for all the pattern sizes. Again, referring to the same table, Filtered-ACPM runs even faster than ACSMF-Simple (most recent efficient algorithm of [1]) in all cases.

We have plotted a graph for the values of Table 4.7 to show the comparisons among the algorithms in a graphical manner. Figure 4.4 reports the elapsed-time (in seconds) and speed-up comparisons among the algorithms. Here, we have kept values of m (pattern size) in X-axis, number of mismatches k in Y-axis and elapsed-time (running time) values of each algorithms in Z-axis. Here, note that, for better & precise representation of graph, we have converted the elapsed-time values into \log (10-based \log) values. From Figure 4.4, we can see that, the elapsed time for ACSMF-Simple is less than the elapsed time of *FredNava* for all values of m and k . Again, referring to the same figure we can also see that, the elapsed time for Filtered-ACPM is less than the elapsed time for ACSMF-Simple for all pattern sizes and mutations. Figure 4.4, reports elapsed time for $2000 \leq m \leq 1200000$ and $2 \leq k \leq 20$.

Table 4.11: Elapsed-time (in seconds) and speed-up comparisons among ACSMF-Simple and Filtered-ACPM-[1..3] (considering first three combination of the filters) for a text of size 1GB

m	k	Elapsed Time(s) of ACSMF-simple	Elapsed Time(s) of Filtered-ACPM[1..3]	Speed up of Filtered-ACPM
2000	2	33	26	1.3
3000	2	28	26	1.1
4000	2	32	26	1.2

5000	5	40	24	1.7
6000	5	36	24	1.5
2000	5	38	24	1.6
3000	5	40	24	1.7
4000	5	28	24	1.2
5000	5	40	24	1.7
6000	5	36	24	1.5
5000	7	44	30	1.5
6000	7	64	27	2.4
7000	7	59	27	2.2
8000	7	63	27	2.3
9000	7	59	27	2.2
10000	10	46	26	1.8
50000	10	47	71	0.7
100000	10	100	58	1.7
150000	10	105	758	0.1
200000	10	122	537	0.2
250000	12	126	453	0.3
300000	12	138	101	1.4
350000	12	145	272	0.5
400000	12	155	335	0.5
450000	12	170	450	0.4
500000	12	200	311	0.6
550000	15	251	506	0.5
650000	15	270	647	0.4
700000	15	292	1126	0.3
750000	15	330	1214	0.3
800000	15	401	1350	0.3
850000	15	498	1580	0.3
900000	20	561	1289	0.4
950000	20	706	894	0.8
1000000	20	870	259	3.4

1100000	20	901	1329	0.7
1150000	20	931	1636	0.6
1200000	20	955	1203	0.8

Table 4.12: Elapsed-time (in seconds) and speed-up comparisons among ACSMF-Simple and Filtered-ACPM-[1..4] (considering first four combination of the filters) for a text of size 1GB

m	k	Elapsed Time(s) of ACSMF-simple	Elapsed Time(s) of Filtered-ACPM[1..4]	Speed up of Filtered-ACPM
2000	2	33	17	1.9
3000	2	28	16	1.8
4000	2	32	13	2.5
5000	5	40	14	2.9
6000	5	36	14	2.6
2000	5	38	10	3.8
3000	5	40	14	2.9
4000	5	28	11	2.5
5000	5	40	12	3.3
6000	5	36	14	2.6
5000	7	44	22	2.0
6000	7	64	35	1.8
7000	7	59	28	2.1
8000	7	63	23	2.7
9000	7	59	26	2.3
10000	10	46	27	1.7
50000	10	47	44	1.1
100000	10	100	50	2.0
150000	10	105	70	1.5

200000	10	122	100	1.2
250000	12	126	102	1.2
300000	12	138	90	1.5
350000	12	145	101	1.4
400000	12	155	100	1.6
450000	12	170	120	1.4
500000	12	200	99	2.0
550000	15	251	100	2.5
650000	15	270	130	2.1
700000	15	292	150	1.9
750000	15	330	220	1.5
800000	15	401	205	2.0
850000	15	498	350	1.4
900000	20	561	330	1.7
950000	20	706	450	1.6
1000000	20	870	430	2.0
1100000	20	901	501	1.8
1150000	20	931	600	1.6
1200000	20	955	700	1.4

Table 4.13: Elapsed-time (in seconds) and speed-up comparisons among ACSMF-Simple and Filtered-ACPM-[1..5] (considering first five combination of the filters) for a text of size 1GB

m	k	Elapsed Time(s) of ACSMF-simple	Elapsed Time(s) of Filtered-ACPM[1..5]	Speed up of Filtered-ACPM
2000	2	33	15	2.2
3000	2	28	17	1.6
4000	2	32	13	2.5

