M.Sc. Engineering Thesis

AN ADVANCED DNA SEQUENCE COMPRESSION ALGORITHM

by
Sadia Sharmin
Student No. 100605052P

Supervisor
Dr. Mohammad Kaykobad
Professor
Department of CSE, BUET

Submitted to
Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka-1000

December, 2009

The thesis titled "An Advanced DNA Sequence Compression Algorithm", submitted by Sadia Sharmin, Roll No. 100605052P, Session October 2006, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on December 3, 2009.

Board of Examiners

1. [illegible]
Dr. Mohammad Kaykobad
Professor
Department of CSE
BUET, Dhaka 1000.

Chairman
(Supervisor)

2. M. M. Islam
Dr. Md. Monirul Islam
Professor & Head
Department of CSE
BUET, Dhaka 1000.

Member
(Ex-officio)

3. [illegible]
Dr. ~~Md.~~ Mostofa Akbar
Associate Professor
Department of CSE
BUET, Dhaka 1000.

Member

4. [illegible]
Dr. Mahmuda Naznin
Assistant Professor
Department of CSE
BUET, Dhaka 1000.

Member

5. [illegible]
Dr. Muhammed Zafar Iqbal
Professor & Head
Department of CSE
Shahjalal University of Science and Technology, Sylhet.

Member
(External)

Candidate's Declaration

It is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

Sadia Sharmin

Sadia Sharmin
Candidate

Content

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thank my supervisor Prof. Dr. Mohammad Kaykobad for introducing me in the beautiful field of bioinformatics algorithms long before this thesis had begun. It is his supervision that has laid down the ground for me to build a research career in the academia. I acknowledge to him for teaching me how to carry on a research work, for providing me with numerous resources and giving useful directions in preparing my presentations. I express my deep gratitude for his patience in reading my numerous inferior drafts, constant supervision, valuable advice and continual encouragement, without which this thesis would have not been possible.

My special thanks goes to all other members of our department; in particular, to Dr. Md. Sohel Rahman, Assistant Professor for his helpful counsel. My parents, my sisters, my son and for sure my husband have given their best support to me throughout my work, i convey my deep reverence to them also.

Finally, every honor and every victory on earth is due to Allah, descended from him and must be ascribed to him. He has endowed me with good health and with the capability to complete this work. I convey my utmost praise to him for letting me the opportunity to submit this thesis.

Abstract

In this thesis we propose an advanced DNA sequence compression algorithm, which is efficient and lossless. None of the available general-purpose compression algorithm compresses these sequences. This is due to the specificity of the genetic sequences. It is believed that DNA sequence encodes life in a nonrandom way. Some important sections of the sequences are shifted, reserved, duplicated, or reversed during evolutions. Despite the apparently chaotic randomness, those repeats undergo elementary mutations mostly during DNA replication in successive cell divisions, and then become approximate repeats. This idea motivates us to locate these segments. Our DNA compression algorithm consists of three phases. First, we build a suffix tree to locate exact repeats. Then, we extend the exact repeats into approximate repeats and the best possible combinations of non-overlapping repeats are found to compress the DNA sequence. Finally, we use two sets of codes: two bit per base for exact and approximate repeats, and arithmetic encoding for non-repeat regions. For encoding the numbers in self-delimited way, the Fibonacci encoding method is used.

The time complexity of our algorithm is bounded by $O(|S| + l|k|^2)$, where $|S|$ is the length of the input sequence, l is the average length of repeats and $|k|$ is the maximum number of repeats among all the repeat groups. We tested our proposed algorithm using some standard DNA test Sequences and the result shows that our algorithm do better than the Cfact, GenCompress and some other algorithm by several orders of magnitude. Moreover, the proposed algorithm can be used to compress long sequences with millions of bases or more.

Chapter 1

Introduction

In the twentieth century, two areas of study have grown tremendously, molecular biology and computer science. In one sense these two areas are quite different: molecular biology is concerned with understanding natural processes; whereas computer science is concerned with solving problems using a machine. However, in another sense these areas are quite similar: they both deal with the processing of information. Moreover, throughout their history they have faced similar challenges. Today, more and more DNA sequences are becoming available. The information about DNA sequences are stored in molecular biology databases. The size and importance of these databases will be bigger and bigger in the future; therefore this information must be stored or communicated efficiently. Because the DNA sequences consist of four bases, two bits are enough to store each base, but they are hard to compress further.

1.1 An Epigrammatic Account

While the idea that information is inherited through genes was developed by Mendel in the mid 1800s, it was not until 1902 that Sutton and Boveri noticed the parallel between Mendel's notion of genes and the action of chromosomes during cell division.

1.1.1 The Picture Comes Out

The process of mitosis and meiosis were discovered in the 1870s and 1890s. It was observed that, as cells divide, chromosomes moved around in a cell, and people began to wonder what their function was. People knew that DNA was also in the chromosomes, but because its structure was unknown and people didn't know much about it, few people thought it was the genetic material. In the late 1920s, and throughout the 1950s, both areas began to research how information could be stored. In molecular biology, research was going on the structure of DNA, realizing not only how genetic information was stored, but also how it could be replicated as well.

In 1919, Phoebus Levene identified the base, sugar and phosphate nucleotide unit [1]. Levene suggested that DNA consisted of a string of nucleotide units linked together through the phosphate groups. However, Levene thought the chain was short and the bases repeated in a fixed order. In 1937 William Astbury produced the first X-ray diffraction patterns that showed that DNA had a regular structure [2].

In 1928, Frederick Griffith discovered that traits of the "smooth" form of the Pneumococcus could be transferred to the "rough" form of the same bacteria by mixing killed "smooth" bacteria with the live "rough" form [3]. This system provided the first clear suggestion that DNA carried genetic information—the Avery-MacLeod-McCarty experiment—when Oswald Avery, along with coworkers Colin MacLeod and Maclyn McCarty, identified DNA as the transforming principle in 1943 [4]. DNA's role in heredity was confirmed in 1952, when Alfred Hershey and Martha Chase in the Hershey-Chase experiment showed that DNA is the genetic material of the T2 phage [5].

In 1953 James D. Watson and Francis Crick suggested what is now accepted as the first correct double-helix model of DNA structure in the journal *Nature* [6]. Their double-helix, molecular model of DNA was then based on a single X-ray diffraction image (labeled as "Photo 51") [7] taken by Rosalind Franklin and Raymond Gosling in May 1952,as well as the information that the DNA bases were paired also obtained through private communications from Erwin Chargaff in the previous years. Chargaff's rules played a very important role in establishing double-helix configurations for B-DNA as well as A-DNA.

1.2 Structure of a DNA

Deoxyribonucleic acid (DNA) is a nucleic acid that contains the genetic instructions used in the development and functioning of all known living organisms and some viruses. The main role of DNA molecules is the long-term storage of information. DNA is often compared to a set of blueprints or a recipe, or a code, since it contains the instructions needed to construct other components of cells, such as proteins and RNA molecules. The DNA segments that carry this genetic information are called genes, but other DNA sequences have structural purposes, or are involved in regulating the use of this genetic information.

A DNA sequence is a long stretch consisting of four types of nucleotides: Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). The lengths of the 24 chromosomes in human range from 50 to 250 million base pairs.

DNA sequences are made of coding and non-coding regions. Coding regions, also called "exons", code for proteins and non-coding regions called "introns" or "junk" DNA. 97% of Human Genome is made of junk DNA.

1.2.1 Use of DNA Technology

→ Genetic engineering

→ Forensics

→ Bioinformatics

→ DNA nanotechnology

→ History and anthropology

→ DNA has also been used to look at modern family relationships, and in criminal investigations.

1.2.2 Patterns in DNA

Genome sequencing is finding the order of DNA nucleotides, or bases, in a genome the order of As, Cs, Gs, and Ts that make up an organism's DNA. Sequencing the genome is an important 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. The human genome is made up of over 3 billion of these genetic letters. The human genome is about 20-40 percent repetitive DNA, but bacterial and viral genomes contain almost no repetition [8].

In repetitive DNA, the same short sequence is repeated over and over again. Somewhere in the genome the sequence GCA may be repeated 100 times in a row; elsewhere there may be 30 consecutive copies of the sequence ACTTCTG.

For example, the following DNA sequence is just a small part of telomere located at the ends of each human chromosome:

...GGGTTAGGGTTAGGGTTAGGGTTAGGGTTAGGGT

TAGGGTTAGGGTTAGGGTTAGGGTTAGGGTTAGGGTT

AGGGTTAGGGTTAGGGTTAGGGTTAGGGTTAGGG...

An entire telomere, about 15 kb, is constituted by thousands of the repeated sequence "GGGTTA".

Based on the repetition rate, DNA sequences are divided into three classes:

– Single copy : This class accounts for 50-60% of mammalian DNA.

– Moderately repetitive: Roughly 25-40% of mammalian DNA re-associates at an intermediate rate. This class includes interspersed repeats.

– Highly repetitive: About 10-15% of mammalian DNA re-associates very rapidly. This class includes tandem repeats.

The patterns found in DNA sequences are:

- Exact repeats.

  Example:

  AT**CGTAATGATTCCA**TCAGATACAGATAATATACATACAGAGAT**CGTAAT**
  GATTCCA

- Approximate repeats: Repeats with mutation – Replace, Insert, Delete.

  Example:

  Replace: GACCGTCA (the T replaced by A)

  GACCGACA

  Insert/Delete: GACCGT _CA

  GACCG_ ACA (a T is deleted and an A is inserted)

- Complementary palindromes: Two sequences are complementary if one can be obtained by the other replacing the symbol A with T (and vice-versa) and the symbol C with G (and vice-versa). We say that a sequence x is a complementary palindrome (reverse complement) of y, if x and the reverse of y are complementary.

  Example: **GACCGTCA**TCAGATACAGATAAT **TGACGGTC**.

- Local non-uniform frequencies of bases.

1.3 Why DNA Compression

The compression of DNA sequences is one of the most challenging tasks in the field of data compression. Since DNA sequences are "the code of life" we expect them to be non-random and to present some regularities. It is natural to try to take advantage of such regularities in order to compactly store the huge DNA databases which are routinely handled by molecular biologists.

Compression of DNA sequences is indispensable because,

- There are approximately 44,575,745,176 bases in 40,604,319 sequence records in the GenBank database;Source: Genbank Database *http://www.ncbi.nlm.nih.gov/Genbank/*
- Newly determined genomes have to be stored and compressed in an efficient manner.
- For related species, they have to be organized in such a way that simple cross-referencing is possible.
- Efficient compression may also reveal some biological functions, aid in phylogenic tree reconstruction.
- Compression distance can be defined for comparing two sequences.
- It is just a computer science challenge to compress better.

