DNA Compression using Referential Algorithm

Project Report submitted in partial fulfillment of the requirement for the degree of

Master of Technology

in

Computer Science and Engineering

under the Supervision of

Dr. S.P. Ghrera (H.O.D)

by

Kanika Mehta (132225)

to

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled DNA Compression using Referential Algorithm, submitted by Kanika Mehta in partial fulfillment for the award of degree of Master of Technology in Computer Science and Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date:
Supervisor's Name
Designation

Acknowledgements

"Foremost, I would like to express my sincere gratitude to my supervisor Dr.S.P.Ghrera for the continuous support for my masters study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my masters study.

Besides my supervisor, I would like to thank Prof. Suman Saha, for his valuable guidance,support, encouragement and insightful comments.

My sincere thanks also goes to Prof. Pardeep Khokhar , my coordinator for his support and leading me working on this exciting project.

Last but not the least, I would like to thank my family: my parents Vinod Mehta and Rama Mehta, for giving birth to me at the first place and supporting me spiritually throughout my life."

Date:
Kanika Mehta

CONTENTS

LIST OF FIGURES

ABBREVIATIONS

DNA	**D**eoxyrib **N**ucleic **A**cid
A	**A**denine
C	**C**ytosine
G	**G**uanine
T	**T**hymine
FRESCO	**F**ramework for **R**eferential Compression of Highly **S**imilar Sequences
C1	**C**ampylobacter coli 15-537360
C2	**C**ampylobacter Coli CVM)
C3	**C**ampylobacter jenjuni 002426
C4	**C**ampylobacter jenjuni 002538
C5	**C**ampylobacter jenjuni002425
C6	**C**ampylobacter jenjuni RM1221
C7	**C**ampylobacter jenjuni NCTC11168
E1	**E**scherichia coli strain k12(E1)
S1	**S**higella sonnie 53G

Jaypee University of Information and Technology Waknaghat,
Solan – 173234 Himachal Praesh.

Abstract

Prof. S.P.Ghrera
Computer Science and Engineering

Master Of Technology

DNA compression using Referential Compression

by Kanika Mehta

With rapid technological development and growth of sequencing data, an umpteen gamut of biological data has been generated. As an alternative, Data Compression is employed to reduce the size of data. In this direction, this paper proposes a new reference-based compression approach ,which is employed as a solution. Firstly, a reference has been constructed from the common sub strings of randomly selected input sequences. Reference set is a pair of key and value, where key is a fingerprint (or a unique id) and value is a sequence of characters. Next, these given sequences are compressed using referential compression algorithm. This is attained by matching the input with the reference and hence, replacing the match found in input by its fingerprints contained in the reference, thereby achieving better compression. The experimental results of this paper show that the approach proposed herein, outperforms the existing approaches and methodologies applied so far.

CHAPTER 1

INTRODUCTION

1.1 Problem statement

Molecular sequence databases (e.g., EMBL , Genbank ,etc) store hundreds or thousands of DNA sequences reaching up to thousands of gigabytes and are continuously growing with storage doubling time of 18 months [18]. Due to this large size, they cannot be downloaded or shared over the network by anyone except those with large amount of available resources. In such a situation, the data is transferred in hard disks from one place to the other, which results in wastage of time and money. Even though one downloads specific data sets, the load on local storage can be very high, and such large-scale analysis of these sequences is only possible to those with large computing resources.

While using cloud computing, one again faces the problem of large data size during the transmission of the data to the cloud. However, with the growth of sequence data largely exceeding reasonable storage capability, the bio medical community has to confront the challenges with the management, transfer and storage of sequence data.

1.2 Motivation

The expansion of high-throughput sequencing data has given birth to the generation of a large amount of biological data. Although the cost involved in DNA sequencing is decreasing, yet the storage expenditure of these sequences is increasing exponentially. There are three approaches to handle such huge data[6]:

- add more storage to accommodate more data
- remove some of the data that is either redundant or not so important
- compress the stored data to avail more storage

Data compression that reduces the space for storage and speeds up the data transmission is one of the key technologies that can be used in the management of such vast data.

1.3 Objective

- To suggest an improvement in the compression of files in real time
- To study inter-species referential compression as so far referential compression only works well, if input and reference belong to the same species. The development of a multi-species reference sequence would allow for multi - purpose genome compression algorithms.

1.4 Organization of thesis

This section discusses the framework of this thesis which is organized as follows:

- Chapter 1: This chapter introduces the problem statement, motivation, objective of thesis.
- Chapter 2: About DNA read sequencing, need for compression, types of compression, performance parameters and referential compression.
- Chapter 3: Literature survey
- Chapter 4: Proposed Approach
- Chapter 5: Implementation Results
- Chapter 6: Conclusion and future directions

CHAPTER 2

ABOUT DNA READ COMPRESSION

2.1 What is DNA sequencing?

The Deoxyribonucleic acid (DNA) constitutes the physical medium in which all properties of living organisms are encoded. The biological information stored inside a DNA helps in determining the weaknesses or abilities of an organism. DNA sequence is a long stretch consisting of four types of nucleotides: Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). There can also be many unknown

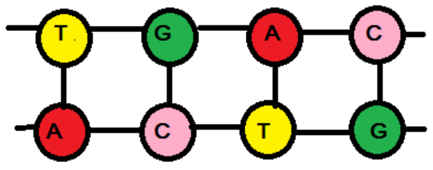


FIGURE 2.1: Nucleotide ladder

nucleotides in a DNA sequence that are represented by N.DNA sequencing means to determine the arrangement of the nucleotides in a sample of a DNA. This can be helpful in finding the properties of the organism, determining as to which diseases an organism is sensitive, treating people according to their DNA and lot more. Due to the growth in technology and new discoveries DNA sequencing has become faster and cheaper. One complete DNA sequence of a human being contains three billions nucleotides and roughly needs around three gigabytes of memory.

2.2 Need for compression

Generally for presenting digital data we use encoding which produces large amount of data, since data transmission and storage is very expensive. So we need to look in two things, firstly save transmission bandwidth for transmission of data and secondly save the space for storage of data. For this reason we use compression techniques. By compressing data, we mean to compress or reduce the size of data for easy transmission and easy data storage. Converting back the compressed data back to original data is known as decompression. Compression methods are classified into: "lossy" and "lossless". A lossless technique means that the restored data file is exactly similar to its original file as in case of word document .Whereas, in lossy compression technique the original data is lost, that is, unimportant data is discarded (but no harm is done) as in the case of images or signals where the loss of certain bits is affordable.

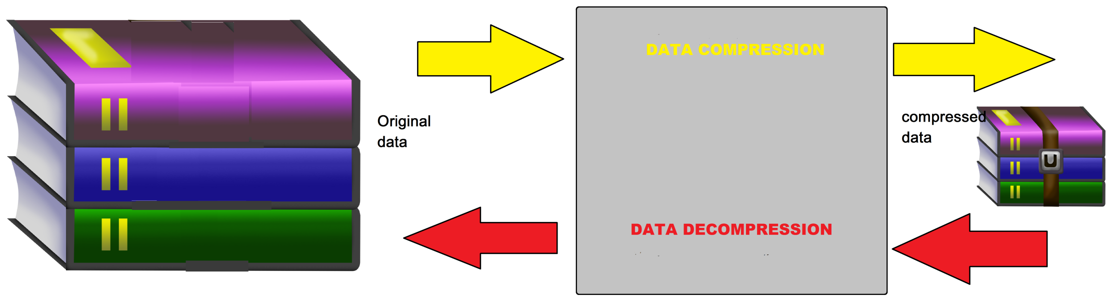


FIGURE 2.2: File compression and decompression

It was back in 1948 when Claude Shannon discovered that there is certain limit to lossless data compression called as entropy (H). The value of this entropy depends on statistical behavior of the source. This means that lossless compression with compression rate greater or equal to H is possible but impossible to achieve better than H. Suppose X is a random variable with alphabets (x1,x2,x3.........xn) where P(x) is the probability of every letter present in the alphabet, the H is calculated as :

$$H(X) = -\sum_{1}^{N} P_i \log P_i$$

2.3 Compression Performance Parameters

In order to compare different compression and decompression algorithms we need to define certain parameters that can be consistently used to evaluate their performance. In this dissertation three criteria are taken, namely compression speed of the algorithm, compression ratio and percentage of storage saved, defines as follows.

Compressed ratio Compressed ratio is defined as the reduction in the size of file after we apply some compression algorithm on the original file. Basically it is ratio of original file size to the reduced file size. The higher the compression ratio ,the better is the compression. This is calculated as

$$Compressionratio = \frac{originalfilesize}{compressedfilesize}$$

Compression and decompression speed Compression speed is the total time taken in seconds, to compress the given data. It can also denote as the total number of characters compressed per second. The lower it takes to compress the better is the speed. After compressing and storing the data it is possible to decompress it in order to access the original data. Decompression speed is measured as the time taken to get back the original data from the compressed data.

Memory saving Memory saving is measured as the reduction in the amount of storage after we apply the compression algorithm. Same data now occupies less storage space. Memory saving in percentage is calculated as

$$Memorysaved = 1 - \frac{compressedsize}{uncompressedsize}$$

$$PercentageS = memorysaved \times 100$$

2.4 Types of DNA compression

The increasing number of (re-)sequenced genomes has lead to many compression algorithms. In general, these compression algorithms can be separated into bit manipulating, dictionary-based, statistical, and referential approaches [4]:

- **Naive bit encoding :** is the most simple algorithm that uses fixed-length encoding of two or more characters in a single byte. The four nucleotides bases in a DNA can easily encoded with two bits. However this encoding technique is difficult to read by user. If a DNA sample contains N spaces then we require three bits for encoding .One example for naive bit encoding is shown in Figure 2.3 . Each symbol in the input is replaced by two bits using the replacement (A=00;C=01;G=10; T=11).
- **Dictionary-based ::** Dictionary-based encoding are compression schemes that doesn't depend on specific characteristics of the input data. The main

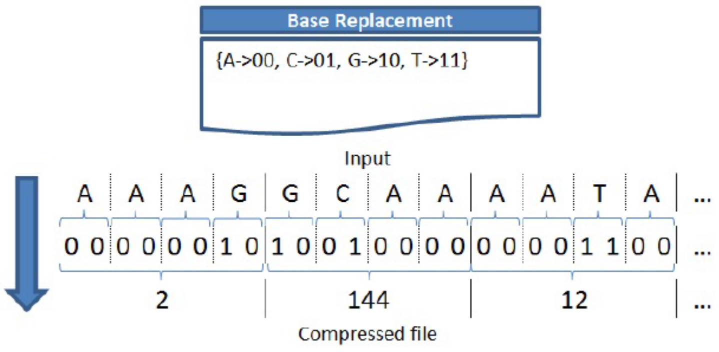


FIGURE 2.3: Example Naive bit encoding

idea is to replace the repeated data items of the input DNA sequence with references to a dictionary. These reference are stored as codeword that are used for encoding the input sequence. Repetitions are usually detected by keeping a counter for previously occurring sequences. This reference dictionary need not be stored along with the compressed data as it can be recreated dynamically during the decompression process. One example for a dictionary-based algorithm is shown in Figure 2.4