5000	5	40	20	2.0
6000	5	36	17	2.1
2000	5	38	15	2.5
3000	5	40	19	2.1
4000	5	28	14	2.0
5000	5	40	17	2.4
6000	5	36	20	1.8
5000	7	44	20	2.2
6000	7	64	30	2.1
7000	7	59	25	2.4
8000	7	63	20	3.2
9000	7	59	30	2.0
10000	10	46	25	1.8
50000	10	47	45	1.0
100000	10	100	52	1.9
150000	10	105	69	1.5
200000	10	122	78	1.6
250000	12	126	90	1.4
300000	12	138	93	1.5
350000	12	145	140	1.0
400000	12	155	102	1.5
450000	12	170	72	2.4
500000	12	200	101	2.0
550000	15	251	120	2.1
650000	15	270	140	1.9
700000	15	292	145	2.0
750000	15	330	170	1.9
800000	15	401	207	1.9
850000	15	498	331	1.5
900000	20	561	341	1.6
950000	20	706	433	1.6
1000000	20	870	542	1.6

1100000	20	901	600	1.5
1150000	20	931	605	1.5
1200000	20	955	650	1.5

In order to analyze and understand the effect of our filters we have run a second set of experiments on Filtered-ACPM and ACSMF-simple as follows. We have run experiments on three variants of Filtered-ACPM algorithm where the first variant (Filtered-ACPM-[1..3]) only employs Filters 1 (in Chapter 2, Section 2.10.1) through 3 (in Chapter 2, Section 2.10.2), the second variant (Filtered-ACPM-[1..4]) only employs Filters 1 (in Chapter 2, Section 2.10.1) through 4 (in Chapter 2, Section 2.10.3), and finally the third variant (Filtered-ACPM-[1..5]) employs Filters 1 (in Chapter 2, Section 2.10.1) through 5 (in Chapter 2, Section 2.10.4). Tables 4.7, 4.7 and 4.7 report the elapsed time and speed-up comparisons considering various pattern sizes ($2000 \leq m \leq 1200000$) for ACSMF-Simple and the above-mentioned three variants of Filtered-ACPM algorithm. As can be checked from Tables 4.7, 4.7 and 4.7, ACSMF-Simple is able to beat Filtered-ACPM-[1..3] in a number of cases. However, Filtered-ACPM-[1..4] and Filtered-ACPM-[1..5] run significantly faster than ACSMF-Simple of [1] in all cases. This also indicates that as more and more effective filters are imposed, Filtered-ACPM algorithm performs better.

4.8 Experimental Results for CSC

In this section we discuss the experimental result of our Algorithm, Filtered-CSC. We compare the results of our algorithm with most recent state of art algorithm of [2]. For this experiment we have used environment & experimental settings: **Setup 3** (see description in Section 4.2).

We have implemented Filtered-CSC and conducted extensive experiments to analyze its performance. We show the comparison based on the experimental results between Algorithm saCSC of [2] and our Algorithm Filtered-CSC. Algorithm saCSC [2] has been implemented as library functions in the *C* programming language under *GNU/Linux* operating system. The

