B.Sc. in Computer Science and Engineering Thesis

Algorithms for Finding Planted Motif Search in DNA

Submitted by

Itisha Nowrin
201314026

Majidur Rahman
201314028

Asfaqur Rahman
201314029

Supervised by

Dr. Md. Abul Kashem Mia

Professor

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

Department of Computer Science and Engineering
Military Institute of Science and Technology

January 2017

CERTIFICATION

This thesis paper titled **"Algorithms for Finding Planted Motif Search in DNA"**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in January 2017.

Group Members:

Itisha Nowrin

Majidur Rahman

Asfaqur Rahman

Supervisor:

Dr. Md. Abul Kashem Mia
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

CANDIDATES' DECLARATION

This is to certify that the work presented in this thesis paper, titled, "Algorithms for Finding Planted Motif Search in DNA", is the outcome of the investigation and research carried out by the following students under the supervision of Dr. Md. Abul Kashem Mia, Professor, Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology.

It is also declared that neither this thesis paper nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Itisha Nowrin
201314026

Majidur Rahman
201314028

Asfaqur Rahman
201314029

ACKNOWLEDGEMENT

We wish deep respect and express gratitude to our supervisor, Dr. Md. Abul Kashem Mia, Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, for his constant supervision, affectionate guidance and motivation. We are so grateful for his encouragement and effort and without him this thesis would not have been completed or written. His keen interest on the topic and valuable advices are what made this achievement possible. One simply could not wish for a better or friendlier supervisor. We are very grateful and would like to give thanks to the respected teachers and staffs of the Department of Computer Science and Engineering (CSE) of Military Institute of Science and Technology (MIST) for providing their all possible supports. Finally, we want to dedicate the essence of our purest respect to our parents and also thank our families and our course mates for their appreciable assistance, patience and suggestions during the course of our thesis.

Dhaka
January 2017

Itisha Nowrin

Majidur Rahman

Asfaqur Rahman

ABSTRACT

Identification of rare events happening in a set of biological patterns is an important event in discovering new biological aspects. Finding such patterns called motifs in DNA sequences is one such important issue. For example, regulatory regions in genome such as promoters, enhancers etc. contain motifs that control many biological processes such as gene expression, determination of open reading frames, identification of gene promoter elements, location of RNA degradation signals etc. In this paper, we have researched on algorithm for finding special type of motif named (l, d) motif where integer l indicates the length of the motif to be discovered and integer d indicates the maximum number of mutations (mismatches) allowed in that particular motif. This motif search problem falls in the category of Planted Motif Search (PMS). It takes n strings and two integers l and d as input. So, our target is to find all possible motifs M of length l that appear in each of the input sequences with at most d mutations. Our proposed heuristic approach is a slight improvement of the previously proposed algorithms of PMS8 and qPMS9. We have improved the neighborhood generation technique by rearranging the DNA sequences in descending order according to their profile matrix value before applying the pruning condition. So, the probability of finding the candidate motif, within few iterations , increases significantly. This will improve the runtime of motif search algorithm to a considerable extent. Our heuristic approach, therefore, betters the runtime of previously worked algorithms.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

LIST OF ABBREVIATION

DNA : Deoxyribonucleic Acid
RNA : Ribonucleic Acid
A : Adenine
G : Guanine
C : Cytosine
T : Thymine
PMS : Planted Motif Search
qPMS : Quorum Planted Motif Search

LIST OF SYMBOLS

$\in$	: Elements of set
$\emptyset$	: Empty set
$\leftarrow$	: assign

CHAPTER 1
INTRODUCTION

Bioinformatics is a combination of science fields like computer science, biology, mathematics and engineering. In others words we can say bioinformatics is a management information system for molecular biology and has many practical applications. It utilizes computer algorithm for collecting, storing, analyzing and integrating biological data and genetic macromolecules.

1.1 Overview of Planted Motif Search

Finding rare events happening in a set of DNA or protein sequences could lead to new biological discoveries. Such kind of rare events is the presence of motif in DNA or protein. DNA and protein are two important part of a living organism. DNA holds the characteristics and stores heredity information with in the cells. It is a molecule that encodes the genetic instructions used in the development and functioning of all known living cells. Proteins do most of the work and required for the structure, functions and regulation of the bodys tissue.

1.1.1 Identifying Motif

Genes are turned on or off by regulatory proteins. These proteins bind to upstream regulatory regions of genes to either attract or block an RNA polymerase. Regulatory protein binds to a short DNA sequence called a motif. So finding the same motif in multiple genes regulatory regions suggests a regulatory relationship amongst those genes. In motif finding problem the complications are:

- We do not know the motif sequence
- We do not know where it is located relative to the genes start
- Motifs can differ slightly from one gene to the next
- How to discern it from random motifs

Now if a given sample of DNA is:
cctgatagacgctatctggctatccacgtacgtaggtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgatacgtacgtacaccggcaacctgaaacaaacgctcagaaccagaagtgc
aaacgtacgtgcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt
agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatcttacgtacgtataca
ctgttatacaacgcgtcatggcggggtatgcgttttggtcgtcgtacgctcgatcgttaacgtacgtc

In these rows of DNA sequences motif is the pattern that is implanted in each of the individual sequences. The length of the hidden sequence of motif is given. We assume that the standard length of the hidden sequence is 8. The pattern is not exactly the same in each array because random point mutations may occur in the sequences.
If there are no mutation in the then the pattern :
cctgatagacgctatctggctatcc**acgtacgt**aggtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgat**acgtacgt**acaccggcaacctgaaacaaacgctcagaaccagaagtgc
aa**acgtacgt**gcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt
agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatctt**acgtacgt**ataca
ctgttatacaacgcgtcatggcggggtatgcgttttggtcgtcgtacgctcgatcgtta**acgtacgt**c

acgtacgt- This is called Consensus String.
If there are two points mutation in the then the pattern :
cctgatagacgctatctggctatcc**aGgtacTt**aggtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgat**CcAtacgt**acaccggcaacctgaaacaaacgctcagaaccagaagtgc
aa**acgtTAgt**gcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt
agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatctt**acgtCcAt**ataca
ctgttatacaacgcgtcatggcggggtatgcgttttggtcgtcgtacgctcgatcgtt**CcgtacGt**c

1.1.2 Profile Matrix

To define a motif, let us say we know the starting index of the motifs in DNA sequence. The motif start positions can be represented as s=(s_1, s_2, s_3, ..., s_t).
We, then, line up their patterns according to their start indexes s=(s_1, s_2, s_3, ..., s_t).
Alignments

aGgtacTt
CcAtacgt
acgtTAgt
acgtCcAt
CcgtacgG

Profile Matrix

A	3 0 1 0 3 1 1 0
C	2 4 0 0 1 4 0 0
G	0 1 4 0 0 0 3 1
T	0 0 0 5 1 0 1 4

1.1.3 Consensus

We can find the consensus string from profile matrix. We check the highest score in each column of profile matrix. We will get a letter consisting the highest score for each column. By concataning all these letters according to column number, we will get the consensus string.
so the Consensus is

A C G T A C G T

1.2 This Thesis Studies the Following Problem

Recognizing patterns in Molecular biological is a complicated process. In our thesis we study the (l, d) motif or Planted Motif Search (PMS). Where there are n input strings . This process returns all sequences of motif that have length l and they are mutated in d positions. In a set of DNA or protein sequences detection of rare events can lead to a new biological discoveries. One kind of such rare events in DNA sequences is the present of pattern called motif. Discovering motifs is a challenging problem because finding motifs have been proven to be uncompromising. For finding the motif problem initially we had to study a lot about the DNA sequences and all other factors related to motif problems. We initially implemented the brute force technique, branch and bound technique, median string search for finding the motif. There are lots of other algorithm which had discovered later. Motif prediction is usually the first stage in the process of identifying motifs. An extensive amount of research has been done on this topic over the past twenty years. Among the combinatorial variations, the PMS Problem has been the most widely studied perhaps because it offers a higher level of accuracy in modeling the true motifs than the others. Motifs typically occur with mutations at binding sites. The binding sites are referred to as instances of a motif. From [1] we know about what is motif search initially and all other speed up techniques [2].

1.3 Literature Review

Motif searching is an important step in the detection of rare events occurring in a set of DNA or protein sequences. Detection of rare events happening in a set of DNA/protein sequences could lead to new biological discoveries .One formulation of the motif finding problem is known as (l, d)- motif search or Planted Motif Search(PMS). There are many algorithms on finding the motif in DNA sequence. The most known algorithms are qPMS7, PMS8, qPMS9, TraverStringRef. The general aim of these algorithms is to improve the time complexity and find out more motif with larger sequence and mutations. In pursuit of improving the motif searching technique there are a few steps which plays crucial role in the algorithm and a slight improvement of any of these steps can improve the search result significantly. The main steps are:

- Neighborhood generation
- Pruning condition

Pruning condition reduces the search time by discarding the search in unwanted l-mers. In qPMS7 the qPMSPrune technique was used to prune unwanted l-mers in search. Algorithm qPMSPrune is based on the following observation. Any (l, d, q)-motif of the input strings must be in $B_i(x)$ for some l-mer x in some input string s_i and also it must be a (l, d, q-1)-motif of the input strings excluding s_i.

1.3.1 qPMSPrune

qPMSPrune prunes certain nodes (and their descendants) in $T_d(x)$ that cannot possibly be (l, d, q)-motifs. Let q" be the number of input strings s_j such that $d_H(t,s_j)$ 2d-$ds_H(t,x)$. Observe that if q"$<$ q-1 then none of the nodes in the subtree rooted at node (t,p) could be a (l, d, q-1)-motif. This is because if there is a node (t', p') in the subtree which is a (l, d, q-1)-motif, then there are at least q-1 input strings s_j such that $d_H(t,s_j)$ d. Consider such an input string s_j . By the triangle inequality, $d_H(t,s_j)$ $d_H(t',s_j) + d_H(t,t') \leq d + (d_H(t', x) - d_H(t,x)) \leq 2d - d_H(t,x)$. This inequality will infer that q" ¡ q-1. Therefore, if the condition q" ¡ q-1 occurs, it can safely prune the subtree rooted at node (t,p) without missing any (l, d, q-1)-motif.It is easy to see that the time and space complexities of Algorithm qPMSPrune are O((n-q+1)$nm^2N(l, d)$) and O(nm^2),respectively.

It is easy to see that the time and space complexities of Algorithm qPMSPrune are O((n-q+1)$nm^2N(l, d)$) and O(nm^2), respectively.

