Fall 1991

Error detection and correction in compressed data

Nadir Sezgin
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

Error Detection and Correction in Compressed Data

by

Nadir Sezgin

Thesis submitted to the Faculty of the Graduate School
of the New Jersey Institute of Technology in partial
fulfillment of the requirements for the degree of
Master of Science in Electrical and Computer Engineering
1991

APPROVAL SHEET

Title of Thesis: Error Detection and Correction in Compressed Data

Name of Candidate: Nadir Sezgin
Master of Science in Electrical and Computer Eng.,
1991

Thesis and Abstract Approved:

Dr. Y. Bar-Ness Date
Distinguished Professor
Department of Electrical and Computer Engineering

Dr. C. Lu Date
Associate Professor
Department of Electrical and Computer Engineering

Dr. A. Akansu Date
Assistant Professor
Department of Electrical and Computer Engineering

VITA

Name: Nadir Sezgin

Degree and date to be conferred: M. Sc., Dec 16, 1991.

Collegiate institutions attended:	Date	Degree	Date of Degree
New Jersey Institute of Technology	9/90-12/91	M. Sc.	Dec 1991
Middle East Technical University	9/85-5/90	B.Sc.	May 1985

Major: Electrical Engineering.

Abstract

Error Detection and Correction in Encoded Data

Nadir Sezgin, Master of Science in Electrical Engineering

Thesis directed by Dr. Yeheskel Bar-Ness

Encoded data is very sensitive to the channel errors. Especially if the data is compressed by Arithmetic Encoding procedure, then the error propagation is very high. The error propagation in Arithmetic Coding is studied. Exploiting the high error propagation property when compressing data by Arithmetic encoding procedure, two different algorithms have been proposed for error detection and correction. Under certain conditions these algorithms detect and with a very high probability correct the errors introduced to the compressed data.

to my parents and dear brother...

Acknowledgement

I would like to express my appreciation to my supervisor Dr. Bar-Ness for his support, knowledge and insight during my study in NJIT. His support was far beyond of reasonable expectations.

I would also like to express my thankfulness to Dr. Lu and Dr. Akansu for their valuable suggestions for improving the quality of my thesis.

Many thanks to the members of Communication Lab., especially my dear friends Abdulkadir Dinç, Hakan Çağlar and Dr. Kim.

Contents

List of Figures

Chapter 1

Introduction

Data compression is an important topic of information theory. It broadly can be defined as transformation of a large volume of data into a smaller volume. Data compression is needed because of large space occupation of uncompressed data. Especially transmission of data, from one place to another place requires data compression. The first implementation of data compression was introduced by Shannon[1] and Fano [2]. Then Huffman[3] proposed a coding technique which gives optimum average code length. But all of these coding techniques require the knowledge of source statistics before the encoding process. Therefore these techniques are not applicable to the data sources with unknown statistics.

Davisson[4] proved that an optimum coding scheme can be designed without priori knowledge of the source statistics. These type of coding is known as *universal coding.* The most up to date universal coding techniques are Lampel-Ziv[5][6][7] coding, LZW coding[8] and arithmetic coding algorithms[9][10].

For convenience of the reader, these data compression techniques are discussed in the next chapter.

Because of the noisy nature of the communication channels, we cannot expect an error-free transmission from transmitter side to receiver side. There are several channel coding techniques[11]. But all of these techniques introduce signifi-

cant amount of redundancy to the data in the receiver side. This process contradicts with our goal of reducing the volume of source data.

Transmission of encoded data through communication channels, is more vital than the transmission of un-encoded data. Because encoded data is very intolerant to the channel errors. Even a single error in the encoded source may destroy most of the data following the error. The amount of data decoded incorrectly following an error is defined as error propagation.

Error propagation, depends on the data compression technique used for encoding the source data. Narasimhan[12] in his Master thesis studied the error propagation in the two widely-known universal data compression algorithms, namely LZW coding and arithmetic coding algorithms. He showed that error propagation in arithmetic coding algorithm is higher than it is in LZW coding algorithm.

In this thesis we proposed two different algorithms whose purpose to detect and correct the errors introduced to the encoded data in the channel. For the first algorithm the source data must be a text file. The second algorithm does not bring any constraints on the type of the source data. For both algorithms we took the advantage of very high error propagation in arithmetic coding scheme. In order to be able to understand the nature of the error propagation in arithmetic coding, in Chapter 3 the arithmetic coding procedure is explained in detail. A mathematical analysis of the error propagation in the arithmetic coding and the proposed algorithms are given in Chapter 4. Finally in Chapter 5 the results obtained from the simulation of these algorithms are discussed.

Chapter 2

Data Compression

Data can be represented as a sequence of symbols. The source symbols may be digital or analog which are drawn from an alphabet. Hence we can classify data sources as analog data sources and digital data sources. However, by "analog to digital" conversion we can represent the analog data in digital format. Therefore we will deal with only digital data. Some examples of digital data are English text, numerical data, image or video, speech, music, programming language source or object code etc.

Almost all kind of data contain significiant amount of redundancy. We start by defining redundancy in source data.

2.1 Redundancy

We can think of a data set as made up of information and redundancy. Information is the part of the data we wish to preserve on the other hand redundancy is the overhead in the source data we can get rid of. Without redundacy, we can reconstruct the original data without loosing any information. Therefore during the data transmission or data storage we do not need to transmit or store the redundancy. Even though we do not normally transmit reudundancy in a data compression system, it is often necessary to reinsert it at the receiving end simply because we may

not recognize the original data without redundant information. As an example in a scanned and digitized picture, for each scan line of the picture, we have regions that contain the same picture elements or pixels. If we represent these uniform regions on the scan lines with the value of the one pixel and run-length of the same pixels, then we reduce the redundant information in the scan lines. However if we represent a picture by such a representation, we will not be able to recognize the original one.

In general, in all kind of data sources there are four kind of redundancy[8]. These are as follows:

Character Distribution

In a typical character string some characters are used more frequently than the others. To be specific, in eight bit ASCII representations nearly three-fourths of the possible 256 bit combinations may not be used in a specific file. Therefore, each character used in file can be represented by 6 bits instead of 8 bits. Hence the text file volume might be reduced by 25%.

Character Repetition

When a string of repetitions of a single character occurs, the message can usually be represented in a more compact form if we write the single character followed by a number which indicates the number of repetition.

High Usage Patterns

Certain sequences of characters appears with relatively high frequency in a text file and therefore can be represented with relatively fewer bits. For example in English text files, a period followed by two spaces is more common than most other

three-character combinations and could therefore be represented by using fewer bits.

Positional Redundancy

If certain characters appear consistently at a predictable place in each block of data, then they are at least partially redundant. An example of this is a roster scanned picture containing a vertical line appears as a blip in the same position in each scan, and could be more compactly coded.

These four types of redundancy mentioned above, overlap to some extend in a data source.

2.2 Data Source Models

In 1948 C. E. Shannon[1], modelled the data source mathematically. He defined the information content of a source by a probability function as follows:

Let a is a symbol used in the data file and let $p(a)$ be the probability of occurance of a in the file, then the event of "occurance of the symbol a" gives,

$$I = -logp(a) \tag{2.1}$$

amount of information. The reason why *log* function is chosen is explained in great detail in the referred paper. The base of *log* function is not important, it only scales the amount of the information. For convention if we choose the base as 2 then the unit of information is *bit*.

The average information content of the source file with c different symbols is then given by,

$$H = -\sum_{k=1}^{c} p(a_k)logp(a_k) \tag{2.2}$$

where $a_1, a_2, \ldots, a_c$ are the symbols used in the source file. The average information content of a source file ,(2.2), is called the "entropy" of the file. Entropy is important

for comparing the amount of information that the files have. Again if we use base 2 for *log* function then the unit of entropy is *bits/symbol*. Hence it indicates that in average each symbol used in the source file can be represented by that many bits. Therefore the entropy of the source file determines the maximum amount of compressibility of the file (i.e. we cannot achieve a better compression)

Data sources are customarily represented in two mathematical models. These are *Markov dependent* and *statistically independent* source models.

2.2.1 Markov Dependent Model

For the Markov model, each sample depends on some number of contiguous previous samples. The depth of dependency gives the order of Markov model. If we know this dependency, we can predict the next sample. All we have to encode then will be the information about our prediction to allow the receiver to reconstruct the original sample sequence. As an example, let us consider the first order Markov model:

Let s_k and s_{k+1} be kth and $(k+1)$st characters from the data file, and let $p(s_k)$ and $p(s_{k+1})$ be the probabilities of their occurance. Then for this source model,

$$p(s_k, s_{k+1}) = p(s_k)p(s_{k+1}|s_k) \tag{2.3}$$

where $p(s_k, s_{k+1})$ is the joint probability and $p(s_{k+1}|s_k)$ is the conditional probability of occurance of source characters s_k and s_{k+1}.