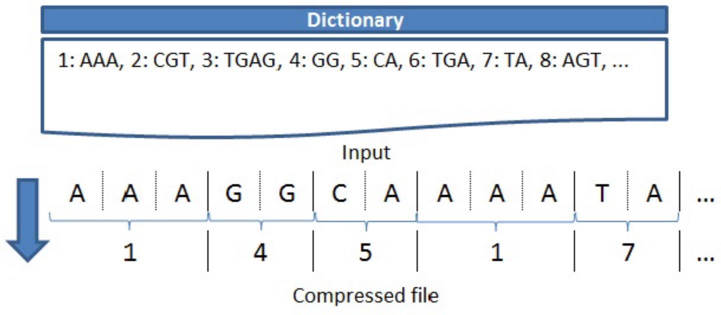

Dictionary
1: AAA, 2: CGT, 3: TGAG, 4: GG, 5: CA, 6: TGA, 7: TA, 8: AGT, ...
Input
A A A G G C A A A A T A ...
1
4
5
1
7
...
Compressed file

FIGURE 2.4: Example Dictionary-based encodings

- **Statistical based Algorithms :**Statistical algorithms generate a statistical model of the given input data, which is usually denoted as probabilistic or prefix tree data structure. More frequent Sub sequences are represented with shorter codes. This scheme can be considered better than that dictionary based algorithm as it performs repeat detection and reference encoding in one pass. Compression ratio depends occurrence of visible patterns in the input as well as the quality of model used. One example for a Statistical based algorithm is shown in Figure 2.5.

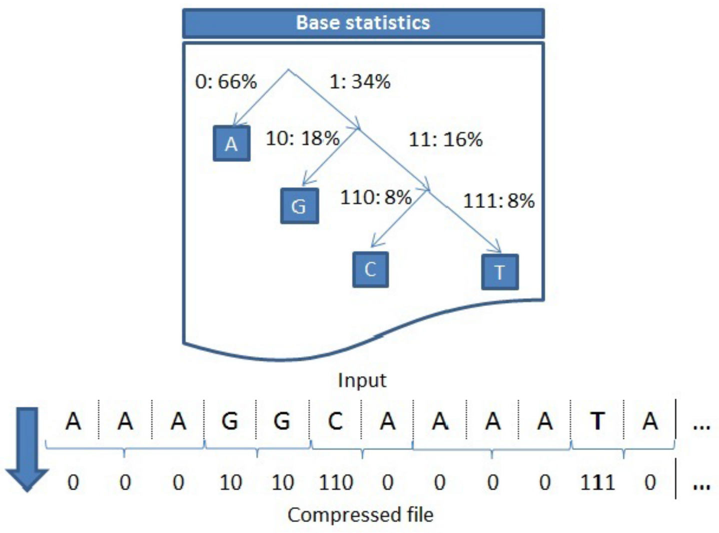


FIGURE 2.5: Example Statistical-based encodings

- **Referential or reference-based approaches:** to some extend similar to dictionary based techniques, as they replace long sub sequences with references. But the disadvantage of this scheme is that the references refer to external sequences which are not part of the to-be-compressed input [16] and also need to be stored along with compressed inputs. The idea is to compress the input sequence of same specie with respect to an external reference sequence.Index structures like suffix tree or hash based indexing is used to search for Long matches in the reference 2.4 Referential compression

2.5 Referential compression

Referentially compressing a string means to encode the string as a concatenation of substrings from a given reference sequence. A number of schemes are used to

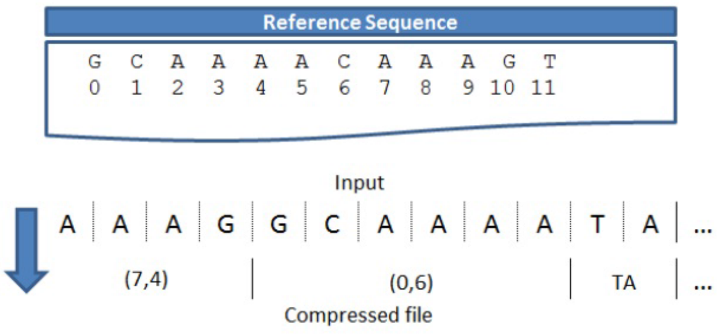


FIGURE 2.6: Example Reference-based encodings

denote the matches found in a reference sequence. The first scheme is to encode the matches as a set of pairs, where each pair consists of a position and length of a match [3][11]. Second scheme is to encode parts of a sequence with original text entries instead of matches into the reference[1]. This scheme is only useful if the referential match values are small enough forming a compressed representation of the text occupying less storage. Third scheme is to encode each match value found in the reference as a triplet, where first item denotes starting position of the match, second denotes the length of the match, and third denotes first character appearing after the match. This technique yields very good results in case the target input sequence and reference sequence are almost similar with a difference of single nucleotide.

Example 1:We are given two sets : –

reference ref = ATGCGAGCT

Sequence s = ATTCGAGACT

This could be represented as given in the figure 2.7 . A referential match entry is denoted as a triplet of (start; length; mismatch) where *start* is an integer indicating the start of a match within the reference, *length* denotes the match length, and *mismatch* denotes a symbol following the match. The length of a referential match entry(rme), is length + 1.

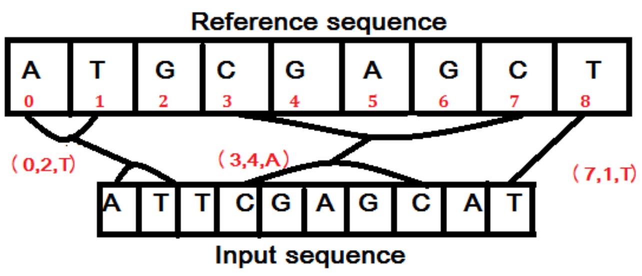


FIGURE 2.7: Example 1 Reference-based compression

Given strings s and ref, a referential compression of s with respect to ref is a list of referential match entries,

Comp (s,ref)=({ *start1; length1; mismatch1*}...... *startn; lengthn; mismatchn*});

such that

(ref start1; length1* mismatch 1)*

(ref start2; length2 * mismatch 2)*...... *

(ref startn; lengthn * mismatch n) = s

Algorithm 1 : Referential compression algorithm

```
1: while input sequence is not empty do
2: find longest matching substring in reference for current input position
3: if length of match greater than X then
4: encode match as (match position; length)
5: else
6: encode match with raw symbols
7: end if
8: end while
```

The general algorithm for referential compression in pseudo code is shown in Algorithm 1. In the above pseudo code the value of X is used to find out whether the entry is encoded as short matches of reference or as raw strings. The inverse of a referential compression is the decompression of a referential compressed sequence using the same reference in order to get back the original string as it was. More is the similarity between reference and input sequences higher are the compression rates. For example if we compress a human genome sequence with a mouse genome we get very low rates of compression

CHAPTER 3

BRIEF LITERATURE SURVEY

3.1 Title 1:FRESCO -Referential Compression of Highly Similar Sequences

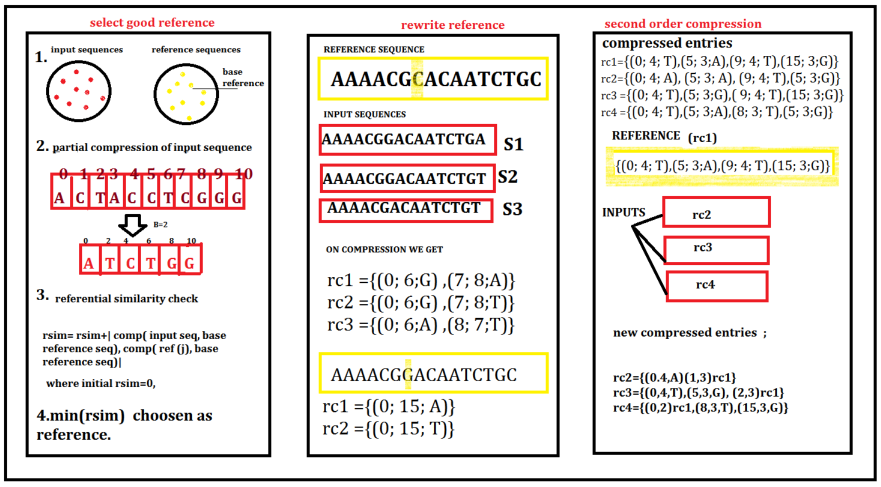

select good reference
1. input sequences
reference sequences
base reference
2. partial compression of input sequence
0 1 2 3 4 5 6 7 8 9 10
A C T A C C T C G G G
B=2
0 2 4 6 8 10
A T C T G G
3. referential similarity check
rsim= rsim+| comp(input seq, base reference seq), comp(ref (j), base reference seq)|
where initial rsim=0,
4.min(rsim) choosen as reference.
rewrite reference
REFERENCE SEQUENCE
AAAACGCACAATCTGC
INPUT SEQUENCES
AAAACGGACAATCTGA S1
AAAACGGACAATCTGT S2
AAAACGACAATCTGT S3
ON COMPRESSION WE GET
rc1 ={(0; 6;G) ,(7; 8;A)}
rc2 ={(0; 6;G) ,(7; 8;T)}
rc3 ={(0; 6;A) ,(8; 7;T)}
AAAACGGACAATCTGC
rc1 ={(0; 15; A)}
rc2 ={(0; 15; T)}
second order compression
compressed entries
rc1={(0; 4; T),(5; 3;A),(9; 4; T),(15; 3;G)}
rc2={(0; 4; A), (5; 3; A), (9; 4; T),(5; 3;G)}
rc3 ={(0; 4; T),(5; 3;G),(9; 4; T),(15; 3;G)}
rc4 ={(0; 4; T),(5; 3;A),(8; 3; T),(5; 3;G)}
REFERENCE (rc1)
{(0; 4; T),(5; 3;A),(9; 4; T),(15; 3;G)}
INPUTS
rc2
rc3
rc4
new compressed entries ;
rc2={(0.4,A)(1,3)rc1}
rc3={(0,4,T),(5,3,G), (2,3)rc1}
rc4={(0,2)rc1,(8,3,T),(15,3,G)}

FIGURE 3.1: FRESCO: Block diagram

Referential Compression of Highly Similar Sequences[17] is an open source framework proposed by Sebastian Wandelt and Ulf Leser that compresses large volumes of biological sequence data. The algorithm is much faster as compared to related work while attaining similar compression ratios. Apart from this they have also proposed three techniques to increase the compression ratio. This includes selecting a good reference, Rewriting a good reference and second order compression. Using modern hardware they are able to perform real time compression on highly similar sequences. But rewriting a reference sequence algorithm does not work with long chains.