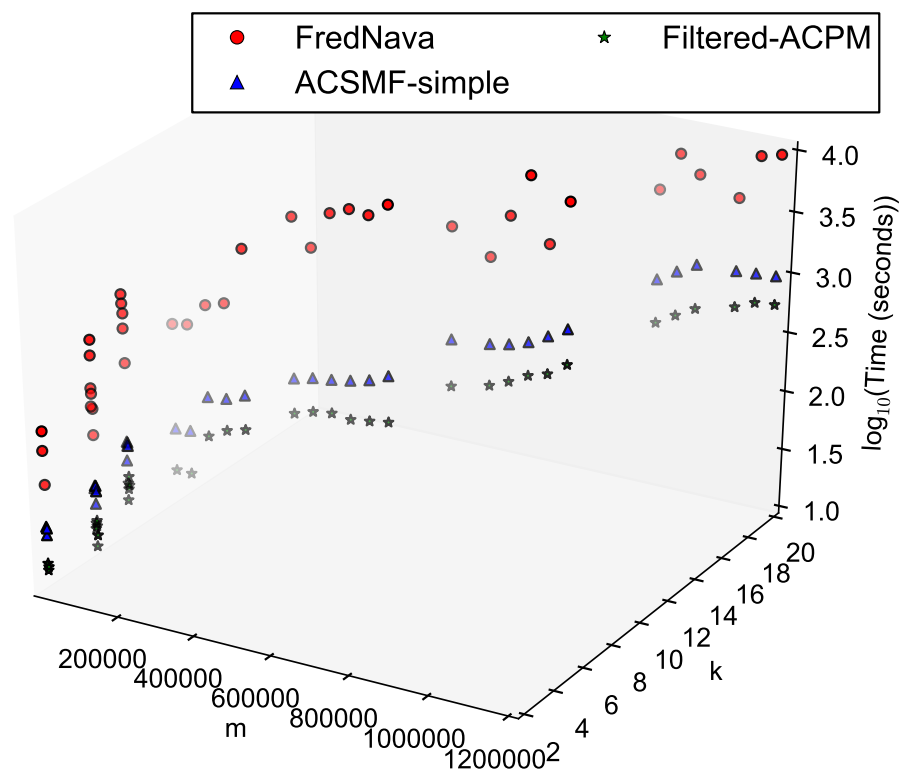


Figure 4.4: A graph representing elapsed-time (in seconds) and speed-up comparison among FredNava [3], ACSMF-Simple [1] and Filtered-ACPM considering all the six filters in a single pass for a text of size 1GB.

library implementation is distributed under the GNU General Public License (GPL). It takes as input the pattern \mathcal{P} of length m , the text \mathcal{T} of length n , integers block-size $\beta > 1$, q -grams $q < m$, and returns the rotation of \mathcal{P} for which the blockwise q -gram distance is minimal with \mathcal{T} according to Problem 4.

Again, as has been mentioned before, we have used real genome data as the text string, \mathcal{T} in our experiments for the Algorithm Filtered-CSC. Here, we have taken *700MB* of data for our experiments. Again, we have generated random patterns of different length by a random indexing technique in this *700MB* of text string.

Table 4.14: Elapsed-time (in seconds) and speed-up comparison between saCSC[2] and Filtered-CSC considering all the six filters for a text of size 700MB. Here, $\beta = \sqrt{m}$.

m	q	Elapsed Time(s) of saCSC	Elapsed Time(s) of Filtered-CSC	Speed up of Filtered-CSC
2000	5	41	22	1.9
3000	5	42	21	2.0
4000	5	45	25	1.8
5000	5	44	23	1.9
6000	5	47	25	1.9
2000	10	39	18	2.2
3000	10	44	20	2.2
4000	10	47	26	1.8
5000	10	52	28	1.9
6000	10	57	26	2.2
5000	15	60	31	1.9
6000	15	63	32	2.0
7000	15	60	29	2.1
8000	15	66	32	2.1
9000	15	67	34	2.0
10000	20	56	27	2.1
50000	20	63	33	1.9
100000	20	111	52	2.1
150000	20	106	51	2.1

200000	20	123	62	2.0
250000	25	129	66	2.0
300000	25	140	69	2.0
350000	25	147	74	2.0
400000	25	160	78	2.1
450000	25	172	81	2.1
500000	30	203	99	2.1
550000	30	255	105	2.4
650000	30	266	123	2.2
700000	30	301	143	2.1
750000	30	333	192	1.7
800000	35	402	195	2.1
850000	35	498	240	2.1
900000	35	578	256	2.3
950000	35	710	280	2.5
1000000	35	880	423	2.1
1100000	40	902	440	2.1
1150000	40	935	510	1.8
1200000	40	1020	550	1.9
1200000	40	1022	530	1.9
1200000	40	1045	501	2.1

Here we report the experimental results and comparisons between our algorithm Filtered-CSC and saCSC of [2]. Table 4.8 reports the elapsed time and speed-up comparisons for various pattern sizes ($2000 \leq m \leq 1200000$) and for various q -grams sizes ($5 \leq q \leq 40$) and block-size $\beta = \sqrt{m}$. We set $\beta = \sqrt{m}$ for better performance of saCSC as has been described in [2]. As can be seen from Table 4.8, our algorithm runs faster than saCSC in all cases. And in fact Filtered-CSC achieves around a two-fold speed-up for all the pattern and q -gram sizes described here.

We have plotted a graph for the values of Table 4.8 to show the comparisons between the