2.2.2 Statistically Independent Model

For the case of the statistically independent model, we cannot predict and eliminate dependent symbols. Because all the symbols are assumed to be independent from each other. Therefore if the kth character of the data source,s_k, is known, it does not give any information about the occurance of the next character, s_{k+1}. For this

type of data,

$$p(s_k, s_{k+1}) = p(s_k)p(s_{k+1}) \tag{2.4}$$

and hence,

$$p(s_{k+1}|s_k) = p(s_{k+1}) \tag{2.5}$$

For compression we take the advantage of the non uniform probability distribution of the source symbols used in the data source. If this distribution were uniform then entropy would be a maximum, equal to

$$H = log_2 c \tag{2.6}$$

where c is the number of different symbols used in the source data. But real sources rarely have uniform probability distributions. Thus entropy is always less than $log_2 c$. Then we can define redundancy mathematically as,

$$Redundancy = (log_2 c) - H \tag{2.7}$$

and therefore zero redundancy source has

$$H = log_2 c \tag{2.8}$$

It should be emphasized that we are dealing with discrete sources and for these sources the uniform distribution of symbols maximizes the entropy. For the continuous sources Gaussian distribution maximizes the source entropy.

2.3 Data Compression Techniques

Data Compression is the process of encoding a body of data D into a smaller body of data $\Delta(D)$. It must be possible for $\Delta(D)$ to be decoded back to, or close to, D[13].

Basically we can compress the data by two different methods:

The first is the *entropy reduction* technique. In this method the compression is achieved by somewhat distorting the original data. It also called irreversible or lossy compression. The apparent example to this technique is quantizing an analog source, since once we quantize the amplitude values, we can never reconstruct them exactly.

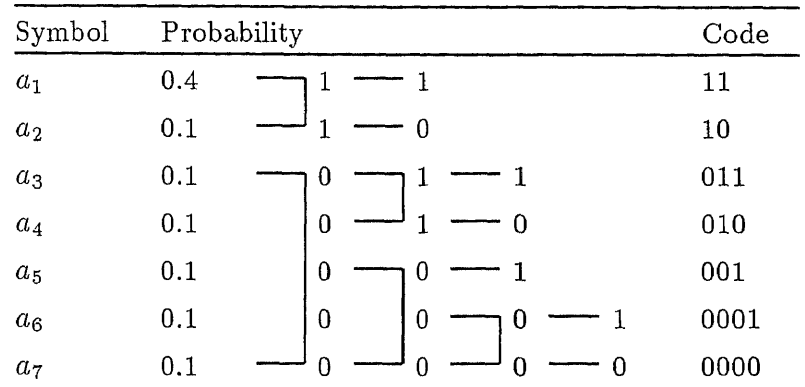
The second is the *redundancy reduction* technique. In this method we remove or reduce the portion of data which can be reinserted or reconstituted at the receiving end of the system with no distortion. For this reason, redundancy reduction has been called, in the literature, reversible coding, transparent coding or lossless coding[14].

Since we do not want to loose any information in the original data source we will only deal with redundancy reduction techniques.

Both the statistically independent and the Markov dependent source models are used in analyzing and designing data compression schemes for discrete sources. Neither model however is very realistic. A source with statistically independent symbols is not very common, nor is a source in which inter-symbol dependency is over only one previous sample.

A situation often arises in which we have insufficient information about the source to enable us to choose an appropriate model for it. In this case we design a compression scheme that is independent in its performance of source statistics. Such a code is called a "universal code".

We will only deal with statistically independent data sources. For this kind of sources the coding that makes the average word length approach the entropy is often referred to as "optimum source coding" or "entropy coding". Extension to Markov dependent sources is possible.



Enropy = 2.529 bits

Avg. Code Lenght = 2.7 bits

Figure 2.1: Shannon-Fano Coding Procedure

2.3.1 Shannon-Fano Coding

C. E. Shannon and R. M. Fanno[2] have developed source coding procedures in which the average number of binary digits required per symbol approaches the average amount of information of the source. The Shannon-Fano coding procedure provides a means of constructing reasonably efficient codes with instantaneous decodebility. Efficiency is defined here as,

$$Efficiency = \frac{H}{\bar{l}} \times 100 \tag{2.9}$$

where $\bar{l}$ is the average length of the codeword. Shannon-Fano code reaches an efficiency of 100% only when the source symbol probabilities are negative powers of 2. The coding procedure is as follows:

1. Arrange the source message probabilities in descending order.

2. Divide the message set into two subsets of equal or almost equal, total probability and assign a "zero" as first code digit to one subset, and a "one" as the first code digit to the second subset.

Symbol	Probability		Code
a_1	0.5	0	0
a_2	0.25	0 1	10
a_3	0.125	0 1	110
a_4	0.125	1	111

Figure 2.2: Huffman Code

3. Continue this process until each subset contains only one message.

The process of coding is depicted in Figure 2.1. This coding procedure is not optimum but approach the optimum behavior when the source length approaches infinity. Kraft[15] has derived a coding method which gives an average code length as close as possible to optimum when the source contains a finite number of members. In 1952 D. A. Huffman[3] realized such a code which is known as *Huffman Code*. This code is detailed in the next section.

2.3.2 Huffman Coding

Consider ,for example, four symbol alphabet $A = \{a_1, a_2, a_3, a_4\}$ in which the probability of occurance of the symbols are $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{8}$ respectively. With Huffman encoding, we first order the symbols according to their probability and generate the code as in Figure 2.2. Notice that as the Shannon-Fano code, this code is also a variable length code.

When using this code ,for example, in encoding the data string "$a_1a_1a_2a_4a_3a_2$" we get "0.0.10.111.110.10" where "." is used as a delimiter to show the substitution of the codeword for the symbol. It is easy to see that the code has a prefix property(i.e. no codeword is the prefix of another).

Decoding is performed by a matching or comparison process, starting with the first bit of the code string. For decoding of the code string 001011111010 the first

symbol is decoded as a (the only codeword beginning with 0). We remove codeword 0 and the remaining code is "01011111010". The second symbol is similarly decoded as a_1 leaving 1011111010. For the string 1011111010, the only codeword beginning with 10 is a_2, so we are left with 11111010. Continuing in this manner, we decode the remaining data string as a_4, a_3 and a_2.

One disadvantage of Huffman's coding is that it needs two passes over the source data: one pass to collect frequency counts(probability) of the characters in the message, from which we construct a Huffman tree that is also transmitted to the receiver. In the second pass, the tree is used to encode the characters. Clearly such process causes delay when used in network communication, and for the file compression applications the extra access to the memory can slow down the algorithm.

Faller[16] and Gallager[17] independently proposed a one-pass algorithm for Huffman coding which is called, in the literature, *dynamic Huffman coding.* This algorithm, later improved by Knuth[18]

In dynamic Huffman coding, the binary tree that the encoder uses to encode the $(k+1)$st letter in the message (and that the decoder uses to reconstruct the $(k+1)$st letter) is a Huffman tree for the first k letters of the message. Both encoder and decoder start with the same initial tree and thereafter stay synchronized; they use the same algorithm to modify the tree after each letter is processed. Thus there is no need for the encoder to communicate the tree to the decoder.

2.3.3 Lampel- Ziv Coding

In 1976 J. Ziv and A. Lampel[6] proposed a universal coding algorithm. The algorithm, basically matches the maximum length recognized data string to a fixed length codeword. The encoded codeword consists of the buffer address and the source symbol.

The algorithm is also known as "incremental parsing" algorithm. It parses the source string into a collection of substrings. The length of substrings increases gradually. Both encoder and decoder generate a dictionary which is initially empty. Each time a substring is encoded, it is also put into the encoder dictionary as a new entry. Similarly each time a substring is decoded, it is put into the decoder dictionary. Hence encoder and decoder dictionaries are the same at each stage of encoding and decoding processes.

As an example assume the source alphabet is $A = \{a_1, a_2, a_3\}$,and the source string is "$a_1a_2a_2a_3a_1a_2a_2a_3a_3$" Initially the dictionary is empty. Then the source string is parsed into substrings as $\{a_1, a_2, a_2a_3, a_1a_2, a_2a_3a_3\}$. Each substring in the dictionary can be encoded as (x, y), where x is the address of the longest matched substring encoded before, and y is the last source symbol concatenated to this substring. For example "$a_2a_3a_3$" is encoded as $(3, a_3)$. Because the address of the longest matching substring, "a_2a_3", in the dictionary is 3 and the concatenated symbol to this substring is "a_3".

Decoding algorithm, undoes what encoding algorithm does. The dictionary for the decoder is also empty initially. The decoder builds its own dictionary during the decoding process. The strategy of building the dictionary is the same as encoder. Therefore at every stage of encoding and decoding process, both encoder and decoder have exactly the same dictionary.

One disadvantage of this algorithm is in the fact that, both encoder and decoder need a large amount of memory for constructing their dictionaries. Another disadvantage is it gives a poor compression for short files or the files with rapidly changing characteristics.

Substring	Address
a_1	1
a_2	2
a_3	3
a_1a_2	4
a_2a_1	5
$a_1a_2a_3$	6
a_3a_1	7
$a_1a_2a_2$	8
a_2a_3	9
$a_3a_1a_2$	10

Figure 2.3: Dictionary for LZW Algorithm

2.3.4 LZW Coding

This is an improved version of Lampel-Ziv coding[8]. The difference of this coding method from the Lampel-Ziv method is, initially encoder and decoder have the same dictionary which is *not empty*. The dictionary is initialized with the source data alphabet. In comparison to Lampel-Ziv coding, instead of using (x, y); the address of the longest matched substring and a raw character, in the encoding process, encoder only encodes the address of the longest matching substring x. The new encoded substring is added to the dictionary and the last character of the new substring becomes the first character of the next substring.