3.2 Title 2:Adaptive efficient compression of genomes

In another paper by Sebastian Wandelt* and Ulf Leser, entitled as[1] , they propose a highly scalable, efficient and adaptive algorithm for referential compression of genomes. They were able to achieve a very good compression ratio by a graceful trade off between compression time and space requirement. The main idea is to divide the reference genome into number of blocks where local search is used to find long matches of the given sequence in the reference blocks. Longest suffix and prefix matches are computed using suffix tree[16]. For obtaining better compression speed and handling data easily they have considered fixed length blocks whereas different blocks size can have provided better performance in terms of compression ratio. The limitation here is that their algorithm mainly works when compressing a sequence with respect to a reference from the same species.

3.3 Title 3:GReEn- a tool for efficient compression of genome resequencing data

GReEn[2] is another compression tool designed by Ar-mando J. Pinho, Diogo Pratas and Sara P. Garcia that can even handle arbitrary values that may be present in biological sequences without imposing any restriction or specification

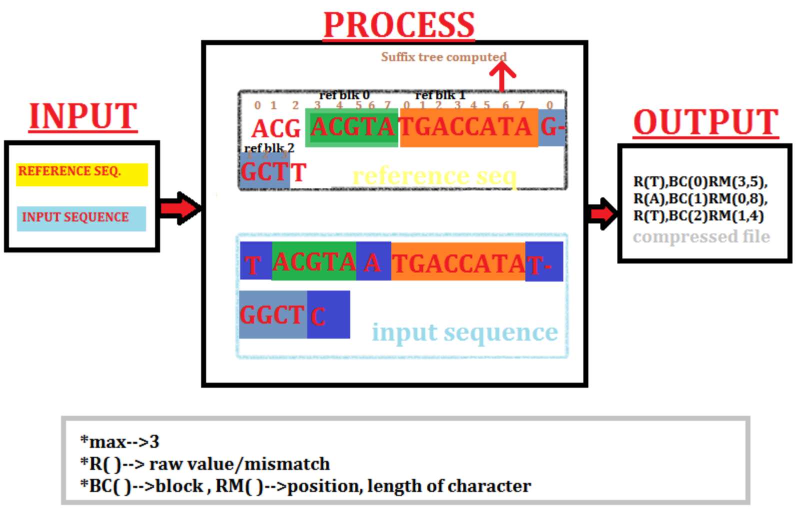


Figure 3.2: Adaptive efficient compression of genomes

on the to-be- compressed sequence. In this reference based compression, each character in the input sequence along with its probability distribution is encoded using an arithmetic encoder. They have used either static model or adaptive mode to provide probability distribution for the characters. The best portion in this tool is that it works on statistical method that uses probabilistic copy model that allows a high compression rate of the input sequence, especially when the input and reference sequence are almost similar. The copy model relies on a pointer to a position in the reference sequence that has a good chance of containing a character identical to that being encoded. As the encoding process of the input sequence proceeds, the pointer associated with the copy model may be re positioned to different locations of the reference sequence. On re-positioning ,all the parameters are reset. They have used three counters to record how many times copy model is used, how many times the model made correct guess including both upper and lower case and the number of times the model guessed the character but failed the case. The only Limitation is that, its running time depends only on the size of the sequence being compressed.

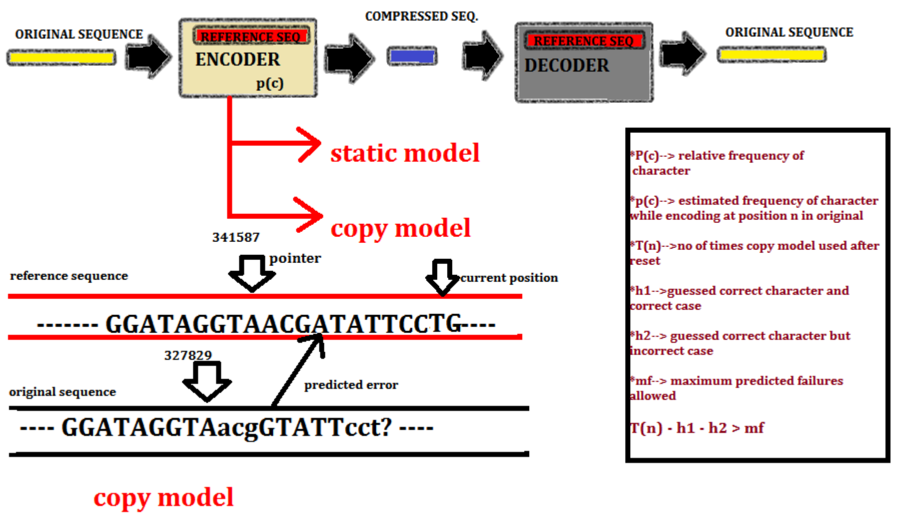

ORIGINAL SEQUENCE
REFERENCE SEQ
ENCODER
p(c)
COMPRESSED SEQ.
REFERENCE SEQ
DECODER
ORIGINAL SEQUENCE
static model
copy model
341587
pointer
reference sequence
current position
------- GGATAGGTAACGATATTCCTG----
327829
original sequence
predicted error
---- GGATAGGTAacgGTATTcct? ----
copy model
*P(c)--> relative frequency of character
*p(c)--> estimated frequency of character while encoding at position n in original
*T(n)-->no of times copy model used after reset
*h1-->guessed correct character and correct case
*h2--> guessed correct character but incorrect case
*mf--> maximum predicted failures allowed
T(n) - h1 - h2 > mf

Figure 3.3: Copy Model

3.4 Title 4:String Searching in Referentially Compressed Genomes

[10]*String Searching in Referentially Compressed Genomes* by Wandelt1 et al have addressed the problem of decompressing before an analysis by proposing an algorithm for exact string search over compressed data. The algorithm performs partial decompression over the referentially compressed sequence of a genome without any index structure over the input sequence. The problem of string matching is solved by first finding exact matches in the indexed reference sequence and then used those matches for finding all exact matches in the given genomes . This concept of searching a biological sequence directly in a compressed sequence introduces new techniques to manage data in research groups.

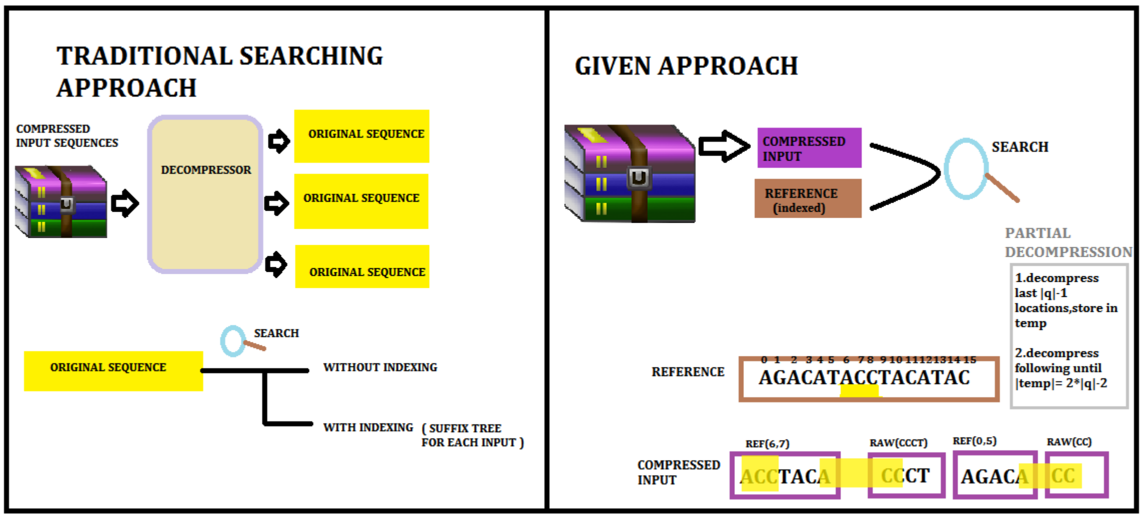


FIGURE 3.4: String Searching in Referentially Compressed Genomes Read Compression

3.5 Title 5:MCUIUC– A New Framework for Metagenomic Read Compression

MCUIUC [11] is a framework proposed by Jonathan et al, for compressing Metagenomics data. This includes (1) roughly classifying the species in the metagenomic sample and select group of genera,(2) partitioning the dataset where a set of reference genomes that best aligned to the reads has to be selected. Bowtie2 is used for giving best alignment score up to the genus level. Both aligned and unaligned reads are treated differently. (3) Compression and Distribution: After number of conversions from SAM format to BAM format and finally to CRAM format the compressed files are packed as tar archive.This package can be distributed and the files are recreated with the help of CRAM toolkit , given the CRAM files and representative genomes. The representative genomes may be compressed using standard compressors such as bzip2 or specialized compressors such as DNA Compress.

