DNA Sequences Compression Using Exact and Approximate Matching

A Thesis
Submitted To the Council of College of Science
University of Babylon
In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

By

Mohammed U. Mahdi Al-Juboori

December-2005 **Dhulqa'da-1426**

بسم الله الرحمن الرحيم

﴿وَأَنْ لَّيسَ لِلإِنسانِ إِلَّا مَا سَعَى* وَأَنَّ سَعيَهُ سَوفَ يُرَى* ثُمَّ يُجزاهُ الجَزَاءَ الأَوفَى﴾

صدق الله العلي العظيم

سورة النجم (39- 41)

Supervisor Certification

I certify that this thesis was prepared under my supervision at the department of computer science /college of science / Babylon University, by **Mohammed U. Mahdi** as partial fulfillment of the requirements for the degree of Master of Science in computer science.

Signature:
Name: **Dr. Tawfiq A. Abbas**
Title: **Professor Assistant**
Date: / 1 / 2006

In view of the available recommendations, I forward this thesis for debate by the examination committee.

Signature:
Name: **Dr. Abbas M. Albakry**
Title: Head of Computer Science Department
Date: / 1 / 2006

Certification of the Examination Committee

We chairman and members of the examination committee, certify that we have studied the thesis entitled (**DNA Sequences Compression Using Exact and Approximate Matching**) presented by the student **Mohamed U. Mahdi** and examined him in its content and in what is related to it, and we have found it worthy to be accepted for the degree of Master of Science in Computer Science with (**Excellent**) degree.

Signature
Name: **Dr. Nabeel H. Kaghed**
Title: **Professor**
Date: **/1/2006**
(Chairman)

Signature
Name: **Dr**. **Sattar B. Sadkhan**
Title: **Professor**
Date: **/1/2006**
(Member)

Signature
Name: **Esra'a H. ali**
Title: **professor Assistant**
Date: **/1/2006**
(Member)

Signature:
Name: **Dr. Tawfiq A. Abbas**
Title: **Professor Assistant**
Date**:** **/1/ 2006**
(Supervisor)

Signature
Name: **Dr**. **Oda Mizi'l Yasser Alzamely**
Title: **Professor**
Date**:** **/1/2006**
(Dean of College of Science – Babylon university)

DEDICATION

To

My Family

My Friends

M Department's Staff

ACKNOWLEDGEMNT

Its pleasure to acknowledge my debt for many people involved in presenting this thesis: First of all I would like to express my sincere gratitude and appreciation to my supervisor Dr. Tawfiq Alasadi for his invaluable guidance, supervision and untiring efforts during the course of this work. Thanks are to my close friend Mahdi A. Salman. Special thanks go to my wife and family for their continuous support and encouragement during the period of my studies. Finally, I would like to thank the staff of the department of computer science for their great help they've introduced to me.

Abstract

A model has been described for sequences of characters that is lossless based on regularities of strings. In addition to that a repeated subsequence can be an Exact and Approximate match to the original subsequence; this is close to the situation of DNA. It is well-known that the compression of DNA sequences is very difficult problem. Since DNA sequences only contain the four bases {a, c, g, t} they can be stored using two bits per input symbol. The standard compression tools, such as zip, lzw, usually fail to achieve any compression since they use more than two bits per symbol.

Proposed algorithm achieves a compression ratio very close to the best DNA compressors. Proposed algorithm can be summarize as two stage: The first stage is to extract data base or a references using GA. The second stage it proceeds to search exact or approximate matching sequence and encode it economically.

The model has a small number of parameters and these can be estimated from the given string. The primary purpose of the work is not only to compress sequences, and in particular DNA sequences; so as to save computer storage or to reduce data transmission costs, but also the purpose is to model the statistical properties of structure within them.

LIST OF CONTENTS

LIST OF ABBRIVIATIONS

ARM	**Approximate Repeat Matching**
DCT	**Discrete Cosine transform**
DFT	**Discrete Fourier Transform**
DNA	**Deoxyribonucleic acid**
ERM	**Exact Repeat Matching**
GA	**Genetic Algorithm**
GCAK	**Genetic Clustering A Known K**
GIF	**Graphic Interchange Format**
JPEG	**Joint Photographic Experts Groups**
LZW	**Lemple Ziv and Welch**
MPEG	**Moving Pictures Experts Group**
NP Problem	**Non Deterministic Polynomial Problem**
PPM	**Prediction by Partial Matching**
RNA	**Ribonucleic acid**
RLE	**Run Length Encoding**
RWS	**Roulette wheel Selection**
SNP	**Single nucleotide polymorphism**
SQ	**Scalar Quantization**
SSR	**Simple sequence repeat**
TR	**Tandem repeat**
tRNA	**Transfer RNA**
VNTR	**Variable number tandem repeat**
VQ	**Vector Quantization**

LIST OF FIGURES

LIST OF TABLES

1.1 Introduction

Compression is used just about everywhere. All the images you get on the Web are compressed, typically in the JPEG or GIF format, most modems use compression and several file systems automatically compress files when stored, and most of us do it by hand [16].

In this chapter we will use the generic term *message* for the objects we want to compress, which could be either files or messages. The task of compression consists of two components, an *encoding* algorithm that takes a message and generates a "compressed" representation (hopefully with fewer bits), and a *decoding* algorithm that reconstructs the original message or some approximation of it from the compressed representation. These two components are typically intricately tied together since they both have to understand the shared compressed representation[5].

We distinguish between *lossless algorithms*, which can reconstruct the original message exactly from the compressed message, and *lossy algorithms*, which can only reconstruct an approximation of the original message. Lossless algorithms are typically used for text, and lossy for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable. Lossy is used in an abstract sense, however, and does not mean random lost pixels, but instead it means loss of a quantity such as a frequency component, or perhaps loss of noise. For example, one might think that lossy text compression would be unacceptable because one may imagine missing or switched characters[16]. Consider instead a system that reworded sentences into a more standard form, or replaced words with synonyms so that the file can be better compressed. Technically the compression would be lossy since the text has changed, but

the "meaning" and clarity of the message might be fully maintained, or even improved.

Is there a lossless algorithm that can compress all messages? This is not possible, at least if the source messages can contain any bit-sequence.

Because one can't hope to compress everything, all compression algorithms must assume that there is some bias on the input messages so that some inputs are more likely than others, *i.e.* that there is some unbalanced probability distribution over the possible messages. Most compression algorithms base this "bias" on the structure of the messages *i.e.*, an assumption that repeated characters are more likely than random characters, or that large white patches occur in "typical" images. Compression is therefore all about probability.

When discussing compression algorithms it is important to make a distinction between two components: the model and the coder. The *model* component somehow captures the probability distribution of the messages by knowing or discovering something about the structure of the input. The *coder* component then takes advantage of the probability biases generated in the model to generate codes. It does this by effectively lengthening low probability messages and shortening high-probability messages. A model, for example, might have a generic "understanding" of human faces knowing that some "faces" are more likely than others. The coder would then be able to send shorter messages for objects that look like faces. This could work well for compressing teleconference calls. The models in most current real-world compression algorithms, however, are not so sophisticated, and use more mundane measures such as repeated patterns in text [16]. Although there are many different ways to design the model component of

compression algorithms and a huge range of levels of sophistication, the coder components tend to be quite generic—in current algorithms are almost exclusively based on either Huffman or arithmetic codes. Let us try to make a fine distinction here, it should be pointed out that the line between model and coder components of algorithms is not always well defined.

It turns out that information theory is the glue that ties the model and coder components together. In particular, it gives a very nice theory about how probabilities are related to information content and code length. As we will see, this theory matches practice almost perfectly, and we can achieve code lengths almost identical to what the theory predicts.

Another question about compression algorithms is how one judge the quality of one versus another. In the case of lossless compression, there are several criteria, the time to compress, the time to reconstruct, the size of the compressed messages, and the generality [8]. In the case of lossy compression the judgment is further complicated since we also have to worry about how good the lossy approximation is. There are typically tradeoffs between the amount of compression, the runtime, and the quality of the reconstruction. Depending on the application, one might be more important than another and one would want to pick your algorithm appropriately.

1.2 Information Theory

The importance of information theory is that it quantifies information. It shows how to measure information, so that we can answer the question how much information is included in this piece of data [8, 16].

Another approach to the same problem is to ask the question "given a nonnegative integer N. the greater N, the more digits are necessary. The first 100 non-negative integers (0 to 99) can be expressed by two decimal digits. The first 1000 such integers can be expressed by three digits. Again it does not take long to see the connection. The number of digits required to represent N equals approximately log N. The base of the logarithm is the same as the base of the digits. For decimal digits, use base 10; for binary ones (bits) use base 2. If we agree that the number of digits it takes to express N equals the information contents of N, then again the logarithm is the function that uses a measure of the information.

1.2.1 Entropy

Shannon borrowed the definition of *entropy* from statistical physics to capture the notion of how much information is contained in a piece of data and their probabilities [5, 16]. For a set of possible messages, Shannon defined entropy [32] as,

$$H(S) = \sum_{s \in S} p(s) \log_2 \frac{1}{p(s)}. \tag{1.1}$$

Where $p(s)$ is the probability of messages. The definition of Entropy is very similar to that in statistical physics- in physics S is the set of possible states a system can be in its data and p(s) is the probability the system is in state *s*. We might remember that the second law of thermodynamics basically says that the entropy of a system and its surroundings can only increase.

Getting back to messages, if we consider the individual messages $s \in S$, Shannon defined the notion of the *self information* of a message as

$$i(s) = \log_2 \frac{1}{p(s)}. \tag{1.2}$$

This self information represents the number of bits of information contained in it and, roughly speaking, the number of bits we should use to send that message. The equation says that messages with higher probability will contain less information.

The entropy is simply a weighted average of the information of each message, and therefore the average number of bits of information in the set of messages. Larger entropies represent more information, and perhaps counter-intuitively, the more random a set of messages are (the more even the probabilities) the more information they contain on average.

1.3 Probability Coding

As mentioned in the introduction, coding is the job of taking probabilities for messages and generating bit strings based on these probabilities. How the probabilities are generated is part of the model component of the algorithm.

In practice we typically use probabilities for parts of a larger message rather than for the complete message, *e.g.*, each character or word in a text. To be consistent with the terminology in the previous section, we will consider each of these components a message on its own, and we will use the term *message sequence* for the larger message made up of these components. In general each little message can be of a different type and come from its own probability distribution. For example, when sending an image we might send a message specifying a color followed by messages specifying a frequency component of that color. Even the messages specifying the color might come from different probability distributions since the probability of particular colors might depend on the context.

We distinguish between algorithms that assign a unique code (bit-string) for each message, and ones that "blend" the codes together from more than one message in a row. In the first class we will consider Huffman codes, which are a type of prefix code. In the latter category we consider arithmetic codes. The arithmetic codes can achieve better compression, but can require the encoder to delay sending messages since the messages need to be combined before they can be sent [5].

1.3.1 Applications of Probability Coding

To use a coding algorithm we need a model from which generate probabilities. Some simple models are to count characters for text or pixel values for images and use these counts as probabilities. Such counts, however, would only give a compression ratio of about 4.7 / 8 = .59 for English text as compared to the best compression algorithms that give ratios of close to 0.2 [16].

An issue to consider about a model is whether it is static or dynamic as shown in Figure (1.1). A model can be static over all message sequences. For example one could predetermine the frequency of characters and text and "hardcode" those probabilities into the encoder and decoder. Alternatively, the model can be static over a single message sequence. The encoder executes one pass over the sequence to determine the probabilities, and then a second pass to use those probabilities in the code. In this case the encoder needs to send the probabilities to the decoder. This is the approach taken by most vector quantizers. Finally, the model can be dynamic over the message sequence. In this case the encoder updates its probabilities as it encodes messages. To make it possible for the decoder to determine the

probability based on previous messages, it is important that for each message, the encoder codes it using the old probability and then updates the probability based on the message. The advantages of this approach are that the coder need not send additional probabilities, and that it can be adapted to the sequence as it changes. This approach is taken by PPM.

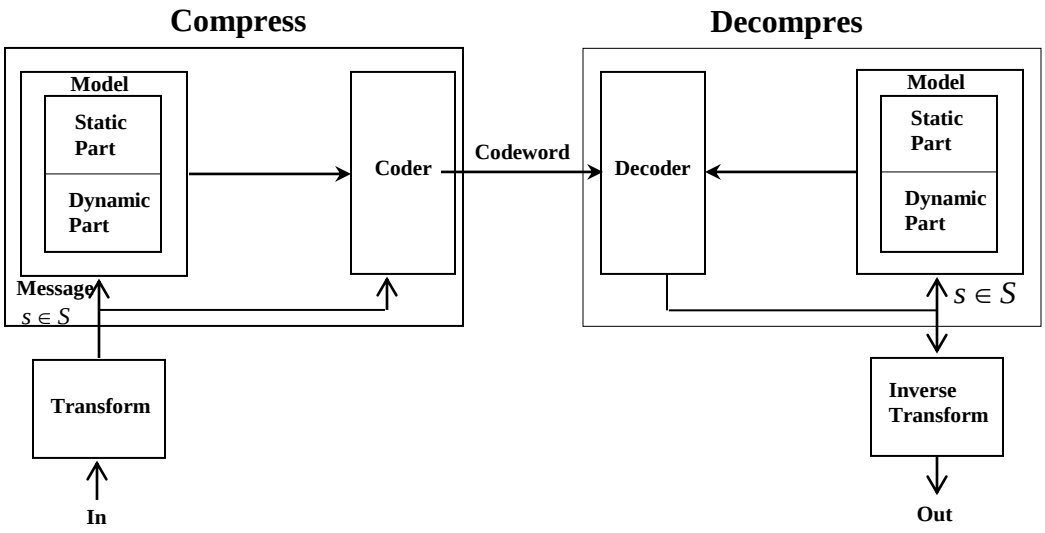


Figure (1.1): The General Framework of a Model and Coder [16]

1.4 Prefix Codes

A code C for a message set S is a mapping from each message to a bit string. Each bit string is called a codeword, and we will denote codes using the syntax $C = \{(s_1, w_1), (s_2, w_2), ..., (s_m, w_m)\}$. Typically in computer science we deal with fixed-length codes, such as the ASCII code which maps every printable character and some control characters into 7 bits [5, 8, 16]. For compression, however, we would like code words that can vary in length to be based on the probability of the message. Such variable length codes have the potential problem that if we are sending one codeword after the other it can be hard or impossible to tell where one codeword finishes and the next

starts. For example, given the {a, 1), (b, 01), (c, 101), (d, 011)}, the bit-sequence 1011 could either be decoded as aba, ca, or ad. To avoid this ambiguity we could add a special stop symbol to the end of each codeword (e.g., a 2 in a 3-valued alphabet), or send a length before each symbol. These solutions, however, require sending extra data. A more efficient solution is to design codes in which we can always uniquely decipher a bit sequence into its code words. We will call such codes uniquely decodable codes.

