M.Sc. ENGG. THESIS

An Efficient Approach for Finding Common DNA Motifs with Gaps

by

Suri Dipannita Sayeed
Student ID: 0416052004

Submitted to

Department of Computer Science and Engineering

in partial fulfillment 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

July 09, 2019
Dedicated to the Department of CSE, BUET

AUTHOR’S CONTACT

Suri Dipannita Sayeed
282, New Elephant Road
Dhaka-1205.
Email: suri.esha6@gmail.com
The thesis titled “An Efficient Approach for Finding Common DNA Motifs with Gaps”, submitted by Suri Dipannita Sayeed, Roll No. 0416052004P, Session April 2016, 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 July 9, 2019.

Board of Examiners

1. [Signature]  
Dr. Atif Hasan Rahman  
Assistant Professor  
Department of Computer Science and Engineering  
Bangladesh University of Engineering and Technology, Dhaka-1000.

2. [Signature]  
Dr. Md. Mostofa Akbar  
Professor and Head of the Department  
Department of Computer Science and Engineering  
Bangladesh University of Engineering and Technology, Dhaka-1000.

3. [Signature]  
Dr. M. Sohel Rahman  
Professor  
Department of Computer Science and Engineering  
Bangladesh University of Engineering and Technology, Dhaka-1000.

4. [Signature]  
Dr. Md. Shamsuzzoha Bayzid  
Assistant Professor  
Department of Computer Science and Engineering  
Bangladesh University of Engineering and Technology, Dhaka-1000.

5. [Signature]  
Dr. Swakkhar Shatabda  
Associate Professor  
Department of Computer Science and Engineering  
United International University (UIU), Dhaka-1212.
Candidate’s Declaration

This is to certify that the work presented in this thesis entitled “An Efficient Approach for Finding Common DNA Motifs with Gaps” is the outcome of the investigation carried out by me under the supervision of Dr. Atif Hasan Rahman in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka. It is also declared that neither this thesis nor any part thereof has been submitted or is being currently submitted anywhere else for the award of any degree or diploma.

______________________________
Suri Dipannita Sayeed
Candidate
Acknowledgment

First of all, I would like to declare that all the appraisals belong to the Almighty ALLAH. I would like to express my heart-felt gratitude to my supervisor, Dr. Atif Hasan Rahman for introducing me to the fascinating and prospective field of bioinformatics and for his constant supervision of this work. I thank him for his patience in reviewing my so many inferior drafts, for correcting my proofs and language, suggesting new ways of thinking, leading to the right way, and encouraging me to continue my research work. I again express my indebtedness, sincere gratitude and profound respect to him for his continuous guidance, suggestions and whole hearted supervision throughout the progress of this work.

I especially want to thank Dr. M. Sohel Rahman and Dr. Md. Shamsuzzoha Bayzid, the other members of my thesis committee for their valuable guidelines and suggestions.

I am also indebted to my undergraduate thesis supervisor professor Dr. Md. Saidur Rahman, from whom I have learned how to carry out quality research.

Next, I would like to thank the ICT Division of Bangladesh for providing me with a research fellowship to work with flexibility. I would like to say that I am grateful to every course teacher for their valuable teaching and my fellow mates for their cooperation, all through the journey of my M.Sc. studies and research.

Finally, once again, I deeply express my sincere gratitude to the endless kindness of Almighty Allah for every blessing that is bestowed upon me.
Abstract

Motifs are repeated patterns in groups of protein or nucleic acid sequences and motif discovery is an important and challenging problem in computational biology. This thesis formulates the gapped motif finding problem as multiple longest common sub-sequence (MLCS) problem and presents an algorithm which solves both of them. The algorithm is based on branch and bound strategy and solves the problem recursively.

Motif finding has been widely studied and several variants have been proposed. It is the problem of identifying recurring patterns in sequences. Here, we address the problem of finding Common Motifs with Gaps (CMG) that are present in all strings of a finite set. Searching the Longest Common Subsequences (LCS) among a set of biosequences is another fundamental problem in bioinformatics. This is a classical NP-hard problem. In this thesis, we prove that the CMG problem is NP-hard by reducing the MLCS problem to it.

To provide efficient exact solution for both of the problems we give a novel algorithm based on branch and bound method. We propose a preprocessing strategy and a data structure based on that preprocessing part. This preprocessed data structure reduces the total space consumption significantly as no additional data structure is required during simulation of the algorithm. We show the result of practical analysis on simulated sequences that our algorithm outperform all the other existing approaches for solving MLCS problem in terms of space. Our implementation of the algorithm also shows promising results in terms of time compared to some extensively used parallel algorithms. We also show how the algorithm can be extended to give an algorithm for CMG after common factors that occur in all the strings have been identified. We have also implemented the algorithm for CMG and it can solve the CMG problem efficiently.
Contents

Board of Examiners ii
Candidate’s Declaration iii
Acknowledgment iv
Abstract v

1 Introduction 1
  1.1 Basic Molecular Biology 2
    1.1.1 Proteins, DNA and RNA 2
    1.1.2 Gene Expression and Regulation 3
    1.1.3 Structure of gene sequence 4
  1.2 Discovering Patterns in Biological Sequences 7
    1.2.1 Sequence alignment 8
    1.2.2 Motifs 9
    1.2.3 Longest Common Subsequence 11
  1.3 Objective of This Thesis 12
  1.4 Summary of Results 12
  1.5 Thesis Organization 13

2 Preliminaries 14
  2.1 Basic Terminology 14
5.3 Measuring Performances

5.3.1 Evaluation under various lengths of sequences

5.3.2 Evaluation under various numbers of sequences

5.4 Discovering Motifs

5.4.1 Summary

6 Conclusion
List of Figures

1.1 Central dogma of life.............................................................................................................3
1.2 Gene structure [1]...............................................................................................................6

2.1 Shows graphs of the common functions from Table 2.1 [2]..............................................19
2.2 Illustration of the search space of B&B............................................................................24

4.1 Simulation of Algorithm 4.1 on an example set..............................................................41
4.2 Simulation of the algorithm for finding common motifs with gaps on an example........43

5.1 Pre-processed list of common factors............................................................................47
5.2 Average running time for input sequences with variable lengths...............................50
5.3 Memory as a function of input length.............................................................................51
5.4 Average running time for input sequences with variable numbers.............................53
5.5 Memory as a function of input number...........................................................................55
5.6 Time as a function of input length.................................................................................57
List of Tables

2.1 Some common order of magnitude functions.................................................................18

5.1 The average running times (sec) of the test algorithms under different lengths of DNA
with the number of sequences fixed at 5 [3]. ........................................................................49

5.2 The memory requirement (MB) of the test algorithms under different lengths of DNA
sequences with the number of sequences fixed at 5 [3]......................................................51

5.3 The average running times (sec) of the test algorithms under different number of
DNA sequences with the length of sequences fixed at 100 [3].........................................52

5.4 The average memory requirement (MB) of the test algorithms under different number
of DNA sequences with the length of sequences fixed at 100 [3]....................................54

5.5 The average running times (sec) of the test algorithm under different lengths of DNA
with the number of sequences fixed at 5.............................................................................56
List of Algorithms

4.1 MLCS ................................................................................................................................. 40
Chapter 1

Introduction

Motif discovery in biological sequences can be termed as the problem of finding short similar sequence elements shared by a set of nucleotide or protein sequences with a common biological function. The identification of gene regulatory elements in nucleotide sequences like transcription factor binding sites, has been one of the most widely studied application of the problem for its biological significance. Motifs that characterize protein families and can be used to predict the family membership, structure and functionality of new proteins is another important one in molecular biology.

Two problems related to pattern matching with wide range of solution spectrum and application in bioinformatics are the problem of Finding Common Motifs with Gaps (CMG) and the problem of finding Multiple Longest Common Subsequences (MLCS). The existing solutions for both of the problems have its own significance and shortcomings. This thesis discuss both of the problem elaborately and tries to resolve the computational hardness of the problem of Finding Common Motifs with Gaps. Moreover it provides an exact algorithm for solving both problems.

This chapter begins with a brief overview of relevant concepts and terminology in molecular biology and biological sequence analysis. Then it discusses the significance of discovering patterns in biological sequences. Finally the chapter concludes with the objective of the thesis and summary of the outcomes of the research.
1.1 Basic Molecular Biology

Molecular biology is the study of living things at the level of the molecules which control them and make them up [4]. Molecular biology provides scientists with a toolkit which they may utilize to understand how life works. They may use it to determine the function of single genes or proteins, and to find out what would happen if that gene or protein was absent or faulty. It is used to examine when and why certain genes are switched on or off. An understanding of each of the factors has granted scientists a deeper understanding of how living things work, and used this knowledge to develop treatments for when living things don’t work so well. Of particular importance to molecular biology are the nucleic acids (DNA and RNA) and the proteins which are constructed using the genetic instructions encoded in those molecules. Much effort has gone into developing computer algorithms to analyze raw biological sequence data and reduce the amount of human expertise and laboratory experimentation necessary in order to understand the properties of these molecules.

1.1.1 Proteins, DNA and RNA

The three most important molecules of life are proteins, DNA and RNA [5]. Among all the organic molecules proteins are most abundant in living systems. Proteins are also way more diverse in structure and function than other classes of macromolecules. A single cell can contain thousands of proteins, each with a unique function and structure. Protein structures are mainly made up of one or more chains of amino acids, although their functions, vary greatly over the cell. Proteins are composed from twenty different amino acids hinged together in long chains by chemical bonds known as peptide bonds. The average protein chain contains approximately 300 to 400 amino acids.

DNA (deoxyribonucleic acid) is called the basic building block of the life. It contains those information that the cell requires to synthesize protein and for replicating itself. In fact, it can be called the storage repository for information that is required for any cell to operate. Based on the initial findings of Rosalind Franklin, Watson-Crick has discovered the current-structure of DNA in 1953. The famous double-helix structure of DNA has its own significance. There are basically four nucleotide bases, which make up the DNA. Adenine (A), Guanine (G), Thymine (T) and Cytosine (C). Each base has its complementary base, which means in the double helical structure of DNA, A will have T
as its complementary and similarly G will have C. This complementary-pairing property is crucial in the replication of DNA as well as in protein synthesis.

RNA is somewhat similar to DNA; they both are nucleic acids of nitrogen-containing bases joined by sugar-phosphate backbone. However structural and functional differences distinguish RNA from DNA. Structurally, RNA is a single-stranded where as DNA is double stranded. DNA has Thymine, where as RNA has Uracil. RNA nucleotides include sugar ribose, rather than the Deoxyribose that is part of DNA. Functionally, DNA maintains the protein-encoding information, whereas RNA uses the information to enable the cell to synthesize the particular protein.

1.1.2 Gene Expression and Regulation

A gene is a string of DNA hidden in a cell’s nucleus. knowing when the gene will be expressed is a widely studied topic in molecular biology for its value in the context of gene regulation. The study of gene expression gives us knowledge about these questions related to gene [6]. Gene expression is the process by which the genetic code of a gene direct protein synthesis. Genes that code for amino acid sequences are known as “structural genes”. The process of gene expression has two basic steps: transcription and translation.

![Central dogma of life](image)

Figure 1.1: Central dogma of life

This is called the **central dogma of life**.
Transcription:

The first step of DNA based gene expression is called transcription. In this process, by using the enzyme called RNA polymerase, a particular segment of DNA is copied into RNA (specifically mRNA). Both DNA and RNA use base pairs of nucleotides for transcription. In summary, during transcription, a DNA sequence is read by an RNA polymerase, which produces a complementary, antiparallel RNA strand called a primary transcript.

