Bachelor of Science in Computer Science and Engineering

Thesis

An Improved Data Structure For Efficient Storage of Multiple BIOsequences

Prepared By:
Md. Zahidul Hasan (084434)
Anik Islam Shimul (084441)

Co-Supervisor
Tareque Mohmud Chowdhury
Assistant Professor, CSE Department

Supervisor
Prof. Dr. M. A. Mottalib
Head, CSE Department

Computer Science and Engineering (CSE) Department

Islamic University Of Technology (IUT)

Organisation of Islamic Cooperation (OIC)

Gazipur, Bangladesh

October, 2012

Certified of Research

This is to certify that the work presented in this thesis is the result of the implementation of **"An Improved Data Structure for Efficient Storage of Multiple BIOsequences"** which has been supervised by Prof. Dr. M. A. Mottalib, Head, CSE Department and co-supervised by Mr. Tareque Mohmud Chowdhury, Assistant Professor, CSE Department of Islamic University of Technology (IUT) .

This thesis was undertaken as a partial fulfillment of requirements for the Bachelor of Science degree in Computer Science and Engineering. It is also declared that this thesis thereof has not been submitted anywhere else for the award of any degree or any publication.

Authors:

Md. Zahidul Hasan ----------

Student ID: 084434 Signature

Anik Islam Shimul ----------

Student ID: 084441 Signature

Co-supervisor:

Tareque Mohmud Chowdhury ----------

Assistant Professor, CSE department Signature

Supervisor:

Prof. Dr. M. A. Mottalib ----------

Head, CSE Department Signature

Department of Computer Science and Engineering, IUT

October, 2012

Acknowledgment

At first we would like to thank our respected supervisor Prof. Dr. M. A. Mottalib, Head, CSE Department for his earnest help, guidance and motivation towards the completion of this thesis work.

We convey our sincere gratitude to our co-supervisor Tareque Mohmud Chowdhury, Assistant Professor, CSE department for his ideas, wise patronization and co-operation.

We would like to thank our parents for their consistent inspiration and encouragement for us to consummate our goal.

We are grateful to all the teachers and members of the department of CSE for their consistent support and enthusiasm.

Last but not the least we thank to our batch mates, our beloved friends, for their supports, assists and for being there no matter what.

Abstract

Compression of large DNA sequences has been a subject of great interest since the availability of genomic databases. Although only two bits are sufficient to encode four bases of DNA (namely A, G, T and C), the massive size DNA sequences forces the need for efficient compression. In this article we are going to propose an improved version of an existing algorithm known as “GtEncseq” which describes the procedure of storing multiple biological sequences of variable Character size, with customizable character transformations, “wildcard” and “separator” support, and a diverse group of internal representations optimized for different arrangements of wildcards and sequence lengths. Our main target is extensive compression of data with an attempt of eliminating the wildcard entries from the sequence but make it available for the reuse. An efficient time requirement for encoding the desired sequence is also a note to consider.

List of figures

Figure name **Page**

- Fig 2.1: Mapping Scheme for Decoding of LUT and LZ77 algorithm **15**
- Fig 2.2: The queries to be solved for T and D **19**
- Fig 2.3: The GtEncseq algorithm representation **21**
- Fig 2.4: comparison for various DNA and protein datasets using different compression model **24**
- Fig 3.1: Wildcard representation **26**
- Fig 4.1: Encoding steps of the improved algorithm **31**
- Fig 4.2 : Decoding steps of the improved algorithm **34**
- Fig 4.3: Interface before taking input **35**
- Fig 4.4: Interface after taking a sample input and encoding **35**
- Fig 5.1: graphical representation regarding space requirement for encoded sequence between the existing and proposed algorithm **38**
- Fig 5.2: graphical representation regarding time requirement for encoded sequence between the existing and proposed algorithm **39**

List of tables

Table Name **Page**

- Table 5.1: space requirement for our proposed method and GtEncseq **36**

- Table 5.2: encoding time requirement for our proposed method and GtEncseq **37**

Table of Contents

	Acknowledgement	**ii**
	Abstract	**iii**
	List of figures	**iv**
	List of tables	**v**
	Table of contents	**vi**
Chapter 1	**Introduction**	**8**
	1.1 Overview	9
	1.2 Motivation	9
	1.3 Problem Statement	11
	1.4 Research Challenges	11
	1.5 Scope	12
	1.6 Thesis Outline	12
Chapter 2	**Literature Review**	**13**
	2.1 Background studies	13
	2.2 Related works	14
	2.3 GtEncseq Data Structure	17
	2.4 Implementation comparison of GtEncseq with various other methods	17
	2.5 Implementation of GtEncseq Data Structure	18
	2.5.1 Sequence Representation	18
	2.5.2 Application Programming Interface	22
	2.6 Performance comparison with existing models	24
Chapter 3	**Proposed Method**	**25**
	3.1 Modified representation of wildcard	26
	3.2 Mathematical analysis of our proposed data structure	27
Chapter 4	**Implementation**	**29**
	4.1 Used hardware and software	29

4.2 Input Specifications 29
4.3 Procedural analysis 29
4.4 Interface 35

Chapter 5 **Result Analysis and Discussion** **36**

5.1 Research data 36
5.2 Analysis and comparison 37

Chapter 6 **Conclusion and Future Scope** **40**

References **41**
Appendix **42**

Chapter 1
Introduction

The introduction of Biological sequencing has significantly accelerated biological research and discovery. The rapid growth of modern Biological sequencing technology has been revolutionary. Many Biological projects, mostly by scientific collaboration across various parts of the world, have generated the complete Biological sequencing of many animal, plant, and microbiological organs. Storing multiple biological sequences of different categories and retrieving these sequences separately as well as accessing any of these sequences randomly are the fundamental necessities in bioinformatics and therefore several algorithms have been developed to make these actions flexible and accurate.

Representations of biological sequences of course need to support nucleotide and amino acid alphabets. Like notation for masked positions, uppercase and lowercase notation and with different sets of wildcard symbols (for sequence positions containing ambiguous nucleotides or amino acids), separator positions while dealing with multiple sequences. A flexible representation handles these variations in a sensible way. GtEncseq is one of the most recent and developed implementation that provides a variety of methods for random and sequential access to individual characters, individual sequences, wildcard sequences and also substrings. GtEncseq fully supports access to sequence descriptions, lengths, and original filenames in constant time per sequence. Regardless the fact that the performance evaluation of this data structure has superseded most of the other methods, potential flaws is still remaining. For gigabase genome sequences consisting billions of base pair sequences and millions of wildcard entries (human genome, 3.1 gigabases and approximate 237 million wildcard entries [1]) can cause redundancy and wastage of memory. We are here to identify the limitations and drawbacks regarding the encoding of multiple BIOsequences and propose an improved structure which can overcome the existing setback of the algorithm and to present a flexible structure which can increase the performance regarding efficiency of storing genome data.

1.1 Overview

Some of the research areas in genetics include Gene regulation, DNA sequence organization, Chromosomal structure and organization, Non-coding DNA types and functions, coordination of gene expression, protein synthesis, and post-translational events interaction of proteins in complex molecular machines, evolutionary conservation among organisms, Protein conservation, correlation of single-base DNA variations with health and disease, etc[7]. But to store these large volumes data in an efficient manner and with a lossless approach is an important aspect of Biological science.

Genes basically are basically composed of one of the following four types of bases-adenine, cytosine, guanine, and thymine often abbreviated as A, C, G, and T. Many compression methods have been discovered to compress DNA sequences. Invariably, all the methods found so far take advantage of the fact that DNA sequences are made of only 4 alphabets [8]. But there are 21 other characters which are also used to represent protein or amino acid sequences. Among them N, S, Y, W, R, K ,V, B, D, H, M and their lowercase representations are known as wildcard entries [1]. These are ambiguous compounds and most of their functionality and physical significance is yet to be discovered. However, these characters can co-exist with large nucleotide sequences making the formations of large chunks. So, at first it is necessary to find the order in which the nucleotides or amino acids are arranged in the DNA.

Deriving meaningful knowledge from DNA sequence will define biological research through the coming decades and require the combined effort of biologists, chemists, engineers, and computational scientists, among others.

1.2 Motivation

Genomic science is confronted with the volume of sequencing and resequencing data which is increasing at a higher pace than that of data storage and communication resources,

causing a shift of a significant part of research budgets from the sequencing component of a project to the computational one. Hence, being able to efficiently store sequencing and resequencing data is a problem of paramount importance. The human genome alone is made up of over 3 billion nucleotides or bases.