A prefix code is a special kind of uniquely decodable code in which no bit-string is a prefix of another one, for example {a, 10), (b, 01), (c, 000), (d, 111)}. All prefix codes are uniquely decodable since once we get a match, there is no longer code that can also match.

1.5 Data Compression Strategies

There are different ways that data compression techniques can be categorized, smith [36] gives a compression classification as below:

a. Lossless or Lossy

Lossless	Lossy
RLE	JPEG
Huffman	MPEG
Arithmetic	Vector quantization

b. Fixed or variable group size

Method	Group Size	
	Input	Output
Huffman	Fixed	Variable
Arithmetic	Variable	Variable
RLE, LZW	Variable	Fixed
Proposed system	variable	variable

Most data compression programs operate by taking a group of data from the original file and compressed it in some way, and then writing the compressed group to the output file.

1.6 DNA Compressibility

In this section, we study one basic question: the compressibility of DNA sequences. Life represents order. It is not chaotic or random. Thus, we expect the DNA sequences that encode life to be nonrandom [4]. 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 contain many tandem repeats; it is also well-known that many essential genes have many copies; it also has been conjectured that genes duplicate themselves sometimes for evolutionary or simple for "selfish" purposes. All this give more concrete support that DNA sequences should be reasonably compressible [11, 13]. Our purpose is to study such subtleties in DNA sequences. Almost all known biological functions can be traced to the sequence of nucleotides or bases known as DNA that is found in the cells of every organism. The base sequence in the DNA is transcribed one-to-one in the cell to another sequence of nucleotides known as RNA (whose bases are identical to the bases of DNA with one small difference) [6, 7]. The latter in the presence of another of catalysts is translated into a set of proteins, which are sequences of amino acids that contribute to a host of cellular higher-level functions of the organism.

DNA’s alphabet is the set {t, c, a, g} (corresponding to the bases thymine, cytosine, adinine, and guanine) [9, 13], while RNA’s is {u, c, a, g}

(corresponding to the bases uracil, cytocine, adinie, and guanine). The protein alphabet is a set of 20 amino acids {a, c, d, e, f, g, h, i, l, m, n, p, q, r, s, t, u, v, w, y}, where the letters in the set are the 1-letter codes for alanine, cysteine, aspartic acid, glutamic acid, phenylalanine, glycine, histidine, isoleucine, lysine, methionine, leucine, asparagines, praline, glutamine, arginine, serine, threonine, valine, trypotophan, and tyrosine.

There are two approaches for compressing DNA sequences:-

A. exact matching

Ex. aagtacagtacagt

Can be written as aagtacac (3, 5)gt where 3 is positon and 5 is length

B-approximate matching

Ex: s_1 gaccgtca

s_2 gaccggca can be written as (R, 6, g)

That means replace six location by g

1.7 Related works

In the first we analyze the outcome of classical text compression algorithms on a sample of DNA sequences.

The probability of each character depends on the previous characters. This motivates the use of higher order models of encoding. The order of a model represents the number of previous symbols which are considered in the computation of the probabilities. Unless statistical methods, all the substituational methods lead to negative compression rates [15]. Ziv and Lempel proposed two substituational methods where words are replaced by a pointer to one of their previous occurrence. In the first one, the *LZ77* algorithm, the occurrences of the factor to encode are searched in a window.

In the second one, the *LZ78* algorithm, a dictionary containing the already encoded factors is used.

Different extensions were proposed to the algorithm *LZ77* and *LZ78*. Among others, the methods *LZSS* (extension of *LZ77*), and *LZW* and *compress* (both extensions of *LZ78*) are tested on DNA sequences, the compression rates are all negative [15].

The Milsoavljevic and Jurka Method [17, 32]. This method takes a DNA sequence and splits it up into windows of fixed length n. These windows overlap by an amount v. Both n and v are parameters to the algorithm.

It considers each window in turn independently of the rest of the sequence. Each window is encoded using 5 code-words one each for A, T, G, C and a pointer P is used to encode a previously occurring subsequence. The code- word is fixed to be of length $\log_2^5$ [4, 6].

Encoding a pointer consists of a pointer code-word, followed by a starting position of the previous run and a length of that run. The starting position and length are each encoded in $\log_2 n$ bits, where n is the size of the window.

Each window is encoded separately, and if this encoding is better than the null encoding ($\log_2 4 = 2$ bits per base) by a fixed threshold, then the window is deemed to be significant. This threshold is also a parameter to the algorithm.

Grumbach and Tahi [14,3] proposed two lossless compression algorithms for DNA sequences, namely Biocompress and Biocompress-2, in the spirit of Ziv and Lempel data compression method. In fact, the difference

between Biocompress and Biocompress-2 is the addition of order-2 arithmetic coding.

The algorithm *biocompress-2* is designed for the compression of DNA sequences. It can also be used on RNA sequences but not on amino-acid sequences of proteins. It encodes a text on the four letter alphabet {a,c,g,t} into a binary sequence. Biocompress-2 is an extension of the algorithm biocompress. It combines a substitutional method, essentially in the spirit of the first approach of Ziv and Lempel's encoding, and a statistical method, the arithmetic encoding is used.

There are specific redundancies in DNA sequences. Such repetitions do not exist in texts in natural or programming languages. *Biocompress-2* detects and encodes them.

Xin Chen, Sam kwong and Ming Li [40] presented a lossless compression algorithm, Gen Compress, for genetic sequences, based on searching for Approximate repeats. Using Hamming Distance for measuring the irrelativeness between two DNA sequences.

É. Rivals et al. [33] give another compression algorithm Cfact, which searches the longest exact matching repeat using suffix tree structure in an sequence. The idea of Cfact is basically the same as Biocompress-2 except the Cfact is a two-pass algorithm. It builds the suffix tree in the first pass. In the encoding phase, the repetitions are coded with guaranteed gain; otherwise, two-bit per base encoding will be used.

1.8 Aim of Research

This work is aim to suggest a new compressions system to compress DNA sequences to save computer storage, reduced data transmission and model the statistical properties of DNA structure

1.9 Thesis Layout

Thesis has been structured as five chapters:

* Chapter two shows some of the compression techniques.
* Chapter three shows genetic data and genetic algorithm.
* Chapter four (practical section) shows suggested system with some examples.
* Chapter five shows proposed system results, conclusions and future works.

2.1 Introduction

In this chapter we will give some examples of more sophisticated models. All these techniques take advantage of the "context" in some way. This can be done either by transforming the data before coding (*e.g.*, run-length coding, move-to-front coding, and residual coding), or directly using conditional probabilities based on a context (PPM).

2.2 Lossless Methods

In this chapter we will explain some lossless compression techniques.

2.2.1 Huffman Codes

Huffman codes are optimal prefix codes generated from a set of probabilities by a particular algorithm, the Huffman Coding Algorithm [16]. David Huffman developed the algorithm as a student in a class on information theory in 1950. The algorithm is now probably the most prevalently used component of compression algorithms, used as the back end of GZIP, JPEG and many other utilities.

The method starts by building a list of all the alphabet symbols in descending order of their probabilities. It then constructs a tree with a symbol at every leaf, from the bottom up. This is done in steps where, at each step, the two symbols with smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing both of them. When the list, is reduced to just one auxiliary symbol (representing the entire alphabet) the tree is complete. The tree is then traversed to determine the codes of the symbols.

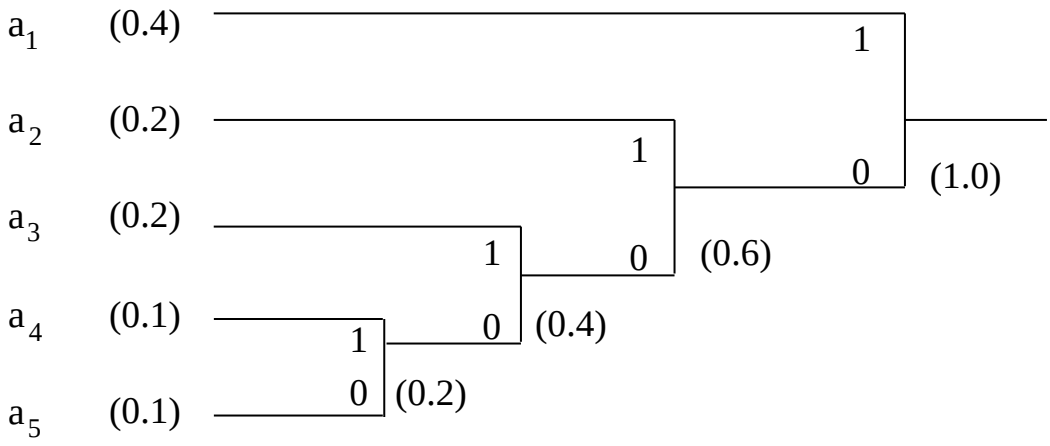
Example 1:

Given five symbols with probabilities as shown in Figure (2.1-a), to encode them. They are paired in the following order:

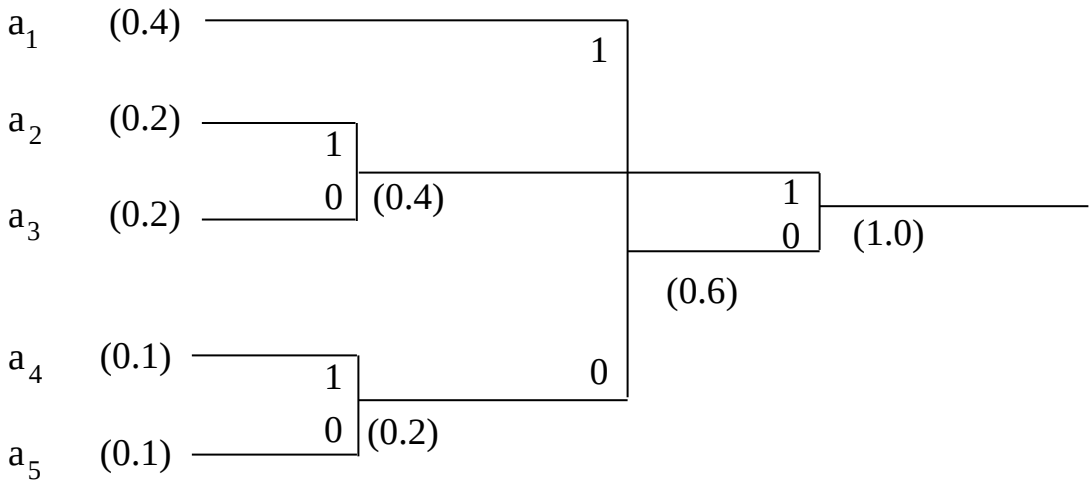
1. a_4 is combined with a_5 and both are replaced by the combined symbol a_{45} whose probability is 0.2.
2. There are now four symbols left a_1, with probability 0.4, and a_2, a_3 and a_{45}, with probabilities 0.2 each. We arbitrarily select a_3 and a_{45}, combine them and replace with auxiliary symbol a_{345} whose probability is 0.4.
3. Three symbols are now left, a_1, a_2 and a_{345}, with probabilities o.4, 0.2 and 0.4, respectively. We arbitrarily select a_2 and a_{345}, Combine them and replaced them with the auxiliary symbol a_{2345} whose probability is 0.6.
4. Finally, we combine the two remaining symbols, a_1 and a_{2345}, and replace them with a_{12345} with probability 1.

The tree is now complete. To assign the codes, we arbitrarily assign a bit of 1 to the top edge, and a bit of 0 to the bottom edge, of every pair of edges. This results in the codes 1, 10, 100, 1000, and 0000. The assignments of bits to the edges are arbitrary but this must be consistent.

The Huffman code is not unique. Some of the steps above were selected arbitrarily, since there were more than two symbols with smallest probabilities. Figure (2.1-b) shows how the same five symbols can be combined differently to obtain a different Huffman code.



(a)



(b)

Figure (2.1): Huffman Codes

Before starting the compression of a data stream, the compressor (encoder) has to determine the codes. It based on the probabilities (or frequencies of occurrence) of the symbols. The probabilities of frequencies have to appear on the compressed stream, so that any Huffman decompressor (decoder) will be able to decompress the stream. This is easy since the frequencies are integers and the probabilities can be written as scaled integers.

It normally adds just a few hundred bytes to the compressed stream. The decoder must know what is at the start, read it, and construct the Huffman tree for the alphabet. Then it can be read and decode the rest of the stream. The algorithm of decoding is simple. Start at the root and read the first bit of the compressed stream. If it is zero follow the bottom edge, if it is one, follow the top edge. Read the next bit and move another edge toward the leaves of the tree.

When the decoder gets to a leaf, it finds the original, uncompressed code of the symbol and that code is emitted by the decoder. The process starts again at the root with the next bit.

2.2.2 Arithmetic Coding

The main steps of arithmetic coding are as follows[5, 8, 16]:-

1. Start by defining the "current interval" as [0,1).
2. Repeat the following two steps for each symbol S in the input stream:
   2.1 Divide the current interval into subintervals whose sizes are proportional to the symbols probabilities.
   2.2 Select the subinterval for S and define it as the new current interval.
3. When the entire input stream has been processed in this way, the output should be any number that uniquely identifies the current interval.