Decoding algorithm is also very simple. Each time, the decoder handles an address of the encoded substring it can accurately decodes it since it also has the same dictionary as encoder. The last character of the new substring is the first character of the next substring. Hence the decoder can construct the new substring and add this to its dictionary.

As an example consider the same data source alphabet, $A = \{a_1, a_2, a_3\}$, giv-

en previously. Initially encoder and decoder have the same dictionary as $\{a_1, a_2, a_3\}$. To encode the data string "$a_1a_2a_1a_2a_3a_1a_2a_2a_3a_1a_2a_3a_1$" we start with "$a_1a_2$". The longest matching substring in the dictionary is "a_1". Therefore we encode the address 1 and add "a_1a_2" to the dictionary in address 4, see Figure 2.3. The next substring is "a_2a_1". The longest matching string is "a_2" whose address is 2. Hence we encode 2 as the address for the next substring. Continuing in this manner gives the codeword of "12434279". The construction of dictionaries is shown in Figure 2.3.

Chapter 3

Arithmetic Coding

The first step toward *arithmetic coding* was taken by Shannon[1]. He observed in the referred paper, that messages N symbols long can be encoded by first sorting the symbols in the order of their probabilities and then sending the cumulative probability of the preceding symbols. The next step was taken by Peter Elias in an unpublished result. Abramson[19] described Elias' improvement in 1963. He observed that Shannon's scheme worked without first sorting the symbols according to their probabilities. Instead the cumulative probability of a source of N characters could be recursively calculated from the individual symbol probabilities and the cumulative probability of the message of the $N-1$ characters. Elias' code was studied by Jelinek[20]. The codes of Shannon and Elias had a serious problem: When the source length increases, it is not possible to represent the codeword by finite precision arithmetic. These codes can be viewed as a mapping of data string to a number which is the codeword. Rissanen[21] over come the precision problem by suitable approximations in designing a "last in first out" arithmetic code. In his algorithm code string of any length could be generated with a fixed precision arithmetic. Pasco[22] instead suggested a "first in first out" arithmetic code which controlled the precision problem in similar idea to that proposed by Risannen. His idea was to use floating point arithmetic to overcome the requirement for unlimited

precision of Elias code.

3.1 Encoding

We can look upon arithmetic encoding as a transformation, which maps the data source strings to a point, in the interval $[0, 1]$. The transformation to the interval $[0, 1]$ depends on the cumulative probabilities of the source symbols.

For encoding operation, we need to define two parameters; one is the code point C, and the other is the code interval W. C represents the leftmost point of the interval. On the other hand W represents the width of the interval.

To make the process of encoding clear; assume, for example, we have a source alphabet of size c, $A = \{a_1, a_2, \ldots, a_c\}$. That is the source data is constructed by c different symbols. Further assume that we have a code alphabet of size d, $B = \{b_1, b_2, \ldots, b_d\}$. Let $p(a_i)$ be the probability of occurance of the different symbols in the source alphabet. Then for each symbol, a_i, we define cumulative probability by

$$P(a_i) = \sum_{k=1}^{i-1} p(a_k) \tag{3.1}$$

where $i = 1, 2, \ldots, c$ and $P(a_1) = 0$.

Code Point

The new leftmost point of the new interval is the sum of the current code point C, and the product of the interval width W of the current interval and the cumulative probability $P(a_i)$ for the symbol a_i, being encoded:

$$New\ C = Current\ C + Current\ W \times P(a_i) \tag{3.2}$$

Code Interval

The width of current code interval W is the product of the probabilities of the data symbols encoded so far. Thus the new interval width is

$$New\ W = Current\ W \times p(a_i) \tag{3.3}$$

where the current symbol is a_i.

To encode a source file, we have to find code point C and code interval W for each character in the source data. Assume that after encoding kth character in the source file, the code point is C_k and the code interval is W_k.

We start encoding process with initial values code point and code interval which are

$$C_0 = 0 \tag{3.4}$$

and

$$W_0 = 1 \tag{3.5}$$

For the first source character, we have

$$C_1 = C_0 + W_0 P(s_1) \tag{3.6}$$

where s_1 is the first character of the source file and $s_1 \in \{a_1, a_2, \ldots, a_c\}$

and

$$W_1 = W_0 p(s_1) \tag{3.7}$$

Since $C_0 = 0$ and $W_0 = 1$ then above equations become

$$C_1 = P(s_1) \tag{3.8}$$

and

$$W_1 = p(s_1) \tag{3.9}$$

To encode the second character in the source file we have

$$C_2 = C_1 + W_1 P(s_2) \tag{3.10}$$

and

$$C_2 = P(s_1) + p(s_1)P(s_2) \tag{3.11}$$

where $s_2 \in \{a_1, a_2, \ldots, a_c\}$.

Continuing in the same manner, after encoding the kth character in the source file we have

$$C_k = C_{k-1} + W_{k-1} P(s_k) \tag{3.12}$$

and

$$W_k = W_{k-1} p(s_k) \tag{3.13}$$

However since,

$$C_{k-1} = C_{k-2} + W_{k-2} P(s_{k-1}) \tag{3.14}$$

then equation (3.12) becomes,

$$C_k = C_{k-2} + W_{k-2} P(s_{k-1}) + W_{k-1} P(s_k) \tag{3.15}$$

But

$$W_{k-1} = W_{k-2} P(s_{k-1}) \tag{3.16}$$

then equation (3.13) becomes

$$W_k = W_{k-2} p(s_{k-1}) p(s_k) \tag{3.17}$$

Continuing in this manner we get

$$C_k = W_0 P(s_1) + W_1 P(s_2) + \cdots + W_{k-2} P(s_{k-1}) + W_{k-1} P(s_k) \tag{3.18}$$

and

$$W_k = W_0 p(s_1) p(s_2) \cdots p(s_{k-1}) p(s_k) \tag{3.19}$$

Substituting $W_0 = 1$ in equation (3.19) we get,

$$W_k = p(s_1)p(s_2)\cdots p(s_{k-1})p(s_k) \tag{3.20}$$

And finally substituting (3.20) into (3.18), we get the code point

$$C_k = P(s_1) + p(s_1)P(s_2) + p(s_1)p(s_2)P(s_3) + \cdots + p(s_1)p(s_2)\cdots p(s_{k-2})p(s_{k-1})P(s_k) \tag{3.21}$$

or

$$C_k = P(s_1) + p(s_1)[P(s_2) + p(s_2)[P(s_3) + \cdots + p(s_{k-2})[P(s_{k-1}) + p(s_{k-1})P(s_k)]\cdots]] \tag{3.22}$$

From the above equation it is clear that

$$C_k \geq P(s_1) \tag{3.23}$$

The term in the innermost parenthesis,

$$P(s_{k-1}) + p(s_{k-1})P(s_k) \leq P(s_{k-1}) + p(s_{k-1}) \tag{3.24}$$

since $P(s_k)$ is less than 1. We notice that

$$P(s_{k-1}) + p(s_{k-1}) = P(s_{k-1} + 1) \tag{3.25}$$

(i.e. if $s_{k-1} = a_j$ then $s_{k-1} + 1 = a_{j+1}$ in the source alphabet). But $P(s_{k-1} + 1)$ is also less than 1. Continuing in this manner, we can verify that the outermost parenthesis content is less than 1. Therefore

$$C_k < P(s_1) + p(s_1) \tag{3.26}$$

From equations (3.23) and (3.26) we get

$$P(s_1) \leq C_k < P(s_1) + p(s_1) = P(s_1 + 1) \tag{3.27}$$

where $P(s_1)$ is the cumulative probability of the first source character.

Assuming the first source character is the symbol a_j from the source alphabet then $P(s_1 + 1)$ represent the cumulative probability of the next symbol a_{j+1} from the source alphabet.

We can conclude that the code point falls into the interval $[a_i, a_{i+1})$ no matter how long the source string is. The value of the code point, representing the source string depends on the cumulative probabilities of the string characters used in the source string. As an extreme case if the source string is "$a_i a_i a_i \cdots$", then the code point is $P(a_i)$. On the other hand if the source string is "$a_i a_{i+1} a_{i+1} \cdots$", then the code point is very close to $P(a_{k+1})$, but not equal to $P(a_{k+1})$. All other source strings begin with the source symbol a_i, are encoded to the interval $[a_i, a_{i+1})$ corresponding to the code point C_k such that $P(a_i) \leq C_k < P(a_{i+1})$.

3.1.1 An Encoding Example

As an example consider again the four symbol alphabet, $A = \{a_1, a_2, a_3, a_4\}$, with the relative frequencies 0.5, 0.25, 0.125 and 0.125 respectively. This is the same alphabet that we used for demonstrating Huffman Coding in Figure 2.2. We relate the arithmetic coding to the process of subdividing the unit interval successively.

Symbol	Probability (Code Interval)	Cumulative Probability (Code Point)
a_1	0.5	0
a_2	0.125	0.5
a_3	0.25	0.625
a_4	0.125	0.875

Figure 3.1: Arithmetic Encoding

Define the "code point" as the sum of the probabilities of the proceeding symbols, and "code interval" as the probability of the sequence encoded so far. If we rearrange the the table in Figure 2.2 then we get the table shown in Figure 3.1.