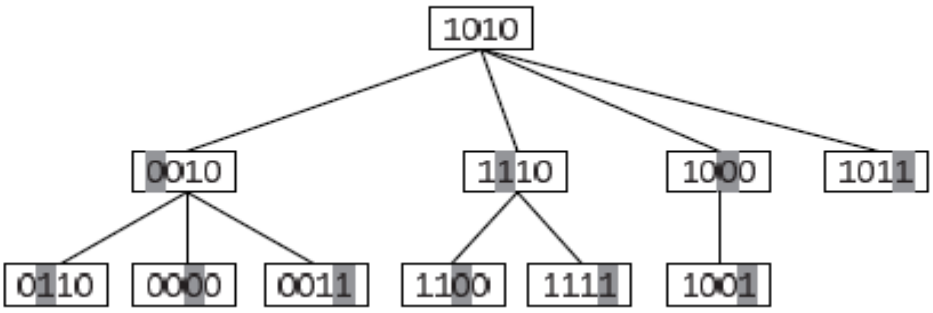

Figure 1.1: Traverse the tree in qPMSPrune

1.3.2 qPMS7

qPMS7 [3] is the improved version of the previous algorithm qPMSPrune. Basically a speedup technique has been used to improved the run time of the motif search. Specifically, the technique will reduce the time taken for computing Hamming distances dH(t,s_j) qPMSPrune. We observe that some l-mers can be ignored without changing the result since we notice that we just need to count q' and q".

The reason for ignoring any l-mer z in sj, as far as a node (t, p) in the tree Td(x) is concerned, if d_H (t, z)>2d - d_H (t, x) is as follows. If this condition occurs, then for any node (t', p') in the subtree rooted at node (t, p) we have: d_H (t', z)$\leq$ d_H (t, z) - d_H (t', t)>2d - dH (t, x) - dH (t', t) = 2d - dH (t', x)$\leq$ d. Therefore, ignoring l-mer z at any node (t, p) in the subtree rooted at node (t, p) will not change its q'. The value of q" at node (t', p') may become smaller as a result of ignoring the l-mer z. However, the pruning condition based on q" in qPMSPrune still holds.

The speedup technique reduces the runtime of Algorithm qPMSPrune drastically because the deeper a node is, the smaller will be the size of its list of surviving l-mers. Note that the number of nodes at a depth of h from the root will be exponential in h. In practice, the runtime of Algorithm qPMSPrune is improved by a factor of around 5 when this technique is used. Both PMS8 and qPMS9 uses the same pruning condition. However improvement on pruning condition is possible and many scientists and researchers are working in this field.

1.3.3 PMS8

The efficiency of PMS8 comes mainly from reducing the search space by using the pruning conditions. PMS8 [4] consists of a sample driven part followed by a pattern driven part. In

Table 1.1: Comparison between qPMS7 and qPMS8

Instances	qPMS7	PMS8[1]	PMS8[16]	PMS8[32]	PMS8[48]
(13,4)	29s	7s	3s	2s	2s
(15,5)	2.1m	48s	5s	4s	3s
(17,6)	10.3m	5.2m	22s	12s	9s
(19,7)	54.6m	26.6m	1.7m	52s	37s
(21,8)	4.87h	1.64h	6.5m	3.3m	2.2m
(23,9)	27.09h	5.48h	21.1m	10.7m	7.4m
(25,10)	-	15.45h	1.01h	30.4m	20.7m
(26,11)	-	-	-	-	46.9h

Table 1.2: Travers length of the strings and Comparison between PMS8 and qPMS9

Instances	TraverStringRef	qPMS7	PMS8
(13,4)	14 s	7 s	6 s
(15,5)	55 s	48 s	34 s
(17,6)	3.5 m	5.2 m	2.7 m
(19,7)	14.5 m	26.6 m	13.4 m
(21,8)	59.8 m	1.64 h	45.4 m
(23,9)	4.08 h	5.48 h	2.26 h
(25,10)	17.55 h	15.45 h	6.3 h

the sample driven part we generate tuples of l-mers originating from different strings. In the pattern driven part we generate the common d-neighborhood of such tuples. Initially we build a matrix R of size n (m-l+1) where row i contains all the l-mers in S_i. We pick an l-mer x from row 1 of R and push it on a stack. We filter out any l-mer in R at a distance greater than 2d from x. Then we pick an l-mer from the second row of R and push it on the stack. We filter out any l-mer in R that does not have a common neighbor with the l-mers on the stack; then we repeat the process. If any row becomes empty, we discard the top of the stack, revert to the previous instance of R and try a different l-mer. If the stack size is above a certain threshold we generate the common d-neighborhood of the l-mers on the stack. For each neighbor M we check whether there is at least one l-mer u in each row of R such that Hd(M, u) d. If this is true then M is a motif. In table 3.5 we can see the time comparison between qPMS7 and PMS8.To know more about string and closest string [5, 6] can be helpful.

1.3.4 qPMS9

qPMS9, a parallel exact qPMS algorithm that offers significant runtime improvements on DNA and protein datasets. qPMS9 solves the challenging DNA (l, d)-instances (28, 12) and (30, 13). The qPMS9 algorithm extends PMS8 in several ways. In table 1.2 we can see the comparison of the given length of the string between PMS8 an qPMS9 method. First, qPMS9 introduces a search procedure which significantly increases performance by allowing for better pruning of the search space. Second, qPMS9 adds support for solving the qPMS problem, which was lacking in PMS8.

In this thesis book we have briefly described our thesis work of finding motif in DNA sequence. And in the later chapters we will see what is motif , how this is formed and how we can find motif by various speedup techniques.

CHAPTER 2
PRELIMINARIES

For better understanding of algorithms on bioinformatics it is essential to know about some essential biological terms and their definition. In this section we will discuss on some of the essential biological terms which are related to our algorithms.

2.1 Definitons

2.1.1 Cell

Every single life is made of fundamental working unit called cell. A cell is the basic structural and biological unit of all living organism. The human body is composed of trillions of cells. They provide structure for the body, take in nutrients from food, convert those nutrients into energy, and carry out specialized functions. Cells also contain the bodys hereditary material and can make copies of themselves. In figure 2.1 we can see the time period of cell division. Every organism composed of different types of cell and they are :

- Prokaryotic cells
- Eukaryotic cells

2.1.2 Genome

In terms of molecular biology genome is genetic material of organism. It consists of a complete set of DNA. To maintain and build a particular organism, the genome contains all the necessary information that are needed. It includes genes and the non-coding sequences of the DNA.

2.1.3 Chromosome

Chromosome is the most organized structure containing DNA of a living organism. Each chromosome is made up of DNA tightly coiled many times around proteins called histones

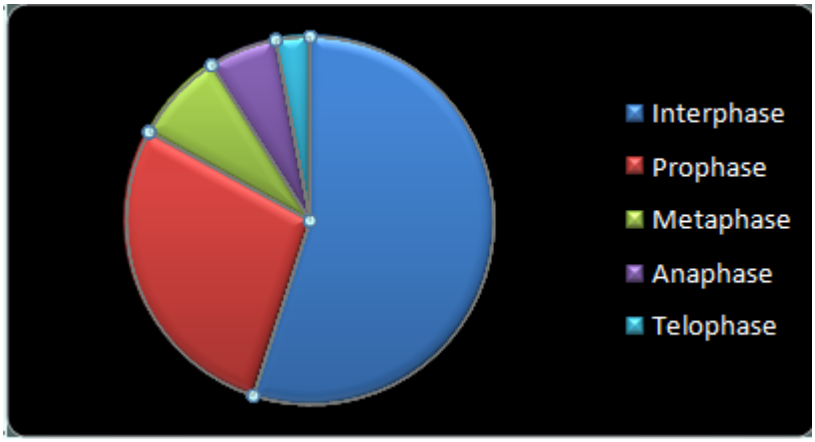

Figure 2.1: Cell cycle pi chart

Figure 2.2: Genome

that support its structure. Chromosomes are not visible in the cells nucleusnot even under a microscopewhen the cell is not dividing. However, the DNA that makes up chromosomes becomes more tightly packed during cell division and is then visible under a microscope. Most of what researchers know about chromosomes was learned by observing chromosomes during cell division. Each chromosome has a constriction point called the centromere, which divides the chromosome into two sections, or arms. The short arm of the chromosome is labeled the p arm. The long arm of the chromosome is labeled the q arm. The location of the centromere on each chromosome gives the chromosome its characteristic shape, and can be used to help describe the location of specific genes. Human genome contains 46 or 23 pairs of distinct chromosome. Which defines the characteristics of that organism. Each chromosome contains of many gene In the figure 2.3 the labels are-

(1) Chromatid

(2)Centromere

(3)Short arm

(4) Long arm

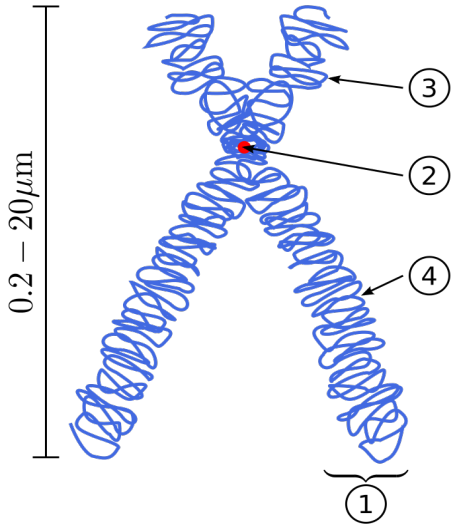

Figure 2.3: Eukaryotic chromosome

2.1.4 Gene

A gene is a region of DNA which consists of basic physical and functional units of heredity. Genes, which are made up of DNA, act as instructions to make molecules called proteins. In humans, genes vary in size from a few hundred DNA bases to more than 2 million bases. The

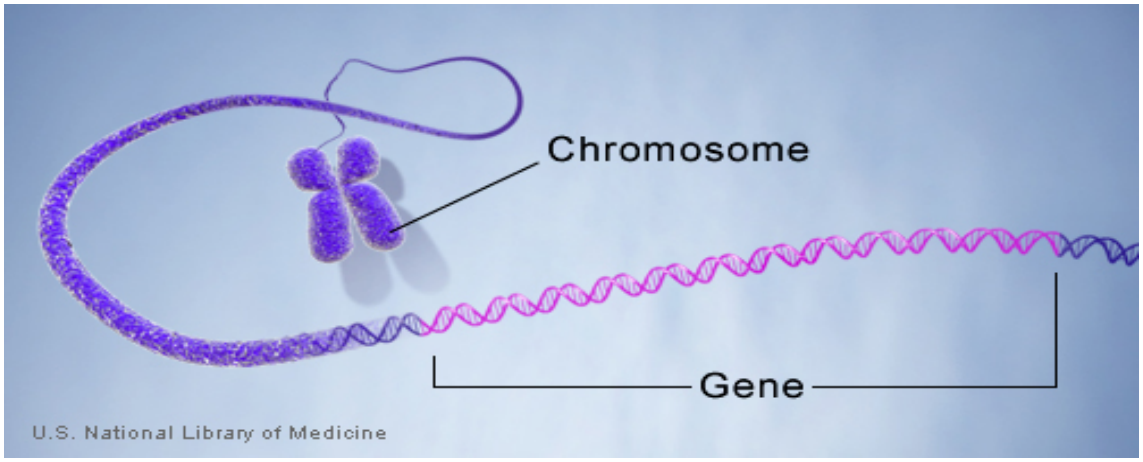

Figure 2.4: Structure of gene

Human Genome Project has estimated that humans have between 20,000 and 25,000 genes. Every person has two copies of each gene, one inherited from each parent. Most genes are the same in all people, but a small number of genes (less than 1 percent of the total) are slightly different between people. Alleles are forms of the same gene with small differences in their sequence of DNA bases. These small differences contribute to each persons unique physical features.In figure 2.4 we can see the structure of gene.