The symbols and probabilities are written on the output stream before any of the bits of the compressed code, and this will be the first thing input by the decoder. The encoding process starts by defining two variables LOW and HIGH which define an interval [LOW, HIGH) as follow:-

$$LOW=LOW+RANGE \times LOWRANGE(X)$$

$$HIGH=LOW+RANGE \times HIGHRANGE(X)$$

Where RANGE=HIGH-LOW, LOWRANGE(X) and HIGHRANGE(X) indicate the low and high limits of the range of symbol X, respectively. The method reads the input stream symbol by symbol and appends more bits to the code each time a symbol is input and processed.

Example:

Given three symbols a_1, a_2 and a_3 with probabilities P1 = 0.4, P2=0.5 and P3 = 0.1, respectively. The interval [0 ,1) is divided among the three symbols by assigning each subinterval proportional in size to its probability. The three symbols are assigned the subintervals [0, 0.4), [0.4,0.9), and [0.9, 1). To encode the string "$a_2\ a_2\ a_2\ a_3$", we start with the interval [0,1).

The first symbol a_2 reduces [0,1) to [0.4, 0.9) as follows:

Low = 0 + (1-0) × 0.4 = 0.4, **High** = 0+ (1-0) × 0.9 = 0.9

The second symbol a_2 reduces [0.4, 0.9) to [0.6, 0.85) as follows:

Low = 0.4 + (0.9 – 0.4) × 0.4 = 0.6, **High** = 0.4 + (0.9 – 0.4) × 0.9 =0.85

The third symbol a_2 reduces [0.6, 0.85) to [0.7, 0.825) as follows:

Low = 0.6 + (0.85 – 0.6) × 0.4 = 0.7 **High**=0.6+ (0.85-0.6) × 0.9 = 0.825

The fourth symbol a_3 reduces [0.7, 0.825) to [0.8125, 0.8250), and this is the final code as follows:

Low = 0.7+(0.825-0.7) 0.9=0.8125 **High** = 0.7+(0.825-0.7) × 1=0.825

2.2.3 Run-length Coding

Probably the simplest coding scheme that takes advantage of the context is run-length coding. Although there are many variants, the basic idea is to identify strings of adjacent messages of equal value and replace

them by a single occurrence along with a count [16]. For example, the message sequence acccbbaaabb could be transformed to (a,1), (c,3), (b,2), (a,3), (b,2). Once transformed, a probability coder (*e.g.*, Huffman coder) can be used to code both the message values and the counts.

An example of a real-world use of run-length coding is for Facsimile (fax) machines [5,16]. At the time of writing (1999), this was the standard for all home and business fax machines used over regular phone lines.

2.2.4 Move-To-Front Coding

Another simple coding scheme that takes advantage of the context is the move-to-front coding. This is used as a sub-step in several other algorithms including the Borrows-Wheeler algorithm [26]. The idea of move-to-front coding is to preprocess the message sequence by converting it into a sequence of integers[8, 16], which is hopefully biased toward integers with low values. The algorithm then uses some form of probability coding to code these values. In practice the conversion and coding are interleaved, but we know that each message comes from the same alphabet, and starts with a total order on the alphabet (*e.g.*, [*a, b, c, d, …*]). For each message, the first pass of the algorithm outputs the position of the character in the current order of the alphabet, and then updates the order so that the character is at the head. For example, coding the character *c* with an order [*a, b, c, d, …*] would output a 3 and change the order to [*c, a, b, d, …*]. This is repeated for the full message sequence. The second pass converts the sequence of integers into a bit sequence using Huffman or Arithmetic coding.

The hope is that equal characters often appear close to each other in the message sequence so that the integers will be biased to have low values. This will give a skewed probability distribution and good compression.

2.2.5 Residual Coding

Residual compression is another general compression technique used as a sub-step in several algorithms. As with move-to-front coding, it preprocesses the data so that the message values have a better skew in their probability distribution, and then codes this distribution using a standard probability coder. The approach can be applied to message values that have some meaningful total order (*i.e.*, in which being close in the order implies similarity), and is most commonly used for integers values. The idea of residual coding is that the encoder tries to guess the next message value based on the previous context and then outputs the difference between the actual and guessed value [28]. This is called the *residual*. The hope is that this residual is biased toward low values so that it can be effectively compressed. Assuming that the decoder has already decoded the previous context, it can make the same guess as the coder and then use the residual it receives to correct the guess. By not specifying the residual to its full accuracy, residual coding can also be used for lossy compression. Residual coding is used in JPEG [30].

2.2.6 Context Coding: PPM

Over the past decade, variants of this algorithm have consistently given either the best or close to the best compression ratios [22].

The main idea of PPM (Prediction by Partial Matching) is to take

advantage of the previous K characters to generate a conditional probability of the current character [28, 30]. The simplest way to do this would be to keep a dictionary for every possible string of characters, and for each string have counts for every character *x* that follows *s*. The conditional probability of *x* in the context *s* is then $C(x|s)/C(s)$, where *C*(*x*|*s*) is the number of times *x* that follows *s* and C(s) is the number of times *s* appearence. The probability distributions can then be used by a Huffman or Arithmetic coder to generate a bit sequence. For example, we might have a dictionary with qu appearing 100 times and *e* appearing 45 times after qu. The conditional probability of the e is then 0.45.

Table (2.1): An Example of the PPM Table for *k=2* on the String accbaccacba

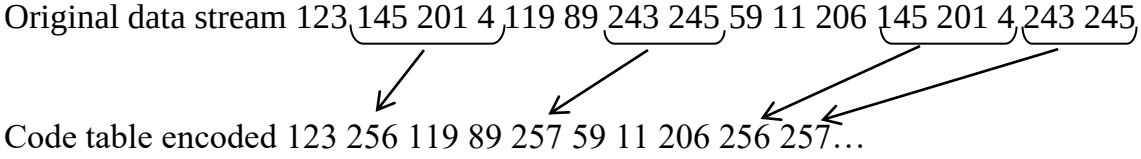
Order 0		Order 1		Order 2	
Context	**Counts**	**Context**	**Counts**	**Context**	**Counts**
Empty	**a= 4**	**a**	**c=3**	**ac**	**b=1**
	b=2				**c=2**
	c=5	**b**	**a=2**		
				ba	**c=1**
		c	**a=1**		
			b=2	**ca**	**c=1**
			c=2		
				cb	**a=2**
				cc	**a=1**
					b=1

2.2.7 LZW Compression

LZW compression is named after its developers, A. Lempel and J. Ziv, with later modifications by Terry A. Welch [17]. It is the foremost technique for general purpose data compression due to its simplicity and versatility. Typically, you can expect LZW to compress text, executable code, and similar data files to about one-half their original size. LZW also performs well when presented with extremely redundant data files, such as tabulated numbers, computer source code, and acquired signals [23, 41]. Compression ratios of 5:1 are common for these cases [38]. LZW is the basis of several personal computer utilities that claim to 'double the capacity of your hard drive. "LZW compression is always used in GIF image files.

Table (2.2) Code Table

Code numbers	translation
0000	**0**
0001	**1**
. . .	. . .
0254	**254**
0255	**255**
0256	**256**
0257	**257**
. . .	. . .
4095	**xxx**



This is the basis of the popular LZW compression method. Encoding occurs by identifying sequences of bytes in the original file which exists in the code table. The 12 bit code representing the sequence is placed in the compressed file instead of the sequence. The first 256 entries in the table correspond to the single byte values, 0 to 255, while the remaining entries correspond to sequences of types. The LZW algorithm is an efficient way of generating the code table based on the particular data being compressed (the code table in this figure is a simplified example, not one actually generated by the LZW algorithm).

2.3 Lossy Compression Techniques

Lossy compression is compression in which some of the information from the original message sequence is lost. This means that the original sequences cannot be regenerated from the compressed sequence. Just because information is lost doesn't mean that the quality of the output is reduced. For example, random noise has a very high information content, but when present in an image or a sound file, we would typically be perfectly happy to drop it. Also certain losses in images or sound might be completely imperceptible to a human viewer (e.g. the loss of very high frequencies). For this reason, lossy compression algorithms on images can often get a factor of 2 better compression than lossless algorithms with an imperceptible loss in quality. However, when quality does start degrading in a noticeable way, it is important to make sure it degrades in a way that is least objectionable to the viewer (*e.g.*, dropping random pixels is probably more objectionable than dropping some color information). For these reasons, the way most lossy compression techniques are used are highly

dependent on the media that is being compressed. Lossy compression for sound, for example, is very different than lossy compression for images.

In this section we go over some general techniques that can be applied in various contexts, and in the next two sections we go over more specific examples and techniques.

2.3.1 Scalar Quantization

A simple way to implement lossy compression is to take the set of possible messages S and reduce it to a smaller set $S`$ by mapping each element of $S$ to an element in $S`$ [16, 18]. For example we could take 8-bit integers and divide by 4 (*i.e.*, drop the lower two bits), or take a character set in which upper and lower case characters are distinguished and replace all the uppercase ones with lowercase ones. This general technique is called *quantization*. Since the mapping used in quantization is many-to-one, it is irreversible and therefore lossy.

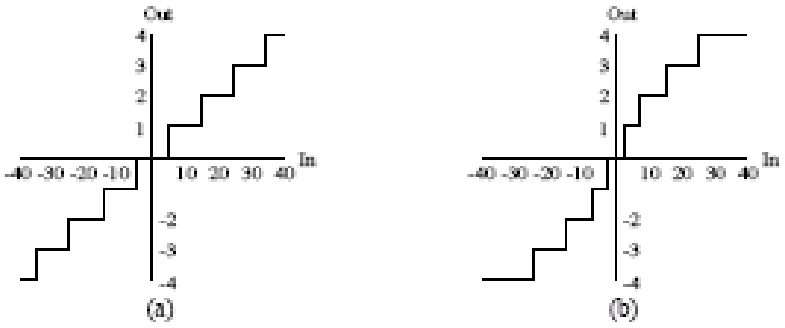


Figure (2.2): Examples of (a) Uniform and (b) Non-uniform Scalar Quantization.

In the case that the set S comes from a total order and the total order is broken up into regions that map onto the elements of $S`$, the mapping is called *scalar quantization*. The example of dropping the lower two bits

given in the previous paragraph is an example of scalar quantization. Applications of scalar quantization include reducing the number of color bits or gray-scale levels in images (used to save memory on many computer monitors), and classifying the intensity of frequency components in images or sound into groups (used in JPEG compression).

The term *uniform scalar quantization* is typically used when the mapping is linear as shown in Figure (2.2). Again, the example of dividing 8-bit integers by 4 is a linear mapping. In practice it is often better to use a *nonuniform scalar quantization*. For example, it turns out that the eye is more sensitive to low values of red than to high values. Therefore we can get better quality compressed images by making the regions in the low values smaller than the regions in the high values. Another choice is to base the nonlinear mapping on the probability of different input values. In fact, this idea can be formalized—for a given error metric and a given probability distribution over the input values, we want a mapping that will minimize the expected error. For certain error-metrics, finding this mapping might be hard. For the root-mean-squared error metric there is an iterative algorithm known as the Lloyd-Max algorithm that will find the optimal mapping. An interesting point is that finding this optimal mapping will have the effect of decreasing the effectiveness of any probability coder that is used on the output. This is because the mapping will tend to more evenly spread the probabilities in S`.

2.3.2 Vector Quantization

Scalar quantization allows one to separately map each color of a color image into a smaller set of output values [35]. In practice, however, it can be

much more effective to map regions of 3-d color space into output values. By more effective we mean that a better compression ratio can be achieved on the basis of an equivalent loss of quality.

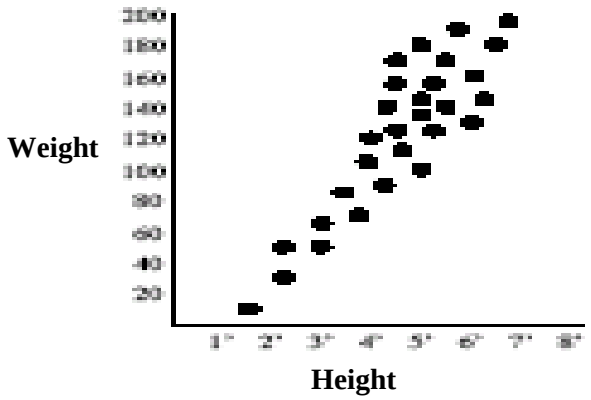


Figure (2.3): Examples of Vector- Quantization for a Hight-Wieght Chart

The general idea of mapping a multidimensional space into a smaller set of messages $S`$ is called *vector quantization*. Vector quantization is typically implemented by selecting a set of representatives from the input space, and then mapping all other points in the space to the closest representative [18, 20, 34]. The representatives could be fixed for all time and part of the compression protocol, or they could be determined for each file (message sequence) and sent as part of the sequence. The most interesting aspect of vector quantization is how one selects the representatives. Typically it is implemented by using a clustering algorithm that finds some number of clusters of points in the data. A representative is then chosen for each cluster by either selecting one of the points in the cluster or using some form of centroid for the cluster. Finding good clusters is a whole interesting topic on its own.

Vector quantization is most effective when the variables along the dimensions of the space are correlated. Figure (2.3) gives an example of

possible representatives for a height-weight chart. There is clearly a strong correlation between people's height and weight and therefore the representatives can be concentrated in areas of the space that make physical sense, with higher densities in more common regions. Using such representatives is very much more effective than separately using scalar quantization on the height and weight.

We should note that vector quantization, as well as scalar quantization, can be used as part of a lossless compression technique. In particular if in addition to sending the closest representative, the coder sends the distance from the point to the representative, then the original point can be reconstructed. The distance is often referred to as the residual. In general this would not lead to any compression, but if the points are tightly clustered around the representatives, then the technique can be very effective for lossless compression since the residuals will be small and probability coding will work well in reducing the number of bits.