The continuing exponential accumulation of full genome data, including full diploid human genomes, creates new challenges not only for understanding genomic structure, function and evolution, but also for the storage, navigation and privacy of genomic data. So it is necessary develop data structures and algorithms for the efficient storage of genomic and other sequence data that may also facilitate querying and protecting the data [10].

Genes basically are composed of one of the following four types of bases-adenine, cytosine, guanine, and thymine often abbreviated as A, C, G, and T. Finding the order in which the nucleotides are arranged in the DNA is a part of genome sequencing. Knowing the sequence of the genome is the first step towards understanding it. A genome sequence does contain some clues about where genes are, even though scientists are just learning to interpret these clues.

In recent years, a substantial effort has been made for the application of textual data compression techniques to various computational biology tasks, ranging from storage and indexing of large datasets to comparison and reverse engineering of biological networks [9].

For a flexible and efficient sequence representation the sequence representation (including the pure sequence and related metadata) needs to be space efficient allowing fast retrieval of the content. The latter can be achieved using byte arrays, as in most common software tools.

Another requirement is support for storing multiple sequences in one representation. An intuitive representation of a collection of sequences is a linear concatenation of all individual sequences, with additional information to mark the sequence boundaries. For index-based techniques, the sequences are usually addressed by absolute sequence

positions, so one also needs to be able to map absolute sequence positions to relative positions [1].

1.3 Problem statement

The primary problems associated with compressed storing of high dimensional multiple biological sequences are handling wildcard characters and distinguishing among two different sequences. Meanwhile it is also difficult to maintain a minimal rate of time and space complexity while performing those operations. If the length of the wildcard sequence chunk is very large or if there is repetitive chunks along the sequence then it becomes difficult. So for these worst-case scenarios, the complexities rise.

1.4 Research challenges

To start with the challenges first it is found that the largeness of data size. While implementing the GtEncseq algorithm we found that for a FASTA-formatted input file of 37.54 GB size, which contains around 26,015,933 sequences [1]. Currently, publicly available genomes are typically stored as flat text files in GenBank, but this approach is unlikely to scale up in many ways. The storage of the diploid genomes of all currently living humans using this simple approach would take 'GenBank', without counting headers or any additional annotations, on the order of 36×1018 bytes, or 36M Terabytes, an amount difficult to store or download over the Internet, even using standard compression technologies (e.g. gzip) [10]. So for the in unavailability of a large number of data and due to a considerable amount of lower configurations for our available equipments compared to them, we needed to generate sample data to do the simulation. Then encoding such large sequence character by character without losing any data is also another sort of challenge. Again, for wildcard sequences with large number of wildcard characters, the decoding time complexity becomes high. Finally, accessing the data and retrieving the information with accuracy and flexibility is necessary to maintain.

1.5 Scope

The scopes for the study are manifold. “GtEncseq” is a new approach in the field of sequence storing and retrieving and the improvement of this method will certainly enable us to store very large number of datasets in an efficient way. It also gives us the easy access to the unused or rather unknown compounds and sequences of DNA and allows us to manage those data distinctly. Also storing multiple sequences in one representation is also very much efficient and retrieving each individual sequence is also possible.

1.6 Thesis outline

In Chapter 1 brief introduction of study and our work has been provided. Besides the motivational factors behind the thesis, scopes and challenges are also mentioned briefly. In chapter 2 the literature reviews are described where few background topics along with related algorithms for our thesis are mentioned. In the later part of the chapter the introduction and analysis for the algorithm that we have worked on and improved are provided. In chapter 3 our proposed method as well as discussion and mathematical analysis is described. The implementation part is briefly described in chapter 4. Chapter 5 introduces result and data analysis. Finally in chapter 6 we have drawn the conclusion with some mentions of future scopes for improvement for our work.

Chapter 2
Literature Review

2.1 Background studies

- Genes basically are composed of one of the following four types of bases-adenine, cytosine, guanine, and thymine often abbreviated as A, C, G, and T. Finding the order in which the nucleotides are arranged in the DNA is a part of genome sequencing.
- Multiple sequencing: refers to a linear concatenation of all individual sequences, with additional information to mark the sequence boundaries (separators).
- Wildcards: ambiguous nucleotides or amino acids.
- GenBank: The GenBank sequence database is an open access, annotated collection of all publicly available nucleotide sequences and their protein translations.
- FASTA: FASTA format is a text-based format for representing either nucleotide sequences or peptide sequences, in which nucleotides or amino acids are represented using single-letter codes. The format also allows for sequence names and comments to precede the sequences.
- BLAST : Basic Local Alignment Search Tool. Input to these tools is FASTA, GeneBank or ASN.1. While DNA sequences are stored in a byte compressed format, protein sequences are stored as simple byte arrays.
- BLAT: BLAT is an alignment tool like BLAST, but it is structured differently. On DNA, BLAT works by keeping an index of an entire genome in memory. Thus, the target database of BLAT is not a set of GenBank sequences, but instead an index derived from the assembly of the entire genome.
- gzip: The format is designed to allow single pass compression without any backwards seek, and without a priori knowledge of the uncompressed input size or the available size on the output media.

2.2 Related works

Many algorithms and data structures have been developed considering the fact in mind that the amount of research being increased in the field of gene sequencing, the volume of data is also increasing. Korodi and Tabusan [11] proposed an algorithm for the lossless compression of DNA files which contain annotation text besides nucleotide sequence. First, a grammar is specifically designed to capture the regularities of the annotation text. A revertible transformation uses the grammar rules in order to equivalently represent the original file as a collection of parsed segments and a sequence of decisions made by the grammar parser.

Another simple and less efficient method was published by Sheng Bao, Shi Chen, Zhi-Qiang Jing and Ran Ren [4] where they used a new DNA sequence compression algorithm which is based on LUT and LZ77 algorithm. Combined a LUT-based precoding routine and LZ77 compression routine, this algorithm can approach a compression ratio1 of 1.9bits /base and even lower. They created a Look Up Table (LUT) which describes a mapping relationship between DNA segment and its corresponding characters.

Every three characters in source DNA sequence (without N2) will be mapped into a character chosen from the character set which consists of 64 ASCII characters. The braces behind each character contain the corresponding ASCII codes of these characters. For easy implementation, characters a, t, g, c and A, T, G, C no longer appear in pre-coded file.

For instance, if a segment “ACTGTCGATGCC” has been read, in the destination file, it was represent as “j2X6”. Obviously, the destination file was case-sensitive.

The character N refers to the segment which is unknown. When being encounter a serial of successive Ns, the algorithm inserts two “/” into destination file to identify the starting and end place of these successive Ns. There is a number which equals to the number of Ns between the “/” pair. For instance, if segment”NNNNNN” has been read, in the destination file, it was representing them as “/6/”.

Another paper by Srinivasa K. G, Jagadish M, Venugopal K R and LMPatnaik [7], propose an encoding scheme to compress non repeat regions of DNA sequences, based on dynamic

programming approach. In order to test the efficiency of the method we incorporate the encoding scheme in a DNA-specifi c algorithm, DNAPack.

For example, the sequence AGTC would be taken as AATT and ATTA. The decoding procedure requires both the sequences, with four possible combinations of A and T representing each base as shown in a table in fig. 2

Sequence 1	Sequence 2	Base
A	A	A
A	T	G
T	T	T
T	A	C

Fig 1: Mapping Scheme for Decoding of LUT and LZ77 algorithm

The above substitutions are made to the entire DNA sequence. Let the pass 1 substitution sequence be *S1* and pass 2 sequence be *S2*. If the length of the sequences is l then each linear sequence is transformed to a matrix of dimension $\alpha \times \beta$ where $\alpha * \beta = l$. The transformation is done row-wise and hence the entry in ith row and jth column (zero-indexed) would correspond to alphabet in $i * \beta + j$ position in the sequence.

The choice of α and β can be made in different ways. A simple approach would be to break the sequence into multiple sequences each of length of a perfect square, chosen greedily, and represent each sub-sequence as a square matrix. This is always possible since any number can be represented as sum of squares. A more efficient way to determine the split is to consider all possible combinations of α and β and take the combination that leads to maximum compression ratio.

This is increases the complexity considerably but can be made non-prohibitive if the length of the longest sequence is restricted to a certain maximum value. The encoding and decoding of *S1* and *S2* are done independently. The compression technique works on the matrices obtained by the sequences. The encoding idea is based on the idea of recursively dividing the matrix into sub-matrices until each sub-matrix is composed of a single alphabet.