Translation:

The second step of gene expression is called translation. Translation is the process by which ribosomes in the cytoplasm synthesize proteins after the transcription of DNA to RNA in the cell’s nucleus. This entire process is called gene expression. In translation, to produce a specific amino acid chain, or polypeptide bond, messenger RNA (mRNA) is decoded in the ribosome’s decoding center. After that, the polypeptide chain folds into an active protein and starts performing its functions in the cell. The ribosome facilitates decoding by inducing the binding of complementary tRNA anticodon sequences to mRNA codons. The tRNAs carry specific amino acids that are chained together into a polypeptide bond while the mRNA passes through and read by the ribosome.

1.1.3 Structure of gene sequence

The structure of a gene involves a number of different components. Here, we represent some definitions from Wikipedia [7] about the most prominent parts related to a DNA sequence.

Start site

The start site or transcription start site is the location where the transcription process starts inside the gene sequence. It is generally assumed to be at the 5′–end of a gene sequence.
Exons

An exon is any part of a gene that encodes a part of the final mature RNA that is produced by that gene after introns are removed by RNA splicing. The term exon refers to the DNA sequence within a gene and also to the corresponding sequence in RNA transcripts. In RNA splicing, as part of generating the mature messenger RNA, introns are removed and exons are covalently joined to one another. The entire set of exons constitutes the exome, just as the entire set of genes for a species constitutes the genome.

Introns

An intron is any nucleotide sequence within a gene that is removed by RNA splicing during the maturation process of the final RNA product. The word intron is derived from the term intragenic region. It means a region inside a gene. Just like exon, the term intron refers to both the DNA sequence within a gene and the corresponding sequence in RNA transcripts. Introns are found in the genes of most organisms and many viruses, also can be found in a wide range of genes. For example, genes that generate proteins, ribosomal RNA (rRNA), and transfer RNA (tRNA). When proteins are generated from intron-containing genes, RNA splicing takes place. It is a part of the RNA processing pathway that follows transcription and precedes translation.

A promoter.

A promoter is a region of DNA sequence that initiates the transcription process of a particular gene. Promoters are located near the transcription start sites of genes. It may start on the same strand and upstream on the DNA (towards the 3’ region of the anti-sense strand). Promoters can be up-to 100 –1000 base pairs long.

Enhancers

An enhancer in gene is a short (50–1500 bp) region of DNA that can be bound by proteins. Its primary activity is to increase the likelihood of transcription of a particular gene. These proteins are usually termed as transcription factors. Enhancers can be located up to 1 Mbp (1,000,000 bp) away from the
CHAPTER 1. INTRODUCTION

gene, upstream or downstream from the transcription start site. There are hundreds of thousands of enhancers in the human genome. Enhancers are found in both prokaryotes and eukaryotes.

Silencers

A silencer is a DNA sequence which is capable of binding transcription regulation factors, called repressors. DNA contains genes. DNA also provides the template to produce messenger RNA (mRNA). mRNA is then translated into proteins. When a repressor protein binds to the silencer region of DNA, RNA polymerase enzyme is prevented from transcribing the DNA sequence into RNA. When transcription is blocked, the translation of RNA into proteins is impossible. Thus, silencers provides a mechanism to prevent genes from being expressed as proteins.

Figure 1.2: Gene structure [1]

Gene regulation

Regulation of gene expression includes a wide range of processes. These processes are used by cells to increase or decrease the production of specific gene products such as protein or RNA. Gene expression is widely analyzed in biology, for many reasons. For example to trigger developmental pathways, respond to environmental catalyst, or adapt to new food sources. Ideally any step of gene expression can be modulated, from transcriptional initiation, to RNA processing, and to the post–translational modification of a protein. In a gene regulatory network, sometimes, this controlling
mechanisms are interchanged between genes such as one gene regulator controls another, and so on.

**Transcription factors**

A transcription factor (TF) is a sequence-specific DNA-binding factor. It is a protein that controls the rate of transcription of genetic information from DNA to messenger RNA. It is done by binding to a specific DNA sequence. The function of TFs is to regulate turning on and off of genes in order to make sure that they are expressed in the right cell at the right time and in the right amount all through the life of the cell and the organism. Groups of TFs function in a coordinated fashion for different purposes. For example, to direct cell division, cell growth, and cell death, cell migration and organization (body plan) during embryonic development etc. Intermittently it also responses to signals from outside the cell, such as a hormone. There are up to 2600 TFs in the human genome.

### 1.2 Discovering Patterns in Biological Sequences

Patterns usually correspond to functionally or structurally important elements in proteins or DNA sequences [8]. There is an assumption that these important regions are better conserved and therefore they occur more frequently than expected. Pattern discovery is one of the fundamental problems in bioinformatics. It can be used in protein structure and function prediction, characterization of protein families, promoter signal detection, gene expression analysis and other areas. Nucleotide and protein sequences contain patterns or motifs that have been preserved through evolution because they are important to the structure or function of the molecule. In proteins, these conserved sequences may be involved in the binding of the protein to its substrate or to another protein, may comprise the active site of an enzyme or may determine the three dimensional structure of the protein. Nucleotide sequences outside of coding regions in general tend to be less conserved among organisms, except where they are important for function, that is, where they are involved in the regulation of gene expression. Discovery of motifs in protein and nucleotide sequences can lead to determination of function and to elucidation of evolutionary relationships among sequences.
1.2.1 Sequence alignment

In bioinformatics, dealing with noise or variability for finding matching sequences is very common. Sequence alignment of two or more sequences provides the basis of most biological sequence learning algorithms proposed until now [9]. The rudimentary application of sequence alignment is to learn new information from known sequences to explore new genes or proteins. It was frequently observed that, if few gaps were inserted in one or the other sequence, many protein sequences could be aligned so that the same letters would line up in each of the sequences. While different letters lined up, it was often the case that the amino acids they represented had similar chemical properties or biological functions. In that situation, substitution was less likely to change properties or functions of the protein than a random substitution induced in the middle of the sequence.

Proteins and genes that have very closely relation, can be aligned globally by inserting some gaps in each of the sequences. But protein relationships that are more distant and many DNA and RNA relationships cannot be easily detected by global sequence alignment. However, even in those cases sequence alignment can be used locally. Short stretches of protein or DNA are sometimes highly conserved or similar. This happens because certain portions of a protein may be more critical to its chemical function and also constrained by evolutionary forces to stay relatively unchanged even in distant variants. In DNA, these similar stretches of sequences are more often identified as regulatory sites. These sites strictly conserve their nucleic acid sequence in order to continue to act their function in the cell. Here is an example. Two given sequences are GCAGT and AATCGAA, a possible alignment of them is:

\[
\begin{array}{ccccccc}
G & C & A & \varphi & \varphi & G & \varphi & T \\
\varphi & A & A & T & C & G & A & A
\end{array}
\]

There are one deletion (1st column), three insertions (4th, 5th and 7th columns), two mismatches (2nd and 8th columns) and two matches (3rd and 6th columns). In the alignment of two DNA or protein sequences, the purpose is to find the best alignment. For this reason the effectiveness of each possible alignment is needed to be calculated. The primary approaches included similarity was based on the number of matches in an alignment where it was possible to assign a score or
penalty for each insertion or deletion. insertions/deletions are otherwise known as indels. A gap is a consecutive stretch of indels. The gap length also defines the number of indels. The simple example here has one gap of length 2. In literature, numbers of computational algorithms for sequence alignment problems are available. The very fundamental one is based on dynamic programming methods. It is computationally slow but formally an optimal solution can be achieved. Needleman and Wunsch first gave a dynamic programming algorithm for the global sequence alignment problem. By global sequence alignment, it means aligning both sequences globally and considering the whole of sequences [10]. A modification of the Needleman-Wunsch algorithm is given in [11]. The goal of this new alignment is to find the highest scoring local match between the two sequences. In a lot of applications, a local alignment is preferred to the global alignment. The aim of local alignment is to find the best pair of sub-sequences from each sequence, such that the best possible optimal alignment of two regions is achieved. The sub-sequences may be a part or all of the both sequences. If all are included then the local alignment is also just like a global alignment.

1.2.2 Motifs

Sequence motifs are short, recurring patterns in DNA that are presumed to have a biological function. Often they indicate sequence-specific binding sites for proteins such as nucleases and transcription factors (TF). Others are involved in important processes at the RNA level, including ribosome binding, mRNA processing (splicing, editing, polyadenylation) and transcription termination. The abundance of both computationally and experimentally derived sequence motifs and their growing usefulness in defining genetic regulatory networks and deciphering the regulatory program of individual genes make them important tools for computational biology in the post genomic era. There could be different variants of motif. And different tools and algorithms are required to retrieve valuable information from them. In this subsection, different variants of motif and available algorithms for them will be discussed thoroughly.
Motifs with gaps

Given a set of strings, the problem of finding common motifs in that set is the problem of finding similar substrings that lie in all of these strings. In some particular applications, motifs do not have to be identical but have to share a certain degree of similarity. This degree is quantified using metrics such as Hamming and Levenshtein distances or by allowing don't care symbols to occur in the motifs. Don't care symbols are occurrences in the string that can match any symbol of the alphabet. In this paper, we are interested in finding common motifs in the strings that have don't care symbols concentrated in distinct parts of contiguous positions in the strings, i.e. common motifs with gaps. The most constrained version of the problem occurs when the size of the gaps remain always the same. For example, given a set of three strings $S$:

$$S_1 = ACGTACGGATGCCTCAAA$$
$$S_2 = GACCTACCGAGCGCTCGTTAA$$
$$S_3 = ACTTACTGATTTCTCAA$$

here, gap length=1.

And there could be motifs with gaps of variable sizes, but whose sum is upper bounded by a user defined parameter. Again, the gapped motif must occur at least once in each one of the input strings. For example,

$$S_1 = ACGTACGGATGCCTCAAA$$
$$S_2 = GACCTACCGAGCGCTCGTTAA$$
$$S_3 = ACTTACTGATTTCTCAA$$

here, gap length ranges from 1 to 3.
Motifs with mismatch

Another important variant of the DNA motif problem is finding an implanted motif with specific length that allows certain mismatches. For example, finding a motif in a sample of 20 random sequences (e.g. 600 nucleotides long). Where,

- Each sequence contains an implanted pattern of length 15.
- Each pattern appears with 4 mismatches.

This problem is more formally defined as finding common \((l, d)\) motifs in a set of genome sequences where, an \((l, d)\) motif is a pattern of length \(l\) which appears with \(d\) mismatches within a DNA sequence. On the other hand, based on this \(d\) mutations a consensus string can be generated which may be a representative of the implanted motif. Finding \((l, d)\) motifs with exactly \(d\) mutations is more challenging than finding \((l, d)\) motifs with up to \(d\) mutations, and algorithms designed for the former can usually be directly used to find the later. Another variant of this problem is known as planted motif search.

1.2.3 Longest Common Subsequence

The Longest Common Subsequence (LCS) problem is to find subsequences in given sequences in which the subsequence is as long as possible. These subsequences are not necessarily contiguous or unique. Measuring the similarity of biological sequences is a fundamental problem in bioinformatics, which has many applications such as in cancer diagnosis and detection of the species common origin, etc. One of the most important ways to measure the similarity of sequences is to find their Longest Common Subsequences (LCS). According to the number of sequences, the problems are classified into two cases: (1) Looking for the longest common subsequence of two sequences is called the Longest Common Subsequence (LCS) problem. (2) Looking for the longest common subsequence of more than two sequences is called the Multiple Longest Common Subsequences (MLCS) problem. The simplest version of the problem is finding LCS between two sequences. Which can be done in polynomial time using dynamic programming. However, searching for the Multiple Longest
Common Subsequences (MLCS) of multiple sequences is a classical NP-hard problem. One of the most effective exact approaches for the MLCS problem is based on dominant point graph, which is a kind of directed acyclic graph (DAG). However, the time and space efficiency of the leading dominant point graph based approaches is still unsatisfactory: constructing the dominated point graph used by these approaches requires a huge amount of time and space, which hinders the applications of these approaches to large-scale and long sequences. Some improvements over these approaches are proposed in some recent works, such as FAST-LCS, Leveled-DAG etc.