1.3.1 Can General purpose Compression Algorithm work for DNA

With more and more complete genomes of prokaryotes and eukaryotes becoming available and the completion of human genome project in the horizon, fundamental questions regarding the characteristics of these sequences arise. Life represents order. It is not chaotic or random. Thus, we expect the DNA sequences that encode Life to be nonrandom. In other words, they should be very compressible. There are also strong biological evidences that support this claim: it is well-known that DNA sequences, especially in higher eukaryotes, contain many (approximate) tandem repeats. The unit of compression ratio is *bit per base*. Using the standard benchmark DNA sequences; compression results are shown in Table 1-1 [9] for gzip -9;corresponding to the widely used gzip with option-9 and lz; which adopts the LZ77 scheme, as gzip, with the sliding dictionary of size 32KB to 1MB [10].

As for widely used compress, gzip, and bzip2 with default options, in all cases compress or gzip only expand in size. The compression ratio is 2.185 and 2.27 for compress and gzip.

Table 1-1: The compression ratio of some general purpose compression algorithm.

DNA sequence name	Sequence Length	gzip-9	LZ (32K)	LZ (1M)
CHMPXX	121024	2.220	2.234	2.276
CHNTXX	155844	2.291	2.300	2.352
HEHCMVCG	229354	2.279	2.286	2.344
HUMDYSTROP	38770	2.377	2.427	2.432
HUMGHCSA	66495	1.551	1.580	1.513
HUMHBB	73308	2.228	2.255	2.286
HUMHDABCD	58864	2.209	2.241	2.264
HUMHPRTB	56737	2.232	2.269	2.287
MPOMTCG	186609	2.280	2.289	2.326
PANMTPACGA	100314	2.232	2.249	2.285
SCCHRIII	315339	2.265	2.268	2.308
VACCG	191737	2.190	2.194	2.245

With the option-9, gzip can really compress "HUMGHCSA", a typical example having so many repeats. When bzip2 is used, the compression ratio of "HUMGHCSA" is 1.729 per base, but the ratios for the other sequences are all more than 2 bits per base with average value 2.138. LZ can compress only "HUMGHCSA" as in gzip-9, bzip2. It does not take account of long distances of repeat in DNA; hence if the size of the buffer becomes big the compression ratio becomes worse in many cases.

1.4 Related Works

There have been developed several special-purpose compression algorithms for DNA sequences [9]. These DNA-oriented compression algorithms use characteristic structures of DNA such as palindromes, approximate matches, and can compress them less than two bits per base.

1.4.1 BioCompress (Grumbach and Tahi 1994)

Biological sequence compression is a useful tool to recover information from biological sequences. Biocompress [11] was the first DNA specific compression algorithms. It is similar to the Ziv-Lampel data compression method. At each step, the longest factor beginning at the current position which matches with a factor starting before is chosen. The problem of this algorithm was the compression ratio was not that much satisfactory.

1.4.2 BioCompress-2

Biocompress-2 [12] detects exact repeats and complementary palindrome located in the already encoded sequence, and then encodes them by the repeat length and the position of a previous repeat occurrence. When no significant repetition is found, Biocompress-2 uses order-2 arithmetic coding (Arith-2).

1.4.3 Cfact (Rivals et al. 1996)

The Cfact algorithm [13] looks for the longest exact matching repeat. For this purpose, it uses a suffix tree on the entire sequence. Using two passes, repetitions are encoded this way when the gain is guaranteed, otherwise the two-bits-per base (2-Bits) encoding is used.It takes too much time for long sequences.

1.4.4 GenCompress (Chen et al. 1999)

The GenCompress algorithm [9,14,15] yields to a significantly better compression ratio than the previous algorithms. The idea is to use approximate (and not exact) repetitions. It exists in two variants: GenCompress-1 uses the Hamming distance (only substitutions) for the repeats while GenCompress-2 uses the edition distance (deletion, insertion and substitution) for the encoding of the repeats. GenCompress use the greedy approach for selection of the repeat segments. Its execution time is also very high.

1.4.5 CTW+LZ (Matsumoto et al. 2000)

CTW+LZ [16] is another algorithm, based on the context tree weighting method. It combines a LZ-77 type method like GenCompress and the CTW algorithm. Long exact/appoximate repeats are encoded by LZ77-type algorithm, while short repeats are encoded by CTW. Although they obtain good compression ratios, its execution time is too high to be used for long sequences.

1.4.6 DNACompress (Chen et al. 2002)

DNACompress is a DNA compression [17] tool, which employs the Ziv-Lampel compression scheme as Biocompress-2 and GenCompress. It consists of two phases: during the first phase, it finds all approximate repeats including complementary palindromes, using a specific software, PatternHunter . Then the approximate repeats and the non-repeat regions are encoded. In practice, the execution time of DNACompress is much less than GenCompress.

1.4.7 DNASequitur (Cherniavsky and Ladner 2004)

DNASequitur is a grammar-based compression algorithm for DNA sequences which infers a context-free grammar to represent the input data. Even if the algorithm is elegant, the practical results show that other methods achieve better compression ratios. DNAsequitur is the modified version of grammar based compression Sequitur adapted for DNA sequences.

Digram Uniqueness: no pair of adjacent symbols appears more than once in the grammar.

Rule Utility: each rule is used at least twice (except for the start rule).

1.4.8 DNAPack (Behzadi and Fessant 2005)

DNAPack, uses Hamming distance for the repeats and complementary palindromes, and either CTW or Arith-2 compression for the non-repeat regions.

Unlike the above algorithms, DNAPack does not choose the repeats by a greedy algorithm, but uses a dynamic programming approach instead.

1.5 Objective of this thesis

The thesis will focus on the following objectives:

1. Finding an advanced compression technique which will be able to detect more regularity in DNA sequences and achieve the best compression results by using this observation.
2. In order to get better performances departing from the approaches from the common practice in DNA compression of searching and encoding repeats/approximate repeats.
3. Making the compression process simpler, faster and efficient.
4. Performance comparison of the proposed algorithm with a few representative existing algorithms with the help of standard DNA Test Sequences.

1.6 Thesis Organization

Analyzing the previous algorithms as well as focusing on our objectives; we can agree on the following steps of a DNA sequence compression algorithm:

1. Finding the candidate repeat segments.
2. Considering approximate repeats.
3. Selecting the best subset of compatible repeats.
4. Encoding of the repeat segments.
5. Encoding of the non-repeat segments.

The content of this thesis is organized as follows: in chapter 2, we describe the different techniques and concepts used in our algorithm. In chapter 3, we present the algorithm we designed. In chapter 4 we have shown and compared our results on a standard set of DNA sequences with results published for the other algorithms. In the very last chapter we shared some concluding remarks and a few future directions.

Chapter 2

Preliminaries

In this chapter we define some basic terminology of string and compression theory. Definitions which are not included in this chapter will be introduced as they are needed. We start, in section 2.1, by giving definitions of suffix tree terms used throughout the remainder of this thesis. We devote section 2.2 to define terms related to arithmetic encoding. In section 2.3 we introduce some techniques of number encoding, e.g., Fibonacci encoding. In the last section we try to outline some basic notations used in our algorithm.

2.1 Suffix Tree

2.1.1 Definitions

Definition: A suffix tree T is built for a string $S[1...m]$. The tree is rooted and directed with m leaves, which are numbered from 1 to m. Each edge is labeled with a nonempty substring of S. The internal nodes of the tree (other than the root) all have at least two outgoing edges, and the labels of all outgoing edges are labeled with different characters.

By following the path from the root to leaf i and concatenating the edge labels, one obtains the suffix $S[i...m]$.

Definition: The *label of a path* from the root that ends at a *node* is the concatenation, in order, of the substrings labeling the edges of that path. The *pathlabel of a node* is the label of the path from the root of T to that node.

Definition: For any node *v* in a suffix tree, the *string-depth* of *v* is the number of characters in *v*'s label.

Definition: A path that ends in the middle of an edge (*u*, *v*) splits the label on (*u*, *v*) at a designated point. Define the label of such a path as the label of *u* concatenated with the characters on edge (*u*, *v*) down to the designated split point.

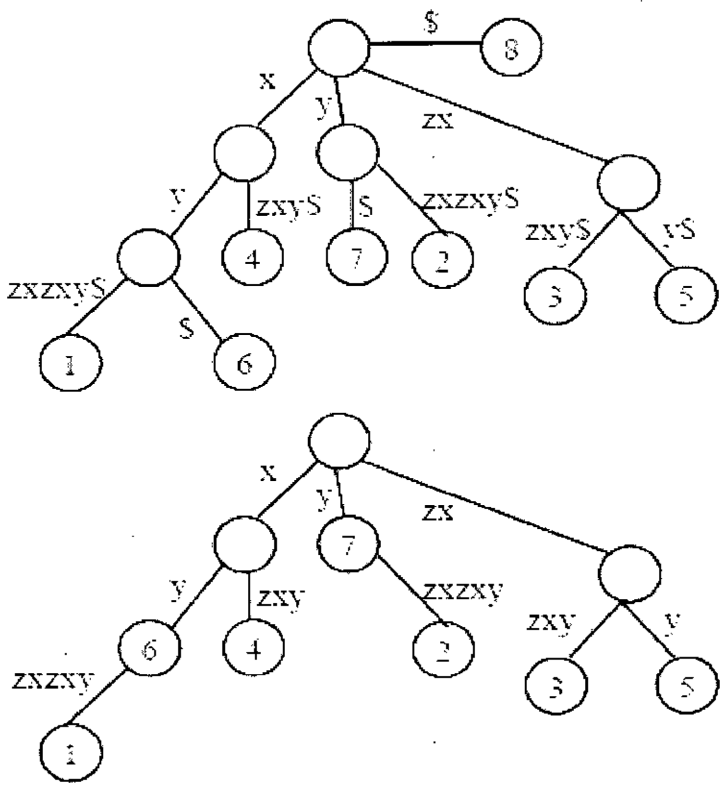


Figure 2-1 : A true suffix tree (top); why we need the $ character (bottom).

An example of a suffix tree for the string *xyzxzxy*$ is given in Figure 2-1. The figure helps understand some important points about the suffix tree.

First, each internal node has two or more children with different starting characters along the edges, since otherwise the node could be removed or moved in order to make this the case. Also, it is important that the last character of the string be a "unique" character, as this guarantees that the suffix tree as defined actually exists. For example, suppose our string was just *xyzxzxy*. The suffix tree would remain largely the same. In particular, in the not-quite-suffix tree in Figure 2-1 the path for the suffix *xy* does not end at a leaf, violating the definition. The problem is that the suffix *xy* is also the prefix of the string. This problem can be avoided by terminating the string with a special character that does not appear elsewhere, since then no suffix can also be a prefix (except for the entire string itself). Hence, from now on, we will assume all strings end with a special character $.

It is also worth noting that a more convenient representation of the suffix tree does not actually label the edges with characters. Instead, these labels can be represented by a pair of indices; labeling an edge $[i, j]$ represents that the edge label corresponds to characters $S[i \ldots j]$. Besides saving space and ensuring that each edge is conveniently represented by two numbers, this scheme is important for the linear time algorithm for suffix tree construction.