2.1.5 Proteins

Human body needs protein for many important functions of the body, including repairing and building tissue, acting as enzymes, aiding the immune system, and serves as hormone. Each of this functions required different types of protein. In spite of differences in structure in their shape and size, all proteins contain the basic structure. Proteins are long chains of amino acids. Amino acids are the building blocks of protein. In other words, amino acids are like the links in a chain. The chain itself represents the protein molecule. Protein chains are then twisted and folded together in specific ways to create certain molecules. A linear chain of amino acid residues is called a polypeptide. A protein contains at least one long polypeptide. Short polypeptides, containing less than 2030 residues, are rarely considered to be proteins and are commonly called peptides. The individual amino acid residues are bonded together by peptide bonds and adjacent amino acid residues. The sequence of amino acid residues in a protein is defined by the sequence of a gene, which is encoded in the genetic code.

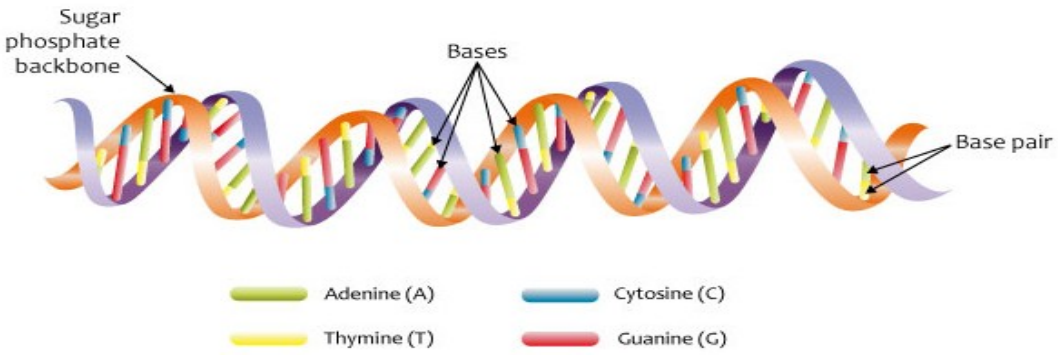

Figure 2.5: DNA double helix structure

2.1.6 DNA

Deoxyribonucleic acid or DNA is a molecule that contains the instructions an organism needs to develop, live and reproduce. These instructions are found inside every cell, and are passed down from parents to their children.

DNA Structure

DNA is made up of molecules called nucleotides. Each nucleotide contains a phosphate group, a sugar group and a nitrogen base. The four types of nitrogen bases are adenine (A), thymine (T), guanine (G) and cytosine (C). The order of these bases is what determines DNA's instructions, or genetic code. Similar to the way the order of letters in the alphabet can be used to form a word, the order of nitrogen bases in a DNA sequence forms genes, which in the language of the cell, tells cells how to make proteins. Another type of nucleic acid, ribonucleic acid, or RNA, translates genetic information from DNA into proteins. The entire human genome contains about3 billion bases and about 20,000 genes.
Nucleotides are attached together to form two long strands that spiral to create a structure called a double helix. The double helix structure can be thought of as a ladder, the phosphate and sugar molecules would be the sides, while the bases would be the rungs. The bases on one strand pair with the bases on another strand: adenine pairs with thymine, and guanine pairs with cytosine.

DNA testing

DNA contains information about heritage, and can sometimes reveal whether the body is at risk for certain diseases. DNA tests, or genetic tests, are used for a variety of reasons, in-

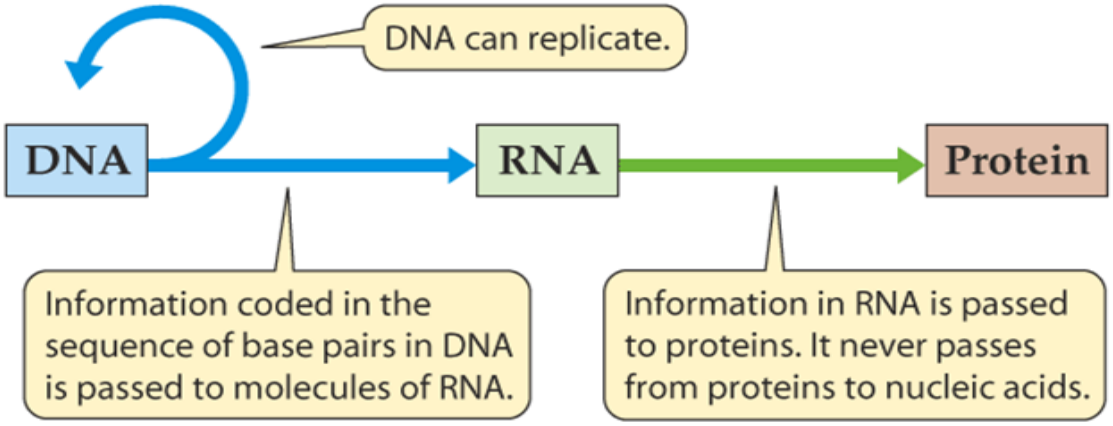

Figure 2.6: The flow of information

cluding to diagnose genetic disorders, to determine whether a person is a carrier of a genetic mutation that they could pass on to their children, and to examine whether a person is at risk for a genetic disease. For instance, mutations in the BRCA1 and BRCA2 genes are known to increase the risk of breast and ovarian cancer, and analysis of these genes in a genetic test can reveal whether a person has these mutations.
Genetic test results can have implications for a person's health, and the tests are often provided along with genetic counseling to help individuals understand the results and consequences of the test.

2.1.7 RNA

One of a group of molecules similar in structure to a single strand of DNA. T(hyamine) is replaced by U(racil)The function of RNA is to carry the information from DNA in the cell's nucleus into the body of the cell, to use the genetic code to assemble proteins, and to comprise part of the ribosomes that serve as the platformon which protein synthesis takes place. Several types exist, classified by function.

- mRNA this is what is usually being referred to when a Bioinformatician says "RNA". This is used to carry a genes message out of the nucleus.
- tRNA transfers genetic information from mRNA to an amino acid sequence
- rRNA ribosomal RNA. Part of the ribosome which is involved in translation
- Eukaryotic cells

2.2 Other Methodologies

2.2.1 Mutation

A Mutation occurs when a DNA gene is damaged or changed in such a way as to alter the genetic message carried by that gene. A Mutagen is an agent of substance that can bring about a permanent alteration to the physical composition of a DNA gene such that the genetic message is changed.Once the gene has been damaged or changed the mRNA transcribed from that gene will now carry an altered message. The polypeptide made by translating the altered mRNA will now contain a different sequence of amino acids. The function of the protein made by folding this polypeptide will probably be changed or lost. In this example, the enzyme that is catalyzing the production of flower color pigment has been altered in such a way it no longer catalyzes the production of the red pigment.

Normal DNA sequence : **TATCTAG**
Mutated DNA sequence:**ATCGAG**

2.2.2 Transcription of DNA

Transcription is the first step of gene expression, in which a particular segment of DNA is copied into RNA. DNA is double-stranded, but only one strand serves as a template for transcription at any given time. The process of transcription begins when an enzyme called RNA polymerase attaches to the template DNA strand and begins to catalyze production of complementary RNA. Transcription is highly regulated. Most DNA is in a dense form where it cannot be transcribed.

To begin transcription requires a promoter, a small specific sequence of DNA to which

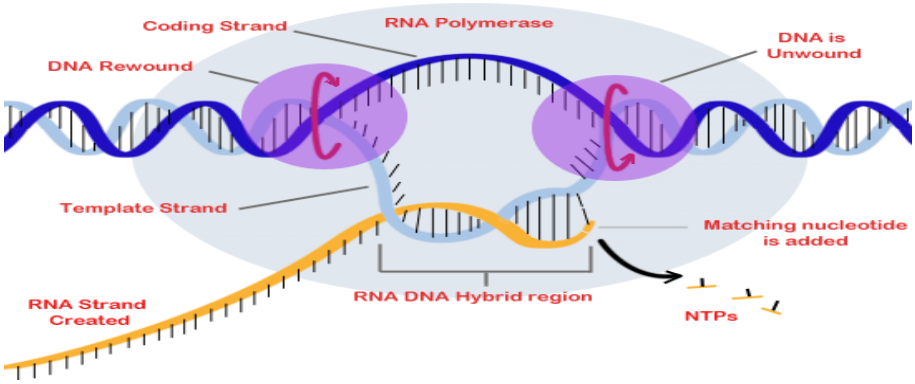

Figure 2.7: Transcription of DNA

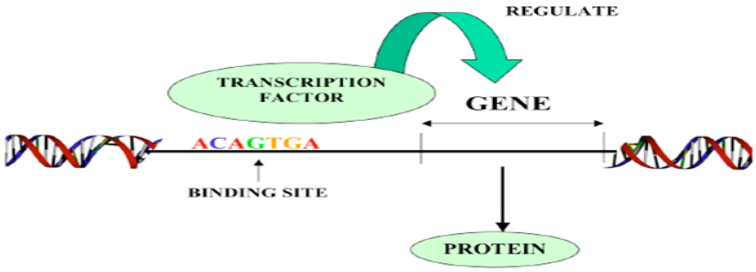

Figure 2.8: Motifs (transcription factor binding sites).

polymerase can bind (40 base pairs ”upstream” of gene). Finding these promoter regions is a partially solved problem that is related to motif finding. There can also be repressors and inhibitors acting in various ways to stop transcription. This makes regulation of gene transcription complex to understand.

2.2.3 Transcription Factor

Transcription factors are proteins involved in the process of converting DNA into RNA. Transcription factor can be located anywhere within the Regulatory Region and may vary slightly across different regulatory regions since non-essential bases could mutate.From [7] we know about the transcription factor.

2.2.4 Regulation

One of the mechanisms through which protein levels in the cell are controlled is through transcriptional regulation. A regulatory sequence is a segment of a nucleic acid molecule which is capable of increasing or decreasing the expression of specific genes within an organism.

2.2.5 Motifs

Every gene contains a regulatory region (RR) typically stretching 100-1000 bp upstream of the transcriptional start site. Located within the RR are the Transcription Factor Binding Sites (TFBS), also known as motifs, which are specific for a given transcription factor. A motif can be located anywhere within the regulatory region. Motifs may vary slightly across different regulatory regions since non-essential bases could mutate.
Genes are turned on or off by motifs. So, we can say they act like a switch; a switch which turns on genes or turns off genes. We know more about motif and mutations from [8–11]

CHAPTER 3
ALGORITHMS FOR FINDING MOTIF IN DNA

We started working on some existing algorithms initially to get the basic idea of motifs and which are the primary levels for studying DNA motif. We started implementing the initial algorithms with C. At first we implemented those algorithms that did not allow mutation such as brute force method and branch and bound method. After that we implemented median string search method that allows mutations.
Then we started studying some advanced algorithm for finding motif. We implemented qPMS7 that uses pruning condition. Then we try latest motif searching algorithms PMS8 and qPMS9 and implement it that uses neighborhood generation technique. Finally we apply some heuristic rules in neighborhood re-arrangement to see if there is any improvement or not. We have also read others algorithm [12, 13] but they are very old technique.