Another Algorithm is proposed by A. Cannane and H. Williams, known as RAY method, allows random access so that specific sections of the compressed data can be individually accessed.

RAY makes multiple passes over the input data. These passes incrementally construct a dictionary of symbol sequences that are repeated in the text, with each pass discovering longer sequences. Once the dictionary has been built, the data and dictionary can be coded using mechanisms such as Huffman coding, which in this application are reasonably efficient due to the fact that most of the occurrences of the most common symbols have typically been removed, reducing their frequency and thus reducing the difference between the 0-order entropy and whole-bit coding that can result from a Huffman process.

Another algorithm was proposed by Shanika Kuruppu, Bryan Beresford-Smith, Thomas Conway, and Justin Zobel [2] used RAY such that it is efficient for DNA compression. First, a symbol is defined to be a substring of, say, 16 consecutive symbols in the first iteration (first execution of steps 1-4 in RAY). The motivation here is that almost all substrings of length 16 or 16-mers occur with reasonable frequency in large DNA collections, and thus naive RAY would in four iterations discover the vast majority of these 16-mers as dictionary entries.

2.3 GtEncseq Data Structure

This is developed by Sascha Steinbiss and Stefan Kurtz. The GtEncseq Data Structure is used for storing multiple biological sequences of variable alphabet size, with customizable

character transformations, wildcard support, and an assortment of internal representations optimized for different distributions of wildcards and sequence lengths.

Representations of biological sequences of course need to support nucleotide and amino acid alphabets. However, these come in many different variations, like notation for masked positions, uppercase and lowercase notation and with different sets of wildcard symbols (for sequence positions containing ambiguous nucleotides or amino acids).

2.4 Implementation comparison of GtEncseq with various other methods

In many cases it is necessary to read the sequence in different reading directions (forward, reverse) and, in case of DNA optionally deliver the Watson-Crick complement of a sequence of nucleotides. Other applications require enumeration of k-mers or codon translation.

Current sizes of sequence collections are too large to allow converting or storing these in different formats. Therefore, software implementing a sequence representation needs to accept common sequence formats (e.g. FASTA, GenBank, EMBL, FASTQ), ideally compressed (gzip or bzip2) and uncompressed. While the time-critical low-level part of the code should be implemented in a fast, compiled language like C or C++, it is important to support popular scripting languages like Python or Ruby.

A widespread sequence representation is produced by the formatdb/makedatabase tool from the (Basic Local Alignment Search Tool) BLAST software package. Input to these tools is FASTA, GeneBank or ASN.1. While DNA sequences are stored in a byte compressed format, protein sequences are stored as simple byte arrays. An API to this sequence representation is implemented in the NCBI C/C++ Toolkit [12].

Another widely used sequence representation is the disk-based “2bit” format of the BLAT alignment software [13], which is written in C. It is restricted to DNA sequences in which all non-ACGT characters are automatically mapped to N before encoding takes place. The input sequences must be uncompressed plain sequences or multiple FASTA files.

The SeqAn C++ library [14] provides another sequence representation, specifically designed to be used by other software developers. It allows arbitrarily large alphabets and stores the sequences in an alphabet size-dependent, compressed bit representation. The generic programming approach applied in SeqAn facilitates extensive compile-time optimizations done by the C++ compiler. Therefore SeqAn is only usable in C++ programs.

2.5 Implementation of GtEncseq Data Structure

The implementation occurs maintaining two parts. The first part describes the data structures and retrieval algorithms we use. Special focus is on the efficient handling of wildcard characters and separator positions.

The second part of this section gives an overview of the application programming interface (API).

2.5.1 Sequence Representation

Given a set of sequences, each over the input alphabet Σ∪ Ω where Σ denotes the set of regular characters and Ω denotes the set of ambiguity characters, here called wildcards. In many applications, it is convenient or sometimes even necessary to divide the set of regular characters into α disjoint subsets of equivalent characters A regular character is encoded into the ordinal number of the set it belongs to, i.e. $a \in \Sigma_i$ is encoded into *i.* A wildcard is encoded into some integer # ≥ α . To conveniently handle a set of, say q, sequences s_0,s_1.... , s_{q-1}, each character of each s_1 is encoded, $0 \leq i \leq q - 1$ and concatenated the resulting sequence of integers into one long sequence T, separating two consecutive sequences by some integer $ ≥ α such that $≠ #. $ is the sequence separator. T is a sequence of integers from the set {0,...,α- 1} ∪ {#, $}. The total length of T and by |T| and the number of sequences it represents is ||T||.

Suppose that each sequence s_i comes with a description string d_i. Let D be the concatenation of all descriptions in the order consistent with the order of the s_i in T, using

the new line character as a separator between two consecutive descriptions. The queries to be solved for T and D:

1. For any position j, $0 \leq j \leq |T| - 1$, the query $\mathrm{char}(T, j)$ asks for $T[j]$.
2. For any i, $0 \leq i \leq \|T\| - 1$, $\mathrm{seqstart}(T, i)$ delivers $i + \sum_{t=0}^{i-1} |s_t|$, i.e., the start position of the encoding of s_i in T.
3. For any position j, $0 \leq j \leq |T| - 1$ and any ℓ, $0 \leq \ell \leq |T| - j$, the query $\mathrm{substring}(T, j, \ell)$ asks for the substring $T[j \cdots j + \ell - 1]$.
4. For any j, $0 \leq j \leq |T| - 1$, $T[j] \neq \$$, the query $\mathrm{seqnum}(T, j)$ asks for the largest integer i, $0 \leq i \leq \|T\| - 1$ such that $\mathrm{seqstart}(T, i) \leq j$. In other words, j is an absolute position in T inside the encoding of sequence s_i.
5. For any i, $0 \leq i \leq \|T\| - 1$, $\mathrm{description}(D, i)$ delivers d_i.

Fig 2.2: The queries to be solved for T and D

In the following, the explanations of how to represent T and the input sequences to efficiently answer these queries are presented:

Answering char (T, *j*):

- It is usually better to represent the wildcards and separators in additional data structures. T can be stored in 2 .|T| bits as in each position one of four different integers are stored.
- To decide T[*j*] = {#,$} a new data structure needed to be used. The positions of all separators ($) in T are stored in an array S of length ||T||-1. To save space we virtually divide the positions from 0 to ||T|| into units of size 2^h for some h = 8, 16, 32….
- For any j' satisfying T[j] != $, j' is stored as an integer j' mod 2^h using h bits, belonging to unit j'/2^h.

- For each p={0…j'/2^h} additionally store the smallest index F_s(P) in S where a value belonging to unit p' for some p'>= p is stored.
- Then, the values belonging to unit p are stored in S[F_s(p)…..F_s(p+1)-1]. For a given position, say j, the unit number p = j/2^h and perform a binary search determining if j mod 2^h occurs in the unit.
- The described representation requires

$$(\|T\| - 1) \cdot \frac{h}{8} + \left(\left\lceil \frac{|T|}{2^h} \right\rceil + 1 \right) \cdot \omega,$$

  bytes, where ω is 4 or 8 depending on whether we use the 32-bit or 64- bit version of the representation and h {8,16,32}.This representation is referred to by *S*-unit(h).

- To decide T[j]= #, we store all ranges of consecutive wildcard positions as pairs T(j', *l*). Here, j' is the start position of a range and *l* is its length. Two tables, one for positions and one for lengths, each with r values, the size of this representation is

$$2 \cdot r \cdot \frac{h'}{8} + \left(\left\lceil \frac{|T|}{2^{h'}} \right\rceil + 1 \right) \cdot \omega,$$

- bytes. Given |T|, r , and ω we choose h={8,16,32} minimizing (2). This representation is referred to by W-unit(h').