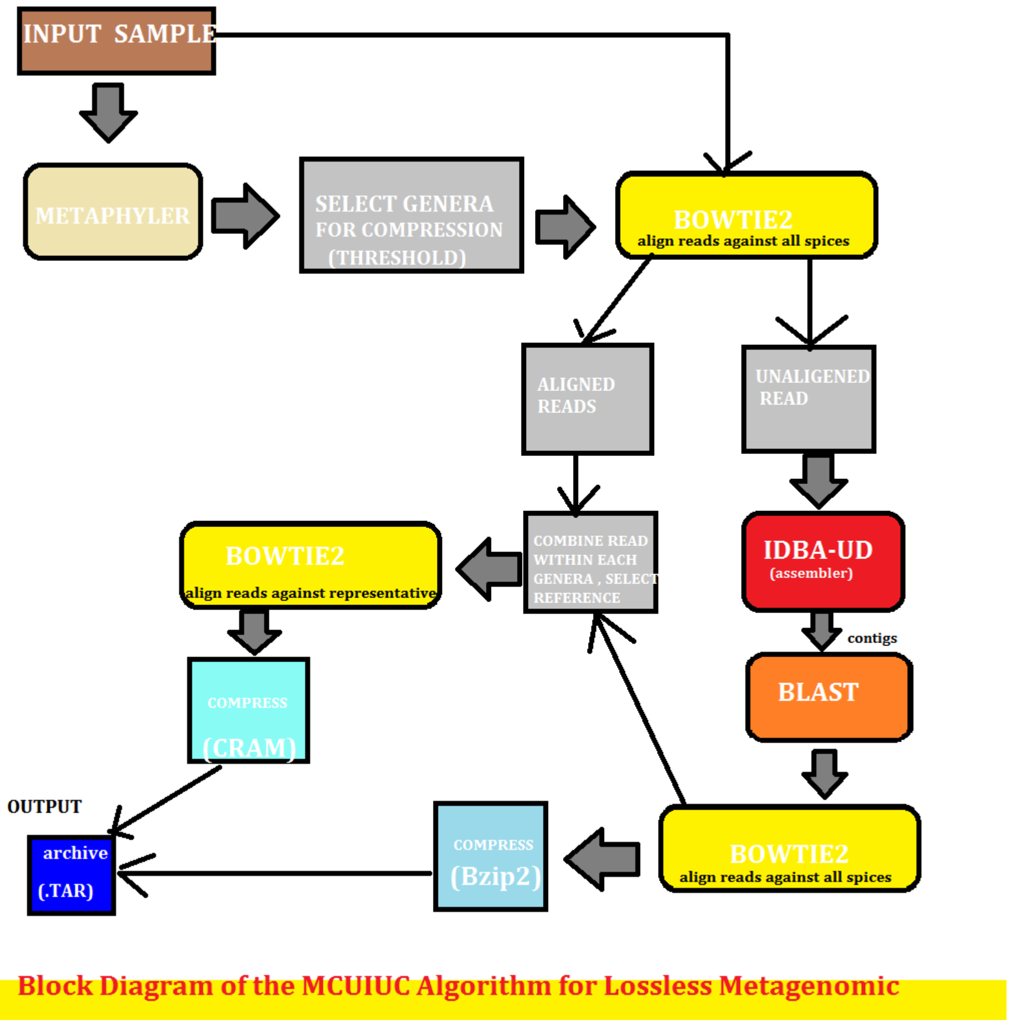

INPUT SAMPLE
METAPHYLER
SELECT GENERA FOR COMPRESSION (THRESHOLD)
BOWTIE2
align reads against all spices
ALIGNED READS
UNALIGENED READ
COMBINE READ WITHIN EACH GENERA , SELECT REFERENCE
IDBA-UD
(assembler)
contigs
BLAST
BOWTIE2
align reads against representative
COMPRESS
(CRAM)
OUTPUT
archive
(.TAR)
COMPRESS
(Bzip2)
BOWTIE2
align reads against all spices
Block Diagram of the MCUIUC Algorithm for Lossless Metagenomic

FIGURE 3.5: Metagenomic Read Compression

3.6 Title 6:Reference Sequence Construction for Relative Compression of Genomes

In Reference Sequence Construction For Relative Compression Of Genomes[9], the author has shown how relative compression of genomes supports fast random access and is an effective method of compressing large DNA sequences that are almost similar. The problem of selecting a good reference is addressed using the dictionary of repeats generated by Comrad, Re-pair and Dna-x algorithms as reference sequences for relative compression. The reference for compressing the input sequence is built from the phrases built by popular dictionary compressors. They concluded that artificially build reference sequences allow superior compression, while keeping the basic advantage of relative compression: fast random access to the collection, also allowing more general repeating data sets to be compressed using relative compression.

3.7 Title 7:HUGO- Hierarchical multi-reference Genome compression for aligned reads

Hierarchical multi-reference Genome compression (HUGO),[5] is a novel compression algorithm for aligned reads in the sorted Sequence Alignment/Map (SAM) format. They have first aligned short reads against a reference genome and stored exactly mapped reads for compression. For the inexact mapped or unmapped reads, they realigned them against different reference genomes using an adaptive scheme by gradually shortening the read length. Considering the base quality value gives both lossy and lossless compression mechanisms. The lossy compression mechanism for the base quality values uses k-means clustering, where a user can adjust the balance between decompression quality and compression rate. The lossless compression can be produced by setting k (the number of clusters) to the number of different quality values. The drawback of this method is that it requires having different reference genomes and prolongs the execution time for additional alignments

3.8 Title 8:Efficient Storage of High Throughput DNA Sequencing Data Using Reference-Based Compression

Efficient storage of high throughput DNA sequencing data using reference-based compression[6] is an effective and tunable method that aligns target sequence with a reference genome and stores the difference between the target and reference genome. Each read is stored by its starting position with respect to the reference, its strand and a tag representing a perfect match with the reference. Rather than storing the absolute position the difference between the two positions is stores as a variable length Golomb code. Each strand and match tags occupy one bit. Mismatches are stored as a list of variations where each variation is represented with its relative position on read that is Golomb encoded, type of variation such as insertion, deletion or substitution and other supplementary information. The proposed method is most efficient when controlled loss of data is allowed and gives large efficiency gains as the length of reads are increased.

3.9 Title 9:BIND – An algorithm for loss-less compression of nucleotide sequence data

In *BIND* by Tungadri et al, a new approach is discussed to for compressing a sequence of nucleotide [8] . They have used a special block length encoding scheme in place of binary data which is a key procedure of this paper. They divide the binary data into two sets and process each one independently and parallel. An optimal unary coding scheme is used to encode the block length. The algorithm also handles lowercase values as well as any other symbol apart from ACGT On comparing

BIND with other general purpose compression algorithms(like lzma,bzip,gzip) better compression was achieved . It is a lossless compression that is freely available online.

CHAPTER 4

PROPOSED APPROCH

4.1 Background

In order to get high compression rates selection of reference is very important factor. Firstly, So far one of the dna sequence is selected as reference from among the set of input sequences using brute force or some heuristic method . An intuitive strategy to find the reference sequence is to compress all the input sequences against all possible reference sequence and select reference that yields maximum number of matches with respect to input sequences. Heuristic strategy is given in FRESCO [17] where instead of compressing a to-be-compressed sequence against all candidate references they compare the referentially compressed input sequence and the referentially compressed reference candidates with respect to one randomly selected base reference. This heuristic only needs to compress each sequence one time with respect to the base reference, independent of the number of candidate references where the candidate references are selected randomly. Secondly, once reference is selected compression is performed by matching input sequence with indexed reference. That is searching for longest prefix of input sequence in indexed referenced until whole input sequence is scanned till the end.

4.1.1 Basic Referential Algorithm

Referentially compressing a string means to encode the string as a concatenation of substrings from a given reference sequence. A number of schemes are used to denote the matches found in a reference sequence. The first scheme is to encode the matches as a set of pairs, where each pair consists of a position and length of a match [3],[11]. Second scheme is to encode parts of a sequence with original text entries instead of matches into the reference [1]. This scheme is only useful if the referential match values are small enough forming a compressed representation of the text occupying less storage. Third scheme is to encode each match value found in the reference as a triplet, where first item denotes starting position of the match, second denotes the length of the match, and third denotes first character appearing after the match. This technique yields very good results in case the target input sequence and reference sequence are almost similar with a difference of single nucleotide.

To compress an input sequence with respect to a reference using the basic referential compression the algorithm finds longest prefix match of the input sequence in reference sequence. The matches are replaced by triplets that contain position, length and following mismatch entry. This procedure is repeated until the whole input sequence is processed from left to right. Algorithm 1 shows the pseudo-code of the above procedure.

Algorithm 1 : Referential compression algorithm

```
input: to-be-compressed sequence, reference sequence and X 1: while input
sequence is not empty do
2: find longest matching substring in reference for current input position
3: if length of match greater than X then
```

```
4: encode match as (match position; length)
5: else
6: encode match with raw symbols
7: end if
8: end while
output: compressed sequence
```

In the above pseudo-code the value of X is used to find out whether the entry is encoded as short matches of reference or as raw strings. The inverse of a referential compression is the decompression of a referential compressed sequence using the same reference in order to get back the original string as it was. The more the similarity between reference and input sequences, the higher the compression rates. For example, if we compress a human genome sequence with a mouse genome we get very low rates of compression.

4.2 Proposal

This paper has two key components for performing DNA compression. Firstly, a reference set is constructed that is a set of common sub-strings of number of DNA sequences that are randomly chosen. Secondly, the sample of target input sequence is compressed by replacing longest matching prefix of input sequence with the fingerprints of that match present in the reference set. Given below is the flowchart for the given approach.

4.2.1 Reference Construction

Suppose we choose k DNA sequences from different species, Reference construction algorithm concatenates these k sequences into one large sequence Z

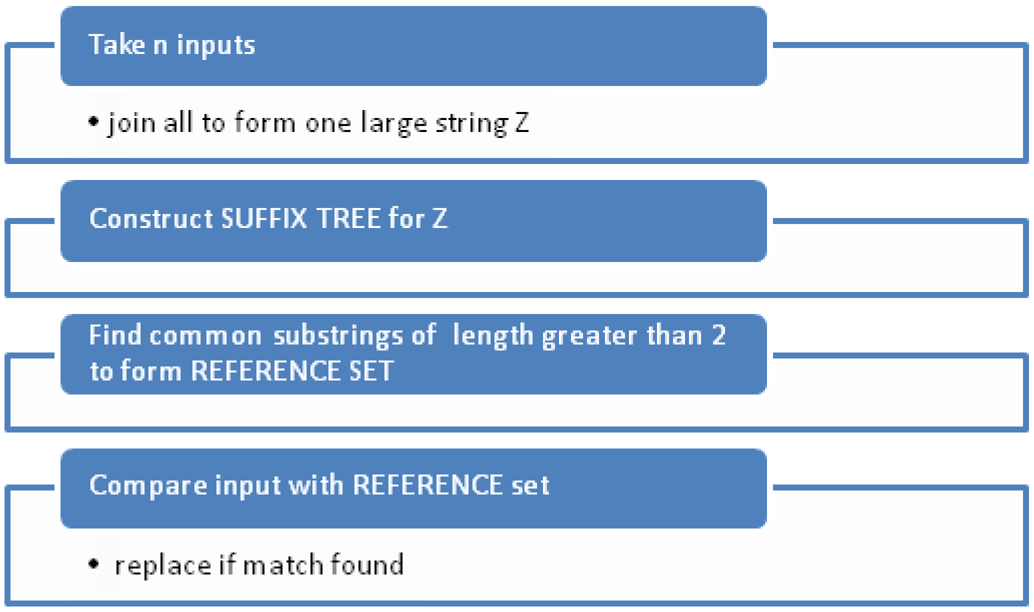

Take n inputs
join all to form one large string Z
Construct SUFFIX TREE for Z
Find common substrings of length greater than 2 to form REFERENCE SET
Compare input with REFERENCE set
replace if match found

FIGURE 4.1: Flowchart for proposed work

where Z=(Z1,Z2,Z3.......Zk). A suffix array is constructed using this sequence Z and longest prefix match of two immediate neighboring suffix is found that belong to two different input sequences(i.e. Zi and Zi+1).