3.1 Algorithms We Worked On

3.1.1 Brute force method

In brute force method the goal is to find a set of l-mers, one from each sequence, that maximizes the consensus score. The set of DNA is given. Some notations are :

- t - number of sample DNA sequences
- n - length of each DNA sequence
- DNA - sample of DNA sequences (t x n array)
- l - length of the motif (l-mer)
- s_i - starting position of an l-mer in sequence i
- s=(s_1, s_2,..... s_t) - array of motifs starting positions

The input of this method is t x n matrix of DNA, and l, the length of the pattern to find. The output of this method is an array of t starting positions

s = (s_1, s_2,.....,s_t) that maximizes the Score(s,DNA). The solution of this method is to compute the scores for each possible combination of starting positions s. The best score will determine the best profile and the consensus pattern in DNA The goal of this method is to maximize Score(s,DNA) .

Algorithm 1 BruteForceMotifSearch(DNA, t, n, l)

```
1: bestScore ← 0
2: for each s = (s_1, s_2, ...., s_t)from(1, 1....., 1)to(n − l + 1, ....., n − l + 1) do
3:     if Score(s,DNA)>bestScore then
4:         bestScore ← score(s, DNA)
5:         bestMotif ← (s_1, s_2, ....., s_t)
6:     end if
7: end for
```

Varying (n-l+1) positions in each of t sequences, we are looking at (n-l+1)^t sets of starting positions. For each set of starting positions, the scoring function makes l operations,so complexity is l (nl+1)^t = O(l n^t) That means that for t = 8, n = 1000, l = 10 we must perform approximately 1024 computations. So by the above description we understood that it would take billion of years.

3.1.2 Median String Search

In median string search problem mutation is allowed. Here a set of t DNA sequences is given. The goal is to find a pattern that appears in all t sequences with the minimum number of mutations. This pattern will be the motif. Here hamming distance is used. It is denoted as d_H(v,w). It is the number of nucleotide pairs that do not match when v and w are aligned. For example :

$$d_{\mathrm{H}}(AAAAAA, ACAAAC) = 2$$

Here in this method for each DNA sequence i, compute all d_H(v, x), where x is an l-mer with starting position s_i ,($1 \leq s_i \leq$ n l+ 1). We have to find minimum of d_H(v, x) among all l-mers in sequence i. The function TotalDistance(v, DNA) is the sum of the minimum Hamming distances for each DNA sequence i and TotalDistance(v, DNA) = mins d_H(v,s).
The goal of this method is to find a median string, where set of DNA sequence is given. The input is a txn matrix DNA, and l, the length of the pattern to find. The output is a string v of l nucleotides that minimizes TotalDistance(v,DNA) over all strings of that length.

The Median String Problem needs to examine all 4^l combinations for v. This number is relatively smaller.

Algorithm 2 MedianStringSearch (DNA, t, n, l)

```
    bestWord ← AAA....A
2:  bestDistance ← inf
    for l-mer s := AAAAtoTTT....T do
4:      if TotalDistance(s, DNA) < bestDistance then
            bestWord ← s
6:      end if
    end for
```

3.1.3 qPMSPrune

qPMSPrune is the first algorithm that comes up with the d-neighborhood concept and it is the base of all the new qPMS algorithms.

Definition 1 A string x=x[1].....x[l] of length l is called an l-mer.

Definition 2 Given two x=x[1].....x[l] and s=s[1].....s[m] with l<m, we use the notation x∈l s if x is a contiguous substring of s. In other words, x∈l s if there exists 1≤i≤(m-l+1) such that x[j]=s[j+i-1] for every 1≤j≤l. We also say that x is an l-mer in s.

Definition 3 Given two strings x=x[1].....x[l] and Y=y[1].....y[l] of equal length, the Hamming distance between x and y, denoted by d_H(x,y), is the number of mismatches between them. In other words, $d_H(x,y)=\Sigma_{1\leq i\leq l}I_i$ where Ii is the indicator at position i. I_i=1 if x[i]≠y[i], and Ii=0 otherwise.

Definition 4 Given two strings x and s with |x|<|s|, the Hamming distance between x and s, denoted by d_H(x,s).

Definition 5 Given a set of n strings s1,... ,n_n of length m each, a string M of length l is called an (l, d, q)-motif of the strings if there are at least q out of the n strings such that the Hamming distance between each one of them and M is no more than d. M is called an (l, d, q)-motif for short if the set of strings is clear. Definition 6: Given a string x=x[1].....x[l], we define the d-neighborhoodof x, B_d(x), to be y—d_H(x,y)≤d.

Algorithm qPMSPrune is based on the following observation. Any (l, d, q)-motif of the input strings must be in Bd(x) for some l-mer x in some input string si and also it must be a (l,d,q-1)-motif of the input strings excluding si . This observation can be rewritten formally as follows.

Observation 1. Let M be any (l, d, q)-motif of the input strings s1, . . . ,sn. Then there exists an i (with 1≤i≤n) and a l-mer xsi sucht hat M is in Bd (x) and M is a (l,d,q-1)-motif of the input strings excluding s_i.

The above observation suggests the following algorithm. Compute Bd (x) for every l-mer x in each input string si for 1≤i≤n. For each l-mer in the neighborhoods thus computed,

check if it is a (l,d,q-1)-motif of the Input strings excluding si . This simple algorithm can be improved further as shown in [13]. The key observation is that it is sufficient to consider each input string si for $1 \leq i \leq n-q+1$.

Observation .2 Let M be any (l, d, q)-motif of the input strings $s_1,....,s_n$. Then there exists an i (with $1 \leq i \leq n-q+1$) and a l-mer $x \in s_i$ such that M is in $B_(d)$ and M is a (l,d,q-1)-motif of the input strings excluding s_i.
Algorithm qPMSPrune is based on the above observation. For any l-mer x, it represents $B_d(x)$ as a tree T_d (x) using the following

It is easy to see that the time and space complexities of Algorithm qPMSPrune are O((n-q+1)nm^2 N (l, d)) and O(nm^2), respectively.

3.2 qPMS7

qPMS7 is faster technique than qPMSPrune. Algorithm qPMS7 is a generalized version of Algorithm qPMSPrune. Recall that Algorithm qPMSPrune considers one l-mer x in a specific input string si at a time. Algorithm qPMS7 extends Algorithm qPMSrune by considering two l-mers x and y in two different input strings $s_i$and s_j . An observation similar to that of Algorithm qPMSPrune can be obtained as follows.

In the qPMS7 algorithm a set of DNA strings that likely contains transcription factor-binding sites is given, we propose a general framework to find them. The framework consists of two phases. The first phase will select a set of motifs by repeatedly calling the qPMS Algorithm on different values of l,d,and q. The second phase will use a scoring function to eliminate some of the motifs returned in the first phase, and then identify the transcription factor-binding sites based on the surviving motifs. In the first phase we employ different values, ranging between lmin and lmax, for the length l of motifs, where d_{max}, where d_{max} is another user-specified parameter, and call the best qPMS algorithm to (let it be Algorithm A) find (l, d, q)-motifs. In this process, if some (l, d, q)-motif(s) are found, we add them to the set of motifs. The pseudo-code of the first phase follows.

Input a set of strings.
Parameters: lmin,lmax,dmax and q.
Output a set of (l, d, q)-motifs M.

In the second phase, we sort the (l, d, q)-motifs according to their scores and pick the top k motifs. The following pseudocode describes the second phase.

Algorithm 3 Phase I: selecting candidate motifs

```
M ← ∅
for l = l_min → l_max do
    for d = 0 to d_max do
        Run the fastest qPMS Algorithm A to find (l, d, q)-motifs of the input strings
        if algorithm A take too long then
            Terminate algorithm A
            break the for loop d
        end if
        Let M(l, d, q) be the set of (l, d, q)-motifs returned by
        if M(l, d, q) is NOT empty then
            M←M(l, d, q)
            break for loop of d
        end if
    end for
end for
```

Inputa set of strings and a set of (l, d, q)-motifs M.

Parameters: a scoring function and k.

Outputa set of binding sites on the input strings.

Algorithm 4 Phase II: selecting candidate motifs

```
Sort (l, d, q)-motifs in M according to the scoring function
Pick the top k (l, d, q)-motifs in M after sorting
for each picked (l, d, q) motif M do
    for each input string s_i do
        : Identify all the l-mers z in s_i such that d_H(M,z)≤d
        : Output the location of each such l-mer z in s_i as a transcription factor binding site
    end for
end for
```

3.3 PMS8

PMS8 can efficiently solve instances with large l and instances with large d. The efficiency of PMS8 comes mainly from reducing the search space by using the pruning conditions presented later in the paper, but also from a careful implementation which utilizes several speedup techniques and emphasizes cache locality.

PMS8 consists of a sample driven part followed by a pattern driven part [14]. In the sample driven part we generate tuples of l-mers originating from different strings. In the pattern

driven part we generate the common d-neighborhood of such tuples. Initially we build a matrix R of size n (m-l+1) where row icontains all the l-mers in S_i. We pick an l-mer x from row 1 of R and push it on a stack. We filter out any l-mer in R at a distance greater than 2d from x. Then we pick an l-mer from the second row of R and push it on the stack. We filter out any l-mer in R that does not have a common neighbor with the l-mers on the stack; then we repeat the process. If any row becomes empty, we discard the top of the stack, revert to the previous instance of R and try a different l-mer. If the stack size is above a certain threshold (see section on Memory and Runtime) we generate the common d-neighborhood of the l-mers on the stack. For each neighbor M we check whether there is at least one l-mers in each row of R such that $H_d(M, u) \leq d$. If this is true then M is a motif.

Definition 1. Let T be a set of l-mers, where k =|T|. For every i, the set T_1[i], T_2[i],....,Tk[i] is called the i-th column of T. Let mi be the maximum frequency of any character in column i.Then Cd(T) $\Sigma^{k}_{ii=1}$k-mi is called the consensus total distance of T.

Theorem 1. Let T be a set of 3 l-mers and d_1, d_2, d_3 be non-negative integers. There exists a l-mer M such that $H_d(M,T_i) \leq d_i$, $\forall_i$, $1 \leq i \leq 3$ if and only if the following conditions hold:

- **Case 1**$C_d(T_i, T_j) \;\; d_i + d_j$, $\forall_i$, j, $1 \leq i < j \leq 3$
- **Case 2**$C_d(T) \leq d_1 + d_2 + d_3$

Case 1 is shown in 3.2 and case 2 is shown in 3.3

3.4 qPMS9

qPMS9 is the most efficient among all other motif searching algorithm. It's based upon the PMS8 algorithm with additional heuristic support in sorting candidate motifs which makes it more efficient.