1.3 Objective of This Thesis

In this thesis, first we formally settle the question of complexity of the problem of finding common motifs with gaps. We prove that the problem of finding common motif with gaps is NP-hard, for alphabet size of four or more, by reducing to it the multiple longest common subsequence (MLCS) problem. We also present a branch and bound algorithm for computing the longest common subsequences of a set of strings and further extend this algorithm to find common motifs with gaps. We first find the common factors appearing in all the strings and then use a branch and bound algorithm to chain together the factors. Although this problem is treated as a hard problem in the literature and is handled accordingly, to the best of our knowledge this is the first attempt to prove the hardness thereof formally. And the reduction from MLCS in the sequel allows us to adapt the branch and bound algorithm for MLCS to solve the problem of finding common motifs with gaps.

1.4 Summary of Results

In this thesis, we address two most fundamental problems of computational biology and theoretical computer science research. The main results of this theses are as follows:

1. A proof of NP-hardness of the problem of finding common motifs with gaps.

2. A branch and bound algorithm for solving the MLCS problem and an extension of that algorithm for finding the common motifs with gaps.
3. Implementation and performance analysis of the algorithms.

1.5 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we formally define the two problems of our interest. Besides, we discuss the relevant ideas and necessary definitions from algorithm theory to understand our research work, in this section, we have also discussed the strategies for practical implementation. In Chapter 3 we provide a detailed overview of the related literature that we have studied for the sake of this research. In Chapter 4 we show the proof of NP-hardness of the CMG problem and gradually formulate the problem space for which we provide an exact algorithm. We also discussed in detail, the experimental set-up and programming strategies for practical implementation. We discuss in detail the intuition and logic behind our algorithm for solving MLCS problem. The complete process of the algorithm is shown here. We also give an extension of the algorithm that solves the CMG problem. In chapter 5, we present an extensive experimental analysis comparing the performance of our algorithm with the algorithm of Peng et al. [3], the algorithm of Wang et al. [12], the algorithm of Li et al. [13]. Finally we conclude the thesis in Chapter 6 with some future directions.
Chapter 2

Preliminaries

In this chapter we define some basic terminologies relevant to our research and algorithm theory. Definitions that are not included in this chapter will be introduced as they are needed. We start, in Section 2.1, by giving definitions of Common Motifs with Gaps (CMG) and Multiple Longest Common Sub-sequence (MLCS). The notion of time complexity is discussed in Section 2.2.

2.1 Basic Terminology

2.1.1 Common Motifs

In general, motif is a constantly repeated pattern throughout a work. In bioinformatics, a sequence motif is a nucleotide or amino-acid sequence pattern that is widespread in the protein or DNA sequence and has been proven or assumed to have a biological significance. Motifs have many different kinds of variants. For example: motifs with gaps, motifs with mismatch etc. are widely studied variants of motifs in bioinformatics research.

2.1.2 Common Motifs with Gaps

Consider a set of strings $S = \{S_1, S_2, \ldots, S_d\}$ over the alphabet $\Sigma = \{A, C, G, T\}$ and two integers $p$ and $q$, where, $1 \leq p \leq q \leq \min(|S_j| : j \in \{1, \ldots, d\})$ are given. The problem of finding common motifs with gaps (CMG) aims at finding common factors $P_1, P_2, \ldots, P_m$ such that: $P_1 \star^{d_{1,1}} P_2 \star^{d_{1,2}} \cdots \star^{d_{1,m-1}} P_m$
occurs in each $S_i$, for all $i \in \{1, \ldots, d\}$, where, $m > 1$ and $p \leq |P_j| \leq q$ for all $j \in \{1, \ldots, m\}$. Also, $d_{i,j} \geq 1$ for all $i \in \{1, \ldots, d\}$ and $j \in \{1, \ldots, m-1\}$. That is the factor lengths are bounded by the integers $p$ and $q$ and the minimum gap length is 1, which is introduced by $d$ don’t care symbols. Here, $\ast$ is the don’t care symbol that matches any character in $\Sigma = \{A, C, G, T\}$.

For example, given a set of three strings $S = S_1$, $S_2$, $S_3$ and minimum factor size, $p = 1$ and maximum factor size, $q = 2$, the substrings $AC, AA$ and $CA$ form the common motifs that satisfy the required criteria as highlighted below:

\[
S_1 = \text{ACAAAC} \text{ACA} A A S_2 \\
= \text{ACACCAAC} \text{ACCACA} S_3 = \\
\text{CACA} \text{AC} \text{CA} \text{CA} \text{CA} \text{CA} \text{CA} \text{CA}
\]

In the optimization version of the problem, we want to maximize $m$ i.e. we seek a common motif with gaps with maximum number of factors and in the decision version of the problem, for a given $m$, we want to check whether there is a common motif with gaps with $\geq m$ factors [14].

### 2.1.3 Longest Common Subsequence

We are given two strings: string $S$ of length $n$, and string $T$ of length $m$. Our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings [15]. For example, consider:

\[
S = \text{ABAZDC} \\
T = \text{BACBAD}
\]

In this case, the LCS has length 4 and is the string $ABAD$. Another way to look at it is we are finding a 1-1 matching between some of the letters in $S$ and some of the letters in $T$ such that none of the edges in the matching cross each other.

For instance, this type of problem comes up frequently in genomics: given two DNA fragments, the
LCS gives information about what they have in common and the best way to line them up.

2.1.4 Multiple Longest Common Subsequence

The Multiple Longest Common Subsequence (MLCS) problem aims at finding a longest subsequence shared among a set of sequences. Let, $S = \{S_1, S_2, S_3, \ldots, S_d\}$ be a set of sequences over a finite alphabet $\Sigma$. The Longest Common Subsequence (LCS) of set $S$ is a sequence $s$ with length $A$, such that it is of the highest length among all subsequences that are shared among all $S_i$, $i \in \{1, \ldots, d\}$. For example,

$$S_1 = \text{informatics}$$

$$S_2 = \text{bioinformatics}$$

$$S_3 = \text{proteomics}$$

One of the subsequences for this example is $s_1 = \text{mics}$, one is $s_2 = \text{tics}$ and another is $s_3 = \text{omics}$. Notably, this sequence may not necessarily be unique.

In the optimization and decision versions of the problem we intend to find an LCS of the maximum length and decide if there is an LCS greater or equal to a given length, respectively [16].

2.2 Algorithms and Complexity

In this section we will introduce some terminologies related to algorithms and complexity of algorithms. Many books and research works are already available on complexity analysis of algorithms. Interested readers can take a look into the book by Skiena [17].

2.2.1 Big-O Notation

Trying to characterize an algorithm's efficiency in terms of execution time, which is not dependent on any particular program or computer, is largely dependent on quantifying the number of operations or steps that the algorithm will be needed. If each of these steps is considered to be a basic unit of computation, then the most widely accepted complexity measure for an algorithm is the running time
which is expressed by the number of operations it performs before producing the final answer. But deciding an appropriate basic unit of computation can itself be a complicated problem and will be dependent on how the algorithm is implemented. For a simple summation algorithm for example, a naive approach for computing the basic unit of computation might be to count the number of assignment statements performed to compute the sum. It can be denoted by a function, call it \( T \), where \( T(n) = 1 + n \). Where, 1 is for \((\text{Sum} = 0)\) plus the value of \( n \) (the number of times \( \text{Sum} = \text{Sum} + i \)) is performed. In this example of summation algorithm \( n \) is the size of the problem and \( T(n) \) is the time it takes to solve a problem of size \( n \). For analyzing the performance of an algorithm it is reasonable then to show how the algorithms execution time changes with respect to the size of the problem. Computer scientists have taken this technique for analysis one step further. Computing the exact number of operations is not as important or feasible as determining the most significant part of the \( T(n) \) function. As the problem gets larger, some portion of the \( T(n) \) function tends to overpower the rest. This dominant term is in the end used for comparison. It is called the order of magnitude function that describes the part of \( T(n) \) that increases the fastest as the value of \( n \) increases. Order of magnitude is often called Big-O notation or the order and written as \( O(f(n)) \). It provides a useful approximation to the actual number of steps in the computation. The function \( f(n) \) provides a simple representation of the dominant part of the original \( T(n) \). In this example of summation, as \( n \) gets large, the constant 1 will become less significant to the final result. For an approximation of \( T(n) \), then dropping the 1 and simply saying that the running time is \( O(n) \) is completely rational. The 1 is definitely significant for \( T(n) \). However, as \( n \) gets large, the approximation will be more accurate with \( T(n) \). A second example, with another algorithm could be, \( T(n) = 5n^2 + 27n + 1005 \). Here, \( T(n) \) denotes the exact no of steps. When \( n \) is small, the constant 1005 is the dominant part of the function. However, as \( n \) gets larger, the \( n^2 \) term becomes the most important. In fact, when \( n \) is really large, the other two terms become insignificant in the role that they play in determining the final result. Again, to approximate \( T(n) \) as \( n \) gets large, we can ignore the other terms and focus on \( 5n^2 \). In addition, the coefficient 5 becomes insignificant as \( n \) gets large. We would say then that the function \( T(n) \) has an order of magnitude \( f(n) = n^2 \), or simply that it is \( O(n^2) \).
Sometimes the performance of an algorithm depends on the exact values of the data rather than simply the size of the problem, although we do not see this in the summation example. For algorithms like these we need to characterize their performance in terms of best case, worst case, or average case performance. The worst case performance refers to a situation when the algorithm performs especially poorly as the input size increases. Whereas a different input for the exact same algorithm might have extraordinarily good performance. However, in most cases the algorithm performs somewhere in between these two extremes and that is termed as the average case. It is important to understand these differences specially for computer scientists so that they are not misled by any particular case.
2.2.2 Polynomial Algorithms

The runtime of an algorithm is polynomial if for some $k, C > 0$, its running time on inputs of size $n$ is at most $Cn^k$. Equivalently, an algorithm is polynomial if for some $k > 0$, its running time on inputs of size $n$ is $O(n^k)$. It means its complexity is bounded by a polynomial of the size of a problem instance. This includes linear $O(n)$, logarithmic $O(\log n)$, quadratic $O(n^2)$, cubic $O(n^3)$ and more. On the other hand, algorithms with exponential running times $O(2^n)$ are not polynomial.

2.2.3 Problem classes

Whether there is a polynomial time algorithm or not that decides whether a problem is NP-complete or not. A number of interesting computational problems are there in literature for which it is unde-
cided yet. Most of them are NP-complete [18]. In this section we will briefly explain NP-complete problems as well as the complexity class of P and NP.

Before proceeding further, we first introduce some important concepts about complexity classes. Decision problems are those which refer to the algorithmic questions that can be answered by either yes or no. For an example, “Is a particular instance of a vertex set satisfies the solution of maximum vertex cover problem?” The state of algorithms is the one which consists of the current values of all the variables and the location of the current instruction to be executed. Deterministic algorithms are those for which each state, upon execution of the instruction, uniquely determines at most one of the following state (next state). All modern computers, which exist now, run deterministically. In contrast, a nondeterministic algorithm is one for which a state may simultaneously determine many next states. We may define a nondeterministic algorithm as having the capacity of branching off into many copies of itself, one for the each next state. In this way, a deterministic algorithm must explore a set of alternatives one at a time, whereas, a nondeterministic algorithm examines all alternatives at the same time.