A suffix tree array is constructed using number of DNA sequences of similar or dismilar species. Using this suffix structure we have found all the common sub strings in the input sequences and formed a Reference set of sequences. All the common substrings along with a unique id (fingerprint) are contained in this refderence set.

Common sub strings and suffix tree :Given the set of DNA sequences S =(S1, ..., SK), where length Si=ni and total length of S = N. We find for each

$$2 < k < K \tag{4.1}$$

the longest strings which occur as sub strings of at least k strings. So we use a

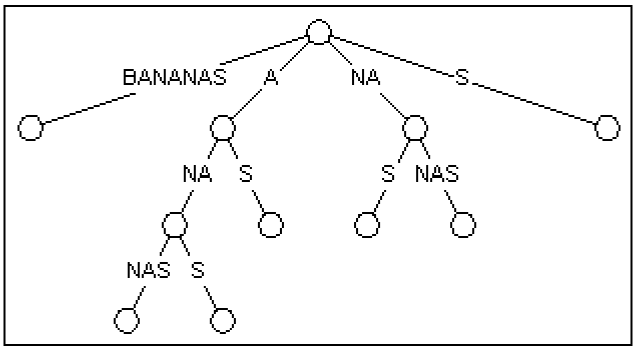


FIGURE 4.2: Example of suffix tree

suffix tree data structure to find the longest sub strings. A suffix tree is a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values. In simple terms a suffix tree is a trie of n strings that are suffixes of an n character string S(where n is a number). Ukkonen's algorithm constructs an implicit suffix tree Ti for each prefix S[l ..i] of S with length x. It first constructs T1 using 1st character, then T2 using 2nd character, then T3 using 3rd character, Tx using xth character. Likewise, the entire suffix tree Ti+1 is constructed on top of implicit suffix tree Ti. The actual suffix tree for S is built from Tx by adding.

Algorithm 2: Ukkonen's algorithm

```
1: Construct tree T1
2: For i from 1 to x-1 do
3: begin (level i+1)
4: For j from 1 to i+1
5: begin (extension j)
6: Find the end of the path from the root marked S[j..i] in referred tree.
7: Extend that path by adding character S[i+l] if not present
8: end;
```

```
9: end;
```

Suffix extension here is adding a new character each time into the suffix tree constructed till then. The extension j of level i+1 discovers the end of S[j..i] which is present in advance because of previous level i and then it extends S[j..i] to be sure the suffix S[j..i+1] is in the tree.
There are 3 extension rules:
Rule 1: If the path from the root marked as S[j..i] is the last character of the leaf edge then the character S[i+1] is simply added to the end of the label on that leaf edge.
Rule 2: If the path from the root marked as S[j..i] ends when there are still more characters to come after S[i] on path and next upcoming character is not s[i+1], then a new leaf edge with label s[i+1] and number j is created starting from character S[i+1]. Also a new internal node is created if s[1..i] ends in between a non-leaf edge.
Rule 3: If the path from the root marked as S[j..i] ends when there are still more characters to come after S[i] on path and next upcoming character is s[i+1] then do nothing.
Finally these common sequences (or longest prefix matches) are arranged in increasing order of length greater than two and stored as a Reference Set.
Algorithm 3 shows the pseudo-code of the above procedure.

Algorithm 3: Reference construction algorithm

```
input: k random sequences 1: merge k input strings into one single string
2: construct the suffix array
```

3: locate the longest common prefix match of two immediate neighboring suffix that belong to two different input strings
4: Sort in increasing orders of length all the common strings with length greater than 2.
output: sequence of reference blocks

Building the suffix tree takes O(N) time when the size of the alphabet is constant and O(NK) time is taken by k common sub string problem. If we have a long reference sequences that we need to match with input sequence, we trim down our reference to a much smaller fingerprint which is easier and faster to match with input sequence. However this approach is only useful if the fingerprints of different blocks of reference are expected to be dissimilar.

4.2.2 Reference Compression

We suppose that our reference set is represented as r=(r1,r2,r3,...rn) and input sequence as s=(s1,s2,s3,...sn). Our aim is to find out if ri=si without matching all the n values where n is the length of the sequences. For matching this, we break up the reference sequence into blocks of common sequences that was found using Algorithm 2 and swap the longest prefix match of the input sequence with its fingerprint. In Algorithm 4 an input sequence s is compressed with respect to reference set. Firstly, we select first m characters of input string (where m is the length of the sliding window) and check whether this block has a match in the reference set. If a match of this selected block is present, then we replace that whole block with a smaller fingerprint of that block. Otherwise, we split the sliding window size m into half and check again for a match of this block in the reference set. This procedure of dividing the window size into half each time is repeated until a match is found. If no match is found and m is reduced to 1 then the character in the block is written as it is and sliding window is moved to next m characters. Similarly, the entire input sequence is scanned from left to right and compressed using the given algorithm. Here m is the length of the longest common string present in the reference set. Given below are the steps for compressing a target sequence using proposed reference based compression.

Algorithm 4: Proposed referential compression algorithm

input:to-be-compressed input sequence, reference set, m that is the length of longest sub string in reference set.

```
1: take first m characters of input and check if it is present in the reference set.
2: if a match is found swap the match with its fingerprint and move the sliding
window to next m substrings.
3: if not then repetitively decrease the window size by half (m/2) and search for
a match until no match is found.
4: if m is reduced to 1 with no match found then the character in the block is
written as it is and sliding window is moved to next m characters.
5: repeat the above process until end of the input sequence.
```

output: compressed input sequence

Searching for a match in reference set takes constant time as we have stored the reference set as a hash table.

4.2.3 Decompression algorithm

Decompression algorithm is very simple and fast in which the compressed sequence is read from the file. The fingerprints are mapped to their corresponding values and a single character is written as it is. Complexity of decompression algorithm depends on the length of compressed sequence, giving O(n) time complexity where n is the length of compressed sequence.

Algorithm 5: Proposed decompression algorithm

```
input: compressed sequence of length n .

1: Read the compressed input sequence from the file.
2: If the current character read is a fingerprint then map the fingerprint with is
corresponding value
3: Reapeat this procedure until the whole sequence is read .
4: write the obtained results back to the file

output: Uncompressed sequence
```

CHAPTER 5

IMPLEMENTATION AND RESULTS

5.1 Evaluation

The platform used for the implementation of the algorithms is JAVA using Netbeans 8.0.2v . All experiments have been run on a Lenovo E1922s with Intel Core i5-4460, 4-GB installed RAM and Windows 8(64 bit operating system.

All sizes of files are measured in bytes where 1kb equals to 1,000 bytes. Compression ratio , compression speed and savings are the performance parameters used. Here, compression ratio represents the decrease in the size of original file after performing compression; compression speed is the rate at which the original file is compressed and memory saving is defined as decrease in size relative to original file. Two different Biological data sets are taken for evaluation of the proposed algorithm. Both these data sets are adopted from ncbi ftp in fasta format.

In the first place, we have taken a DNA sequence of *Arabidopsis Thaliana* (a plant specie) containing five chromosomes, to show the performance of our algorithm on three different performance parameters namely compression ratio, speed and memory saving. Results show that algorithm in this paper achieves commendable compression. Chromosome 1 of approximately 8 and storage space upto 87 percent can be saved.

Then we have taken ten different data sets of *Bacteria* to compare the performance of our proposed algorithm with an existing method FRESCO [17]. Besides the basic referential compression algorithm, FRESCO has proposed three variants, that are :1)good reference selection;2)reference re-writing;3)second order compression, which result in decreased compression size, achieving the ratios 12 percent ;35 percent and 75 percent, respectively. This paper does not depict a comparison with the basic, and undertakes only a comparison with the basic proposed referential compression algorithm of FRESCO. Out of 10 different data sets, we have chosen *Campylobacter coli 76339* as the reference sequence and compressed rest of the 9 data sets with respect to it. *Campylobacter coli 15-537360(C1)* and *Campylobacter Coli CVM(C2)* are DNA sequence of same specie for which FRESCO achieves 1.421 and 1.89 compression ratio saving upto 29.63 and 47.1 storage space respectively. Our proposal gives better results with compression ratio 9 and 8.207 and storage space is saved upto 88.88 and 87.81 respectively. Remaining five DNA sequences,namely *Campylobacter jenjuni 002426(C3), Campylobacter jenjuni 002538 (C4) ,Campylobacter jenjuni 002425(C5), Campylobacter jenjuni RM1221(C6), Campylobacter jenjuni NCTC 11168 (C7),* belong to a different specie with respect to the reference. Fresco gives compression ratio of 1.44, 1.452, 1.466 , 1.459 and 1.461 respectively. This time, again our approach gives better results 8.247, 8.245, 8.227, 8.229 and 8.212 respectively for inter species sequences than the existing. Finally the last 2 DNA sequence belong to different genus that are *Escherichia coli strain k12(E1)* and