Methods First lets define the PMS and qPMS problems more formally. A string of length l

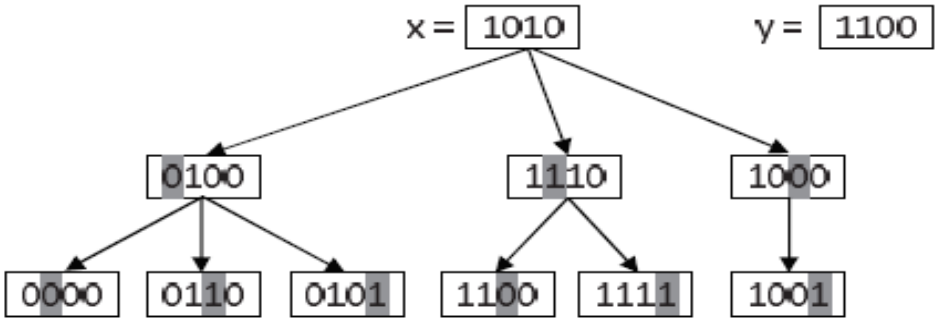

Figure 3.1: Traverse the tree in qPMS7

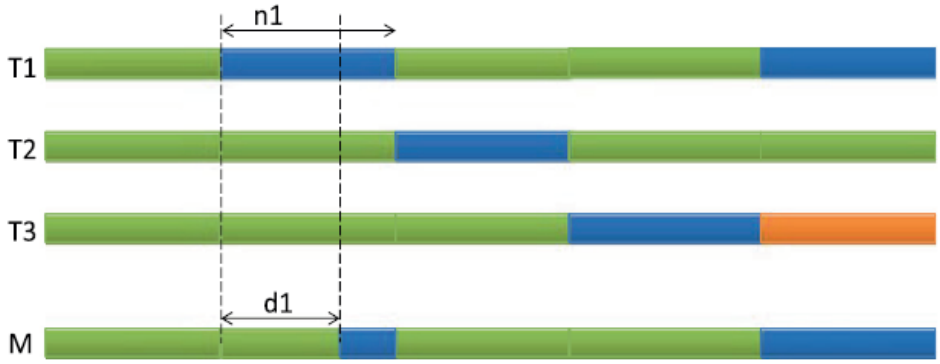

Figure 3.2: Case 1 for theorem 1

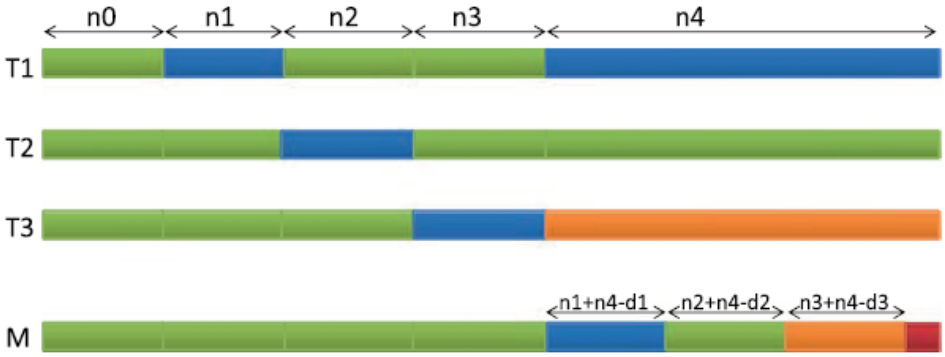

Figure 3.3: Case 2 for theorem 1

Algorithm 5 GenerateNeighborhood(T, r, p)

```
1:  if p ≤ l then
2:      if notprune(T, r) then
3:          for α ∈ Σ do
4:              x_p := α
5:              for i = 1 →| T | do
6:                  T'_i := T_i[2... | S_i |]
7:                  r'_i := r_i
8:                  if (T_i[0] ≠ α) then
9:                      r'_i := r'_i − 1
10:                 end if
11:             end for
12:             GenerateNeighborhood(T', r', p + 1)
13:         end for
14:     end if
15: else
16:     reportl = nerx
17: end if
```

is called an l-mer. Given two l-mers u and v, the number of positions where the two l-mers differ is called their Hamming distance and is denoted as $H_d(u, v)$. For any string T, we denote the substring of T starting at position i and ending at position jby T[i...j]

Definition 1. The PMS problem: Given n sequences s_1, s_2,...., s_3, over an alphabet Σ, and two integers l and d, identify all l-mers M, $M \in \Sigma^l$, such that $\forall_i$, $1 \leq i \leq n$, $\exists j_i$, $1 \leq j_i \leq |si|l+1$, such that H_d (M, s_i [j_i.....j_i+l 1])$\leq$d.

Definition 2. The qPMS problem: same as the PMS problem, however the motif appears in at least q% of the strings, instead of all of them. PMS is a special case of qPMS for which q = 100%. Another useful notion is that of a d-neighborhood. Given a tuple of l-mers T= (t_1,..., t_k) the common d-neighborhood of T includes all the l-mers r such that $H_d(r,t_i) \leq$ d, $1 \leq i \leq$ s.

Now lets define the consensus l-mer and the consensus total distance for a tuple of l-mers. The consensus l-mer for a tuple of l-mers T=(t_1,..., t_k) is an l-mer u where u[i] is the most common character among (t_1[i],..., t_k[i]) for each $1 \leq i \leq l$. If p is the consensus l-mer for Tthen the consensus total distance of T is defined as $C_d = \Sigma_{u \in} H_d(u,p)$. While the consensus string is generally not a motif, the consensus total distance provides a lower bound on the total distance between any motif and a tuple of l-mers.

Tuple Generation In the sample driven part of PMS8, tuples T = (t_1, t_2,....., t_k), where t_i is an l-mer from string s_i, $\forall$i = 1....k, are generated based on the following principles. First, if T has a common d-neighborhood, then every subset of T has a common d-neighborhood. Second, for a motif to exist, there has to be at least one l-mer u in each of the remaining strings s_k + 1, s_k + 2,....., s_n such that Tłu has a common d-neighborhood. We call such l-mers u alive with respect to tuple T. As we add l-mers to T we update the alive l-mers and reorder the strings in increasing order of the number of alive l-mers. This reordering reduces the running time because it leads to generating fewer tuples overall.

qPMS9change the criteria by which the strings are reordered, as follows. Let T be the current tuple of l-mers and let u be an alive l-mer with respect to T. If we add u to T, then the consensus total distance of T increases.We can see tuple generation from [4, 15] .We compute this additional distance C_d(Tłu) C_d(T). For each of the remaining strings, we compute the minimum additional distance for any alive l-mer in that string. Then we sort the strings decreasingly by the minimum additional distance. Therefore, we give priority to the string with the largest minimum additional distance. If two strings have the same minimum additional distance, we give priority to the string with fewer alive l-mers. The intuition is that larger additional distance could indicate more diversity among the l-mers in the tuple, which means smaller common d-neighborhoods. We invoke the algorithm as GenTuples(, k, R) where the matrix R contains all the l-mers in all the input strings, grouped as one row per string.We know more about d-neighborhood from [16].

Algorithm 6 GenerateTuples(T, k, R)

Input:= $(t_1, ..., t_i)$, current tuple of l-mers
k, desired size of the tuple
R, array of n-i rows, where R_i contains all live l-mers from string si+j
Result: Generates tuples of size k and passes them to the GenerateNeighborhood function;
begin
if |T|==k **then**
    GenerateNeighborhood(T, d);
    return;
end if
for u$\in R_1$ **do**
    $T' := T \cup u$;
    for $j \leftarrow 1 \rightarrow n - i - 1$ **do**
        R'$_j$=$v \in R_{j+1}d$-neighborhood for T'$\cup$v
        if $\mid R'_j \mid$==0 **then**
            continue oyerloop;
        end if
        $minAdd$:=min$_{v \in R'_j}$Cd(T'$\cup v$)-Cd(T')
        aliveLmers:=$\mid s_{i+j+1} \mid - \mid R'_j \mid$;
        sortKey[j] := (minAdd,- aliveLmers);
    end for
    sort R' decreasingly by sortKey;
end for
GenerateTuples (T', k, R');

Neighborhood Generation For every tuple T, obtained as described in the previous section, we generate the common d-neighbors of the l-mers in the tuple. In qPMS9, the neighbor generation uses the same process as in PMS8. For the sake of completeness, we briefly review the process.

Given a tuple T = (t_1,t_2,...., t_k) of l-mers, we want to generate all l-mers M such that H_d(t, M)$_i \leq$ d,$\forall_i$ = 1....k. We traverse the tree of all possible l-mers. A node at depth r, which represents an r-mer, is not explored deeper if certain pruning conditions are met. Necessary and sufficient conditions for 2 and 3 l-mers to have a common neighbor are given in Ref. 7. The same paper gives necessary conditions for more than 3 l-mers to have a common neighbor. The interested reader is referred to the PMS8 paper7 for a more in depth description of neighborhood generation.

Adding Quorum Support The algorithm is extended to solve the qPMS problem.In the qPMS problem, while generating tuples, some of the strings can be skipped entirely. This translates to the implementation as follows:

Algorithm 7 GenerateTuples($qTolerance, T, k, R$)

Input: qTolerance, number of strings we can
T, current tuple of l-mers;
i, last string processed;
k, desired size of the tuple;
R=($R_1, ..., R_{n-i}$), where R_j contains all live l-mers in s_{i+j} ;
Result: Generate tuples of size k and pass them on to the GenerateNeighborhood function;
begin
if |T|==k **then**
    GenerateNeighborhood(T, d);
    return;
end if
for u$\in R_1$ **do**
    $T' := T \cup u$;
    $incompat$:=0
    for $j \leftarrow 1 \rightarrow n - i - 1$ **do**
        R'$_j$=$v \in R_{j+1}d$-neighborhood for T'$\cup$v
        if $\mid R'_j \mid$==0 **then**
            if $incompat \geq qTo, erence$ **then**
                continue outerloop;
            end if
            $incompat$++;
        end if
        $minAdd$:=min$_{v \in R'_j}$Cd(T'$\cup v$)-Cd(T')
        aliveLmers:=$\mid s_{i+j+1} \mid - \mid R'_j \mid$;
        sortKey[j] := (minAdd,- aliveLmers);
    end for
    sort R' decreasingly by sortKey;
    QGenerateTuples ($qToleranceincompat, T', k, R'$);
end for
GenerateTuples (T', k, R');
QGenerateTuples($qTolerance1, T, k, R'\ R_1$);

in the PMS version we successively try every alive l-mer in a given string by adding it to the tuple T and recursively calling the algorithm for the remaining strings.For the qPMS version there is an additional step where, if the value of q permits, it skips the current string and try l-mers from the next string. At all times we keep track of how many strings we have skipped. We invoke the algorithm as QGenerateTuples(n-Q +1,$\{\ \}$, 0 , k ,R) where Q=$\left\lfloor \frac{qn}{100} \right\rfloor$ and R contains all the l-mers in all the strings.