A problem \( P_1 \) is reducible to problem \( P_2 (P_1 \leq \text{p} P_2) \) in polynomial time, if there exists a polynomial time algorithm that reduces every instance \( I_1 \) of \( P_1 \) to an instance \( I_2 \) of \( P_2 \). It means, the answer to \( I_1 \) is yes (\( I_1 \in P_1 \)) if and only if the answer to \( I_2 \) is yes (\( I_2 \in P_2 \)).

The Class P

The class of problems that can be solved by deterministic polynomial time algorithm belong to the class \( P \). This implies that there is a deterministic algorithm that takes as input an instance \( I \) and finds a solution \( S \) in polynomial time; and if \( I \) has no solution, the algorithm correctly reports a failure. So, \( P \subseteq NP \). But whether, \( P = NP \) ? that is still unresolved. It is widely accepted that \( P \neq NP \). However, this is considered as one of the deepest and most important unsolved puzzles of mathematics.
The Class NP

The class of problems for which solutions can be verified deterministically in polynomial time is called NP. It means that there is an efficient low-order polynomial time deterministic checking algorithm $C$ that takes as input the given instance $I$ (the data which specifies the problem to be solved), as well as the proposed solution $S$, and outputs true if and only if $S$ really is a solution to instance $I$. Moreover the running time of $C(I, S)$ is bounded polynomially. On the other hand, NP stands for “nondeterministic polynomial time”. So, NP defines the class of decision problems that can be solved nondeterministically in polynomial time.

The class NP-complete

A problem $Y$ is NP-complete if it is NP-hard, and also in NP itself. NP-complete problems are considered to be the hardest problems in NP. These problems have the following interesting properties.

- No NP-complete problem can be solved by any known polynomial algorithm.
- If there is a polynomial algorithm for any NP-complete problem, then there are polynomial algorithms for all NP-complete problems.

2.2.4 Exact Algorithms

An algorithm is a set of constructions and rules for solving a problem. NP-hard problems are problems for which most probably no polynomial time algorithm is known to be existed. An exact algorithm solves an NP-hard problem to optimality, even if, the algorithm requires an unreasonable amount of time. For example, an exact algorithm for the Traveling Salesman Problem (TSP) is NP Complete. In TSP, the problem is to find a path of minimum length $L$ for visiting each of $M$ cities in a given instance. According to the rules of NP completeness, the best known algorithms for TSP need time that is an exponential function of $M$. 

2.2.5 Heuristic Algorithms

A heuristic algorithm, or simply a heuristic in computer science, is an algorithm that is able to produce an acceptable solution to a problem in many practical scenarios, but for which there is no formal proof of its correctness or no way to ensure optimal result [19]. In other words, it is a procedure that determines good or near-optimal solutions to an optimization problem. It may be correct, but may not be proven to produce an optimal solution, or to use reasonable resources. So, heuristics are generally used when there is no known method to find an optimal solution, under a given constraints (of time, memory etc.) or at all. These algorithms, usually find a solution closer to the best one and they find it fast and easily. Sometimes these algorithms can be accurate, that is they actually find the best solution, but the algorithm is still called heuristic until this best solution is proven to be the best. Heuristic algorithms mostly uses greedy algorithms, but in order to the easy and faster execution the algorithm ignores or even suppresses some of the problems’ demands. Heuristics rely on ingenuity, intuition, logical feasibility and a good understanding of the application and also a detailed experimentation to tackle the problem [20].

2.2.6 Algorithmic Paradigms

An algorithmic paradigm, algorithm design paradigm, algorithmic technique, or algorithmic strategy is a generic method or approach which underlies the design of a class of algorithms. It is an abstraction higher than the notion of an algorithm, just as an algorithm is an abstraction higher than a computer program. Examples of algorithmic paradigms include the greedy algorithm in optimization problems, dynamic programming, prune and search, branch and bound and divide and conquer algorithms. Here we will discuss about branch and bound algorithmic paradigm as it will be further used in the algorithm design of our input problems.

Branch and Bound

NP-hard problems are difficult to solve up to optimality and it often requires an intensive task to complete with very efficient algorithms, and the B&B algorithmic paradigm provides a major way out for these [21] scenario. B&B algorithm searches for the full search space of solutions for a given
problem for the best solution. However, explicitly enumerating every path is normally impossible due to the exponentially increasing number of potential solutions. That’s why effective bounding technique is very important for this algorithmic approach and the concept of bounding is introduced. The proper use of bounds for the function to be optimized in combination with the value of the current best solution makes the algorithm enable to search the solution space partially and only implicitly.

During the solution process, the current state of the solution with respect to the search of the solution space at any point is decided by a pool of yet to explore subset of possibilities and the best solution found up to that point. at the beginning, only one subset exists, that is the complete solution space, and the best solution in the beginning is generally 0. The not yet searched sub-spaces are represented as nodes in a search tree which is generated on the fly in dynamic process, which at the beginning contains only the root, and each iteration of a traditional B&B algorithm expands the nodelist by one such node. These iteration process has three main parts: selecting which nodes to process next, calculating an effective bound, and branching based on the pruning condition. In Figure 1, the initial situation and the first step of the process is illustrated. These sequences may change according to the strategy chosen for selecting the next node to process. If the selection of next subproblem is based on the bound value of the current subproblem, then the first operation of an iteration after choosing the node is branching, i.e. subdividing the solution space of the node into two or more sub-spaces to be explored in a subsequent iteration. For each of these, it is checked whether the subspace consists of a single solution or more than one, in that case it is compared to the current best solution keeping the best of the nodes. Otherwise the bounding or pruning function for the subspace is estimated and compared to the current best solution. If it can be decided that the subspace cannot contain the optimal solution, then the whole subspace is discarded, else it is kept stored in the queue of live nodes together with the bound. This strategy is called the eager strategy for node evaluation, since the bounds are calculated when the nodes are ready for estimation. The alternative strategy for node evaluation is to start by calculating the bound of the selected node and then branch on the node if require. The nodes which are created this way are then stored together with the bound of the node which is being calculated before. This strategy is called lazy evaluation and is often used when the next node to be processed is chosen to be a live node of maximal depth in the search tree. The search terminates when there is no unexplored parts of the solution space left, and the optimal solution is
then the one recorded as “current best”.
The solution of a problem with a B&B algorithm is generally described as a search through a search tree, in which the root node corresponds to the original problem to be solved, and each other node corresponds to a subproblem of the original problem. Given a node \( N \) of the tree, the children of \( N \) are subproblems derived from \( N \) through imposing (usually) a single new constraint for each subproblem, and the descendants of \( N \) are those subproblems, which satisfy the same constraints as \( N \) and additionally a number of others. The leaves correspond to possible solutions, and for all NP-hard problems, there exist instances with an exponential number of leaves in the search tree. To each node in the tree a bounding function \( f \) connects a real number called the bound for the node. For leaves the bound equals the value of the corresponding solution, whereas for internal nodes the value is a lower bound for the value of any solution in the subspace corresponding to the node. These state that \( f \) is a bounding function, which for any leaf agrees with the objective function, and which provides closer and closer (or rather not worse) bounds when more information in terms of extra constraints for a sub-problem is added to the problem description.

![Figure 2.2: Illustration of the search space of B&B](image)

The search tree is initially developed dynamically during the search and consists of only the root node. For many problems, a feasible solution to the problem is produced in advance using
a heuristic, and the value hereof is used as the current best solution. In each iteration of a B&B algorithm, a node is selected for exploration from the pool of live nodes corresponding to unexplored feasible subproblems using some selection strategy. If the eager strategy is used, a branching is performed: Two or more children of the node are constructed through the addition of constraints to the subproblem of the node. In this way the subspace is subdivided into smaller subspaces. For each of these the bound for the node is calculated, possibly with the result of finding the optimal solution to the subproblem below. In case the node corresponds to a feasible solution or the bound is the value of an optimal solution, the value hereof is compared to the best solution found so far, and the best solution and its value are kept. If the bound is no better than the best solution upto that point, the subproblem is discarded, since no feasible solution of the subproblem can be better that the best solution already found. In case no feasible solutions to the subproblem exist the subproblem is also discarded. Otherwise the possibility of a better solution in the subproblem cannot be ruled out, and the node (with the bound as part of the information stored) is then merged to the pool of alive subproblems.
Chapter 3

Literature Review

3.1 Motif discovery algorithms

The motif finding problem has engrossed biologists because of its applications in that area. It can be applied in understanding the fundamental process of gene expression. Gene expression consists of two parts, transcription and translation. During transcription an mRNA molecule is created by copying a gene from the DNA and during translation the mRNA molecule is decoded to produce a protein. In order though for the transcription process to begin, one or more proteins, called transcription factors, have to bind to some specific regions of the gene called binding sites. These binding sites share common patterns which are the common motifs of the genes. If these common motifs are identified and extracted from the genes, they will give the opportunity to biologists to match these binding sites to their corresponding transcription factors in order to be able to fully understand the way gene expression works.

Most of the earlier algorithms designed to find motifs used a set of promoter sequences of coregulated genes to identify statistically over-represented motifs. The main classification of these algorithms falls in to two classes. Those are (1) the word-based way that relies on exhaustive enumeration or counting frequencies and (2) the probabilistic way that relies on optimizing a scalar-based scoring matrix, which is visualized conveniently by a sequence logo.

In text algorithm applications, finding common motifs with gaps has been mainly handled using suffix trees which provided exact results. Besides algorithm using automata to index common gapped
motifs are also available. In combinatorial approach, finding multi-sites motif such as dimer and dyad is also a common problem.

In this chapter we will provide a detailed overview of the motif finding algorithms based on their variations.

### 3.1.1 Combinatorial algorithms

**Motifs with gaps**

Antoniou et al. [22] presents algorithms that allow stretching of the length of the motifs as well as elasticity in the length of gaps between the motifs. The main data structure used in these algorithms is the suffix tree. The suffix tree has been the main data structure used in string algorithms for finding common motifs in sequences. This paper presents two algorithms to find the common motifs with gaps in a set of strings, with the patterns that comprise the motifs having fixed lengths or variable lengths. Both algorithms rely on the data structure suffix tree. During preprocessing based on the suffix tree they created four basic data structures of the algorithms; namely the common segments trie, the reverse segments trie and two stacks based on the two tries. The reason behind using these data structures is the fact that the common segments trie can help to find the right part of repeats in strings and the reverse segments trie can give the left part of repeats, considering a motif consisting of two parts separated by a block of don’t cares (the gap). By combining the two tries we can acquire the right and left parts of the motif. Both algorithms require $O(nrm + km)$ time, where $n$ is the total length of the sequences, $r$ is the number of the sequences, $m$ is the number of nodes in the reverse common tree and $k$ is the number of times we combine a right part with a left part of the string to form a motif.

Antoniou et al. [23] presents another algorithm that uses finite automata to find the common motifs with gaps occurring in all strings belonging to a finite set. The algorithm takes advantage of the fact that one can find common motifs of a set of strings by intersecting their corresponding factor automata which were created by the common factors residing in the strings. The algorithm’s time and space complexity is exponential due to the step that requires the intersection of many finite
automata. Other variants of the problem with constraints on lengths of gaps have also been proposed and algorithms have been provided [24].