2.1.2 Functionality

A naive algorithm to build a suffix tree

This naïve method first enters a single edge for suffix $S[1..m]\$$ (the entire string) into the tree; then it successively enters suffix $S[i..m]\$1$ into the growing tree, for i increasing from 2 to m. We let tree_1, denote the intermediate tree that encodes all the suffixes from 1 to i.

In detail, tree_1 consists of a single edge between the root of the tree and a leaf labeled 1. The edge is labeled with the string $S\$$. Tree tree_{i+1} is constructed from tree_1, as follows:

Staring at the root of tree, find the longest path from the root whose label matches a prefix of $S[i + 1..m]\$$. This path is found by successively comparing and matching characters in suffix

$S[i + 1..m]\$$ to characters along a unique path from the root, until no further matches are possible. The matching path is unique because no two edges out of a node can have labels that begin with the same character. At some point, no further matches are possible because no suffix of $S\$$ is a prefix of any other suffix of $S\$$. When that point is reached, the algorithm is either at a node, to say, or it is in the middle of an edge. If it is in the middle of an edge, (u, v) say, then it breaks edge (u, v) into two edges by inserting a new node, called w, just after the last character on the edge that matched a character in $S[i + 1..m]$ and just before the first character on the edge that mismatched. The new edge (u, w) is labeled with the part of the (u, v) label that matched with $S[i + 1..m]$, and the new edge (w, v) is labeled with the remaining part of the (u, v) label. Then (whether a new node w, was created or whether one already existed at the point where the match ended), the algorithm creates a new edge $(w, i + 1)$ running from w to a new leaf labeled $i + 1$ and it labels the new edge with the unmatched part of suffix $S[i + 1..m]\$$.

The tree now contains a unique path from the root to leaf $i + 1$, and this path has the label $S[i + 1. .m]\$$. Note that all edges out of the new node w have labels that begin with different first characters, and so it follows inductively that no two edges out of a node have labels with the same first character.

2.1.3 Application of Suffix Tree

- Exact string and substring matching
- Longest common substrings
- Finding and representing repeated substrings efficiently
- Applications that lead to alternative, space efficient implementations
  - Matching statistics
  - Suffix Arrays

- Comparing suffix trees and keyword trees for exact set matching
- The substring problem for a database of patterns
- Recognizing DNA contamination
- Common substrings of more than two strings

2.1.4 Implementation

There are two implementation issues

- Alphabet size
- Generalizing to multiple strings

Building a Suffix Tree, Slowly

We show how a suffix tree might be built "by hand". Three dots, `...', are used to show the current end of any suffix that will grow as more characters are processed. Starting with the empty suffix tree, consider the string `m':

```
           tree1

tree-->----m...
```

Adding the second character to get `mi' there are now suffixes `mi' and `i':

```
           tree2

tree-->|---mi...
       |
       |---i...
```

Next `mis'

```
           tree3

tree-->|---mis...
       |
       |---is...
       |
       |---s...
```

There is no need to add any more splits for `miss' because `s' is part of `ss'.

tree$_4$

```
tree-->|---miss...
       |
       |---iss...
       |
       |---ss...
```

However, with `missi' there must be a split because one `s' is followed by `i', the other by `s'

tree$_5$

```
tree-->|---missi...
       |
       |---issi...
       |
       |---s-->|---si...
               |
               |---i...
```

The 6th character, `s', brings us to `missis' and no split because both `i's are followed by `s's.

tree$_6$

```
tree-->|---missis...
       |
       |---issis...
       |
       |---s-->|---sis...
               |
               |---is...
```

`mississ'

tree$_7$

```
tree-->|---mississ...
       |
       |---ississ...
       |
       |---s-->|---siss...
               |
               |---iss...
```

`mississi'

tree$_8$

```
tree-->|---mississi...
       |
       |---ississi...
       |
       |---s-->|---sissi...
               |
               |---issi...
```

A lot suddenly happens for `mississip', because it brings the first `p', and causes the third `i' to be followed by `p' where the other two are followed by `ssi'. Consequently one of the `ssi' is followed by `p', the other by `ssip', ditto `si'.

tree$_9$

```
tree-->|---mississip...
       |
       |---i-->|---ssi-->|---ssip...
       |       |         |
       |       |         |---p...
       |       |
       |       |---p...
       |
       |---s-->|---si-->|---ssip...
       |       |        |
       |       |        |---p...
       |       |
       |       |---i-->|---ssip...
       |               |
       |               |---p...
       |
       |---p...
```

By comparison `mississipp' is very quiet

tree$_{10}$

```
tree-->|---mississipp...
       |
       |---i-->|---ssi-->|---ssipp...
       |       |         |
       |       |         |---pp...
       |       |
       |       |---pp...
       |
        |---s-->|---si-->|---ssipp...
        |       |        |
        |       |        |---pp...
        |       |
        |       |---i-->|---ssipp...
        |               |
        |               |---pp...
        |
        |---pp...
```

`mississippi' is an anti-climax

```
            tree11

  tree-->|---mississippi
         |
         |---i-->|---ssi-->|---ssippi
         |       |         |
         |       |         |---ppi
         |       |
         |       |---ppi
         |
         |---s-->|---si-->|---ssippi
         |       |        |
         |       |        |---ppi
         |       |
         |       |---i-->|---ssippi
         |               |
         |               |---ppi
         |
         |---p-->|---pi
                 |
                 |---i
```

The challenge, to a computer scienctist, is to make sure $tree_i$ is updated to $tree_{i+1}$ *efficiently*. This can be done (Ukkonen 1992, 1995) so that $tree_n$ can be built, starting from $tree_0$, in O(n)-time overall.

2.1.5 Cost

Assuming, as usual, a bounded-size alphabet, the above naive method takes $O(m^2)$ time to build a suffix tree for the strings of length *m*.

Building Suffix Trees in O(m) time

- Weiner had first linear time algorithm in 1973
- McCreight developed a more space efficient algorithm in 1976
- Ukkonen developed a simpler to understand variant in 1995
  - This is what we will use in our algorithm.

Effects of alphabet size on suffix trees

- We have generally been assuming that the trees are built in such a way that from any node, we can find an edge in constant time for any specific character in S .
- an array of size |S| at each node
- This takes $O(m|S|)$ space.

2.2 Encoding of text

There are different alternative approaches for text encoding. Such as:

1. Fix number of bits per symbol

2. Huffman Encoding

3. Adaptative Huffman Encoding

4. Arithmetic Coding

5. Adaptative Arithmetic Coding

6. Context Tree Weighted method

2.2.1Arithmetic Encoding

Arithmetic coding is a method for lossless data compression. Normally, a string of characters such as the words "hello there" is represented using a fixed number of bits per character, as in the ASCII code.

The method of arithmetic coding was suggested by Elias, and presented by Abramson in his text on Information Theory. Implementations of Elias' technique were developed by

Rissanen, Pasco, Rubin, and, most recently, Witten et al. We present the concept of arithmetic coding first and follow with a discussion of implementation details and performance.

2.2.2 How it works

In arithmetic coding a source ensemble is represented by an interval between 0 and 1 on the real number line. Each symbol of the ensemble narrows this interval. As the interval becomes smaller, the number of bits needed to specify it grows. Arithmetic coding assumes an explicit probabilistic model of the source. It is a defined-word scheme which uses the probabilities of the source messages to successively narrow the interval used to represent the ensemble. A high probability message narrows the interval less than a low probability message, so that high probability messages contribute fewer bits to the coded ensemble. The method begins with an unordered list of source messages and their probabilities. The number line is partitioned into subintervals based on cumulative probabilities.

Static arithmetic encoder starts from collection of statistics.

Cumulative probability is sum of all previous probabilities in the right column. It starts from 0.0 because there is no previous probability value. While probabilities may be repeated the cumulative probabilities are disjoints and can be uniquely associated with symbols. Next step is computation of two limits `low`, L and `high` by formula.

$$L = C_1 + \sum_{i=2}^{n} C_i \prod_{j=1}^{i-1} P_j + r \qquad 0 \leq r < \prod_{j=1}^{n} P_j$$

Here, Cumulative probability C_k and Individual Probability P_k.

2.2.3 An Intuitive Example

A small example will be used to illustrate the idea of arithmetic coding. Given source messages {*A,B,C,D,#*} with probabilities {.2, .4, .1, .2, .1}, Figure 2-2 demonstrates the initial partitioning of the number line. The symbol *A* corresponds to the first 1/5 of the interval [0,1); *B* the next 2/5; *D* the subinterval of size 1/5 which begins 70% of the way from the left endpoint to the right. When encoding begins, the source ensemble is represented by the entire interval [0,1). For the ensemble *AADB#*, the first *A* reduces the interval to [0,0.2) and the seco[illegible] *A* to [0,0.04) (the first 1/5 of the previous interval). The *D* further narrows the interval [illegible] [0.028,0.036) (1/5 of the previous size, beginning 70% of the distance from left to right). The *B* narrows the interval to [0.0296,0.0328), and the # yields a final interval of [0.03248,0.0328). The interval, or alternatively any number *i* within the interval, may now be used to represent the source ensemble.

Source message	Probability	Cumulative Probability	Range
A	0.2	0.2	[0,0.2)
B	0.4	0.6	[0.2,0.6)
C	0.1	0.7	[0.6,0.7)
D	0.2	0.9	[0.7,0.9)
#	0.1	1.0	[0.9,1.0)

Figure 2-2: The Arithmetic coding model.

The size of the final subinterval determines the number of bits needed to specify a number in that range. The number of bits needed to specify a subinterval of [0,1) of size *s* is -lg *s*. Since the size of the final subinterval is the product of the probabilities of the source messages in the ensemble (that is, *s*=PROD{*i*=1 to *N*} *p*(source message *i*) where *N* is the length of the ensemble), we have -lg *s* = -SUM{*i*=1 to *N* lg *p*(source message *i*) = - SUM{*i*=1 to *n*} *p*(*a*(*i*)) lg *p*(*a*(*i*)), where *n* is the number of unique source messages *a*(1), *a*(2), ... *a*(*n*). Thus, the number of bits generated by the arithmetic coding technique is exactly equal to entropy, *H*.

This demonstrates the fact that arithmetic coding achieves compression which is almost exactly that predicted by the entropy of the source.

In order to recover the original ensemble, the decoder must know the model of the source used by the encoder (e.g.; the source messages and associated ranges) and a single number within the interval determined by the encoder. Decoding consists of a series of comparisons of the number *i* to the ranges representing the source messages. For this example, *i* might be 0.0325 (0.03248, 0.0326, or 0.0327 would all do just as well). The decoder uses *i* to simulate the actions of the encoder. Since *i* lies between 0 and0.2, he deduces that the first letter was *A* (since the range [0, 0.2) corresponds to source message *A*). This narrows the interval to [0, 0.2). The decoder can now deduce that the next message will further narrow the interval in one of the following ways: to [0, 0.04) for *A*, to [0.04, 0.12) for *B*, to [0.12, 0.14) for *C*, to [0.14, 0.18) for *D*, and to [0.18, 0.2) for #. Since *i* falls into the interval [0, 0.04), it knows that the second message is again *A*. This process continues until the entire ensemble has been recovered.