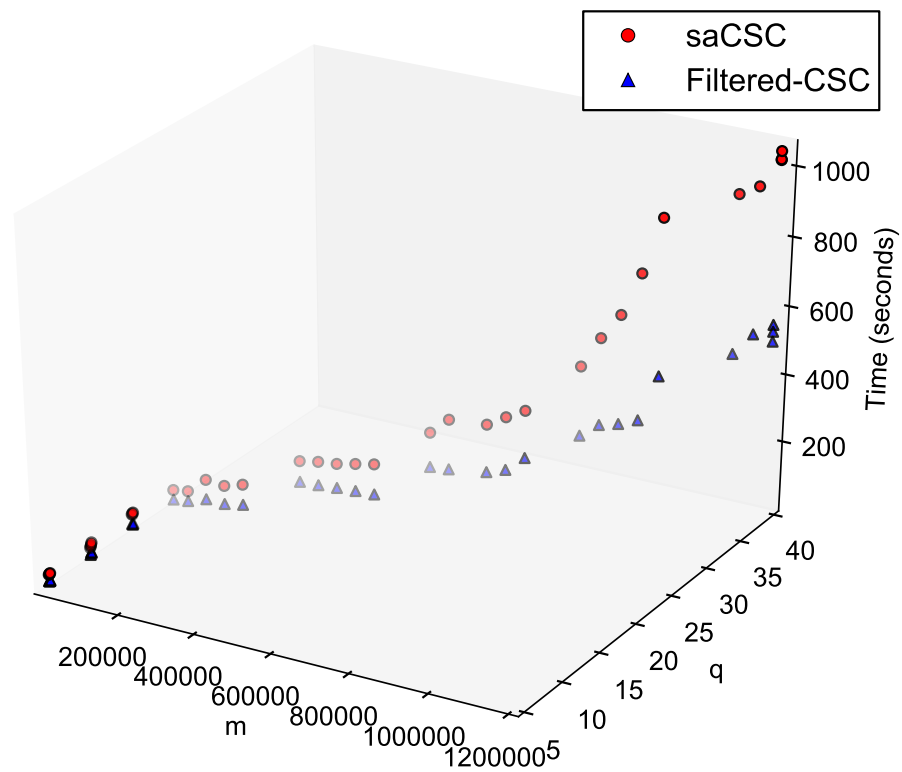


Figure 4.5: A graph representing elapsed-time (in seconds) and speed-up comparison between saCSC[2] and Filtered-CSC considering all the six filters for a text of size 700MB. Here, $\beta = \sqrt{m}$.

algorithms saCSC and Filtered-CSC in a graphical manner. Figure 4.5 reports the elapsed-time (in seconds) and speed-up comparisons among the algorithms. Here, we have kept values of m (pattern size) in X-axis, the size of q -gram q in Y-axis and elapsed-time (running time) values of each algorithms in Z-axis. From Figure 4.5, we can see that, the elapsed time for Filtered-CSC is less than the elapsed time of saCSC for all values of m and q . Figure 4.5, reports elapsed time for $2000 \leq m \leq 1200000$ and $5 \leq q \leq 40$. Here, also note that, we have used $\beta = \sqrt{m}$.

Same as our previous experiments like Filtered-ECPM and Filtered-ACPM, we like to analyze and understand the effect of our filters. For this reason, we have run a second set of experiments as follows. We have run experiments on three variants of Filtered-CSC algorithm where the first variant (Filtered-CSC-[1..3]) only employs Filters 1 (in Chapter 2, Section 2.9.1) through 3 (in Chapter 2, Section 2.9.2), the second variant (Filtered-CSC-[1..4]) only employs Filters 1 (in Chapter 2, Section 2.9.1) through 4 (in Chapter 2, Section 2.9.3), and finally the third variant (Filtered-CSC-[1..5]) employs Filters 1 (in Chapter 2, Section 2.9.1) through 5 (in Chapter 2, Section 2.9.4). Tables 4.8, 4.8 and 4.8 report the elapsed time and speed-up comparisons considering various pattern sizes ($2000 \leq m \leq 1200000$) for saCSC and the above-mentioned three variants of Filtered-CSC algorithm. As can be checked from Tables 4.8, 4.8 and 4.8, saCSC is able to beat Filtered-CSC-[1..3] in a number of cases. However, Filtered-CSC-[1..4] and Filtered-CSC-[1..5] run significantly faster than saCSC of [2] in all cases. Again, this indicates that as more and more effective filters are imposed, Filtered-CSC algorithm performs better.

Table 4.15: Elapsed-time (in seconds) and speed-up comparisons between saCSC and Filtered-CSC-[1..3] (considering first three combination of the filters) for a text of size 700MB. Here, $\beta = \sqrt{m}$.

m	q	Elapsed Time(s) of saCSC	Elapsed Time(s) of Filtered-CSC-[1..3]	Speed up of Filtered-CSC-[1..3]
2000	5	41	32	1.3
3000	5	42	38	1.1
4000	5	45	38	1.2