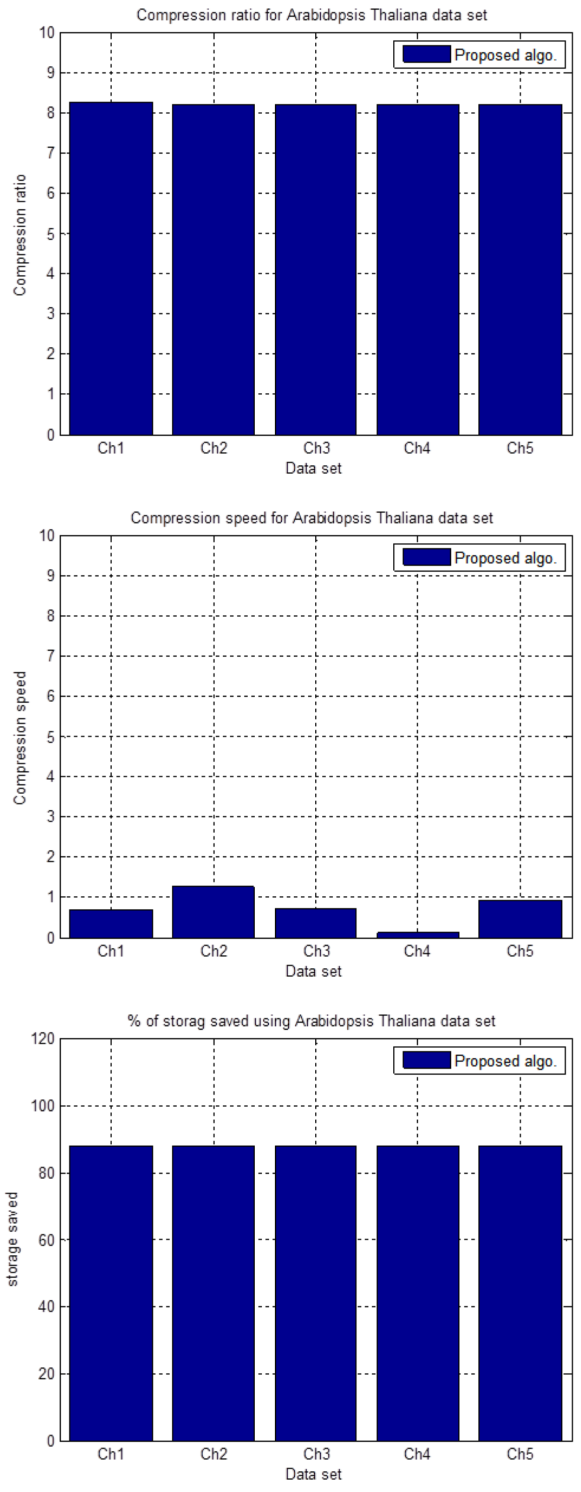
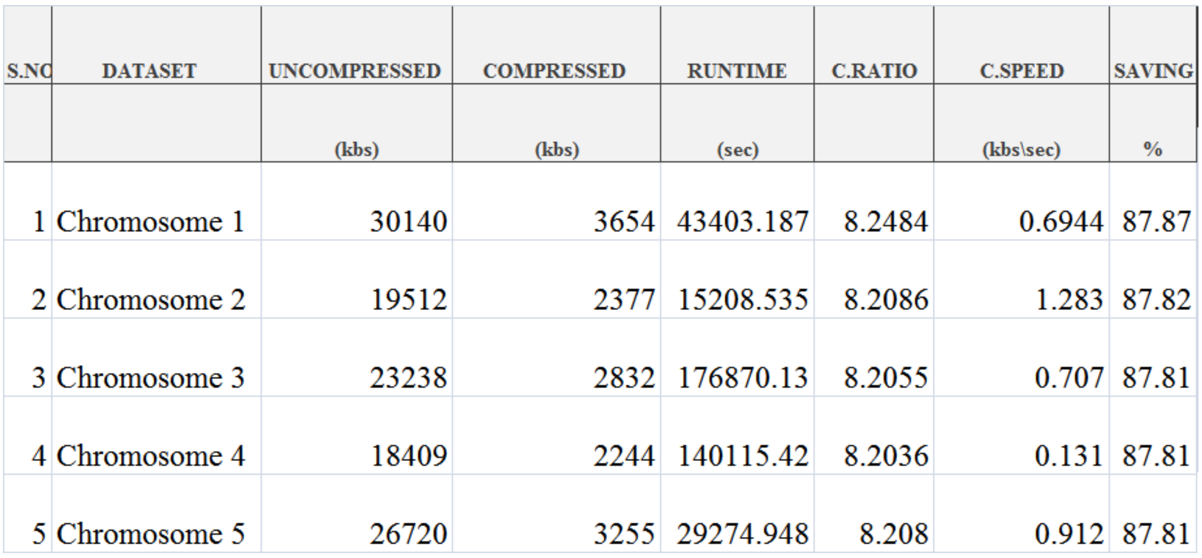
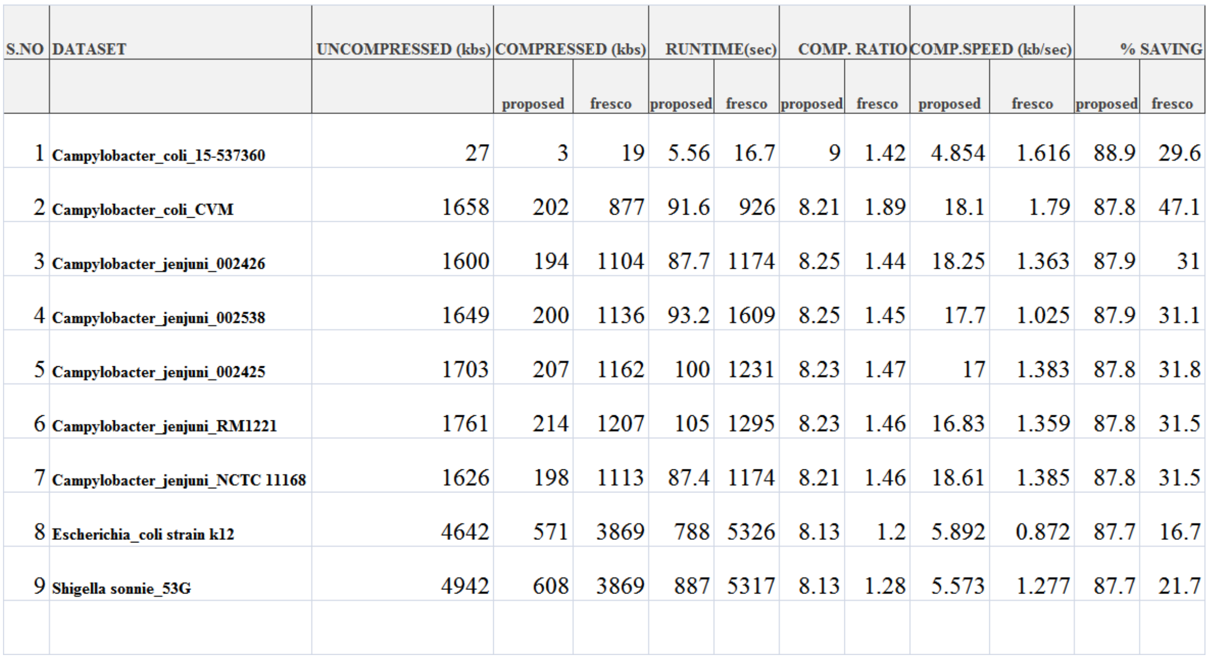


FIGURE 5.1: Shows the results for proposed algorithm in terms of (1)compression ratio ,(2) compression speed and (3) storage saving using chromosomes of *Arabidopsis Thaliana* dataset



S.NO	DATASET	UNCOMPRESSED	COMPRESSED	RUNTIME	C.RATIO	C.SPEED	SAVING
		(kbs)	(kbs)	(sec)		(kbs\sec)	%
1	Chromosome 1	30140	3654	43403.187	8.2484	0.6944	87.87
2	Chromosome 2	19512	2377	15208.535	8.2086	1.283	87.82
3	Chromosome 3	23238	2832	176870.13	8.2055	0.707	87.81
4	Chromosome 4	18409	2244	140115.42	8.2036	0.131	87.81
5	Chromosome 5	26720	3255	29274.948	8.208	0.912	87.81

FIGURE 5.2: Results obtain after applying proposed approach on five chromosomes of *Arabidopsis Thaliana* dataset



S.NO	DATASET	UNCOMPRESSED (kbs)	COMPRESSED (kbs)		RUNTIME(sec)		COMP. RATIO		COMP.SPEED (kb/sec)		% SAVING	
			proposed	fresco	proposed	fresco	proposed	fresco	proposed	fresco	proposed	fresco
1	Campylobacter_coli_15-537360	27	3	19	5.56	16.7	9	1.42	4.854	1.616	88.9	29.6
2	Campylobacter_coli_CVM	1658	202	877	91.6	926	8.21	1.89	18.1	1.79	87.8	47.1
3	Campylobacter_jenjuni_002426	1600	194	1104	87.7	1174	8.25	1.44	18.25	1.363	87.9	31
4	Campylobacter_jenjuni_002538	1649	200	1136	93.2	1609	8.25	1.45	17.7	1.025	87.9	31.1
5	Campylobacter_jenjuni_002425	1703	207	1162	100	1231	8.23	1.47	17	1.383	87.8	31.8
6	Campylobacter_jenjuni_RM1221	1761	214	1207	105	1295	8.23	1.46	16.83	1.359	87.8	31.5
7	Campylobacter_jenjuni_NCTC 11168	1626	198	1113	87.4	1174	8.21	1.46	18.61	1.385	87.8	31.5
8	Escherichia_coli strain k12	4642	571	3869	788	5326	8.13	1.2	5.892	0.872	87.7	16.7
9	Shigella sonnie_53G	4942	608	3869	887	5317	8.13	1.28	5.573	1.277	87.7	21.7

FIGURE 5.3: Results obtained on comparing FRESCO and Proposed algorithm (1)compression ratio ,(2) compression speed and (3) storage saving using different species of *Bacteria*

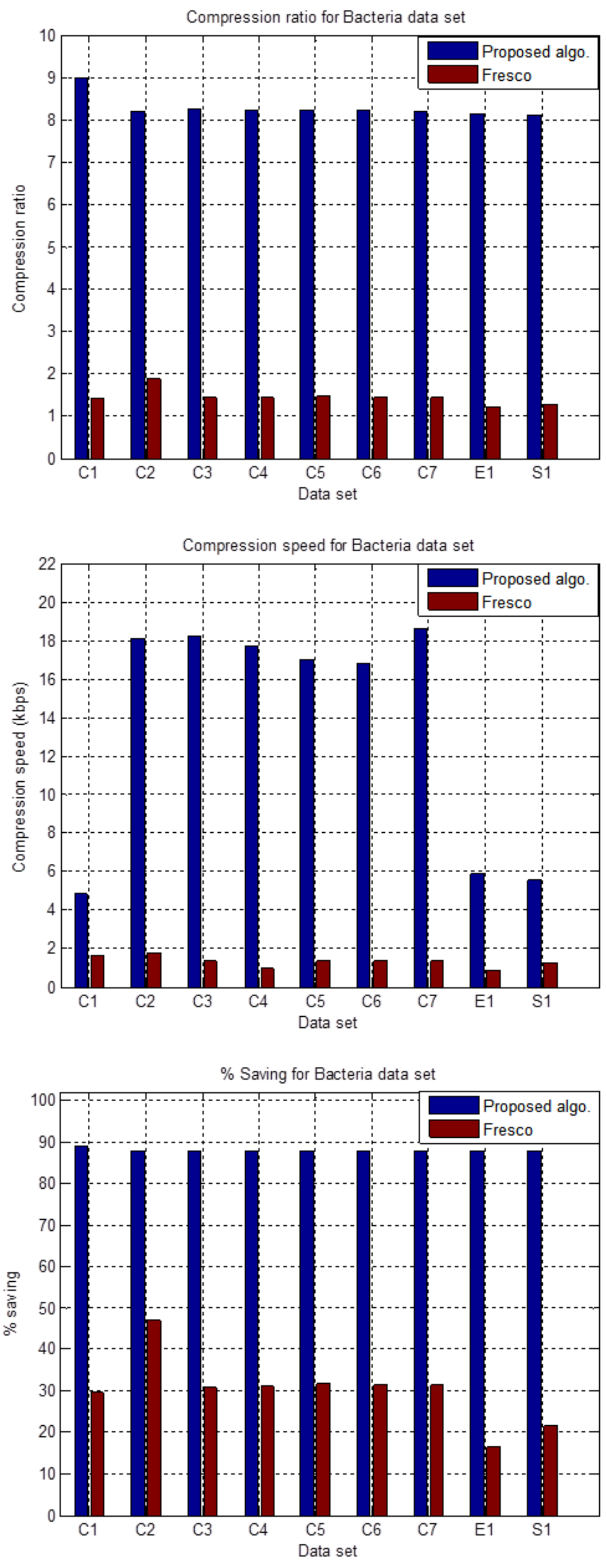


FIGURE 5.4: Comparison between FRESCO and Proposed algorithm (1)compression ratio ,(2) compression speed and (3) storage saving using different species of *Bacteria*

Shigella sonnie 53G (S1)where again our proposed approach achieves compression ratio of 8.129 and 8.128 and saving upto 87.69 percent in both the cases. Fig shows the comparison bewteen two methods in terms of compression ratio and percentage storage saved.