Motifs with mismatch

Zachariah et al. proposed three approaches in their paper [25] it includes, a brute force algorithm, a branch and bound algorithm, and a greedy algorithm. The brute force algorithm finds out the consensus string by determining which set of starting positions produce the best DNA score. It is a comprehensive search because it checks every possible set of starting positions produced by a method called NextLeaf. The complexity of this search is \( O(ln^t) \) where \( l \) is the length of the motif, \( n \) is the length of the DNA samples, and \( t \) is the number of DNA samples. The more DNA strands are checked the run time of the method increases exponentially. The branch and bound algorithm uses a combination of a method called NextVertex and the Bypass method to skip branches of the tree if it is determined that they cannot produce a better score than the score that has already been obtained. The greedy method does an exhaustive search on the first two strands of DNA to determine the best motif in these two strands. This motif is called the seed. The method then sequentially searches the remaining DNA strands for the motif in each strand that best matches the seed and the motifs that have already been found. The complexity of the greedy algorithm is \( O(ln^2 + nlt) \) where \( l \) is the length of the motif, \( n \) is the length of the DNA samples, and \( t \) is the number of DNA samples. So this method has a squared term for the exhaustive search of the first two DNA strands, and then the rest of the program is a linear search. This is much faster than the exponential brute force algorithm and branch and bound algorithm.

Planted motif search

Hieu et al. [26] proposed an efficient exact algorithm for the \((l, d)\)-motif finding problem. Many facets of the motif search problem have been identified in the literature. One of them is Planted Motif Search (PMS) [27]. PMS is stated as follows. It takes as input \( n \) sequences, two integers \( l \) and \( d \). It is assumed that the length of each sequence is \( m \). The problem is to identify all strings \( M \) of length \( l \) such that \( M \) occurs in each of the \( n \) sequences with at most \( d \) mismatches. Formally, string \( M \) has to satisfy the following constraint: there exists a string \( M_i \) of length \( l \) in sequence \( i \), for every \( i \)
(1 <= i <= n), such that the number of mismatches between M and M_i is less than or equal to d. The number of mismatches between two strings of equal length is known as the Hamming distance between them. String M is called a motif. This paper proposed an exact algorithm called PMS5. There is many existing algorithms to solve PMS but PMS5 can solve many challenging instances of PMS.

Davila et al. proposed two exact algorithms for planted motif search problem in their paper [28]. These algorithms are special as they use less space than all other exact PMS algorithms known thus far. Another speciality of these algorithms is, it can handle very large value of d. In fact the algorithm is known to solve the challenging instance of (17,6), which wasn't found solved before in literature.

3.1.2 Motifs in protein sequence

Rupali et al. [29] introduces a new algorithm for multiple local alignment for protein sequences, based on the de Bruijn graph approach which is first proposed by Zhang and Waterman for aligning DNA sequence. Zhang and Waterman introduced a new algorithm for multiple local alignment of DNA sequences [30]. This was an extension of their EulerAlign approach for global multiple alignment [31]. The aim is to construct a consensus pattern that is most consistent with all input sequences from a de Bruijn graph built from the input sequences. The consensus is then used as a query to locate all instances of the pattern. Their algorithm reduces the time complexity of the problem to approximately linear, thus significantly reducing the computational time when the input size is large. In spite of its success with DNA sequences, it is not straightforward to apply this method to local multiple alignment of protein sequences. There are several important distinctions between homologue protein and DNA sequences. These differences complicate the application of de Bruijn graph approach to the protein sequence alignment. In this paper this issue is addressed by introducing the concept of approximate de Bruijn graph. Then a similar approach is adopted as Zhang and Waterman to perform the local multiple sequence alignment by traversing this graph. It first finds a heaviest path in the approximate de Bruijn graph using a heuristic greedy algorithm and deduces a consensus protein sequence from this path. Next these consensus sequences are aligned to each of the input sequence and finally construct the multiple alignments from these pair wise alignments. The process is repeated to find other patterns. This method is applied to motif discovery in protein sequences.
3.1.3 Probabilistic algorithms

In probabilistic approaches model parameters are estimated using the maximum-likelihood principle or Bayesian inference. Probabilistic methods have the advantage of requiring few search parameters but rely on probabilistic models of the regulatory regions, which can be very sensitive with respect to small changes in the input data. Many of the algorithms developed from the probabilistic approach are designed to find longer or more general motifs than are required for transcription factor binding sites. Therefore, they are more appropriate for motif finding in prokaryotes, where the motifs are generally longer than eukaryotes. However, these algorithms are not guaranteed to find globally optimal solutions, since they employ some form of local search, such as Gibbs sampling, expectation maximization (EM) or greedy algorithms that may converge to a locally optimal solution.

Expectation Maximization

Lawrence and Reilly first introduced the expectation maximization technique for motif finding [8]. Their work was an extension of the greedy algorithm for motif finding by Hertz et al. [32]. This method was initially developed for finding motifs inside protein sequences, however, DNA motif can also be identified by EM method. No alignment of the sites is required for this approach of motif finding. Initially, from basic the model assumption is that each sequence must contain at least one common site. But the uncertainty in the location of the sites is handled by employing the missing information principle for developing the algorithm. This approach allows for the simultaneous identification of the sites. It also helps to characterize the binding motifs. MEME is the most widely used tool for motif finding that has been developed so far. The algorithm of MEME by Bailey and Elkan [8] extended the EM algorithm for identifying motifs in unaligned sequences. The goal of MEME is to discover new motifs in a set of biopolymer sequences where almost nothing is known about any motifs in advance that may be present in any sequence. MEME incorporated three unique ideas for discovering motifs. First, subsequences that actually occur in the biopolymer sequences are used as starting points for the EM algorithm to increase the probability of finding globally optimum motifs. Second, the assumption that each sequence contains exactly one occurrence of the shared
CHAPTER 3. LITERATURE REVIEW

motif is totally discarded. Third, a method for probabilistically erasing shared motifs after they are found is employed. It was for finding several distinct motifs in the same set of sequences, when different motifs appear in different sequences and also when a single sequence may contain multiple motifs.

Gibbs Sampling

Among the probabilistic methods Gibbs sampling method has been used extensively for motif finding algorithms. Gibbs sampling technique is specially used for finding common \((l, d)\) motifs. A short description of the original Gibbs sampler method for motif finding developed by Lawrence et al. [33] is presented here. They applied their algorithm to protein sequences instead of DNA, in the original article. One of the original assumptions of this algorithm was that there exists at least one instance of a motif in every sequence, so, the method is sometimes called the “site sampler”. Gibbs sampler follows a Markov Chain Monte Carlo (MCMC) approach: “Markov-Chain” here, the results from every step depends only on the results of the preceding one like in expectation maximization technique. In “Monte-Carlo” the next step is not deterministic but rather based on random sampling. In the book by Liu [34] and in the article by Liu et al. [35], the statistical background of MCMC methods and that of gibbs sampling is explained in detail. In Gibbs sampling algorithm it is assumed that a set of \(N\) sequences \(S_1, \ldots, S_N\) are given and within each sequence mutually similar segments of specified width \(W\) are identified. Two evolving data structures are maintained all-through the algorithm. The first data structure contains the pattern description, It is maintained in the form of a probabilistic model of residue frequencies. And it is contained for each position \(i\) from 1 to \(W\), which is consisted of the variables \(r_{i1}, \ldots, r_{i20}\) indexed by \(W\) positions and also contains the 20 possible residues. This pattern description data structure is additionally accompanied by an analogous probabilistic description of the “background-frequencies” \(p_1, \ldots, p_{20}\) with which residues may occur in sites that has not been described by the original pattern. The second data structure, constitutes the alignment of positions. It is a set of positions \(s_k\), for \(k\) from 1 to \(N\), for the common patterns within the sequences. The objective of this alignment is to identify the “best” or most probable, common pattern. By locating the alignment that maximizes the ratio of the corresponding pattern probability to background prob-
ability, this pattern is generally identified. The algorithm is initialized from random starting positions within various sequences. Then it proceeds through many iterations to execute the following steps of the Gibbs sampler. (1) Predictive update step: One of the \( N \) sequences, \( q_i \), is chosen either randomly or in a specified order. The pattern description \( s_{i,j} \) and background frequencies \( p_j \) are then calculated from the current positions \( a_k \) in all sequences excluding \( z \). (2) Sampling step: Within sequence \( z \), every possible segment of width \( W \) is considered as a possible instance of the pattern. The probabilities \( P_x \) of generating each segment \( x \) according to the current pattern probabilities \( q_{i,j} \) are calculated for every possible pattern, similarly, the probabilities \( Q_x \) of generating these segments by the background probabilities \( p_j \) are also calculated. The weight \( A_x = Q_x/P_x \) is assigned to each segment \( x \), and from each segment weighted this way, a random one is selected (segment \( x \) is chosen with probability \( A_x/\sum_j A_j \), where the sum is taken over all possible segments). The position then updates to the new \( a_z \). This simple two step iterative procedure constitutes the basic algorithm of Gibbs sampler.

**Machine learning approaches**

Liu et al. proposed a self-organizing neural network structure for motif finding in DNA and protein sequences [36]. The neural network contains several layers and each layer performs classifications at different level. The authors maintained a low computational complexity in practical scenario. They developed this through the use of layered structure so that each pattern's classification is performed with respect to a small subspace of the whole input space. The authors also ensured a high reliability of their search algorithm using the self-organizing neural network. It is designed in a way that it will grow as needed to make sure that all input patterns are considered and are given the same amount of significance. From practical experiment results, the authors reported that their algorithm outperformed the algorithms of MEME and Gibbs Sampler in certain criterion. The algorithm also works well for long DNA sequences.

The algorithm FMGA (Finding Motifs by Genetic Algorithm) by Liu et al. was one of the initial approaches to find motif using machine learning technique. The algorithm was based on genetic algorithms (GAs) [36] and it was developed for finding potential motifs in the regions located from the -2000 bp upstream to +1000 bp downstream of the transcription start site. The mutation in ge-
netic algorithm is achieved by using position weight matrices to reserve the completely conserved positions. To produce the optimal child pattern, crossover is implemented with specially designed gap penalties. This algorithm also uses a rearrangement method based on position weight matrices. It is for avoiding the presence of a very stable local minimum, which may make it difficult for the other candidates to generate the optimal pattern. According to the author of the original work, FMGA performs better in comparison to MEME and Gibbs sampler algorithms.

### 3.1.4 Multiple longest common subsequence for motif finding

The multiple longest common subsequence (MLCS) problem, i.e., the problem of finding the longest common subsequences among a set of sequences. MLCS is also a well studied problem in theoretical computer science and has applications in computational genomics. MLCS was proved NP-hard by Maier[16] and dynamic programming algorithms are known that run in time $O(n^d)$, for $d$ sequences with maximum length, $n$. Hakata et al. [37] gave an algorithm for MLCS in 1998. This paper proposed a method for computing efficiently the LCS between three and more strings of small alphabet size, evaluates its theoretical time complexity, and estimates the computing time by computational experiments. Using this method, the LCS problem for eight strings of more than 120 length could be solved in about 40min on a slow workstation. Wang et al. [12] gave a dominant point based algorithm with the divide and conquer approach to compute the dominant points and designed a Quick-DP algorithm using those points and later Yang et al. [38] presented a progressive algorithm with efficient parallelization. In 2016, Li et al. [13] proposed a new efficient parallel MLCS algorithm for long and large-scale sequences alignments. Among the newest works, in 2017, Peng et al. [3] provided a novel efficient graph model for the multiple longest common subsequences(MLCS) problem. As MLCS problem is widely used in protein and genome sequence analysis, further improvement of the algorithm can contribute significantly in the studies of computational genomics.
Chapter 4

Methodologies

In this chapter we will analyze the computational complexity of the Common Motifs with Gaps (CMG) problem. Then we will introduce our algorithm and the implementation procedure. Although this problem is treated as a hard problem in the literature and is handled accordingly, to the best of our knowledge this is the first attempt to prove the hardness thereof formally. We will show a reduction from the MLCS problem. MLCS is an already known NP-hard problem [16]. And the reduction from MLCS in the sequel allows us to adapt the branch and bound (B&B) algorithm for MLCS to solve the problem of finding common motifs with gaps.