  The representation of wildcard and separator sequences are representated in the following figure:

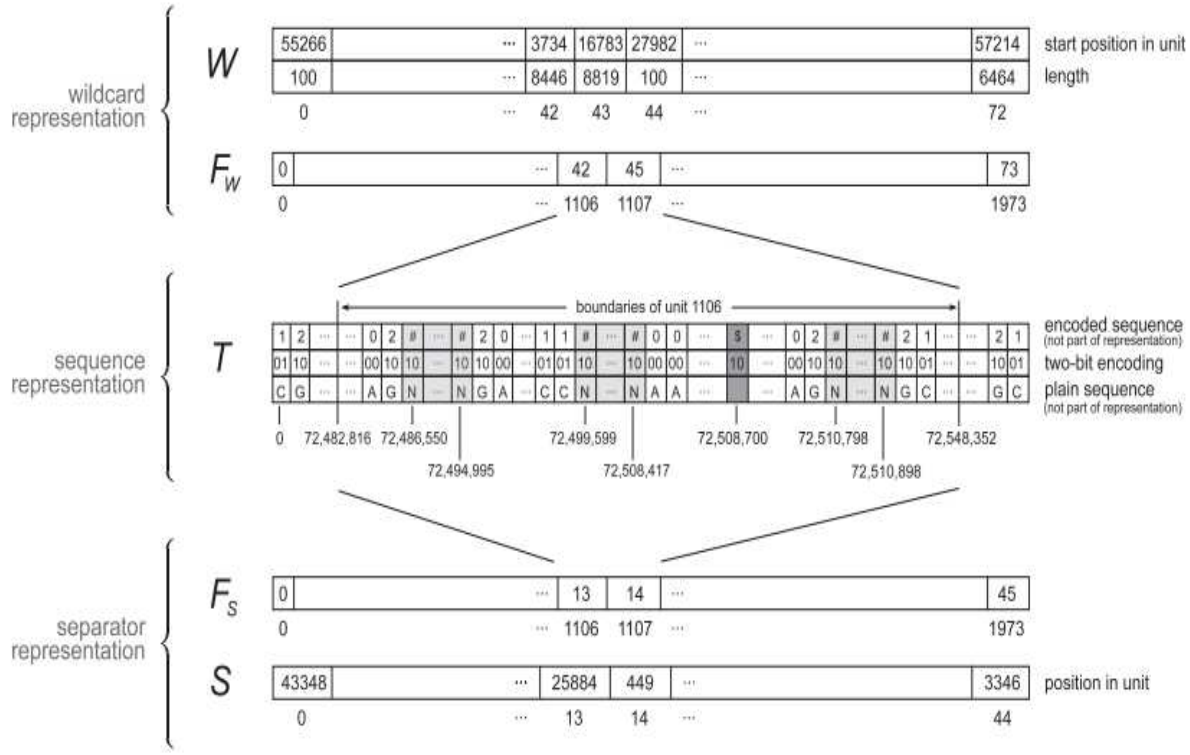

Fig 2.3: the GtEncseq algorithm representation

Answering *substring* (T, *j* ,*L*):

To answer *substring* (T, *j,L*):, only two binary searches for the first position are necessary (one for separators and one for wildcard ranges).

Answering *seqnum* (T,*j*):

First determine the unit number and perform a binary search in unit p to determine some value j' such that either $j'$is the first value in this unit and $j \bmod 2^h < j'$ or j' is the largest value in unit p smaller than $j \bmod 2^h$.

Answering *seqstart* (T, *i*):

In case all sequences have equal length, say m, the *i*th sequence starts at position $i.(m + 1)$ for all $0 \leq i \leq ||T|| - 1$. Thus, the query can be answered in constant time and constant space.

Answering *description* (D, *i*):

To answer the query in constant time, we store the positions of all but the last newline-characters in D.

2.5.2 Application Programming Interface

API for Encoded Sequence Access:

The central class in the encoded sequence interface presented here is called *GtEncseq*. *GtEncseq* objects allow random and sequential access to any substring of the sequence.

In addition to character and substring queries, the *GtEncseq* allows to retrieve the following metadata:

1. The number of sequences, a sequence description of unlimited length for each individual sequence s_i,

2. The length of each individual sequence s_i($|s_i|$), the starting position of each sequence in T.

3. The length of the concatenated sequence (|T|),

4. The number and names of the original input files,

5. The starting positions of each input file in the concatenated sequence,

6. The distribution of encoded characters and the number of wildcards and wildcard ranges.

API for Encoded Sequence Creation and Loading:

The first task is *encoding*, that is, creating a persistent sequence representation on file. Encoding is done by an instance of the *GtEncseqEncoder* class. Use of the encoded sequence requires loading these tables, which use (*GtEncseqLoader*) returns a *GtEncseq* object.

GtEncseqBuilder class implements stateful building of a GtEncseq in memory by successively concatenating C strings.

Alphabet Mapping Definitions:

Alphabet definitions are stored in an alphabet definition file in which the characters appearing on the ith line $0 \leq i \leq \alpha - 1$ are encoded by the integer i. The characters appearing in the last line are the wildcards.

For example, consider the following alphabet definition:

aA

cC

gG

tTuU

nsywrkvbdhmNSYWRKVBDHM

This encodes a and A by 0, c and C by 1, g and G by 2, t, T, u, and U by 3. The characters N, S, Y, W, R, K, V, B, D, H, and M (including their lower case versions) are wildcards. When decoded back to printable ASCII characters, the first character of each line is used to represent characters from the respective line. These are the lower case DNA characters a, c, g, t, and n. While this approach cannot reproduce the original sequence, but rather a sequence of representative characters from each line, additional data structures are used to also (optionally) retrieve the original sequence.

These data structures use a similar combination of the unit-based approach described and bit compression to store positions where a character other than the most frequent character in the line occurs, and a bit-compressed integer to identify the correct character at that position in the set of characters for the class in question. This is a simple and efficient way to implement fully lossless storage, which is useful in situations where, for example, input sequences are soft masked by lower case characters.

2.6 Performance comparison with existing models

Before elaborating the topic we need some overview regarding current data analysis models which have been used effectively. Previously we have talked about BLAST and BLAST. Another very efficient model is “gzip”. The format was designed to allow single pass compression without any backwards seek, and without a priori knowledge of the uncompressed input size or the available size on the output media. If input does not come from a regular disk file, the file modification time is set to the time at which compression started.

Also “bzip2” is a popular model. bzip 2 is a freely available, patent free (see below), high-quality data compressor. It typically compresses files to within 10% to 15% of the best available techniques (the PPM family of statistical compressors), whilst being around twice as fast at compression and six times faster at decompression.

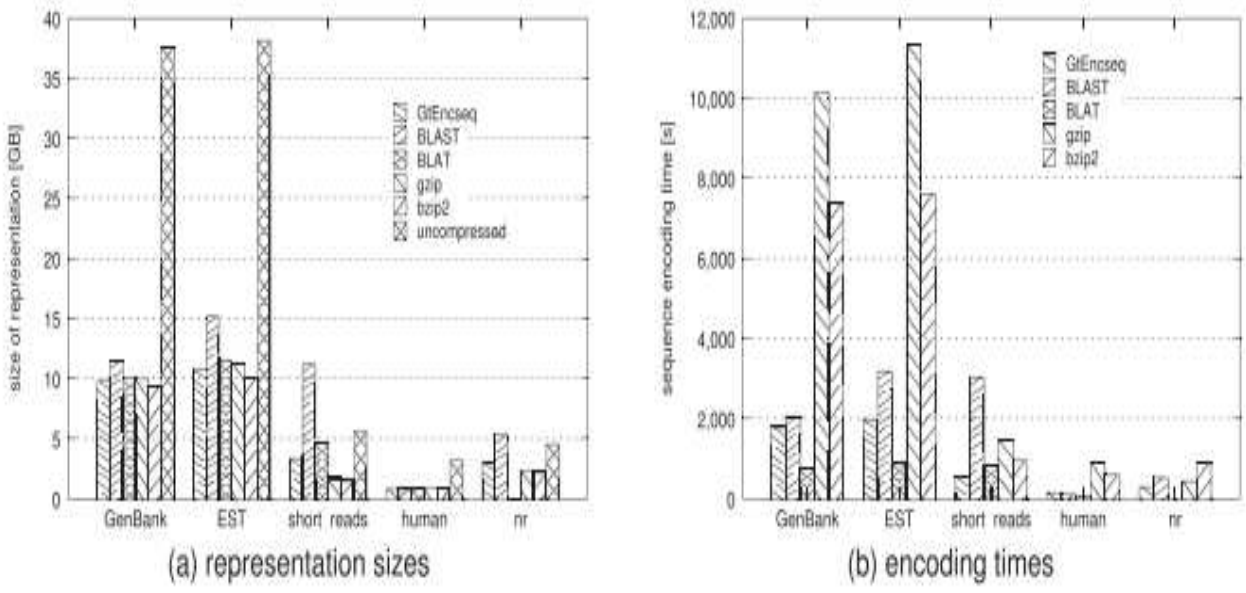

Fig 2.4: comparison for various DNA and protein datasets using different compression model

Chapter 3
Proposed method

We have designed a data structure modifying the existing GtEncseq method for better storage facilities. Our proposal consists of followings:

- Decreasing the concatenated sequence length T.
- Keeping wildcard characters apart from the concatenated sequence into another table for simplicity and reduced storage.
- Using efficient data structure to decode the encoded sequence while necessary.
- Making either sequential or random access efficient despite modification.