Parallel Algorithm

In PMS8 the search space is split into m = $|s1|\ \ l + 1$ independent subproblems P_1, P_2, , P_m, where P_i explores the d-neighborhood of l-mer s_l [i..i + l 1]. In the parallel implementation, processor 0 acts as both a master and a worker, the other processors are workers. Each worker requests a subproblem from the master, solves it, then repeats until all subproblems have been solved. Communication between processors is done using the Message Passing Interface (MPI). In qPMS9, we extend the previous idea to the q version. We split the problem into subproblems where r = n-Q + 1 and $\left\lfloor \frac{qn}{100} \right\rfloor$. Problem P_{ij} explores the d-neighborhood of the j-th l-lmer in string s_i and searches for l-mers M such that there are Q-1 instances of M in strings s_{i+1} ,....., s_n . Notice that Q is fixed, therefore subproblems P_{ij} get progressively easier as i increases. We know more about parallel implementation from [17, 18].

3.5 Our Contribution

Suppose we pick an l-mer like 'agctagct'. If we allow 1 mutation only, we will get 24 mutated strings. These strings are as follows:

ggctagct	agatagct	agctggct	agctagat
cgctagct	aggtagct	agctcgct	agctaggt
tgctagct	agttagct	agcttgct	agctagtt
aactagct	agcaagct	agctaact	agctagca
acctagct	agcgagct	agctacct	agctagcg
atctagct	agccagct	agctatct	agctagcc

Now if we allow 2 mutations only, we will get 252 mutated strings. These strings are as follows:

gatcagtc	gctcagtc	gttcagtc	catcagtc	cctcagtc	cttcagtc
tatcagtc	tctcagtc	tttcagtc	ggacagtc	gggcagtc	ggccagtc
cgacagtc	cggcagtc	cgccagtc	tgacagtc	tggcagtc	tgccagtc
ggtaagtc	ggtgagtc	ggttagtc	cgtaagtc	cgtgagtc	cgttagtc
tgtaagtc	tgtgagtc	tgttagtc	ggtcggtc	ggtccgtc	ggtctgtc
cgtcggtc	cgtccgtc	cgtctgtc	tgtcggtc	tgtccgtc	tgtctgtc
ggtcaatc	ggtcactc	ggtcattc	cgtcaatc	cgtcactc	cgtcattc
tgtcaatc	tgtcactc	tgtcattc	ggtcagac	ggtcaggc	ggtcagcc
cgtcagac	cgtcaggc	cgtcagcc	tgtcagac	tgtcaggc	tgtcagcc
ggtcagta	ggtcagtg	ggtcagtt	cgtcagta	cgtcagtg	cgtcagtt
tgtcagta	tgtcagtg	tgtcagtt	aaacagtc	aagcagtc	aaccagtc
acacagtc	acgcagtc	acccagtc	atacagtc	atgcagtc	atccagtc
aataagtc	aatgagtc	aattagtc	actaagtc	actgagtc	acttagtc
attaagtc	attgagtc	atttagtc	aatcggtc	aatccgtc	aatctgtc
actcggtc	actccgtc	actctgtc	attcggtc	attccgtc	attctgtc
aatcggtc	aatccgtc	aatctgtc	actcggtc	actccgtc	actctgtc
attcggtc	attccgtc	attctgtc	aatcagac	aatcaggc	aatcagcc
actcagac	actcaggc	actcagcc	attcagac	attcaggc	attcagcc
aatcagta	aatcagtg	aatcagtt	actcagta	actcagtg	actcagtt
attcagta	attcagtg	attcagtt	agaaagtc	agagagtc	agatagtc
aggaagtc	agggagtc	aggtagtc	agcaagtc	agcgagtc	agctagtc
agacggtc	agaccgtc	agactgtc	aggcggtc	aggccgtc	aggctgtc
agccggtc	agcccgtc	agcctgtc	agacaatc	agacactc	agacattc
aggcaatc	aggcactc	aggcattc	agccaatc	agccactc	agccattc
agacagac	agacaggc	agacagcc	aggcagac	aggcaggc	aggcagcc
agccagac	agccaggc	agccagcc	agacagta	agacagtg	agacagtt
aggcagta	aggcagtg	aggcagtt	agccagta	agccagtg	agccagtt
agtaggtc	agtacgtc	agtatgtc	agtgggtc	agtgcgtc	agtgtgtc
agttggtc	agttcgtc	agtttgtc	agtaaatc		

algorithm, we have generated tuples of l-mers originating from different strings and the common d-neighborhood of such tuples. Initially we build a matrix R of size n (m-l+1) where row i contains all the l-mers in Si. We pick an l-mer x from row 1 of R and push it on a stack. We filter out any l-mer in R at a distance greater than 2d from x. Then we pick an l-mer from the second row of R and push it on the stack. We filter out any l-mer in R that does not have a common neighbor with the l-mers on the stack; then we repeat the process. If any row becomes empty, we discard the top of the stack, revert to the previous instance of R and try a different l-mer. If the stack size is above a certain threshold, we generate the common d-neighborhood of the l-mers on the stack. For each neighbor M we check whether there is at least one l-mer u in each row of R such that Hd(M, u) d. If this is true, M is a motif.

Then we have tried to implement a new technique of motif finding. In this technique, we

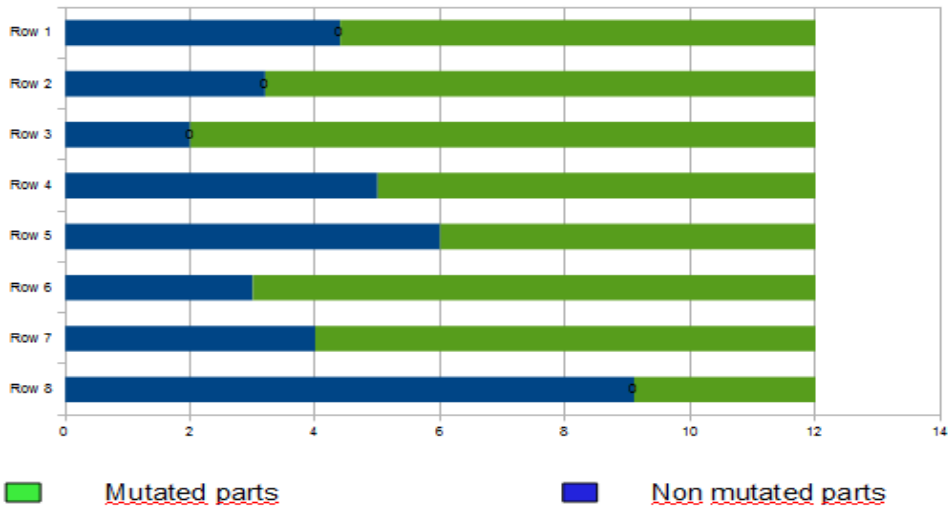

Figure 3.4: Initial matrix of l-mers

use a heuristic approach which is different from the heuristic approach in qPMS9. We first arrange the l-mers in descending orders according to their profile matrix value and pick first l-mer x from row 1 of R and push it on a stack. After filtering the l-mers(which are at a distance greater than 2d from x), we arrange them according to the increasing distance from x. Then we arrange l-mers of the second row according to their distance from x and push it on the stack. If any row becomes empty, we remove the top of the stack, revert to the previous instance of R and try the next l-mer. Thus, we generate the common d-neighborhood of the l-mers on the stack. And similarly, meeting all the previous conditions we will get a motif, M.

In qPMS9 we change the criteria by which the strings are reordered, as follows. Let T be the current tuple of l-mers and let u be an alive l-mer with respect to T. If we add u to T, then the consensus total distance of T increases. We compute this additional distance Cd(T∪u) - Cd(T). For each of the remaining strings, we compute the minimum additional distance for any alive l-mer in that string. Then we sort the strings decreasingly by the minimum additional distance. Therefore, we give priority to the string with the largest minimum additional distance. If two strings have the same minimum additional distance, we give priority to the string with fewer alive l-mers. The intuition is that larger additional distance could indicate more ”diversity” among the l-mers in the tuple, which means smaller common d-neighborhoods. We invoke the algorithm as GenTuples({}, k, R) where the matrix R contains all the l-mers in all the input strings, grouped as one row per string.

In qPMS9 the main objective of heuristic is to maximize d neighbourhood size.The perspective behind the heuristic is the more diversity the more motif possibility. On the other hand our heuristic goal is to find motif in the shortest time. So, we arrange the row heuristically

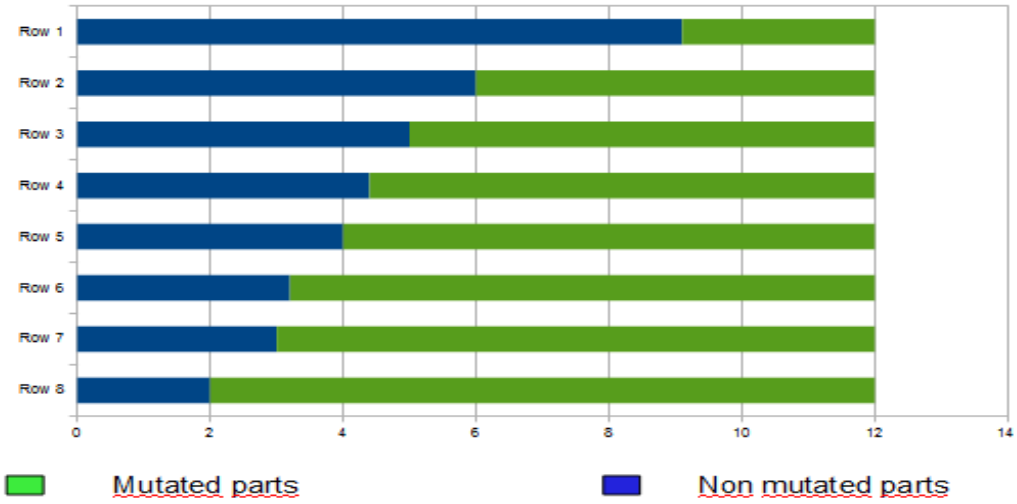

Figure 3.5: Matrix of l-mers after sorting according to mutations

to get candidate motifs in the first positions of rows.

CHAPTER 4
CONCLUSION

Welfare of all living beings is one of the most important issues of the world. Our characteristics are controlled by motifs in DNA. So, the study of Planted Motif Search is very important topic in the field of Bioinformatics. It may help in various aspects of modern science. It can also help in the medical science to detect various diseases and get prevention of the diseases. Suppose, we have discovered such an instance of motif which can turn on some of latent genetic properties inside a living being. As a result, we may be able to get a cure for disabled human or other living organisms. We may also be able to unlock some dormant hereditic characterisitics.

Finding motif in appropriate technique with less time is a very important issue. The complexity increases with increase in both length and mutations. If we are able to decrease the runtime of motif finding algorithms, we will be able to take decisions quickly. It will also help in medical operation for detecting mutation and unlocking some of the key features and also remove some hereditary diseases early. So in short, our main target is to optimize the time complexity for finding motif. We have read about many techniques for finding motif. At first, we have applied brute force method, branch and bound method and median string search method. After applying these techniques, we read the papers and algorithm of quorum Planted Motif Search (qPMS) and applied all the algorithms.

In branch and bound method of motif search, we have taken all the input l-mers as candidate motifs. But in qPMSPrune, we have started considering those l-mers as candidate motif which satisfies the pruning condition. In qPMS7, we have extended the research and used neighborhood generation as the key factor in improving the runtime of motif finding algorithm. Algorithm qPMS7 is a search-based algorithm, it uses a small amount of memory. This feature of Algorithm qPMS7 is a major advantage compared to other algorithms of the previous. PMS5, and PMS6 which require a large amount of memory when solving instances with large values of l and d.