2.3.3 Transform Coding

The idea of transform coding is to transform the input into a different form which can then either be compressed better, or for which we can more easily drop certain terms without much qualitative loss in the output. One form of transform is to select a linear set of basis functions (ϕ_i) that span the space to be transformed. Some common sets include, cos, and wavelets [16, 31]. Next sections show examples of the two basis functions for discrete cosine, and wavelet transformations. For a set of *n* values, transforms can be expressed as an $n \times n$ matrix *T*. Multiplying the input by this matrix *T* gives the transformed coefficients. Multiplying the coefficients by T^{-1} will

convert the data back to the original form. For example, the coefficients for the discrete cosine transform (DCT) are

$$T_{ij} = \begin{cases} \sqrt{1/n}\cos\dfrac{(2j+1)i\pi}{2n} & i = 0, 0 \le j < n \\ \sqrt{2/n}\cos\dfrac{(2j+1)i\pi}{2n} & 0 < i < n, 0 \le j < n \end{cases} \quad (2.1)$$

The DCT is one of the most commonly used transforms in practice for image compression, more so than the discrete Fourier transform (DFT) [8,21]. This is because the DFT assumes periodicity, which is not necessarily true in images. In particular to represent a linear function over a region requires many large amplitude high-frequency components in a DFT. This is because the periodicity assumption will view the function as a saw tooth, which is highly discontinuous at the teeth requiring the high-frequency components. The DCT does not assume periodicity and will only require much lower amplitude high-frequency components. The DCT also does not require a phase, which is typically represented by using complex numbers in the DFT.

For the purpose of compression, the properties we would like of a transform are (1) to decor-relate the data, (2) have many of the transformed coefficients be small, and (3) have it so that from the point of view of perception, some of the terms are more important than others.

2.4 Other Lossy Transform Codes

2.4.1 Wavelet Compression

JPEG and MPEG decompose images into sets of cosine waveforms. Unfortunately, cosine is a periodic function; this can create problems when an image contains strong a periodic features [33]. Such local high-frequency spikes would require an infinite number of cosine waves to encode properly. JPEG and MPEG solve this problem by breaking up images into fixed-size blocks and transforming each block in isolation. This effectively clips the infinitely-repeating cosine function, making it possible to encode local features.

An alternative approach would be to choose a set of basis functions that exhibit good locality without artificial clipping. Such basis functions, called "wavelets", could be applied to the entire image, without requiring blocking and without degenerating when presented with high-frequency local features [36, 39].

How do we derive a suitable set of basis functions? We start with a single function, called a "mother function". Whereas cosine repeats indefinitely, we want the wavelet mother function, ϕ, to be contained within some local region, and approach zero as we stray further away:

$$\lim_{x \to \pm\infty} \phi(x) = 0 \tag{2.2}$$

The family of basis functions are scaled and translated versions of this mother function. For some scaling factor *s* and translation factor *l*,

$$\phi_{sl}(x) = \phi(2^s x - 1) \tag{2.3}$$

A well known family of wavelets are the Haar wavelets, which are derived from the following mother function:

$$\phi(x) = \begin{cases} 1 & : 0 < x \le 1/2 \\ -1 & : 1/2 < x \le 1 \\ 0 & : x \le 0 \text{ or } x > 1 \end{cases} \quad (2.4)$$

2.4.2 Model-Based Compression

We briefly present one last transform coding scheme, model-based compression. The idea here is to characterize the source data in terms of some strong underlying model [16]. The popular example here is faces. We might devise a general model of human faces, describing them in terms of anatomical parameters like nose shape, eye separation, skin color, cheekbone angle, and so on. Instead of transmitting the image of a face, we could transmit the parameters that define that face within our general model. Assuming that we have a suitable model for the data at hand, we may be able to describe the entire system using only a few bytes of parameter data.
Both sender and receiver share a large body of a priori knowledge contained in the model itself (e.g., the fact that faces have two eyes and one nose). The more information is shared in the model, the less need be transmitted with any given data set. Like wavelet compression, model-based compression works by characterizing data in terms of a deeper underlying generator. Model-based encoding has found applicability in such areas as computerized recognition of four-legged animals or facial expressions.

3.1 Introduction

There are plenty of specific types of data which need to be compressed, for ease of storage and communication. Among them are texts such as natural language and programs, images, sounds, etc. In this chapter, we focus on the compression of a specific kind of texts only, namely genetic sequences. We consider primarily DNA and RNA sequences that constitute the physical medium in which all properties of living organisms are encoded. The knowledge of its sequences is fundamental in molecular biology. Important molecular biology databases are developed around the world to store nucleotide sequences (DNA, RNA) and amino-acid sequences of proteins. It is well known that their size increases nowadays exponentially fast. Not as big yet as some other scientific database. The compression of genetic sequences constitutes therefore a very challenging task [4, 11, 13].

3.2 Genetic Data

Some insights in molecular biology are necessary to understand the specificity of genetic sequences as inputs for compression algorithms. We describe genetic sequences, DNA, RNA and proteins, and recall shortly the biological machinery. Due to space limitation, we only sketch the purely biological aspects, and focus on the regularities of the sequences relevant for their compression.

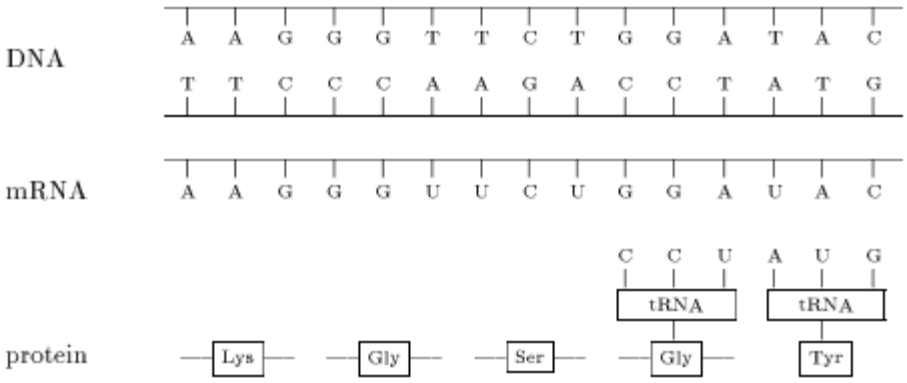


Figure (3.1): DNA to Proteins Translation

The deoxyribonucleic acid (DNA) is a molecule composed of deoxyribonucleotides connected by phosphodiester linkages [1, 13]. There are four kinds of nucleotides as shown in Figure (3.1): adenine (a), cytosine (c), guanine (g), and thymine (t). The DNA is constructed of a double helix held together by hydrogen bonds. The two strands of the helix are exact complement of each other, and each nucleotide of one strand is match to its complement on the other strand, where a pairs with t, and g pairs with c.

Part of the DNA is functional and encodes for different sorts of RNA's or proteins. The ribonucleic acid (RNA) is another kind of nucleic acid found in the cells. It contains the same nucleotides but the thymine (t) which is replaced by the uracil (u)[13]. The RNA plays an important role in the machinery used to translate DNA into proteins Figure (3.1). There are different kinds of RNA molecules. The RNA *polymerase* makes a complementary copy of one strand of the DNA, which is called the messenger RNA (mRNA). This is called the transcription phase. For parts of DNA sequences coding for proteins, the *m*RNA contains the information which serves for the synthesis of proteins. The transfer RNA's (tRNA) are used to translate the mRNA into proteins. Each coden, triplet of nucleotides,

on the mRNA, is recognized by a specific tRNA containing the complementary coden, and it specifies an amino acid of the protein. This is the translation phase. There is the translation phase. There is also another kind of RNA, the ribosomal RNA (rRNA) which is a fundamental constituent of the ribosomes [1, 6].

3.2.1 Defining Repetition

A *repeat* is a recurrence of a pattern. DNA contains repetition of many features. Genes for transfer RNAs (tRNAs) are abundant in genomes and retain similar sequences. Gene clusters that likely evolve via sequence duplication are groups of genes proximally located having similar sequence and often similar, though different, function.

A DNA pattern recurs in four ways; *direct*, *indirect*, *complement* or *reverse complement*. A *direct* or *forward repeat* is the same pattern recurring on the same strand in the same nucleotide order, e.g. accg recurs as accg. An *indirect*, *inverse* or *reverse repeat* recurs on the same strand but, the order of the nucleotides is reversed, e.g. the indirect recurrence of accg is gcca. A *complement repeat* recurs on the same strand with the order preserved but, the nucleotides are complemented such that a and t replace one another and c and g replace one another, e.g. the complement of accg is tggc.

A *reverse complement repeat* recurs on the same strand but, the nucleotides are complemented and the order of the nucleotides is reversed, e.g. the reverse complement of accg is cggt. In DNA, most repetitions occur as forward or reverse complement repeats and rarely as reverse or complement repeats (Grumbach & Tahi 1994) [1, 7].

3.2.2 Biological Classes of Repetitive DNA

The biological literature contains many repetitive DNA classification schemas. Each schema classifies repetitive DNA characteristics as measured by different techniques.

The classification scheme makes a distinction between repetitive regions exhibiting tandem repetition and interspersed repetition. Classification of elements as tandem or interspersed is not precise since each class retains characteristics of both. Nevertheless, tandem and interspersed repetition are prominent in biological descriptions of repetitive classes and are important structurally for computational identification. The number of occurrences of a pattern is referred to as the *copy number*. The *region copy number* refers to the number of copies in a particular tandem repetitive region. The *genome copy number* refers to the number of copies of tandem or interspersed repeats in the entire genome.

3.2.2.1 Tandem Repeats

Tandem repeat regions in DNA exhibit *periodic* recurrence of the same sequence of nucleotides [1, 14]. The pattern structure underlying this periodicity is either a simple sequence of nucleotides or a complex pattern structure having multiple periodicities. The pattern structures vary from several nucleotides to thousands of nucleotides in length and vary from several copies of the pattern to many pattern copies. Regions occur in specific locations in a genome, e.g. in telomeric regions, but are also scattered throughout the genome. Furthermore, some regions have structural or functional roles in the genome and other regions appear to have no role.
Satellite DNA is a common term that can refer to all or a subset of tandem repeats. DNA density studies led to the discovery of tandem repeats having

large copy numbers, termed satellites. As tandem repeats were discovered in different locations and exhibiting different copy numbers, new terms arose such as minisatellite and microsatellite. Today, the distinction between satellite, microsatellite and minisatellite regions is less prominent. Some researchers refer to all types of satellites as tandem repeats and describe a specific tandem repeat region according to its location within the genome, its periodicity, its pattern structure and its copy number.

3.2.2.2 Interspersed Repeats

Interspersed repeats disperse throughout the genome and have no restriction on the relative positions of identical occurrences occurring in tandem and in non-contiguous locations. Amy[1] indicates that interspersed repeats are inserts since they resemble either processed RNAs, i.e. *retroposons* or viruses, i.e. *proretroviral transposons*. Also, a suspected target sequence for insertion occurs at both ends of these repeats as expected for a circular DNA, crossover insertion. Furthermore, some repeats actively move within the genome.

3.3 Exact Repeats

The basic Lempel Ziv model (1976)[25] considers a string to be made up of a mixture of random characters and repeated substring [1, 27]. For example, one possible explanation of the string aagtacacgtacagt is aagtacac (3, 5) gt where (3, 5) indicates a repeat from position 3 of length 5, i.e. gtaca. Random characters are drawn from an alphabet with some probability distribution, in the simplest case a uniform distribution. A repeated substring is a copy of a substring that starts somewhere earlier in the string. The

statistical properties of repeats are the probability with which repeats occur, the probability distribution on starting points and the probability distribution on the lengths of repeats.

3.4 Approximate Repeats

The second approach is Approximate repeat where a repeated substring can be an approximate [15, 24], rather than an exact match to the original substring. There are two prior reasons to believe that this could be useful. First, it in effect increases the number of contexts that match the last characters of the string, at least approximately if not exactly, and therefore increases the amount of available information about what could come next. Second, for some data sets and notably for DNA there are well known processes by which instances of repeats can differ from each other.

Events in the replication of DNA strings can lead to the duplication of substrings (repeats) and also duplication from the complementary strand in the reverse direction (reverse complementary repeats or thousands of characters. Once a substring has been duplicated, the individual copies are subject to the usual evolutionary processes of mutation by which characters can be changed, inserted and deleted. The subsequent mutation process is well modeled by variations on the edit-distance problem which also model spelling errors in text and errors and noise in many other kinds of sequence.

3.5 Informatics: algorithms and concepts for compressing DNA sequences

A DNA sequence ***S*** is regarded computationally as a linear string of ***n*** characters over the alphabet $\sum = \{a, c, g, t\}$. Algorithms for processing strings

about in computer science. Most involve exact or inexact matching of substrings of *S*. A substring of *S* is denoted *S* [*i*, *j*] and represents the characters that start at position *i* and end at position *j* of *S* for $1 \leq i \leq j \leq n$.

1. Windows: Target Regions Versus Word Content

Researchers use fixed length windows for two primary purposes: to restrict analysis to the substring contained in the window, termed the ***target region***, and to measure content of the substring contained in the window, termed ***window*** or ***word content***. A target region restricts analysis to the consecutive positions demarked by two positions in the string. The window content is a measure of the occurrences within the window such as the word content composed of all consecutive positions in the window or the single, double or triple character content the occurrence of single characters or multiple character consecutive combinations [29].

Window length affects algorithm's efficiency and accuracy. Often, a tradeoff exists between time efficiency and identification accuracy. For instance, large target region windows may increase the time to process a single window but on the other hand, they may reduce the number of windows needed to process the string for non-overlapping or slightly overlapping windows. For large content windows, the occurrence probabilities usually spread across more words than for small content windows resulting in fewer identical but larger words in a string or target region. Fewer recurring identical words decrease the number of items to process. Of course, additional identical words exist and could be found with a smaller content window. On the flip side, a small content window may generate a massive number of identical words, overwhelming the identification system and magnifying the time required to process the string.

Also, small content windows do not always lead to a more accurate identification scheme. All in all, the selection of a window size impacts most algorithms significantly.

2. Exact and Inexact Repeats: Distance Measures between Recurrences

A repeat is a substring that occurs at least twice in the string. Definitions of repeats vary from exact or identical recurrence of a substring to inexact or similar recurrence of a substring Table (3.1). Now, it is agreed that two identical substrings represent a repeat, but inexact recurrence requires a measure of similarity. The two most common measures are the ***Hamming distance*** and the ***Levenshtein*** or ***edit distance*** (Levenshtein 1966).