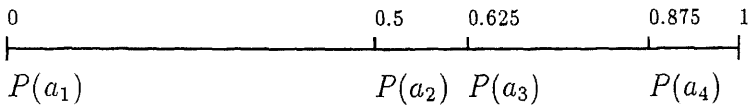


Figure 3.2: Code Points

We now view the code points as fractional values 0, 0.5, 0.625 and 0.875. Notice that these code-points are actually the sum of the probabilities of the preceding symbols for each symbol. In other words each code-point is a cumulative probability P.

Further we can view the codewords as code-points on the number line from 0 to 1, or the unit interval, as shown in Figure 3.2. The four code points divide the unit interval into four subintervals. We identify each subinterval with the symbol corresponding to its leftmost point. For example the interval for symbol "a_1" goes from 0 to 0.5, and for symbol "a_4" goes from 0.875 to 1. Note also that the width or size of the subinterval to the right of each code point corresponds to the probability of the symbol. The codeword for the symbol "a_1" has 0.5 the interval, the codeword for "a_3" has 0.25 the interval and "a_2" and "a_4" each have 0.125 of the interval.

Again suppose that we want to encode the data string "$a_1a_1a_2a_3$" as we did for Huffman encoding. The first symbol is "a_1". From equations (3.4) and (3.5) the initial values of code point and code interval are

$$C_0 = 0 \text{ and } W_0 = 1$$

From the equations (3.2) and (3.3) the code point and code interval for the first

data character, $s_1 = a_1$, are calculated as follows

$$\begin{aligned} C_1 &= C_0 + P(s_1)W_0 \\ &= 0 + 0 \times 1 \\ &= 0 \end{aligned}$$

and

$$\begin{aligned} W_1 &= p(s_1)W_0 \\ &= 0.5 \times 1 \\ &= 0.5 \end{aligned}$$

The corresponding interval on the unit interval is $[0, 0.5)$. This means that the fractions equal to or greater than 0, but less than 0.5 are in the interval. The code point and the code interval for the second source character, $s_2 = a_1$, are

$$\begin{aligned} C_2 &= C_1 + P(s_2)W_1 \\ &= 0 + 0 \times 0.5 \\ &= 0 \end{aligned}$$

and

$$\begin{aligned} W_2 &= p(s_2)W_1 \\ &= 0.5 \times 0.5 \\ &= 0.25 \end{aligned}$$

The interval for "$s_1 s_2$" is $[0, 0.25)$. Similarly, the code point and code interval for the third character, $s_3 = a_2$ of the data string are calculated as

$$\begin{aligned} C_3 &= C_2 + P(s_3)W_2 \\ &= 0 + 0.5 \times 0.25 \\ &= 0.125 \end{aligned}$$

and

$$\begin{aligned} W_3 &= p(s_3)W_2 \\ &= 0.125 \times 0.25 \\ &= 0.03125 \end{aligned}$$

Hence the interval for "$s_1 s_2 s_3$" is [0.125, 0.15625). Finally for the last character, $s_4 = a_3$, of the data string we have,

$$C_4 = 0.14453125$$

and

$$W_4 = 0.0078125$$

Then the data string "$s_1 s_2 s_3 s_4$" (i.e. "$a_1 a_1 a_2 a_3$") corresponds to the codeword 0.14453125.

3.2 Decoding

For decoding the first step is comparing the code point, C_k, with the cumulative probabilities of the source symbols. The first decoded character is the source symbol which has the largest cumulative probability less than or equal to C_k. Then to find the code point for the second decoded symbol; first we subtract $P(s_1)$ from C_k:

$$C_k - P(s_1) = p(s_1)P(s_2) + p(s_1)p(s_2)P(s_3) + \cdots + p(s_1)p(s_2)p(s_3)\cdots p(s_{k-1}P(s_k) \tag{3.28}$$

then we divide $C_k - P(s_1)$ by $p(s_1)$ to get the new code point for decoding the second symbol:

$$C_k^{(2)} = P(s_2) + p(s_2)P(s_3) + p(s_2)p(s_3)P(s_4) + \cdots + p(s_2)p(s_3)\cdots p(s_{k-1})P(s_k) \tag{3.29}$$

where $C_k^{(2)}$ represents the new code point to decode the second source character.

In order to decode the second source data character we compare $C_k^{(2)}$ with the cumulative probabilities of the source data symbols. The symbol with the largest cumulative probability which is smaller than or equal to the code point is the second decoded character.

Similarly the code point for decoding the third source character,

$$C_k^{(3)} = P(s_3) + p(s_3)P(s_4) + p(s_3)p(s_4)P(s_5) + \cdots + p(s_3)p(s_4)\cdots p(s_{k-1})P(s_k) \tag{3.30}$$

Continuing in this manner, for decoding the last source character we have:

$$C_k^{(k)} = P(s_k) \tag{3.31}$$

which is the cumulative probability of the kth source character, and the decoding process is complete.

3.2.1 A Decoding Example

Decoding procedure is basically the reverse procedure of the encoding process. That is decoder undoes whatever the encoder does. Considering the same example that we have given for encoding process the encoded string 0.14453125. This code point is in the interval [0, 0.5), therefore, the first data symbol, s_1, must be "a_1".

From Figure 3.2, $P(s_1) = 0$ and $p(s_1) = 0.5$. Using this in equation (3.28) we get

$$\begin{aligned} C_k^{(2)} &= \frac{0.14453125 - 0}{0.5} \\ &= 0.2890625 \end{aligned}$$

This point is again in the interval [0, 0.5) therefore the second decoded character is also a_1. For this character, $P(s_2 = 0$ and $p(s_2) = 0.5$. Similarly the third code point is found as

$$\begin{aligned} C_k^{(3)} &= \frac{0.2890625 - 0}{0.5} \\ &= 0.578125 \end{aligned}$$

This point is in the interval [0.5, 0.625) and hence the third decoded character is a_2, with $P(s_3) = 0.5$ and $p(s_3) = 0.125$. Hence the fourth code point is found as

$$\begin{aligned} C_k^{(4)} &= \frac{0.578125 - 0.5}{0.125} \\ &= 0.625 \end{aligned}$$

This is exactly the code point which correspond to a_3, the last symbol decoded. The decoded string is thus "$a_1 a_1 a_2 a_3$.

3.3 Integer Technique

In conventional arithmetic coding the interval of decoding becomes more finely divided as more symbols are included in the source sequences. The capacity of the coder to accept more source symbols is limited by the ability of its fixed registers to resolve the boundaries between intervals. The capacity of the system is greatly extended by the adoption of what is essentially floating point representation of these boundaries. This method is implemented by the explicit manipulation of integers[23].

In this case it is necessary to find an alternative representation of the probabilities $P(s)$ by applying a scale factor u that effectively converts a probability to a frequency rate per u source symbols.

Assume the *source alphabet* is $A = \{a_1, a_2, a_3, \ldots, a_c\}$ and the *code alphabet* is $B = \{b_1, b_2, b_3, \ldots, b_d\}$. Then define

α; as an arbitrary sequence of symbols (i.e. data source string),

αs_i; as concatenation of α and a new character s_i, where $s_k \in \{a_1, a_2, \ldots, a_c\}$,

$\lfloor x \rfloor$; as an integer such that $x - 1 < \lfloor x \rfloor \leq x$.

The boundaries of the interval corresponding to the source sequence are represented by tree functions, namely $X(\alpha)$, $Y(\alpha)$ and $L(\alpha)$.

$X(\alpha)$ and $Y(\alpha)$ are integers, represent the code point and code interval respectively. $L(\alpha)$ supplies an exponent $-L(\alpha) - w$ by which base d is raised to give a scale factor. The relation between the conventional code point, C_k and $X(\alpha)$ is as follows

$$C_k = \frac{X(\alpha s_k)}{d^{L(\alpha s_k)+w}} \tag{3.32}$$

Similarly, the relation between the conventional code interval, W_k, and $Y(\alpha)$ is defined as

$$W_k = \frac{Y(\alpha s_k)}{d^{L(\alpha s_k)+w}} \tag{3.33}$$

Further we define

F_{a_j}: the cumulative frequency of occurance of a_j in the source data file.

$$F_{a_j} = P(a_j) \times u \tag{3.34}$$

where u is a constant, chosen such that F_{a_j} is an integer for all a_j in the data source alphabet.