5000	5	44	26	1.7
6000	5	47	31	1.5
2000	10	39	24	1.6
3000	10	44	26	1.7
4000	10	47	39	1.2
5000	10	52	31	1.7
6000	10	57	38	1.5
5000	15	60	40	1.5
6000	15	63	26	2.4
7000	15	60	27	2.2
8000	15	66	29	2.3
9000	15	67	30	2.2
10000	20	56	31	1.8
50000	20	63	90	0.7
100000	20	111	65	1.7
150000	20	106	1060	0.1
200000	20	123	615	0.2
250000	25	129	430	0.3
300000	25	140	100	1.4
350000	25	147	294	0.5
400000	25	160	320	0.5
450000	25	172	430	0.4
500000	30	203	338	0.6
550000	30	255	510	0.5
650000	30	266	665	0.4
700000	30	301	1003	0.3
750000	30	333	1110	0.3
800000	35	402	1340	0.3
850000	35	498	1660	0.3
900000	35	578	1445	0.4
950000	35	710	888	0.8
1000000	35	880	259	3.4

1100000	40	902	1289	0.7
1150000	40	935	1558	0.6
1200000	40	1020	1275	0.8
1200000	40	1022	530	1.9
1200000	40	1045	501	2.1

Table 4.16: Elapsed-time (in seconds) and speed-up comparisons between saCSC and Filtered-CSC-[1..4] (considering first four combination of the filters) for a text of size 700MB. Here, $\beta = \sqrt{m}$.

m	q	Elapsed Time(s) of saCSC	Elapsed Time(s) of Filtered-CSC-[1..4]	Speed up of Filtered-CSC-[1..4]
2000	5	41	22	1.9
3000	5	42	23	1.8
4000	5	45	18	2.5
5000	5	44	15	2.9
6000	10	47	18	2.6
2000	10	39	10	3.8
3000	10	44	15	2.9
4000	10	47	19	2.5
5000	10	52	16	3.3
6000	15	57	22	2.6
5000	15	60	30	2
6000	15	63	35	1.8
7000	15	60	29	2.1
8000	15	66	24	2.7
9000	20	67	29	2.3
10000	20	56	33	1.7
50000	20	63	57	1.1
100000	20	111	56	2
150000	20	106	71	1.5
200000	25	123	103	1.2

250000	25	129	108	1.2
300000	25	140	93	1.5
350000	25	147	105	1.4
400000	25	160	100	1.6
450000	30	172	123	1.4
500000	30	203	102	2
550000	30	255	102	2.5
650000	30	266	127	2.1
700000	30	301	158	1.9
750000	35	333	222	1.5
800000	35	402	201	2
850000	35	498	356	1.4
900000	35	578	340	1.7
950000	35	710	444	1.6
1000000	40	880	440	2
1100000	40	902	501	1.8
1150000	40	935	584	1.6
1200000	40	1020	729	1.4
1200000	40	1022	530	1.9
1200000	40	1045	501	2.1

Table 4.17: Elapsed-time (in seconds) and speed-up comparisons between saCSC and Filtered-CSC-[1..5] (considering first five combination of the filters) for a text of size 700MB. Here, $\beta = \sqrt{m}$.

m	q	Elapsed Time(s) of saCSC	Elapsed Time(s) of Filtered-CSC-[1..5]	Speed up of Filtered-CSC-[1..5]
2000	5	41	19	2.2
3000	5	42	26	1.6
4000	5	45	18	2.5
5000	5	44	22	2
6000	10	47	22	2.1

2000	10	39	16	2.5
3000	10	44	21	2.1
4000	10	47	24	2
5000	10	52	22	2.4
6000	15	57	32	1.8
5000	15	60	27	2.2
6000	15	63	30	2.1
7000	15	60	25	2.4
8000	15	66	21	3.2
9000	20	67	34	2
10000	20	56	31	1.8
50000	20	63	63	1
100000	20	111	58	1.9
150000	20	106	71	1.5
200000	25	123	77	1.6
250000	25	129	92	1.4
300000	25	140	93	1.5
350000	25	147	147	1
400000	25	160	107	1.5
450000	30	172	72	2.4
500000	30	203	102	2
550000	30	255	121	2.1
650000	30	266	140	1.9
700000	30	301	151	2
750000	35	333	175	1.9
800000	35	402	212	1.9
850000	35	498	332	1.5
900000	35	578	361	1.6
950000	35	710	444	1.6
1000000	40	880	550	1.6
1100000	40	902	601	1.5
1150000	40	935	623	1.5

1200000	40	1020	680	1.5
1200000	40	1022	530	1.9
1200000	40	1045	501	2.1

4.9 Summary

In this chapter, we have presented the experimental study of all problems mentioned in this thesis. We have shown our experimental setup, environment of the program and computer as well as the comparisons among all state of art algorithms and our proposed algorithms. We have seen that our proposed algorithms runs much faster than the state of art algorithm for Problem 2 to 4. But, for Problem 1, it has been seen that the state of art algorithm runs better than ours in all cases.