Table (3.1): The Exact and Similar Matches to a Pattern.

Pattern	**accgtga**
Identical (exact) match	accgtga
Similar (inexact) match with 3 mismatches (Hamming distance of k=3)	ac**gg**a**g**g
Similar match with 1 deletion, 1 insertion and 1 mismatch (edit distance of k=3)	a**a**cgtgga

A gray background represents an insertion or deletion in the string relative to the pattern with the inserted nucleotide in the foreground.

A distance measure compares two strings by selecting a set of operations that transforms one string into the other string. The Hamming distance allows mismatch operations while the edit distance allows mismatch, deletion and insertion operations. A single mismatch operation changes a single character into another character in alphabet. A single deletion operation removes a character from a string. A single insertion

operation adds a character in alphabet to a string. Each operation has an associated cost, e.g. a cost of one per operation. The final distance is the set of operations that transforms one string into the other string with minimum cost.

Most computational algorithms apply a threshold to denote the maximum number of operations or maximum cost allowed between similar strings [1, 40]. For instance, in the ***k***-mismatch problem, one finds all occurrences in a string of a pattern having at most ***k*** mismatches, i.e. the Hamming distance with a threshold of ***k*** where mismatches have a cost of one. On the other hand, the ***k***-difference problem accepts ***k*** mismatches, insertions or deletions, i.e. the edit distance with a threshold of ***k*** where each operation has a cost of one.

3. Regular Expressions

A regular expression is a common computational method for expressing a pattern as a set of possible strings [1, 10]. Three basic operations are concatenation, union and closure. Concatenation is a way of representing something following something else. Union is a way of choosing between several substrings. Closure is a way of allowing a substring to occur zero or more times.

Definition [1]: A **regular expression** is a set defined recursively over the alphabet $\sum$ such that $\sum$ does not contain the symbols $\in$, |, (,) and *. The set is specified by the following rules:

1. *Empty String*: The symbol $\in$ is a regular expression.
2. A single character from $\sum$ is a regular expression.

3. *Concatenation*: A regular expression followed by another regular expression is a regular expression.
4. *Union*: Two regular expressions separated by a "|" form a regular expression.
5. A regular expression enclosed in parentheses is a regular expression.
6. *Closure*: A regular expression enclosed in parentheses and followed by "*" is a regular expression. The symbol * is called the Kleene closure.

Example: **tg** is a simple concatenation of two characters.

Example: (**tg**)* is a regular expression expressed as a closure expression.

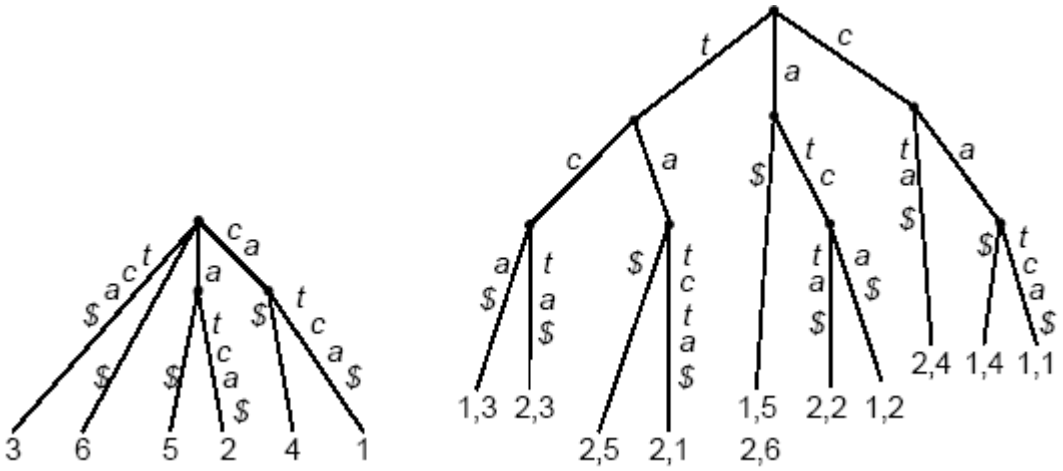
Example: (**cagta**(tg)*)* is a nested closure expression that contains a closure expression

Example: **cagta**|cagccacaata|cagca is a union of four substrings.

4. Suffix Trees

A complete suffix tree represents all suffixes in a string (see Figure 3-2). A suffix is a substring that extends from a position in the string to the end of the string. Suffixes are inserted into a tree such that two suffixes that begin with an identical series of characters traverse the same path through the suffix tree. The path begins at the root node and proceeds down the same path until a difference between the suffixes occurs at which point they each proceed along separate paths. Conceptually, each node in the tree has one branch entering the node such that all suffixes that enter the node have an identical series of characters and each node has one or more separate branches leaving the node such that suffixes that continue to be identical traverse the same branch and suffixes which differ traverse via different branches. Actual implementations vary considerably using trees, directed

acyclic graphs (dags), directed acyclic weighted graphs (dawgs), lists, arrays and hash tables.



Figure(3.2): Suffix trees for one String

As shown in Figure (3.2) (leftmost tree) and a generalized suffix tree for two strings (rightmost tree). A suffix is a substring from a position in the string to the end of the string. The leftmost tree contains every suffix in the string **catca**. The rightmost tree contains every suffix for the strings 1 ***S*** =**catca** and 2 ***S*** =**tatcta**. A suffix is read from the topmost, root node to a leaf. In the left tree, the leaves are composed of a number indicating the starting position of the suffix in the string. The leaves in the right tree are composed of two numbers; the first number indicates the string and the second number indicates the starting position of the suffix in that string. The black dots are nodes in the tree and represent points where the paths of two or more suffixes divide. Each line segment between nodes represents one or more characters in the suffix. The '$' symbol represents the end of a string.

3.6 Genetic Algorithm

Genetic Algorithms are a part of **evolutionary computing**, which is a rapidly growing area of artificial intelligence [12, 42]. As you can guess, genetic algorithms are inspired by Darwin's theory of evolution. Simply said, problems are solved by an evolutionary process resulting in a best (fittest) solution (survivor) - in other words, the solution is evolved.

3.6.1 NP-hard Problems

One example of a class of problems which cannot be solved in the "traditional" way is NP problems.

There are many tasks for which we may apply fast (polynomial) algorithms. There are also some problems that cannot be solved algorithmically [42].

There are many important problems in which it is very difficult to find a solution, but once we have it, it is easy to check the solution. This fact led to **NP- problems**. NP stands for nondeterministic polynomial and it means that it is possible to "guess" the solution (by some nondeterministic algorithm) and then check it.

3.6.2 Basic Description

Genetic algorithms are inspired by Darwin's theory of evolution. Solution to a problem solved by genetic algorithms uses an evolutionary process (it is evolved).

Algorithm begins with a **set of solutions** (represented by **chromosomes**) called **population**. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are then selected

to form new solutions (**offspring**) are selected according to their fitness - the more suitable they are the more chances they have to reproduce.

This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied.

3.6.3 Operators of GA

The crossover and mutation are the most important parts of the genetic algorithm. The performance is influenced mainly by these two operators. Before we can explain more about crossover and mutation, some information about chromosomes will be given.

1. Encoding of a Chromosome

A chromosome should in some way contain information about solution that it represents. The most used way of encoding is a binary string. A chromosome then could look like this:

Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

Each chromosome is represented by a binary string. Each bit in the string can represent some characteristics of the solution. Another possibility is that the whole string can represent a number - this has been used in the basic GA Of course, there are many other ways of encoding. The encoding depends mainly on the solved problem. For example, one can encode directly integer, string or real numbers, sometimes it is useful to encode some permutations and so on.

2. Crossover

After we have decided what encoding we will use, we can proceed to crossover operation. Crossover operates on selected genes from parent chromosomes and creates new offspring. The simplest way how to do that is to choose randomly some crossover point and copy everything before this point from the first parent and then copy everything after the crossover point from the other parent.

Crossover can be illustrated as follows: (| is the crossover point):

Chromosome 1	11011 | 00100110110
Chromosome 2	11011 | 11000011110
Offspring 1	11011 | 11000011110
Offspring 2	11011 | 00100110110

There are other ways how to make crossover, for example we can choose more crossover points. Crossover can be quite complicated and depends mainly on the encoding of chromosomes. Specific crossover made for a specific problem can improve performance of the genetic algorithm.

3. Mutation

After a crossover is performed, mutation takes place. Mutation is intended to prevent falling of all solutions in the population into a local optimum of the solved problem. Mutation operation randomly changes the offspring resulted from crossover. In case of binary encoding we can switch

a few randomly chosen bits from 1 to 0 or from 0 to 1. Mutation can be then illustrated as follows:

Original offspring 1	1101111000011110
Original offspring 2	1101100100110110
Mutated offspring 1	1100111000011110
Mutated offspring 2	1101101100110110

The technique of mutation (as well as crossover) depends mainly on the encoding of chromosomes. For example when we are encoding permutations, mutation couldbe performed as an exchange of two genes.

3.6.4 Parameters of GA

Crossover and Mutation Probability

There are two basic parameters of GA-crossover probability and mutation probability [19].

1. **Crossover probability**: How often crossover will be performed? If there is no crossover, offsprings are exact copies of parents. If there is crossover, offsprings are made from parts of both parent's chromosome. If crossover probability is **100%**, then all offsprings are made by crossover. If it is **0%**, whole new generation is made from exact copies of chromosomes from old population (but this does not mean that the new generation is the same!). Crossover is made in hope that new chromosomes will contain good parts of old chromosomes and therefore the new chromosomes will be better. However, it is good to leave some part of old population survive to next generation.

2. **Mutation probability**: How often will parts of chromosome be mutated? If there is no mutation, offspring are generated immediately after crossover (or directly copied) without any change. If mutation is performed, one or more parts of a chromosome are changed. If mutation probability is **100%**, whole chromosome is changed, if it is **0%**, nothing is changed. Mutation generally prevents the GA from falling into local extremes. Mutation should not occur very often, because then GA will in fact change to **random search**.

3.6.5 Other Parameters

There are also some other parameters of GA. One another particularly important parameter is population size [12].

1. Population Size:

How many chromosomes are in population (in one generation)? If there are too few chromosomes, GA have few possibilities to perform crossover and only a small part of search space is explored. On the other hand, if there are too many chromosomes, GA slows down. Research shows that after some limit (which depends mainly on encoding and the problem) it is not useful to use very large populations because they do not solve the problem faster than moderate sized populations.

2. Selection

Chromosomes are selected from the population to be parents for crossover. The problem is how to select these chromosomes. According to Darwin's theory of evolution the best ones survive to create new offspring. There are many methods in selecting the best chromosomes. Examples are

roulette wheel selection, Boltzman selection, tournament selection, rank selection, steady state selection and some others.

Some of them will be described in this chapter.

Roulette Wheel Selection

Parents are selected according to their fitness [42]. The better the chromosomes are, the more chances to be selected they have. Imagine a **roulette wheel** where all the chromosomes in the population are placed. The size of the section in the roulete wheel is proportional to the value of the fitness function of every chromosome - the bigger the value is, the larger the section is. See the figure (3.3) for an example.

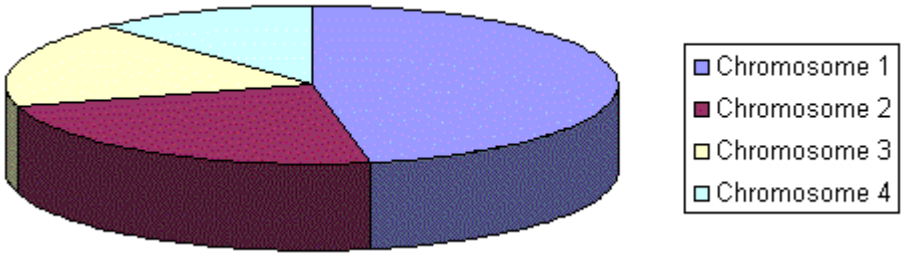


Figure (3.3): Roulette Wheel Selection

3. Elitism

The idea of the elitism has been already introduced [12,19]. When creating a new population by crossover and mutation, we have a big chance, that we will loose the best chromosome.

Elitism is the name of the method that first copies the best chromosome (or few best chromosomes) to the new population. The rest of the population is constructed in ways described above. Elitism can rapidly increase the performance of GA, because it prevents a loss of the best found solution.

4.1 Introduction:

In the previous chapters a number of compression methods have been presented. In this chapter we will explain a proposed compression method. Two algorithms were suggested, the first one for compression and the other for de-compression. GA was applied to extract the data base or references on the basis of which the compression algorithm encodes the original data.