Hence we get,

$$P(a_j) = \frac{F_{a_j}}{u} \tag{3.35}$$

and

$$p(a_j) = \frac{F_{a_{j+1}} - F_{a_j}}{u} \tag{3.36}$$

Conventional code point and code interval equations, 3.12 and 3.13 can be represented as follows

$$\frac{X(\alpha a_j)}{d^{L(\alpha a_j)+w}} = \frac{X(\alpha)}{d^{L(\alpha)+w}} + (\frac{Y(\alpha)}{d^{L(\alpha)+w}})\frac{F_{a_j}}{u} \tag{3.37}$$

and

$$\frac{Y(\alpha a_j)}{d^{L(\alpha a_j)+w}} = (\frac{Y(\alpha)}{d^{L(\alpha)+w}})\frac{(F_{a_{j+1}} - F_{a_j})}{u} \tag{3.38}$$

Let $L(\alpha a_j) = L(\alpha) + l$ where l is an integer to be chosen to satisfy the accuracy requirement such that

$$d^w \leq Y(\alpha a_j) < d^{w+1}$$

where the parameter w determines the number of d-ary digits of precision used to represent the width of the interval. The scaling by d^l insures that smaller an interval becomes, its width is still represented with sufficient precision for further division. Then the equations (3.37) and (3.38) simplify as follows

$$X(\alpha a_j) = (X(\alpha) + Y(\alpha)\frac{F_{a_j}}{u})d^l \tag{3.39}$$

and

$$Y(\alpha a_j) = (Y(\alpha)\frac{F_{a_{j+1}} - F_{a_j}}{u})d^l \tag{3.40}$$

In order to be able to to represent $X(\alpha a_j)$ and $Y(\alpha a_j)$ as integers then we use $\lfloor x \rfloor$ function. To eliminate the effect of the rounding performed by $\lfloor x \rfloor$ function "$\frac{1}{2}$" is added to x. Hence we get

$$X(\alpha a_j) = (X(\alpha) + \lfloor Y(\alpha)\frac{F_{a_j}}{u} + \frac{1}{2} \rfloor)d^l \tag{3.41}$$

and

$$Y(\alpha a_j) = (\lfloor Y(\alpha)\frac{F_{a_{j+1}}}{u} + \frac{1}{2} \rfloor - \lfloor Y(\alpha)\frac{F_{a_j}}{u} + \frac{1}{2} \rfloor)d^l \tag{3.42}$$

The value of l needed for the above equation such that $Y(\alpha a_j)$ satisfies

$$d^w \leq Y(\alpha a_j) < d^{w+1}$$

can always be satisfied as long as

$$\lfloor Y(\alpha)\frac{F_{a_{j+1}}}{u} + \frac{1}{2} \rfloor > \lfloor Y(\alpha)\frac{F_{a_j}}{u} + \frac{1}{2} \rfloor \tag{3.43}$$

For this inequality to be true for any $Y(\alpha)$, F_{a_j} and u, it is sufficient to require

$$Y(\alpha)(F_{a_{j+1}} - F_{a_j}) \geq u$$

since $Y(\alpha) \geq d^w$ then it is sufficient to find l so that

$$d^w(F_{a_j} - F_{a_{j-1}}) \geq u$$

where

$$a_j, a_{j-1} \in \{a_1, a_2, \ldots, a_c\}$$

This is constrained value of u needed after the value of w has been decided upon.

Chapter 4

Error Propagation, Detection and Correction

4.1 Error Propagation

From the previous chapter, the transformation of data string "$s_1 s_2 s_3 \cdots s_k$" to the corresponding code word is given in equation (3.22) as

$$C_k = P(s_1) + p(s_1)P(s_2) + p(s_1)p(s_2)P(s_3) + \cdots + p(s_1)p(s_2)\cdots p(s_{k-1})P(s_k) \quad (4.1)$$

An error introduced to the codeword at this point, changes the value of C_k by an amount of Δ. Then erroneous codeword can be written as

$$C_k^{e_1} = P(s_1) + p(s_1)P(s_2) + p(s_1)p(s_2)P(s_3) + \cdots + p(s_1)p(s_2)\cdots p(s_{k-1})P(s_k) + \Delta \quad (4.2)$$

Depending on the size of Δ, $C_k^{e_1}$ may not fall into the interval $[P(s_1), P(s_1 + 1))$ which causes the next source character being incorrectly decoded. However if Δ is such that $C_k^{e_1}$ falls into this interval, that is

$$P(s_1) \leq C_k^{e_1} < P(s_1 + 1) \quad (4.3)$$

then the next source character is decoded correctly.

Assuming that Δ is a uniformly distributed random variable over the interval [0, 1] then $C_k^{e_1}$ is also a random variable over the same interval. That is $C_k^{e_1}$ can be any point in the interval [0, 1] with equal probability.

Therefore the probability of

$$C_k^{e_1} \in [P(s_1),\ P(s_1+1)) \tag{4.4}$$

is proportional with the length of code interval $[P(s_1),\ P(s_1+1))$ which is $p(s_1)$. Hence the probability of "first source character being incorrectly decoded" is

$$p(s_1') = 1 - p(s_1) \tag{4.5}$$

where $s_1 \in \{a_1, a_2, a_3, \ldots, a_c\}$ and s_1' indicates that s_1 is incorrectly decoded.

In average the the probability of first character being incorrectly decoded after introducing an error to the encoded source is

$$E\{s_1'\} = \sum_{n_1=1}^{c} p(a_{n_1})(1 - p(a_{n_1})) \tag{4.6}$$

Assume that the first source symbol is decoded correctly. To decode the second source symbol, from equations (3.29) and (4.2) we get

$$C_k^{e_2} = P(s_2)+p(s_2)P(s_3)+p(s_2)p(s_3)P(s_4)+\cdots+p(s_2)p(s_3)\cdots p(s_{k-1})P(s_k)+\frac{\Delta}{p(s_1)} \tag{4.7}$$

$C_k^{e_2}$ is compared with the cumulative probabilities of the source symbols.
If $C_k^{e_2} \in [P(s_2),\ P(s_2+1))$ then the second source character is also decoded correctly. the probability of "$C_k^{e_2} \in [P(s_2),\ P(s_2+1))$ given $C_k^{e_1} \in [P(s_1),\ P(s_1+1))$" is proportional to the length of new code interval $[P(s_2),\ P(s_2+1))$ which is $p(s_1)p(s_2)$.

Hence the the probability of "the second symbol will be incorrectly decoded, given the first symbol decoded correctly" is

$$p(s_2'|s_1) = 1 - p(s_1)p(s_2) \tag{4.8}$$

where $s_1, s_2 \in \{a_1, a_2, a_3, \ldots, a_c\}$.

In average this probability is

$$E\{s_2'|s_1\} = \sum_{n_1=1}^{c} \sum_{n_2=1}^{c} p(a_{n_1})p(a_{n_2})(1 - p(a_{n_1})p(a_{n_2})) \tag{4.9}$$

where we assume the statistical independence between the characters.

Continuing in the same manner, after k iteration we have the average probability of the event "the kth source symbol will be decoded correctly, given $k-1$ previous source symbols are decoded correctly" is

$$E\{s_k'|s_1 s_2 \cdots s_{k-1}\} = \sum_{n_1=1}^{c} \sum_{n_2=1}^{c} \cdots \sum_{n_k=1}^{c} p(a_{n_1})p(a_{n_2}) \cdots p(a_{n_k})(1-p(a_{n_1})p(a_{n_2}) \cdots p(a_{n_k})) \tag{4.10}$$

Finally from equations (4.6), (4.9) and (4.10) we can conclude that the the expected value of "decoding the source data incorrectly after an error being introduced to the encoded data" is

$$E\{error\} = E\{s_1'\} + E\{s_2'|s_1\} + \cdots + E\{s_k'|s_1 s_2 \cdots s_{k-1}\} \tag{4.11}$$

From equation (4.6),

$$E\{s_1'\} = 1 - \sum_{n=1}^{c} p^2(a_n) \tag{4.12}$$

From equation (4.9),

$$\begin{aligned} E\{s_2'|s_1\} &= 1 - \sum_{n_1=1}^{c} \sum_{n_2=1}^{c} p^2(a_{n_1})p^2(a_{n_2}) \\ &= 1 - [\sum_{n=1}^{c} p^2(a_n)]^2 \end{aligned} \tag{4.13}$$

Similarly, from (4.10)

$$E\{s_k'|s_1 s_2 \cdots s_{k-1}\} = 1 - [\sum_{n=1}^{c} p^2(a_n)]^k \tag{4.14}$$

Substituting equations (4.12) through (4.14) into equation (4.11) we obtain

$$\begin{aligned} E\{error\} &= k - [\sum_{n=1}^{c} p^2(a_n) + (\sum_{n=1}^{c} p^2(a_n))^2 + \cdots + (\sum_{n=1}^{c} p^2(a_n))^k] \\ &= k - \sum_{n=1}^{c} p^2(a_n)[\frac{1 - [\sum_{n=1}^{c} p^2(a_n)]^k}{1 - \sum_{n=1}^{c} p^2(a_n)}] \end{aligned} \tag{4.15}$$

which is a measure of error propagation in the arithmetic coding.

It is clear that for k not necessarily very large, $E\{error\} = k$. That is with probability almost one, all the characters after introducing the error will be incorrectly decoded.

4.2 Error Detection and Correction

When compressed data is transmitted through a communication channel, depending on the characteristics of the channel, errors are generated. However, since the redundancy in the source data is removed by the data compression algorithm, the encoded data is very sensitive to channel errors. A single error introduced to compressed data garbles a significiant amount of decoded data at the receiver end.

In this section we describe two different algorithms for detecting and correcting the errors in the compressed data. Both algorithms exploit the advantage of high error propagation in the compressed data. We have chosen Arithmetic Coding as the data compression technique because of the very high error propagation inherent to this technique.

The first algorithm can be used only for text files. However the second algorithm can be used for any kind of data representation. Another drawback of the first algorithm is in using the assumption that if a word is a new word (i.e. it occurs for the first time in the source data) then the corresponding codeword is error-free. However unless we use some initialization at the beginning of the source file most of the words are "new". Experimental results showed that this assumption may hold for very big files, if the errors are not introduced at the beginning of the file.