Chapter 5

Conclusion

Circular molecular structures are abundant in nature. They can be composed of both amino and nucleic acids. Finding the circular molecular structures in genome sequences and/or alignment of circular sequences with genome sequences are common and important task in Computational Biology. Traditional algorithms are computationally too expensive in this regard. We have presented a new algorithmic framework based on filtering techniques to solve these problems efficiently and effectively. We have reported experimental results that demonstrate superior efficiency of our algorithms relative to the state of art algorithms.

In this study, we have employed some effective lightweight filtering techniques to reduce the search space of the following problems: Classical Pattern Matching problem, Exact Circular Pattern Matching (ECPM) problem, Approximate Circular Pattern Matching (ACPM) problem and Circular Sequence Comparison (CSC) problem. We have used a concept of a pattern signature for both linear and circular strings. Then we have proposed *RSS_FT* algorithm for search space reduction in the text. We have presented Filtered-ECPM and Filtered-ACPM algorithms, both are extremely fast algorithm based on the above-mentioned filters. Subsequently, we have presented Filtered-CSC algorithm, also a fast algorithm than the state of art algorithm. Initially, we have presented a filter based algorithm for classical pattern matching problem. Though, the filter based classical algorithm is not that much fast than the state of art algorithm, using a comparative analysis we have showed that our filtering technique works effectively and efficiently. Hence, much of the speed of our algorithms comes from the fact that

our filters are effective but extremely simple and lightweight. The most intriguing feature of our algorithmic framework is perhaps its capability to plug in any algorithm to solve all the actual problems (Problems 1 to Problem 4) and take advantage of it.

The contributions that have been made in this thesis can be enumerated as follows.

- We have presented the concept of a pattern signature for both linear and circular strings using filtering techniques. This pattern signature has been used for the reduction of search space in the text string. This pattern signature Algorithm *PSF_FT* runs in $\mathcal{O}(m)$ time, where m is the length of pattern.
- We have presented a framework for search space reduction. This framework uses the concept of the pattern signature described above to reduce the search space of the text string. The procedure *RSS_FT* of this framework runs in $\mathcal{O}(n)$ time, where n is the length of text string.
- We have presented a filter-based algorithm for classical pattern matching. Though, our filter-based algorithm runs slower than the state of art, we showed a comparative analysis behind this slowness and we have shown the effectiveness of our filters.
- We have presented Filtered-ECPM algorithm which runs faster than the state of the art algorithm. In fact, our Filtered-ECPM achieves a minimum of three-fold speed-up than the state of art [1].
- We have also presented Filtered-ACPM algorithm which also runs faster than the state of the art algorithm. In fact, our Filtered-ACPM achieves almost two-fold speed-up than the state of art [1].
- Furthermore, we have presented a filter-based algorithm for exact circular sequence comparison. Our Filtered-CSC also runs faster than the state of art [2]. In fact, our Filtered-CSC achieves a two-fold speed-up than the state of art [2].

5.1 Future Works

In this study, we have focused on developing effective and efficient approaches to solve ECPM, ACPM and CSC problems. The effectiveness and the efficiency of our approaches are gained by using extremely light-weight filtering techniques. We have implemented our solutions in C programming language as a library. At this point, our primary future task is to develop a web-based online tool which will be used throughout the world. Besides this, some other issues need to be considered to enhance our current contribution. They are listed below.

- To develop a new filter-based algorithm of Heuristic (approximate) Circular Sequence Comparison (HCSC).
- To improve our filtering techniques to solve multiple circular sequence alignment.
- We will explore the possibility of optimising our filters and the corresponding library implementation for the both exact and approximate CPM case by using lossless filters for eliminating a possibly large fraction of the input that is guaranteed not to contain any exact/approximate occurrence.

Bibliography

- [1] Barton, C., Iliopoulos, C., Pissis, S.: Fast algorithms for approximate circular string matching. *Algorithms for Molecular Biology* **9** (2014) 9.
- [2] Grossi, R., Iliopoulos, C.S., Mercas, R., Pisanti, N., Pissis, S.P., Retha, A., Vayani, F.: Circular sequence comparison: algorithms and applications. *Algorithms for Molecular Biology* **11** (2016) 1.
- [3] Fredriksson, K., Grabowski, S.: Average-optimal string matching. *J Discrete Algorithms* **7** (2009) 579–594.
- [4] Weil, R., Vinograd, J.: The cyclic helix and cyclic coil forms of polyoma viral DNA. *Proc Natl Acad Sci* **50** (1963) 730–738.
- [5] Dulbecco, R., Vogt, M.: Evidence for a ring structure of polyoma virus DNA. *Proc Natl Acad Sci* **50** (1963) 236–243.
- [6] Thanbichler, M., Wang, S., Shapiro, L.: The bacterial nucleoid: A highly organized and dynamic structure. *J Cell Biochem* **96** (2005) 506–521.
- [7] Lipps, G.: *Plasmids: Current Research and Future Trends*. Norfolk, UK: Caister Academic Press (2008)
- [8] Allers, T., Mevarech, M.: Archaeal genetics – the third way. *Nat Rev Genet* **6** (2005) 58–73.
- [9] Gusfield, D.: *Algorithms on Strings, Trees and Sequences*. New York, NY, USA: Cambridge University Press (1997)