Another advantage of Algorithm qPMS7 over these algorithms is that they cannot deal with the qPMS problem and in particular they only handle the PMS problem. Algorithm qPMS7 traverses the graph Gd (x,y) in a depth-first manner. However PMS8, an efficient algo-

rithm for the PMS problem. an efficient algorithm for the PMS problem. PMS8 is able to efficiently generate neighborhoods for t l-mers at a time, by using the pruning conditions presented. Previous algorithms generate neighborhoods for only up to three l-mers at a time whereas in PMS8 the value of t is increased as the instances become more challenging and therefore the exponential explosion is postponed. The second reason for the efficiency of PMS8 comes from the careful implementation which employs several speedup techniques and emphasizes cache locality. we have used a matrix for neighborhood generation for the purpose of finding candidate motifs easily. We have basically analysed what would be the possible number of mutated motifs for different values of l and d, worked on PMS8 and tried to improve the neighborhood traversing technique for motif search. We have tried to apply a heuristic value for traversing the matrix of l-mers using the shortest possible time. So we found that if we arrange the matrix in ascending order according to their profile matrix value, we would be able to speedup the motif search.

In qPMS9, the matrix has been rearrnaged in descending order of profile matrix value. So further we proceed, greater is the probability of finding the motif.From all those techniques, qPMS9 is the latest and fastest technique which uses the neighborhood generation technique of qPMS8. If we optimize the time for neighborhood generation, the whole time for finding motifs will be optimized. So ,we applied heuristic value to neighborhood generation technique to check if there is any improvement or not.

Though it is extremely hard to find motif with mutation. Increasing number of mutations means increasing the number of mismatches in patterns. But we have tried to simulate many techniques for complete understanding. We successfully count numbers of motif when there is one mutation and when there are two mutations. We have given the idea of heuristic over qPMS9 but it can not be proceed in normal computer that we use personally. In future we would like to work on these heuristics value and if we get enough resource and support we would like to develop our idea.

REFERENCES

[1] S. Rajasekaran, "1. abstract 2. introduction 3. experimental techniques 4. computational techniques 4.1. statistics based techniques 4.2. discrete algorithmic techniques," *Frontiers in Bioscience*, vol. 14, pp. 5052–5065, 2009.

[2] S. Rajasekaran and H. Dinh, "A speedup technique for (l, d)-motif finding algorithms," *BMC research notes*, vol. 4, no. 1, p. 1, 2011.

[3] H. Dinh, S. Rajasekaran, and J. Davila, "Qpms7: a fast algorithm for finding (, d)-motifs in dna and protein sequences," *PloS one*, vol. 7, no. 7, p. e41425, 2012.

[4] M. Nicolae and S. Rajasekaran, "Efficient sequential and parallel algorithms for planted motif search," *BMC bioinformatics*, vol. 15, no. 1, p. 1, 2014.

[5] J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang, "Distinguishing string selection problems," *Information and Computation*, vol. 185, no. 1, pp. 41–55, 2003.

[6] J. Gramm, R. Niedermeier, and P. Rossmanith, "Exact solutions for closest string and related problems," in *International Symposium on Algorithms and Computation*, pp. 441–453, Springer, 2001.

[7] L. Duret and P. Bucher, "Searching for regulatory elements in human noncoding sequences," *Current opinion in structural biology*, vol. 7, no. 3, pp. 399–406, 1997.

[8] N. Pisanti, A. M. Carvalho, L. Marsan, and M.-F. Sagot, "Risotto: Fast extraction of motifs with mismatches," in *Latin American Symposium on Theoretical Informatics*, pp. 757–768, Springer, 2006.

[9] E. Rocke and M. Tompa, "An algorithm for finding novel gapped motifs in dna sequences," in *Proceedings of the second annual international conference on Computational molecular biology*, pp. 228–233, ACM, 1998.

[10] A. Price, S. Ramabhadran, and P. A. Pevzner, "Finding subtle motifs by branching from sample strings," *Bioinformatics*, vol. 19, no. suppl 2, pp. ii149–ii155, 2003.

[11] N. Dasari, R. Desh, and M. Zubair, "Solving planted motif problem on gpu," in *International Workshop on GPUs and Scientific Applications, GPUScA 2010, Vienna, Austria, September 11*, 2010.

[12] S. Bandyopadhyay, S. Sahni, and S. Rajasekaran, "Pms6: A fast algorithm for motif discovery," *International Journal of Bioinformatics Research and Applications 2*, vol. 10, no. 4-5, pp. 369–383, 2014.

[13] H. Dinh, S. Rajasekaran, and V. K. Kundeti, "Pms5: an efficient exact algorithm for the (, d)-motif finding problem," *BMC bioinformatics*, vol. 12, no. 1, p. 410, 2011.

[14] Q. Yu, H. Huo, Y. Zhang, and H. Guo, "Pairmotif: a new pattern-driven algorithm for planted (l, d) dna motif search," *PLoS One*, vol. 7, no. 10, p. e48442, 2012.

[15] M. Nicolae and S. Rajasekaran, "qpms9: An efficient algorithm for quorum planted motif search," *Scientific reports*, vol. 5, 2015.

[16] K. Morland, S. Wing, A. D. Roux, and C. Poole, "Neighborhood characteristics associated with the location of food stores and food service places," *American journal of preventive medicine*, vol. 22, no. 1, pp. 23–29, 2002.

[17] N. S. Dasari, D. Ranjan, and M. Zubair, "High performance implementation of planted motif problem using suffix trees," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pp. 200–206, IEEE, 2011.

[18] B. Sahoo, R. Sourav, R. Ranjan, and S. Padhy, "Parallel implementation of exact algorithm for planted motif search problem using smp cluster," *European J Scientific Res*, vol. 64, no. 4, pp. 484–496, 2011.