With the second algorithm we do not impose any restriction on the position of the errors in the encoded file. We assumed that the errors are uniformly dis-

tributed through the whole length of encoded source. However we require that the errors are not of burst type, (two errors must occur approximately at least 500 bits away from each other) which is a reasonable assumption even for the very noisy channels. The need for this constraint is made clear in the process of describing the algorithm.

We can view the second algorithm as an improved version of the first algorithm. Therefore after a brief explanation of the first algorithm, the second algorithm is described in great detail.

4.2.1 The System Structure

The system model is depicted in Figure 4.1. It consists of an encoder unit, a noisy channel and a decoder unit.

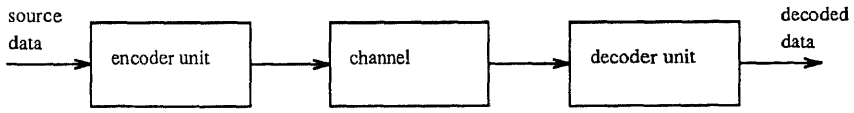


Figure 4.1: The System Model

4.2.2 Algorithm I

Source data is a text file whose symbols are withdrawn from two different sets. The first set contains the symbols that construct words, while the second set, contains the symbols that separate the words. A word is defined as a group of characters separated by one or more separators.

Encoder encodes the incoming data, symbol by symbol. It uses arithmetic encoding algorithm. It has a buffer for storing the current word which is being encoded, and a dictionary for storing the words which are used in the data source

for the first time. If an encoded word is a new word, after encoding the last character in the word, a special character is generated and encoded for informing the decoder that the received word is a new word.

Channel is a noisy communication channel. But we assume that the probability of introducing error to a new word is very low. As we indicated before, this assumption is reasonable only if the source file is big enough to construct the word dictionary at the decoder side. In addition, we assume that the errors are not burst type and two errors are far from each other, in such a way that before encountering the second error, we should be able to correct the first one.

Decoder decodes the received data. It uses arithmetic decoding algorithm. Mainly it has three code buffers, one string buffer and one incomplete word buffer. It also has a dictionary which is initially empty and built during the decoding process.

Error Detection

Initially all the buffers are empty. Received encoded symbols are stored in the "current code buffer". At the same time these symbols are decoded and the decoded data is stored in the "string buffer". Whenever the code-buffer is full, we check the content of the string buffer word by word. If the word ends with previously introduced special character, we decide that it is a new word and store it into the dictionary. On the other hand, if it is not a new word then it must be used before and should be in the dictionary. If it is not in the dictionary then we conclude that it is an erroneous word and one of the bits in the corresponding code buffer must be erroneous.

Error Correction

To correct the error we toggle each bit in the code buffer and decode the new buffer content each time. If the decoded data (i.e. the string in the string buffer) is not valid, we replace the original bit in the code buffer and toggle the next one. This process continue until we find the valid combination of the words in the string buffer.

4.2.3 Algorithm II

The source data is fed into encoder sequentially. There are no restrictions on the type of data source being encoded. It may be a text file or some other data file represented by a binary alphabet. A "word" is defined as a fixed number of data source symbols its length being "word size" many symbols. Further we define a "word marker" to be a special character which is used for punctuating the source data after every word size many symbols.

The internal structure of encoder unit is shown in Figure 4.2. It has a counter for counting the incoming characters. After every word has been fed into the encoder, a word marker is automatically generated as the next character to be encoded. Hence we can summarize the function of encoder unit, as marking the original source data by word markers appropriately and then encoding the marked data by arithmetic encoding procedure.

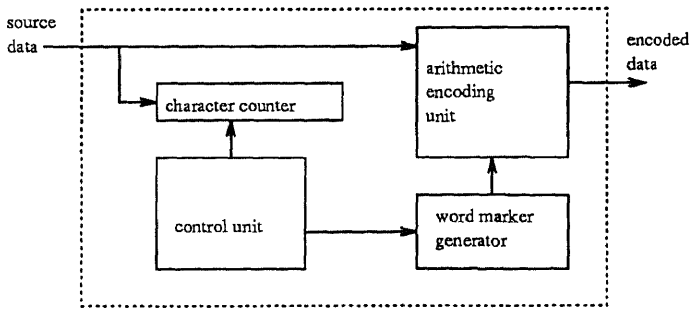


Figure 4.2: Encoder Unit

It may seem that by introducing a word marker after every word we are adding a large amount of redundancy into the source file. However, since the marked file is encoded, this redundancy is decreased significantly. Especially for large data files, the increase in the size of the encoded file due to the marking

process is insignificant.

Channel is assumed to be a noisy communication channel. The algorithm is tested for the channels which have different bit error rates. We assume that errors do not occur in a burst and that two errors occur at least "two current code buffer size" plus "one next code buffer size" away from each other. The reason for this is made clear in the discussion of the algorithm. If the current code buffer size and next code buffer size is not very large then this assumption is reasonable.

Decoder decodes the incoming encoded source by arithmetic decoding algorithm. The internal structure of the decoder is depicted in Figure 4.3.

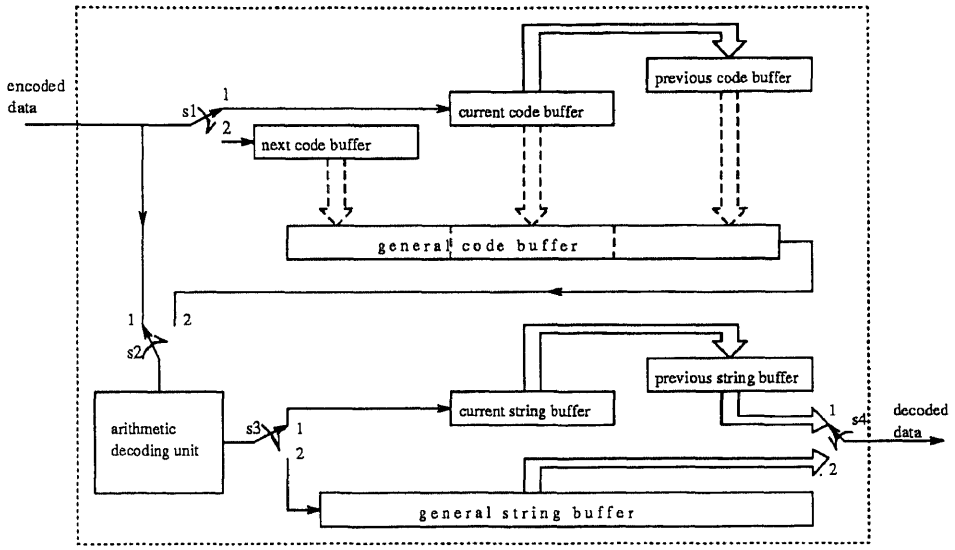


Figure 4.3: Decoder Unit

Other than the arithmetic decoding unit the decoder has four code buffers: Previous code buffer, current code buffer, next code buffer and general code buffer. The previous code buffer and current code buffer have a size of "code buffer size". The next code buffer has "next code buffer size" and finally the general code buffer has a size of previous, current and next code buffer sizes added together (i.e. 2

code buffer size and 1 next code buffer size).

Initially all the switches (i.e. s1, s2, s3 and s4) are in Position 1. Encoded data, enters the decoder sequentially. The data is decoded by the arithmetic decoding unit and the stored in the current string buffer. After word size many decoded characters we must have a word marker as the next decoded character in the current string buffer. If a word marker does not appear in this location in the decoded file, then it is clear that an error in the encoded data has caused the change. An erroneous bit in that part of the encoded source must have caused the error. The encoded source which is most recently decoded is in the current code buffer. However depending on the propagation speed the erroneous bit which cause the error may not be in the current code buffer (i.e. the length of the current code buffer may not be enough for the error to propagate). Therefore in order not to loose the erroneous bit in the buffer when the current code buffer is full, the content of this buffer are transferred to another code buffer which we call the previous code buffer. Similarly the content of the current string buffer is transferred to another string buffer, called the previous string buffer. At this point the current code buffer and the current string buffer are cleared for the next iteration.

Control unit (not shown in the figure) keeps checking for the presence of the word marker at specified locations. If the word marker is not in its location a flag, called error flag, is set and an error have been detected. For the error correction all the switches in the decoder (i.e. s1, s2, s3 and s4) are switched to the position 2. Then we fill the next code buffer with encoded data and transfer the contents of the previous, current and next code buffer into a code buffer which we call "general code buffer". Due to the assumptions about the error characteristics of the communication channel, it is assumed that there is only one erroneous bit in the general code buffer. But since we do not have any idea about which bit it

is, starting from the first bit in the general code buffer, each bit is reversed, one at a time, and for each change the content of the general code buffer is decoded and the decoded data is checked for the presence of the word markers at the expected positions. If the word markers are present where they are expected to be, the error is assumed to have been corrected. If we toggle the wrong bit, then including the erroneous bit we have 2 erroneous bits in the general code buffer. Because of very high error propagation in the encoded data which is encoded by arithmetic encoding procedure, it is unlikely that we would find the word markers in their correct positions. However if the next code buffer is not long enough then decoded source may not be enough for checking word marker positions. To avoid such a situation, the size of the next code buffer is chosen such that it should be long enough to give at least one word size many characters at the decoder output to be able to check at least the word marker position. Various code buffer sizes have been tried for various data files.

After observing the word markers in their correct positions in the decoded source which is located in the general string buffer, we decide that the general code buffer content must be the correct combination and hence the error is corrected.