4.2 The Suggested Compression System

The basic idea of the suggested compression algorithm as shown in Figure (4.1) is:

The suggested algorithm is a ***two stages algorithm, the first stage is to extract the database or references which are the sequences that have more copies, the second stage (encoding) proceeds as follows: For input w, assume that a part of it, say v, is database that has already been extracted in first part and remaining part is u i.e. w= vu the compression algorithm finds an "optimal sequences" of u such that it exactly or approximately matches substring in v so that this sequence of u can be encoded economically after outputting the code of the sequence, remove it from u, and append it to the suffix of v, continue the process. In the final we will apply Huffman Coding as a back end.***

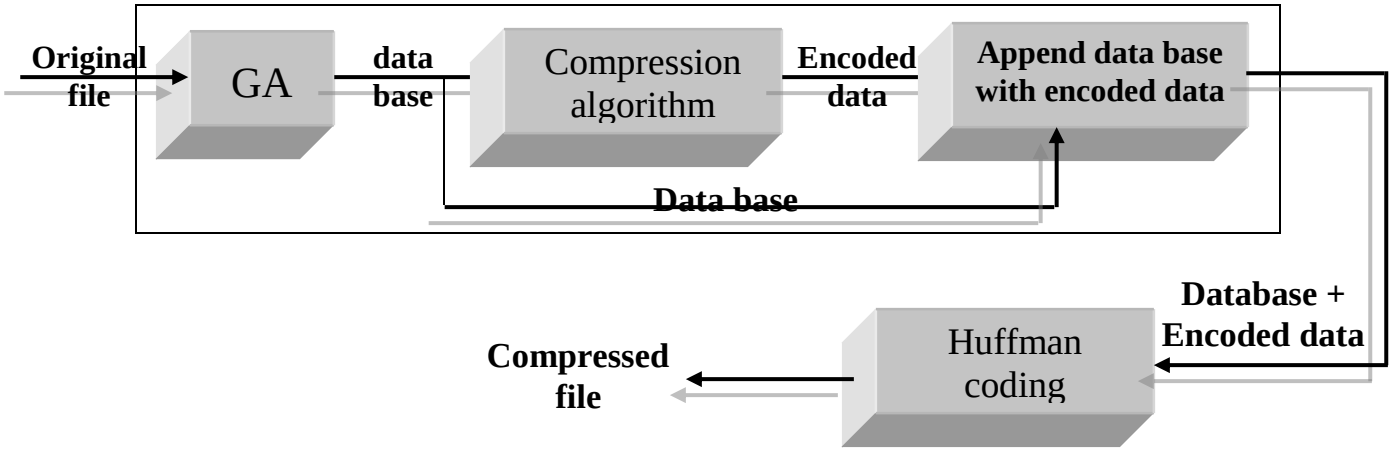


Figure (4.1) Compression System Structure

4.2.1 Encoding Edit Operation

We consider four standard edit operations in our suggested algorithm These are:-

a. *Replace*. This operation is expressed as (*R*, *p*, *char*) which means replacing the character at position *p* by character *char*.

b. *Insert*. This operation is expressed as (*I*, *p*, *char*), which means inserting character *char* at position *p*.

c. *Delete*. This operation is written as (*D*, *p*), which means deleting the character at position *p*.

d. *Inverse*. This operation is written as (*V*), which means inversing the sequence.

It can be easily seen that there are infinitely many edit sequences to transform one string to another. A list of edit operations that transform a string *u* to another string *v* is called an *Edit Transcription* of the two strings. This will be represented by an edit operation sequence λ (*u*, *v*) that orderly lists the edit operations. For example, the edit operation sequence of the first edit transcription in the this example is λ(“gaccgtca”, “gaccttca”)=(*R*, 5, *g*);

and for the second edit transcription, λ("gaccgtca", "gacctca")= (*I*, 5, *g*), (*D*, 5).

If we know the string *u* and an edit operation sequences λ(*u*, *v*) from *u* to *v*, then the string *v* can be constructed correctly using λ. There are many ways to encode one string by given another as follows:-

1. Exact matching method. We can use (repeat position, repeat length) to represent an exact repeat.
2. Approximate matching method. In this case, the string "gaccgtca" can be encoded as (*R*, 4, *g*) depending on given string "gaccttca".

4.2.2 Condition C

We adopt the following constraint in the suggested algorithm to limit the search. If the number of edit operation located in any substring of length k in the sequence, s of u for an edit operation sequence T(s, t) is not larger than a threshold value b, we say that T(s, t) satisfies condition c. in this way we limit our search space. Experiments show that setting c to (k, b)= (16, 3) gives good results

4-2-3 Compression Flag

It is used to distinguish between coding data and non coding data.

4.3 The Suggested System Stages

We can summarize the suggested system stages as shown in Figure (4.2) as follows:

Stage 1: extract database using genetic algorithm.

Stage 2: set u = w, u complete input string and v = extracted database.

2.a Search for optimal sequence s of u, with exact or approximate match t in database v.

2.b Encode a sequence s with (L, P) or (T(s, t), P) where L is length of sequence, P is the position of t in database and T(s, t) is shortest edit operation. Output code.

Remove s from u and append a coded output to the database v if u is not empty go to 2.a else exit.

Stage 3: apply Huffman Coding.

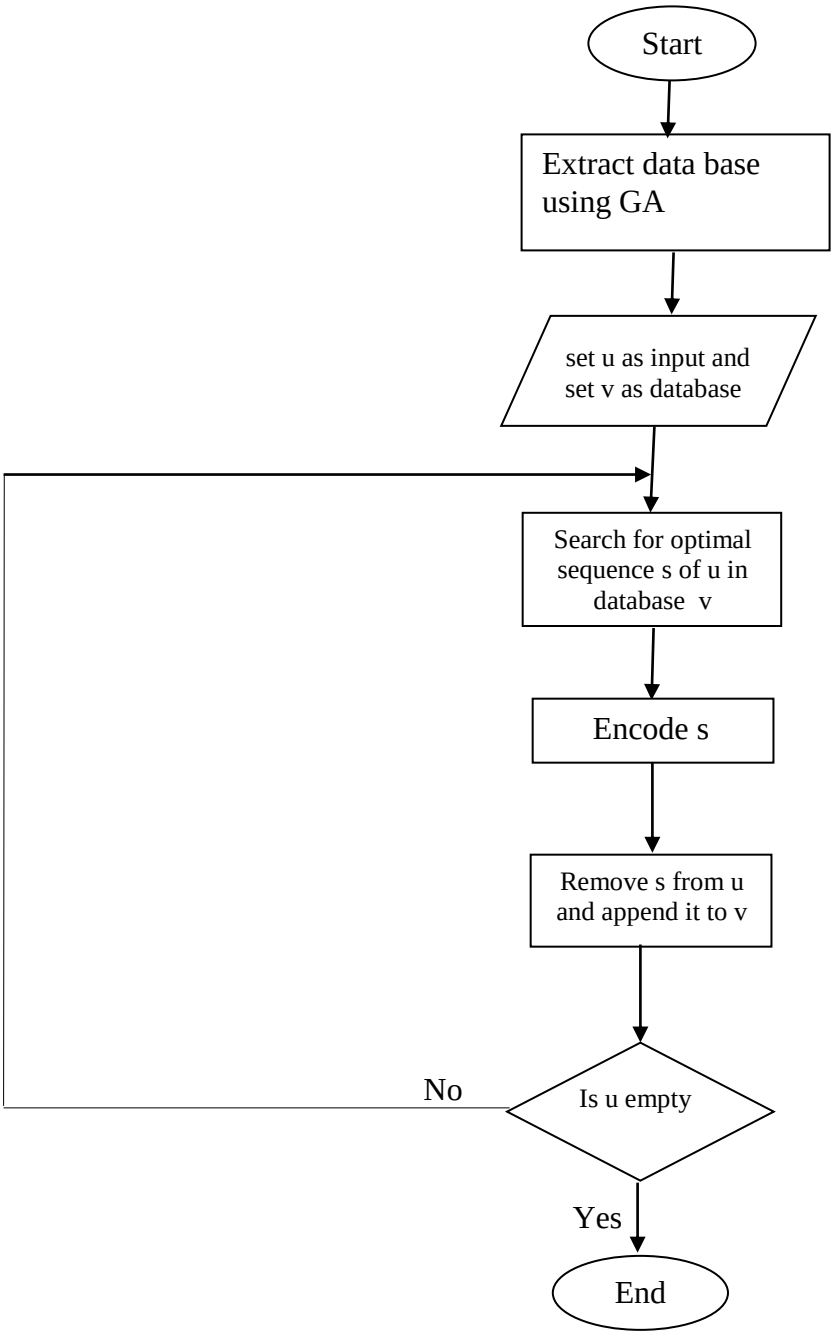


Figure (4.2): The Flow Chart of the Proposed Compression System

4.4 The Suggested System Steps

1- Step1: Input the source file

let u = source file

and v = empty (data base)

2- Step2: Extract Database or References (GCAK):

Genetic algorithms (GAs) [goldberg] belong to a class of search techniques that mimic the principles of natural selection to develop solutions of large optimization problems. GAs operate by maintaining and manipulating a population of potential solutions called chromosomes. Each chromosome has an associated fittness value which is a qualitative measure of the goodness of the solution encoded in it. This fittness value is used to guide the stochastic selection of chromosomes which are then used to generate new candidate solutions through crossover and mutation. Crossover generates new chromosomes by combining sections of two or more selected parents. Mutation acts by randomly selecting genes which are then altered; thereby preventing suboptimal solutions from persisting and increases diversity in the population. The process of selection, crossover and mutation continues for a fixed number of generations or until a termination Condition is satisfied.

GCAK algorithm [3, 42] is used to find optimal sequences that have more copies as shown in Figure (4.3):

1- Generate random solution & Initialize population

2- Evaluate population

3- If stopping criteria is not met do

a. Selection

b. Crossover

c. Mutation

d. Go to step 2

4- Sselect best chromosome and use it as a good database.

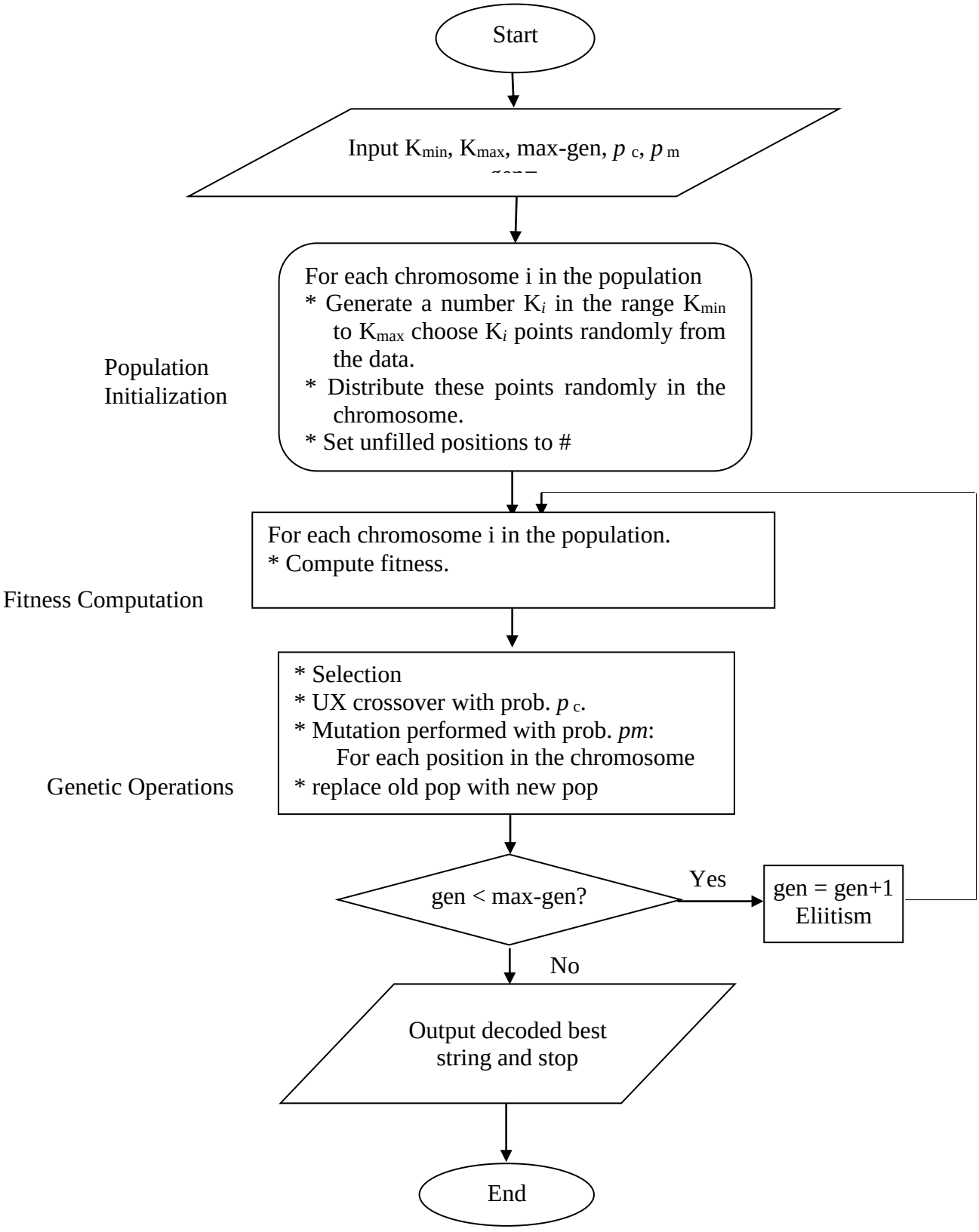


Figure (4.3): flow chart of GCAK

7. Representation (Encoding of Solution)

The chromosomes are made up of a list of sequences. If the sequence at any gene is not null, this means that there are copies of it in database. This sequence is generated randomly from the source data. On the other hand gene (sequence) with null means that the sequnce does not have copies in source data. The value of K is assumed to lie in the range [K_{min}, K_{max}], where K_{min} is chosen to be 4 unless specified otherwise. The length of a sequence is taken to be K_{max} where each individual gene position represents either sequence or a null.

Example of chromosome structure for DNA data:

acgtac	acgtaaata	aaggtt	ttggacgtgta	….	….	….	….	#	ccgatcaagg
Gene 1	gene 2	gene 3	gene 4	-		gene i			gene l

(a) chromosome structure for DNA data

194 92 150 140 110	11 110 100 50 67 67 70	…	…	….	#	100 210 211 50 70 56 48 38
Gene 1	gene 2	-		gene i		gene L

(b) chromosome structure for Image data

Figure (4.4 a, b): Chromosome Structure of DNA Data and Image data

** Population Initialization*

For each sequence i in the chromosome ($i=1,\ldots..,\ l$, where l is the length of the chromosome), a random number K_i in the range [K_{min} , K_{max}], and P in the range [*1, length of original file*- K_{max}] are generated. The sequence S_i with length K_i at position P_i is assumed to encode the $gene_i$ in $chromosome_j$ as shown in Figure (4.4). For initializing, these sequences are chosen randomly from the source data. These sequences are distributed randomly in the chromosome. Let us consider the following example.

Example: Let K_{min} =4 and K_{max} =16. Let the random number K_i be equal to 8 and P_i=64 for chromosome *j*. Then this gene will encode the sequence at location 64 in source data with length 8 (8 randomly chosen points from source data) be "acgtgaat".