4.1 Complexity of Common Motifs with Gaps

**Theorem 1** CMG problem is NP-complete.

*Proof.* To show that the decision version of CMG ∈ NP, for a given set $S = \{S_1, S_2, \ldots, S_d\}$ of sequences, a sub-sequence $s^j$ of common factors, and an integer $m > 1$, we can easily check in polynomial time whether $s^j$ consists of $m$ or more factors and satisfy the length constraints if any, and whether the factors in $s^j$ appear in the right order in every sequence of $S$.

We next prove MLCS $\leq_p$ CMG which shows that CMG is NP-hard. Given an instance of MLCS over an alphabet $\Sigma$ given by sequences $T = \{T_1, T_2, \ldots, T_d\}$, we construct an instance of CMG, $S = \{S_1, S_2, \ldots, S_d\}$ over the alphabet $\{A, C, G, T\}$ such that there exists an LCS of length $k$ for set $T$ if,
and only if, $S$ has a CMG of $k$ factors as follows:

1. First we relabel the characters of the MLCS instance using integers from $\{1, \ldots, |\Sigma|\}$ and convert each integer into its binary form.

2. We then replace ‘0’s and ‘1’$s$ by ‘A’s and ‘C’s respectively to get strings over $\{A, C\}$ for each integer.

3. Finally we put these strings in the same order in each $S_i$ as the corresponding integers appeared in $T_i$ separated by ‘G’s in $S_1$ and ‘T’s in all other $S_i$ and we set minimum factor length $p = \lceil \log |\Sigma| \rceil$.

Following is an example of the construction:

$$
\begin{align*}
\text{a b c d} &\quad 1\ 2\ 3\ 4 \quad 001\ 010\ 011\ 100 \quad \text{AAC G ACA G ACC G CAA} \\
\text{b a c d} &\Rightarrow 2\ 1\ 3\ 4 \Rightarrow 010\ 001\ 011\ 100 \Rightarrow \text{ACA T AAC T ACC T CAA} \\
a\ c\ b\ d &\quad 1\ 3\ 2\ 4 \quad 001\ 011\ 010\ 100 \quad \text{AAC T ACC T ACA T CAA}
\end{align*}
$$

Now we show that this transformation of $T$ into $S$ is a reduction.

First suppose that $T$ has a solution, that is a sequence $t^l$ of $k$ characters are present in every sequence of $T$ in exactly the same order. These characters, i.e., integers in $t^l$ will correspond to substrings of each $S_i$ and these substrings will form a sequence $s^l$ of $k$ factors. $s^l$ is a common motif sequence with $k$ factors since according to the construction the factors must appear in each $S_i$ in exactly the same order and there must be a gap of length at least one between any two factors.

Conversely, suppose, $S$ has a common gapped motif sequence $s^l$ with $k$ factors, $k > 1$, that follows a certain order in every sequence $S_i$. Note that since ‘G’ was used as the separator in $S_1$ and ‘T’ was used in all other strings, each factor must be strings over $\{A, C\}$ and corresponds to an integer in the LCS instance by construction, and they will follow the same order in every sequence giving us a common subsequence of length $k$ in all strings in $T$.

Q
4.2 Overview of B&B for MLCS and CMG

As both of the problems of our interest are proved to be NP-hard, the objective of our algorithm is to provide a solution that will find the longest sequence of common gapped motifs among a set of DNA sequences in an efficient way. In doing so, our algorithm will initially provide solution for multiple longest common sub-sequences. After that modifying the basic algorithm, we will find the solution for longest common gapped motif. The principal input to the B& B algorithm for MLCS and CMG is a set of DNA sequences. Its principal output for MLCS is the length of the longest common subsequence for that set. And for CMG it is the length of the longest common gapped motif sequence, the one that maximizes the number of factors. In a nutshell, the B&B algorithm is a combination of

- A preprocessing step.
- Some pruning conditions.
- A branch and bound algorithm for MLCS based on that pruning conditions.

The first step of the algorithm is the preprocessing step. The objective of this step is to analyze the DNA sequences and generate some auxiliary data structures to help guide the search through the remaining path. In this step, some three element candidate lists are generated based on their initial occurrences. Later, the initial list guides to generate more candidate lists for every individual common elements.

The second phase of the algorithm constitutes developing some bounding conditions for reducing the search space. With the increase of the length of the sequences, the possible search space for common motifs increases exponentially. As we are approaching towards a branch and bound algorithm, a bounding condition is an essential component of that algorithm. The pruning or bounding methods help us to eliminate unpromising path from our search space.

The third phase is the algorithm itself. Based on the initial data structures it recursively explores every path for the possible longest common sub-sequence. But it expands its search space based on the pruning conditions, so that, unpromising paths do not get evaluated.
4.3 A branch and bound algorithm for MLCS problem

Algorithms are known for both MLCS and CMG problems that run in time polynomial in lengths of sequences and exponential in the number of sequences. Complexity results in [16] and in this thesis indicate that asymptotically faster algorithms are unlikely. However, search space may be reduced by pruning leading to faster algorithms in practice. Here we present branch and bound algorithms for both MLCS and CMG problems.

Given a set of sequences, \( S = \{S_1, S_2, \ldots, S_d\} \), where \(|S_i| \leq n\) for \(1 \leq i \leq d\), we preprocess the sequences to explore promising paths first and prune the search space using the value of the best solution found so far and the upper bound on values of solutions the path being explored may lead to. The algorithm uses memoization to avoid redundancy of work, i.e., the values of subproblems already calculated are stored in a table, \( V \) indexed by vectors of indices into the sequences.

Preprocessing

We preprocess the sequences to generate candidate lists that will be used to decide in what order nodes are visited during the search process. The candidate list for the first element of the longest common subsequence, \( C_{\text{init}} \), consisting of triples \(<\text{element}, \text{minMultiplicity}, \text{minDistance}>\), is generated as follows:

1. Process each sequence, \( S_i \) and list each element \( e \), the distance of its first occurrence to the end of the sequence, \( d_{e,i} \) and the number of times it appears in the sequence, \( m_{e,i} \).

2. Intersect the lists to get elements common in all sequences. When we intersect we retain the minimum of distances to ends of the sequences for an element, and the minimum multiplicity of the element. Therefore, \( \text{minMultiplicity} \) and \( \text{minDistance} \) entries corresponding to \( \text{element} \), which appears in all the sequences, are given by:

\[
\text{minMultiplicity} = \min_{1 \leq i \leq d} m_{e,i}
\]

\[
\text{minDistance} = \min_{1 \leq i \leq d} d_{e,i}
\]
3. Sort the triples in descending order of minimum distance to ends of sequences and record the sum of multiplicities.

The elements will be explored according to the order in the candidate list, the intuition being an element more distant to the ends of the sequences has more room for other elements to follow it in the LCS. Similarly, for each element $x$ that appears $K_x$ times in all the sequences, we construct lists $C_{x,k}$ for $1 \leq k \leq K_x$ of triples corresponding to elements that follow the $k$-th occurrence of $x$ in all the sequences. For a finite alphabet, each such list can be constructed in time $O(nd)$ and since there can be at most $n$ such lists, preprocessing takes $O(n^2d)$ time.

**Branch and bound**

At each node, we take as input a vector of indices $\hat{I} = <i_1, i_2, \ldots, i_d>$ and a common subsequence, $\alpha$ of the sequences $S_1[1 \ldots i_1], S_2[1 \ldots i_2], \ldots, S_d[1 \ldots i_d]$, i.e., common subsequence up to $\hat{I}$. We also maintain the best solution found so far globally. We start at $<0, \ldots, 0>$ with the common subsequence $\varepsilon$.

Then, at each node, we do the following:

1. Look up the last character and its multiplicity in $\alpha$ and retrieve the corresponding candidate list.

2. Iterate through the $<element, minMultiplicity, minDistance>$ triples in the candidate list.

3. Estimate upper bound (see **Pruning conditions** discussed shortly) to check if the branch can be pruned.

4. Find positions $\hat{P} = <p_1, p_2, \ldots, p_d>$ in each sequence following the input indices where $element$ occurs. Note that such positions may not exist in some sequences as the $k$-th occurrence of $x$ in $\alpha$ may correspond to a position in $S_i$ to the right of the position of the $k$-th occurrence of $x$ in $S_i$. We skip such entries.

5. If $<p_1, p_2, \ldots, p_d>$ has already been computed, then look up the value. Otherwise, explore $<p_1, p_2, \ldots, p_d>$ and update the best solution if needed.
Pruning conditions

Suppose we are considering for exploration a triple, $<element, minMultiplicity, minDistance>$ and suppose this would be the $A$-th occurrence of $element$ in the common subsequence. The following properties can be used to calculate an upper bound on the maximum possible value, $\bar{v}$ in the subtree rooted at the node:

1. $\bar{v}$ can not exceed $1 + minDistance$ since there are only $minDistance$ elements after $element$ in at least one of the sequences.

2. Similarly, sum of multiplicities of $C_{element,A}$ is an upper bound on the number of elements that can follow $element$ in the common subsequence.

3. Let $y = element$ and $p_i(y, A)$ be the position of the $A$-th occurrence of $y$ in the $i$-th sequence. Now $V[p_1(y, A), \ldots, p_d(y, A)]$ is an upper bound on $\bar{v}$ because the position in $S_i$ that corresponds to the $A$-th occurrence of $y$ in the common subsequence must be greater than or equal to $p_i(y, A)$ for $1 \leq i \leq d$. 
The algorithm

The algorithm is summarized in Algorithm 4.1.

Algorithm 4.1 MLCS

1: Initialize: bestSolution ← 0
2: MLCS-B&B (<0, . . . , 0 >, σ)
3: procedure MLCS-B&B(\hat{I}, α)
4: x ← lastElement(α)
5: k ← multiplicity(α, x)
6: v ← 0
7: for each <y, m, d> ∈ C_{k,x} do
8: A ← multiplicity(α,y)+1
9: \tilde{v} ← min(d + 1, multiplicitySum(C_{α,x}) + 1, V[p_1(y, A), . . . , p_d(y, A)])
10: if \tilde{v} + |α| > bestSolution then
11: \hat{P} ← getNextPos(\hat{I}, y)
12: if \hat{P} is valid then
13: if V[\hat{P}] is not null then
14: v ← max(v, 1 + V[\hat{P}])
15: else
16: v ← max(v, 1 + MLCS-B&B (\hat{P}, α, y))
17: if v + |α| > bestSolution then
18: bestSolution ← v + |α|
19: V[\hat{I}] ← v
20: Return v
An illustrative example

A simulation of the algorithm on an example is shown in Figure 4.1.