APPENDIX A
CODES

A.1 Sample Code

We use this part of code to find out motif with mutations.

```
#include <iostream>
#include "stdio.h"
#include <fstream>
#include "time.h"

using namespace std;
#define INFTY 2147483647

int BestScore(char **sDNA, int l, int t, char *bestCha)   //calculate t
{
        int sA, sC, sG, sT, Tbest=0;
        int best = 0;



        for(int i=0; i<l; i++)

        {


                sA=0; sC=0; sG=0; sT=0;
                for(int j=0;j<t;j++)
                {

                        if(sDNA[j][i]=='a')
                        {
                                sA++;
```

```
28                          }else if(sDNA[j][i]=='c')
29                          {
30
31                                  sC++;
32                          }else if(sDNA[j][i]=='g')
33                          {
34                              sG++;
35                          }else if(sDNA[j][i]=='t')
36                          {
37
38                              sT++;
39                          }
40
41
42                  }
43
44                  if(sA>best)
45                  {
46                          best=sA;
47                          bestCha[i]='a';
48                  }
49                  if(sC>best)
50                  {
51                          best=sC;
52                          bestCha[i]='c';
53                  }
54                  if(sG>best)
55                  {
56                          best=sG;
57                          bestCha[i]='g';
58                  }
59                  if(sT>best)
60                  {
61                          best=sT;
62                          bestCha[i]='t';
63                  }
64
65                  Tbest=Tbest+best;
66                  best=0;
```

```
67          }
68
69
70          return Tbest;
71  }
72
73
74
75  char* BruteForceMotifSearchAgain(char **DNA, int t, int n, int l)
76  {
77          int *a = new int[n-l+1];
78          char **sDNA=new char *[t];
79          char *bestCha=new char[l];
80          char *mutif = new char[l];
81          int k=0,Tbest=0,bScore=0,endflag=0;
82          for (int i=0; i<n-l+1 ; i++)
83          {
84                  a[i]=0;
85          }
86
87          for (int i=0; i<t; i++)
88          {
89                  sDNA[i]=new char [l];
90          }
91
92
93          while (1)
94          {
95
96                  for(int i=0;i<t;i++)
97                  {
98                          k=a[i];
99                          for(int j=0; j<l; j++ )
100                         {
101                                 sDNA[i][j]= DNA[i][k+j];
102                         }
103                 }
104                 for(int i=t-1;i>=0;i--)      //calculate the next leaf
105                 {
```

```
106                         if(a[i]==(n-l))
107                         {
108                                 a[i]=0;
109                                 if (i==0) {
110                                         endflag=1;
111                                 }
112
113                         }else
114                         {
115                                 a[i]++;
116                                 break;
117                         }
118
119                 }
120
121
122                 Tbest= BestScore(sDNA,l, t, bestCha);
123
124                 if(Tbest>bScore)    //find the leaf with a best score a
125                 {
126                         bScore=Tbest;
127                         for(int i=0; i<l; i++ )
128                         {
129                                 mutif[i]=bestCha[i];
130                         }
131
132                 }
133                 if(endflag==1)   //if go over all the leaves, break loo
134                         break;
135
136         }
137
138         return mutif;    //return the best mutif
139
140 }
141
142
143
144 int Score(char **sDNA, int l, int ti, char *bestCha)   //calculate the
```

```
{
        int sA, sC, sG, sT, Tbest=0;
        int best = 0;



        for(int i=0; i<l; i++)

        {
                sA=0; sC=0; sG=0; sT=0;
                for(int j=0;j<=ti;j++)
                {

                        if(sDNA[j][i]=='a')
                        {
                                sA++;
                        }else if(sDNA[j][i]=='c')
                        {

                                sC++;
                        }else if(sDNA[j][i]=='g')
                        {
                            sG++;
                        }else if(sDNA[j][i]=='t')
                        {

                            sT++;
                        }


                }

                if(sA>best)
                {
                        best=sA;
                        bestCha[i]='a';
                }
                if(sC>best)
```

```
184                 {
185                         best=sC;
186                         bestCha[i]='c';
187                 }
188                 if(sG>best)
189                 {
190                         best=sG;
191                         bestCha[i]='g';
192                 }
193                 if(sT>best)
194                 {
195                         best=sT;
196                         bestCha[i]='t';
197                 }
198
199                 Tbest=Tbest+best;
200                 best=0;
201         }
202
203         //cout <<bestCha[0]<<bestCha[1]<<bestCha[2]<<bestCha[3]<<endl;
204         //cout << Tbest<<endl;
205         return Tbest;
206 }
207
208
209
210 char* BranchAndBoundMotifSearch(char **DNA, int t, int n, int l)
211 {
212         int *a = new int[n-l+1];
213         char **sDNA=new char *[t];
214         char *bestCha=new char[l];
215         char *mutif = new char[l];
216         int k=0,iscore=0,tscore=0,bScore=0,endflag=0,optimisticScore;
217         for (int i=0; i<n-l+1 ; i++)
218         {
219                 a[i]=0;
220         }
221
222         for (int i=0; i<t ; i++)
```

```
223         {
224                 sDNA[i]=new char [l];
225         }
226
227
228         while (1)
229         {
230                 for(int i=0;i<t;i++)
231                 {
232                         k=a[i];
233                         for(int j=0; j<l; j++ )
234                         {
235                                 sDNA[i][j]= DNA[i][k+j];
236
237                         }
238
239         iscore= Score(sDNA,l, i, bestCha);
240         optimisticScore= iscore+(t-i-1)*l;
241         if(i!=(t-1))
242         {
243             if (optimisticScore<bScore)
244             {       //bypasss
245                 for(int j=i; j>=0; j--)
246                 {
247                     if(a[j]==(n-l))
248                     {
249                         a[j]=0;
250                         if (j==0)
251                         {
252                             endflag=1;
253                         }
254                     }else
255                     {
256                         a[j]++;
257                         break;
258                     }
259
260
261                 }
```

```
262                 for(int j=i+1; j<t; j++)
263                 {
264
265                     a[j]=0;
266                 }
267             }
268         }
269     }
270
271     for(int i=(t-1);i>=0;i--)      //calculate the next leaf
272     {
273         if(a[i]==(n-l))
274         {
275             a[i]=0;
276             if (i==0)
277             {
278                 endflag=1;
279             }
280
281         }else
282         {
283                                 a[i]++;
284                                 break;
285         }
286
287     }
288
289     tscore= Score(sDNA,l, t-1, bestCha);
290
291     if(iscore>bScore)
292     {
293         bScore=tscore;
294         for(int i=0; i<l; i++ )
295         {
296                                 mutif[i]=bestCha[i];
297         }
298
299     }
300
```

```
301
302     if(endflag==1)   //if go over all the leaves, break loop
303                         break;
304
305 }
306
307         return mutif;    //return the best mutif
308
309 }
310
311
312 int TotalDistance(char *s, char **DNA, int t, int n, int l)
313 {
314     char *v = new char[l];
315         int Distance, MinDistance=INFTY, SumDistance=0;
316
317         for (int i=0; i< t; i++)
318         {
319                 MinDistance=INFTY;
320                 for (int j=0; j<(n-l+1); j++)
321                 {
322                         Distance=0;
323                         for(int k=0; k<l; k++)
324                         {
325                                 v[k]=DNA[i][j+k];
326
327                                 if(v[k]!=s[k])
328                                 {
329                                         Distance++;
330
331                                 }
332                         }
333
334                         if(Distance<MinDistance)
335                         {
336                                 MinDistance=Distance;
337                         }
338
339
```

```
340                 }
341
342
343                 SumDistance=SumDistance+MinDistance;
344         }
345
346         return SumDistance;
347 }
348
349
350 char* BruteForceMedianSearch(char **DNA, int t, int n, int l)
351 {
352         int  *a = new int[l];
353         char *s = new char[l];
354         char *mutif = new char[l];
355         int endflag=0,totaldistance=0,BestDistance=INFTY;
356
357         for (int i=0; i<l; i++) {
358                 a[i]=0;
359
360         }
361
362         while (1)
363         {
364
365                 //cout << "while";
366                 for (int i=0; i<l; i++)
367                 {
368                         if (a[i]==0)
369                         {
370                                 s[i]='a';
371                         }else if(a[i]==1){
372                                 s[i]='c';
373                         }else if (a[i]==2) {
374                                 s[i]='g';
375                         }else if(a[i]==3) {
376                                 s[i]='t';
377                         }
378
```

```
379
380                 }
381
382                 totaldistance=TotalDistance(s,DNA,t,n,l);
383
384                 if(totaldistance<BestDistance)
385                 {
386                         BestDistance=totaldistance;
387                         //cout<<BestDistance<<"\n";
388                         for(int i=0; i<l; i++)
389                         {
390                                 mutif[i]=s[i];
391                                 //cout << mutif[i];
392                         }
393                         //cout<<"\n";
394                         cout << BestDistance<<endl;
395                         //cout << endl;
396                 }
397
398     for(int i=(l-1);i>=0;i--)      //calculate the next leaf
399     {
400         if(a[i]==3)
401         {
402             a[i]=0;
403             if (i==0) {
404                 endflag=1;
405             }
406
407         }else
408         {
409                                 a[i]++;
410                                 break;
411         }
412
413                 }
414
415
416     if(endflag==1)   //if go over all the leaves, break loop
417                         break;
```

```
418
419 }
420
421         return mutif;    //return the best mutif
422
423 }
424
425
426 int TotalDistance2(char *s, char **DNA, int t, int n, int si)
427 {
428     char *v = new char[si+1];
429         int Distance, MinDistance=INFTY, SumDistance=0;
430
431         for (int i=0; i< t; i++)
432         {
433                 MinDistance=INFTY;
434                 for (int j=0; j<(n-si); j++)
435                 {
436                         Distance=0;
437                         for(int k=0; k<(si+1); k++)
438                         {
439                                 v[k]=DNA[i][j+k];
440
441                                 if(v[k]!=s[k])
442                                 {
443                                         Distance++;
444
445                                 }
446                         }
447
448                         if(Distance<MinDistance)
449                         {
450                                 MinDistance=Distance;
451                         }
452
453
454                 }
455
456
```

```
457                 SumDistance=SumDistance+MinDistance;
458         }
459
460         return SumDistance;
461 }
462
463
464 char* BranchAndBoundMedianSearch(char **DNA, int t, int n, int l)
465 {
466         int  *a = new int[l];
467         char *s = new char[l];
468         char *mutif = new char[l];
469         int endflag=0,optimisticDistance=0,BestDistance=INFTY,si=0;
470
471         for (int i=0; i<l; i++) {
472                 a[i]=0;
473         }
474
475         while (1)
476         {
477
478
479
480                 for (int i=0; i<l; i++)
481                 {
482                         if (a[i]==0)
483                         {
484                                 s[i]='a';
485                         }else if(a[i]==1){
486                                 s[i]='c';
487                         }else if (a[i]==2) {
488                                 s[i]='g';
489                         }else if(a[i]==3) {
490                                 s[i]='t';
491                         }
492                 }
493
494
495     optimisticDistance=TotalDistance2(s,DNA,t,n,si);
```

```
496
497     if(si!=(l-1))
498     {
499         if(optimisticDistance>BestDistance)
500         {
501             for(int i=si;i>=0;i--)      //calculate the next leaf
502             {
503                 if(a[i]==3)
504                 {
505                     a[i]=0;
506                     if (i==0) {
507                     endflag=1;
508                     }
509
510                 }else
511                 {
512                     a[i]++;
513                     break;
514                 }
515
516             }
517     for(int j=si+1; j<l; j++)
518             {
519                 a[j]=0;
520             }
521
522             }
523         else {
524             si++;
525             a[si]=0;
526             }
527
528
529         }else
530         {
531     if(optimisticDistance<BestDistance)
532         {
533
534         BestDistance=optimisticDistance;
```

```
535         for(int i=0; i<l; i++)
536             {
537                 mutif[i]=s[i];
538 
539             }
540 
541             }
542 
543         for(int i=(l-1);i>=0;i--)      //calculate the next leaf
544         {
545             if(a[i]==3)
546             {
547             si--;
548 
549             if (i==0) {
550             endflag=1;
551                 }
552 
553             }else
554             {
555                 a[i]++;
556                 break;
557             }
558 
559         }
560 
561     }
562  if(endflag==1)   //if go over all the leaves, break loop
563         break;
564 
565         }
566 
567         return mutif;    //return the best mutif
568 
569 }
570 
571 
572 
573 
```

```
574
575
576
577 int main () {
578     //*************************************
579         //change the t, n, l ,ntimes here
580         //*************************************
581         int t=4, n=57, l=8, ntimes=1;
582     char *mutif=new char [l];
583         char** DNA = new char*[t];
584
585         clock_t start, finish;
586         double duration;
587
588         for(int i=0; i<t; i++)
589         {
590             DNA[i]=new char [n];
591         }
592
593         /*cout<<"please␣input␣the␣value␣of␣t:";
594         cin>>t;
595         cout<<endl;
596         cout << "please␣input␣the␣value␣of␣l:␣";
597         cin>>l;
598         cout<<endl;
599         */
600         cout << "t="<<t<<",lmer␣length="<<l<<endl;
601
602         ifstream fin("data.txt");
603
604         for(int i=0;i<t;i++)
605         {
606         for(int j=0;j<n;j++)
607                 {
608                         fin>>DNA[i][j];
609                         //cout << DNA[i][j];
610                 }
611                 //cout<<endl;
612         }
```

```
613     fin.close();
614
615
616
617             //cout << "please␣input␣the␣times␣you␣want␣to␣run:␣";
618                 //cin>>ntimes;
619                 //cout << endl;
620
621
622                     start   =   clock();
623                 for(int i=0;i<ntimes;i++)
624                         mutif = BruteForceMotifSearchAgain(DNA, t, n, l
625                         finish   =   clock();
626
627                  duration = (double)(finish-start)/CLOCKS_PER_SEC;
628                  duration=duration/ntimes;
629                  cout<< "Average␣time␣for␣BruteForceMotifSearchAgain␣is␣
630
631                  cout << "the␣motif␣is␣";
632                  cout<<"'";
633                  for(int i=0; i<l; i++)
634                      cout << mutif[i];
635                      cout<<"'";
636                      cout << endl;
637                      cout << endl;
638
639
640                     start   =   clock();
641                  for(int i=0;i<ntimes;i++)
642                      mutif = BranchAndBoundMotifSearch(DNA, t, n, l);
643                finish   =   clock();
644
645                duration = (double)(finish-start)/CLOCKS_PER_SEC;
646                duration=duration/ntimes;
647                 cout << "Average␣time␣for␣BranchAndBoundMotifSearch␣is␣"
648             cout << endl;
649                     cout << "the␣motif␣is␣";
650                     cout<<"'";
651                     for(int i=0; i<l; i++)
```

```
652                         cout << mutif[i];
653                 cout<<"'";
654                 cout << endl;
655                 cout << endl;
656
657
658
659
660                 start   =   clock();
661              for(int i=0;i<ntimes;i++)
662                  mutif = BruteForceMedianSearch(DNA, t, n, l);
663             finish   =   clock();
664
665             duration = (double)(finish-start)/CLOCKS_PER_SEC;
666              duration=duration/ntimes;
667               cout << "Average␣time␣for␣BruteForceMedianSearch␣is␣"<<d
668           cout << endl;
669               cout << "the␣motif␣is␣";
670
671                 cout<<"'";
672                 for(int i=0; i<l; i++)
673                         cout << mutif[i];
674                 cout << "'";
675                 cout << endl;
676                 cout << endl;
677
678
679                 start  =   clock();
680              for(int i=0;i<ntimes;i++)
681                  mutif = BranchAndBoundMedianSearch(DNA, t, n, l);
682              finish   =   clock();
683            duration = (double)(finish-start)/CLOCKS_PER_SEC;
684              duration=duration/ntimes;
685               cout << "Average␣time␣for␣BranchAndBoundMedianSearch␣is␣
686               cout << endl;
687
688                 cout << "the␣motif␣is␣";
689           cout<<"'";
690                 for(int i=0; i<l; i++)
```

```
691                             cout << mutif[i];
692                     cout<<"'";
693                     cout << endl;
694                     cout << endl;
695
696
697
698         return 0;
699
700 }
```