-
- [10] Del Castillo, C.S., Hikima, J.i., Jang, H.B., Nho, S.W., Jung, T.S., Wongtavatchai, J., Kondo, H., Hirono, I., Takeyama, H., Aoki, T.: Comparative sequence analysis of a multidrug-resistant plasmid from *aeromonas hydrophila*. *Antimicrobial agents and chemotherapy* **57** (2013) 120–129.
- [11] Hopper, G.M.: The education of a computer. In: *Proceedings of the 1952 ACM national meeting (Pittsburgh)*, ACM (1952) 243–249.
- [12] Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology* **48**(3) (1970) 443–453.
- [13] Green, L.C., Wagner, D.A., Glogowski, J., Skipper, P.L., Wishnok, J.S., Tannenbaum, S.R.: Analysis of nitrate, nitrite, and [15 n] nitrate in biological fluids. *Analytical biochemistry* **126** (1982) 131–138.
- [14] Mosig, A., Hofacker, I., Stadler, P., Zell, A.: Comparative analysis of cyclic sequences: viroids and other small circular RNAs. *German Conference on Bioinformatics, Volume 83 of LNI* (2006) 93–102.
- [15] Fernandes, F., Pereira, L., Freitas, A.: CSA: An efficient algorithm to improve circular DNA multiple alignment. *BMC Bioinformatics* **10** (2009) 1–13.
- [16] Lee, T., Na, J., Park, H., Park, K., Sim, J.: Finding optimal alignment and consensus of circular strings. *Proceedings of the 21st annual Conference on Combinatorial Pattern Matching* (2010) 310–322.
- [17] Taanman, J.W.: The mitochondrial genome: structure, transcription, translation and replication. *Biochimica et Biophysica Acta (BBA)-Bioenergetics* **1410** (1999) 103–123.
- [18] Goios, A., Pereira, L., Bogue, M., Macaulay, V., Amorim, A.: *mtdna* phylogeny and evolution of laboratory mouse strains. *Genome research* **17** (2007) 293–298.

-
- [19] Wang, Z., Wu, M.: Phylogenomic reconstruction indicates mitochondrial ancestor was an energy parasite. *PloS one* **9** (2014) e110685.
- [20] Cohen, S., Houben, A., Segal, D.: Extrachromosomal circular dna derived from tandemly repeated genomic sequences in plants. *The Plant Journal* **53** (2008) 1027–1034.
- [21] Kuttler, F., Mai, S.: Formation of non-random extrachromosomal elements during development, differentiation and oncogenesis. In: *Seminars in cancer biology*. Volume 17., Elsevier (2007) 56–64.
- [22] Brodie, R., Smith, A.J., Roper, R.L., Tcherepanov, V., Upton, C.: Base-by-base: single nucleotide-level analysis of whole viral genome alignments. *BMC bioinformatics* **5** (2004) 1.
- [23] Bray, N., Pachter, L.: Mavid: constrained ancestral alignment of multiple sequences. *Genome research* **14**(4) (2004) 693–699.
- [24] Mosig, A., Hofacker, I.L., Stadler, P.F., Zell, A.: Comparative analysis of cyclic sequences: Viroids and other small circular rnas. In: *German Conference on Bioinformatics*. Volume 83. (2006) 93–102.
- [25] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. *Journal of molecular biology* **215** (1990) 403–410.
- [26] Knuth, D.E., Morris, Jr, J.H., Pratt, V.R.: Fast pattern matching in strings. *SIAM journal on computing* **6** (1977) 323–350.
- [27] Boyer, R.S., Moore, J.S.: A fast string searching algorithm. *Communications of the ACM* **20** (1977) 762–772.
- [28] Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. *Commun. ACM* **35** (October 1992) 74–82.
- [29] Wu, S., Manber, U.: Fast text searching: Allowing errors. *Commun. ACM* **35**(10) (October 1992) 83–91.