Input:  
\[ S_1 : ABCXBCY Z \]  
\[ S_2 : ABXY CZXBC \]  
\[ S_3 : ABCXY BCZBC \]  
\[ S_4 : ABXXCCY ZBC \]  

Preprocessing:  
\[ C_{\text{init}} : \langle A; 1; 7 \rangle; \langle B; 2; 6 \rangle; \langle C; 2; 4 \rangle; \langle X; 1; 4 \rangle; \langle Y; 1; 1 \rangle; \langle Z; 1; 0 \rangle \]  
\[ C_{A;1} : \langle B; 2; 6 \rangle; \langle C; 2; 4 \rangle; \langle X; 1; 4 \rangle; \langle Y; 1; 1 \rangle; \langle B; 1; 1 \rangle; \langle Z; 1; 0 \rangle \]  
\[ C_{B;1} : \langle C; 2; 4 \rangle; \langle X; 1; 4 \rangle; \langle Y; 1; 1 \rangle; \langle B; 1; 1 \rangle; \langle Z; 1; 0 \rangle \]  
\[ C_{X;1} : \langle C; 1; 2 \rangle; \langle Y; 1; 1 \rangle; \langle B; 1; 1 \rangle; \langle Z; 1; 0 \rangle \]  
\[ C_{C;1} : \langle B; 1; 1 \rangle; \langle Z; 1; 0 \rangle; \langle C; 1; 0 \rangle \]  
\[ C_{B;1} \cdot C_{X;1} : \langle C; 1; 0 \rangle \]  
\[ C_{X;1} \cdot C_{C;1} : \langle Y; 1; 1 \rangle \]  
\[ C_{Z;1} = f^g \]  
\[ C_{C;1} = f^g \]  

Branch and bound:  
\[ \text{MLCS: } ABCBC \]  

Figure 4.1: Simulation of Algorithm 4.1 on an example set

In this example, 4 strings \( S_1, S_2, S_3, S_4 \) with variable lengths have been considered. First of all, these strings have been pre-processed to generate the candidate lists. The \( C_{\text{init}} \) list is containing the initial candidate list. A candidate list is a list of triple. These triples are consisted of all the common elements, their minimum distance and minimum multiplicity. After constructing the \( C_{\text{init}} \) list, rest of the other lists are constructed beginning from their corresponding positions. After getting all the candidate lists for all the elements, the Branch & Bound algorithm is applied on the pre-processed list. Initially, the solution string is empty and best solution is 0. The simulation starts from the first node of the \( C_{\text{init}} \) list. Then, the solution string follows the most promising path until reaching to an empty list. When the pruning condition meets, that branch is discarded. Finally, the largest path starting from root is considered as solution. Here, in this example, the best solution is provided by the path with length 5 and the string is \( ABCBC \). When, simulation reaches to the point of next \( C \) after \( X \), there the branch does not extends as \( 3 + 1 < 5 \). Again at the position of next \( Y \), if the branch is
extended the path length will be $3 + 2 = 5$, which is also not greater than 5, so, that branch is also pruned. So, final result is found from the path with length 5.

### 4.3.1 A branch and bound algorithm for CMG

We now extend the method discussed above and present a branch and bound algorithm for finding common motifs with gaps (CMG). We are given a set of strings, $S = \{S_1, S_2, \ldots, S_d\}$, and integers $p, q$ giving upper and lower bounds on factor lengths respectively.

The first step is to find common factors. Then we use an approach similar to the one for finding MLCS taking into account that there may be multiple factors overlapping a position in a string, at most one of which can be present in the final solution.

**Identifying common factors**

We first identify common factors, i.e., substrings with lengths between $p$ and $q$ that appear in all the strings and record start indices (and end indices implicitly) of their occurrences in every string. This can be done efficiently using approaches such as suffix trees [39], finite automata [40]. For each string $S_i$, where $1 \leq i \leq d$, we create a list of factors, $F_i$ consisting of all occurrences of all the common factors in the string sorted in ascending order of their end indices.

**Candidate list generation**

The factor lists are then processed to generate candidate lists in a similar approach to the one used for preprocessing MLCS instances. In this context, a factor, $f_1$ will be in the list of candidates to follow the $k$-th occurrence of factor $f_2$ if in each factor list there is an entry for $f_1$ with start index exceeding the end index of $k$-th occurrence of factor $f_2$ by at least 2. The candidate lists are sorted in descending order of minimum distances of factors from the ends of factor lists.

**Branch and bound and pruning conditions**

The branch and bound algorithm now proceeds as the one for finding MLCS producing a common motif with highest number of factors.
An illustrative example

Figure 4.2 shows simulation of the algorithm for finding common motifs with gaps on a sample instance. The factor lists consist of pairs denoting the factor string and its start position sorted in increasing order of their end positions i.e. sum of the start positions and the lengths of factors. Note that although TCG follows TG in each of the factor lists, it is not included in the list of candidates to follow the first occurrence of TG since it overlaps with TG in two of the strings. We also see that although TGC is a common factor of the strings, it does not appear in the final common motif with gap as that would lead a motif with only one factor. However, substrings of TGC are considered as factors - TG in S1 and S3 and GC in S2 - to obtain a common motif with three factors.

Input:
S1: tCcCTuCAuTn
S2: TnCTATuATCTuT
S3: mCAuATATnC

\[ p = 2; q = 3 \]

Factor lists:
\[ F1: (\langle \langle T \rangle \langle T \rangle; 1 \rangle; < nC; 3 ); < T n; 5 ); < T nC; 5 ); < nC; 6 ); < T n; 10 ) \]
\[ F2: (\langle \langle T \rangle \langle T \rangle; 1 \rangle; < T n; 1 ); < T nC; 1 ); < nC; 2 ); < T n; 6 ); < T n; 10 ) \]
\[ F3: (\langle \langle T \rangle \langle T \rangle; 2 \rangle; < T n; 5 ); < T n; 9 ); < T nC; 9 ); < nC; 10 ) \]

Candidate Lists:
\[ C_{\text{init}}: (\langle \langle T \rangle \langle T \rangle; 2 \rangle; 3 ); < nC; 1 ; 2 ); < T nC; 1 ; 1 ) \]
\[ C_{TGC; 1}: (\langle \langle T \rangle \langle T \rangle; 1 ; 0 ) \]
\[ C_{GC; 1}: (\langle \langle T \rangle \langle T \rangle; 2 ; 1 ) \]
\[ C_{TGC; 1}: fg \]
\[ C_{TGC; 2}: fg \]

Branch and bound:

\[ \text{CMG: } nC \{ T n \{ T n \} \]
candidate list generation, the Branch & Bound algorithm is applied on the list starting from the root node. The simulation produces the top down parsing tree. The longest path in this tree is giving the common gapped motif sequence and all the other unpromising paths are getting pruned. Here the longest path is the path with three factors $GC - TG - TG$.

### 4.4 Summary

In this chapter, we proved that CMG problem is NP-complete. We then provided an exact algorithm for solving the MLCS problem and a slight modified version of that algorithm that provides exact solution for CMG problem also. We also discussed about different programming strategies and language support for practical implementation of the algorithm.
Chapter 5

Results

This chapter presents the experimental results of the algorithm. We compared the performance of MLCS B&B algorithm with four state of the art approaches in terms of time and space requirements. We also implemented and represented results of the CMG algorithm. However, no other existing approach have showed practical implementation results for the finding common motifs with gaps problem until now. So, it was not possible to compare the experimental results with other state of the art approaches.

Though the program is implemented and tested in a single threaded environment with sequential computation, it has some great advantages over other existing approaches that are implemented and tested in a high performance computing setup with multithreading. When, the sequence length is gradually increased, the branch and bound method for computing MLCS do not exceed the performance of existing methods in terms of time. But it shows tremendous improvement in space over other existing approaches at that input sequences. On the other hand, if the no of input is gradually increased, keeping the sequence length fixed, the performance of the B&B algorithm shows slow increase in time, whereas the other existing methods show rapid incremental growth. Similarly, the memory consumption of the B&B algorithm is limited in comparison to the exponential growth of the existing methods.

In this chapter, the performance of both the MLCS and CMG algorithm will be analyzed thoroughly under different performance measuring metrics. First, the experimental setup will be discussed, next the evaluation of the algorithm under various lengths of sequences will be represented in terms of time
and memory. After that, the evaluation of the algorithm under various numbers of sequences will be discussed again with respect to time and memory. Finally, Evaluation of the CMG algorithm will be represented.

5.1 Experimental Setups

We choose random DNA sequences with alphabet size $|\Sigma| = 4$. Since the MLCS method can be applied to many areas of bioinformatics and computational genomics as well outside the biological domain, the sequence representations of the objects from each application domain as well as the distributions of the letters in the sequences can be drastically different. Therefore, to make an unbiased assessment of our algorithms, we first used as test set a set of strings randomly and independently generated from the alphabet. We conduct two kinds of experiments:

- Evaluation under various lengths of sequences: for each type of the sequences, the number of sequences is fixed to 5, but the length of the sequences increases from 10 to 100.

- Evaluation under various numbers of sequences: the number of sequences used increases from 5 to 200 and the length of all used sequences is fixed to 100.

For each test, in the experiments, all the test algorithms are run on a HP PROBOOK 450 G4 with processor Intel Core i5 7200U and Processor Clock Speed - 2.50-3.10GHz with 8 GB RAM. The operating system is GNU/Linux (64 bit), and all the algorithms are implemented with C/C++ and julia 1.1.

5.2 Implementation details

In this section, we will discuss about the programming languages and programming strategies that we have applied for the implementation of the algorithm.
5.2.1 Programming Languages

C++

We have used standard C++ language with g++ compiler support for the prepossessing part of the algorithm. We have used the STL library of C++ as our primary data structure. First we have developed a suffix array data structure for finding the common factors. Then using those common factors we have found the minimum multiplicity and minimum distance for each factor. Finally we represented the pre-processed list as an array of vector like data structure. Here is a sample of how our data structure looks like after pre-processing.

Figure 5.1: Pre-processed list of common factors
Julia 1.1

We have implemented and executed the B&B algorithm for MLCS using version 1.1 of julia language. We have used the multi-core and distributed processing facilities provided by julia language for the parallel execution of the B&B algorithm. The detail of the language can be found in this paper by Bezanson et al. [41].

In our implementation of the MLCS B&B algorithm, we wanted to spawn a separate process for each element in the $C_{init}$ list. But practically for gaining speed up we initiated a separate process with the element which has the largest remaining distance and largest minimum multiplicity. As the total data structure has a finite dimension, so the no of recursive call in worst case can also be bounded by an upper bound. So, no process can get the optimal result after reaching that maximum no of function call. So, there we put another bounding condition based on the best result and no of process call. If the best result doesn’t changes for certain no of recursive call, after crossing the maximum bound of process call, the process execution terminates concurrently.

5.3 Measuring Performances

In this section, performance measurement with respect to time and memory will be represented as a tabular form and as scatter plot. We will measure these performances under both type of evaluation metrics.

5.3.1 Evaluation under various lengths of sequences

The major objective of these research is to design a better intuitive solution for the MLCS and CMG problem. Performance of same algorithm can vary depending on hardware resources. So, optimzing the solution from the algorithmic logic and data structure level can play a crucial role in optimizing the overall performance of the solution.
Table 5.1: The average running times (sec) of the test algorithms under different lengths of DNA with the number of sequences fixed at 5 [3].