2.2.4 Order-2 Arithmetic Coding

In comparison to the Huffman Coding algorithm, Arithmetic Coding overcomes the constraint that the symbol to be encoded has to be coded by a round number of bits. This leads to higher efficiency and a better compression ratio in general. The adaptative arithmetic coding of order 2 has the best compression ratio on the DNA sequences; in this arithmetic encoding the adaptative probability of a symbol is computed from the context after which it appears.

2.3 Fibonacci Encoding

Different integer numbers have to be encoded in our algorithm. For example, the segments have not a fixed length, so this length has to be encoded. For any repeat segment, the position of the reference substring of the input, from which we copy this segment, should be encoded. When the copies are approximate (and not exact) the positions of the modifications should be encoded. There is no bound on any of these numbers, so these integers should be encoded in a self delimited way rather than being encoded in a fix number of bits. For encoding the reference position, encoding the relative difference of position of the reference and the copy itself is preferable.

2.3.1 Characterization

An efficient self-delimited representation of arbitrary numbers is the Fibonacci encoding [18]. The Fibonacci encoding is based on the fact that any positive integer n can be uniquely expressed as the sum of distinct Fibonacci numbers, so that no two consecutive Fibonacci numbers are used in the representation. This means that if we use binary representation of a Fibonacci encoding of a number, there are no two consecutive 1 bits. So by adding a 1 after the 1 corresponding to the biggest Fibonacci number in the sum, the representation becomes self-delimited.For a number N, if $[d(0), d(1), d(2),\ldots, d(k)]$ represent the digits of the coded form of N then we have:

$$N = \sum_{i=0}^{k} d(i)F(i)$$

$$d(i) = 1 \Rightarrow d(i+1) = 0$$

where $F(i)$ is the ith Fibonacci number. No two adjacent coefficients $d(i)$ can be 1.

To encode an integer *X*:

1. Find the largest Fibonacci number equal to or less than *X*; subtract this number from *X*, keeping track of the remainder.
2. If the number we subtracted was the *N*th unique Fibonacci number, put a one in the *N*th digit of our output.
3. Repeat the previous steps, substituting our remainder for *X*, until we reach a remainder of 0.
4. Place a one after the last naturally-occurring one in our output.

Table 2-1: Fibonacci representation of some integers

Symbol	**Fibonacci representation**	**Fibonacci code**
1 =	F(1)	11
2 =	F(2)	011
3 =	F(3)	0011
4 =	F(1)+F(3)	1011
5 =	F(4)	00011
6 =	F(1)+F(4)	10011
7 =	F(2)+F(4)	01011
8 =	F(5)	000011
10 =	F(2)+F(5)	010011

2.3.2 Comparison with other universal codes

Fibonacci coding has a useful property that sometimes makes it attractive in comparison to other universal codes: it is an example of a self-synchronizing code, making it easier to recover data from a damaged stream. With most other universal codes, if a single bit is altered, none of the data that comes after it will be correctly read. With Fibonacci coding, on the other hand, a changed bit may cause one token to be read as two, or cause two tokens to be read incorrectly as one, but reading a "0" from the stream will stop the errors from propagating further. Since the only stream that has no "0" in it is a stream of "11" tokens, the total edit distance between a stream damaged by a single bit error and the original stream is at most three.

2.4 Basic Notations

Let S be a DNA sequence. S[i,j], i<j, denotes the substring of S starting with the ith character and ending with the jth character of S. An exact repeat r is a string s occurs exactly twice in S. The first occurrence of s is never encoded. It is usually used as a reference. The second occurrence of s is encoded by a reference pointer to the first occurrence. A reference pointer of an exact repeat contains the following information: the position of the first occurrence, the length of the first occurrence and the position of the second occurrence. For example, if s occurs at positions i and j, the reference pointer will be (i, j, |p|), where |p| is the length of s. F(i, j, |p|) is the bit-stream used to encode the reference pointer and |F(i, j, |p|)| is the number of bits, where F is the encoding method. For the purpose of implementation, every occurrence s has a gain which is equal to $2 \times |p| - |F(i, j, |p|)|$. That is, gain(s) = $2 \times |p| - |F(i, j, |p|)|$. The higher the gain is, the more bits this encoding can save. If a string occurs more than twice in the sequence, we will encode every occurrence except for the first one.

Two strings are similar or approximate if their edit-distance is smaller than a pre-defined threshold τ. The edit-distance is the minimum number of edit operations (insertion, deletion and replacement) to transform one string into another. Let EO(a,b) denotes the edit operations transforming a string a into another string b. |EO(a,b)| is the number of edit operations and is said to be the edit-distance between a and b.

To locate the approximate repeats, we first locate the exact repeats. Then we try to extend the latter into the former as long as their edit-distance is still smaller than τ. An exact repeat is a special case of an approximate repeat when τ is equal to zero. In order to encode an approximate repeat, the reference pointer contains one additional field: a list of edit operations, EO. For example, if two similar strings, a and b positioned at i and j respectively, are extended to approximate repeats, the reference pointer would be: ((i, |a|, j), EO(a, b)).

For DNA sequences, it is enough to encode each base with two bits because there are only four kinds of nucleotides. That is, a string s can be encoded by $2 \times |s|$ bits without any compression. We define s as compressible if it can be encoded with less than $2 \times |s|$ bits. So we do not encode a string unless it is compressible

Chapter 3

Our Algorithm (DNASqueeze)

The goal of the compression algorithm for DNA sequences aims at lowering the compression ratio and reducing the running time. Since a DNA sequence may contain up to millions of bases, or even more, the time and space complexity of the compression algorithm is required to be as low as possible.

3.1 The Algorithm

Some of the DNA compression algorithms use only exact repeats to encode DNA sequences. While some other algorithm uses approximate repeats to encode DNA sequences; and choose sequences in a greedy approach. Therefore, by combining the advantages of both approaches, we propose an efficient lossless compression method to compress DNA sequences.

Our algorithm consists of three phases. First, we build a suffix tree to locate exact repeats. Second, all the exact repeats are extended into approximate repeats by the MDR algorithm [19] and we find the optimal combination of non-overlapping repeats from the approximate repeats found in the previous phase to compress the DNA sequence. Finally, we use two bit/base for repeat encoding and arithmetic encoding for non-repeat regions.

The Fibonacci encoding method is used to encode the repeats in a self-delimited way.

The Algorithm is listed below:

```
1. Read the DNA sequence and build a suffix tree for it.
2. Locate the maximal exact repeats.
3. Extend the maximal exact repeats into approximate repeats.
4. Find the best possible combination of approximate repeats
   for each repeat group.
5. Compress the exact and approximate repeats and the non-
   repeat segments.
```

Figure 3-1: The proposed algorithm

Our proposed method consists of some phases, each of which will be described in detail in the following subsections

3.2 Building the Suffix tree to locate exact repeats

According to the property of suffix tree, every internal node can be mapped to a repeat in the input sequence S. The corresponding repeat can be located by concatenating the substrings of edges along the path from the root to the internal node. Our proposed method first constructs a suffix tree for S. Then we traverse all the internal nodes, looking for the maximal repeats. A repeat is maximal if it is not contained by any other repeats. To get the maximal repeats from the suffix tree, we can search for the internal nodes whose children are all leaf nodes. The maximal repeats can be formed by concatenating the substrings of edges along the paths from the root to those internal nodes.

3.2.1 Locating repeats

Build suffix tree $\mathscr{T}$ for text T

- Match pattern P against tree starting at root until
    - Case 1, P is completely matched
        - Every leaf below this match point is the starting location of P in T
    - Case 2: No match is possible
        - P does not occur in T

Example:

T = ACTACG

- suffixes ={ACTACG,CTACG,TACG,ACG,CG,G}

Pattern P_1: AC

Pattern P_2: AT

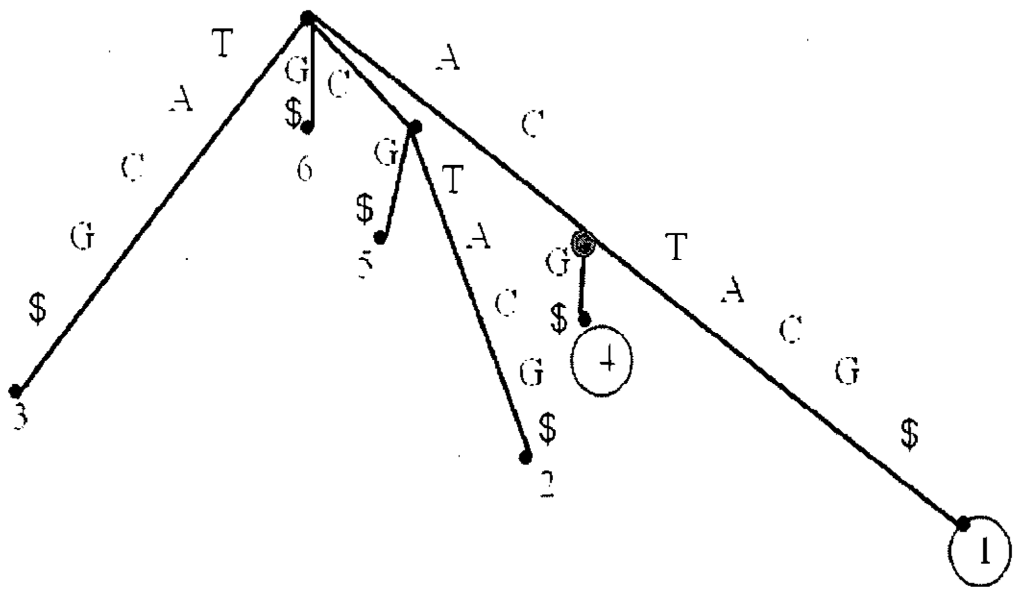


Figure 3-2: Searching repeats in a suffix tree

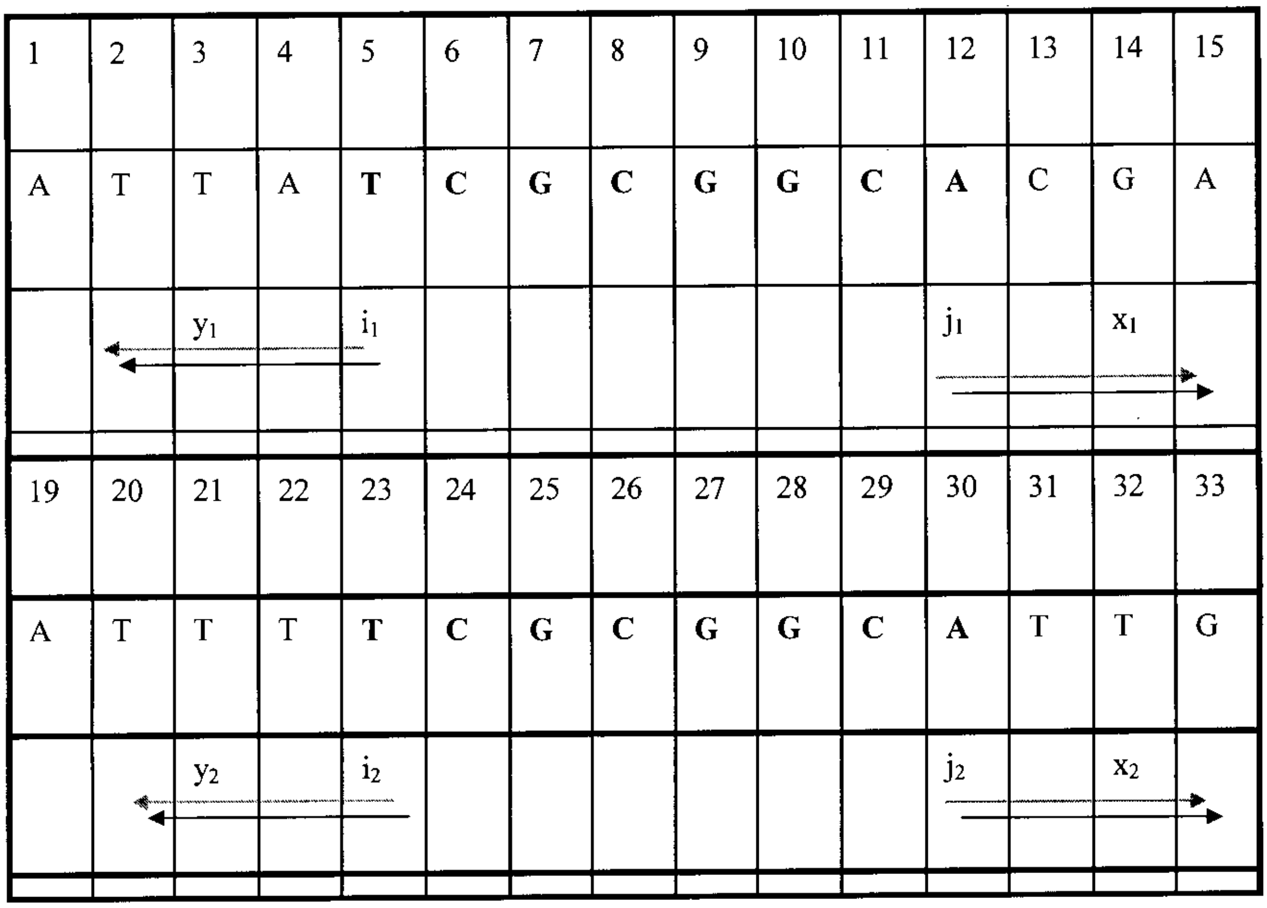
3.2.2 The code for suffix tree

```
Update( new_suffix )
{
  current_suffix = active_point
  test_char = last_char in new_suffix
  done = false;
  while ( !done ) {
    if current_suffix ends at an explicit node {
      if the node has no descendant edge starting with test_char
        create new leaf edge starting at the explicit node
       else
         done = true;
     } else {
       if the implicit node's next char isn't test_char {
         split the edge at the implicit node
         create new leaf edge starting at the split in the edge
       } else
         done = true;
     }
     if current_suffix is the empty string
       done = true;
     else
        current_suffix = next_smaller_suffix(current_suffix )
    }
    active_point = current_suffix
  }
```

Figure 3-3: Suffix tree code

3.3 Extending exact repeats into approximate repeats

After locating all the exact repeats by the suffix tree, we can use the MDR algorithm to extend each of them into an approximate repeat. Let $S[i_1, j_1]$ and $S[i_2, j_2]$, $i_1 \neq i_2$, be an exact repeat in S, and τ be the threshold of the number of edit operations. To extend the exact repeat into an approximate repeat with less than τ edit operations, we first find two tables Left and Right of size $\tau+1$. Right[z] records the maximum x_1 and x_2 such that the edit distance between $S[j_1+1, x_1]$ and $S[j_2+1, x_2]$ is less than or equal to z, where $j_1+1 < x_1 < |S|$ and $j_2+1 < x_2 < |S|$. Left[z] records the minimum y_1 and y_2 such that the edit distance between S[y_1, i_1-1] and S[y_2, i_2-1} is less than or equal to z, where 1< y_1 < i_1-1 and 1< y_2 < i_2-1. For each z, 1<z<τ, we can use Right[z] and Left[τ-z] to compute the score of the possible extended repeat. Then, we can choose among them the highest score to extend the repeat.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	T	T	A	T	C	G	C	G	G	C	A	C	G	A
	←	y_1		i_1							j_1		x_1	→
19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
A	T	T	T	T	C	G	C	G	G	C	A	T	T	G
	←	y_2		i_2							j_2		x_2	→

Figure 3-4: Probing the left and right wings of the repeated regions

3.4 Rearranging overlapping and non- overlapping repeats

Some repeats may overlap the other repeats. Two repeats overlap each other when one's ending position locates between another's beginning and ending positions. How to choose the optimal combination of repeats from a sequence of overlapping repeats to be compressed is a critical problem in compressing DNA sequences. The Cfact algorithm always chooses to encode the longest exact repeat and skip the shorter ones. The GenCompress algorithm scans the sequence from left to right and therefore always choose the approximate repeat first encountered.

In each step, our algorithm chooses the most profitable segment which does not intersect with the already chosen segments. To choose the optimal combination of repeats from a sequence of overlapping repeats, let's first consider the example shown.

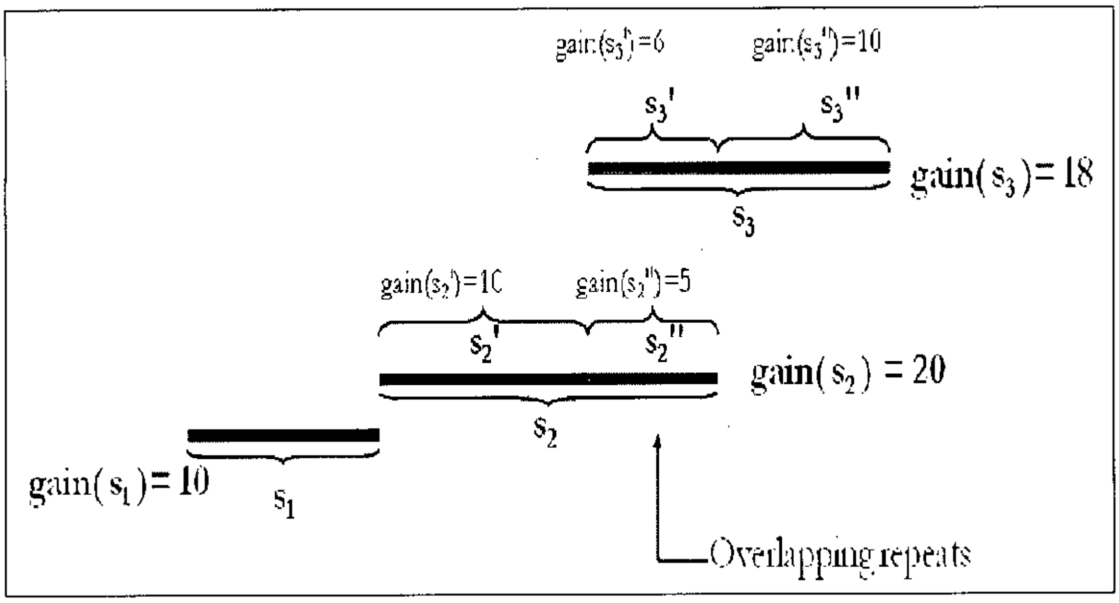


Figure 3-5: Considering three repeats where two overlapping and one is non-overlapping.

In Figure 3-5, there are three repeats of S; the value in the parenthesis denotes the score of encoding the repeat.

Case 1: If two repeats s_1 and s_2 do not overlap each other, the gain of encoding both repeats

equal to their total gain ; gain (s_1 , s_2) = gain(s_1)+gain (s_2).

For example, gain (s_1 , s_2) = gain(s_1)+gain (s_2) =10+20=30.

Case 2: If two repeats s_2 and s_3 overlap each other, we can encode in two ways

i. the prefix of s_2 with the whole of s_3, or

ii. Encode the whole s_2 with the suffix of s_3.

i. The gain of encoding the prefix of s_2 and the whole s_3 = gain (s_2') +gain (s_3)

= 10+18 = 28.

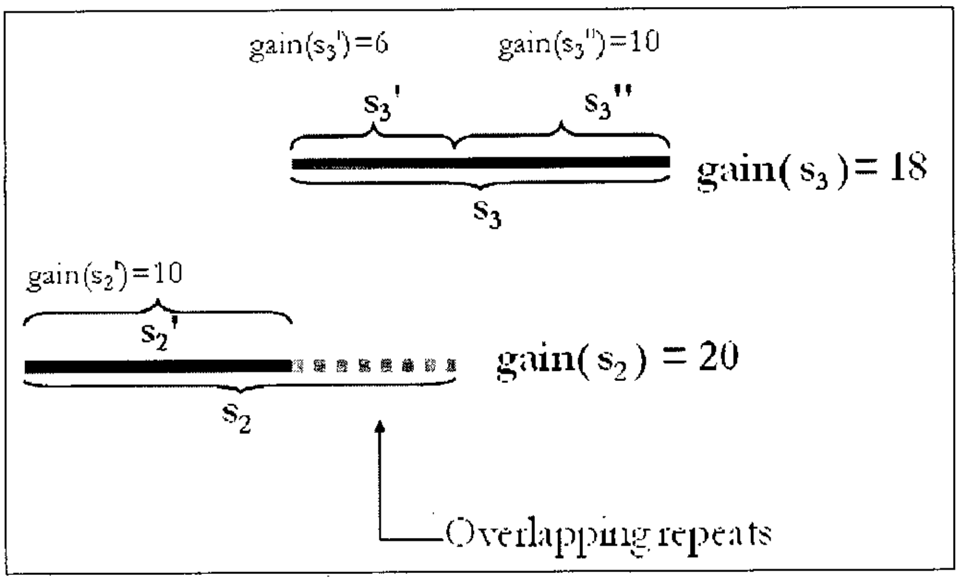


Figure 3-6: Encoding the prefix of s_2 and the whole s_3

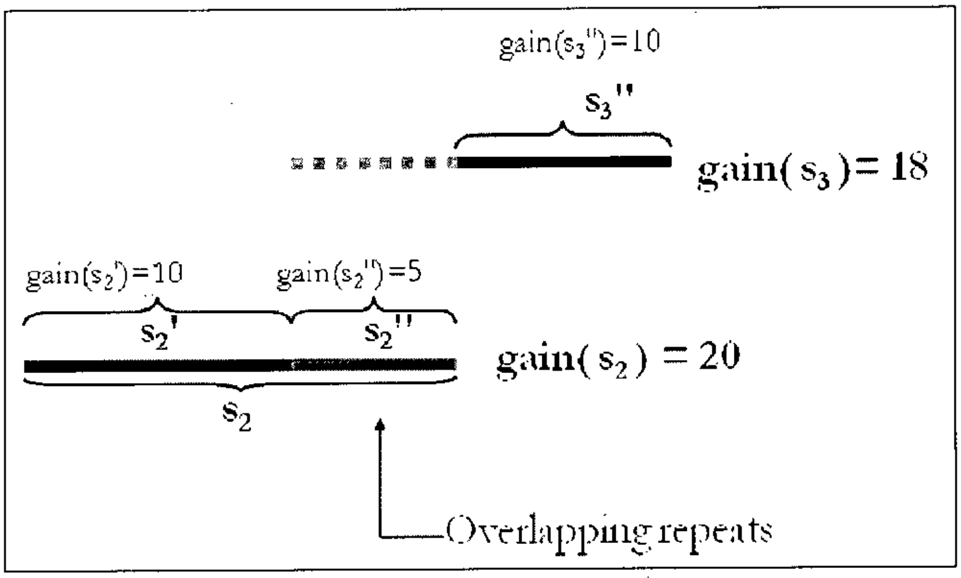


Figure 3-7: Encode the whole s_2 with the suffix of s_3.

ii. The gain of encoding the whole s_2 and the suffix of s_3 = gain (s_2) +gain (s_3'')

= 20+10 = 30

So, we can obtain a higher score to encode the whole s_2 and the suffix of s_3''.