There are two main routines in the algorithm: Encoding and Decoding routines. To simulate the channel characteristics a random number generator is used. These random numbers are used for introducing errors to the encoded source. As long as the constraint that we specified about the number of error free characters between two consecutive errors is satisfied and regardless of channel type every error is detected. Depending on the code buffers (current code buffer and next code buffer) sizes, almost every error is corrected successfully. In our simulation we assume that the introduced errors are distributed uniformly.

Encoding Routine:

The encoding routine is depicted in Figure 4.4.

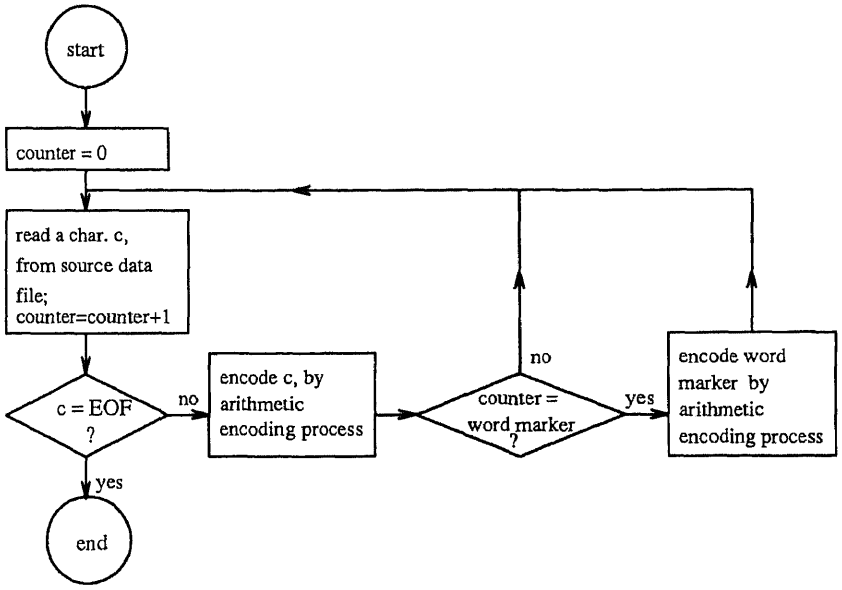


Figure 4.4: Encoding Routine

Channel Routine:

The channel routine is depicted in Figure 4.5. "Bound" is a variable number which varies depending on the bit error rate of the channel.

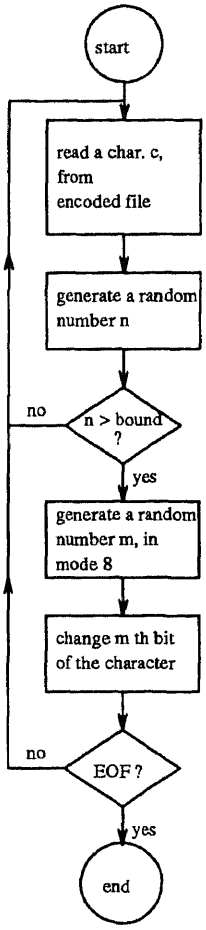


Figure 4.5: Channel Routine

Decoding Routine:

The decoding routine is depicted in Figure 4.6 below.

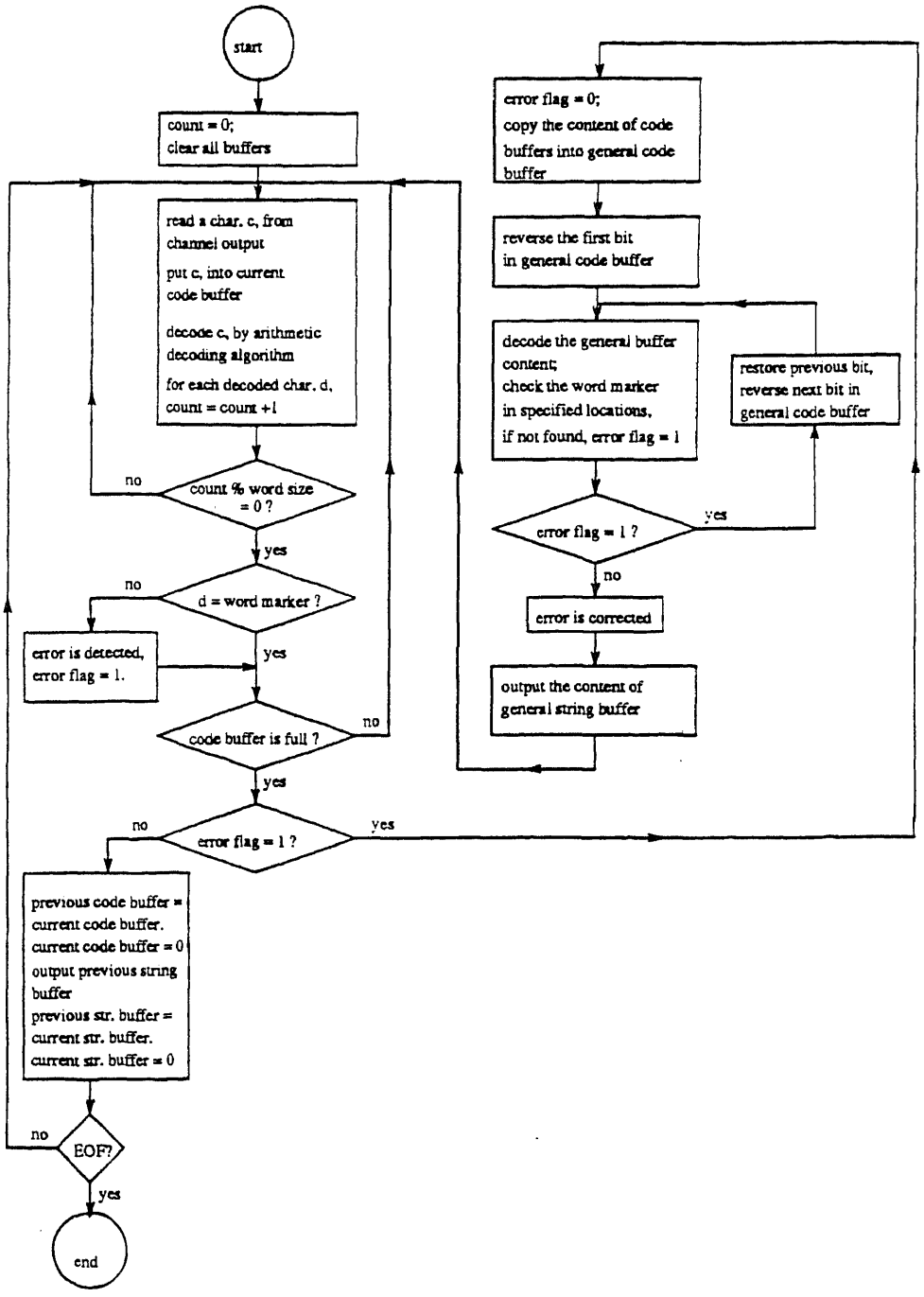


Figure 4.6: Decoding Routine

Chapter 5

Results and Conclusion

First, we observe the amount of expansion in the encoded file, as a result of the marking process. Various characters have been used as word markers, on various type of the data files[1]. Percentage increase in the size of the encoded files are depicted in the following figures for word size = 5 characters and word size = 10 characters. Different results are obtained are shown for different characters used as word markers.

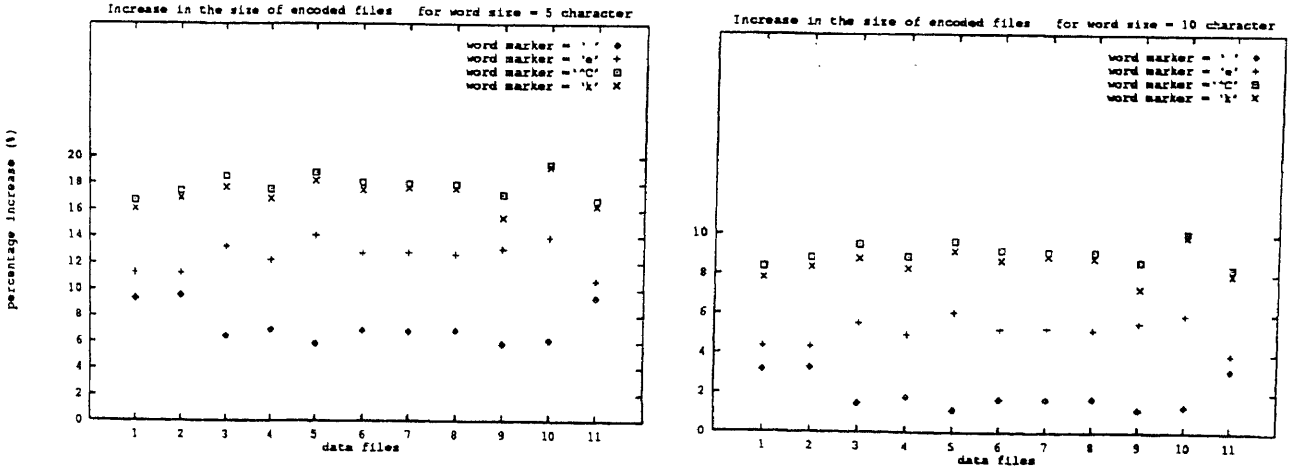


Figure 5.1: Percentage increase in the encoded file size

[1] *File 1* is an non technical article with a size of 8252 characters. *File 2* is technical book with 127537 characters. *File 3* through *File 8* are the chapters of a scientific book each one has approximately 15000 characters. *File 9* is a well documented C computer language program with 30627 characters. *File 10* is the manual for Unix mail facility with 29405 characters. *File 11* is a non technical report written in latex format with 4490 characters.