<table>
<thead>
<tr>
<th>No of Input</th>
<th>FAST_LCS</th>
<th>Quick-DP</th>
<th>TOP_LCS</th>
<th>Leveled-DAG</th>
<th>B&amp;B</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.57</td>
<td>0.13</td>
<td>0.038</td>
<td>0.026</td>
<td>0.8</td>
</tr>
<tr>
<td>100</td>
<td>2.7</td>
<td>1.4</td>
<td>0.23</td>
<td>.96</td>
<td>96</td>
</tr>
<tr>
<td>200</td>
<td>244.1</td>
<td>10.6</td>
<td>8.5</td>
<td>6.8</td>
<td>50</td>
</tr>
<tr>
<td>300</td>
<td>4064.8</td>
<td>95.3</td>
<td>38.7</td>
<td>32.6</td>
<td>60</td>
</tr>
<tr>
<td>400</td>
<td>-</td>
<td>312.4</td>
<td>77.8</td>
<td>59.5</td>
<td>60</td>
</tr>
<tr>
<td>500</td>
<td>-</td>
<td>1,566.9</td>
<td>132.6</td>
<td>112.2</td>
<td>1800</td>
</tr>
<tr>
<td>600</td>
<td>-</td>
<td>4384.1</td>
<td>201.1</td>
<td>165.3</td>
<td>4000</td>
</tr>
<tr>
<td>700</td>
<td>-</td>
<td>10,347.5</td>
<td>287.3</td>
<td>223.4</td>
<td>4800</td>
</tr>
<tr>
<td>800</td>
<td>-</td>
<td>27489.2</td>
<td>373.2</td>
<td>313.8</td>
<td>7200</td>
</tr>
<tr>
<td>900</td>
<td>-</td>
<td>-</td>
<td>487.3</td>
<td>399.1</td>
<td>12000</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
<td>-</td>
<td>644.7</td>
<td>513.5</td>
<td>14400</td>
</tr>
</tbody>
</table>

As in this research very simple and easily available hardware resources are used for implementation, so here the performance of the algorithm is mainly bounded by the length of the sequences.
Figure 5.2: Average running time for input sequences with variable lengths

We compared the result with another existing sequential algorithm in literature named FAST LCS. The trend of growth in running time is quite alike both for B&B and FAST LCS. On the other hand the space requirement for B&B algorithm is just the opposite to FAST LCS algorithm. With increasing length of sequences, the B&B algorithm consumes very limited space and the growth is reasonably slow. Whereas, in FAST LCS, the growth for memory shows incremental trend.
CHAPTER 5. RESULTS

Table 5.2: The memory requirement (MB) of the test algorithms under different lengths of DNA sequences with the number of sequences fixed at 5 [3]

<table>
<thead>
<tr>
<th>Input Length</th>
<th>FAST_LCS</th>
<th>Quick_DP</th>
<th>Top_MLCS</th>
<th>Leveled_DAG</th>
<th>B&amp;B</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>47</td>
<td>56</td>
<td>21</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>100</td>
<td>1352</td>
<td>1481</td>
<td>99</td>
<td>82</td>
<td>22</td>
</tr>
<tr>
<td>200</td>
<td>8331</td>
<td>8652</td>
<td>2353</td>
<td>1469</td>
<td>22</td>
</tr>
<tr>
<td>300</td>
<td>16874</td>
<td>16993</td>
<td>4050</td>
<td>3051</td>
<td>30</td>
</tr>
<tr>
<td>400</td>
<td>-</td>
<td>27355</td>
<td>5866</td>
<td>4787</td>
<td>31</td>
</tr>
<tr>
<td>500</td>
<td>-</td>
<td>41257</td>
<td>8297</td>
<td>6654</td>
<td>31</td>
</tr>
<tr>
<td>600</td>
<td>-</td>
<td>60912</td>
<td>12063</td>
<td>8598</td>
<td>31</td>
</tr>
<tr>
<td>700</td>
<td>-</td>
<td>85733</td>
<td>18550</td>
<td>10163</td>
<td>49</td>
</tr>
<tr>
<td>800</td>
<td>-</td>
<td>126483</td>
<td>26070</td>
<td>14250</td>
<td>300</td>
</tr>
<tr>
<td>900</td>
<td>-</td>
<td>-</td>
<td>36341</td>
<td>20539</td>
<td>550</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
<td>-</td>
<td>49442</td>
<td>27985</td>
<td>619</td>
</tr>
</tbody>
</table>

Figure 5.3: Memory as a function of input length
5.3.2 Evaluation under various numbers of sequences

The second metric that has been considered for measuring performances is the different number of input sequences. In this case, the performances of B&B algorithm is compared to some existing approaches that are implemented using effective parallelization. The surprising fact is B&B algorithm is implemented using same sequential procedure. As the number of input increases the running time of B&B algorithm keeps decreasing. Whereas, in every other approach the running time shows exponential increment with the increasing number of input. So, B&B algorithm outperforms every other approach with respect to running time while increasing the number of input gradually.

Table 5.3: The average running times (sec) of the test algorithms under different number of DNA sequences with the length of sequences fixed at 100 [3].

<table>
<thead>
<tr>
<th>Length</th>
<th>FAST_LCS</th>
<th>Quick_DP</th>
<th>TOP_LCS</th>
<th>LEVELED_DAG</th>
<th>B&amp;B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2.9</td>
<td>1.5</td>
<td>0.12</td>
<td>0.082</td>
<td>96</td>
</tr>
<tr>
<td>10</td>
<td>25671.4</td>
<td>3385.4</td>
<td>30.0</td>
<td>25.3</td>
<td>102</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>64.8</td>
<td>51.2</td>
<td>89</td>
</tr>
<tr>
<td>30</td>
<td>-</td>
<td>-</td>
<td>136.7</td>
<td>96.7</td>
<td>82</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>-</td>
<td>250.3</td>
<td>191.4</td>
<td>100</td>
</tr>
<tr>
<td>50</td>
<td>-</td>
<td>-</td>
<td>463.2</td>
<td>380.1</td>
<td>100</td>
</tr>
<tr>
<td>60</td>
<td>-</td>
<td>-</td>
<td>665.4</td>
<td>530.3</td>
<td>60</td>
</tr>
<tr>
<td>70</td>
<td>-</td>
<td>-</td>
<td>1088.1</td>
<td>875.5</td>
<td>78</td>
</tr>
<tr>
<td>80</td>
<td>-</td>
<td>-</td>
<td>1684.6</td>
<td>1233.2</td>
<td>90</td>
</tr>
<tr>
<td>90</td>
<td>-</td>
<td>-</td>
<td>2217.9</td>
<td>1764.6</td>
<td>80</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>-</td>
<td>3041.5</td>
<td>2417.8</td>
<td>84</td>
</tr>
</tbody>
</table>
The space requirement of the B&B algorithm is also very low compared to other approaches. In B&B approach, the memory increases slowly with the increase of input. Whereas all the other approaches show incremental growth of space with time. In B&B approach, the memory requirement for input sequences with length 200 is 96 MB. While in TOP_LCS it is 252247 MB and in LEVELED_DAG it is 163948 MB. A chart for space requirement is given below for input sequences ranging from 20 to 200.
**Table 5.4:** The average memory requirement (MB) of the test algorithms under different number of DNA sequences with the length of sequences fixed at 100 [3].

<table>
<thead>
<tr>
<th>Length</th>
<th>FAST_LCS</th>
<th>Quick_DP</th>
<th>TOP_LCS</th>
<th>LEVELED_DAG</th>
<th>B&amp;B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1358</td>
<td>1485</td>
<td>93</td>
<td>85</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>35769</td>
<td>36934</td>
<td>5813</td>
<td>5232</td>
<td>14</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>32329</td>
<td>28126</td>
<td>21</td>
</tr>
<tr>
<td>30</td>
<td>-</td>
<td>-</td>
<td>48765</td>
<td>39291</td>
<td>28</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>-</td>
<td>67813</td>
<td>52607</td>
<td>35</td>
</tr>
<tr>
<td>50</td>
<td>-</td>
<td>-</td>
<td>91128</td>
<td>68103</td>
<td>42</td>
</tr>
<tr>
<td>60</td>
<td>-</td>
<td>-</td>
<td>121268</td>
<td>87359</td>
<td>49</td>
</tr>
<tr>
<td>70</td>
<td>-</td>
<td>-</td>
<td>156470</td>
<td>118600</td>
<td>89</td>
</tr>
<tr>
<td>80</td>
<td>-</td>
<td>-</td>
<td>197387</td>
<td>141859</td>
<td>120</td>
</tr>
<tr>
<td>90</td>
<td>-</td>
<td>-</td>
<td>209145</td>
<td>146402</td>
<td>150</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>-</td>
<td>229372</td>
<td>151386</td>
<td>200</td>
</tr>
<tr>
<td>200</td>
<td>-</td>
<td>-</td>
<td>252247</td>
<td>163948</td>
<td>300</td>
</tr>
</tbody>
</table>
5.4 Discovering Motifs

In this research the second problem that has been considered is finding common motifs with gaps. The CMG problem is also another frequently studied problem of literature. But this is the first approach where the experimental results are represented. In this case, performance of the algorithm is again measured with respect to time and memory. Though the performance of the B&B MLCS algorithm was bounded by the length of sequence, in CMG case performance showed no dependence on length of input. Strings ranging from length 100 upto 2000 are processed in less than 1 sec. Runtime didn’t showed much variation with increasing length of input sequence. Similarly memory also showed very small increment with variable length of input sequence.
Table 5.5: The average running times (sec) of the test algorithm under different lengths of DNA with the number of sequences fixed at 5

<table>
<thead>
<tr>
<th>Length</th>
<th>B&amp;B</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.04</td>
</tr>
<tr>
<td>100</td>
<td>0.1623</td>
</tr>
<tr>
<td>200</td>
<td>0.9456</td>
</tr>
<tr>
<td>300</td>
<td>0.9567</td>
</tr>
<tr>
<td>400</td>
<td>0.8689</td>
</tr>
<tr>
<td>500</td>
<td>0.6890</td>
</tr>
<tr>
<td>600</td>
<td>0.7679</td>
</tr>
<tr>
<td>700</td>
<td>0.7890</td>
</tr>
<tr>
<td>800</td>
<td>0.8673</td>
</tr>
<tr>
<td>900</td>
<td>0.8967</td>
</tr>
<tr>
<td>1000</td>
<td>0.9763</td>
</tr>
<tr>
<td>2000</td>
<td>0.9870</td>
</tr>
</tbody>
</table>
5.4.1 Summary

The performance of the B&B algorithm is quite different under different condition. In this research, both memory and time are considered as performance measurement criteria. In every testing situation B&B outperformed all the other existing algorithms with respect to memory. With respect to time, performance is always bounded by the length of input sequence when different input sequences with different lengths are considered. On the other hand, the performance of the algorithm under fixed length input sequences with varied number of input is always better compared to other existing approaches.
Chapter 6

Conclusion

This thesis has addressed two problems with applications in genomics - finding longest common subsequence of multiple sequences (MLCS) and finding common motifs with gaps (CMG). MLCS is known to be NP-hard and its reduction to CMG in this paper formally proves that CMG is NP-hard as well. While this makes polynomial time algorithms for the problems unlikely, We have proposed a branch and bound algorithm in this thesis. The proposed algorithm has two fundamental characteristics. It has a pre-processing step that formulates a data structure to guide the searching process more efficiently. It also has a pruning condition, that helps to reduce unpromising solution paths and find the optimal solutions. Initially, implementing in a traditional sequential manner it has been found that the algorithm faces difficulties with growing input size. Later we implemented the algorithm using two processes and we carefully considered the case whether the execution is falling into any kind of deadlock situation or not? So, testing the parallel version of the algorithm on different inputs, it has been found that the B&B algorithm performs better than all the other state of the art approaches in terms of memory and no of inputs. Using two threads we also found promising results with variable lengths of inputs. From experimental results, it is found that the performance with respect to time, is highly dependent upon the length of sequence. With increasing length the solution space grows voluminously. Exploring every path for finding the optimal solution turns out to be a highly tedious job. So, finding the optimal path within reasonable time gets critical for large sequences. To solve this real challenge, the algorithm needs to be parallelized effectively with more threads. So, the future works include implementing the algorithm using more threads and measuring
the performance in a high performance computing environment. However, the most attractive side of our algorithm is, the performance becomes better with additional no of input. Where most of the existing algorithm have shown increment in running time with additional input, our branch and bound technique shows just the opposite. For our algorithm, the running time decreases with additional input. On the other hand, memory requirement of our algorithm is also very low. It remains almost constant with increasing input and changes very little with the increment in the length of the input.
Bibliography