-
- [30] Lothaire, M.: Applied Combinatorics on Words. New York, NY, USA: Cambridge University Press (2005) .
- [31] Chen, K., Huang, G., Lee, R.C.: Bit-parallel algorithms for exact circular string matching. *Comput. J.* **57** (2014) 731–743.
- [32] ACSMF-Simple-Algorithm. <http://www.inf.kcl.ac.uk/research/projects/asmf/> [Online; Last accessed 30-November-2016].
- [33] Iliopoulos, C., Rahman, M.: Indexing circular patterns. *Proceedings of the 2nd International Conference on Algorithms and Computation* (2008) 46–57.
- [34] Lin, J., Adjero, D.: All-against-all circular pattern matching. *Comput J* **55**(7) (2012) 897–906.
- [35] Maes, M.: On a cyclic string-to-string correction problem. *Inf. Process. Lett.* **35** (June 1990) 73–78.
- [36] Marzal, A., Barrachina, S.: Speeding up the computation of the edit distance for cyclic strings. In: *Pattern Recognition, 2000. Proceedings. 15th International Conference on. Volume 2.* (2000) 891–894 vol.2.
- [37] Bunke, H., Bhlér, U.: Applications of approximate string matching to 2d shape recognition. *Pattern Recognition* **26** (1993) 1797 – 1812.
- [38] Barton, C., Iliopoulos, C., Kundu, R., Pissis, S., Retha, A., Vayani, F.: Proceedings of lecture notes in computer science. In: *Accurate and efficient methods to improve multiple circular sequence alignment. In experimental algorithms 14th international symposium, SEA, Springer* (2015) 247–258.
- [39] Fernandes, F., Pereira, L., Freitas, A.T.: Csa: An efficient algorithm to improve circular dna multiple alignment. *BMC bioinformatics* **10** (2009) 1.
- [40] Lee, T., Na, J.C., Park, H., Park, K., Sim, J.S.: Finding consensus and optimal alignment of circular strings. *Theoretical Computer Science* **468** (2013) 92–101.

-
- [41] Ukkonen, E.: Approximate string-matching with q-grams and maximal matches. *Theoretical computer science* **92** (1992) 191–211.
- [42] Rasmussen, K.R., Stoye, J., Myers, E.W.: Efficient q-gram filters for finding all ε -matches over a given length. *Journal of Computational Biology* **13** (2006) 296–308.
- [43] Peterlongo, P., Sacomoto, G.A.T., do Lago, A.P., Pisanti, N., Sagot, M.F.: Lossless filter for multiple repeats with bounded edit distance. *Algorithms for Molecular Biology* **4** (2009) 1.
- [44] Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. *siam Journal on Computing* **22** (1993) 935–948.
- [45] Fischer, J.: Inducing the lcp-array. *Algorithms and Data Structures, Volume 6844 of Lecture Notes in Computer Science* (2011) 374–385.
- [46] Nong, G., Zhang, S., Chan, W.: Linear suffix array construction by almost pure induced-sorting. *Proceedings of the 2009 Data Compression Conference* (2009) 193–202.
- [47] Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J Comput* **40** (2011) 465–492.
- [48] Ilie, L., Navarro, G., Tinta, L.: The longest common extension problem revisited and applications to approximate string searching. *J Discrete Algorithms* **8**(4) (2010) 418–428.
- [49] Dori, S., Landau, G.: Construction of aho corasick automaton in linear time for integer alphabets. *Inf Process Lett* **98** (2006) 66–72.
- [50] Helinski, D.R., Clewell, D.: Circular dna. *Annual review of biochemistry* **40** (1971) 899–942.
- [51] Ehlers, T., Manea, F., Mercaş, R., Nowotka, D.: k-abelian pattern matching. *Journal of Discrete Algorithms* **34** (2015) 37–48.
- [52] Burcsi, P., Cicalese, F., Fici, G., Lipták, Z.: Algorithms for jumbled pattern matching in strings. *International Journal of Foundations of Computer Science* **23**(02) (2012) 357–374.

-
- [53] saCSC Algorithm. <http://www.github.com/solonas13/csc/> [Online; Last accessed 30-November-2016].
- [54] Genome-Data. <http://hgdownload-test.cse.ucsc.edu/goldenPath/hg19/bigZips/> [Online; Last accessed 30-November-2016].
- [55] Genome-Reference-Consortium. <http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/> [Online; Last accessed 30-November-2016].
- [56] Filter-CSC. <https://goo.gl/A31Y4w> [Online; Last accessed 30-November-2016].
- [57] Filter-ECPM, Filter-ACPM. <http://goo.gl/qKgcaU/> [Online; Last accessed 30-November-2016].