That is, gain (s_2,s_3) = 30. Since repeat s_1 does not overlap repeats s_2 and s_3, the gain of encoding repeats s_1, s_2 and s_3, gain(s_1, s_2, s_3)=score(s_1)+ gain(s_2, s_3)=10+30=40.

To find the best possible combination of repeats, we first remove the repeats with negative scores. Next, we scan the sequence from left to right to collect the endpoints of the repeats found. The sequence of endpoints collected will be automatically sorted in non-decreasing order. Then, we scan the sequence of endpoints from left to right.

```
Algorithm: locate_arrangements

Input: a sequence of sorted endpoints of repeats in S.

Output: the optimal combination of repeats.

1.  Initialization
2.  for each endpoint p in the sequence do
3.      if p is a left endpoint of a repeat r then
4.         set its gain and compute its prefix and suffix gain;
5.         for each repeat s' that overlaps s and its left
                          endpoint is less than that of s do
6.            compute gain of encoding s' itself, with its prefix
                                                     or suffix;
7.            update maximum_gain if it is greater than the previous
              gain
8.         end for;
9.      end if;
10. end for;
11. comparing maximum_gain for each overlapping repeat we can obtain
    the best possible combination of repeats.
```

Figure 3-8: Algorithm for locating best possible arrangements of repeats

Theorem 1

The time complexity of the locate_arrangements algorithm is bounded by $O(l|k|^2)$, where l is the average length of the repeats and |k| is the number of repeats.

Proof:

The time complexity of computing the gain of encoding difference in endpoints in step 6 is bounded O(the length of s'). Similarly, the time complexity of computing gain of encoding prefix in step 6 is bounded by O(the length of s'). Likewise, the time complexity of gain of encoding suffix in step 6 is bounded by O(the length of s). The time complexity of the other associated steps needed is bounded by O(1). Consequently, the time complexity of step 5 to step 10 is bounded by O(average length of repeat). The time complexity of steps 3~4 and steps 6~7 is bounded by O(1). For each repeat, there are at most |k| repeats overlapping it. Consequently, the time complexity of steps 2~9 is bounded by $O(l|k|^2)$, where l is the average length of the repeats and |k| is the number of repeats. The time complexity of step 11 is bounded by O(|k|). Therefore, the complexity of the locate_arrangements algorithm is bounded by $O(l|k|^2)$, where l is the average length of the repeats and |k| is the number of repeats.

A repeat s is said to be reachable from s′ if s = s′ or if there is a series of repeats s_1, s_2, …, s_k such that s overlaps s_1, s_i overlaps s_{i+1}, i=1, 2,…, k-1, and s_k overlaps s′. By the property of reachability, we can divide the repeats in a sequence into several groups so that the repeats in the same group are reachable from each other. Then, for the repeats in a group, we can apply the locate_arrangements algorithm to find the best possible combination of repeats. Therefore, the time complexity of the locate_arrangements algorithm for a sequence is bounded by $O(l|k|^2)$ where l is the average length of repeats and |k| is the maximum number of repeats among all the groups.

3.5 Encoding the repeat and non-repeat region

The repeat and non-repeat regions will be encoded differently.

- Non-repeat regions will be represented by segment length and bases and repeats will have specific codes to distinguish between exact and approximate repeats.

For the encoding of the repeat regions, we used two bit/base encoding.

To encode non-repeat regions we use arithmetic encoding.

Two bit/base:

A	00
T	01
C	10
G	11

Algorithm: Arithmetic Encoding

Input: String of any length

Output: Message encoded into a number

```
1. Set low to 0.0
2. Set high to 1.0
3. While there are still input symbols do
4.     get an input symbol
5.     code_range = high - low.
6.     high = low + range*high_range(symbol)
7.     low = low + range*low_range(symbol)
8. End of While
9. output low
```

Figure 3-9: Algorithm for arithmetic encoding

Algorithm: Decoding

Input : Encoded number

Output: The original message

```
1. get encoded number
2. Do
3.   find symbol whose range straddles the encoded number
4.   output the symbol
5.   range = symbol low value - symbol high value
6.   subtract symbol low value from encoded number
```

```
7.   divide encoded number by range
8. until no more symbols
```

Figure 3-10: Algorithm for arithmetic decoding

3.6 Encoding the numbers

Different integer numbers have to be encoded in our algorithm.

The segments have not a fixed length, so this length has to be encoded.

For any repeat segment, the position of the reference substring of the input, from which we copy this segment, should be encoded.

When the copies are approximate (and not exact) the positions of the modifications should be encoded.

There is no bound on any of these numbers, so these integers should be encoded in a self delimited way rather than being encoded in a fix number of bits.

The Fibonacci encoding is based on the fact that any positive integer n can be uniquely expressed as the sum of distinct Fibonacci numbers, so that no two consecutive Fibonacci numbers are used in the representation.

- In Binary representation of a Fibonacci encoding of a number, there are no two consecutive 1 bits.

So by adding a 1 after the 1 corresponding to the biggest Fibonacci number in the sum, the representation becomes self-delimited.

Algorithm: Fibonacci encoding

Input: Any integer x

Output: Fibonacci representation of x

```
Fibonacci_encode(x):
   1.  "Returns the fibonacci encoding of the integer `x'"
   2.  R = []
   3.  i = 0
   4.  while fibonacci(i) < x:
        a.   i += 1
   5.  while x > 0:
        a.   if fibonacci(i) <= x:
              i.     R.append(i)
             ii.     x -= fibonacci(i)
        b.   i -= 1
   6.  retv = ""
   7.  for i in range(max(R) + 1):
        a.   if i in R:
              i.     retv += '1'
        b.   else:
              i.     retv += '0'
   8.  retv += '1'
   9.  return retv
```

Figure 3-11: Algorithm for Fibonacci encoding

Algorithm: Fibonacci decoding

Input: Fibonacci representation of x

Output: Integer representation of x

```
decode(x):
   1.  "Returns the fibonacci decoding of the string `x'"
   2.  retv = 0
   3.  for i in range(len(x) - 1):
        a.   if x[i] == '1':
              i.     retv += fibonacci(i)
   4.  return retv
```

Figure 3-12: Algorithm for Fibonacci decoding

3.6.1 Transformation of index into offset

Since a DNA sequence may contain millions bases or more, the distance between the first occurrence and the second occurrence of a repeat may be large. If the distance between the occurrences is too large, the occurrence may be incompressible. For example, if we want to encode an exact repeat whose length is equal to 10, which is a very common length of a repeat in human DNA sequences, the reference pointer is (i, j, 10). We've known that the Fibonacci encoding bit stream of 10 is "010011" if m=2. The bit stream has 6 bits in total. Therefore, to make the repeat compressible, the Fibonacci encoding of i and j must have no more than 2*10 - 6 = 14 bits since the repeat can be encoded by 2*10 bits without any compression. The delimiter of an index contains 1 bits, so 2 bits are required for two indexes. The maximum integer represented by 12 (=14-2) bits of Fibonacci encoding bit stream is 232. In other words, for an exact repeat whose length is equal to 10, the sum of the indexes i and j can't be larger than 232. Otherwise, the exact repeat is not compressible.

"1 0 1 0 1 0 1 0 1 0 1 1" = 1+3+8+21+55+144 = 232

1	0	1	0	1	0	1	0	1	0	1	1
F(1) = 1	F(2) = 2	F(3) = 3	F(4) = 5	F(5) = 8	F(6) = 13	F(7) = 21	F(8) = 34	F(9) = 55	F(10) = 89	F(11) =144	De-limiter

Figure 3-13: Maximum number represented by 12 bit (Fibonacci encoding)

Theorem 2:

In the Fibonacci encoding method, the maximum number represented with n bits is not larger than F(n+1), where F(n+1) is the (n+1)-th Fibonacci number.

Proof:

Let's prove the lemma by contradiction. Assume that the maximum number represented with *n* bits in the Fibonacci encoding method is larger than *F(n+1)*. Let *m* be the order of Fibonacci encoding. We first consider the case where *n* is a multiple of *m*. That is, *n=km* where *k* is a positive integer. According to the property 2 of the *Fibonacci encoding method* (the Fibonacci encoding number does not contain *m* consecutive 1-bits),the maximum number *M* represented with *n* bits is

$$M = \overset{mbits}{11\ldots.\,10}\;\overset{mbits}{11\ldots.10}\;\overset{mbits}{11\ldots.10}$$

where 11...10 is an *m*-bit pattern containing *m*-1 1-bits and 1 0-bit, and the patterns are repeated *k* times.

$$M=\sum_{i=1}^{km}F(i)-\sum_{j=0}^{k-1}F(j*m+1).$$

$$
\begin{aligned}
&F(n+1)-M\\
&=F(n)+F(n-1)+\ldots+F(n-m+1)-\left(\sum_{i=1}^{km}F(i)-\sum_{j=0}^{k-1}F(j*m)\right)\\
&=F(n-m+1)-\left(\sum_{i=1}^{(k-1)m}F(i)-\sum_{j=0}^{k-2}F(j*m+1)\right)\\
&=F(n-m)+F(n-m-1)+\ldots+F(n-2m+1)-\left(\sum_{i=1}^{(k-1)m}F(i)-\sum_{j=0}^{k-2}F(j*m+1)\right)\\
&=F(n-2m+1)-\left(\sum_{i=1}^{(k-2)m}F(i)-\sum_{j=0}^{k-3}F(j*m+1)\right)\\
&\qquad\vdots\\
&=F(n-(k-1)*m+1)-\left(\sum_{i=1}^{m}F(i)-\sum_{j=0}^{0}F(j*m+1)\right)\\
&=F(m)+F(m-1)+\ldots+F(1)-(F(m)+F(m-1)+\ldots+F(2))\\
&=F(1)\geq 0
\end{aligned}
$$

That is, *F(n+1)>M*. But this contradicts the earlier claim that the maximum number represented with *n* bits in the Fibonacci encoding method is larger than *F(n+1)*.