3.1 Modified representation of wildcards

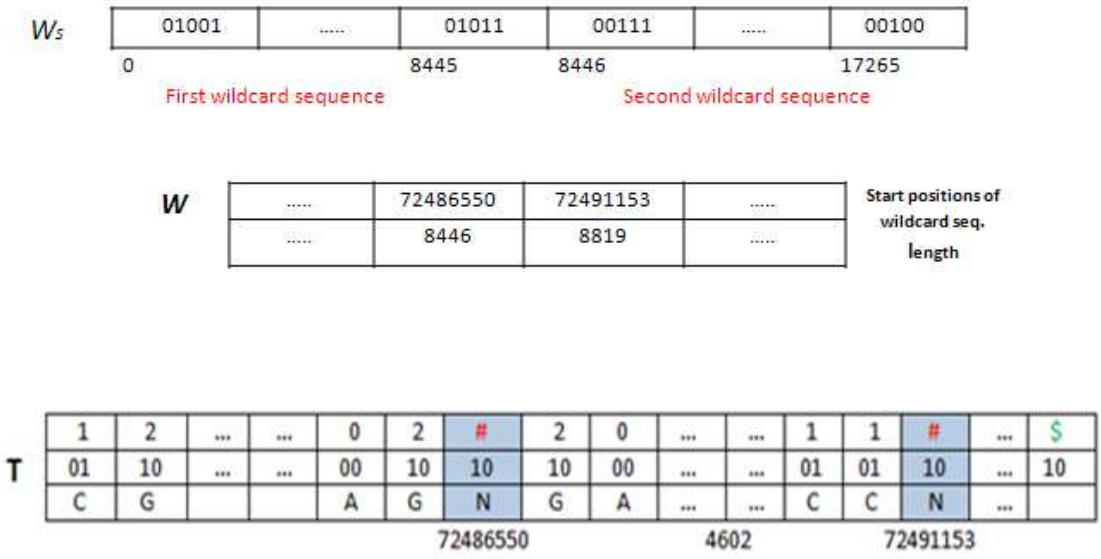

Fig3.1: Wildcard representation for modified algorithm.

In the sequence above represents start position in unit. The sequence in middle represents wildcard length. The sequence below shows the wildcard of our proposed method. Here we can find that the first encounter of wildcard sequence is found at the index 72486550. So in our modified sequence we are using only one index indicating the start position. In the ***Ws*** we are storing the bit compressed values of every wildcard characters. In main sequence the there are 8446 (72494995-72486550) indexes for first wildcard sequence. So in ***Ws*** array index 0 to 8445 occupies the first sequence. Similarly second sequence starts at 72491153 and we keep only one index in the main sequence for that and use index 8446 to 17264 of ***Ws.*** Though the index positions for main sequence are changed for these transformations, we can easily access to previous indexes as we are keeping tracks of each wildcard sequence length. Just adding the length with the previously encountered wildcard sequence start position will result the original indexes.

Similarly we can store the separator indexes in another array and can access from the main sequence without losing data.

3.2 Mathematical analysis of our proposed data structure

To solve char (T, j), we extract the j^{th} pair of bits from the representation of T and interpret it as an integer b. If

b ≠ α, then b = T[j] and we determine it in the following way.

To decide T[j] = $, we store the positions of all separators in T in an array S of length ||T|| - 1.

Consider j = 72486551. Now we convert value of j to original sequence value and find the exact lower value of the wildcard or separator start position. Here 72486551/ 2^{16} = 1106. By unit 1106 we obtain unit start position and length from W. So the absolute start position of this wildcard is $1106 * 2^{16}$ + 3734 = 72486550. The difference between the asked position and the obtained wildcard start sequence is 72486551 - 72486550 = 1, which is less than the length of the wildcard.

So for this query T[j] = #. Which is exactly T[j] = W [100+1] =W [101] = H. Time requirement and space requirement will be same.

Again,

Considering j = 72499598; where j is the index to access. Let wildcard sequence counter c=1.

Converting this value to absolute sequence value 72499598/2^{16} =1106(Unit Value). Absolute start position of wildcard = 1106 * 2^{16} + 3734 = 72486550. The difference between j and absolute wildcard start sequence is 72499598 - 72486550 = 13048 which is larger than the length of wildcard. The value of regular character count is 4602 which is equal to the difference (4602) between 13048 and 8446. So, T[j] = T [72486550+8446+4602]= integer representation.

Finally,

Considering j = 72508702; where j is the index to access. Let wildcard sequence counter c=1. Converting this value to absolute sequence value $72508702/2^{16}$=1106(Unit Value). In Fs we see there is one separator in this unit and its start position at s[13]. After co-ordinate transformation, we obtain correct separator position $1106 * 2^{16}$+ 25886 = 72508700.
So, T[j] = $.

Chapter 4
Implementation

4.1 Used hardware and software

We used a machine consisting 32-bit Intel core i5 microprocessor with 4 GB RAM.
We used Java platform (Netbeans IDE 6.9.1) for the purpose of encoding, decoding, finding time and space requirements, queries and for building a front end which allow user input and display the outputs of these operations.

4.2 Input Specifications

As our primary concern of this thesis is to improve the efficiency of the existing "GtEncseq" data structure in terms of space requirements and time requirements for encoding purposes of large multiple sequences those also consist a considerable amount of wildcard characters, we generated test data of variable sizes. As a 32-bit machine only can support up to 10^8 Boolean array indexes, we are only allowed to make the input according to array size. This is applicable for both main sequence and for the wildcard sequence. We are generating input of multiple sequences where wildcard characters have been placed within the nucleotide sequences. We are using a newline which performs as a separator between two sequences.

4.3 Procedural analysis

For the input we are using text files of variable sizes which contain the test sequences presented in FASTA format. We have used separate arrays for both the encoded main sequence and wildcard sequence.

- **Encoding**

While iterating through the input, the program looks for either any of the nucleotide character, wildcard or separator. If any of the A, T, C or G is found, it encodes into binary 00, 01, 10 and 11. Each time a nucleotide is found, it is encoded in similar fashion and then concatenates with the previous 2-bit binary. For separators, the index value is stored. Another array is made to store the wildcard characters. Whenever such a character is found, it is encoded to the binary '10' in the main sequence and the start position of this character is stored. Then as long as the wildcard sequence continues, the characters are encoded into a 5-bit binary representation whose value is previously determined. Then same iteration and concatenation occurs until the sequence ends. These values are stored in the array. Finally the length of the total wildcard sequence is calculated and stored. So, for any number of wildcard sequence, we have to represent it only with a 2-bit binary in the main encoded sequence.
This procedure is clarified in the following flowcharts:

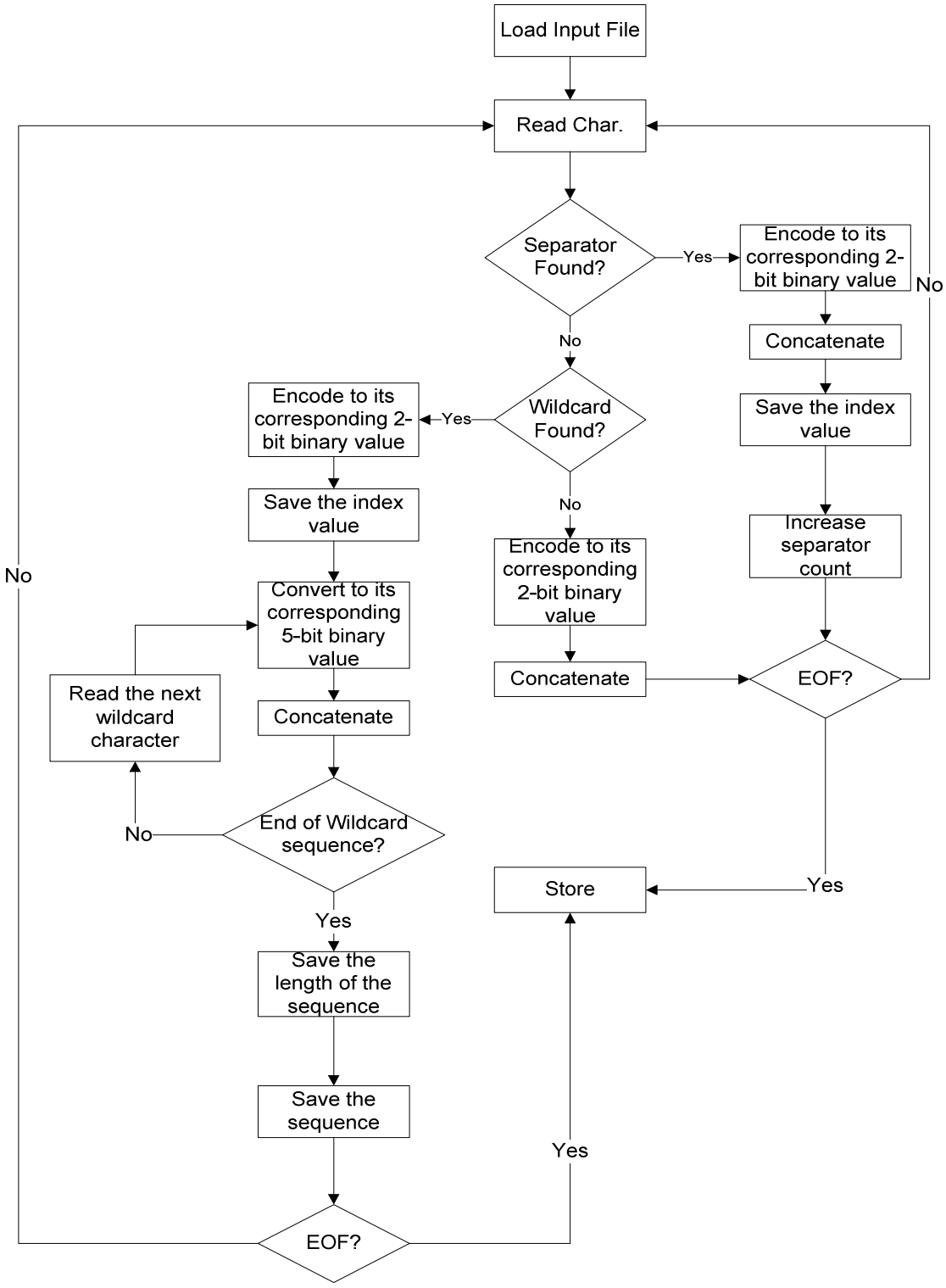

Fig 4.1: Encoding steps of the improved algorithm

- **Decoding**

While decoding the binary representation, whenever the program encounters any of the binary 00, 01 and 11, it decodes to its corresponding character. For the binary '10', three possibilities arise. First it looks into the separator index and matches the index number of the binary. If matches, it concatenates a newline. Else it checks for wildcards and decodes the corresponding 5-bit binary similarly. This continues until the wildcard length ends. If none of these are true, then the program concatenates the character 'C' with the decoded sequence. Fig. shows the steps for decoding.

- **Implementing queries**

There are some query specifications which must be performed accurately with the algorithm. If these queries can produce the expected result for our improved algorithm then it can be said that the algorithm is working as it is expected. The description of these functions was discussed earlier at section 5.3.1. We have used five global functions to perform the same operations.

1. **Finding a specific Character**

We have taken the index number of the main sequence as the input parameter. Then it is checked against the encoded sequence and finally outputs the desired character by decoding. To find the character for a position, both main sequence and wildcard sequence indexes needed to be accessed. If the given index is for a separator, it outputs a newline.

2. **Finding a substring**

The input parameters were given as two integer values which represent as the starting and ending index for the required substring. The output returns the exact subsequence within the range. We have used a built in function to find substring.

3. **Find the sequence start position**

The input parameter was given as an integer value which defines the sequence number. The output returns the index number from which the sequence has been started.

4. **Find the sequence number of a given index**

The input parameter was given as an integer value which defines the index number of a character from the main sequence. The output returns the sequence number where the character is placed. Accessing both main sequence and wildcard sequence is required for calculation purpose.

5. **Find Description**

This function takes the input of an entire sequence number at displays it in output.

The decoding procedure is clarified in the following flowcharts:

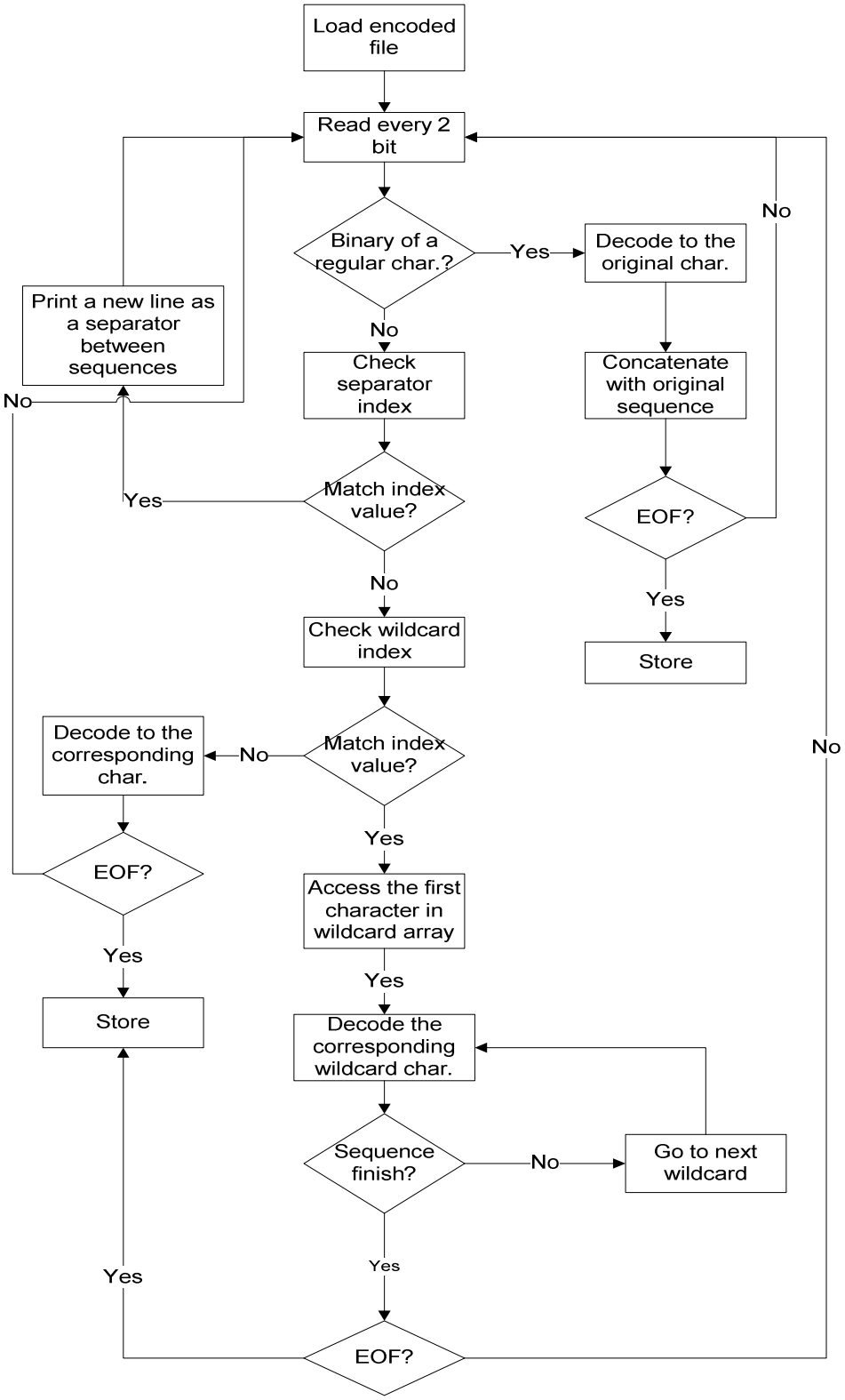

Fig 4.2: Decoding steps of the improved algorithm

4.4 Interface

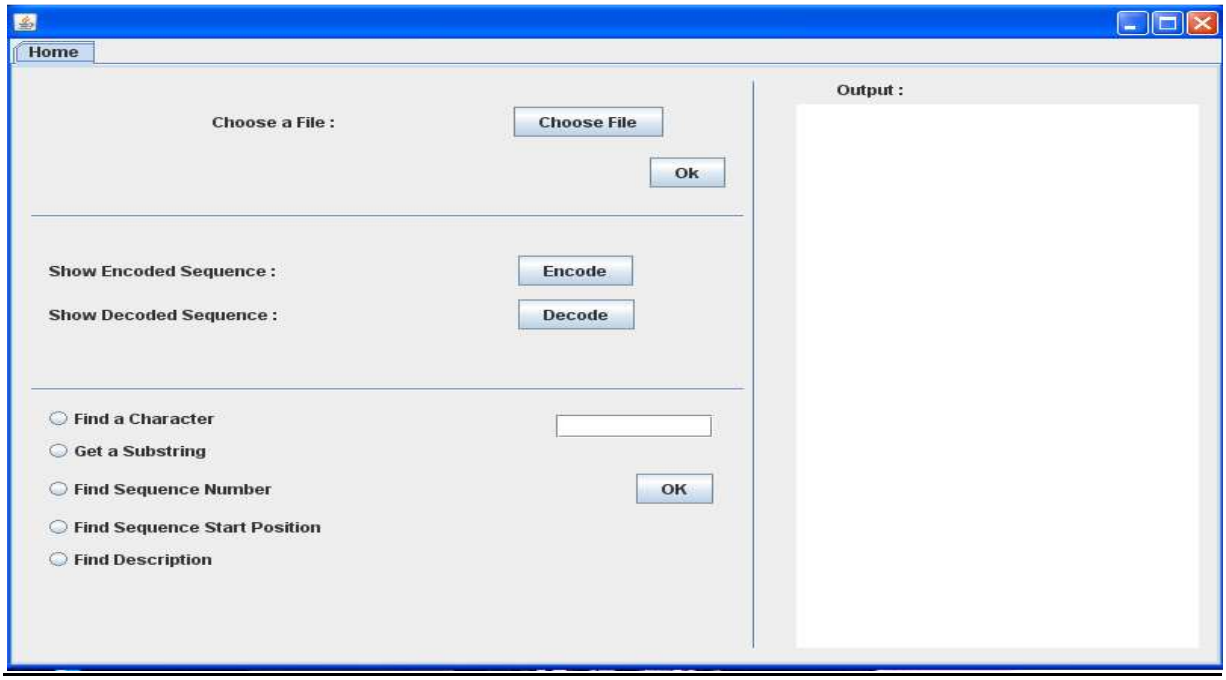

Fig 4.3: Interface before taking input

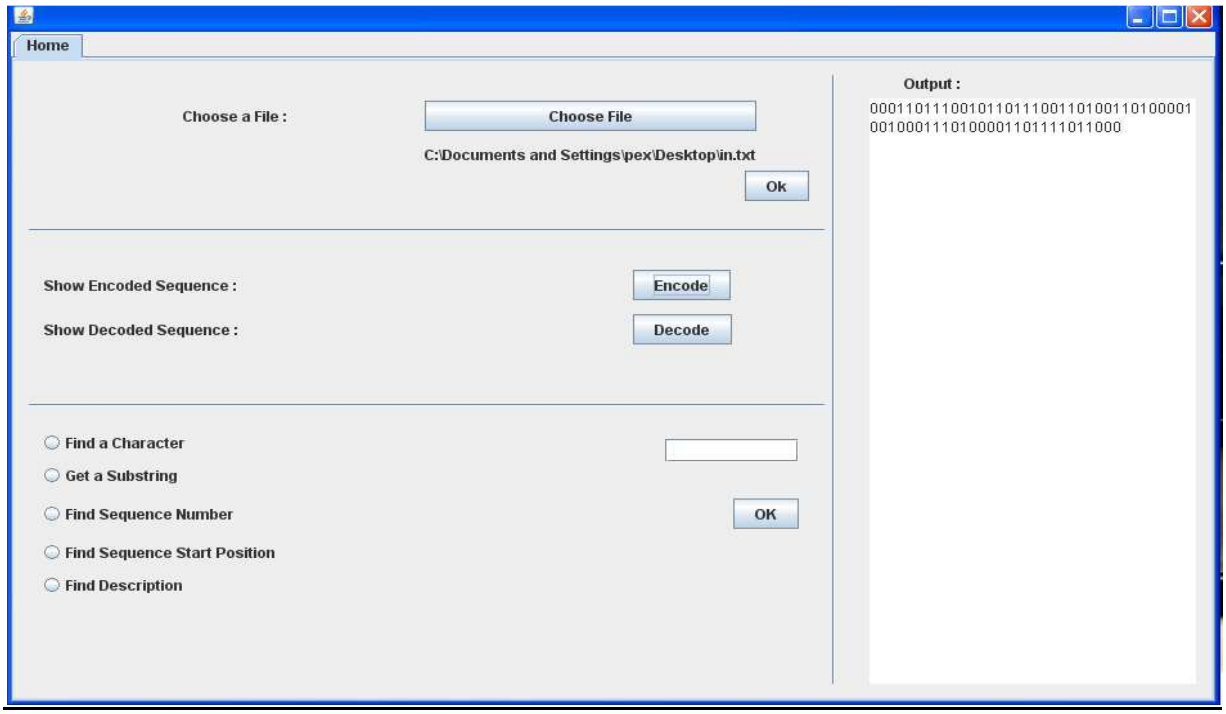

Fig 4.4: Interface after taking a sample input and encoding

Chapter 5
Result Analysis and Discussion

5.1 Research data

The input data was collected from the database https://www.ebi.ac.uk/ipd/mhc/download.html
We took the nucleotide and protein samples in FASTA format for "Canine" and took it from https://www.ebi.ac.uk/ipd/mhc/dla/index.html

We have generated five input text files all consists of variable sizes of DNA sequence manipulated with wildcard characters of different length. At first we have run the test for our proposed data structure and then we have used the same input files to find the output for the existing algorithm. In both cases we have collected the required memory space and time for encoding the whole input file

The following table shows the comparison regarding space requirement between our proposed method and GtEncseq data structure:

Table 5.1: space requirement for our proposed method and GtEncseq

Main sequence size(MB)	Size obtained by our proposed method(MB)	Size obtained using GtEncseq algorithm(MB)	Memory requirement decreased by
0.35	0.08	0.09	11.1%
1.0	0.23	0.26	11.5%
2.01	0.46	0.53	13.2%
7.77	1.75	2.04	14.21%
10.2	2.33	2.71	14.02%

The following table shows the comparison regarding encoding time requirement between our proposed method and GtEncseq data structure:

Table 5.2: encoding time requirement for our proposed method and GtEncseq

Main sequence size(MB)	Encoding time for our proposed method(ms)	Encoding time for GtEncseq algorithm(ms)	Encoding time decreased by
1.34	230	244	5.73%
3.69	486	565	13.98%
6.36	809	848	4.6%
7.77	919	1054	12.81%
10.2	1270	1373	7.50%

5.2 Analysis and comparison

We have calculated the encoded sequence space against the main sequence space for both the existing and proposed algorithm.

Also we have used the built-in JAVA function to calculate the encoding time for both the algorithms.

We have generated the percentage of reduced required space for every sample data. It is clear in the data table that for various sizes of input data the efficiency is increased and saw that the memory space required for encoded sequence in our proposed existing algorithm is less than the requirement for the existing algorithm. The compression efficiency increment is proportional to the increment of main sequence data size. For example for 1.0 MB input data the storage requirement for our proposed method is 11.5% lower than the existing method, and for 10.2 MB data the percentage increases up to 14.02% .

Encoding time efficiency also increases in our proposed method. This time the increment is non-linear, but for every input size, there is a substantial amount of efficiency.

We have prepared graphical representations where the comparison is shown among the methods (fig. 5.1 and 5.2).

The drawback of the proposed algorithm is the increased amount of decoding time than that of the existing algorithm. This can be explained by the fact that we are following a linear approach while decoding the encoded data and as soon as the program finds a wildcard sequence start position, it needs to add the sequence length for retrieving the exact indexes of the original sequence. This addition operation for every wildcard sequence appearance takes more decoding time than the existing algorithm.