Comparing the compression speed of the two methods we found that *Campylobacter coli 15-537360* and *Campylobacter Coli CVM* takes 4.854 kbpsec and 18.096 kbpsec respectively, while FRESCO gives 1.616 kbpssec and sec 1.79kbpsec respectively(for same species).With different species C3 TO C7 our method gives 18.246 kbpsec, 17.699 kbpesec, 16.999 kbpsec, 16.826 kbpsec and 18.612 kbpsec for the five inter-species data sets where as FRESCO takes 1.363 kbpsec, 1.025 kbpsec,1.383 kbpsec,1.359 kbpsec and 1.385 kbpsec, respectively. Taking genus level data set E1 and S1 our method takes 5.892 kbpsec and 5.573kbpsec while FRESCO gives 0.872 kbpsec and 1.277kbpsec respectively. Fig shows the comparison between two methods in terms of compression speed.

Main starting page

5.2 Code

```
//Javastart.java

package javaapplication1;

/**
 *
 * @author user
 */
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Date;
import java.util.Scanner;

public class Javastart {

public static void main(String[] args) throws
FileNotFoundException, IOException, Exception
 {
File text1 = new File ("C:/Users/user/Documents
/NetBeansProjects/JavaApplication11/virus/hepatitis_gb.txt");

Scanner scnr1 = new Scanner(text1);

       while(scnr1.hasNext())
       {
           str1 = str1+ scnr1.next();
       }
        str1 = str1.substring(0, str1.length());

        // reference construction

SuffixArray c = new SuffixArray();

bf2 b = new bf2();
String str3 = "C:/Users/user/Documents/NetBeansProjects
/JavaApplication11/refscopy.txt";

long lStartTime = System.currentTimeMillis();

b.has(str1, str3);

long lEndTime = System.currentTimeMillis();

long difference = lEndTime - lStartTime;

System.out.println("Elapsed milliseconds: " + difference);

    }

}
```

Reference construction using suffix array

```
//SuffixArray.java

package javaapplication1;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Scanner;
import java.util.Set;

public class SuffixArray
 {
private Suffix[] suffixes;

public SuffixArray(String text)

 {
int N = text.length();

this.suffixes = new Suffix[N];

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

suffixes[i] = new Suffix(text, i);

 Arrays.sort(suffixes);

 }



private static class Suffix implements Comparable<Suffix>
 {
        private final String text;
        private final int index;

        private Suffix(String text, int index)
        {
            this.text = text;

            this.index = index;
        }
        private int length()
         {
            return text.length() - index;
         }
        private char charAt(int i)
         {
            return text.charAt(index + i);
         }

        public int compareTo(Suffix that)
         {
            if (this == that) return 0;  // optimization

            int N = Math.min(this.length(), that.length());

            for (int i = 0; i < N; i++)
            {
                if (this.charAt(i) < that.charAt(i)) return -1;

                if (this.charAt(i) > that.charAt(i)) return +1;
          }


            return this.length() - that.length();
        }

        public String toString()
        {

            return text.substring(index);

        }
    }

    /**
     * Returns the length of the input string.
     * @return the length of the input string
     */
    public int length()

     {

        return suffixes.length;
      }



    public int index(int i)

    {
        if (i < 0 || i >= suffixes.length) throw new
        IndexOutOfBoundsException();

        return suffixes[i].index;
    }




      public String lcp1(int i, String vv)

    {
        if (i < 1 || i >= suffixes.length) throw new
        IndexOutOfBoundsException();

        return lcp(suffixes[i], suffixes[i-1], vv);

    }
      private static String lcp(Suffix s, Suffix t, String vvv) {

        String vv="";

        int N = Math.min(s.length(), t.length());

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

        {
            if (s.charAt(i) == t.charAt(i) && s.charAt( i) != '@')

            {

                vv=vv+s.charAt(i);
            }
        if(s.charAt(i) != t.charAt(i))

        {

            break;

         }
   }
   return vv;

    }

    public String select(int i)
    {
        if (i < 0 || i >= suffixes.length) throw new
        IndexOutOfBoundsException();

         return suffixes[i].toString();
    }

    public static String concateLines(String[] s, String separator)
    {

        StringBuilder sb = new StringBuilder();

        if (s.length > 0)

         {
            sb.append(s[0]);

            for (int i = 1; i < s.length; i++)

            {
                sb.append(separator);

                sb.append(s[i]);
            }
        }

        return sb.toString();
    }

    public int rank(String query)
     {
        int lo = 0, hi = suffixes.length - 1;

        while (lo <= hi)
         {
            int mid = lo + (hi - lo) / 2;

            int cmp = compare(query, suffixes[mid]);

            if (cmp < 0) hi = mid - 1;

            else if (cmp > 0) lo = mid + 1;

            else return mid;
        }

        return lo;
    }

    // compare query string to suffix
    private static int compare(String query, Suffix suffix)
     {
        int N = Math.min(query.length(), suffix.length());

        for (int i = 0; i < N; i++)
        {
            if (query.charAt(i) < suffix.charAt(i)) return -1;

            if (query.charAt(i) > suffix.charAt(i)) return +1;
        }

        return query.length() - suffix.length();
    }



 SuffixArray() throws FileNotFoundException, IOException  {
  // long lStartTime = System.currentTimeMillis();

  System.out.println("input sequences are :");

        String str2="o";int len=0;

        File text1 = new File("C:/Users/user/Documents
        /NetBeansProjects/JavaApplication11/virus/
        hepatitis_e.txt");
        Scanner scnr1 = new Scanner(text1);

        while(scnr1.hasNext())

        {
            str2 = str2+ scnr1.next();
        }
      str2 = str2.substring(1, str2.length());

       String str22="o";

        File text11 = new File("C:/Users/user/Documents
        /NetBeansProjects/JavaApplication11/virus/
        hepatitis_c_3.txt");

        Scanner scnr11 = new Scanner(text11);

        while(scnr11.hasNext())

        {
            str22 = str22+ scnr11.next();
        }

str22 = str22.substring(1, str22.length());

String str222="o";

        File text111 = new File("C:/Users/user/Documents/
        NetBeansProjects/JavaApplication11/virus/hepatitis_gb.txt");

        Scanner scnr111 = new Scanner(text111);

   while(scnr111.hasNext())

        {
            str222 = str222+ scnr111.next();
        }

  str222 = str222.substring(1, str222.length());

  String str21="o";

        File text12 = new File("C:/Users/user/Documents
        /NetBeansProjects/JavaApplication11/virus
        /hepatitis_b.txt");

        Scanner scnr12 = new Scanner(text12);

        while(scnr12.hasNext())
        {
            str21 = str21+ scnr12.next();
        }

      str21 = str21.substring(1, str21.length());

      str2=str2+"@";

      String[] input = {str2,str22,str222,};

      String s=SuffixArray.concateLines(input, "@");

      String sa="";

      SuffixArray suffix = new SuffixArray(s);

      HashSet ht=new HashSet();

      System.out.println("length"+s.length());

        for (int i = 0; i < s.length(); i++)

        {
 int index = suffix.index(i);

String ith = "\"" + s.substring(index,

Math.min(index + 50, s.length())) + "\"";

assert s.substring(index).equals(suffix.select(i));

int rank = suffix.rank(s.substring(index));

 String v= "";

  if (i == 0)
   {
// System.out.printf("%3d %3d %3s %3d %s\n", i, index, "-", rank, ith);
            }
            else {
                String lcp1=suffix.lcp1(i,v);



                if(lcp1.length()>=2 )
                 {
                    ht.add(lcp1);

                    sa="reference";}
                else
                {
                sa=" not a reference";
                }


        }
            }
 int l=0;

 for (Object value : ht)
  {

  String g=(String)value;

 l=g.length();

 if(len<l)len=l;

 }
HashMap<Integer, String> hm = new HashMap<Integer, String>();

File f11 = new File("C:/Users/user/Documents/
NetBeansProjects/JavaApplication11/refscopy.txt");
FileWriter f12 = new FileWriter(f11,false);

File f111 = new File("C:/Users/user/Documents
/NetBeansProjects/JavaApplication11/length.txt");
FileWriter f122 = new FileWriter(f111,false);

f122.write(len+"\r\n");

int iii=0;

for (Object value : ht)
```

```
 {
hm.put(iii, (String) value);

iii++;

}
Set set = hm.entrySet();

Iterator it = set.iterator();

while (it.hasNext())

{
Map.Entry m = (Map.Entry) it.next();

String aaaa=(String)m.getValue();

f12.write(aaaa+ "\r\n");

System.lineSeparator();

f12.flush();

}
 f122.close();

 f12.close();

   }

   }
```