For the case where n is not a multiple of m, we can prove it similarly. Let n=km+l where both k and l are positive integers and m>l. In this case, the maximum number M' represented with n bits is

$$M' = \underbrace{11\cdots10}_{\text{mbits}}11\cdots10\cdots11\cdots1011\cdots1$$

Similarly, we have F(n+1)-M' > 0. But this contradicts the earlier claim that the maximum number represented with n bits in the Fibonacci encoding method is larger than F(n+1). Therefore, we can prove that the maximum number represented with n bits in the Fibonacci encoding method is not larger than F(n+1).

According to Theorem 2, we can have the following theorem.

Theorem 3:

A reference pointer (i, j, |p|) of an exact repeat is compressible if i + j < F(2*|p|-|E(|p|)|-2*(m + 1)), where p is the string that occurs at i and j, |p| is the length of the repeat, E(|p|) is the Fibonacci encoding bit stream of |p|, |E(|p|)| is the number of bits used to encode |p|, m is the order of Fibonacci encoding.

***Proof*:**

If the string s is compressible, 2*|p| - |E(i)| - |E(|p|)| - |E(j)| > 0. That is, |E(i)| + |E(j)| < 2*p - |E(|p|)|. The left-hand side is the total bits used to encode i and j. The delimiters of |E(i)| and |E(j)| contain m+1 bits. Thus, there are 2*(m+1) bits for delimiters. From Theorem 2, we can imply that i+ j < F(2*|p| - |E(|p|)| - 2*(m + 1)).

3.7 Time Complexity of the Algorithm

We will establish the running time complexity of our algorithm by proving a theorem.

Theorem 4:

The time complexity of the proposed algorithm is bounded by $O(|S| + l|k|^2)$, where $|S|$ is the length of the input sequence, and l is the average length of the repeats and $|k|$ is the maximum number of repeats among all the repeat groups.

Proof:

According to [20], the time complexity to build the suffix tree in Step 1 is bounded by $O(|S|)$. In Step 2, we can use the depth-first search approach to traverse the suffix tree, therefore the time complexity of Step 2 is bounded by $O(|S|)$, too. According to the result in Ref.[19], the time complexity of finding tables Right and Left is bounded by $O(\tau^2)$ where τ is the threshold of edit-distance. The time complexity of using Right[z] and Left[τ-z], $1<z<\tau$, to compute the score of the possible extended repeat is bounded by $O(\tau)$. So, the time complexity of extending an exact repeat into an approximate repeat is bounded by $O(\tau^2)$. Thus, the time complexity of Step 3 is bounded by $O(\rho\tau^2)$ where ρ is the number of exact repeats in S. Since τ is a constant and $\rho <|S|$, the time complexity of Step 3 is bounded by $O(|S|)$. According to Theorem 1, the time complexity of Step 4 is bounded by $O(l|k|^2)$ where l is the average length of the repeats and $|k|$ is the maximum number of overlapping repeats among all the repeat groups. The time complexity of Step 5 is bounded by $O(|S|)$ because the sequence is scanned sequentially from left to right. Therefore, the time complexity of the algorithm is bounded by $O(|S| + l|k|^2)$.

Chapter 4

Experimental Studies

We have proposed a new algorithm for DNA sequence compression and implemented it to compare and rate our algorithm alongside some standard algorithms.

4.1 Experimental Setup

To experiment our algorithm, we tried to compress a standard set of DNA sequences with our algorithm, and we compare with results published for other efficient DNA compressors.

4.1.1 Platform

We implement our algorithm with J2SE 1.4.1. Our programs are run on an Intel® Core™2 Duo personal computer with Windows XP operating systems, 2.0GHz-processor and 2 gigabytes of main memory.

4.1.2 Initialization

The compression ratio is defined as the number of bits /base

In our experiments, we set the threshold τ to be 3. That is, the distance between two approximate repeats is at most 3 edit operations.

4.2 Data set

The table below lists some DNA sequences and their length(number of bases) used in performance analysis.

Table 4-1: Some benchmark DNA sequences

Sequence	Length
CHMPXX	121024
CHNTXX	155844
HEHCMVCG	229354
HUMDYSTROP	33770
HUMGHCSA	66495
HUMHBB	73308
HUMHDABCD	58864
HUMHPRTB	56737
MPOMTCG	186609
PANMTPACGA	100314
VACCG	191737

The DNA sequences of Table 4-1 include complete genomes, genes and viruses. Here are their precise descriptions:

Complete genomes two chloroplasts:

CHMPXX: chloroplast;

CHNTXX: tobacco chloroplast;

Human genes:

HEHCMVCG: human cytomegalovirus strain AD169.

HUMDYSTROP: human dystrophin gene;

HUMGHCSA: human growth hormone and chorionic somatomammotropin genes;

HUMHBB: human beta globin region on chromosome 11;

HUMHDABCD: human DNA sequence of contig comprising 3 cosmids;

HUMHPRTB: human hypoxanthine phosphoribosyltransferase (HPRT).

Complete genome mitochondries:

MPOMTCG: mitochondry of marchantia polymorpha;

The complete genomes of viruses:

VACCG: vaccinia virus;

4.3 Results

Table 4-2: Comparison of compression ratio of our algorithm with other algorithms

		Different DNA Compression Algorithms (Unit : bit/base)					
Sequence	**Size**	**Bio-Com--press2**	**GenCom-press2**	**DNA Residual [21]**	**DNA Pack**	**CTW-LZ**	**Our Algorithm**
CHMPXX	121024	1.68	1.67	1.90	1.66	1.67	1.67
CHNTXX	155844	1.62	1.61	1.97	1.61	1.61	1.61
HEHCMVCG	229354	1.85	1.85	1.99	1.83	1.84	1.85
HUMDYSTR	38770	1.93	1.92	1.97	1.03	1.92	1.91
HUMGHCSA	66495	1.31	1.1	1.98	1.91	1.1	1.03
HUMHDABC	58864	1.88	1.82	1.98	1.74	1.82	1.8
HUMHPRTB	56737	1.91	1.85	1.98	1.78	1.84	1.82
MPOMTCG	186608	1.94	1.91	1.98	1.74	1.9	1.89
MTPACG	100314	1.88	1.86	1.94	1.86	1.86	1.86
VACCG	191737	1.76	1.76	1.96	1.76	1.76	1.76
AVERAGE	-	1.7706	1.7350	1.9650	1.692	1.7320	1.7200

We compare our algorithm's compression ratio with the Bio-Compress2, GenCompress , DNA Residual, CTW-LZ and DNAPack algorithm. The experimental result shows that our algorithm outperforms the BIO-Compress2, GenCompress, DNA Residual and CTW-LZ algorithm by several orders of magnitude.

4.4 Comparison with other algorithms

The graph of figure 4-1 below shows the comparison of compression ratio of Gencompress and our algorithm. The chart shows that in all the cases our algorithm outperforms Gencompress algorithm. From figure 4-2 we can see that our algorithm competes DNAPack in almost 80% cases and goes better than it in some cases.

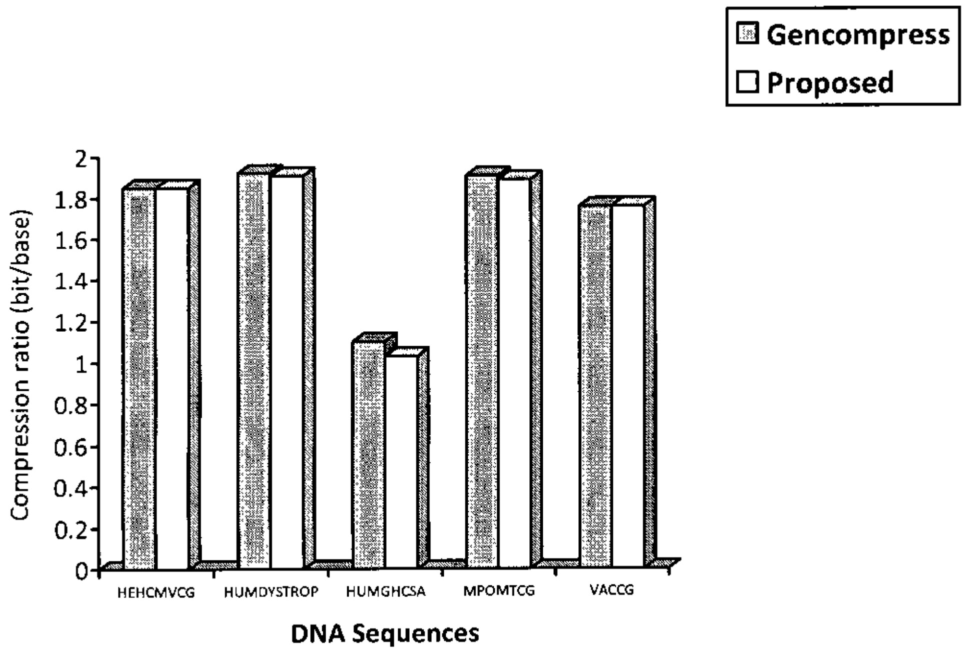


Figure 4-1: Comparison of compression ratio (Gencompress vs proposed algorithm)

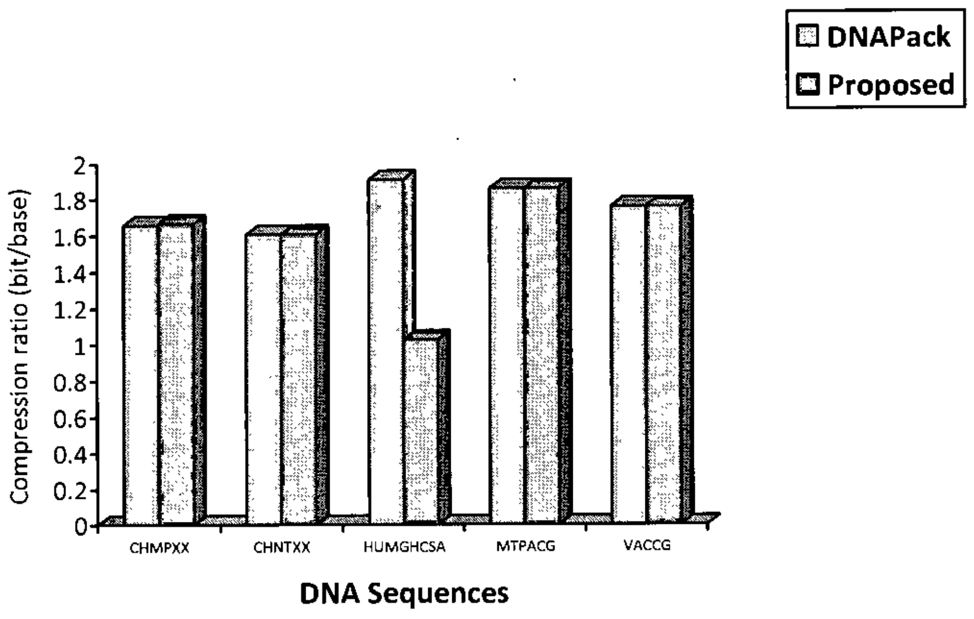


Figure 4-2: Comparison of compression ratio (DNAPack vs Proposed algorithm)

4.5 Running time Analysis

Here, we have constructed the suffix tree for our algorithm offline and it is not included and for GenCompress running time of these sequences are known only. This Gencompress algorithm was simulated on an Intel Pentium III processor 700 MHz and Windows operating system. We compared for these sequences only based on the availability of those results.