It is clear that inserting word markers into the source data, cause an expansion in the encoded data which is smaller than the expansion in the source data. The choice of word marker is the main factor for the amount of the expansion in the encoded file. If the word marker is a commonly used character (such as ' ' character) in the source file then the amount of expansion is not significant. However if the word marker is rarely used or not used in the source file then the expansion in the encoded data is more significant.

In the following tables, we observe the performance of the algorithm for various code buffer lengths and word markers. Since error detection and correction depends on the error propagation in the code buffers, the sizes of these buffers effect the performance of the algorithm significantly. To be able to correct an error after detecting it in the encoded data we need to check at least one word marker position after decoding the code string which starts from the erroneous bit position. Therefore especially the next code buffer length is an important factor in error correction procedure.

The choice of word marker is also an important factor in error correction. If it is a commonly used character in the source file then from the nature of arithmetic coding, its corresponding decoding interval in $[0, 1]$ interval is much larger than the decoding intervals corresponding to other data source characters. Hence the probability of decoding a word marker in its correct position without correcting the erroneous bit is relatively high. On the other hand if the word marker is a symbol, rarely used in the data source or not used at all, then such a false decoding is rare. That is, in this case decoding a word marker in its correct position is an indication with high probability that the error has been corrected.

These results are shown in the following tables:

word marker = ' ', word size = 5 char									
Data Files	current code buffer size								
	5 char			10 char			15 char		
	gen.	*det.*	*cor.*	*gen.*	*det.*	*cor.*	*gen.*	*det.*	*cor.*
next code buffer size = 5 char									
File 1	49	49	32	49	49	40	49	49	46
File 2	57	57	22	56	56	34	57	57	44
File 3	18	18	6	19	19	13	19	19	16
File 4	26	26	13	26	26	16	26	26	20
next code buffer size = 10 char									
File 1	49	49	33	50	50	48	49	49	47
File 2	57	57	42	58	58	49	57	57	55
File 3	18	18	13	18	18	14	18	18	15
File 4	27	27	24	27	27	26	26	26	26
next code buffer size = 15 char									
File 1	50	50	46	50	50	50	50	50	50
File 2	57	57	44	57	57	51	57	57	56
File 3	19	19	14	19	19	18	18	18	18
File 4	26	26	26	26	26	26	26	26	26

Table 5.1: Performance of Algorithm II, for word size = 5 characters and word marker = "space"

word marker = ' ', word size = 10 char									
Data Files	current code buffer size								
	10 char			20 char			30 char		
	gen.	*det.*	*cor.*	*gen.*	*det.*	*cor.*	*gen.*	*det.*	*cor.*
next code buffer size = 10 char									
File 1	47	47	25	47	47	40	48	48	43
File 2	54	54	27	55	55	31	55	55	52
File 3	17	17	9	18	18	12	18	18	13
File 4	25	25	19	25	25	22	25	25	22
next code buffer size = 20 char									
File 1	48	48	44	47	47	47			
File 2	55	55	43	54	54	53			
File 3	18	18	15	18	18	16			
File 4	25	25	25	25	25	23			
next code buffer size = 30 char									
File 1	47	47	43						
File 2	56	56	44						
File 3	18	18	12						
File 4	25	25	21						

Table 5.2: Performance of Algorithm II, for word size = 10 characters and word marker = "space"

word marker = '^ C', word size = 5 char									
Data Files	current code buffer size								
	5 char			10 char			15 char		
	gen.	*det.*	*cor.*	*gen.*	*det.*	*cor.*	*gen.*	*det.*	*cor.*
next code buffer size = 5 char									
File 1	52	52	44	53	53	52	53	53	51
File 2	64	64	57	64	64	61	64	64	63
File 3	20	20	15	21	21	20	21	21	21
File 4	29	29	23	28	28	28	28	28	28
next code buffer size = 10 char									
File 1	52	52	52	52	52	52	52	52	52
File 2	64	64	58	64	64	62	64	64	64
File 3	21	21	19	20	20	20	20	20	20
File 4	28	28	26	28	28	28	28	28	28
next code buffer size = 15 char									
File 1	52	52	50	53	53	53	52	52	52
File 2	64	64	64	63	63	63	64	64	64
File 3	20	20	16	21	21	21	21	21	21
File 4	28	28	28	28	28	28	28	28	28

Table 5.3: Performance of Algorithm II, for word size = 5 characters and word marker = "^ C"

word marker = '^C', word size = 10 char									
Data Files	current code buffer size								
	10 char			20 char			30 char		
	gen.	*det.*	*cor.*	*gen.*	*det.*	*cor.*	*gen.*	*det.*	*cor.*
next code buffer size = 10 char									
File 1	50	50	47	50	50	48	50	50	48
File 2	60	60	60	60	60	60	61	61	61
File 3	19	19	19	19	19	19	19	19	19
File 4	26	26	25	27	27	27	27	27	27
next code buffer size = 20 char									
File 1	49	49	49	50	50	50			
File 2	60	60	60	60	60	60			
File 3	19	19	17	20	20	20			
File 4	26	26	25	27	27	27			
next code buffer size = 30 char									
File 1	50	50	48						
File 2	59	59	59						
File 3	19	19	17						
File 4	26	26	26						

Table 5.4: Performance of Algorithm II, for word size = 10 characters and word marker = "^ C"

Bibliography

[1] C. E. Shannon, *A Mathematical Theory of Communication,* Bell System Technical Journal, Vol. 27, 1948, pp.379-423,623-656.

[2] R. M. Fano, *The Transmission of Information,* Technical Report number 65, MIT Research Lab of Electronics, 1949.

[3] D. A. Huffman, *A Method of Constructing Minimum Redundancy Codes,* Proceedings of the IRE, Vol. 40, Sept. 1952, pp. 1098-1101.

[4] L. D. Davisson, *Universal Noiseless Coding,* IEEE Transactions on Information Theory, IT-19, No. 6, November 1973, pp. 783-795.

[5] J. Ziv, *Coding Theorems for Individual Sequences,* IEEE Transactions on Information Theory, IT-24, No. 3, July 1978, pp. 405-412.

[6] A. Lempel and J. Ziv, *A Universal Algorithm for Sequential Data Compression,*, IEEE Transactions on Information Theory, Vol. IT-23, No. 3, May 1977, pp. 337-343.

[7] J. Ziv and A. Lempel, *Compression of Individual Sequences via Variable-Rate Coding,* IEEE Transactions on Information Theory, IT-24, No. 5, September 1978, pp. 530-536.

[8] T. A. Welch, *A Technique for High Performance Data Compression,* IEEE Computer Journal, June 1984, pp. 8-19.

[9] Jorma Rissanen, *Generalised Kraft Inequality and Arithmetic Coding,* IBM Journal of Research and Development - 20, 1976, pp. 198-203.

[10] Glen G. Langdon Jr., *An Introduction to Arithmetic Coding,* IBM Journal of Research and Development, Vol. 28, No. 2, March 1984.

[11] A. J. Viterbi and J. K. Omura, *Principles of Digital Communication and Coding,* McGraw-Hill Book Company, New York, 1979.

[12] P. Narasimhan, *Error Propagation and Error Correction in Universal Coding Systems,* M. Sc. Thesis, Department of Electrical Engineering, NJIT, 1990.

[13] J.A. Storer, *Data Compression : Methods and Theory,* Computer Science Press, 1988.

[14] T.J. Lynch, *Data Compression : Techniques and Application,* Lifetime Publications, Belmont, CA, 1985.

[15] L. G. Kraft, *A Device for Quantizing, Grouping and Coding Amplitute-modulated Pulses,* Electrical Engineering Thesis, MIT, Cambridge, Mass., 1949.

[16] N. Faller, *An Adaptive System for Data Compression,* In record of the 7th Asilomar Conference on Circuits, Systems and Computers, 1973, pp. 593-597.

[17] R. G. Gallager, *Variations on a Theme by Huffman,* IEEE Transactions on Information Theory, IT-24, 6, November 1978.

[18] D. E. Knuth, *Dynamic Huffman Coding,* J. Algorithms 6 (1985) pp. 163-180.

[19] N. Abramson, *Information Theory and Coding,* McGraw-Hill Book Co., Inc., New York, 1963.

[20] F. Jelenik, *Probabilistic Information Theory,* McGraw-Hill Book Co., Inc., New York, 1968.

[21] J. Rissanen, *A Universal Data Compression System,* IEEE Transactions on Information Theory, Vol. 29 No. 5, pp 656-664, September 1983

[22] R. Pasco, *Source Coding Algorithms for Fast Data Compression,* Ph.D. Thesis, Department of Electrical Engineering, Stanford University, California, 1976.

[23] C. B. Jones, *An Efficient Coding System for Long Source Sequences,* IEEE Transactions on Information Theory, Vol. IT-27, No. 3, May 1981, pp. 280-291.