compression algorithm

```
\\bf2.java

package javaapplication1;

import static com.sun.org.apache.xalan.internal.
lib.ExsltDynamic.map;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.ObjectOutputStream;
import static java.lang.System.in;
import java.security.NoSuchAlgorithmException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Scanner;
import java.util.Set;
import static oracle.jrockit.jfr.events.Bits.intValue;

/**
 *
 * @author user
 */
public class bf2
 {
     HashMap< String,Integer> hm = new HashMap<String,Integer>();

     private static int ny;

     public int fun2(String copystr2, String k ,int n) throws Exception
 {
String l=copystr2.substring(0, n+1);

 boolean a=  hm.containsKey(l);

if(a==true)
```

```
 {
n=n+1;

fun2(copystr2,l,n);

return n;

  }
 else {
             n=n;
     }

         return n;
    }
public  int fun ( String copystr2, String k ,int n)
 throws Exception{
 ny=(n/2);

    String kp=k.substring(0,ny);

    boolean a= hm.containsKey(kp);

    if (a==true)
        {
               String l=copystr2.substring(0, ny+1);

               boolean aa=  hm.containsKey(l);

                if(aa==true )
                   {
                       if(hm.containsKey(l))
                       {
                       ny= fun2(copystr2,l,ny+1);



                       }
                       else
                       {
                          return ny;
                       }

                    }
                else
                 {
                  return ny;

                }



            // int iop=qq.indexOf(k);
            // f.write(iop);System.out.println(iop);

         }
        else {

              fun(copystr2,kp,ny);
             }

         return ny;


    }

    public void has(String str1,String str3) throws
    FileNotFoundException, IOException, NoSuchAlgorithmException, Exception
    {
    ArrayList<String> qq = new ArrayList<String>();

    String copystr2=str1;

    File text1 = new File(str3);

    String str11="";

    int nn;

    File f1 = new File("C:/Users/user/Documents/
    NetBeansProjects/JavaApplication11/10output.txt");

    FileWriter f = new FileWriter(f1,false);

    BufferedReader br = new BufferedReader(new
    FileReader("C:/Users/user/Documents/NetBeansProjects
    /JavaApplication11/length.txt"));

    Scanner scnr1 = new Scanner(text1);

    String line=br.readLine();
    nn=Integer.parseInt(line);

    br.close();

    int count=0;

     while(scnr1.hasNext())

       {
       str11 = scnr1.next();

       hm.put(str11,count );

       count++;
       }
      int m1=nn;

        while(copystr2.length()>=m1)
        {

int n= m1;

///System.out.println("ooo"+n);

 String k= copystr2.substring(0,n);

         while(k.length()>1 && n>=2 )
            {
            boolean a= hm.containsKey(k);

           // System.out.println(a);

            if (a==true)
               {

                Integer iop=hm.get(k);

               f.write(iop);

                copystr2 = copystr2.substring(m1, copystr2.length());break;
               }

             else
             {
            int o= fun(copystr2,k,n);
```

```
              if(o==1)
               {
                f.write(copystr2.charAt(0));//f.write("\n");

                copystr2 = copystr2.substring(1, copystr2.length()); break; //break;}
               }
             else
                {   k=k.substring(0, o);

                    Integer iop=hm.get(k);

                    f.write(iop);

                    //System.lineSeparator();

                    //f.write("\n");

                  // System.out.println("mmm for greater"+iop);

                    copystr2 = copystr2.substring(o, copystr2.length()); break;

                }


            }


           }


        }
     while(copystr2.length()<m1 && copystr2.length()>1)
       {


        int n=copystr2.length();
        String k= copystr2.substring(0, n);

        if(k.length()==1)
        {
f.write(k);
}
        boolean a=  hm.containsKey(k);
       if (a==true)
            {
              Integer iop=hm.get(k);
              f.write(iop);//System.lineSeparator();
             // f.write("\n");
             // System.out.println("k is"+iop);
               break;

            }
            else
            {
                if(n==3)
                {

                     n=n+1;

                }
               int o= fun(copystr2,k,n);

               if(o==1)
               {
                 f.write(copystr2.charAt(0));
                // f.write("\n");
                 //System.lineSeparator(); //
               //  System.out.println("here"+copystr2.charAt(0));
                 copystr2 = copystr2.substring(1, copystr2.length()); break;
               }
               else
                 {
                    k=k.substring(0, o);
                    Integer iop=hm.get(k);
                   f.write(iop);
                //   f.write("\n");

                   //System.lineSeparator(); System.out.println("mmm---"+iop);
                    copystr2 = copystr2.substring(o, copystr2.length()); break;
                  }


            }

       }
     if(copystr2.length()==1){f.write(copystr2);

    // System.out.println(copystr2);
     }
```

```
        f.close();




    }
}
```

decompression algorithm

```
//decompression.java

package javaapplication1;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;



/**
 *
 * @author user
 */
public class decompression {

     public static void main(String[] args)
     throws IOException
      {
           HashMap< String,Integer> hm =  new HashMap<>();
               String strKey = null;

File f2 = new File("C:/Users/user/Documents/
NetBeansProjects/JavaApplication11/9output.txt");

    FileWriter f3 = new FileWriter(f2,false);

    File text1 = new File("C:/Users/user/Documents
    /NetBeansProjects/JavaApplication11/refscopy.txt");

    String str11=""; int count=0; Scanner scnr1 = new Scanner(text1);

    ArrayList<Integer> in = new ArrayList<Integer>();

     FileReader fr=new FileReader("C:/Users/user/Documents/NetBeansProjects/JavaApplication11/10

     BufferedReader br1=new BufferedReader(fr);
     int aa;

   long lStartTime = System.currentTimeMillis();

   while(scnr1.hasNext())

       {
        str11 = scnr1.next();

       hm.put(str11,count );

       count++;
       }

      while( (aa=br1.read())!= -1)
     {

     in.add(aa);

     }
     fr.close();

      for(int l=0;l<in.size();l++)
      {
          int s=in.get(l);



           for(Map.Entry entry: hm.entrySet())

          {
            if(in.get(l).equals(entry.getValue()))
            {
                strKey = (String) entry.getKey();

                break;
                 //breaking because its one to one map
            }

        }



          f3.write(strKey);

      }
  long lEndTime = System.currentTimeMillis();
 long difference = lEndTime - lStartTime;

 System.out.println("Elapsed milliseconds: " + difference);

    fr.close(); f3.close();
    }

}
```

CHAPTER 6

CONCLUSION AND FUTURE DIRECTIONS

6.1 Conclusion

Growth of high throughput DNA sequencing has resulted in voluminous amount of of Biological data. Managing this huge data, transferring it over the network and storing it for future purpose becomes very difficult.This work deal with such problems by presenting a new approach of compressing these Biological data.In this work it has been shown that the new approach of reference construction and reference based compression is very efficient and fast to compress DNA data sets.In the proposal, the constructed reference is a multi-species reference sequence that also works with inputs belonging to different species. Reference sequence that is constructed using all the common sequences is used to compress DNA sequences by matching the fingerprints of reference over the input sequence.Unlike FRESCO, the proposed algorithm gives better compression ratio for different spice DNA sequence as well as for same spice DNA sequence The given approach also works at genus level giving a good lossless compression unlike the existing methods proposed so far.

6.2 Future work

Selection of the reference plays a vital part in compressing down a sequence.Higher compression rates can be obtained using a good reference. So, the future work would include constructing a re-writable reference that can further improve compression ratio for a same set of data. Another aspect of the research would be to suggesting a mathematical model for the optimum size of the reference sequence.

References

JOURNALS

1. Sebastian Wandelt and Ulf Leser,"Adaptive efficient compression of genomes", Algorithms for Molecular Biology , Published:12 November 2012, Publisher BioMed Central.

2. Armando J. Pinho, Diogo Pratas and Sara P. Garcia ,"GReEn: a tool for effcient compression of genome resequencing data", Portugal, Nucleic Acids Research, 2012 Feb, Vol. 40, Issue 4,pp e27

3. S. Deorowicz and S. Grabowski, "Robust Relative Compression of Genomes with Random Access", in Oxford Journals Science and Mathematics Bioinformatics Volume 25, Issue 2Pp. 274-275.

4. Nour S. Bakr, Amr A. Sharawi,"DNA lossless compression algorithms : Review", in American Journal of Bioinformatics Research 11,2013; Vol. 3(No. 3):pp. 72-81.

5.Pinghao Li , Xiaoqian Jiang , Shuang Wang , Jihoon Kim , Hongkai Xiong , Lucila Ohno-Machado "HUGO: Hierarchical mUlti-reference Genome cOmpression for aligned reads",in Journal of the American Medical Informatics AssociationVolume 21, Issue 2Pp. 363 - 373

6. M.H. Fritz, R. Leinonen, G. Cochrane, and E. Birney,'Efficent Storage of High Throughput DNA Sequencing Data Using Reference-Based

Compression',Genome Research, vol. 21, no. 5,pp. 734-740, May 2011, United Kingdom September 5, 2014 - Published by Cold Spring Harbor Laboratory Press GENOME RESEARCH

7. M.C. Schatz, B. Langmead and S.L. Salzberg,'Cloud Computing and the DNA Data Race', Nature Biotechnology, vol. 28, no. 7, pp. 691-693, July-2010.

8. Tungadri Bose, Monzoorul Haque Mohammed, Anirban Dutta And Sharmila S Mande,'BIND – An algorithm for loss-less compression of nucleotide sequence data', in Journal of Biosciences37(4), September 2012, 785–789, * Indian Academy of Sciences

BOOKS

9. Kuruppu, S., Puglisiz, S. J. and Zobely, J.,'Reference Sequence Construction for Relative Compression of Genomes', in String Processing and Information Retrieval. Springer Berlin Heidelberg (18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings), p. pp 420 425.

CONFERENCE

10. Sebastian Wandelt and Ulf Leser, 'String Searching in Referentially Compressed Genomes', Proc. Int'l Joint Conf. on Knowledge Discovery , Knowledge Eng. and Knowledge Management, 2012.

11. Jonathan G. Ligo, Minji Kim Amin Emad, Olgica Milenkovic and Venugopal V. Veeravalli, 'MCUIUC — A new framework for metagenomic read compression', in Information Theory Workshop (ITW), 9-13 Sept 2013, Sevilla,

IEEE pp 1 - 5.

12. S. Kuruppu, S.J. Puglisi, and J. Zobel,"Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval", in Proc. 17th Int'l Conf. String Processing and Information Retrieval (SPIRE '10), pp. 201-206, 2010.

13. A.Gupta, K.K Dubey,'An efficient compressor for biological sequences',in Advance Computing Conference (IACC), 2013 IEEE 3rd International, 22-23 Feb. 2013, Ghaziabad , India, Publisher: IEEE pp 690 - 695

14. Ms. Priyanka , Dr. Savita Goel, 'A compression algorithm for DNA that uses ASCII values', in Advance Computing Conference (IACC), 21-22 Feb. 2014 IEEE International, Gurgaon, pp 739 - 743

15. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel,'Iterative Dictionary Construction for Compression of Large DNA Data Sets', in Computational Biology and Bioinformatics, IEEE-ACM Transactions on (Volume:9 ,Issue: 1) Feb. 2012, pp 137 - 149

16. M. Cohn and R. Khazan,'Parsing with Prefix and Suffix Dictionaries', in Proc. Data Compression Conf., pp. 180-189, 1996

17. Sebastian Wandelt and Ulf Leser, "FRESCO: Referential Compression of Highly Similar Sequences", IEEE /ACM TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS, VOL. 10, NO. 5, SEPTEMBER/OCTOBER 2013

ARTICLE

18. L.D. Stein,'The Case for Cloud Computing in Genome Informatics', Genome Biology, vol. 11, no. 5, article 207, May 2010.