* *Fitness Computation*

For each chromosome in the population, the fitness of the chromosome is computed as follows:-

$$\sum_{i=1}^{l} c(s_i) * k_i$$

where *c* is the number of copies of s_i in source data.

S_i is the chosen sequence.

K_i is the length of sequnce s_i.

* *Genetic Operators*

The following genetic operators are performed on the population of strings for a number of generations.

a. ***Selection***: Roulette Wheel Selection (RWS) is applied

Roulette Wheel Selection (RWS) is applied, here Parents are selected according to their fitness. The better the chromosomes are, the more chances to be selected they have.

$$e_{i=} f_i \diagup \sum f_i$$

Where e_i is expected value of chromosome i

b. ***Crossover***: uniform crossover (UX) [12, 19] is applied as shown in Figure (4.5). UX crosover creats offspring by deciding for each allele of one

parent, whether to swap that allele with the corresponding allele in the other parent with Pc= 0.5.

UX is more disruptive of schemata than one cut (1X) and two cut (2X). But it does not have a length bias.

UX is more likely to constract instances of new schemata than one cut (1X) and two cut (2X).

Example of crossover, let parent1 & parent2 are:

aaatgcgta	ggttacgta	aagccaatcc	Ggcctgtgaac	….	….	….	….	#	ccgatcaagg

acgtac	acgtaaata	aaggtt	ttggacgtgta	….	….	….	….	#	gatcaagg

Then the child1 & child2 are

acgtac	ggttacgta	aaggtt	ggcctgtgaac	….	….	….	….	#	ccgatcaagg

Pc=0.6 pc=0.4 pc=.5 pc=.2 pc=.1

aaatgcgta	acgtaaata	aagccaatcc	ttggacgtgta	….	….	….	….	#	gatcaagg

C. ***Mutation***: Each position in a chromosome is mutated with probability *pm* in the following way. If the value at that position is not null, then it becomes null else a new sequence is generated by selecting random points from dataset.

3. Step 3 How Algorithm Find Optimal Sequence

We consider the problem of DNA compression. It is well known that one of the main features of DNA sequences is that they contain substrings which are duplicated except for a new random mutations. For this reason most of DNA compressors work by searching and encoding approximate repeats. We depart from this strategy by searching and encoding exact and

approximate repeats. However, we use an encoding designed to take advantage of the possible presence. Another important feature of our algorithm is its small space occupancy which makes it possible to compress sequences.

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 database which are routinely handled by molecular biologists.

Although the saving in space is important, it is not the only application of DNA compression. DNA compression has been used to distinguish between coding and non-coding regions of a DNA sequence, to evaluate the 'distance' between DNA sequences and to quantify how close two organisms are in the evolutionary tree, and in other biological applications.

It is well known that the compression of DNA sequences is a very difficult problem.

4. Step 4 Finding Exact Matches

Let T [1, N] denote a string over the alphabet {a, c, g, t}. Our first building block for the design of a DNA compressor is an efficient procedure for finding the occurrences of repeated substrings. We know that repetitions in DNA sequences are more frequent and longer than in linguistic texts. Moreover, repeated substrings may appear far apart in the input sequence.

5. Step 5 Encoding the Matches

Having established a procedure for finding repeated strings and complemented palindromes, we now show how to use it for the actual compression of DNA sequences. Suppose we have already compressed an initial portion T [1, z] of the input sequence. Using the procedure in the previous section we check whether the block T [z+ 1, z+ B] (or its reverse complement) matches a stored block. If not, we try with T [z+ 2, z+ B+ z+ 1], T [z+ 3, z+ B+ 2] and so on, until we find a block T [z+ b, z+ b+ B -1] which matches. Then, we extend this match as far as possible and we get a sequence T which is a copy without error or with some error of substring starting in T [1, z]. We are now ready to encode the string T.

6. Remove T From u

4.4 Coding Format: There are Two Format:

a. Coding Format of approximate is shown in Figure(4.6)

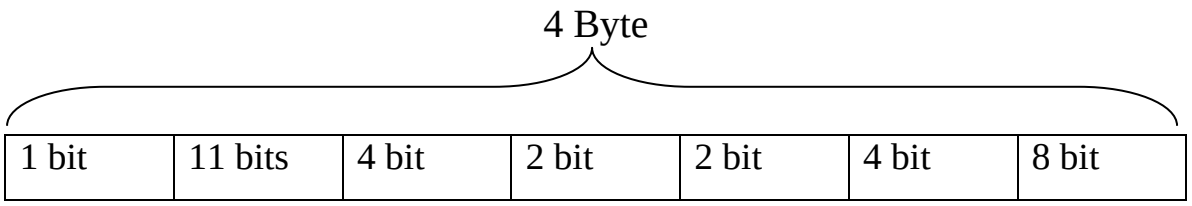


Figure (4.5): Coding Format of Approximate Matching

1 bit= control bit to distinguish whether the code format is exact or approximate

11 bit= index of reference, therefore the data base size is 2^11=2KB

4 bit= length of sequence, therefore the maximum length of sequence is 16

2 bit= number of edit operation, therefore the maximum number of edit operation is 4

2 bit= edit operation, 4 edit operations (R,I,D,V)

4 bit= position

8 bit= character (one byte)

b. Coding format of exact is shown in Figure(4.7)

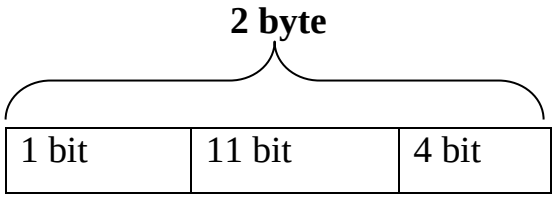


Figure (4.6): Coding Format of Exact Matching

1 bit= control bit to distinguish whether the code format is exact or approximate

11 bit= index of reference, therefore the data base size is 2^11=2KB 4 bit= length of sequence

4 bit= length of sequence, therefore the maximum length of sequence is 16

4.5 Example of DNA Compression

Let W be the input file

W=*acgttgaagagtacgaccgttcaattacctgtacgaccgtcaacgtgaattacgttgaagtacgacc gtggaattacacacgttgaatcacgtttgaaggaattac*

In first stage the algorithm extracts database u

v= *acgttgaagtacgaccgtaattac*

In second stage the algorithm encodes the data depending on the already existing references.

The final output data will be:

(1, 8)ga (9, 10)tc (10, 6)ct (9, 10)ca (1, 8, D, 5)tt (1, 8) (9, 10)gg (10, 6)ac (1m 8)tc (1, 8, I, 6, t) (10, 6)

4.6 Image Data

A digital image is made up of small dots called pixels. Each pixel can be either one bit, indicating a black or a white dot, or several bits, indicating one of several colors or shades of grey. We assume that the pixels are stored in an array called a bitmap in memory, so the bitmap is the input stream for the image. Pixels are normally arranged in the bitmap in scan lines, so the first bitmap pixel is the dot at the top left corner of the image, and the last pixel is the one at the bottom right corner.

If the number of gray scale is 256, it can be reduced to 255 with one value reserved as a flag to precede every byte with encoded data. If the flag is, say, 255, then the sequence is encoded data.

Example of the image file

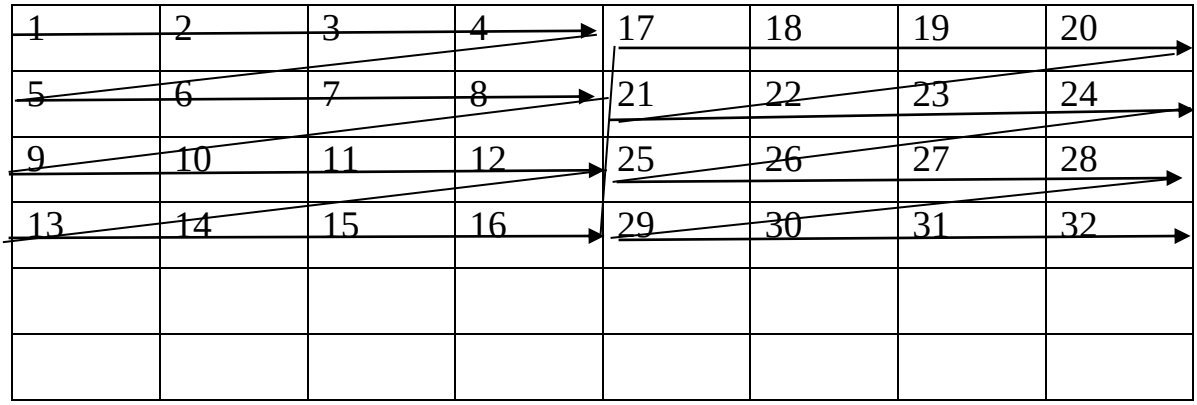


Figure (4.7): Scanning of Image

Step 1: The first step convert bitmap array into single sequence as follows:-

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32……

Step2: Apply GA to extract database or references which are regions that have more copies in the Image.

Step3: Apply suggested algorithm to detect the regions that are similar to the regions already existing in database and encode it.

4.7 Decompression Algorithm

We can summarize decompression algorithm, as shown in Figure (4.9), into:

Step 1: Split database from encoded data

Step 2: Input single byte from encoded data

if the byte is not compression flag then output it

Step 3: If the control bit denotes the exact matching

then input two bytes and output the sequence at position p from database.

Step 4: Else if the control bit denotes the approximate matching

Then input four bytes and output the sequence at position p from database with correct error.

Step 5: If the encoded data is empty then end else go to step 2

Figure (4.8): Decompression System

4.8 Example of DNA Decompression

Let the encoded data be

(1, 8)ga (9, 10)tc (10, 6)ct (9, 10)ca (1, 8, D, 5)tt (1, 8) (9, 10)gg (10, 6)ac (1m 8)tc (1, 8, I, 6, t) (10, 6)

In first stage the algorithm split the database from encoded data

In second stage the algorithm determine the encoded data whether it is exact or approximate as shown in Figure(4.10).

In the next stage, the algorithm decode the data depending on the already existing references.

The final output data will be:

W=*acgttgaagagtacgaccgttcaattacctgtacgaccgtcaacgtgaattacgttgaagtacgaccgtggaattacacacgttgaatcacgtttgaaggaattac*

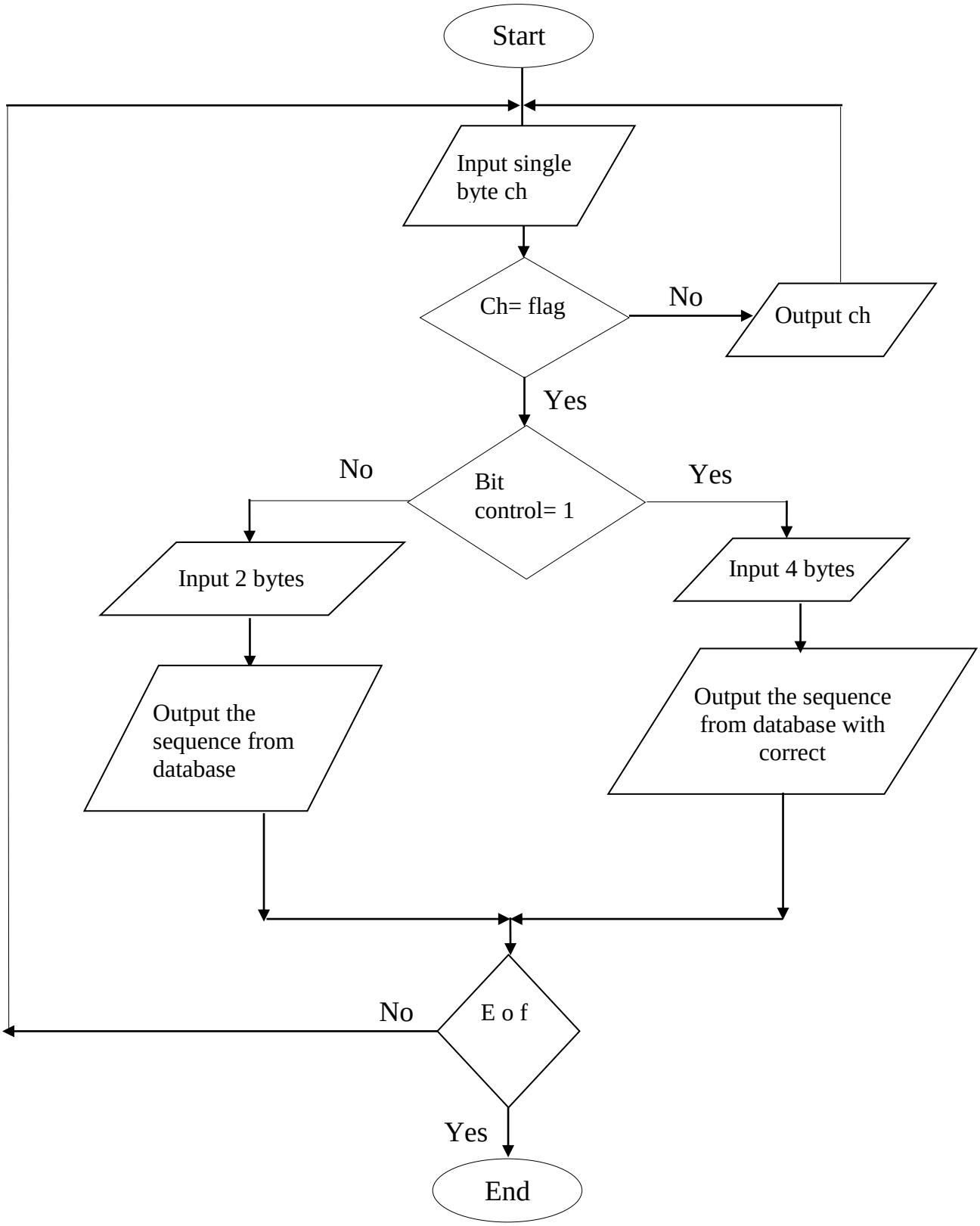


Figure (4.9): Flow Chart of Decompression Algorithm

5.1 The Experimental Results:

5.1.1 Results of DNA Files

A wide variety of techniques are used for data compression, they include lossy and lossless as explained in detail in the previous chapters. In this chapter to test our compression algorithm we have downloaded the complete genome of DNA sequences from:

http://www.mfn.unipmn.it/~manzini/dnacorpus

The resulting test suite is summarized in Table (5.1).