Table 4-3: Running time comparison (Gencompress vs proposed algorithm)

Sequence	**Size**	**GenCompress Time (s)**	**Proposed Algorithm Time (s)**
HUMDYSTROP	38770	7	1.7
HUMHPRTB	56737	10.24	3.1
HUMHDABCD	58864	11	3.13
HUMGHCSA	66495	13	9.2
HEHCMVCG	229354	58	13.41
AVERAGE	-	**439**	**6.108**

During our experiments, we compared the running time of our algorithm against the length of the DNA sequences. The graph below plots the time in seconds vs. the sequence length in number of base pairs

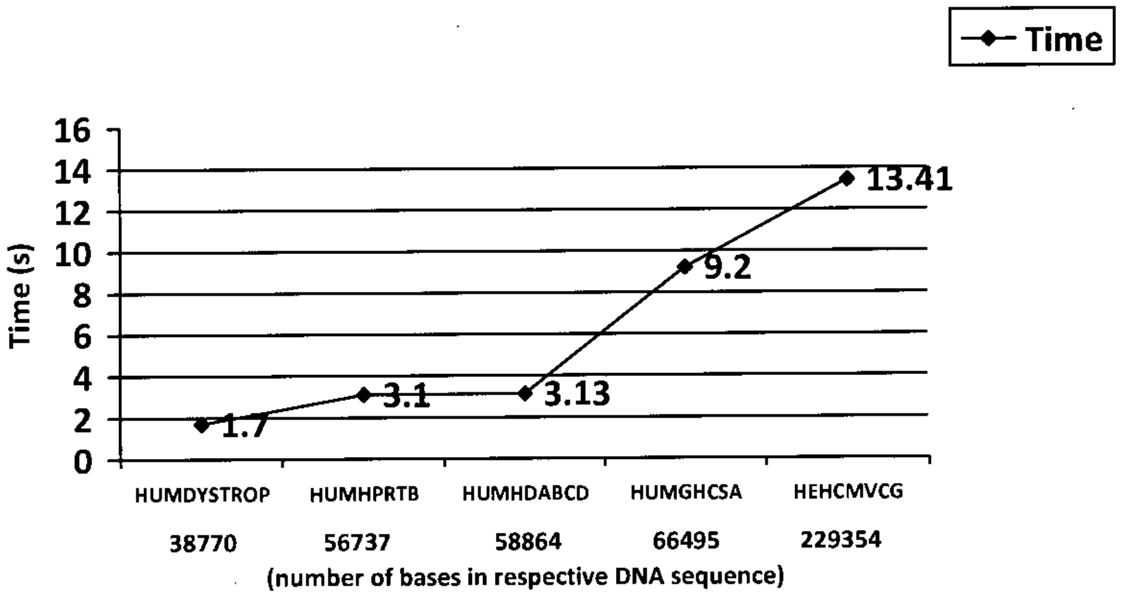


Figure 4-3: Stratagem of running time vs. sequence length

4.6 Discussion

During our experiments, we tried to compress all the sequences while varying a few parameters used by our algorithm:

- We set the threshold τ to be 3. That is, the distance between two approximate repeats is at most 3 edit operations. But for some sequences, 2 or 4 as the value of threshold τ also offer better results.
- the size of the exact prefix that two repeats must have in common to be compared.
- the set of compressions algorithms used to compress BASES, we use arithmetic encoding but Arith-2 or CTW can also be experienced.

Bioinformatics applications (and problems) can be broadly divided into data-intensive and CPU-intensive ones.

The former ones are usually centered on accessing a database and manipulating data in a rather straightforward (from the algorithm's viewpoint) way. In such case there is no reason to deploy a sophisticated and complex algorithm, as the gain in running time would not be balanced from the increase of developing times

On the other hand, CPU-intensive problems ask for highly efficient algorithmic solutions, just consider the (ab initio) protein folding problem, where we would like to compute the tertiary structure of a protein given only its primary structure: the amount of data to be read or output is not very large.

But no algorithm (or program) available nowadays is believed to be the final solution. For this class of problems, there are great benefits for algorithmic improvements, but good implementations are required to transform theoretical improvements into practical ones.

Analyzing the experimental results we bring into being that our algorithm, DNASqueeze outperforms most of the previous algorithms and also trying to beat some of them. Comparing with DNAPack we can institute that it runs better on those sequences which have more and more repeat sequences. It checks each repeat to see whether it saves bits to encode. If not, it will be discarded. Athe end, all the remaining regions other than repeats are concatenated together and then sent as input to a two-order arithmetic coder.

From the plot of running time vs. sequence length it is noticeable that the time grows almost quadratic ally with sequence length and this observation is consistent with our running time analysis. The time complexity of our algorithm is bounded by $O(|S| + l|k|^2)$, where $|S|$ is the length of the input sequence, l is the average length of repeats and $|k|$ is the maximum number of repeats among all the repeat groups.

Chapter 5

Conclusion

Different species have different kinds of mutation and crossover to repeat their DNA segments. Some functional genome segments in one species may repeat itself for many times while the same genome in another species may not. So there is a variety of repetitive structures among different species. Therefore, we have utilized the approximate repeats to compress DNA sequences and propose an efficient lossless compression algorithm.

5.1 Up-shots

The module to locate repeats can be replaced by any other repeat locating strategy to reduce the storage requirement of suffix tree.

Our algorithm go one better than many algorithms.

It is lossless as alteration in DNA sequences while de-compressing is not desired.

Our algorithm works better on sequences with large number of repeats.

(HUMGHCSA). Tested on some random sequences.

Though we have designed the process to use Arithmetic Encoding for non-repeat regions but "Two bit per base '' works as good as it is.

Our algorithm can be used to compress long sequences with millions of bases or more.

5.2 Future Directions

In this paper, we discuss only the compression of DNA sequences by forward repeats. In reality, there are others kinds of repeats such as reverse repeats, complemented repeats and complemented palindromes [22]. How to utilize those repeat structure to compress DNA sequences and obtain the better compression ratio is worth studying the in the near future. Besides, how to apply our proposed method to compress protein sequences, where the size of the alphabets is larger than that of DNA sequences, is also worth studying.

BIBLIOGRAPHY

1. P. Levene, "The structure of yeast nucleic acid". J Biol Chem, 01 December 1919. 40 (2): pp-415–24.

2. W. Astbury , "Nucleic acid". Symp. SOC (1947). Exp. Bbl 1 (66).

3. MG. Lorenz, W. Wackernagel , "Bacterial gene transfer by natural genetic transformation in the environment". Microbiol. Rev.(01 September 1994). 58 (3): pp-563–602.

4. O. Avery, C. MacLeod, M. McCarty, "Studies on the chemical nature of the substance inducing transformation of pneumococcal types. Inductions of transformation by a desoxyribonucleic acid fraction isolated from pneumococcus type III". J Exp Med (1944). 79 (2): pp-137–158.

5. A. Hershey, M. Chase, "Independent functions of viral protein and nucleic acid in growth of bacteriophage" (PDF). J Gen Physiol (1952). 36 (1): pp-39–56.

6. J.D. Watson and F.H.C.Crick, "A Structure for Deoxyribose Nucleic Acid". 1953, Nature 171, pp- 737-738.

7. The B-DNA X-ray pattern was obtained by Rosalind Franklin and Raymond Gosling in May 1952 at high hydration levels of DNA and it has been labeled as "Photo 51".

8. K.G. Srinivasa, M. Jagadish, K.R.Venugopal and L.M. Patnaik, "Non-repetitive DNA sequence compression using Memoization" ISBMDA 2006 , LNBI 4345, pp-402-412, Springer-Verlag Berlin Heidelberg 2006.

9. X.Chen, S. Kwong and M.Li, "A compression algorithm for DNA Sequences and its applications in genome comparison ", Genome Informatics , 1999,10: pp-51-61.

10. K. Sadakane , and H. Imai, "Improving the speed of LZ77 compression by hashing and suffix sorting, submitted.

11. S. Grumbach and F. Tahi, ”Compression of DNA Sequences. In Data compression conference”, IEEE Computer Society Press, 1993.pp-340-350.

12. S. Grumbach and F. Tahi, “A new Challenge for compression algorithms: geneticsequences”. Journal of Information Processing and Management,1994, 30, pp-875-866.

13. F.M.J. Willems, Y.M. Shtrakov and T.J. Tjalkens, ”The Context Tree Weighting Method: Basic Properties”. IEEE Trans. Inform. Theory,1995, IT-41(3), pp-653-664.

14. X. Chen, S. Kwong, M. Li, “A compression Algorithm for DNA sequences”. IEEE Engineering in Medicine and Biolgoy Magazine, Jul/Aug 2001, 20(4), pp-61-66.

15. M. Li., J.H. Badger, X. Chen., S. Kwong, P. Kearney, H. Zhang, “An information based sequences distance and its application to whole motochondrial genome phylogeny,” Bioinformatics, 2001, 17(2): pp-149-154.

16. T. Matsumuto, K. Sadakane, H. Imai, “Biological sequence compression algorithms”. Genome Inform. Ser. Wokrshop Genome Inform, 2000. 11: pp-43-52.

17. X. Chen, M. Li, B. Ma, B. and J. Tromp, “DNACompress: fast and effective DNA sequence compression”. Bioinformatics, 2002, 18: pp-1696-1698.

18. A. Apostolico and A.S. Fraenkel, “Robust transmission of unbounded strings using Fibonacci representations”. IEEE trans. inform, 1987, 33(2), pp- 238-245.

19. S. Kurtz, E. Ohlebusch, C. Schleiermacher, “Computation and visualization of degenerate repeats in complete genomes”. Intelligent Systems for Molecular Biology , 2000, pp-228-238.

20. E. Ukkonen, “On-line construction of suffix trees”. Algorithmica 14, 1995, pp- 249-260.

21. R. Ferzli and L. J. Karam, "DNA-RESIDUAL: A DNA Compression Algorithm Using Forward Linear Prediction," IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), May 2006.

22. A. Delcher, S. Kasif, R. Fleischmann, J. Peterson, O. White, S. Salzberg, "Alignment of whole genomes". Nucleic Acids Research, 1999, 27 (11), pp- 2369-2376.