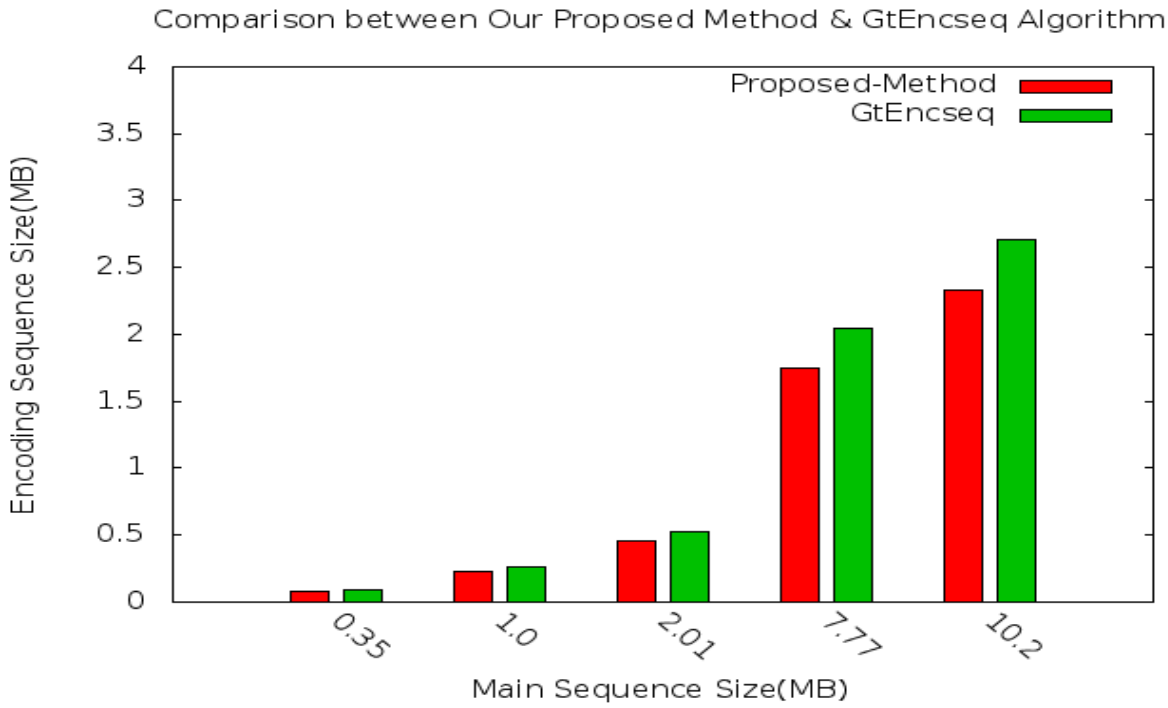

Fig 5.1: graphical representation regarding space requirement for encoded sequence between the existing and proposed algorithm.

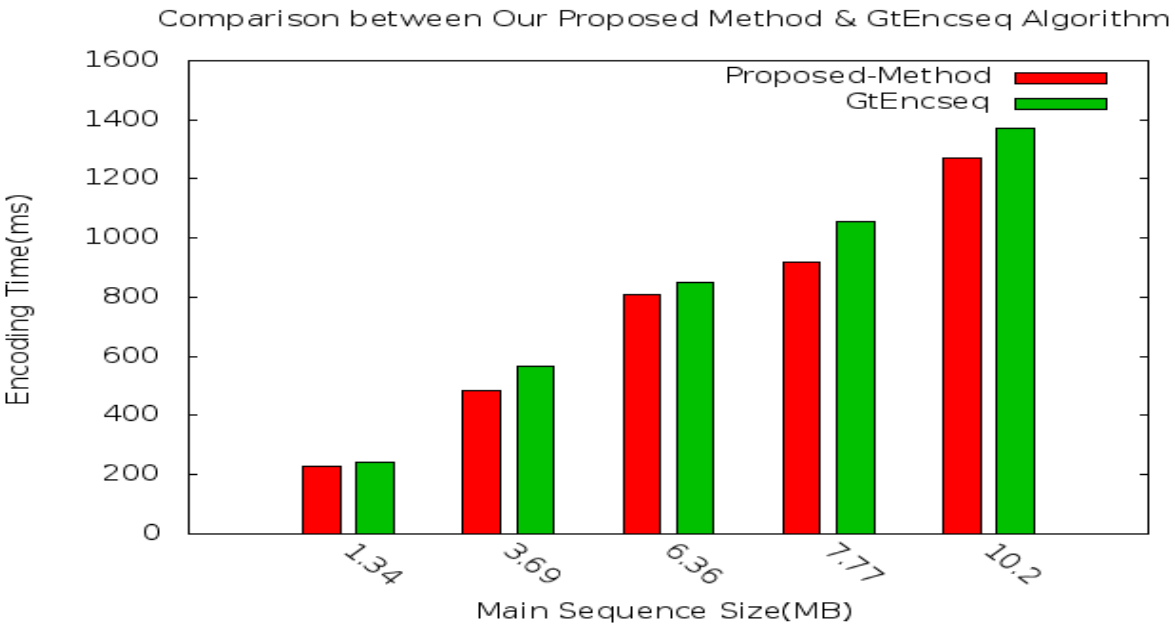

Fig 5.2: graphical representation regarding time requirement for encoded sequence between the existing and proposed algorithm.

Chapter 6

Conclusion and future scope

Our proposed method will save a large amount of space and it will be a lossless method. The encoded sequence is much smaller than the GtEncseq representation, especially in the presence of large number of wildcard chunks. Still there are opportunities to improve its performance in terms of time requirement for accessing main sequence and decoding the encoded sequence. To work with huge amount of data (Gigabyte data) is another goal to achieve. For this, we require higher configuration machineries which are not available for the moment.

.

References

[1] Sascha Steinbiss and Stefan Kurtz, "*A New Efficient Data Structure for Storage And Retrieval of Multiple BIOsequences".*

[2] Shanika Kuruppu, Bryan Beresford-Smith, Thomas Conway, and Justin Zobel, "*Iterative Dictionary Construction for Compression of Large DNA Data Sets".*

[3] Hieu Dinh and Sanguthevar Rajasekaran, "*A memory-efficient data structure representing exact-match overlap graphs with application for next-generation DNA assembly*".

[4] Sheng Bao, Shi Chen, Zhi-Qiang Jing and Ran Ren, " *A DNA Sequence Compression Algorithm Based on LUT and LZ77*".

[5] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, and E.W.Sayers, "*GenBank," Nucleic Acids Research*, vol. 38, (Database Issue), pp. D46-D51, 2010.

[6] A. Morgulis, G. Coulouris, Y. Raytselis, T.L. Madden, R. Agarwala, and A.A. Schaffer, "*Database Indexing for Production MegaBLAST Searches*," Bioinformatics, vol. 24, no. 16, pp. 1757-1764, 2008.

[7] Srinivasa K. G , Jagadish M , Venugopal K R ,LMPatnaik, "*Efficient Compression of non-repetitive DNA sequences using Dynamic Programming".*

[8] E. Rivals, J-P. Delahaye, M. Dauchet, and 0. Delgrange. *A guaranteed compression scheme for repetitive dna sequences*." LIFL Lille I Univerisity technical report, page 285, 1995.

[9] Raffaele Giancarlo∗, Davide Scaturro and Filippo Utro ,"*Textual data compression in computational biology: a synopsis"* Dipartimento di Matematica ed Applicazioni, Università di Palermo, Palermo, Italy.

[10] Marty C. Brandon, Douglas C. Wallace and Pierre Baldi, "*Data structures and compression algorithms for genomic sequence data".*

[11] Gergely Korodi and Ioan Tabus, "*Compression of Annotated Nucleotide Sequences".*

[12] "*The NCBI C Toolkit*," ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools, 2011.

[13] W.J. Kent, "*BLAT-the BLAST-Like Alignment Tool*," Genome Research, vol. 12, no. 4, pp. 656-664, 2002

[14] A. Do ¨ ring, D. Weese, T. Rausch, and K. Reinert, "*SeqAn an Efficient, Generic C++ Library for Sequence Analysis*," BMC Bioinformatics, vol. 9, article 11, 2008.

Appendix

- **Used Java classes and Functions:**

➢ **Classes:**

✓ GtAlphabet.java
✓ Mainframe.java
✓ MainClass.java

➢ **Functions:**

- ❖ public static void main(String[] args) {}
- ❖ public void openFile(String file) {}
- ❖ public void readFile() {}
- ❖ public void closeFile() {}
- ❖ public String noOfSubbSequence(){}
- ❖ public String totalLength(){
- ❖ public String encodeSequence() {}
- ❖ public String decodeSequence() {}
- ❖ public String getSubstring(int j, int l){}
- ❖ public String getChar(int temp) {}
- ❖ public String getSequenceNumber(int temp) {}
- ❖ public String getSequenceStartPosition(int temp) {}
- ❖ public String getDescription(int temp) {}
- ❖ public MainFrame() {}
- ❖ private void chooseFileButtonActionPerformed(java.awt.event.ActionEvent evt) {}
- ❖ private void okButtonActionPerformed(java.awt.event.ActionEvent evt) {}
- ❖ private void decodeButtonActionPerformed(java.awt.event.ActionEvent evt) {}
- ❖ private void encodeButtonActionPerformed(java.awt.event.ActionEvent evt) {}
- ❖ private void okButton2ActionPerformed(java.awt.event.ActionEvent evt) {}