All tests have been carried out on a 1.7 GHz Pentium 4 with 256 MB of main memory. The programming language was VB ver.6.

Table (5.1) Shows the Compression Ratio and Time complexity for DNA Files

Source file name	Source file size	Compressed file size	Compression ratio	Time
Dnal	54572 B	10195 B	82%	2.11225s
Dna2	52238 B	13402 B	75%	2.5456 s
Dna 3	1179648 B	281692 B	77%	3.26525 s

5.1.2 Comparison with other DNA Compressors [4,40]

Our algorithm was compared with other algorithms designed to compress DNA sequence. Unfortunately, to carry out such a comparison turned out to be a difficult task. The reason is that difference of source data.

Table (5.3) shows the compression ratios of several algorithms on a set of DNA sequences which have been used for testing. In almost DNA compressor from Table (5.2) we can see that proposed algorithm achieve

a compression ratio very close to the best DNA compressors "Gencompress".

For top to bottom we have the results for mouse DNA sequences and some viruses. Each row displays the three maximum of compression ratia for a single algorithm and shows the compression ratio for each sequence in bits per symbol. The last row shows the compression ratio of our algorithm.

Table (5.2) the Comparison with some algorithm

Algorithm	Compression Ratio		
gzip	2.3275	2.3618	2.0648
bzip	2.1685	2.1802	1.7289
bioc	1.848	1.9262	1.5993
Genc	1.8414	1.9231	1.3074
Proposed algorithm	2.0524	1.9103	1.4945

5.1.3 Results of Images Test

We have used some image files to test our algorithm, the resulting test is summarized in Table (5.3). 256 (8bpp) Images were used as the best images by proposed Method. Figure (5.1) shows the test images.

Table (5.3) the Compression Ratio and Time complexity for Image Files

Source file name	**Source file size**	**Compressed file size**	**Compression ratio**	**Time**
Image 1	27478 B	7420 B	73%	1.96712 s
Image2	96342 B	33196 B	66%	2.62375 s
Image3	31104 B	18670 B	40%	2.18725 s
Image4	29302 B	12057 B	59%	2.827 s
Image5	12022 B	5278 B	57%	0.67175 s
Image6	24134 B	11730 B	52%	1.609375 s

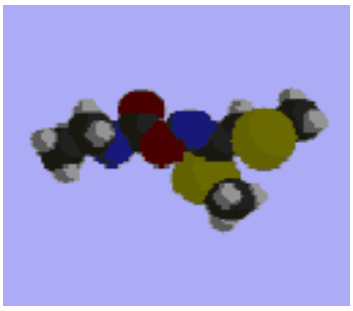


Image1 173×150

Image2 181×177

Image3 135×190

Image4 166×168

Image5 94×114

Image6 174×131

Figure (5.1) Experimental images

5.2 Conclusions

In this section we will present some conclusions:

1- Our algorithm is limited by two parameters, the data base size and max length of sequence, if we increase the size of database, the number of bits which indicate the position in data base will increase, if we increase the size of max length, we will need to increase the number of bits which indicate the length of sequence.

2- Our algorithm detects the exact matching easily, but the difficult in approximate matching because there are more errors in sequences and proposed algorithm can only detect three errors in one sequence.

3- It is well known that DNA sequences do not present the same regularities which are usually found in linguistic texts. This explains why these standard tools such as GZIP and LZW fail to compress them. To design a DNA compressor we must take advantage of the regularities which are usually found in this kind of data. The following three properties have been observed in many sequences and have been the basis, so far, of every DNA compressor.

Dna-1 DNA sequences contain repeated substring. Repeated substring in DNA sequences are often longer than in linguistic texts, but they are less frequent. Another important difference with linguistic texts is that in DNA sequences repeated substrings may appear far apart in the input sequence, whereas in linguistic texts the occurrences of a substring are often clustered in a small region of the input.

Dna-2 DNA sequences contain repeated complemented palindromes, that is, pairs of substrings which are the reverse complement of each other. The characteristics of the repeated palindromes are the

same as those of repeated substrings: they are not very frequent, they can be long, and they may appear far apart in the input.

Dna-3 DNA sequences contain repeated strings with errors. With this term we denote a repetition in which the second occurrence is not identical to the first one (or to the reverse complement of the first one) because a few symbols have been deleted, inserted, or replaced by other symbols.

5.3 Suggestions:

The suggestions are:

1- The proposed algorithm can be applied to other data files such as sound or text files.

2- It can also be applied to other transformations such as wavelet or DCT, where the data is processed by transformations and then the proposed algorithm is applied to the coefficients, but this will lead to a lossy method.

3- We suggest that the size of database, max length of sequence and min length of sequences can be variable instead fixed, so we can add header to the compressed file include all this information.

ضغط سلاسل الحامض النووي باستخدام المطابقة الكلية والتقريبية

رسالة مقدمة الى

مجلس كلية العلوم- جامعة بابل

وهي جزء من متطلبات نيل درجة الماجستير علوم
في علوم الحاسبات

من
محمد عبيد مهدي

كانون الاول- 2005 ذو القعدة- 1426

الخلاصة:

في هذا البحث نقترح نموذج لضغط سلاسل الحامض النووي ال DNA بدون فقدان يعتمد على انتظامات السلاسل بالاضافة الى ان السلاسل الجزئية المتكررة يمكن ان تكون مطابقة كليا او تقريبيا الى بعض السلاسل الجزئية الاصلية, هذا قريب لحالة الحامض النووي الDNA . من المعروف ان ضغط سلاسل الـ DNA من المشاكل الصعبة كونها تحتوي على اربعة رموز {a, c, g, t} التي يمكن خزنها باستخدام بتين لكل رمز. ان طرق الضغط التقليدية مثل Gzip, lZW فشلت في ضغط سلاسل الـ DNA كونها تستخدم اكثر من بتين لكل رمز.

ان الخوارزمية المقترحة توصلت الى نسبة ضغط قريبة جداً الى افضل خوارزميات ضغط الـ DNA. الخوارزمية المقترحة تم تطبيقها على ملفات الصور. ويمكن تلخيص العمل المقترح كمرحلتين: المرحلة الاولى هي لاستخلاص قاعدة البيانات او المصادر باستخدام الخوارزميات الجينية. في المرحلة الثانية ستقوم الخوارزمية بالبحث عن السلاسل المتشابهة كلياً او تقريبياً وترميزها بشكل اقتصادي.

للنموذج عدد من المعاملات يمكن تخمينها من السلسلة المراد ضغطها. ان هدف البحث ليس ضغط سلاسل الحامض النووي ال DNA او تقليل مساحة تخزين البيانات فقط ولكن ايضا تقليل كلفة انتقال البيانات عبر وسائط الاتصال.

REFERENCES

[1] Amy M. Hauth, "**Identification of Tandem Repeats: Simple and Complex Pattern Structures in DNA Sequences**" University of **Wisconsin-Madison, 2002.**

[2] Arne Andersson and Stefan Nilsson, "**Efficient Implementation of Suffix Trees**", Software– Practice and Experience, 1995.

[3] Bandyopadhyay, Sanghamitra and Ujjwal Maulik, "**Genetic Clustering for Automatic Evolution of Clusters and Application to Image Classification**", Pattern Recognition, (35) 2002, 1197-1208.

[4] Chen, X., Kwong, S., and Li, M. "**A Compression Algorithm for DNA Sequences and its Applications to Genome Comparison**". Genomic Informatics, New York, 2000.

[5] David Cary, "**Data Compression**", http://www.ics.uci.edu/~dan/pubs/DataCompression.html, 2003.

[6] David Loewenstern, Peter N. Yianilos, "**Significantly Lower Entropy Estimates for Natural DNA Sequences**", Department of Computer Science Princeton University, Princeton, New Jersey, Technical Report, 1996.

[7] David R. Powell, David L. Dowe, Lloyd Allison and Trevor I. Dix "**Discovering Simple DNA Sequences by Compression**", Department of Computer Science, Monash University, Clayton, Report, Australia 1999.

[8] David S., "**Data Compression the complete reference**", spring verlag Newyourk, 1998.

[9] Demiriz, A., Bennett, K. P., and Embrechts, M. "**A Genetic Algorithm Approach for Semi-Supervised Clustering**", Dept. of Decision Sciences and Engineering System, Report, 2002.

[10] Eric Lehman Abhi Shelat, "**Approximation Algorithms for Grammar-Based Compression**", MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, 2000.

[11] Giovanni Manzini and Marcella Rastero, "**A Simple and fast DNA Compressor**", Bioinformatics, vol.18, no.12, 2004.

[12] Goldberg D., "**Genetic Algorithm In Search, Optimizing an Machine Learning**", Addison-Wesley, 1989.

[13] Grumbach, S. and Tahi, F., "**A New Challenge for Compression Algorithms: Genetic Sequences**", J. Information Processing and Management, vol.30, no.6, 1994.

[14] Grumbach, S. and Tahi, F., "**Compression of DNA Sequences, Proc**." IEEE. On Data Compression, pp 340-350, 1993.

[15] G. Sampath, "**A Block Coding Method that Leads to Significantly Lower Entropy Values for the Proteins and Coding Sections of Haemophilus Influenzae**", monash university, Report, 2003.

[16] Guy E. Blelloch, "**Introduction to Data Compression**", Computer Science Department, Carnegie Mellon University, 2001.

[17] Jacob Ziv and Abraham Lempel, "**Compression of Individual Sequences Via Variablerate Coding**", IEEE Transactions on IT(18), 1978.

[18] James E. Fowler, "**An Open-Source Software Library for Quantization, Compression**", and coding Dept. of Electrical & Computer Engineering, Mississippi State University, Mississippi State, MS, 2000.

[19] J.L.R. Filho, P.C. Treleaven, C. Alippi, "**Genetic Algorithm Programming Environments**", Mississippi State University, 1994.

[20] John A. Bullinaria, "**Learning Vector Quantization (LVQ)**", Monash University, 2003.

[21] John G. Cleary and Ian H.Witten, "**Data Compression Using Adaptive Coding and Partial String Matching**", 1984.

[22] John G. Cleary and W. J. Teahan, "**Unbounded length contexts for PPM**", Computer Journal, vol(40), 1997.

[23] J. ZIV, A. LEMPEL, "**A Universal Algorithm for Sequential Data Compression**", IEEE TRANSACTIONS ON IT, vol.23, no.3, MAY 1977.

[24] L. Allison, T. Edgoose, T. I. Dix, "**Compression of Strings with Approximate Repeats**", **School** of Computer Science and Software Engineering, Monash University, Australia, 1998.

[25] Lempl, A. and Ziv, J., "**Compression of Individual Sequences Via Variable- rate Coding**", IEEE trans. IT,vol(24) 1978,pp 530-536.

[26] Michael Burrows and David J. Wheeler, "**A Bock-Sorting Lossless Data Compression Algorithm**", Digital Systems Research Center, Palo Alto, California, USA, May 1994.

[27] Milosavljevic, A.; and Jurka, J. "**Discovering Simple DNA Sequences by the Algorithmic Significance Method**". Comp. Appl. BioSc, 1993

[28] Marcelo J. Weinberger, Gadiel Seroussi, and Guillermo Sapiro, "**A Low Complexity, Context-Based, Lossless Image Compression Algorithm**", Hewlett-Packard Laboratories, Palo Alto, 1997.

[29] N. Jesper Larsson, "**Structures of String Matching and Data Compression**", Department of Computer Science, Lund University, 1999.

[30] Pedro V. Sander, "**Context-Based Color Image Compression**", Computer Science 276r, Harvard University, Spring 2000.

[31] Peter Fenwick, "**Block sorting text compression**", Proceedings of the 19th Australasian Computer Science Conference (Melbourne, Australia), January–February 1996.

[32] Rivals, È., Delahaye, J.-P., Dauchet, M., and Delgrange, O., "**A Guaranteed Compression Scheme for Repetition DNA Sequences**", LIFL Little I University, Technical report, 1995.

[33] Rivals, È., Delegrange, O., Delahaye, J.-P., Dauchet, M., Delorme, M.-O., Hènaut, A., and Ollivier, E., "**Detection of Significant Patterns by Compression Algorithms**", the case of Approximate Tandom repeats in DNA sequences, CABIOS, vol. 13, no 2, 1997.

[34] S. Bandyopadhyay, C.A. Murthy, S.K. Pal, "**Pattern Classification using Genetic Algorithms**", Pattern Recog.Lett. 1995.

[35] Sharon M. Perlmutter, Pamela C. Cosman, Chien-Wen Tseng, Richard A. Olshen, Robert M. Gray, King C. P. Li and Colleen J. Bergin, "**Medical Image Compression**", and Vector Quantization, Department of Electrical and Computer, Engineering, University of California, 1998

[36] Smith W., "**Digital Signal Processing**" California Technical Publishing, 1999.

[37] Suzanne Bunton, "**On-Line Stochastic Processes in Data Compression**", Ph.D. thesis, Department of Computer Science and Engineering, University of Washington, December 1996.

[38] Wojciech Rytter, "**Application of Lempel-Ziv Factorization**", to the Approximation, of Grammar-Based Compression, Department of Computer Science, Liverpool University 2001.

[39] Wu, Q., Z. Xiong, Y. Wang, and K. R. Castleman, "**Wavelet_Based Lossy-to-Lossless Coding of Cytogenetic Images with Arbitrary**

Regions of Support", Proc. IEEE International Symposium on Intelligent Signal Processing and Communication Systems, 2000.

[40] Xin Chen, Ming Li, Binma and John Tromp, "**DNA Compress: Fast and Effective DNA Sequences Compression**", Bioinformatics journal, vol. 18, no. 12,December 2002.

[41] Ziv, and Lempel, A., "**Anew Universal Algorithm for Sequential Data Compression**", IEEE Trans. IT, vol. 23, no.3, 1977.

[42] Z. Michalewicz, "**Genetic Algorithms + Data Structures = Evolution Programs**", Springer-Verlag, New York, 1992.